



Developing and Deploying Modules



Developing and Deploying Modules

Note

Before using this information, be sure to read the general information in the Notices section at the end of this document.

1 February 2008

This edition applies to version 6, release 1, modification 0 of WebSphere Process Server for Multiplatforms (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, send an e-mail message to 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.

© Copyright International Business Machines Corporation 2005, 2008. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures v

Tables vii

Part 1. Developing applications . . . 1

Chapter 1. Overview of developing modules 3

Developing service modules	4
Developing service components	5
Invoking components	7
Overview of isolating modules and targets	10
HTTP bindings	14
Overriding the generated Service Component Architecture implementation.	15
Overriding a Service Data Object to Java conversion	16
Runtime rules used for Java to Service Data Objects conversion.	17

Chapter 2. Developing client applications for business processes and tasks 21

Developing EJB client applications for business processes and human tasks	21
Accessing the EJB APIs	22
Querying business-process and task-related objects	27
Developing applications for business processes	61
Developing applications for human tasks	80
Developing applications for business processes and human tasks	97
Handling exceptions and faults	102
Developing Web service API client applications	104
Introduction: Web services	104
Web service components and sequence of control	105
Overview of the Web services APIs	105
Requirements for business processes and human tasks	106
Developing client applications.	106
Copying artifacts	107
Developing client applications in the Java Web services environment	115
Developing client applications in the .NET environment.	124
Querying business-process and task-related objects.	129
Developing JMS client applications	132
Introduction to JMS	132
Requirements for business processes.	133
Accessing the JMS interface	133
Structure of a Business Process Choreographer JMS message	135
Authorization for JMS renderings	136

Overview of the JMS API	137
Developing JMS applications	138
Developing Web applications for business processes and human tasks, using JSF components	140
Adding the List component to a JSF application	145
Adding the Details component to a JSF application	152
Adding the CommandBar component to a JSF application	154
Adding the Message component to a JSF application	158
Developing JSP pages for task and process messages	161
User-defined JSP fragments.	162
Creating plug-ins to customize human task functionality.	163
Creating API event handlers	164
Creating notification event handlers	166
Creating plug-ins to post-process people query results.	167
Installing plug-ins.	169
Registering plug-ins	170

Part 2. Deploying applications. . . 171

Chapter 3. Overview of preparing and installing modules 173

Libraries and JAR files overview	173
EAR file overview	175
Preparing to deploy to a server	175
Considerations for installing service applications on clusters	177

Chapter 4. Installing a module on a production server 179

Creating an installable EAR file using serviceDeploy	180
Deploying applications using Apache Ant tasks	180

Chapter 5. Installing business process and human task applications 183

Installing business process and human task applications interactively	185
Configuring process application data source and set reference settings	185
Uninstalling business process and human task applications, using the administrative console	187
Uninstalling business process and human task applications, using administrative commands	188

Chapter 6. Installing adapters 191

Chapter 7. Installing EIS applications 193

Deploying an EIS application module to the J2SE platform 194
Deploying an EIS application module to the J2EE platform 195

Chapter 8. Troubleshooting a failed deployment 197

Deleting J2C activation specifications 198
Deleting SIBus destinations. 199

Part 3. Appendixes 201

Notices 203

Figures

1. Simple invocation model	11	4. Isolated invocation model invoking UpdatedCalculateFinal.	14
2. Multiple applications invoking a single service	12	5. Relationship amongst module, component and library	174
3. Isolated invocation model invoking UpdateCalculateFinal	13		

