WebSphere® Process Server

IBM

Version 6.0

Developing and deploying modules

Note

Before using this information, be sure to read the general information in “Notices” on page 95.

September 29 2005

This edition applies to version 6, release 0, of WebSphere Process Server (product number 5724-L01) and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, email doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Developing and deploying modules

Developing and deploying modules are fundamental tasks.

WebSphere Process Server documentation PDFs

The following topics describe the concepts and tasks involved in developing modules for use with WebSphere® Process Server and deploying modules to the server.

Overview of developing modules

A module is a basic deployment unit for a WebSphere Process Server application. A module contains one or more component libraries and staging modules used by the application. A component may reference other components. Developing modules involves ensuring that the components, staging modules, and libraries (collections of artifacts referenced by the module) required by the application are available on the production server.

WebSphere Integration Developer is the main tool for developing modules for deployment to WebSphere Process Server. Although you can develop modules in other environments, it is best to use WebSphere Integration Developer.

The following sections address how to implement and update modules on WebSphere Process Server.

A synopsis on components

A component is the basic building block to encapsulate reusable business logic. A component is associated with interfaces, references and implementations. The interface defines a contract between a component and a calling component. With WebSphere Process Server, a module can either export a component for use by other modules or import a component for use. To invoke a component, a calling module references the interface to the component. The references to the interfaces are resolved by configuring the references from the calling module to their respective interfaces.

To develop a module you must do the following activities:
- Define, modify, or manipulate business objects used by components
- Define or modify components through its interfaces.

  Note: A component is defined through its interface.
- Export or import components.
- Use the serviceDeploy command to create an EAR file to install a module that uses components.

Development types

WebSphere Process Server provides a component programming model to facilitate a service-oriented programming paradigm. To use this model, a provider exports interfaces of a component so that a consumer can import those interfaces and use the component as if it were local. A developer uses either strongly-typed interfaces or dynamically-typed interfaces to implement or invoke the component. The interfaces and their methods are described in the References section within this information center.

After installing modules to your servers, you can use the administrative console to change the target component for a reference from an application. The new target must accept the same business object type and perform the same operation that the reference from the application is requesting.

Component development considerations

When developing a component, ask yourself the following questions:

- Will this component be exported and used by another module?

  If so, provide an interface that can be used by another module.
- Will the component take a relatively long time to run?

  If so, consider implementing an asynchronous interface to the component.
- Is it beneficial to decentralize the component?

  If so, consider having a copy of the component in a module that is deployed on a cluster of servers to benefit from parallel processing.
- Does your application require a mixture of 1-phase and 2-phase commit resources?

  If so, make sure you enable last participant support for the application.

  Note: If you create your application using WebSphere Integration Developer or create the installable EAR file using the serviceDeploy command, these tools automatically enable the support for the application. See the topic, “Using one-phase and two-phase commit resources in the same transaction” in the WebSphere Application Server Network Deployment v6.0 information center.

Developing modules

A component must be contained within a module. Developing modules to contain components is key to providing services to other modules.

This task assumes that an analysis of requirements shows that implementing a component for use by other modules is beneficial.

After analyzing your requirements, you might decide that providing and using components is an efficient way to process information. If you determine that reusable components would benefit your environment, create a module to contain the components.

1. Identify components other modules can use.

   Once you have identified the components, continue with “Developing components” on page 3.
2. Identify components within an application that could use components in other modules.

   Once you have identified the components and their target components, continue with “Invoking components” on page 5.
3. Connect the client components with the target components through wires.
4. Configure the wires to the correct bus.

   The “Administering” section of this information center describes how to configure the wires to the correct bus.

Developing components

Develop components to provide reusable logic to multiple components within your server.

This task assumes that you have already developed and identified processing that is useful for multiple modules.

Multiple modules can use a component. Exporting a component makes it available to other modules that refer to the component through an interface. This task describes how to build the component so that other moduels can use it.

Note: A single component can contain multiple interfaces.

1. Define the data object to move data between the caller and the component.

   The data object and its type is part of the interface between the callers and the component.
2. Define an interface that the callers will use to reference the component.

   This interface definition names the component and lists any methods available within the component.
3. Develop the class that defines the implementation.
   - If the component is long running (or asynchronous), continue with step 4.
   - If the component is not long running (or synchronous), continue with step 5.
4. Develop an asynchronous implementation.
   a. Define the interface that represents the synchronous component.
   b. Define the interface that modules use to asynchronously invoke the component.

      Important: An asynchronous component interface cannot have a joinsTransaction property set to `true`.
   c. Define the implementation of the component.
   d. Define the interface that contacts the calling modules when the response is ready.
   e. Continue with step 6.
5. Develop a synchronous implementation.
   a. Define the interface that represents the synchronous component.
   b. Define the implementation of the component.
6. Save the component interfaces and implementations in files with a .java extension.
7. Package the module and necessary resources in a JAR file.

   See "Deploying a module to a production server" in this information center for a description of steps 7 through 9.
8. Run the serviceDeploy command to create an installable EAR file containing the application.
9. Install the application on the server node.
10. **Optional:** Configure the wires between the callers and the corresponding component, if calling a component in another module.

    The "Administering" section of this information center describes configuring the wires.

Examples of developing components

This example shows a synchronous component that implements a single method, CustomerInfo. The first section defines the interface to the component that implements a method called getCustomerInfo.

```
public interface CustomerInfo {
 public Customer getCustomerInfo(String customerID);
}
```

The following block of code implements the component.

```
public class CustomerInfoImpl implements CustomerInfo {
 public Customer getCustomerInfo(String customerID) {
  Customer cust = new Customer();

  cust.setCustNo(customerID);
  cust.setFirstName("Victor");
  cust.setLastName("Hugo");
  cust.setSymbol("IBM");
  cust.setNumShares(100);
  cust.setPostalCode(10589);
  cust.setErrorMsg("");

  return cust;
 }
}
```

This example develops an asynchronous component. The first section of code defines the interface to the component that implements a method called getQuote.

```
public interface StockQuote {

 public float getQuote(String symbol);
}
```

The following section is the implementation of the class associated with StockQuote.

```
public class StockQuoteImpl implements StockQuote {

 public float getQuote(String symbol) {


     return 100.0f;
 }
}
```

This next section of code implements the asynchronous interface, StockQuoteAsync.

```
public interface StockQuoteAsync {

 // deferred response
 public Ticket getQuoteAsync(String symbol);
 public float getQuoteResponse(Ticket ticket, long timeout);

 // callback
 public Ticket getQuoteAsync(String symbol, StockQuoteCallback callback);
}
```

This section is the interface, StockQuoteCallback, which defines the onGetQuoteResponse method.

```
public interface StockQuoteCallback {

 public void onGetQuoteResponse(Ticket ticket, float quote);
}
```

Invoke the service.

Invoking components

Modules can use components on any node of a WebSphere Process Server cluster.

Before invoking a component, make sure that the module containing the component is installed on a WebSphere Process Server.

Modules can use any component available within a WebSphere Process Server cluster by using the name of the component and passing the data type the component expects. Invoking a component in this environment involves a number of steps to locate and then create the reference to the required component.

Note: A module can also call a component within itself, known as an intra-module invocation. Implement eExternal calls (inter-module invocations) by exporting the interface in the providing component and importing the interface in the calling component.

1. Determine the components required by the calling module.

   Note the name of the interface within a component and the data type that interface requires.
2. Define a data object.

   Although the input or return can be a Java™ class, a service data object is optimal.
3. Locate the component.
   a. Use the ServiceManager class to obtain the references available to the calling module.
   b. Use the locateService() method to find the component.

      Depending on the component, the interface can either be a Web Service Descriptor Language (WSDL) port type or a Java interface.
4. Invoke the component either synchronously or asynchronously.

   You can either invoke the component through a Java interface or use the invoke() method to dynamically invoke the component.
5. Process the return.

   The component might generate an exception, so the client has to be able to process that possibility.

Example of invoking a component

The following example uses the ServiceManager class to obtain a list of components that the calling module can access directly.

```
ServiceManager serviceManager = new ServiceManager();
```

The following example uses the ServiceManager class to obtain a list of components from a file that contains the component references.

```
InputStream myReferences = new FileInputStream("MyReferences.references");
ServiceManager serviceManager = new ServiceManager(myReferences);
```

The following code locates a component that implements the StockQuote Java interface.

```
StockQuote stockQuote = (StockQuote)serviceManager.locateService("stockQuote");
```

The following code locates a component that implements either a Java or WSDL port type interface. The calling module uses the Service interface to interact with the component.

Tip: If the component implements a Java interface, the component can be invoked through either the interface or the invoke() method.

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
```

The following example shows a component, MyValue, that also calls other components.

```
public class MyValueImpl implements MyValue {

 public float myValue(String customerID) throws MyValueException {

  ServiceManager serviceManager = new ServiceManager();

     // variables
         Customer customer = null;
         float quote = 0;
         float value = 0;

     // invoke
         CustomerInfo cInfo =
      (CustomerInfo)serviceManager.locateService("customerInfo");
         customer = cInfo.getCustomerInfo(customerID);

     if (customer.getErrorMsg().equals("")) {

         // invoke
       StockQuoteAsync sQuote =
      (StockQuoteAsync)serviceManager.locateService("stockQuote");
       Ticket ticket =  sQuote.getQuoteAsync(customer.getSymbol());
    // ... do something else ...
       quote =  sQuote.getQuoteResponse(ticket, JService.WAIT);

         // assign
          value = quote * customer.getNumShares();
     } else {

         // throw
         throw new MyValueException(customer.getErrorMsg());
     }
     // reply
         return value;
 }
}
```

Configure the wires between the calling module references and the component interfaces.

Dynamically invoking a component:

When an module invokes a component that has a Web Service Descriptor Language (WSDL) port type interface, the module must invoke the component dynamically using the invoke() method.

This task assumes that a calling component is invoking a component dynamically.

With a WSDL port type interface, a calling component must use the invoke() method to invoke the component. A calling module can also invoke a component that has a Java interface this way.

1. Determine the module that contains the component required.
2. Determine the array required by the component.

   The input array can be one of three types:
   - Primitive uppercase Java types or arrays of this type
   - Ordinary Java classes or arrays of the classes
   - Service Data Objects (SDOs)
3. Define an array to contain the response from the component.

   The response array can be of the same types as the input array.
4. Use the invoke() method to invoke the required component and pass the array object to the component.
5. Process the result.

Example of dynamically invoking a component

In the following example, a module uses the invoke() method to call a component that uses primitive uppercase Java data types.

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
Float quote = (Float)stockQuote.invoke("getQuote", new Object[] {"IBM"})[0];
```

Developing applications for business processes and tasks

You can use a modeling tool, such as WebSphere Integration Developer to build and deploy business processes and tasks. These processes and tasks are interacted with at runtime, for example, a process is started, tasks are claimed and completed, and running processes are terminated. You can use Business Process Choreographer Explorer to interact with processes and tasks, or the Business Process Choreographer APIs to develop customized applications for these interactions.

The API provides generic methods that can be used with all processes and tasks that are deployed to Business Process Choreographer. The Business Process Choreographer API is provided as two stateless session enterprise beans:

- BusinessFlowManagerService interface provides the methods for business process applications
- HumanTaskManagerService interface provides the methods for task-based applications

For more information on the Business Process Choreographer APIs, see the Javadoc in the com.ibm.bpe.api package and the com.ibm.task.api package.

1. Decide on the functionality that the application is to provide.

   Examples for the following business process and human task functionality are provided:
   - “Developing applications for human tasks” on page 22
   - “Managing the life cycle of a business process” on page 18
   - “Sending a message to a waiting activity” on page 16
   - Administering applications, such as deleting process instances or managing work items
2. Decide which of the Business Choreographer APIs you are going to use.

Depending on the scenarios that you want to implement with your application, you can use one, or both, of the session beans.

3. Determine the authorization authorities needed by users of the application.

   When an instance of the appropriate Business Process Choreographer API session bean is created, WebSphere Application Server associates a session context with the instance. The session context contains the caller's principal role. This information is used to check the caller's authorization for each call. The caller must be authorized to call methods, and view objects and the attributes of these objects.
4. Decide how to render the application.

   The Business Process Choreographer APIs can be called locally or remotely.
5. Develop the application.
   a. Access the API.
   b. Use the API to interact with processes or tasks.
      - Query the data.
      - Work with the data.

Accessing the generic APIs

Business process applications and task applications access the appropriate session bean through the home interface of the bean.

The BusinessFlowManagerService interface and the HumanTaskManagerService interface are the common interfaces for the session beans. These interfaces expose the functions that can be called by an application program. The application program can be any Java program, including another Enterprise JavaBeans™ (EJB) application.

You can access the generic APIs in one of the following ways.

- Access the remote session bean.
- Access the local session bean.

Accessing the remote session bean

An application accesses the appropriate remote session bean through the home interface of the bean.

The session bean can be either the BusinessFlowManager session bean for process applications or the HumanTaskManager session bean for task applications.

1. Add a reference to the remote session bean to the application deployment descriptor. Add the reference to one of the following files:
   - The `application-client.xml` file, for a Java 2 Platform, Enterprise Edition (J2EE) client application
   - The `web.xml` file, for a Web application
   - The `ejb-jar.xml` file, for an Enterprise JavaBeans (EJB) application

   The reference to the remote home interface for process applications is shown in the following example:

```
<ejb-ref>
 <ejb-ref-name>ejb/BusinessFlowManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
 <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
```

   The reference to the remote home interface for task applications is shown in the following example:

```
<ejb-ref>
 <ejb-ref-name>ejb/HumanTaskManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.task.api.HumanTaskManagerHome</home>
 <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
```

If you use WebSphere Integration Developer to add the EJB reference to the deployment descriptor, the binding for the EJB reference is automatically created when the application is deployed. For more information on adding EJB references, refer to the WebSphere Integration Developer documentation.

2. Package the generated stubs with your application. If your application runs on a different Java Virtual Machine (JVM) from the one where the BPEContainer application or the TaskContainer application runs, complete the following actions:
   a. For process applications, package the files contained in the `WebSphere/AppServer/ProcessChoreographer/client/bpe137650.jar` file with the enterprise archive (EAR) file of your application.
   b. For task applications, package the files contained in the `WebSphere/AppServer/ProcessChoreographer/client/task137650.jar` file with the enterprise archive (EAR) file of your application.
   c. Set the **Class-Path** parameter in the manifest file of the application module to include the JAR file. The application module can be a J2EE application, a Web application, or an EJB application.
3. Make the home interface of the session bean available to the application using Java Naming and Directory Interface (JNDI) lookup mechanisms. The following example shows this step for a process application:

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the remote home interface of the BusinessFlowManager bean
  Object result =
         initialContext.lookup("java:comp/env/ejb/BusinessFlowManagerHome");

// Convert the lookup result to the proper type
   BusinessFlowManagerHome processHome =
           (BusinessFlowManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,BusinessFlowManagerHome.class);
```

   The home interface of the session bean contains a create method for EJB objects. The method returns the remote interface of the session bean.
4. Access the remote interface of the session bean. The following example shows this step for a process application:

```
BusinessFlowManager process = processHome.create();
```

5. Call the business functions exposed by the service interface. The following example shows this step for a process application:

```
process.initiate("MyProcessModel",input);
```

   Calls from applications are run as transactions. A transaction is established and ended in one of the following ways:
   - Automatically by WebSphere Application Server (the deployment descriptor specifies TX_REQUIRED).
   - Explicitly by the application. You can bundle application calls into one transaction:

```
// Obtain user transaction interface
   UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");
```

```
    // Begin a transaction
    transaction.begin();

      // Applications calls ...

    // On successful return, commit the transaction
    transaction.commit();
```

Here is an example of how steps 3 through 5 might look for a task application.

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the remote home interface of the HumanTaskManager bean
  Object result =
         initialContext.lookup("java:comp/env/ejb/HumanTaskManagerHome");

// Convert the lookup result to the proper type
   HumanTaskManagerHome taskHome =
           (HumanTaskManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,HumanTaskManagerHome.class);

...
//Access the remote interface of the session bean.
HumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

Accessing the local session bean

An application accesses the appropriate local session bean through the home interface of the bean.

The session bean can be either the LocalBusinessFlowManager session bean for process applications or the LocalHumanTaskManager session bean for human task applications.

1. Add a reference to the local session bean to the application deployment descriptor. Add the reference to one of the following files:
   - The `application-client.xml` file, for a Java 2 Platform, Enterprise Edition (J2EE) client application
   - The `web.xml` file, for a Web application
   - The `ejb-jar.xml` file, for an Enterprise JavaBeans (EJB) application

   The reference to the local home interface for process applications is shown in the following example:

   ```
   <ejb-local-ref>
    <ejb-ref-name>ejb/LocalBusinessFlowManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
    <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
   </ejb-local-ref>
   ```

   The reference to the local home interface for task applications is shown in the following example:

   ```
   <ejb-local-ref>
    <ejb-ref-name>ejb/LocalHumanTaskManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
    <local>com.ibm.task.api.LocalHumanTaskManager</local>
   </ejb-local-ref>
   ```

If you use WebSphere Integration Developer to add the EJB reference to the deployment descriptor, the binding for the EJB reference is automatically created when the application is deployed. For more information on adding EJB references, refer to the WebSphere Integration Developer documentation.

2. Make the local home interface of the local session bean available to the application, using Java Naming and Directory Interface (JNDI) lookup mechanisms. The following example shows this step for a process application:

   ```
   // Obtain the default initial JNDI context
   InitialContext initialContext = new InitialContext();

     // Lookup the local home interface of the LocalBusinessFlowManager bean

      LocalBusinessFlowManagerHome processHome =
           (LocalBusinessFlowManagerHome)initialContext.lookup
           ("java:comp/env/ejb/LocalBusinessFlowManagerHome");
   ```

   The home interface of the local session bean contains a create method for EJB objects. The method returns the local interface of the session bean.

3. Access the local interface of the local session bean. The following example shows this step for a process application:

   ```
   LocalBusinessFlowManager process = processHome.create();
   ```

4. Call the business functions exposed by the service interface. The following example shows this step for a process application:

   ```
   process.initiate("MyProcessModel",input);
   ```

   Calls from applications are run as transactions. A transaction is established and ended in one of the following ways:

   - Automatically by WebSphere Application Server (the deployment descriptor specifies TX_REQUIRED).
   - Explicitly by the application. You can bundle application calls into one transaction:

     ```
     // Obtain user transaction interface
        UserTransaction transaction=
            (UserTransaction)initialContext.lookup("jta/usertransaction");

        // Begin a transaction
        transaction.begin();

          // Applications calls ...

        // On successful return, commit the transaction
        transaction.commit();
     ```

Here is an example of how steps 2 through 4 might look for a task application.

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the local home interface of the LocalHumanTaskManager bean

   LocalHumanTaskManagerHome taskHome =
        (LocalHumanTaskManagerHome)initialContext.lookup
        ("java:comp/env/ejb/LocalHumanTaskManagerHome");

...
//Access the local interface of the local session bean
LocalHumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

Developing applications for business processes

A business process is a set of business-related activities that are invoked in a specific sequence to a achieve a business goal. A business process can be either a microflow or a long-running process:

- Microflows are short running business processes. A microflow is invoked with input parameters, and the caller waits while the process is executed synchronously. After a very short time, the result is returned to the caller.
- Long-running, interruptible processes are executed as a sequence of activities that are chained together. Parallel branches of the process can be navigated synchronously. Depending on the type and the transaction setting of the activity, an activity can be run in its own transaction.

Examples are provided that show how you might develop applications for the following typical actions on microflows and long-running processes.

- Start a process.
- Send a message to a waiting activity.
- Handle events.
- Analyze the results of a process.
- Manage the life cycle of a process.
- Delete process instances.

Starting business processes

The way in which a business process is started depends on whether the process is a microflow or a long-running process. The service that starts the process is also important to the way in which a process is started; the process can have either a unique starting service or several starting services.

Examples are provided that show how you might develop applications for typical starting scenarios.

- Run a microflow.

  Examples are provided for microflows that contain a and those that contain a .
- Start a long-running process.

  Examples are provided for long-running processes that contain a and those processes that contain a .

Running a microflow that contains a unique starting service:

A microflow can be started by a receive activity or a pick activity. If the microflow implements a request-response operation, that is, the process contains a reply, you can use the call method to run the process.

The starting service is unique if the microflow starts with a receive activity or when the pick activity has only one onMessage definition. You can start this type of process with the call method and pass the process template name as a parameter.

1. **Optional:** List the process templates to find the name of the process you want to run.

   This step is optional if you already know the name of the process.

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
 "PROCESS_TEMPLATE_NAME",
  new Integer(50),
  null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started by the call method.

2. Start the process with an input message of the appropriate type.

   When you create the message, you must specify its message type name so that the message definition is contained.