Tables

1. WSDL type to Java class conversion	19	26. API methods for controlling the life cycle of process instances.	79
2.	28	27. API methods for controlling the life cycle of activity instances.	79
3. Columns in the ACTIVITY view	40	28. API methods for variables and custom properties	80
4. Columns in the ACTIVITY_ATTRIBUTE view	41	29. API methods for task templates.	94
5. Columns in the ACTIVITY_SERVICE view	42	30. API methods for task instances	95
6. Columns in the APPLICATION_COMP view	42	31. API methods for working with escalations	96
7. Columns in the ESCALATION view	43	32. API methods for variables and custom properties	96
8. Columns in the ESCALATION_CPROP view	44	33. Mapping of the reference bindings to JNDI names	142
9. Columns in the ESCALATION_DESC view	45	34. How Business Process Choreographer interfaces are mapped to client model objects .	145
10. Columns in the ESC_TEMPL view	45	35. bpe:list attributes	151
11. Columns in the ESC_TEMPL_CPROP view	46	36. bpe:column attributes	152
12. Columns in the ESC_TEMPL_DESC view	47	37. bpe:details attributes	154
13. Columns in the PROCESS_ATTRIBUTE view	47	38. bpe:property attributes	154
14. Columns in the PROCESS_INSTANCE view	47	39. bpe:commandbar attributes	157
15. Columns in the PROCESS_TEMPLATE view	48	40. bpe:command attributes	158
16. Columns in the QUERY_PROPERTY view	49	41. bpe:form attributes.	161
17. Columns in the TASK view	50	42. Mapping from bindings to J2EE artifacts	193
18. Columns in the TASK_CPROP view	53	43. Mapping from bindings to J2EE artifacts	195
19. Column in the TASK_DESC view	53		
20. Columns in the TASK_TEMPL view	53		
21. Columns in the TASK_TEMPL_CPROP view	55		
22. Columns in the TASK_TEMPL_DESC view	55		
23. Columns in the WORK_ITEM view	56		
24. API methods for process templates.	78		
25. API methods are related to starting process instances	78		

Part 1. Developing applications

Chapter 1. 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 service 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.

WebSphere Process Server supports two types of service modules: modules for business services and mediation modules. A module for business services implements the logic of a process. A mediation module allows communication between applications by transforming the service invocation to a format understood by the target, passing the request to the target and returning the result to the originator.

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 service component is associated with interfaces, references and implementations. The interface defines a contract between a service component and a calling component. With WebSphere Process Server, a service module can either export a service component for use by other modules or import a service component for use. To invoke a service component, a calling module references the interface to the service 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:

1. Define interfaces for the components in the module
2. Define, modify, or manipulate business objects used by service components
3. Define or modify service components through its interfaces.

Note: A service component is defined through its interface.

4. Optionally, export or import service components.
5. Create an EAR file you use to install a module that uses components. You create the file using either the export EAR feature in WebSphere Integration Developer or the serviceDeploy command to create an EAR file to install a service module that uses service 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 service component so that a consumer can import those interfaces and use the service component as if it were local. A developer uses either

strongly-typed interfaces or dynamically-typed interfaces to implement or invoke the service component. The interfaces and their methods are described in the References section within this information center.

After installing service 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.

Service component development considerations

When developing a service component, ask yourself the following questions:

- Will this service component be exported and used by another module?
If so, make sure the interface you define for the component can be used by another module.
- Will the service component take a relatively long time to run?
If so, consider implementing an asynchronous interface to the service component.
- Is it beneficial to decentralize the service component?
If so, consider having a copy of the service component in a service 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 information center.

Developing service modules

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

Before you begin

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

About this task

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

Procedure

1. Identify service components other modules can use.
Once you have identified the service components, continue with Developing service components.

2. Identify service components within an application that could use service components in other service modules.
Once you have identified the service components and their target components, continue with Invoking components.
3. Connect the client components with the target components through wires.

Developing service components

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

Before you begin

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

About this task

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

Note: A single service component can contain multiple interfaces.

Procedure

1. Define the data object to move data between the caller and the service component.
The data object and its type is part of the interface between the callers and the service component.
2. Define an interface that the callers will use to reference the service component.
This interface definition names the service component and lists any methods available within the service 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.

Important: An asynchronous component interface cannot have a `joinsTransaction` property set to `true`.

- a. Define the interface that represents the synchronous service component.
 - b. Define the implementation of the service component.
 - c. Continue with step 6.
5. Develop a synchronous implementation.
 - a. Define the interface that represents the synchronous service component.
 - b. Define the implementation of the service component.
 6. Save the component interfaces and implementations in files with a `.java` extension.
 7. Package the service 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 on page 6.

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 service component, if calling a service component in another service module. The “Administering” section of this information center describes configuring the wires.

Examples of developing components

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

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

The following block of code implements the service 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 service component. The first section of code defines the interface to the service 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);  
}
```

What to do next

Invoke the service.

Invoking components

Components with modules can use components on any node of a WebSphere Process Server cluster.

Before you begin

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

About this task

Components can use any service 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 locating and then creating the reference to the required component.

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

Important: When invoking a component that resides on a different server than the one on which the calling module is running, you must perform additional configurations to the servers. The configurations required depend on whether the component is called asynchronously or synchronously. How to configure the application servers in this case is described in related tasks.

Procedure

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 creates a `ServiceManager` class.

```
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 `MyValue`, code that calls another component.

```
public class MyValueImpl implements MyValue {  
  
    public float myValue 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, Service.WAIT);  
  
            // assign  
            value = quote * customer.getNumShares();  
  
        }  
    }  
}
```

```

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

```

What to do next

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.

Before you begin

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

About this task

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.

Procedure

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.

Examples 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 service = (Service)serviceManager.locateService("multiParamInf");

Reference reference = service.getReference();

OperationType methodMultiType =
    reference.getOperationType("methodWithMultiParameter");

Type t = methodMultiType.getInputType();

BOFactory boFactory = (BOFactory)serviceManager.locateService

```

```

        ("com/ibm/websphere/bo/BOFactory");

    DataObject paramObject = boFactory.createbyType(t);

    paramObject.set(0,"input1")
    paramObject.set(1,"input2")
    paramObject.set(2,"input3")

    service.invoke("methodMultiParamater",paramObject);

```

The following example uses the invoke method with a WSDL port type interface as the target.

```

Service serviceOne = (Service)serviceManager.locateService("multiParamInfWSDL");

DataObject dob = factory.create("http://MultiCallWSServerOne/bos", "SameBO");
    dob.setString("attribute1", stringArg);

DataObject wrapBo = factory.createByElement
("http://MultiCallWSServerOne/wsd1/ServerOneInf", "methodOne");
    wrapBo.set("input1", dob); //wrapBo encapsulates all the parameters of methodOne
    wrapBo.set("input2", "XXXX");
    wrapBo.set("input3", "yyyy");