```
ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
                           (template.getID(),
                            template.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}

//run the process
ClientObjectWrapper output = process.call(template.getName(), input);
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

   This action creates an instance of the process template, CustomerTemplate, and passes some customer data. The operation returns only when the process is complete. The result of the process, OrderNo, is returned to the caller.

Running a microflow that contains a non-unique starting service:

A microflow can be started by a receive activity or a pick activity. If the microflow implements a request-response operation, that is, the process contains a reply, you can use the call method to execute the process.

If the starting service is not unique, that is, the process starts with a pick activity that has multiple onMessage definitions, then you must identify the service to be called.

1. **Optional:** List the process templates to find the name of the process you want to run.

   This step is optional if you already know the name of the process.

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
 "PROCESS_TEMPLATE_NAME",
  new Integer(50),
  null);
```

   The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started as long-running processes.

2. Determine the starting service to be called.

```
ProcessTemplateData template = processTemplates[0];
ActivityServiceTemplateData[] startActivities =
        process.getStartActivities(template.getID());
```

3. Start the process with an input message of the appropriate type.

   When you create the message, you must specify its message type name so that the message definition is contained.

```
ActivityServiceTemplateData activity = startActivities[0];
//create a message for the service to be called
ClientObjectWrapper input =
      process.createMessage(activity.getServiceTemplateID(),
                            activity.getActivityTemplateID(),
                            activity.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//run the process
ClientObjectWrapper output = process.call(activity.getServiceTemplateID(),
                                          activity.getActivityTemplateID(),
                                          input);
//check the output of the process, for example, an order number
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

   This action creates an instance of the process template, CustomerTemplate, and passes some customer data. The operation returns only when the process is complete. The result of the process, OrderNo, is returned to the caller.

Starting a long-running process that contains a unique starting service:

If the starting service is unique, you can use the initiate method and pass the process template name as a parameter. This is the case when the long-running process starts with either a single receive or pick activity and when the single pick activity has only one onMessage definition.

1. **Optional:** List the process templates to find the name of the process you want to start.

   This step is optional if you already know the name of the process.

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
  ("PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
   "PROCESS_TEMPLATE.NAME",
    new Integer(50),
    null);
```

   The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started by the initiate method.

2. Start the process with an input message of the appropriate type.

   When you create the message, you must specify its message type name so that the message definition is contained. If you specify a process-instance name, it must not start with an underscore. If a process-instance name is not specified, the process instance ID (PIID) in String format is used as the name.

```
ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
                            (template.getID(),
                            template.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
```

```
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//start the process
PIID piid = process.initiate(template.getName(), "CustomerOrder", input);
```

This action creates an instance, CustomerOrder, and passes some customer data. When the process starts, the operation returns the object ID of the new process instance to the caller.

The starter of the process instance is set to the caller of the request. This person receives a work item for the process instance. The process administrators, readers, and editors of the process instance are determined and receive work items for the process instance. The follow-on activity instances are determined. These are started automatically or, if they are staff, receive, or pick activities, work items are created for the potential owners.

Starting a long-running process that contains a non-unique starting service:

A long-running process can be started through multiple initiating receive or pick activities. You can use the initiate method to start the process. If the starting service is not unique, for example, the process starts with multiple receive activities or a pick activity that has multiple onMessage definitions, then you must identify the service to be called.

1. **Optional:** List the process templates to find the name of the process you want to start.

   This step is optional if you already know the name of the process.

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
     ("PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
      "PROCESS_TEMPLATE.NAME",
       new Integer(50),
       null);
   ```

   The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started as long-running processes.

2. Determine the starting service to be called.

   ```
   ProcessTemplateData template = processTemplates[0];
   ActivityServiceTemplateData[] startActivities =
           process.getStartActivities(template.getID());
   ```

3. Start the process with an input message of the appropriate type.

   When you create the message, you must specify its message type name so that the message definition is contained. If you specify a process-instance name, it must not start with an underscore. If a process-instance name is not specified, the process instance ID (PIID) in String format is used as the name.

   ```
   ActivityServiceTemplateData activity = startActivities[0];
   //create a message for the service to be called
   ClientObjectWrapper input = process.createMessage
                             (activity.getServiceTemplateID(),
                              activity.getActivityTemplateID(),
                              activity.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   //start the process
   ```

```
PIID piid = process.initiate(activity.getServiceTemplateID(),
                             activity.getActivityTemplateID(),
                             "CustomerOrder",
                              input);
```

This action creates an instance, CustomerOrder, and passes some customer data. When the process starts, the operation returns the object ID of the new process instance to the caller.

The starter of the process instance is set to the caller of the request and receives a work item for the process instance. The process administrators, readers, and editors of the process instance are determined and receive work items for the process instance. The follow-on activity instances are determined. These are started automatically or, if they are staff, receive, or pick activities, work items are created for the potential owners.

Sending a message to a waiting activity

Pick, receive, and onMessage activities can be used to synchronize a running process with events from the ″outside world″. For example, the receipt of an e-mail from a customer in response to a request for information might be such an event.

1. List the activity service templates that are waiting for a message from the logged-on user.

   ```
   QueryResultSet result =
      process.query("ACTIVITY_SERVICE.VTID,ACTIVITY.ATID",
                   "ACTIVITY.STATE=ACTIVITY.STATE.STATE_WAITING AND
                    WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                    null, null, null);
   ```

2. Send a message.

   The caller must be a potential owner of the activity that receives the message, or an administrator of the process instance.

   ```
   if ( result.size() > 0 )
   {
     result.first();
     VTID vtid = (VTID)result.getOID(1);
     ATID atid = (ATID)result.getOID(2);
     ActivityServiceTemplateData activity =
             process.getActivityServiceTemplate(vtid,atid);

     // create a message for the service to be called
     ClientObjectWrapper message =
           process.createMessage(vtid,atid,activity.getInputMessageTypeName());
     DataObject myMessage = null;
     if ( message.getObject()!= null && message.getObject() instanceof DataObject )
     {
       myMessage = (DataObject)message.getObject();
       //set the strings in the message, for example, chocolate is to be ordered
       myMessage.setString("Order", "chocolate");
     }

     // send the message to the waiting activity
     process.sendMessage(vtid, atid, message);
   }
   ```

   This action sends the specified message to the waiting activity service and passes some order data.

   You can also specify the process instance ID to ensure that the message is sent to the specified process instance. If the process instance ID is not specified, the message is sent to the activity service, and the process instance that is identified by the correlation values in the message. If the process instance ID is specified, the process instance that is found using the correlation values is checked to ensure that it has the specified process instance ID.

Handling events

An entire business process and each of its scopes can be associated with event handlers that are invoked if the associated event occurs. Event handlers are similar to receive or pick activities in that a process can provide Web service operations using event handlers. You can invoke an event handler any number of times. In addition, multiple instances of an event handler can be activated concurrently.

The following code snippet shows how to get the active event handlers for a given process instance and how to send an input message.

1. Determine the data of the process instance ID and list the active event handlers for the process.

   ```
   ProcessInstanceData processInstance =
          process.getProcessInstance( "CustomerOrder2711");
   EventHandlerTemplateData[] events = process.getActiveEventHandlers(
                                         processInstance.getID() );
   ```

2. Send the input message.

   ```
   EventHandlerTemplateData event = null;
   if ( events.length > 0 )
   {
       event = events[0];

       // create a message for the service to be called
       ClientObjectWrapper input = process.createMessage(
       event.getID(), event.getInputMessageTypeName());

       if (input.getObject() != null && input.getObject() instanceof DataObject )
       {
          DataObject inputMessage = (DataObject)input.getObject();
          // set content of the message, for example, a customer name, order number
          inputMessage.setString("CustomerName", "Smith");
          inputMessage.setString("OrderNo", "2711");

           // send the message
          process.sendMessage( event.getProcessTemplateName(),
                               event.getPortTypeNamespace(),
                               event.getPortTypeName(),
                               event.getOperationName(),
        input );
        }
    }
   ```

   This action sends the specified message to the active event handler for the process.

Analyzing the results of a process

A long-running process runs asynchronously. Its output message is not automatically returned when the process completes. The message must be retrieved explicitly. The results of the process are stored in the database only if the process template from which the process instance was derived does not specify automatic deletion of the derived process instances.

Analyze the results of the process, for example, check the order number.

```
QueryResultSet result = process.query
                      ("PROCESS_INSTANCE.PIID",
                       "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
                        PROCESS_INSTANCE.STATE =
                                PROCESS_INSTANCE.STATE.STATE_FINISHED",
                        null, null, null);
if (result.size() > 0)
```

```
{
  result.first();
  PIID piid = (PIID) result.getOID(1);
  ClientObjectWrapper output = process.getOutputMessage(piid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject )
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

Managing the life cycle of a business process

A process instance comes into existence when a Business Process Choreographer API method that can start a process is invoked. The navigation of the process instance continues until all of its activities are in an end state. Valid end states are finished, skipped, failed, expired, or terminated.

Sometimes, the process instance, or one of its activities, might encounter a fault that cannot be processed as part of the process logic. In these cases, a process administrator can act on the activity or the process instance in a number of ways.

Examples are provided that show how you might develop applications for the following typical life-cycle actions on processes.
- Force the completion of an activity.
- Retry the execution of a stopped activity.
- Suspend and resume a process instance.
- Restart a process instance.
- Terminate a process instance.

Forcing the completion of an activity:

If an activity in a long-running process encounters a fault and the fault is not caught in the enclosing scope and the associated activity template specifies that the activity stops when an error occurs, the activity is put into the stopped state so that it can be repaired. In this state, you can force the completion of the activity.

Additional requirements exist for certain types of activities.

Staff activities
: You can pass parameters in the force-complete call, such as the message that should have been sent or the fault that should have been raised.

Script activities
: You cannot pass parameters in the force-complete call. However, you must set the variables that need to be repaired.

Invoke activities
: You can also force the completion of invoke activities that call an asynchronous service that is not a subprocess if these activities are in the running state. You might want to do this, for example, if the asynchronous service is called and it does not respond.

1. List the stopped activities.

```
QueryResultSet result =
     process.query("DISTINCT ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    PROCESS_INSTANCE.NAME='CustomerOrder'",
                    null, null, null);
```

This action returns the stopped activities for the CustomerOrder process instance.

2. Complete the activity.

   In this example, an output message is passed:

```
if (result.size() > 0)
{
   result.first();
   AIID aiid = (AIID) result.getOID(1);
   ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper output =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( output.getObject()!= null && output.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)output.getObject();
     //set the parts in your message, for example, an order number
     myMessage.setInt("OrderNo", 4711);
   }

   boolean continueOnError = true;
   process.forceComplete(aiid, output, continueOnError);
}
```

   This action completes the activity. If an error occurs, the **continueOnError** parameter determines whether the activity stays in the stopped state. In the example, **continueOnError** is true. This value means that if an error occurs during processing of the forceComplete request, the activity is put into the failed state. The fault is propagated to the enclosing scopes of the activity until it is either handled or the process scope is reached. The process is then put into the failing state and it eventually reaches the failed state.

Retrying the execution of a stopped activity:

If an activity in a long-running process encounters an uncaught fault in the enclosing scope and if the associated activity template specifies that the activity stops when an error occurs, the activity is put into the stopped state so that it can be repaired. You can retry the execution of the activity.

You can set variables that are used by the activity. With the exception of script activities, you can also pass parameters in the force-retry call, such as the message that was expected by the activity.

1. List the stopped activities.

```
QueryResultSet result =
      process.query("DISTINCT ACTIVITY.AIID",
                    "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                     PROCESS_INSTANCE.NAME='CustomerOrder'",
                     null, null, null);
```

   This action returns the stopped activities for the CustomerOrder process instance.

2. Retry the execution of the activity.

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper input =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
    {
        myMessage = (DataObject)input.getObject();
        //set the strings in your message, for example, chocolate is to be ordered
        myMessage.setString("OrderNo", "chocolate");
      }

    boolean continueOnError = true;
    process.forceRetry(aiid, input, continueOnError);
}
```

This action retries the activity. If an error occurs, the **continueOnError** parameter determines whether the activity stays in the stopped state. In the example, **continueOnError** is true. This means that if an error occurs during processing of the forceRetry request, the activity is put into the failed state. The fault is propagated to the enclosing scopes of the activity until it is either handled or the process scope is reached. The process is then put into the failing state and it eventually reaches the failed state.

Suspending and resuming a business process:

You can suspend a process instance and resume it again to complete it.

The caller must be an administrator of the process instance or a business process administrator. To suspend a process instance, it must be in the running or failing state.

You can suspend a long-running, top-level process instance while it is running. You might want to do this, for example, so that you can configure access to a back-end system that is used later in the process. When the prerequisites for the process are met, you can resume the process instance.

1. Get the process that you want to suspend.
   ```
   ProcessInstanceData processInstance =
                       process.getProcessInstance("CustomerOrder");
   ```
2. Suspend the process instance.
   ```
   PIID piid = processInstance.getID();
   process.suspend( piid );
   ```
   This action suspends the specified top-level process instance and its subprocesses. The process instance is put into the suspended state. Subprocesses are suspended if they are in the running, failing, terminating, or compensating state.
3. Resume the process instance.
   ```
   process.resume( piid );
   ```
   This action puts the process instance and its subprocesses into the states they had before they were suspended.

Restarting a business process:

You can restart a process instance that is in the finished, terminated, failed, or compensated state.

The caller must be an administrator of the process instance or a business process administrator.

Restarting a process instance is similar to starting a process instance for the first time. However, when a process instance is restarted, the process instance ID is known and the input message for the instance is available.

1. Get the process that you want to restart.

```
ProcessInstanceData processInstance =
                    process.getProcessInstance("CustomerOrder");
```
2. Restart the process instance.
   ```
   PIID piid = processInstance.getID();
   process.restart( piid );
   ```
   This action restarts the specified process instance.

Terminating a process instance:

Sometimes, it is necessary for someone with process administrator authorization to terminate a process instance that is known to be in an unrecoverable state. For example, when an application is invoked and fails, and it does not return to a dormant state.

Because a process instance terminates immediately, without waiting for any outstanding subprocesses or activities, you should terminate a process instance only in exceptional situations.
1. Retrieve the process instance that is to be terminated.
   ```
   ProcessInstanceData processInstance =
          process.getProcessInstance("CustomerOrder");
   ```
2. Terminate the process instance.

   If you terminate a process instance, you can terminate the process instance with or without compensation.

   To terminate the process instance with compensation:
   ```
   PIID piid = processInstance.getID();
   process.forceTerminate(piid, CompensationBehaviour.INVOKE_COMPENSATION);
   ```
   To terminate the process instance without compensation:
   ```
   PIID piid = processInstance.getID();
   process.forceTerminate(piid);
   ```
   If you terminate the process instance with compensation, the compensation handler defined for the process template is called. If the process template does not have a compensation handler defined, the default compensation handler is called. If you terminated the process instance without compensation, the process instance is terminated immediately without waiting for activities to end normally.

Deleting process instances

Completed processes instances are automatically deleted from the Business Process Choreographer database if the corresponding property is set for the process template in the process model.

You might want to keep process instances in your database, for example, to query data from process instances that are not written to the audit log, or if you want to defer the deletion of processes to off-peak times. However, process instance data that is no longer needed can impact disk space and performance. Therefore, you should regularly delete process instance data.

The following example shows how to delete all of the finished process instances.
1. List the process instances that are finished.
   ```
   QueryResultSet result =
        process.query("DISTINCT PROCESS_INSTANCE.PIID",
                      "PROCESS_INSTANCE.STATE =
                              PROCESS_INSTANCE.STATE.STATE_FINISHED",
                       null, null, null);
   ```

This action returns a query result set that lists process instances that are finished.

2. Delete the process instances that are finished.

```
while (result.next() )
{
   PIID piid = (PIID) result.getOID(1);
   process.delete(piid);
}
```

This action deletes the selected process instance from the database.

Developing applications for human tasks

A task is the means by which components invoke humans as services or by which humans invoke services. Examples of the following typical applications for human tasks are provided.

- "Starting an originating task that implements a synchronous interface"
- "Starting an originating task that implements an asynchronous interface" on page 23
- "Processing participating or purely human tasks" on page 23
- "Analyzing the results of a task" on page 25
- "Terminating a task instance" on page 25
- "Deleting task instances" on page 25
- "Canceling a claimed task" on page 26
- "Managing work items" on page 26

For more information on the Business Process Choreographer API, see the Javadoc in the com.ibm.task.api package.

Starting an originating task that implements a synchronous interface

This scenario creates an instance of a task template and passes some customer data. The operation returns only when the task is complete. The result of the task, OrderNo, is returned to the caller.

1. **Optional:** List the task templates to find the name of the originating task you want to run.

   This step is optional if you already know the name of the task.

```
TaskTemplate[] taskTemplates = task.queryTaskTemplates
("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
 "TASK_TEMPL.NAME",
  new Integer(50),
  null);
```

   The results are sorted by name. The query returns an array containing the first 50 sorted originating templates.

2. Create an input message of the appropriate type.

```
TaskTemplate template = taskTemplates[0];

// create a message for the selected task
ClientObjectWrapper input = task.createInputMessage( template.getID());
DataObject myMessage = null ;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
```

3. Create the task and run it synchronously.

   For a task to run synchronously, it must be a two-way operation.

   ```
   ClientObjectWrapper output = task.createAndCallTask( template.getName(),
                                                        template.getNamespace(),
                                                        input);
   ```

4. Analyze the result of the task.

   ```
   DataObject myOutput = null;
   if ( output.getObject() != null && output.getObject() instanceof DataObject )
   {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
   }
   ```

Starting an originating task that implements an asynchronous interface

This scenario creates an instance of a task template and passes some customer data.

1. **Optional:** List the task templates to find the name of the originating task you want to run.

   This step is optional if you already know the name of the task.

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
     ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
      "TASK_TEMPL.NAME",
       new Integer(50),
       null);
   ```

   The results are sorted by name. The query returns an array containing the first 50 sorted originating templates.

2. Create an input message of the appropriate type.

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```

3. Create the task and run it asynchronously.

   ```
   task.createAndStartTask( template.getName(),
                            template.getNamespace(),
                            input,
                            null);
   ```

Processing participating or purely human tasks

Participating or purely human tasks are assigned to various people in your organization through work items. Participating tasks and their associated work items are created, for example, when a process navigates to a staff activity. One of the potential owners claims the task associated with the work item. This person is responsible for providing the relevant information and completing the task.

1. List the tasks belonging to a logged-on person that are ready to be worked on.

```
QueryResultSet result =
     task.query("TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_READY AND
                (TASK.KIND = TASK.KIND.KIND_PARTICIPATING OR
                 TASK.KIND = TASK.KIND.KIND_HUMAN)AND
                 WORK_ITEM.REASON =
                   WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                 null, null, null);
```

This action returns a query result set that contains the tasks that can be worked on by the logged-on person.

2. Claim the task to be worked on.

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper input = task.claim(tkiid);
  DataObject taskInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    taskInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

When the task is claimed, the input message of the task is returned.

3. When work on the task is finished, complete the task.

   The task can be completed either successfully or with a fault message. If the task is successful, an output message is passed. If the task is unsuccessful, a fault message is passed. You must create the appropriate messages for these actions.

   a. To complete the task successfully, create an output message.

```
ClientObjectWrapper output =
      task.createOutputMessage(tkiid);
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the task
task.complete(tkiid, output);
```

   This action sets an output message that contains the order number. The task is put into the finished state.

   b. To complete the task when a fault occurs, create a fault message.

```
//retrieve the faults modeled for the task
List faultNames = task.getFaultNames(tkiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      task.createMessage(tkiid, (String)faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
}

task.complete(tkiid, (String)faultNames.get(0), myFault);
```

This action sets a fault message that contains the error code. The task is put into the failed state.

Analyzing the results of a task

A participating or purely human task runs asynchronously. Its output message is not automatically returned when the task completes. The message must be retrieved explicitly. The results of the task are stored in the database only if the task template from which the task instance was derived does not specify automatic deletion of the derived task instances.

Analyze the results of the task, for example, check the order number.

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.NAME = 'CustomerOrder' AND
                                    TASK.STATE = TASK.STATE.STATE_FINISHED",
                                    null, null, null);
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper output = task.getOutputMessage(tkiid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject)
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

Terminating a task instance

Sometimes it is necessary for someone with administrator rights to terminate a task instance that is known to be in an unrecoverable state. For example, when an application is invoked and fails and does not return to a dormant state.

It is recommended that you terminate a task instance only in exceptional situations. The task instance is terminated immediately.

1. Retrieve the task instance to be terminated.

   ```
   Task taskInstance = task.getTask(tkiid);
   ```

2. Terminate the task instance.

   ```
   TKIID tkiid = taskInstance.getID();
   task.terminate(tkiid);
   ```

   The task instance is terminated immediately without waiting for any outstanding tasks.

Deleting task instances

Task instances are only automatically deleted when they complete if this is specified in the associated task template from which the instances are derived. The following example shows how to delete all of the task instances that are finished.

1. List the task instances that are finished.

   ```
   QueryResultSet result =
        task.query("DISTINCT TASK.TKIID",
                   "TASK.STATE = TASK.STATE.STATE_FINISHED",
                    null, null, null);
   ```

This action returns a query result set that lists task instances that are finished.

2. Delete the task instances that are finished.

   ```
   while (result.next() )
   {
    TKIID tkiid = (TKIID) result.getOID(1);
    task.delete(tkiid);
   }
   ```

Canceling a claimed task

Sometimes it is necessary for someone with administrator rights to cancel a task that is claimed by someone else. This situation might occur, for example, when a task must be completed but the owner of the task is absent.

1. List the claimed tasks owned by a specific person, for example, Smith.

   ```
   QueryResultSet result =
        task.query("DISTINCT TASK.TKIID",
                   "TASK.STATE = TASK.STATE.STATE_CLAIMED AND
                    TASK.OWNER = 'Smith'",
                    null, null, null);
   ```

   This action returns a query result set that lists the tasks claimed by the specified person, Smith.

2. Cancel the claimed task.

   ```
   if (result.size() > 0)
   {
     result.first();
     TKIID tkiid = (TKIID) result.getOID(1);
     task.cancelClaim(tkiid);
   }
   ```

   This action returns the task to the ready state so that it can be claimed by one of the other potential owners.

Managing work items

A work item represents the assignment of an object to a user or group of users for a particular reason. The object is typically a staff activity instance, a process instance, or a human task. The reasons are derived from the role that the user has for an activity or task. An activity or task can have multiple work items because a user can have different roles in association with the activity or task, and a work item is created for each of these roles.

During the lifetime of an activity instance or a task instance, the set of people associated with the object can change, for example, because a person is on vacation, new people are hired, or the workload needs to be distributed differently. To allow for these changes, you can develop applications to create, delete, or transfer work items.

- Create a work item.

  ```
  // query the task instance for which an additional
  // administrator is to be specified
  QueryResultSet result = task.query("TASK.TKIID",
                                    "TASK.NAME='CustomerOrder'",
                                     null, null, null);
  if ( result.size() > 0 )
  {
    result.first();
    // create the work item
    task.createWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_ADMINISTRATOR,"Smith");
  }
  ```

This action creates a work item for the user Smith who has the administrator role.

- Delete a work item.

```
// query the task instance for which a work item is to be deleted
QueryResultSet result = task.query("TASK.TKIID",
                                   "TASK.NAME='CustomerOrder'",
                                    null, null, null);
if ( result.size() > 0 )
{
  result.first();
  // delete the work item
  task.deleteWorkItem((TKIID)(result.getOID(1)),
                        WorkItem.REASON_READER,"Smith");
}
```

This action deletes the work item for the user Smith who has the reader role.

- Transfer a work item.

```
// query the task that is to be rescheduled
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.NAME='CustomerOrder' AND
                 TASK.STATE=TASK.STATE.STATE_READY AND
                 WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER AND
                 WORK_ITEM.OWNER_ID='Miller'",
                 null, null, null);
if ( result.size() > 0 )
{
  result.first();
  // transfer the work item from user Miller to user Smith
  // so that Smith can work on the task
  task.transferWorkItem((TKIID)(result.getOID(1)),
                         WorkItem.REASON_POTENTIAL_OWNER,"Miller","Smith");
}
```

This action transfers the work item to the user Smith so that he can work on it.

Querying business-process and task-related objects

You can query business-process and task-related objects in the database to retrieve specific properties of these objects.

During the configuration of Business Process Choreographer, a relational database is associated with both the business process container and the task container. The database stores all of the template (model) and instance (runtime) data for managing business processes and tasks. You use SQL-like syntax to query this data.

You can perform a one-off query to retrieve a specific property of an object. You can also save queries that you use often and include these stored queries in your application.

Queries on business-process and task-related objects

Use the query interface of the service API to retrieve stored information about business processes and tasks.

Predefined database views are provided for you to query the object properties. For process templates and task templates, the query function has the following syntax. The example shows the syntax for querying process templates.

```
ProcessTemplateData[] queryProcessTemplates
                       (java.lang.String whereClause,
                        java.lang.String orderByClause,
                        java.lang.Integer threshold,
                        java.util.TimeZone timezone);
```

For the other business-process and task-related objects, the query function has the following syntax:

```
QueryResultSet query (java.lang.String selectClause,
                      java.lang.String whereClause,
                      java.lang.String orderByClause,
                      java.lang.Integer skipTuples
                      java.lang.Integer threshold,
                      java.util.TimeZone timezone);
```

The query is made up of:
- Select clause
- Where clause
- Order-by clause
- Skip-tuples parameter
- Threshold parameter
- Time-zone parameter

For example, a list of work items IDs accessible to the caller of the function is retrieved by:

```
QueryResultSet result = process.query("WORK_ITEM.WIID",
                                       null, null, null, null, null);
```

The query function returns objects according to the caller's authorization. The query result set contains the properties of only those objects that the caller is authorized to see.

The query interface also contains a queryAll method. You can use this method to retrieve all of the relevant data about an object, for example, for monitoring purposes. The caller of the queryAll method must have Java 2 Platform, Enterprise Edition (J2EE) system administrator or system monitor rights. Authorization checking using the corresponding work item of the object is not applied.

For more information on the Business Process Choreographer APIs, see the Javadoc in the com.ibm.bpe.api package for process-related methods and in the com.ibm.task.api package for task-related methods.

Select clause:

The select clause in the query function identifies the object properties that are to be returned by a query.

The select clause describes the query result. It specifies a list of names that identify the object properties (columns of the result) to return. Its syntax is the same as an SQL select clause; use commas to separate parts of the clause. Each part of the clause must specify a property from one of the predefined views. The columns returned in the QueryResultSet object appear in the same order as the properties specified in the select clause.

The select clause does not support SQL aggregation functions, such as AVG(), SUM(), MIN(), or MAX().

To select properties of name-value pairs, such as custom properties, add a one-digit suffix to the view name.

Examples of select clauses

- ″WORK_ITEM.OBJECT_TYPE, WORK_ITEM.REASON″

  Gets the object types of the associated objects and the assignment reasons for the work items.
- ″DISTINCT WORK_ITEM.OBJECT_ID″

  Gets all of the IDs of objects, without duplicates, for which the caller has a work item.
- ″ACTIVITY.TEMPLATE_NAME, WORK_ITEM.REASON″

  Gets the names of the activities the caller has work items for and their assignment reasons.
- ″ACTIVITY.STATE, PROCESS_INSTANCE.STARTER″

  Gets the states of the activities and the starters of their associated process instances.
- ″DISTINCT TASK.TKIID, TASK.NAME″

  Gets all of the IDs and names of tasks, without duplicates, for which the caller has a work item.
- ″TASK_CPROP1.STRING_VALUE, TASK_CPROP2.STRING_VALUE″

  Gets the values of the custom properties that are specified further in the where clause.
- ″COUNT(DISTINCT TASK.TKIID)″

  Counts the number of work items for unique tasks that satisfy the where clause.

If an error occurs during the processing of the select clause, a QueryUnknownTable or a QueryUnknownColumn exception is thrown with the name of the property that is not recognized as a table or column name.

Where clause:

The where clause in the query function describes the filter criteria to apply to the query domain.

The syntax of a where clause is the same as an SQL where clause. You do not need to explicitly add an SQL from clause or join predicates to the where clause, these constructs are added automatically when the query runs. If you do not want to apply filter criteria, you must specify `null` for the where clause.

The where-clause syntax supports:

- Keywords: AND, OR, NOT
- Comparison operators: =, <=, <, <>, >,>=, LIKE
- Set operation: IN

  The LIKE operation supports the wildcard characters that are defined for the queried database.

The following rules also apply:

- Specify object ID constants as `ID('string-rep-of-oid')`.
- Specify binary constants as `BIN('UTF-8 string')`.

- Use symbolic constants instead of integer enumerations. For example, instead of specifying an activity state expression `ACTIVITY.STATE=2`, specify `ACTIVITY.STATE=ACTIVITY.STATE.STATE_READY`.
- Refer to properties of name-value pairs, such as custom properties, by adding a one-digit suffix to the view name. For example: `"TASK_CPROP1.NAME='prop1' AND "TASK_CPROP2.NAME='prop2'"`
- Specify time-stamp constants as `TS('yyyy-mm-ddThh:mm:ss')`. To refer to the current date, specify `CURRENT_DATE` as the timestamp.

  You must specify at least a date or a time value in the timestamp:
  - If you specify a date only, the time value is set to zero.
  - If you specify a time only, the date is set to the current date.
  - If you specify a date, the year must consist of four digits; the month and day values are optional. Missing month and day values are set to 01. For example, `TS('2003')` is the same as `TS('2003-01-01T00:00:00')`.
  - If you specify a time, these values are expressed in the 24-hour system. For example, if the current date is 1 January 2003, `TS('T16:04')` or `TS('16:04')` is the same as `TS('2003-01-01T16:04:00')`.

Examples of where clauses

- Comparing an object ID with an existing ID
  ```
  "WORK_ITEM.WIID = ID('_WI:800c00ed.df8d7e7c.feffff80.38')"
  ```
  This type of where clause is usually created dynamically with an existing object ID from a previous call. If this object ID is stored in a *wiid1* variable, the clause can be constructed as:
  ```
  "WORK_ITEM.WIID = ID('" + wiid1.toString() + "')"
  ```
- Using time stamps
  ```
  "ACTIVITY.STARTED >= TS('2002-06-1T16.00.00')"
  ```
- Using symbolic constants
  ```
  "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER"
  ```
- Using Boolean values true and false
  ```
  "ACTIVITY.BUSINESS_RELEVANCE = TRUE"
  ```
- Using custom properties
  ```
  "TASK_CPROP1.NAME = 'prop1' AND " TASK_CPROP1.STRING_VALUE = 'v1' OR
   TASK_CPROP2.NAME = 'prop2' AND " TASK_CPROP2.STRING_VALUE = 'v2'"
  ```

Order-by clause:

The order-by clause in the query function specifies the sort criteria for the query result set.

The order-by clause syntax is the same as an SQL order-by clause; use commas to separate each part of the clause. Each part of the clause must specify a property from one of the predefined views.

Sort criteria are applied to the server, that is, the locale of the server is used for sorting. If you identify more than one property, the query result set is ordered by the values of the first property, then by the values of the second property, and so on.

If you do not want to sort the query result set, you must specify `null` for the order-by clause.

Examples of order-by clauses

- ″PROCESS_TEMPLATE.NAME″

  Sorts the query result alphabetically by the process-template name.
- ″PROCESS_INSTANCE.CREATED, PROCESS_INSTANCE.NAME DESC″

  Sorts the query result by the creation date and, for a specific date, sorts the results alphabetically by the process-instance name in reverse order.
- ″ACTIVITY.OWNER, ACTIVITY_TEMPLATE.NAME, ACTIVITY.STATE″

  Sorts the query result by the activity owner, then the activity-template name, and then the state of the activity.

Skip-tuples parameter:

The skip-tuples parameter specifies the number of query-result-set tuples that are to be ignored and not to be returned to the caller in the query result set.

Use this parameter with the threshold parameter to implement paging in a client application.

If this parameter is set to `null` and the threshold parameter is not set, all of the qualifying tuples are returned.

Example of a skip-tuples parameter

- new Integer(5)

  Specifies that the first five qualifying tuples are not to be returned.

Threshold parameter:

The threshold parameter in the query function restricts the number of objects returned from the server to the client in the query result set.

The threshold parameter can be useful, for example, in a graphical user interface where only a small number of items should be displayed. If you set the threshold parameter accordingly, the database query is faster and less data needs to transfer from the server to the client.

If this parameter is set to `null` and the skip-tuples parameter is not set, all of the qualifying objects are returned.

Example of a threshold parameter

- new Integer(50)

  Specifies that 50 qualifying tuples are to be returned.

Timezone parameter:

The time-zone parameter in the query function defines the time zone for time-stamp constants in the query.

Time zones can differ between the client that starts the query and the server that processes the query. Use the time-zone parameter to specify the time zone of the time-stamp constants used in the where clause, for example, to specify local times. The dates returned in the query result set have the same time zone that is specified in the query.

If the parameter is set to `null`, the timestamp constants are assumed to be Coordinated Universal Time (UTC) times.

Examples of time-zone parameters

-
  ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 null,
                 null,
                 java.util.TimeZone.getDefault() );
  ```
  Returns object IDs for activities that started later than 17:40 local time on 1 January 2005.

-
  ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 null, null, null);
  ```
  Return object IDs for activities that started later than 17:40 UTC on 1 January 2005. This specification is, for example, 6 hours earlier in Eastern Standard Time.

Query results:

A query result set contains the results of a query.

The elements of the result set are objects that the caller is authorized to see. You can read elements in a relative fashion using the next method or in an absolute fashion using the first and last methods. Because the implicit cursor of a query result set is initially positioned before the first element, you must call either the first or next methods before reading an element. You can use the size method to determine the number of elements in the set.

An element of the query result set comprises the selected attributes of work items and their associated referenced objects, such as activity instances and process instances. The first attribute (column) of a QueryResultSet element specifies the value of the first attribute specified in the select clause of the query request. The second attribute (column) of a QueryResultSet element specifies the value of the second attribute specified in the select clause of the query request, and so on.

You can retrieve the values of the attributes by calling a method that is compatible with the attribute type and by specifying the appropriate column index. The numbering of the column indexes starts with 1.

Attribute type	Method
String	getString
ID	getOID
Timestamp	getTimestamp getString
Integer	getInteger getShort getLong getString getBoolean
Boolean	getBoolean getShort getInteger getLong getString
CHAR FOR BIT DATA	getBinary

Example:

The following query is run:

```
QueryResultSet resultSet = process.query("ACTIVITY.STARTED,
                                          ACTIVITY.TEMPLATE_NAME AS NAME,
                                          WORK_ITEM.WIID, WORK_ITEM.REASON",
                                          null, null, null, null);
```

The returned query result set has four columns:
- Column 1 is a time stamp
- Column 2 is a string
- Column 3 is an object ID
- Column 4 is an integer

You can use the following methods to retrieve the attribute values:

```
while (resultSet.next())
{
 java.util.Calendar activityStarted = resultSet.getTimestamp(1);
 String templateName = resultSet.getString(2);
 WIID wiid = (WIID) resultSet.getOID(3);
 Integer reason = resultSet.getInteger(4);
}
```

You can use the display names of the result set, for example, as headings for a printed table. These names are the column names of the view or the name defined by the AS clause in the query. You can use the following method to retrieve the display names in the example:

```
resultSet.getColumnDisplayName(1) returns "STARTED"
resultSet.getColumnDisplayName(2) returns "NAME"
resultSet.getColumnDisplayName(3) returns "WIID"
resultSet.getColumnDisplayName(4) returns "REASON"
```

Managing stored queries

A stored query is a query that is stored in the database and identified by a name. Although the query definitions are stored in the database, items contained in the stored query are assembled dynamically when they are queried. All stored queries are publicly accessible. You can have stored queries for business process objects, task objects, or a combination of these two object types.

1. Create a stored query.

   For example, the following code snippet creates a query for process instances and saves it with a specific name.

   ```
   process.createStoredQuery("CustomerOrdersStartingWithA",
               "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
               "PROCESS_INSTANCE.NAME LIKE 'A%'",
               "PROCESS_INSTANCE.NAME",
                null,null);
   ```

   This query returns a sorted list of all the process-instance names that begin with the letter A and their associated process instance IDs (PIID).
2. Run the query defined by the stored query.

   ```
   QueryResultSet result = process.query("CustomerOrdersStartingWithA",
                 new Integer(0));
   ```

   This action returns the objects that fulfill the criteria. In this case, all of the customer orders that begin with A.
3. **Optional:** List the available stored queries.

   For example, the following code snippet shows how to get a list of stored queries for process objects:

   ```
   String[] storedQuery = process.getStoredQueryNames();
   ```

4. **Optional:** Check the query defined by a specific stored query.