DataObject resBo= (DataObject)serviceOne.invoke("methodOne", wrapBo);

```

Overview of isolating modules and targets

When developing modules, you will identify services that multiple modules can use. Leveraging services this way minimizes your development cycle and costs. When you have a service used by many modules, you should isolate the invoking modules from the target so that if the target is upgraded, switching to the new service is transparent to the calling module. This topic contrasts the simple invocation model and the isolated invocation model and provides an example of how isolation can be useful. While describing a specific example, this is not the only way to isolate modules from targets.

Simple invocation model

While developing a module, you might use services that are located in other modules. You do this by importing the service into the module and then invoking that service. The imported service is “wired” to the service exported by the other module either in WebSphere Integration Developer or by binding the service in the administrative console. Simple invocation model illustrates this model.

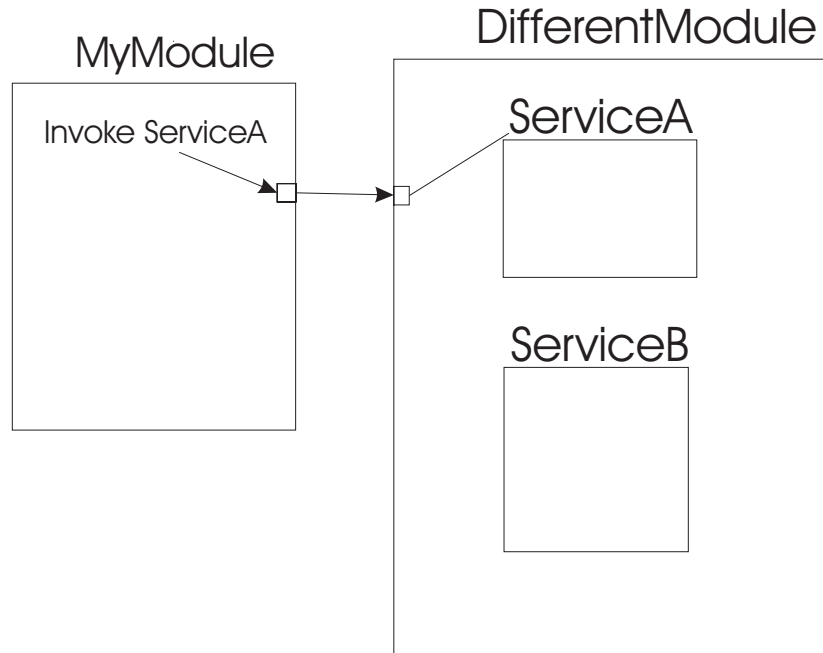


Figure 1. Simple invocation model

Isolated invocation model

To change the target of an invocation without stopping invoking modules, you can isolate the invoking modules from the target of the invocation. This allows the modules to continue processing while you change the target because you are not changing the module itself but the downstream target. Example of isolating applications shows how isolation allows you to change the target without affecting the status of the invoking module.

Example of isolating applications

Using the simple invocation model, multiple modules invoking the same service would look much like Multiple applications invoking a single service . MODA, MODB, and MODC all invoke CalculateFinalCost.

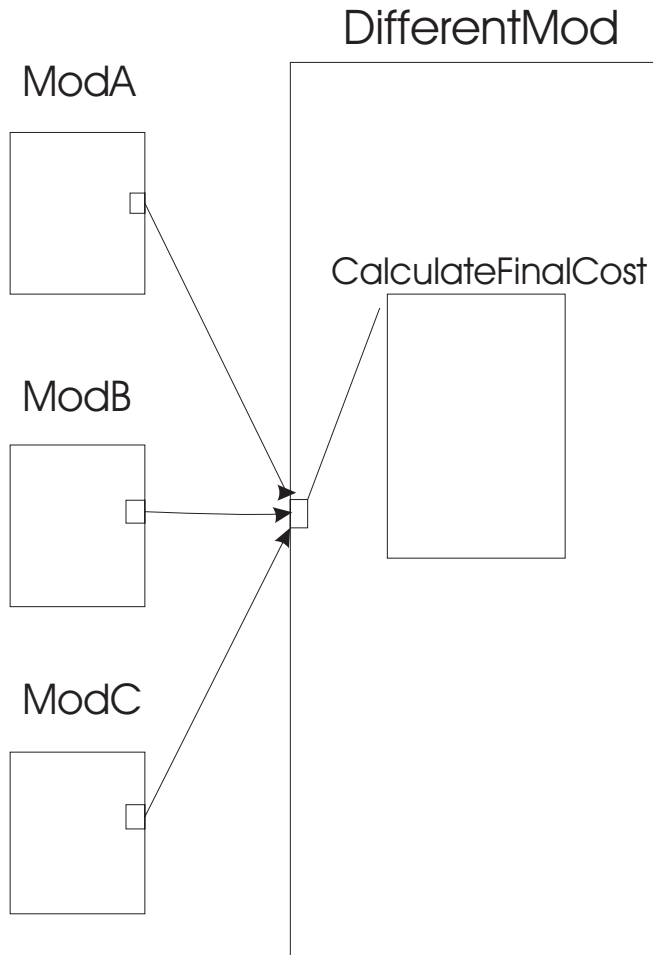


Figure 2. Multiple applications invoking a single service

The service provided by CalculateFinalCost needs updating so that new costs are reflected in all modules that use the service. The development team builds and tests a new service UpdatedCalculateFinal to incorporate the changes. You are ready to bring the new service into production. Without isolation, you would have to update all of the modules invoking CalculateFinalCost to invoke UpdateCalculateFinal. With isolation, you only have to change the binding that connects the buffer module to the target.

Note: Changing the service this way allows you to continue to provide the original service to other modules that may need it.

Using isolation, you create a buffer module between the applications and the target (see Isolated invocation model invoking UpdateCalculateFinal).

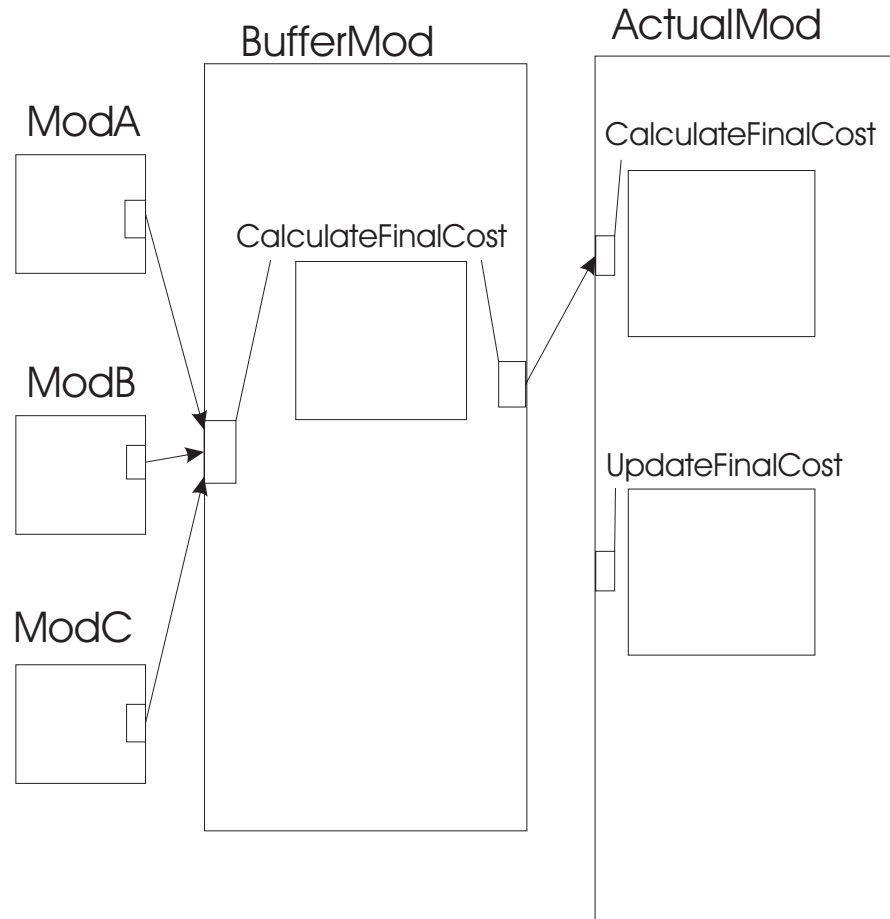


Figure 3. Isolated invocation model invoking UpdateCalculateFinal

With this model, the invoking modules do not change, you just have to change the binding from the buffer module import to the target (see Isolated invocation model invoking UpdatedCalculateFinal).

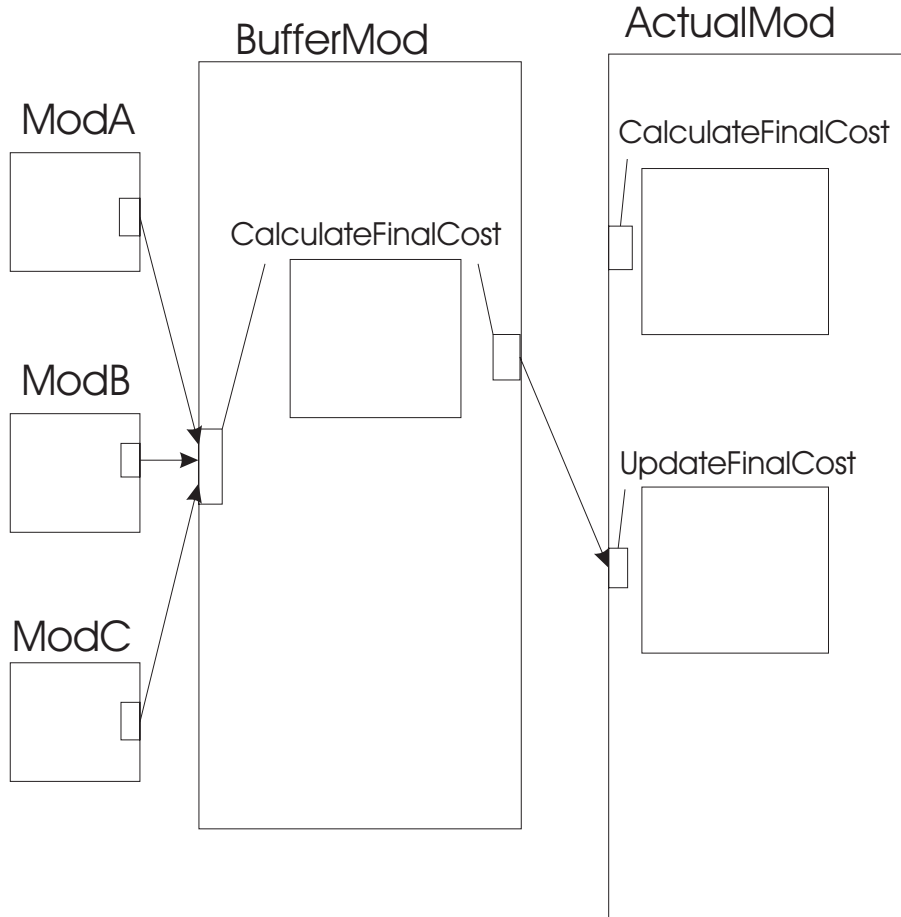


Figure 4. Isolated invocation model invoking UpdatedCalculateFinal