```
StoredQuery storedQuery = process.getStoredQuery("CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
```

5. Delete a stored query.

   The following code snippet shows how to delete the stored query that you created in step 1.

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

Predefined views for queries on business-process and human-task objects

Predefined database views are provided for business-process and human-task objects. Use these views when you query reference data for these objects. When you use these views, you do not need to explicitly add join predicates for view columns, these constructs are added automatically for you. You can use the generic query function of the service API (BusinessFlowManagerService or HumanTaskManagerService) to query this data. You can also use the corresponding method of the HumanTaskManagerDelegate API or your predefined queries provided by your implementations of the ExecutableQuery interface.

ACTIVITY view:

Use this predefined database view for queries on activities.

Table 1. Columns in the ACTIVITY view

Column name	Type	Comments
PIID	ID	The process instance ID.
AIID	ID	The activity instance ID.
PTID	ID	The process template ID.
ATID	ID	The activity template ID.
KIND	Integer	The kind of activity. Possible values are: KIND_INVOKE KIND_RECEIVE KIND_REPLY KIND_THROW KIND_RETHROW KIND_TERMINATE KIND_WAIT KIND_COMPENSATE KIND_SEQUENCE KIND_EMPTY KIND_SWITCH KIND_WHILE KIND_PICK KIND_FLOW KIND_SCOPE KIND_SCRIPT KIND_STAFF KIND_ASSIGN KIND_CUSTOM
COMPLETED	Timestamp	The time the activity is completed.
ACTIVATED	Timestamp	The time the activity is activated.

Table 1. Columns in the ACTIVITY view (continued)

Column name	Type	Comments
FIRST_ACTIVATED	Timestamp	The time at which the activity was activated for the first time.
STARTED	Timestamp	The time the activity is started.
STATE	Integer	The state of the activity. Possible values are: STATE_INACTIVE STATE_READY STATE_RUNNING STATE_PROCESSING_UNDO STATE_SKIPPED STATE_FINISHED STATE_FAILED STATE_TERMINATED STATE_CLAIMED STATE_TERMINATING STATE_FAILING STATE_WAITING STATE_EXPIRED STATE_STOPPED
OWNER	String	Principal ID of the owner.
DESCRIPTION	String	If the activity template description contains placeholders, this column contains the description of the activity instance with the placeholders resolved.
TEMPLATE_NAME	String	Name of the associated activity template.
TEMPLATE_DESCR	String	Description of the associated activity template.
BUSINESS_RELEVANCE	Boolean	Specifies whether the activity is business relevant. The attribute affects logging to the audit trail. Possible values are: **TRUE** The activity is business relevant and it is audited. **FALSE** The activity is not business relevant and it is not audited.

ACTIVITY_ATTRIBUTE view:

Use this predefined database view for queries on custom properties for activities.

Table 2. Columns in the ACTIVITY_ATTRIBUTE view

Column name	Type	Comments
AIID	ID	The ID of the activity instance that has a custom property.
NAME	String	The name of the custom property.
VALUE	String	The value of the custom property.

ACTIVITY_SERVICE view:

Use this predefined database view for queries on activity services.

Table 3. Columns in the ACTIVITY_SERVICE view

Column name	Type	Comments
EIID	ID	The ID of the event instance.
AIID	ID	The ID of the activity waiting for the event.
PIID	ID	The ID of the process instance that contains the event.
VTID	ID	The ID of the service template that describes the event.
PORT_TYPE	String	The name of the port type.
NAME_SPACE_URI	String	The URI of the namespace.
OPERATION	String	The operation name of the service.

APPLICATION_COMP view:

Use this predefined database view to query the application component ID and default settings for tasks.

Table 4. Columns in the APPLICATION_COMP view

Column name	Type	Comments
ACOID	String	The ID of the application component.
BUSINESS_ RELEVANCE	Boolean	The default task business-relevance policy of the component. This value can be overwritten by a definition in the task template or the task. The attribute affects logging to the audit trail. Possible values are: **TRUE** The task is business relevant and it is audited. **FALSE** The task is not business relevant and it is not audited.
NAME	String	Name of the application component.
SUPPORT_ AUTOCLAIM	Boolean	The default automatic-claim policy of the component. If this attribute is set to TRUE, the task can be automatically claimed if a single user is the potential owner. This value can be overwritten by a definition in the task template or task.
SUPPORT_CLAIM_ SUSP	Boolean	The default setting of the component that determines whether suspended tasks can be claimed. If this attribute is set to TRUE, suspended tasks can be claimed. This value can be overwritten by a definition in the task template or the task.
SUPPORT_ DELEGATION	Boolean	The default task delegation-support policy of the component. If this attribute is set to TRUE, the task can be delegated. This value can be overwritten by a definition in the task template or task.

ESCALATION view:

Use this predefined database view to query data for escalations.

Table 5. Columns in the ESCALATION view

Column name	Type	Comments
ESIID	String	The ID of the escalation instance.
ACTION	Integer	The action triggered by the escalation. Possible values are: **ACTION_CREATE_WORK_ITEM** Creates a work item for each escalation receiver. **ACTION_SEND_EMAIL** Sends an e-mail to each escalation receiver. **ACTION_CREATE_EVENT** Creates and publishes an event.
ACTIVATION_STATE	Integer	An escalation instance is created if the corresponding task reaches one of the following states: **ACTIVATION_STATE_READY** Specifies that the human or participating task is ready to be claimed. **ACTIVATION_STATE_RUNNING** Specifies that the originating task is started and running. **ACTIVATION_STATE_WAITING_ FOR_SUBTASK** Specifies that the task is waiting for the completion of subtasks. **ACTIVATION_STATE_CLAIMED** Specifies that the task is claimed.
ACTIVATION_TIME	Timestamp	The time when the escalation is activated.
AT_LEAST_ EXP_STATE	Integer	The state of the task that is expected by the escalation. If a timeout occurs, the task state is compared with the value of this attribute. Possible values are: **AT_LEAST_EXPECTED_STATE_CLAIMED** Specifies that the task is claimed. **AT_LEAST_EXPECTED_STATE_ENDED** Specifies that the task is in a final state (FINISHED, FAILED, TERMINATED or EXPIRED). **AT_LEAST_EXPECTED_STATE_ SUBTASKS_COMPLETED** Specifies that all of the subtasks of the task are complete.
ESTID	String	The ID of the corresponding escalation template.
FIRST_ESIID	String	The ID of the first escalation in the chain.

Table 5. Columns in the ESCALATION view (continued)

Column name	Type	Comments
INCREASE_PRIORITY	Integer	Indicates how the task priority will be increased. Possible values are: **INCREASE_PRIORITY_NO** The task priority is not increased. **INCREASE_PRIORITY_ONCE** The task priority is increased once by one. **INCREASE_PRIORITY_REPEATED** The task priority is increased by one each time the escalation repeats.
NAME	String	The name of the escalation.
STATE	Integer	The state of the escalation. Possible values are: STATE_INACTIVE STATE_WAITING STATE_ESCALATED STATE_SUPERFLUOUS
TKIID	String	The task instance ID to which the escalation belongs.

ESCALATION_CPROP view:

Use this predefined database view to query custom properties for escalations.

Table 6. Columns in the ESCALATION_CPROP view

Column name	Type	Comments
ESIID	String	The escalation ID.
NAME	String	The name of the property.
DATA_TYPE	String	The type of the class for non-string custom properties.
STRING_VALUE	String	The value for custom properties of type String.

ESCALATION_DESC view:

Use this predefined database view to query multilingual descriptive data for escalations.

Table 7. Columns in the ESCALATION_DESC view

Column name	Type	Comments
ESIID	String	The escalation ID.
LOCALE	String	The name of the locale associated with the description or display name.
DESCRIPTION	String	A description of the task template.
DISPLAY_NAME	String	The descriptive name of the escalation.

PROCESS_ATTRIBUTE view:

Use this predefined database view for queries on custom properties for processes.

Table 8. Columns in the PROCESS_ATTRIBUTE view

Column name	Type	Comments
PIID	ID	The ID of the process instance that has a custom property.
NAME	String	The name of the custom property.
VALUE	String	The value of the custom property.

PROCESS_INSTANCE view:

Use this predefined database view for queries on process instances.

Table 9. Columns in the PROCESS_INSTANCE view

Column name	Type	Comments
PTID	ID	The process template ID.
PIID	ID	The process instance ID.
NAME	String	The name of the process instance.
STATE	Integer	The state of the process instance. Possible values are: STATE_READY STATE_RUNNING STATE_FINISHED STATE_COMPENSATING STATE_INDOUBT STATE_FAILED STATE_TERMINATED STATE_COMPENSATED STATE_COMPENSATION_FAILED STATE_TERMINATING STATE_FAILING STATE_SUSPENDED
CREATED	Timestamp	The time the process instance is created.
STARTED	Timestamp	The time the process instance started.
COMPLETED	Timestamp	The time the process instance completed.
PARENT_NAME	String	The name of the parent process instance.
TOP_LEVEL_NAME	String	The name of the top-level process instance. If there is no top-level process instance, this is the name of the current process instance.
STARTER	String	The principal ID of the starter of the process instance.
DESCRIPTION	String	If the description of the process template contains placeholders, this column contains the description of the process instance with the placeholders resolved.
TEMPLATE_NAME	String	The name of the associated process template.

Table 9. Columns in the PROCESS_INSTANCE view (continued)

Column name	Type	Comments
TEMPLATE_DESCR	String	Description of the associated process template.

PROCESS_TEMPLATE view:

Use this predefined database view for queries on process templates.

Table 10. Columns in the PROCESS_TEMPLATE view

Column name	Type	Comments
PTID	ID	The process template ID.
NAME	String	The name of the process template.
VALID_FROM	Timestamp	The time from when the process template can be instantiated.
TARGET_NAMESPACE	String	The target namespace of the process template.
APPLICATION_NAME	String	The name of the enterprise application to which the process template belongs.
VERSION	String	User-defined version.
CREATED	Timestamp	The time the process template is created in the database.
STATE	Integer	Specifies whether the process template is available to create process instances. Possible values are: STATE_STARTED STATE_STOPPED
EXECUTION_MODE	Integer	Specifies how process instances that are derived from this process template can be run. Possible values are: EXECUTION_MODE_MICROFLOW EXECUTION_MODE_LONG_RUNNING
DESCRIPTION	String	Description of the process template.
COMP_SPHERE	Integer	Specifies the compensation behavior of instances of microflows in the process template; either an existing compensation sphere is joined or a compensation sphere is created. Possible values are: COMP_SPHERE_REQUIRED COMP_SPHERE_REQUIRES_NEW COMP_SPHERE_SUPPORTS COMP_SPHERE_NOT_SUPPORTED

TASK view:

Use this predefined database view for queries on task objects.

Table 11. Columns in the TASK view

Column name	Type	Comments
TKIID	ID	The ID of the task instance.
ACTIVATED	Timestamp	The time when the task was activated.
APPLIC_ DEFAULTS_ID	ID	The ID of the application component that specifies the defaults for the task.
APPLIC_NAME	String	The name of the enterprise application to which the task belongs.
BUSINESS_ RELEVANCE	Boolean	Specifies whether the task is business relevant. The attribute affects logging to the audit trail. Possible values are: **TRUE** The task is business relevant and it is audited. **FALSE** The task is not business relevant and it is not audited.
COMPLETED	Timestamp	The time when the task completed.
CONTAINMENT_ CTX_ID	ID	The containment context for this task. This attribute determines the life cycle of the task. When the containment context of a task is deleted, the task is also deleted.
CTX_ AUTHORIZATION	Integer	Allows the task owner to access the task context. Possible values are: **AUTH_NONE** No authorization rights for the associated context object. **AUTH_READER** Operations on the associated context object require reader authority, for example, reading the properties of a process instance.
DUE	Timestamp	The time when the task is due.
EXPIRES	Timestamp	The date when the task expires.
FIRST_ACTIVATED	Timestamp	The time when the task was activated for the first time.
IS_ESCALATED	Boolean	Indicates whether an escalation of this task has occurred.
IS_INLINE	Boolean	Indicates whether the task is an inline participating task in a business process.

Table 11. Columns in the TASK view (continued)

Column name	Type	Comments
KIND	Integer	The kind of task. Possible values are: **KIND_HUMAN** States that the task is created and processed by a human. **KIND_WPC_STAFF_ACTIVITY** States that the task is a human task that is part of a business process. **KIND_ORIGINATING** States that the task supports person-to-computer interactions, which enables people to create, initiate, and start services. **KIND_PARTICIPATING** States that the task supportd computer-to-person interactions, which enable a person to implement a service. **KIND_ADMINISTRATIVE** States that the task is an administrative task.
LAST_MODIFIED	Timestamp	The time when the task was last modified.
LAST_STATE_ CHANGE	Timestamp	The time when the state of the task was last modified.
NAME	String	The name of the task.
NAME_SPACE	String	The namespace that is used to categorize the task.
ORIGINATOR	String	The principal ID of the task originator.
OWNER	String	The principal ID of the task owner.
PARENT_ CONTEXT_ID	String	The parent context for this task. This attribute provides a key to the corresponding context in the calling application component. The parent context is set by the application component that creates the task.
PRIORITY	Integer	The priority of the task.
STARTED	Timestamp	The time when the task was started (STATE_RUNNING, STATE_CLAIMED).
STARTER	String	The principal ID of the task starter.

Table 11. Columns in the TASK view (continued)

Column name	Type	Comments
STATE	Integer	The state of the task. Possible values are: **STATE_READY** States that the task is ready to be claimed. **STATE_RUNNING** States that the task is started and running. **STATE_FINISHED** States that the task finished successfully. **STATE_FAILED** States that the task did not finish successfully. **STATE_TERMINATED** States that the task has been terminated because of an external or internal request. **STATE_CLAIMED** States that the task is claimed. **STATE_EXPIRED** States that the task ended because it exceeded its specified duration. **STATE_FORWARDED** States that task completed with a follow-on task.
SUPPORT_ AUTOCLAIM	Boolean	Indicates whether this task is claimed automatically if it is assigned to a single user.
SUPPORT_CLAIM_ SUSP	Boolean	Indicates whether this task can be claimed if it is suspended.
SUPPORT_ DELEGATION	Boolean	Indicates whether this task supports work delegation with follow-on tasks.
SUSPENDED	Boolean	Indicates whether the task is suspended.
TKTID	String	The task template ID.
TOP_TKIID	String	The top parent task instance ID of the subtask.
TYPE	String	The type used to categorize the task.

TASK_CPROP view:

Use this predefined database view to query custom properties for task objects.

Table 12. Columns in the TASK_CPROP view

Column name	Type	Comments
TKIID	String	The task instance ID.
NAME	String	The name of the property.
DATA_TYPE	String	The type of the class for non-string custom properties.
STRING_VALUE	String	The value for custom properties of type String.

TASK_DESC view:

Use this predefined database view to query multilingual descriptive data for task objects.

Table 13. Column in the TASK_DESC view

Column name	Type	Comments
TKIID	String	The task instance ID.
LOCALE	String	The name of the locale associated with the description or display name.
DESCRIPTION	String	A description of the task.
DISPLAY_NAME	String	The descriptive name of the task.

TASK_TEMPL view:

This predefined database view holds data that you can use to instantiate tasks.

Table 14. Columns in the TASK_TEMPL view

Column name	Type	Comments
TKTID	String	The task template ID.
VALID_FROM	Timestamp	The time when the task template becomes available for instantiation.
APPLIC_ DEFAULTS_ID	String	The ID of the application component that specifies the defaults for the task template.
APPLIC_NAME	String	The name of the enterprise application to which the task template belongs.
BUSINESS_ RELEVANCE	Boolean	Specifies whether the task template is business relevant. The attribute affects logging to the audit trail. Possible values are: **TRUE** The task is business relevant and it is audited. **FALSE** The task is not business relevant and it is not audited.
CONTAINMENT_ CTX_ID	ID	The containment context for this task template. This attribute determines the life cycle of the task template. When a containment context is deleted, the task template is also deleted.
CTX_ AUTHORIZATION	Integer	Allows the task owner to access the task context. Possible values are: **AUTH_NONE** No authorization rights for the associated context object. **AUTH_READER** Operations on the associated context object require reader authority, for example, reading the properties of a process instance.
IS_INLINE	Boolean	Indicates whether this task template describes a staff activity in a business process.

Table 14. Columns in the TASK_TEMPL view (continued)

Column name	Type	Comments
KIND	Integer	The kind of tasks that are derived from this task template. Possible values are: **KIND_HUMAN** Specifies that the task is created and processed by a human. **KIND_ORIGINATING** Specifies that a human can assign a task to a computer. In this case, a human invokes an automated service. **KIND_PARTICIPATING** Specifies that a service component (such as a business process) assigns a task to a human. **KIND_ADMINISTRATIVE** Specifies that the task is an administrative task.
NAME	String	The name of the task template.
NAMESPACE	String	The namespace that is used to categorize the task template.
PRIORITY	Integer	The priority of the task template.
STATE	Integer	The state of the task template. Possible values are: **STATE_STARTED** Specifies that the task template is available for creating task instances. **STATE_STOPPED** Specifies that the task template is stopped. Task instances cannot be created from the task template in this state.
SUPPORT_ AUTOCLAIM	Boolean	Indicates whether tasks derived from this task template can be claimed automatically if they are assigned to a single user.
SUPPORT_CLAIM_ SUSP	Boolean	Indicates whether tasks derived from this task template can be claimed if they are suspended.
SUPPORT_ DELEGATION	Boolean	Indicates whether tasks derived from this task template support work delegation with follow-on tasks.
TYPE	String	The type used to categorize the task template.

TASK_TEMPL_CPROP view:

Use this predefined database view to query custom properties for task templates.

Table 15. Columns in the TASK_TEMPL_CPROP view

Column name	Type	Comments
TKTID	String	The task template ID.
NAME	String	The name of the property.
DATA_TYPE	String	The type of the class for non-string custom properties.

Table 15. Columns in the TASK_TEMPL_CPROP view (continued)

Column name	Type	Comments
STRING_VALUE	String	The value for custom properties of type String.

TASK_TEMPL_DESC view:

Use this predefined database view to query multilingual descriptive data for task template objects.

Table 16. Columns in the TASK_TEMPL_DESC view

Column name	Type	Comments
TKTID	String	The task template ID.
LOCALE	String	The name of the locale associated with the description or display name.
DESCRIPTION	String	A description of the task template.
DISPLAY_NAME	String	The descriptive name of the task template.

WORK_ITEM view:

Use this predefined database view for queries on work items and authorization data for process, tasks, and escalations.

Table 17. Columns in the WORK_ITEM view

Column name	Type	Comments
WIID	ID	The work item ID.
OWNER_ID	String	The principal ID of the owner.
GROUP_NAME	String	The name of the associated group worklist.
EVERYBODY	Boolean	Specifies whether everybody owns this work item.

Table 17. Columns in the WORK_ITEM view (continued)

Column name	Type	Comments
OBJECT_TYPE	Integer	The type of the associated object. Possible values are: **OBJECT_TYPE_ACTIVITY** Specifies that the work item was created for an activity. **OBJECT_TYPE_PROCESS_INSTANCE** Specifies that the work item was created for a process instance. **OBJECT_TYPE_TASK_INSTANCE** Specifies that the work item was created for a task. **OBJECT_TYPE_TASK_TEMPLATE** Specifies that the work item was created for a task template. **OBJECT_TYPE_ESCALATION_ INSTANCE** Specifies that the work item was created for an escalation instance. **OBJECT_TYPE_APPLICATION_ COMPONENT** Specifies that the work item was created for an application component.
OBJECT_ID	ID	The ID of the associated object, for example, the associated process or task.
ASSOC_OBJECT_TYPE	Integer	The type of the object referenced by the ASSOC_OID attribute, for example, task, process, or external objects. Use the values for the OBJECT_TYPE attribute.
ASSOC_OID	ID	The ID of the object associated object with the work item. For example, the process instance ID (PIID) of the process instance containing the activity instance for which this work item was created.
REASON	Integer	The reason for the assignment of the work item. Possible values are: REASON_POTENTIAL_STARTER REASON_POTENTIAL_INSTANCE_ CREATOR REASON_POTENTIAL_OWNER REASON_EDITOR REASON_READER REASON_ORIGINATOR REASON_OWNER REASON_STARTER REASON_ESCALATION_RECEIVER REASON_ADMINISTRATOR
CREATION_TIME	Timestamp	The date and time when the work item was created.

Handling exceptions and faults

Faults can occur when a process instance is created or when operations that are invoked as part of the navigation of a process instance fail. Mechanisms exist to handle these faults and they include:

- Passing control to the corresponding fault handlers
- Stopping the process and let someone repair the situation (force-retry, force-complete)
- Compensating the process
- Passing the fault to the client application as an API exception, for example, an exception is thrown when the process model from which an instance is to be created does not exist

The handling of faults and exceptions is described in the following tasks:

- "Handling API exceptions"
- "Checking which fault is set for an activity"
- "Checking which fault occurred for a stopped invoke activity" on page 49

Handling API exceptions

If a method in the BusinessFlowManagerService interface or the HumanTaskManagerService interface does not complete successfully, an exception is thrown that denotes the cause of the error. You can handle this exception specifically to provide guidance to the caller.

However, it is common practice to handle only a subset of the exceptions specifically and to provide general guidance for the other potential exceptions. All specific exceptions inherit from a generic ProcessException or TaskException. It is a *best practice* to catch generic exceptions with a final `catch(ProcessException)` or `catch(TaskException)` statement. This statement helps to ensure the upward compatibility of your application program because it takes account of all of the other exceptions that can occur.

Checking which fault is set for an activity

1. List the task activities that are in a failed or stopped state.

   ```
   QueryResultSet result =
        process.query("ACTIVITY.AIID",
                      "(ACTIVITY.STATE = ACTIVITY.STATE.STATE_FAILED OR
                        ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED) AND
                        ACTIVITY.KIND=ACTIVITY.KIND.KIND_STAFF",
                        null, null, null);
   ```

   This action returns a query result set that contains failed or stopped activities.

2. Read the name of the fault.

   This fault name is the local part of the fault queue name.

   ```
   if (result.size() > 0)
   {
     result.first();
     AIID aiid = (AIID) result.getOID(1);
     ClientObjectWrapper faultMessage = process.getFaultMessage(aiid);
     DataObject fault = null ;
     if ( faultMessage.getObject() != null && faultMessage.getObject()
          instanceof DataObject )
     {
       fault = (DataObject)faultMessage.getObject();
       Type type = fault.getType();
   ```

```
    String name = type.getName();
    String uri = type.getURI();
  }
}
```

This returns the fault name. You can also analyze the unhandled exception for a stopped activity instead of retrieving the fault name.

Checking which fault occurred for a stopped invoke activity

If an activity causes a fault to occur, the fault type determines the actions that you can take to repair the activity.

1. List the staff activities that are in a stopped state.

   ```
   QueryResultSet result =
        process.query("ACTIVITY.AIID",
                      "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                       ACTIVITY.KIND=ACTIVITY.KIND.KIND_INVOKE",
                       null, null, null);
   ```

   This action returns a query result set that contains stopped invoke activities.
2. Read the name of the fault.

   This is the local part of the fault queue name.

   ```
   if (result.size() > 0)
   {
     result.first();
     AIID aiid = (AIID) result.getOID(1);
     ActivityInstanceData activity = process.getActivityInstance(aiid);

     ProcessException excp = activity.getUnhandledException();
     if ( excp instanceof ApplicationFaultException )
     {
      ApplicationFaultException fault = (ApplicationFaultException)excp;
      String faultName = fault.getFaultName();
     }
   }
   ```

Authorization for business-process applications

Ensure that you enable global security in WebSphere Application Server.

When an instance of the LocalBusinessFlowManager or the BusinessFlowManager session bean is created, WebSphere Application Server associates a session context with the instance. The session context contains the caller's principal role. This information is used to check the caller's authorization for each call. The caller must be authorized to call methods, and view objects and the attributes of these objects.

The following reasons for a work-item assignment are used:
- For processes: reader, starter, administrator
- For activities: reader, editor, potential owner, owner, administrator, potential starter

These assignment reasons are mapped to authorizations:
- Activity reader: can see properties of the associated activity instance, and its input and output messages.
- Activity editor: has the authority of the activity reader, and has write access to messages and other data associated with the activity.
- Potential activity owner: has the authority of the activity editor, and has the right to claim the activity.

- Activity owner: has the authority of the potential activity owner, and has the right to complete the activity. Has the authority to transfer owned work items to an administrator or potential owner.
- Activity administrator: can repair activities that are stopped due to unexpected errors, and force terminate long-running activities.
- Activity potential starter: can send messages to receive or pick activities.
- Process starter: can see properties of the associated process instance, and its input and output messages.
- Process reader: can see properties of the associated process instance, its input and output messages, and everything that the activity reader supports for all of the contained activities but not those of the subprocesses.
- Process administrator: has the authority of the process reader and the process starter, and the right to intervene in a process that has started. Has the authority to create, delete, and transfer work items.

Special authorization authority is granted to people with the following roles:

- Business process administrator and the Java 2 Platform, Enterprise Edition (J2EE) BPESystemAdministrator. These roles have all privileges.
- Business process monitor and the J2EE BPESystemMonitor. These roles can read all of the objects.

Do not delete the user ID of the process starter from your user registry if the process instance still exists. If you do, the navigation of this process cannot continue. You receive the following exception in the system log file:

```
no unique ID for: <user ID>
```

Required authorizations for actions on business processes

Access to the LocalBusinessFlowManager or the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on a process; the caller must also be authorized to perform the action. The following minimum authorization authorities are needed for actions on business processes.

Table 18. Required authorizations for actions on business processes

Action	Required authorization
createMessage	process reader
createWorkItem	process administrator
getAllActivities	process reader
getActivityInstance	process reader
getWaitingActivities	process reader
getAllWorkItems	process reader
geClientUISettings	process reader
getCustomProperty	process reader
getCustomProperties	process reader
getCustomPropertyNames	process reader
getActiveEventHandler	process reader
getFaultMessage	process reader
getInputClientUISettings	process reader
getInputMessage	process reader
getOutputClientUISettings	process reader

Table 18. Required authorizations for actions on business processes (continued)

Action	Required authorization
getOutputMessage	process reader
getProcessInstance	process reader
getVariable	process reader
getWorkItems	process reader
setCustomProperty	process starter
setVariable	process administrator
delete	process administrator
deleteWorkItem	process administrator
transferWorkItem	process administrator
forceTerminate	process administrator
suspend	process administrator
resume	process administrator
restart	process administrator

Required authorizations for actions on business-process activities

Access to the LocalBusinessFlowManager or the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on an activity; the caller must also be authorized to perform the action. The following minimum authorization authorities are needed for actions on business-process activities.

Table 19. Required authorizations for actions on activities in business processes

Action	Required authorization
createMessage	activity reader or process reader
createWorkItem	process administrator or activity administrator
getActivityInstance	activity reader or process reader
getCustomProperty	activity reader or process reader
getCustomProperties	activity reader or process reader
getCustomPropertyNames	activity reader or process reader
getVariable	activity reader or process reader
getFaultMessage	activity reader or process reader
getFaultNames	activity reader or process reader
getInputMessage	activity reader or process reader
getOutputMessage	activity reader or process reader
getClientUISettings	activity reader or process reader
getWorkItems	activity reader or process administrator
getAllWorkItems	process reader or process administrator
setCustomProperty	activity editor or process administrator
setOutputMessage	activity editor or process administrator
setFaultMessage	activity editor or process administrator
setVariable	activity editor or process administrator

Table 19. Required authorizations for actions on activities in business processes (continued)

Action	Required authorization
claim	potential activity owner or process administrator
cancelClaim	activity owner or process administrator
complete	activity owner or process administrator
forceRetry	process administrator or activity administrator
forceComplete	process administrator or activity administrator
forceTerminate	process administrator or activity administrator
deleteWorkItem	process administrator or activity administrator
transferWorkItem	activity owner or process administrator
sendMessage	potential activity owner or process administrator

Authorization for human-task applications

Ensure that you enable global security in WebSphere Application Server.

When an instance of the LocalHumanTaskManager or the HumanTaskManager session bean is created, WebSphere Application Server associates a session context with the instance. The session context contains the caller's principal role. This information is used to check the caller's authorization for each call.

The following reasons for a work-item assignment are used:
- Potential owner is the person or group of people to whom the human or participating task is assigned.
- Owner is the potential owner that claimed the task.
- Editor is the person or group of people that can modify the data that belongs to the human or participating task although they are not owners or administrators of the task.
- Reader is the person or group of people that can read the task, task template, or escalation data although they are not owners, editors, or administrators of the task.
- Originator is the person who created the task.
- Potential starter is the person or group of people that can start an existing originating task. If a potential starter is not specified, the originator becomes the potential starter. For inline tasks without a potential starter, the default is everybody.
- Starter is the person who started an originating task.
- Administrator is the person or group of people that can administer the task, task template, or escalation.
- Escalation receiver is the person or group of people that receive an escalation if the escalation is triggered.
- E-mail receiver is the person or group of people that receive an e-mail if the escalation is triggered.
- Potential instance creator is the person or group of people that can create an instance of a task template.

Special authority is granted to people with the following roles:
- Administrator and the Java 2 Platform, Enterprise Edition (J2EE) TaskSystemAdministrator. These roles have all privileges.
- Reader and the J2EE TaskSystemMonitor. These roles can read all of the objects.

Required roles for actions on tasks

Access to the LocalHumanTaskManager or the HumanTaskManager interface does not guarantee that the caller can perform all of the actions on a task; the caller must also be authorized to perform the action. The following table shows the actions that a specific role can take.

Action	Caller's principal role								
	Owner	Pot owner	Starter	Pot starter	Origin	Admin	Editor	Reader	Esc receiver
callTask				X[1]	X[1]	X[1]			
cancelClaim	X					X			
claim		X				X			
complete	X					X			
createFaultMessage	X	X	X	X	X[1]	X	X	X	X
createInputMessage	X	X	X	X	X[1]	X	X	X	X
createOutputMessage	X	X	X	X	X[1]	X	X	X	X
createWorkItem					X[1, 2]	X			
delete					X[3]	X			
deleteWorkItem					X[1, 2]	X			
getCustomProperty	X	X	X	X	X[1]	X	X	X	X
getDocumentation	X	X	X	X	X[1]	X	X	X	X
getFaultMessage	X	X	X	X	X[1]	X	X	X	X
getFaultNames	X	X	X	X	X[1]	X	X	X	X
getInputMessage	X	X	X	X	X[1]	X	X	X	X
getOutputMessage	X	X	X	X	X[1]	X	X	X	X
getRoleInfo	X	X	X	X	X[1]	X	X	X	X
getTask	X	X	X	X	X[1]	X	X	X	X
getUISettings	X	X	X	X	X[1]	X	X	X	X
resume	X				X[1]	X			
setCustomProperty	X		X			X	X		
setFaultMessage	X					X	X		
setOutputMessage	X					X	X		
startTask				X	X[1]	X			
suspend	X				X[1]	X			
terminate	X		X[1]			X			
transferWorkItem	X		X		X[1]	X			
update	X		X			X	X		

Action	Caller's principal role								
	Owner	Pot owner	Starter	Pot starter	Origin	Admin	Editor	Reader	Esc receiver

Notes:

1. For stand-alone tasks and task templates only.
2. For tasks in the inactive state only.
3. The originator can delete tasks that are in the inactive state only.

Abbreviations:

Admin	Administrator
Esc receiver	Escalation receiver
Origin	Originator
Pot owner	Potential owner
Pot starter	Potential starter

BusinessFlowManagerService interface

The BusinessFlowManagerService interface exposes business-process functions that can be called by a client application.

The methods that can be called by the BusinessFlowManagerService interface depend on the state of the process or the activity and the authorization of the person that uses the application containing the method. The main methods for manipulating business process objects are listed here. For more information about these methods and the other methods that are available in the BusinessFlowManagerService interface, see the Javadoc in the com.ibm.bpe.api package.

Process templates

A process template is a versioned, deployed, and installed process model that contains the specification of a business process. It can be instantiated and started by issuing appropriate requests, for example, initiate(). The execution of the process instance is driven automatically by the server.

Table 20. API methods for process templates

Method	Description
getProcessTemplate	Retrieves the specified process template.
queryProcessTemplate	Retrieves process templates that are stored in the database.

Process instances

The following API methods start process instances.

Table 21. API methods for starting process instances

Method	Description
call	Creates and runs a process instance.

Table 21. API methods for starting process instances (continued)

Method	Description
callWithReplyContext	Creates and runs a process instance from the specified process template and waits asynchronously for the result.
callWithUISettings	Creates and runs a process instance and returns the output message and the client user interface (UI) settings.
initiate	Creates a process instance and initiates processing of the process instance.
sendMessage	Sends the specified message to the specified activity service and process instance.
getStartActivities (for processes with a non-unique starting service)	Returns information about the activities that can start a process instance from the specified process template.
getActivityServiceTemplate	Retrieves the specified activity service template.

Table 22. API methods for controlling the life cycle of process instances

Method	Description
suspend	Suspends the execution of a long-running, top-level process instance that is in the running or failing state.
resume	Resumes the execution of a long-running, top-level process instance that is in the suspended state.
restart	Restarts a long-running, top-level process instance that is in the finished, failed, compensated, or terminated state.
forceComplete	Forces the completion of an activity instance that is in the running or stopped state.
forceRetry	Forces the repetition of an activity instance that is in the running or stopped state.
forceTerminate	Terminates the specified top-level process instance, its subprocesses, and its running, claimed, or waiting activities.
delete	Deletes the specified top-level process instance and its subprocesses.
query	Retrieves the properties from the database that match the search criteria.

Activities

For invoke activities, you can specify in the process model that these activities continue in error situations. If the continue-on-error flag is set to false and an unhandled error occurs, the activity is put into the stopped state. A process administrator can then repair the activity. The continue-on-error flag and the associated repair functions can, for example, be used in a long-running process where an invoke activity fails occasionally, but the effort required to model compensation and fault handling is too high. The following methods are available for repairing activities.

Table 23. API methods for controlling the life cycle of activity instances

Method	**Description**
claim	Claims a ready activity instance for a user to work on the activity.
cancelClaim	Cancels the claim of the activity instance.
complete	Completes the activity instance.
forceComplete	Forces the completion of an activity instance that is in the running or stopped state.
forceRetry	Forces the repetition of an activity instance that is in the running or stopped state.
query	Retrieves the properties from the database that match the search criteria.

Variables and custom properties

The interface provides a get and a set method to retrieve and set values for variables. You can also associate named properties with, and retrieve named properties from, process and activity instances. Custom property names and values must be of the java.lang.String type.

Table 24. API methods for variables and custom properties

Method	**Description**
getVariable	Retrieves the specified variable.
setVariable	Sets the specified variable.
getCustomProperty	Retrieves the named custom property of the specified activity or process instance.
getCustomProperties	Retrieves the named custom properties of the specified activity or process instance.
getCustomPropertyNames	Retrieves the names of the custom properties for the specified activity or process instance.
setCustomProperty	Stores custom-specific values for the specified activity or process instance.

HumanTaskManagerService interface

The HumanTaskManagerService interface exposes task-related functions that can be called by a local or a remote client.

The methods that can be called depend on the state of the task and the authorization of the person that uses the application containing the method. The main methods for manipulating task objects are listed here. For more information about these methods and the other methods that are available in the HumanTaskManagerService interface, see the Javadoc in the com.ibm.task.api package.

Task templates

The following methods are available to work with task templates.

Table 25. API methods for task templates

Method	Description
getTaskTemplate	Retrieves the specified task template.
createAndCallTask	Creates and runs a task instance from the specified task template and waits asynchronously for the result.
createAndStartTask	Creates and starts a task instance from the specified task template.
createTask	Creates a task instance from the specified task template.
createInputMessage	Creates an input message for the specified task template using a string representation of the task template ID. For example, create a message that can be used to start a task.
queryTaskTemplates	Retrieves task templates that are stored in the database.

Task instances

The following methods are available to work with task instances.

Table 26. API methods for task instances

Method	Description
getTask	Retrieves a task instance; the task instance can be in any state.
startTask	Starts a task that has already been created.
resume	Resumes the human or participating task.
suspend	Suspends the human or participating task.
terminate	Terminates the specified task instance. If an originating task is terminated, this action has no impact on the invoked service.
delete	Deletes the specified task instance.

Escalations

The following methods are available to work with escalations.

Table 27. API methods for working with escalations

Method	Description
getEscalation	Retrieves the specified escalation instance.

Variables and custom properties

The interface provides a get and a set method to retrieve and set values for variables. You can also associate named properties with, and retrieve named properties from, process and activity instances. Custom property names and values must be of the java.lang.String type.

Table 28. API methods for variables and custom properties

Method	Description
getCustomProperty	Retrieves the named custom property of the specified task instance.
getCustomProperties	Retrieves the named custom properties of the specified task instance.
getCustomPropertyNames	Retrieves the names of the custom properties for the task instance.
setCustomProperty	Stores custom-specific values for the specified task instance.

Allowed actions for tasks types

The actions that can be carried out on a task depend on whether the task is a participating task, a purely human task, an originating task, or an administrative task.

You cannot use all of the actions provided by the LocalHumanTaskManager or the HumanTaskManager interface for all types of tasks. The following table shows the actions that you can carry out on each type of task type.

	Task type			
Action	**Participating task**	**Human task**	**Originating task**	**Administrative task**
callTask			X[1]	
cancelClaim	X	X[1]		
claim	X	X[1]		
complete	X	X[1]		X
createFaultMessage	X	X	X	X
createInputMessage	X	X	X	X
createOutputMessage	X	X	X	X
createWorkItem	X	X[1]	X	X
delete	X[1]	X[1]	X	X[1]
deleteWorkItem	X	X[1]	X	X
getCustomProperty	X	X[1]	X	X
getDocumentation	X	X[1]	X	X
getFaultMessage	X	X[1]	X	
getInputMessage	X	X[1]	X	
getOutputMessage	X	X[1]	X	
getRoleInfo	X	X[1]	X	X
getTask	X	X[1]	X	X
getUISettings	X	X[1]	X	X
resume	X	X[1]		
setCustomProperty	X	X[1]	X	X
setFaultMessage	X	X[1]		
setOutputMessage	X	X[1]		

	Task type			
Action	**Participating task**	**Human task**	**Originating task**	**Administrative task**
startTask	X[1]	X[1]	X	X
suspend	X	X[1]		
terminate	X[1]	X[1]	X[1]	
transferWorkItem	X	X[1]	X	X
updateTask	X	X[1]	X	X
Notes: 1. For stand-alone tasks and task templates only.				

Overview of preparing and installing modules

Installing modules (also known as deploying) activates the modules in either a test environment or a production environment. This overview briefly describes the test and production environments and some of the steps involved in installing modules.

Note: The process for installing applications in a production environment is similar to the process described in "Developing and deploying applications" in the WebSphere Application Server Network Deployment, v6.0 information center. If you are unfamiliar with those topics, review those first.

Before installing a module to a production environment, always verify changes in a test environment. To install modules to a test environment, use WebSphere Integration Developer (see the WebSphere Integration Developer information center for more information). To install modules to a production environment, use WebSphere Process Server.

This topic describes the concepts and tasks needed to prepare and install modules to a production environment. Other topics describe the files that house the objects that your module uses and help you move your module from your test environment into your production environment. It is important to understand these files and what they contain so you can be sure that you have correctly installed your modules.

Libraries and JAR files overview

Modules often use artifacts that are located in libraries. Artifacts and libraries are contained in Java archive (JAR) files that you identify when you deploy a module.

While developing a module, you might identify certain resources or components that could be used by various pieces of the module. These resources or components could be objects that you created while developing the module or already existing objects that reside in a library that is already deployed on the server. This topic describes the libraries and files that you will need when you install an application.

What is a library?

A library contains objects or resources used by multiple modules within WebSphere Integration Developer. The artifacts can be in JAR, resource archive (RAR), or Web service archive (WAR) files. Some of these artifacts include:
- Interfaces or Web services descriptors (files with a .wsdl extension)

- Business object XML schema definitions (files with an .xsd extension)
- Business object maps (files with a .map extension)
- Relationship and role definitions (files with a .rel and .rol extension)

When a module needs an artifact, the server locates the artifact from the EAR class path and loads the artifact, if it is not already loaded, into memory. From that point on, any request for the artifact uses that copy until it is replaced. Figure 1 shows how an application contains components and related libraries.

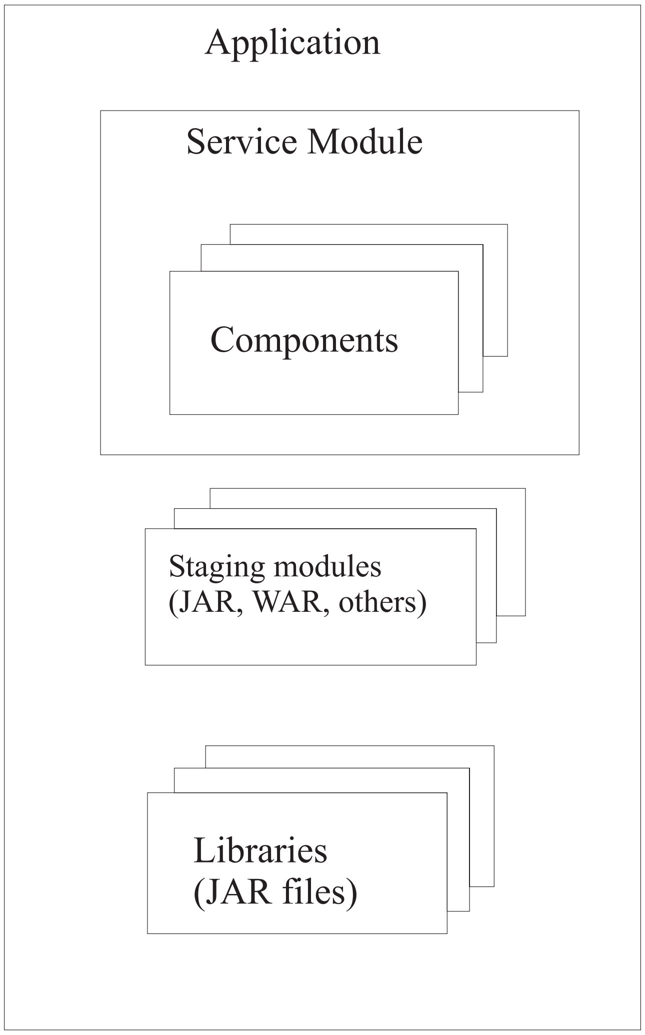

Figure 1. Relationship amongst module, component and libary

What are JAR, RAR, and WAR files?

There are a number of files that can contain components of a module. These files are fully described in the Java 2 Enterprise Edition (J2EE) specification. Details about JAR files can be found in the JAR specification.

In WebSphere Process Server, a JAR file also contains an application, which is the assembled version of the module with all the supporting references and interfaces to any other components used by the module. To completely install the application, you need this JAR file, any other libraries such as JAR files, Web services archive (WAR) files, resource archive (RAR) files, staging libraries (Enterprise Java Beans - EJB) JAR files, or any other archives, and create an installable EAR file using the serviceDeploy command (see "Installing a module on a production server" on page 63).

Naming conventions for staging modules

Within the library, there are requirements for the names of the staging modules. These names are unique for a specific module. Name any other modules required to deploy the application so that conflicts with the staging module names do not occur. For a module named *myService*, the staging module names are:
- *myService*App
- *myService*EJB
- *myService*EJBClient
- *myService*Web

Note: The serviceDeploy command only creates the *myService* Web staging module if the service includes a WSDL port type service.

Considerations when using libraries

Using libraries provides consistency of business objects and consistency of processing amongst modules because each calling module has its own copy of a specific component. To prevent inconsistencies and failures it is important to make sure that changes to components and business objects used by calling modules are coordinated with all of the calling modules. Update the calling modules by:
1. Copying the module and the latest copy of the libraries to the production server
2. Rebuilding the installable EAR file using the serviceDeploy command
3. Stopping the running application containing the calling module and reinstall it
4. Restarting the application containing the calling module

EAR file overview

An EAR file is a critical piece in deploying a service application to a production server.

An enterprise archive (EAR) file is a compressed file that contains the libraries, enterprise beans, and JAR files that the application requires for deployment.

You create a JAR file when you export your application modules from WebSphere Integration Developer. Use this JAR file and any other artifact libraries or objects as input to the installation process. The serviceDeploy command creates an EAR file from the input files that contain the component descriptions and Java code that comprise the application.

Preparing to deploy to a server

After developing and testing a module, you must export the module from a test system and bring it into a production environment for deployment. To install an application you also should be aware of the paths needed when exporting the module and any libraries the module requires.

Before beginning this task, you should have developed and tested your modules on a test server and resolved problems and performance issues.

This task verifies that all of the necessary pieces of an application are available and packaged into the correct files to bring to the production server.

Note: You can also export an enterprise archive (EAR) file from WebSphere Integration Developer and install that file directly into WebSphere Process Server.

Important: If the services within a component use a database, install the application on a server directly connected to the database.

1. Locate the folder that contains the components for the module you are to deploy.

   The component folder should be named *module-name* with a file in it named *module*.module, the base module.
2. Verify that all components contained in the module are in component subfolders beneath the module folder.

   For ease of use, name the subfolder similar to *module/component*.
3. Verify that all files that comprise each component are contained in the appropriate component subfolder and have a name similar to *component-file-name*.component.

   The component files contain the definitions for each individual component within the module.
4. Verify that all other components and artifacts are in the subfolders of the component that requires them.

   In this step you ensure that any references to artifacts required by a component are available. Names for components should not conflict with the names the serviceDeploy command uses for staging modules. See "Naming conventions for staging modules" on page 61.
5. Verify that a references file, *module*.references, exists in the module folder of step 1.

   The references file defines the references and the interfaces within the module.
6. Verify that a wires file, *module*.wires, exists in the component folder.

   The wires file completes the connections between the references and the interfaces within the module.
7. Verify that a manifest file, *module*.manifest, exists in the component folder.

   The manifest lists the module and all the components that comprise the module. It also contains a classpath statement so that the serviceDeploy command can locate any other modules needed by the module.
8. Create a compressed file or a JAR file of the module as input to the serviceDeploy command that you will use to prepare the module for installation to the production server.

Example folder structure for MyValue module prior to deployment

The following example illustrates the directory structure for the module MyValueModule, which is made up of the components MyValue, CustomerInfo, and StockQuote.

```
MyValueModule
   MyValueModule.manifest
   MyValueModule.references
   MyValueModule.wiring
   MyValueClient.jsp
process/myvalue
   MyValue.component
   MyValue.java
   MyValueImpl.java
service/customerinfo
```

```
   CustomerInfo.component
   CustomerInfo.java
   Customer.java
   CustomerInfoImpl.java
service/stockquote
   StockQuote.component
   StockQuote.java
   StockQuoteAsynch.java
   StockQuoteCallback.java
   StockQuoteImpl.java
```

Install the module onto the production systems as described in "Installing a module on a production server."

Installing a module on a production server

This topic describes the steps involved in taking an application from a test server and deploying it into a production environment.

Before deploying a service application to a production server, assemble and test the application on a test server. After testing, export the relevant files as described in "Preparing to deploy to a server" on page 61 and bring the files to the production system to deploy. See the information centers for WebSphere Integration Developer and WebSphere Application Server Network Deployment, v6.0 for more information.

1. Copy the module and other files onto the production server.

   The modules and resources (EAR, JAR, RAR, and WAR files) needed by the application are moved to your production environment.
2. Run the serviceDeploy command to create an installable EAR file.

   This step defines the module to the server in preparation for installing the application into production.
   a. Locate the JAR file that contains the module to deploy.
   b. Issue the serviceDeploy command using the JAR file from the previous step as input.
   c. Select the EAR file to install in the administrative console of the server.
   d. Click **Save** to install the EAR file.
3. Save the configuration. The module is now installed as an application.
4. Start the application.

The application is now active and work should flow through the module.

Monitor the application to make sure the server is processing requests correctly.

Creating an installable EAR file using serviceDeploy

To install an application in the production environment, take the files copied to the production server and create an installable EAR file.

Before starting this task, you must have a JAR file that contains the module and services you are deploying to the server. See "Preparing to deploy to a server" on page 61 for more information.

The serviceDeploy command takes a JAR file, any other dependent EAR, JAR, RAR, WAR and ZIP files and builds an EAR file that you can install on a server.

1. Locate the JAR file that contains the module to deploy.

2. Issue the serviceDeploy command using the JAR file from the previous step as input.

   This step creates an EAR file.
3. Select the EAR file to install in the administrative console of the server.
4. Click **Save** to install the EAR file.

Deploying applications using ANT tasks

This topic describes how to use ANT tasks to automate the deployment of applications to WebSphere Process Server, v6.0. By using ANT tasks, you can define the deployment of multiple applications and have them run unattended on a server.

This task assumes the following:
- The applications being deployed have already been developed and tested.
- The applications are to be installed on the same server or servers.
- You have some knowledge of ANT tasks.
- You understand the deployment process.

Information about developing and testing applications is located in the WebSphere Integration Developer, v6.0 information center.

The reference portion of the information center for WebSphere Application Server, v6.0 contains a section on application programming interfaces. ANT tasks are described in the package com.ibm.websphere.ant.tasks. For the purpose of this topic, the tasks of interest are ServiceDeploy and InstallApplication.

If you need to install multiple applications concurrently, develop an ANT task before deployment. The ANT task can then deploy and install the applications on the servers without your involvement in the process.
1. Identify the applications to deploy.
2. Create a JAR file for each application.
3. Copy the JAR files to the target servers.
4. Create an ANT task to run the ServiceDeploy command to create the EAR file for each server.
5. Create an ANT task to run the InstallApplication command for each EAR file from step 4 on the applicable servers.
6. Run the ServiceDeploy ANT task to create the EAR file for the applications.
7. Run the InstallApplication ANT task to install the EAR files from step 6.

The applications are correctly deployed on the target servers.

Example of deploying an application unattended

This example shows an ANT task contained in a file myBuildScript.xml.

```
<?xml version="1.0">

<project name="OwnTaskExample" default="main" basedir=".">
 <taskdef name="servicedeploy"
   classname="com.ibm.websphere.ant.tasks.ServiceDeployTask" />
 <target name="main" depends="main2">
  <servicedeploy scaModule="c:/synctest/SyncTargetJAR"
   ignoreErrors="true"
   outputApplication="c:/synctest/SyncTargetEAREAR"
   workingDirectory="c:/synctest"
   noJ2eeDeploy="true"
   cleanStagingModules="true"/>
 </target>
</project>
```

This statement shows how to invoke the ANT task.

```
${WAS}/bin/ws_ant -f myBuildScript.xml
```

Tip: Multiple applications can be deployed unattended by adding additional project statements into the file.

Use the administrative console to verify that the newly installed applications are started and processing the workflow correctly.

Installing and uninstalling business process and human task applications

Verify that the business process container or task container is installed and configured for each application server on which you want to install your application.

Before you install a business process or human task application, make sure that the following conditions are true:
- All stand-alone servers are running.
- In each cluster, at least one server on which you want to install Enterprise JavaBeans modules with processes or tasks is running.

You can install business process and task applications from the administrative console, from the command line, or by running an administrative script, for example.

Attention: When you run the administrative commands to install a business process application or a human task application, do not use the `-conntype NONE` option as an installation option.

Deploy the application. For more information, see , which is in the WebSphere Application Server information center.

All business process templates and human task templates are put into the start state.

If you installed a process application on a cluster, verify that the application uses the data source that is named after the cluster. For example, if the application was generated using the default data source `BPEDB`, change the data source for the application to BPEDB_*cluster_name*, where *cluster_name* is the name of the cluster on which you installed the application.

Before you can create process instances or task instances, you must start the application.

Deployment of models

When WebSphere Integration Developer generates the deployment code for your process, the constructs in the process or task model are mapped to various Java 2 Enterprise Edition (J2EE) constructs and artifacts. All deployment code is packaged into the enterprise application (EAR) file. Each new version of a model that is to be deployed must be packaged into a new enterprise application.

When you install an enterprise application that contains business process model or human task model J2EE constructs, the model constructs are stored as *process templates* or *task templates*, as appropriate, in the Business Process Choreographer database. If the database system is not running, or if it cannot be accessed, the deployment fails. Newly installed templates are, by default, in the started state. However, the newly installed enterprise application is in the stopped state. Each installed enterprise application can be started and stopped individually.

New versions of a process template or task template have the same name, but a different valid-from attribute. You can deploy many different versions of a process template or task template, each in a different application. All the different process versions are stored in the database.

If you do not specify a valid-from date, the date is determined as follows:

- For a human task, the valid-from date is the date on which the application was installed.
- For a business process, the valid-from date is the date on which the process was modeled.

Attention: No two versions of the same process can have the same valid-from date. If you want to install different versions of the same process, specify a different valid-from date for each version.

Application management

This topic explains application management considerations that are specific to enterprise applications that contain business processes or human tasks.

Enterprise applications that contain business processes differ in some important respects from enterprise applications that do not contain business processes or human tasks.

Module distribution

You can distribute Service Component Architecture (SCA) Enterprise JavaBeans (EJB) modules that contain business processes or human tasks, or both, to deployment targets. A deployment target can be a server or a cluster.

Stopping a business process or human task application

Before you try to stop a business process or human task application, delete any related business process instance or human task instance. Stop the process template or task template, to prevent new instances from being created. Otherwise, an error message is written to the `SystemOut.log` file, and the application does not stop.

Uninstalling a business process or human task application

Before you can successfully uninstall a business process application, all business process templates must be stopped and all process instances must be deleted.

Before you can successfully uninstall a human task application, all human task templates must be stopped and all task instances must be deleted.

A process administrator can use Business Process Choreographer Explorer to terminate and delete any surviving process instances and human task instances. For details about how to stop business process templates and human task templates, and uninstall these applications, see "Uninstalling business process and human task applications."

Uninstalling business process and human task applications

To uninstall an enterprise application that contains business processes or human tasks, perform the following actions:

1. Stop all process and task templates in the application.

   This action prevents the creation of process and task instances.

   a. Click **Applications** → **Enterprise Applications** in the administrative console navigation pane.

   b. Select the application that you want to stop.

   c. Under Related Items, click **EJB Modules**, then select an Enterprise JavaBeans (EJB) module. If you have more than one EJB module in the list, select the EJB that correspond to your Service Component Architecture (SCA) module that contains the business process or human task. You can find the corresponding EJB by appending EJB to the SCA module name. For example, if your SCA module was named `TestProcess`, the EJB module is `TestProcessEJB.jar`.

   d. Under Additional Properties, click **Business Processes** or **Human Tasks**, or both, as appropriate.

   e. Select all process and task templates by clicking the appropriate check box.

   f. Click **Stop**.

   Repeat this step for all EJB modules that contain business process templates or human task templates.

2. Verify that all the stand-alone servers, the database, and at least one application server for each cluster are running:

   - In a Network Deployment (ND) environment, the ND server, all ND-managed stand-alone application servers, and at least one application server must be running for each cluster where the application is installed.
   - If you use the administrative commands to uninstall applications, make sure that the administrative client is connected to a server process:
     - In an ND environment, the process is the deployment manager.
     - In a stand-alone environment, the process is the application server.

3. Verify that no process instances or task instances exist.

   If necessary, a process administrator can use Business Process Choreographer Explorer to delete any process instances.

4. Stop and uninstall the application:

   a. Click **Applications** → **Enterprise Applications** in the administrative console navigation pane.

   b. Select the application that you want to uninstall and click **Stop**. This step fails if any process instances still exist in the application.

   c. Delete all process instances.

   d. Select again the application that you want to uninstall.

   e. Click **Uninstall**.

The process application is uninstalled.

Installing applications with embedded WebSphere Adapters

If an application is developed with a WebSphere Adapter embedded, the adapter is deployed with the application. You do not need to install the adapter separately. The steps to install an application with an embedded adapter are described.

This task should only be performed if the application is developed with an embedded WebSphere Adapter.

1. Assemble an application with resource adapter archive (RAR) modules in it. See Assembling applications.
2. Install the application following the steps in 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.

   Note: When installing a RAR file onto a server, WebSphere Application Server looks for the manifest (MANIFEST.MF) for the connector module. It looks first in the connectorModule.jar file for the RAR file and loads the manifest from the _connectorModule.jar file. If the class path entry is in the manifest from the connectorModule.jar file, then 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 console. For more information, see Resource Adapter settings and WebSphere relational resource adapter settings.
3. Save the changes. Click **Finish > Save**.
4. Create connection factories for the newly installed application
   a. Open the administrative console.
   b. Select the newly installed application Click **Applications > Enterprise Applications >** *application name*.
   c. Click **Connector Modules** in the Related Items section of the page.
   d. Select the RAR file. Click on *filename.rar*
   e. Click **Resource adapter** in the Additional Properties section of the page.
   f. Click **J2C Connection Factories** in the Additional Properties section of the page.
   g. Click on an **existing connection factory** to update it, or **New** to create a new one.

      Note: If the WebSphere Adapter was configured using an EIS Import or EIS Export a ConnectionFactory or ActivationSpec will exist and can be updated.

   If you install an adapter that includes native path elements, consider the following: If you have more than one native path element, and one of the native libraries (native library A) is dependent on another library (native library B), then you must copy native library B to a system directory. Because of limitations on most UNIX systems, an attempt to load a native library does not look in the current directory.

   After you create and save the connection factories, you can modify the resource references defined in various modules of the application and specify the Java Naming and Directory Interface (JNDI) names of the connection factories wherever appropriate.

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 classloader, 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.

WebSphere Adapter

A WebSphere Adapter (or JCA Adapter, or J2C Adapter) is a system-level software driver that a Java application uses to connect to an enterprise information system (EIS). WebSphere Adapters conform to version 1.5 of the JCA specification.

A WebSphere Adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application.

An application server vendor extends its system once to support the J2EE Connector Architecture (JCA) and is then assured of seamless connectivity to multiple EISs. Likewise, an EIS vendor provides one standard WebSphere Adapter with the capability to plug into any application server that supports the connector architecture.

WebSphere Process Server provides the WebSphere Relational Resource Adapter (RRA) implementation. This WebSphere Adapter provides data access through JDBC calls to access the database dynamically. The connection management is based on the JCA connection management architecture. It provides connection pooling, transaction, and security support. WebSphere Process Server version 6.0 supports JCA version 1.5.

Data access for container-managed persistence (CMP) beans is managed by the WebSphere Persistence Manager indirectly. The JCA specification supports persistence manager delegation of the data access to the WebSphere Adapter without specific knowledge of the back-end store. For the relational database access, the persistence manager uses the relational resource adapter to access the data from the database. You can find the supported database platforms for the JDBC API at the WebSphere Process Server prerequisite Web site.

IBM supplies resource adapters for many enterprise systems separately from the WebSphere Process Server package, including (but not limited to): the Customer Information Control System (CICS), Host On-Demand (HOD), Information Management System (IMS), and Systems, Applications, and Products (SAP) R/3.

In WebSphere Process Server, EIS Imports and EIS Exports are used to interface with WebSphere Adapters. As an alternative, applications with WebSphere Adapters can be written by developing EJB session beans or services with tools such as Rational Application Developer. The session bean uses the javax.resource.cci interfaces to communicate with an enterprise information system through the WebSphere Adapter.

WebSphere Adapter deployment considerations

The deployment of WebSphere Adapters requires specific options regarding scope.

You can deploy a WebSphere Adapter in two ways, using the administrative console:

- Standalone - the adapter is installed at the node level and is not associated with a specific application.

  Note: Deployment of standalone WebSphere Adapters is not supported in WebSphere Process Server v6.0.
- Embedded - the adapter is part of an application, deploying the application also deploys the adapter.

For embedded WebSphere Adapters:

- the RAR file can be application-scoped within an SCA module (with EIS imports or exports).
- the RAR file can be application-scoped within a non-SCA module. The application itself, containing the EIS imports and exports, is a separate SCA module.

You should not install standalone WebSphere Adapters.

Note: The administrative console does not preclude the installation of standalone WebSphere Adapters, but this should not be done. WebSphere Adapters should be embedded in applications.

Only embedded WebSphere Adapters are appropriate for deployment in WebSphere Process Server. Furthermore, deployment of an embedded WebSphere Adapter is only supported for RAR files that are application-scoped within an SCA module; deployment in a non-SCA module is not supported.

Installing Standalone WebSphere Adapters

WebSphere Adapters should be embedded in applications. Standalone WebSphere Adapters are not support in WebSphere Process Server v6.0. These instructions are for reference only. If you intend to use a standalone WebSphere Adapter you should install it, as described here. You can alternatively use an embedded adapter, which is installed automatically as part of the installation of the associated application.

You should configure the database before installing the adapter.

You must have access to, and be part of the necessary security role for, the administrative console to perform this task.

1. Open the Install RAR file dialog window.

   On the administrative console:

   a. Expand **Resources**

   b. Click **Resource Adapters**

c. Select the scope at which you want to define this resource adapter. (This scope becomes the scope of your connection factory). You can choose cell, node, cluster, or server.

d. Click **Install RAR**

A window opens in which you can install a JCA connector and create, for it, a WebSphere Adapter. You can also use the New button, but the New button creates only a new resource adapter (the JCA connector must already be installed on the system).

Note: When installing a RAR file using this dialog, 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. The node on which the file is installed is determined by the scope on the Install RAR page. (The scope you set on the Resource Adapters page determines the scope of the new resource adapters, which you can install at the server, node, or cell level.)

2. Install the RAR file

From the dialog, install the WebSphere Adapter in the following manner:

a. Browse to the location of the JCA connector. If the RAR file is on the local workstation select Local Path and browse to find the file. If the RAR file is on a network server, select **Server path** and specify the fully qualified path to the file.

b. Click **Next**

c. Enter the resource adapter name and any other properties needed under General Properties. If you install a J2C Resource Adapter that includes native path elements, consider the following: If you have more than one native path element, and one of the native libraries (native library A) is dependent on another library (native library B), then you must copy native library B to a system directory. Because of limitations on most UNIX systems, an attempt to load a native library does not look in the current directory.

d. Click **OK**.

WebSphere Adapter applications as members of clusters

Cluster deployment is not supported in WebSphere Process Server v6.0. This information is included for completeness. WebSphere Adapter module applications can be cloned as members of a cluster under certain conditions.

WebSphere Adapter module applications can be one of three types, depending on the flow of information through the adapter:

- A WebSphere Adapter application with only EIS exports - only inbound traffic.
- A WebSphere Adapter application with only EIS imports - only outbound traffic.
- A WebSphere Adapter application with both EIS imports and exports - bidirectional traffic.

Clusters are used to provide scalability and availability to your applications in a network deployment environment.

WebSphere Adapter module applications that have either inbound or bidirectional traffic, cannot be cloned as members of a cluster. An application with purely outbound traffic can be cloned as a member of a cluster.

An application which has an inbound or bidirectional WebSphere Adapter (i.e., including EIS exports) can still be given availability in a network deployment by use of an external Operating System High Availability (HA) management software package, such as HACMP, Veritas or Microsoft Cluster Server.

WebSphere Business Integration Adapter applications as members of clusters

Cluster deployment is not supported in WebSphere Process Server v6.0. This information is included for completeness. WebSphere Business Integration Adapter module applications can be cloned as members of a cluster under certain conditions.

WebSphere Business Integration Adapter module applications can be one of three types, depending on the flow of information through the adapter:

- A WebSphere Business Integration Adapter application with only EIS exports - only inbound traffic.
- A WebSphere Business Integration Adapter application with only EIS imports - only outbound traffic.
- A WebSphere Business Integration Adapter application with both EIS imports and exports - bidirectional traffic.

Clusters are used to provide scalability and availability to your applications in a network deployment environment.

WebSphere Business Integration Adapter module applications that have either inbound or bidirectional traffic, cannot be cloned as members of a cluster. An application with purely outbound traffic can be cloned as a member of a cluster.

An application which has inbound or bidirectional WebSphere Business Integration Adapter (i.e., including EIS exports) can still be given availability in a network deployment by use of an external Operating System High Availability (HA) management software package, such as HACMP, Veritas or Microsoft Cluster Server.

Installing EIS applications

An EIS application module, a service component architecture (SCA) module that follows EIS application module pattern can be deployed to either a J2SE platform or a J2EE platform.

The steps required to deploy an EIS module depend on the platform.

See the subsequent tasks for detailed information.

Deploying an EIS application module to the J2SE platform

The EIS Module can be deployed to J2SE platform however only EIS Import will be supported.

You need to create an EIS application module with a JMS Import binding in the WebSphere Integration Development environment before commencing this task.

An EIS application module would be furnished with a JMS Import binding when you want to access EIS systems asynchronously through the use of message queues.

Deploying to the J2SE platform is the only instance where the binding implementation can be executed in the non-managed mode. The JMS Binding requires asynchronous and JNDI support, neither of which is provided by the base service component architecture or the J2SE. The J2EE Connector Architecture does not support non-managed inbound communication thus eliminating EIS Export.

When the EIS application module with the EIS Import is deployed to J2SE, in addition to the module dependencies, the WebSphere Adapter used by the import has to be specified as the dependency, in the manifest or any other form supported by SCA.

Deploying an EIS application module to the J2EE platform

The deployment of EIS module to the J2EE platform results in an application, packaged as an EAR file deployed to the server. All the J2EE artifacts and resources are created, the application is configured and ready to be run.

You need to create an EIS module with a JMS Import binding in the WebSphere Integration Development environment before commencing this task.

The deployment to the J2EE platform creates the following J2EE artifacts and resources:

Table 29. Mapping from bindings to J2EE artifacts

Binding in the SCA module	Generated J2EE artifacts	Created J2EE resources
EIS Import	Resource References generated on the module Session EJB.	ConnectionFactory
EIS Export	Message Driven Bean, generated or deployed, depending on the listener interface supported by the Resource Adapter.	ActivationSpec
JMS Import	Message Driven Bean (MDB) provided by the runtime is deployed, resource references generated on the module Session EJB. Note that the MDB is only created if the import has a receive destination.	• ConnectionFactory • ActivationSpec • Destinations
JMS Export	Message Driven Bean provided by the runtime is deployed, resource references generated on the module Session EJB	• ActivationSpec • ConnectionFactory • Destinations

When the import or export defines a resource like a ConnectionFactory, the resource reference is generated into the deployment descriptor of the module Stateless Session EJB. Also, the appropriate binding is generated into the EJB binding file. The name, to which resource reference is bound, is either the value of the target attribute, if one is present, or default JNDI lookup name given to the resource, based on the module name and import name.

Upon deployment, the implementation locates the module session bean and uses it to lookup the resources.

During deployment of the application to the server, the EIS installation task will check for the existence of the element resource to which it is bound. If it does not exist, and the SCDL file specifies at least one property, the resource will be created and configured by the EIS installation task. If the resource does not exist, no action is taken, it is assumed that resource will be created before execution of the application.

When the JMS Import is deployed with a receive destination, a Message Driver Bean (MDB) is deployed. It listens for replies to requests that have been sent out. The MDB is associated (listens on) the Destination sent with the request in the JMSreplyTo header field of the JMS message. When the reply message arrives, the MDB uses its correlation ID to retrieve the callback information stored in the callback Destination and then invokes the callback object.

The installation task creates the ConnectionFactory and three destinations from the information in the import file. In addition, it creates the ActivationSpec to enable the runtime MDB to listen for replies on the receive Destination. The properties of the ActivationSpec are derived from the Destination/ConnectionFactory properties. If the JMS provider is a SIBus Resource Adapter, the SIBus Destinations corresponding to the JMS Destination are created.

When the JMS Export is deployed, a Message Driven Bean (MDB) (not the same MDB as the one deployed for JMS Import) is deployed. It listens for the incoming requests on the receive Destination and then dispatches the requests to be processed by the SCA. The installation task creates the set of resources similar to the one for JMS Import, an ActivationSpec, ConnectionFactory used for sending a reply and two Destinations. All the properties of these resources are specified in the export file. If the JMS provider is an SIBus Resource Adapter, the SIBus Destinations corresponding to JMS Destination are created.

Troubleshooting a failed deployment

This topic describes the steps to take to determine the cause of a problem when deploying an application. It also presents some possible solutions.

This topic assumes the following things:
- You have a basic understanding of debugging a module.
- Logging and tracing is active while the module is being deployed.

The task of troubleshooting a deployment begins after you receive notification of an error. There are various symptoms of a failed deployment that you have to inspect before taking action.

1. Determine if the application installation failed.

   Examine the system.out file for messages that specify the cause of failure. Some of the reasons an application might not install include the following:
   - You are attempting to install an application on multiple servers in the same Network Deployment cell.
   - An application has the same name as an existing module on the Network Deployment cell to which you are installing the application.
   - You are attempting to deploy J2EE modules within an EAR file to different target servers.
2. If the application is installed correctly, examine it to determine if it started.

   If the application is not running, the failure occurred when the server attempted to initiate the resources for the application.

   a. Examine the system.out file for messages that will direct you on how to proceed.
   b. Determine if the resources are started.

      Resources that are not started prevent an application from running to protect against lost information. The reasons for a resource not starting include:
      - Bindings are specified incorrectly
      - Resources are not configured correctly
      - Resources are not included in the resource archive (RAR) file
      - Web resources not included in the Web services archive (WAR) file
   c. Determine if any components are missing.

      The reason for missing a component is an incorrectly built enterprise archive (EAR) file. Make sure that the all of the components required by the module are in the correct folders on the test system on which you built the Java archive (JAR) file. "Preparing to deploy to a server" on page 61 contains additional information.
3. Examine the application to see if there is information flowing through it.

   Even a running application can fail to process information. Reasons for this are similar to those mentioned in step 2b.
4. Correct the problem and restart the application.

References

Programming interfaces

BOChangeSummary

This interface provides enhancements to the ChangeSummary interface so that it can manage the Business Graph Change Summary header.

Purpose

BOChangeSummary adds the functionality to the ChangeSummary interface enabling it to manage the Business Graph Change Summary header.

The ChangeSummary interface provides access to change history information for the data objects in a data graph. Change history covers any modifications made to the data graph starting from the point when logging was activated. If logging is no longer active, the log includes only changes that are made up to the point when logging was deactivated. Otherwise, it includes all changes up to the point at which the ChangeSummary is being interrogated.

Note: The addOldValue application programming interface (API) allows you to set the value if you do not have the old value. This API expects that the value is not set prior to being called. If you attempt to call it and the value is already set, you receive an exception.

Example

This example shows how to use BOChangeSummary.

```
BOFactory factoryService =
    (BOFactory) new
ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOChangeSummary changeSummaryService =
    (BOChangeSummary) new
ServiceManager().locateService("com/ibm/websphere/bo/BOChangeSummary");
BODataObject dataObjectService =
    (BODataObject) new
ServiceManager().locateService("com/ibm/websphere/bo/BODataObject");

DataObject productCategoryBG =
  factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
                          "ProductCategoryBG");
DataObject productCategory =
  productCategoryBG.createDataObject("productCategory");
DataObject product1 =
  productCategory.createDataObject("product");
DataObject product2 =
  productCategory.createDataObject("product");

// Two mechanisms to find the change summary:
//
// 1. From the Business Graph.
ChangeSummary changeSummary =
    (ChangeSummary) productCategoryBG.get("changeSummary");

// 2. From any data object using a convenience method.
ChangeSummary changeSummary2 =
  dataObjectService.getChangeSummary(productCategory);

changeSummary.beginLogging();

productCategory.setBoolean("domestic", false);
product1.set("description", "NewValue");
product1.set("description", "NewValue2");
product2.set("description", "NewValue");
product2.set("description", "NewValue2");

changeSummary.endLogging();

List changedDataObjects =
  changeSummary.getChangedDataObjects();
Iterator i = changedDataObjects.iterator();
while (i.hasNext()) {
  DataObject dataObject = (DataObject) i.next();

  if (changeSummary.isDeleted(dataObject)) {
      // ...
      continue;
  }

  if (changeSummary.isCreated(dataObject)) {
      // ...
        continue;
    }

    if (changeSummaryService.isUpdated(dataObject)) {
      // ...
        continue;
  }
}

// Annotate the product category object with an object changed event.
changeSummaryService.setCreated(productCategory);

// Annotate a property on the product category
// object with a property changed event.
changeSummaryService.addOldValue(productCategory, "ID", null, true);

List changeSummarySettings =
    changeSummary.getOldValues(productCategory);
```

```
Iterator i2 = changeSummarySettings.iterator();
while (i2.hasNext()) {
    ChangeSummary.Setting setting = (ChangeSummary.Setting) i2.next();
    System.out.println("setting getProperty(): " + setting.getProperty());
    System.out.println("setting getValue(): " + setting.getValue());
    System.out.println("setting isSet(): " + setting.isSet());
}

DataObject product3 = productCategory.createDataObject("product");

//move product3 to location 0 in the list and
//Make the appropriate changes to the changesummary.
changeSummaryService.markListEntryMoved(product3, productCategory, "product", 0);

product3.set("description", "NewValue");

//Explicitly mark the list entry as created in the changesummary.
changeSummaryService.markSimpleTypeCreated("NewValue", product, "description");

product3.set("description", "NewValue2");
changeSummaryService.markSimpleTypeCreated("NewValue2", product, "description");

//Delete the list entry and add a list change entry in the change summary.
changeSummaryService.markSimpleTypeDeleted("NewValue2", product, "description");

product1.delete();
// The old container will return the productCategory object
// since that is the old container for product1.
changeSummaryService.getOldContainer(product1);

// This will return the old containment property that is
// what the property name is in the productCategory object.
changeSummaryService.getOldContainmentProperty(product1);
```

Related information

Interface BOChangeSummary APIs

BOCopy

This interface facilitates copying a graph of business objects or a business graph that contains a graph of business objects.

Purpose

There are two forms of this interface. The first is used with a graph of a business object or a business graph containing a graph of business objects. The only requirement is that it is a service data object. The other is used when a subset of a source business graph is being copied into a target business graph.

The business object framework defines two different forms of copy. The first is a straightforward copy by the value, with a deep and a shallow variant. This means that you can either copy only the top level, or also copy the children. This copy mechanism can be applied to both a graph of business objects or a business graph containing a graph of business objects. Its implementation is unrelated to the shape of the source object.

The second form is intended for a use model where a subset of a source business graph is being copied into a target business graph. This form not only copies the source business object (and its descendants if it is the deep variant), but also copies the Change, Event, and Verb Header information that is pertinent to the business object(s) being copied into the target business graph.

Example

This is an example of the copy/Shallow form.

```
BOFactory factoryService =
    (BOFactory) new
ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOCopy copyService =
    (BOCopy) new
ServiceManager().locateService("com/ibm/websphere/bo/BOCopy");

// Use the Factory Create model for the top level object.
DataObject productCategoryBG =
  factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
                         "ProductCategoryBG");

// Use the Containment Create model on the contained objects.
DataObject productCategory = productCategoryBG.createDataObject("productCategory");
DataObject product = productCategory.createDataObject("product");

// Copy a business graph (deep).
DataObject newProductCategoryBG = copyService.copy(productCategoryBG);

// Copy a business object (deep).
DataObject newProductCategory = copyService.copy(productCategory);

// Copy a business object (shallow).
DataObject newProductCategory2 = copyService.copyShallow(productCategory);
```

This is an example of the copyIntro form.

```
BOFactory factoryService =
    (BOFactory) new
ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOCopy copyService =
    (BOCopy) new
ServiceManager().locateService("com/ibm/websphere/bo/BOCopy");

// Use the Factory Create model for the top level object.
DataObject productCategoryBG =
  factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
                        "ProductCategoryBG");

// Use the Containment Create model on the contained objects.
DataObject productCategory = productCategoryBG.createDataObject("productCategory");
DataObject product = productCategory.createDataObject("product");

// Copy a child business object with a business graph context,
// into a new business graph.
DataObject productBG =
  factoryService.create("http://www.scm.com/ProductTypes/ProductBG",
                        "ProductBG");
copyService.copyInto(product, productBG, "product");
```

This is an example of the copyIntro Complex form.

```
BOFactory factoryService =
  (BOFactory) new
ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOCopy copyService =
  (BOCopy) new ServiceManager().locateService("com/ibm/websphere/bo/BOCopy");

// If you want to perform a complex copy from a source business graph
// to a target business graph and maintain context where:
//
// - The source business graph contains:
//   - ProductCategoryBG productCategoryBG
//   - ProductCategory productCategory0 (parent productCategoryBG)
//   - ProductCategory productCategory1 (parent productCategory0)
//   - ProductCategory productCategory2 (parent productCategory0)
//   - Product product1 (parent productCategory1)
//   - Product product2 (parent productCategory2)
//
// - The target business graph contains:
//   - ProductInventoryBG productInventoryBG
//   - ProductInventory productInventory (parent productInventoryBG)
//     - Property oldProduct (initially empty)
//     - Property newProduct (initially empty)
//
// The following code copies product1 from the source business graph
// to the target business graph's ProductInventory business object's
// oldProduct property, and product2 from the source business graph
// to the target business graph's ProductInventory business object's
// newProduct property.

// Create the source business graph and its graph of business objects.
DataObject productCategoryBG =
  factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
                        "ProductCategoryBG");
DataObject productCategory0 = productCategoryBG.createDataObject("productCategory");
DataObject productCategory1 = productCategory0.createDataObject("productCategory");
DataObject productCategory2 = productCategory0.createDataObject("productCategory");
DataObject product1 = productCategory1.createDataObject("product");
DataObject product2 = productCategory2.createDataObject("product");

// Create the target business graph.
DataObject productInventoryBG =

factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductInventoryBG",
                       "ProductInventoryBG");
DataObject productInventory =
    productInventoryBG.createDataObject("productInventory");

// Copy product1 and product2 from the source business
// graph, with their associated ChangeSummary/EventSummary context, into
// the target business graph's ProductInventory business object's oldProduct and
// newProduct properties. Use two mechanisms to demonstrate how path can be used.
copyService.copyInto(product1, productInventory, "oldProduct");
DataObject oldProduct = productInventory.getDataObject("oldProduct");
oldProduct.delete();
copyService.copyInto(product1, productInventoryBG, "productInventory/oldProduct");
```

Related information

Interface BOCopy APIs

BODataObject

This interface makes it easier to retrieve a data object business graph, Change Summary, or Event Summary.

Purpose

BODataObject allows additional capability beyond what the data object interface provides by making it easier to retrieve a data object business graph, Change Summary, or Event Summary. If it is contained in a business graph hierarchy, BODataObject provides helper methods.

Example

This example shows how to use BODataObject.

```
BOFactory factoryService =
    (BOFactory) new
ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
```

```
BODataObject dataObjectService =
    (BODataObject) new
ServiceManager().locateService("com/ibm/websphere/bo/BODataObject");

DataObject productCategoryBG =
  factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
                         "ProductCategoryBG");
DataObject productCategory =
    productCategoryBG.createDataObject("productCategory");
DataObject product =
    productCategory.createDataObject("product");

dataObjectService.getChangeSummary(product).beginLogging();

productCategory.setBoolean("domestic", false);
product.set("description", "NewValue");
product.set("description", "NewValue2");


DataObject businessGraph    = dataObjectService.getBusinessGraph(productCategory);
ChangeSummary changeSummary = dataObjectService.getChangeSummary(productCategory);
BOEventSummary eventSummary = dataObjectService.getEventSummary(productCategory);

// This will return the productCategory DataObject
// which is the top level object
dataObjectService.getRootBusinessObject(product);
```

Related information

Interface BODataOBject APIs

BOEquality

This interface provides the ability to determine if two business graphs or business objects are equivalent.

Purpose

BOEquality supports equality of business graphs or business objects. The default form of equality is deep, but a shallow form can also be used. Therefore, you can determine if two business graphs or business objects are equivalent based on including various levels of descendants.

Example

This example shows how to use BOEquality.

```
BOEquality equalityService =
(BOEquality) newServiceManager().locateService("com/ibm/websphere/bo/BOEquality");

// Deep equality check.
if (equalityService.isEqual(dataObject1, dataObject2) == true) {
    // ...
}
// Shallow equality check.
if (equalityService.isEqualShallow(dataObject1, dataObject2) == true) {
    // ...
}
```

Related information

Interface BOEquality APIs

BOEventSummary

This interface provides the interface for managing the content of the business graph Event Summary header.

Purpose

BOEventSummary allows for managing the content of the business graph Event Summary header by associating particular metadata with business objects.

Example

This example shows how to use the BOEventSummary interface.

```
BOFactory factoryService =
    (BOFactory) new
ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BODataObject dataObjectService =
    (BODataObject) new
ServiceManager().locateService("com/ibm/websphere/bo/BODataObject");

DataObject productCategoryBG =

factoryService.create("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
                     "ProductCategoryBG");
DataObject productCategory =
    productCategoryBG.createDataObject("productCategory");
DataObject product1 =
    productCategory.createDataObject("product");
DataObject product2 =
    productCategory.createDataObject("product");

// 1. If you have a business graph.
BOEventSummary eventSummary1 =
    (BOEventSummary) productCategoryBG.get("eventSummary");

// 2. If you do not have a business graph, obtain one.
DataObject businessGraph =
    dataObjectService.getBusinessGraph(product1);
BOEventSummary eventSummary2 =
    (BOEventSummary) productCategoryBG.get("eventSummary");

// 3. If you do not have a business graph, use a helper.
BOEventSummary eventSummary3 =
    dataObjectService.getEventSummary(product1);

// This is an alternate way to obtain the event summary.
eventSummary1.setObjectEventID(productCategory, "PC1_ID");
eventSummary1.setObjectEventID(product1, "P1_ID");
eventSummary1.setObjectEventID(product2, "P2_ID");

// Given a data object, obtain its object event ID.
String objectEventID = eventSummary1.getObjectEventID(product1);
```

Related information

Interface BOEventSummary APIs

BOFactory

This interface provides the capability to create a business graph or a business object.

Purpose

There are three typical models used for creating a business graph or a business object with the BOFactory interface:

- Factory Create - Used to create a data object independent of an existing graph of data.

- Containment Create (containment attach inferred) - Used to create a child data object of an existing data object.
- Factory Create / Containment Attach - Uses a factory create mechanism to create a business object that is attached to a graph of business objects using the DataObject.setDataObject() method.

Examples

This example shows several different options for the Factory Create model.

```
BOFactory factoryService = (BOFactory) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOType typeService = (BOType) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOType");

// 1. Use the business object Factory Create model with a target namespace
// and a complex type definition name.
DataObject productCategory1 = factoryService.create
  ("http://www.scm.com/ProductCategoryTypes", "ProductCategory");

// 2. Use the business object Factory Create model with the type.
DataObject productCategory2 = factoryService.createByType(typeService.getType
  ("http://www.scm.com/ProductCategoryTypes", "ProductCategory"));

// 3. Use the business object Factory Create model with the class.
DataObject productCategory3 = factoryService.createByClass
  (com.scm.pc.model.Product.class);
// Use the business graph Factory Create model with a target namespace
// and a complex type definition name.

// It also automatically creates the ChangeSummary and EventSummary headers.
DataObject productCategoryBG = factoryService.create
  ("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
  "ProductCategoryBG");
```

This is an example of the Containment Create model.

```
BOFactory factoryService = (BOFactory) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");

// Use the Factory Create model for the top level object.
DataObject productCategoryBG = factoryService.create
  ("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
  "ProductCategoryBG");

// Use the Containment Create model on the contained objects.
DataObject productCategory = productCategoryBG.createDataObject
  ("productCategory");
DataObject product = productCategory.createDataObject("product");
```

This is an example of the Factory Create / Containment Attach model.

```
BOFactory factoryService = (BOFactory) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
// Use the Factory Create model on the top level object.
DataObject productCategoryBG = factoryService.create
  ("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
// Use the Factory Create model on what will eventually become a contained object.
DataObject productCategory = factoryService.create
  ("http://www.scm.com/ProductCategoryTypes", "ProductCategory");
// Containment attach
productCategoryBG.setDataObject("productCategory", productCategory);
```

Related information

Interface BOFactory APIs

BOType

This interface provides a mechanism to obtain the service data object (SDO) type of a business graph or business object that mirrors what Class.forName() provides for Java class names.

Purpose

Obtain the SDO type of a business graph or business object, mirroring what Class.forName() provides for Java class names. This information can be obtained by specifying the following information:

- The statically typed SDO interface class name.
- The target namespace and the complex type name of the dynamically typed SDO.
- The target namespace and the global anonymous complex type name

Examples

This is an example of providing an interface class that represents a statically typed SDO.

```
BOType typeService = (BOType) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOType");

Type productType1 = typeService.getTypeByClass(com.scm.pc.model.Product.class);
```

This is an example of providing the target namespace and the complex type name.

```
BOType typeService = (BOType) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOType");

Type productType2 =
  typeService.getType("http://www.scm.com/ProductTypes", "Product");
```

This is an example of providing the target namespace and the anonymous complex type element name.

```
BOType typeService = (BOType) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOType");

Type productType3 =
  typeService.getTypeByElement("http://www.scm.com/Product", "product");
```

Related information

Interface BOType APIs

BOTypeMetadata

This interface provides the capability of taking an annotation blob that conforms to the BOTypeMetadata pattern and transforms it into a set of service data object (SDO) (and performs the reserve transform).

Purpose

Annotations can be read at runtime by using the SDO implementation specific set of APIs. However, the problem with these APIs is that they return a blob. Therefore, the business object framework provides the BOTypeMetadata to read the blob, validate it, and transform it into a usable data object structure.

Example

This example shows how to use the BOTypeMetadata interface.

```
BOFactory factoryService = (BOFactory) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOTypeMetadata typeMetadataService = (BOTypeMetadata) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOTypeMetadata");

DataObject product =
    factoryService.create("http://www.scm.com/ProductTypes",
                          "Product");

// Use EMF to get the annotation blob, then use the BOTypeMetadata service
// to convert it into a data object.
String productIDPropertyInfoString = (String) ((EObject)product).
  eClass().getEStructuralFeature("iD").
  getEAnnotation("http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0").
  getDetails().get("appinfo");
DataObject productIDPropertyInfo =
  typeMetadataService.transformAnnotationToDataObject
  (productIDPropertyInfoString);

// Read and update the annotation.
// ...

// Use the BOTypeMetadata service to transform the data object graph back into
// a string, and then use EMF to set the string in the XML schema.
String productIDPropertyInfoString2 =
  typeMetadataService.transformDataObjectToAnnotation
  (productIDPropertyInfo);
((EObject)product).eClass().getEStructuralFeature("iD").
  getEAnnotation("http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0").
  getDetails().put("appinfo",productIDPropertyInfoString2);
```

Related information

Interface BOTypeMetadata APIs

BOXMLDocument

This interface provides the mechanisms for creating and representing an XML document in memory.

Purpose

This interface allows you to build and to represent an XML document.

Example

This is an example of the BOXMLDocument interface.

```
public interface BOXMLDocument
{
   /**
    * Returns the root DataObject for the XML Document.
    * This object is an instance of the root element's type or subtype.
    *
    * @return root DataObject for the XMLDocument
    */
public DataObject
getDataObject();


   /**
    * Returns the targetnamespace for the root element and null
    *   if there is no targetnamespace.
    * @return the targetnamespace URI for the root element.
    */
public String
getRootElementURI();
```

```
   /**
    * Returns the name of the root element.
    *
    * @return name of the root element.
    */
public String
getRootElementName();


   /**
    * Returns the XML version of the document, or null if not specified.
    *
    * @return XML version of the XML Document
    */
public String
getXMLVersion();


   /**
    * Set the XML version of the document, or null if not specified.
    *
    * @param xmlVersion XML version of the XML Document
    */
public void
setXMLVersion(String xmlVersion);


   /**
    * Returns the XML encoding of the document, or null if not specified.
    *   Default value is UTF-8.
    * @return Encoding of the XML Document
    */
public String
getEncoding();


   /**
    * Sets the XML encoding of the document, or null if not specified.
    *
    * @param encoding encoding used in the XML Document
    */
public void
setEncoding(String encoding);
```

Related information

Interface BOXMLDocument APIs

BOXMLSerializer

This interface provides the mechanisms for serializing and deserializing an XML document.

Purpose

This interface is used to serialize and deserialize a business graph or a graph of business objects.

Example

This is an example of the BOXMLSerializer interface.

```
BOFactory factoryService = BOFactory) new
  ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");
BOXMLSerializer xmlSerializerService =`(BOXMLSerializer) new
```

```
  ServiceManager().locateService("com/ibm/websphere/bo/BOXMLSerializer");

// Create the business object graph.
DataObject productCategoryBG = factoryService.create
  ("http://www.scm.com/ProductCategoryTypes/ProductCategoryBG",
  "ProductCategoryBG");
DataObject productCategory =
    productCategoryBG.createDataObject("productCategory");
DataObject product =
    productCategory.createDataObject("product");

DataObject productCategory3 =
    productCategoryBG3.createDataObject("productCategory");

FileOutputStream outfile2 = new
  FileOutputStream("productCategoryBGDocument.xml");
xmlSerializerService.writeXMLDocument(productCategoryBGDoc, outfile2);
FileInputStream infile2 = new FileInputStream("productCategoryBGDocument.xml");
BOXMLDocument productCategoryBG4Document =
 xmlSerializerService.readXMLDocument(infile2);
DataObject productCategoryBG5 = productCategoryBG4Document.getDataObject();
```

Related information

Interface BOXMLSerializer APIs

Component interface

This interface represents a service component.

Purpose

Use the methods in this interface to obtain information about a component. These methods:
- Return the interface type of a service component.
- Return a list of the interface types for interfaces within the service component.
- Return the component name.
- Return a reference object for a specific reference.
- Return a list of references within the service component.

Examples

This example is a list of interface type objects for the interfaces exposed by a service named myService.

```
ServiceManager serviceManager = new ServiceManager();
Service service = (Service)serviceManager.locateService("myService");
list myList = myService.getInterfaceTypes();
```

This example is a list of references within a service named myService.

```
ServiceManager serviceManager = new ServiceManager();
Service service = (Service)serviceManager.locateService("myService");
list myReferences = myService.getReferences();
```

Related information

Interface Component APIs

DataFactory interface

This interface creates Service Data Objects (SDOs).

Purpose

Use the methods in this interface to create SDOs of specific interface classes, types, or type and URI.

Examples

This example creates an SCartDO interface class SDO.

```
SCartDataAccessService scartDataAccessService =
   (SCartDataAccessService)serviceManager.locateService("scartDataAccessService");
  SCartDO scartDO = scartDataAccessService.create();
```

Related information

Interface DataFactory APIs

EndPointReference interface

This interface represents a Web service addressing endpoint reference.

Purpose

This interface provides the methods that support a service that has a Web Services Description Language (WSDL) port type interface. Use the methods in this interface to obtain information about or set properties for an interface.

Examples

This example shows a client obtaining the EndPointReference for the stockQuote service.

```
ServiceManager serviceManager = new ServiceManager();
Service service = (Service)serviceManager.locateService("stockQuote");
EndpointReference endpointReference = service.getEndpointReference();
```

Related information

Interface EndPointReference APIs

EndPointReferenceFactory interface

This interface is a factory for creating Web Services addressing endpoint references.

Purpose

This interface provides the method needed for a service to create an external EndPointReference. The properties can be set by the EndPointReference interface methods.

After creating the reference, your module must import it before referencing it.

Examples

This example shows how a service would create an EndPointReference and set a property.

```
ServiceManager sM = new ServiceManager();
EndpointReference endpointReference = sM.createEndpointReference("myServicePort");
enpointReference.setReferenceProperty("myId", "12345");
```

Related information

Interface EndPointReferenceFactory APIs

Service exceptions

When processing errors occur, some APIs throw exceptions. Specific classes represent these exceptions.

ServiceBusinessException class:

This class indicates a business exception during the execution of a business operation.

Purpose

This class contains the method that provides error information when a service encounters an exception. The service has to build either a message or a business object to return to the calling component upon failure.

Related information

Class ServiceBusinessException APIs

ServiceRuntimeException class:

This class indicates a timeout condition during the invocation or execution of a service.

Purpose

If a component has invoked a service asynchronously, it can request a response when the component is ready to process the data. On that request, there is a wait period for the response. If response exceeds the wait period, a ServiceRuntimeException is thrown. The component can then determine the cause of the exception using the getCause() or getMessage() methods.

Examples

```
StockQuoteAsync sQ =  (StockQuoteAsync)ServiceManager.locateService("stockQuote");
Ticket ticket = stockQuote.getQuoteRequest("IBM");
 // do something else

try {

 float quote = stockQuote.getQuoteResponse(ticket, Service.NO_WAIT);
  or
 float quote = stockQuote.getQuoteResponse(ticket, 10000);

} catch (ServiceTimeoutRuntimeException) {
 ...
}
```

Related information

Class ServiceRuntimeException APIs

InterfaceType interface

This interface represents a service interface.

Purpose

The methods in this interface provide information about a service interface. Use the methods to obtain the name of a service interface, the URI for a service interface and the valid operation types on the service interface.

Examples

The following example demonstrates how to get the URI for the stockQuote service interface.

```
ServiceManager serviceManager = new ServiceManager();
Service service = (Service)serviceManager.locateService("stockQuote");
URI uri = service.getURI();
```

Related information

Interface InterfaceType APIs

Service interface

This interface provides an interface and access to a service. The ServiceManager locateService() method implements this interface.

Purpose

The methods for this interface allow you to reference and determine information about the service interface. Use the service interface methods to:

- Find the EndPointReference for a service.
- Determine the preferred interaction style for the service.
- Invoke, either synchronously or asynchronously, the service.
- Retrieve the response from invoking a service asynchronously.

Examples

```
somecall invokeAsynch(someTicket, someservice somedata);
response = invokeResponse(someTicket, 10000);
```

Related information

Interface Service APIs

ServiceCallback interface

Service components that need to handle asynchronous callback interactions implement this interface.

Purpose

Use the method in this interface to return a response to a component that has made an asynchronous request to the service. To succeed, the component must pass the service a ticket when the client invokes the service.

Examples

This program implements a service that is an alarm for a client.

```
package sample.alarm;

import java.util.Date;

import com.ibm.websphere.sca.Service;
import com.ibm.websphere.sca.ServiceCallback;
import com.ibm.websphere.sca.ServiceManager;
import com.ibm.websphere.sca.Ticket;
import com.ibm.websphere.sca.scdl.OperationType;
import com.ibm.websphere.sca.scdl.Reference;
import com.ibm.websphere.sca.sdo.DataFactory;
import commonj.sdo.DataObject;
import commonj.sdo.Type;

/*
 *  This code implements the alarm interface and invokes the timer asynchronously.
 */
public class SimpleDIIAlarmImpl implements SimpleAlarm, ServiceCallback {

    public void setAlarm(String name, int duration) {

        ServiceManager serviceManager = new ServiceManager();
        // Submit the request
        // Get the setTimer input type and construct the argument accordingly
        Service asyncTimerService = (Service) serviceManager.locateService("timer");
        Reference reference = asyncTimerService.getReference();
        OperationType operationType = reference.getOperationType("startTimer");
        Type inputType = operationType.getInputType();
        DataFactory dataFactory = DataFactory.INSTANCE;
        DataObject input = dataFactory.create(inputType);
        input.set(0, new Integer(duration));
        input.set(1, name);

        // Invoke the timer service
        Ticket ticket =
      asyncTimerService.invokeAsyncWithCallback("startTimer", input);
        System.out.println("Sent async with callback.");
    }

    /*
     * @see com.ibm.websphere.sca.ServiceCallback#onInvokeResponse
     * (com.ibm.websphere.sca.Ticket, java.lang.Object, java.lang.Exception)
     */
    public void onInvokeResponse(Ticket arg0, Object arg1, Exception arg2) {

        System.out.println("onInvokeResponse()");
        if (arg2 != null) {
            System.out.println("Timer ran into exception: " + arg2.getMessage());
        } else {
            System.out.println("Alarm " + arg1 + " went off at " +
           new Date(System.currentTimeMillis()));
        }
    }
}
```

Related information

Interface ServiceCallback APIs

ServiceImplAsync interface

Service components that generically handle asynchronous service invocations implement this interface.

Purpose

Use this interface to implement an asynchronous interface for a service. The method in this class allows a client to invoke the service, passing it a ticket object so that the service can call back the client once processing is complete.

Examples

The following example shows a client calling a service that has an asynchronous implementation.

```
}
 // initiation and other processing.

 invokeAsynch(OperationType serviceOperation, inputObject,
  serviceCallback clientReturn, clientTicket)
```

Related information

Interface ServiceImplAsync APIs

ServiceImplSync interface

Service components that generically handle synchronous service invocations implement this interface.

Purpose

Use this interface to invoke a service with a synchronous interface. Generally, the services have short duration and respond quickly to the component.

Examples

The following shows a client invoking the stockQuote service synchronously.

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
Float quote = (Float)stockQuote.invoke("getQuote", new Object[] {"IBM"})[0];
```

Related information

Interface ServiceImplSync APIs

ServiceManager class

The ServiceManager class provides an object that facilitates access to services within WebSphere Process Server.

Purpose

Use ServiceManager() to create an instance of a ServiceManager object. Then use the object to locate a service.

Examples

The following example shows using ServiceManager to create a ServiceManager instance for a single service.

```
ServiceManager serviceManager = new ServiceManager();
```

The next example shows how to create a ServiceManager instance to manage multiple services.

```
InputStream myReferences = new FileInputStream("MyReferences.references");
ServiceManager serviceManager = new ServiceManager(myReferences);
```

Related information

Class ServiceManager APIs

Ticket interface

This interface represents a correlation object that ties an asynchronous service request and response together.

Purpose

Use this interface to provide communication between an asynchronous service and a client. When the service completes processing a client request, it uses the ticket to contact the client with the response through the serviceCallback interface.

A ticket is long lived, can be persisted and reused across threads and processes. A ticket also implements the equals and hashCode methods, which allow it to be used as a key.

Examples

This example shows the StockQuoteSync service implementing a ticket interface for getQuoteAsync and getQuoteResponse.

```
public interface StockQuoteAsync {

 // deferred response
 public Ticket getQuoteAsync(String symbol);
 public float getQuoteResponse(Ticket ticket, long timeout);

 // callback
 public Ticket getQuoteAsync(String symbol, StockQuoteCallback callback);
}
```

This example shows the client invoking the StockQuoteAsync service and then requesting the response.

```
StockQuoteAsync sQ =  (StockQuoteAsync)serviceManager.locateService("stockQuote");
Ticket ticket = stockQuote.getQuoteAsync("IBM");

 // do something else

float quote = stockQuote.getQuoteResponse(ticket, Service.WAIT);
```

Related information

Interface Ticket API

Commands

Enter commands from the command line on WebSphere Process Server.

serviceDeploy

Describes the purpose and syntax of the serviceDeploy command including a description of all of the parameters and their values. An example of the command is included.

Purpose

The serviceDeploy command builds an .ear file from a .jar or .zip file that contains service components.

Note: Parameters are not case-sensitive.

Roles

This command can be issued by users with the following roles:
- Administrator
- Deployer

Syntax

serviceDeploy *inputarchive* [<**-workingDirectory** *temppath*> <**-outputAppliation** *outputpathname.ear*> **-noJ2eeDeploy -freeform -cleanStagingModules -keep -ignoreErrors** <**-classpath** *jarpathname;rarpathname...*> **-help**]

Parameters

inputarchive
: A required, positional parameter that specifies the .jar, .zip or .ear file that contains the application to be deployed. If the command is not issued from the path in which the file resides, this must be the full path for the file. The .zip file can be either a nested archive or an Eclipse ProjectInterchange format file.

-classpath
: An optional parameter that specifies the locations of required resource files

(.jar and .rar) files. The path to each file should be a fully-qualified path separated by semicolons (;) with no spaces.

-freeform
An optional parameter that specifies that the J2EE subdirectory in the service.jar should be treated as a free-form project.

-help
An optional parameter used to display the parameters for this command.

-ignoreErrors
An optional parameter that specifies that the serviceDeploy command builds an .ear file regardless of errors while building or validating the application. By default, the serviceDeploy command does not generate an .ear file if there are errors with an application.

-cleanStagingModules
An optional parameter that specifies whether to delete staging modules within an input .ear file before deployment. By default, the serviceDeploy command imports existing staging modules and their contents.

-keep
An optional parameter that specifies whether to save any temporary files generated after deployment. By default, the serviceDeploy command deletes the temporary workspace.

-noJ2eeDeploy
An optional parameter that specifies whether the application requires EJB deployment after generating the .ear file. By default, the serviceDeploy command runs the J2EE deployers for the application.

-outputApplication
An optional parameter that specifies the name of the .ear file the serviceDeploy command creates. The default is *inputjarfile*.ear, where *inputjarfile* is the filename minus the extension specified for the input .jar file.

-workingDirectory
An optional parameter that specifies a directory the serviceDeploy command uses to write temporary files.

Inputs

The following file types can be used as input to the serviceDeploy command:

jar The most useful file type for the simplest applications. The resulting ear file contains a single jar and any needed generated staging modules. The jar must contain the *service*.module file.

zip (Project Interchange)
You can export from WebSphere Integration Developer an archive file in project interchange format. This format is unique to the Eclipse development. The exported zip file must contains exactly one project with the *service*.module file. The resulting ear file contains any number of modules, depending upon exactly what is in the project interchange.

zip You can create a zip file containing jar files, war files, and rar files. Exactly one jar file must contain the *service*.module file. All contained archives become members of the final exported ear file.

ear You can always run the serviceDeploy command against an ear file as long as exactly one jar file in the ear contains a *service*.module file.

Output

When serviceDeploy completes processing, it creates an .ear file in the directory from which the command is run unless the **-outputApplication** parameter is specified.

Exceptions

N/A

Example of serviceDeploy command

The following command example:

- Creates an application file called MyValueModule.ear from the MyValueModule.jar file.
- Specifies that the resources reside in the directories c:\java\myvaluemoduleres.rar and c:\java\commonres.jar.
- Enables the J2EE subdirectory within the .jar file as free-form.
- Keeps the temporary files generated during deployment.

```
servicedeploy MyValueModule.jar
-classpath "c:\java\myvaluemoduleres.rar;c:\java\commonres.jar"
-noj2eedeploy -freeform true -keep
```

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

i5/OS
IBM
the IBM logo
AIX
AIX 5L
CICS
CrossWorlds
DB2
DB2 Universal Database
Domino
HelpNow
IMS
Informix
iSeries
Lotus
Lotus Notes
MQIntegrator
MQSeries
MVS
Notes
OS/390
OS/400
Passport Advantage
pSeries
Redbooks
SupportPac
WebSphere
z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

WebSphere Process Server, Version 6.0

Printed in USA