If the buffer module invokes the target synchronously, when you restart the buffer module (whether a mediation module or a service for business module) the results returned to the original application come from the new target. If the buffer module invokes the target asynchronously, the results returned to the original application come from the new target on the next invocation.

Related tasks

Changing targets

Changing the target of a reference provides applications with the flexibility of taking advantage of advances in components as they happen without recompiling and reinstalling the application.

HTTP bindings

The HTTP binding is designed to provide Service Component Architecture (SCA) connectivity to HTTP. This allows existing or newly-developed HTTP applications to participate in Service Oriented Architecture (SOA) environments.

In addition, a network of SCA runtime environments can communicate across an existing HTTP infrastructure.

The HTTP binding exposes several HTTP features:

- Messages are presented to mediation components in a manner that preserves HTTP format and message header information. This provides a more familiar view to HTTP application programmers, users and administrators.
- An existing data binding framework is extended for HTTP conventions and provides mapping between SCA messages and HTTP message headers and bodies.
- Imports and exports can be configured to support a range of common HTTP features.
- When you install an SCA module containing HTTP imports or exports, the runtime environment is automatically configured appropriately to allow connectivity to HTTP.

Detailed instructions on creating HTTP imports and exports can be found in the information center at **WebSphere Integration Developer > Developing integration applications > HTTP data binding**.

Related tasks

Displaying HTTP Bindings

After deploying an application, you may want to examine the HTTP bindings to make sure they are correct.

Changing HTTP export bindings

The administrative console allows you to change the configuration of HTTP export bindings without changing the original source and then redeploying the application.

Changing HTTP import bindings

The administrative console allows you to change the configuration of HTTP import bindings without changing the original source and then redeploying the application.

Overriding the generated Service Component Architecture implementation

Sometimes, the conversion the system creates between a Java code and a Service Data Object (SDO) may not meet your needs. Use this procedure to replace the default Service Component Architecture (SCA) class implementation with your own.

Before you begin

Make sure that you have generated the Java to Web Services Definition Language (WSDL) type conversion using either WebSphere Integration Developer or the genMapper command.

About this task

You override a generated component that maps a Java type to a WSDL type by replacing the generated code with code that meets your needs. Consider using your own map if you have defined your own Java classes. Use this procedure to make the changes.

Procedure

1. Locate the generated component. The component is named `java_classMapper.component`.
2. Edit the component using a text editor.
3. Comment out the generated code and provide your own method.
Do not change the file name that contains the component implementation.

This is an example of a generated component to replace:

```
private DataObject javatodata_setAccount_output(Object myAccount) {  
  
    // You can override this code for custom mapping.  
    // Comment out this code and write custom code.  
  
    // You can also change the Java type that is passed to the  
    // converter, which the converter tries to create.  
  
    return SDOJavaObjectMediator.java2Data(myAccount);  
  
}
```

Copy the component and other files to the directory in which the containing module resides, and either wire the component in WebSphere Integration Developer or generate an enterprise archive (EAR) file using the `serviceDeploy` command.

Related concepts

“Runtime rules used for Java to Service Data Objects conversion” on page 17
To correctly override generated code, or to determine possible run time exceptions related to Java to Service Data Object (SDO) conversions, an understanding of the rules involved is important. The majority of the conversions are straightforward, but there are some complex cases that the run time provides the best possibility when it converts the generated code.

Related reference

 [Java to XML conversion](#)

The system generates XML based on Java types using predefined rules.

 [genMapper command](#)

Use the `genMapper` command to generate a component that bridges as Service Component Architecture (SCA) reference to a Java interface.

Overriding a Service Data Object to Java conversion

Sometimes, the conversion the system creates between a Service Data Object (SDO) and a Java type object may not meet your needs. Use this procedure to replace the default implementation with your own.

Before you begin

Make sure that you have generated the WSDL to Java type conversion using either WebSphere Integration Developer or the `genMapper` command.

About this task

You override a generated component that maps a WSDL type to a Java type by replacing the generated code with code that meets your needs. Consider using your own map if you have defined your own Java classes. Use this procedure to make the changes.

Procedure

1. Locate the generated component. The component is named `java_classMapper.component`.
2. Edit the component using a text editor.
3. Comment out the generated code and provide your own method.
Do not change the file name that contains the component implementation.

This is an example of a generated component to replace:

```
private Object datatojava_get_customerAcct(DataObject myCustomerID,
    String integer)
{
    // You can override this code for custom mapping.
    // Comment out this code and write custom code.

    // You can also change the Java type that is passed to the
    // converter, which the converter tries to create.

    return SDOJavaObjectMediator.data2Java(customerID, integer) ;
}
```

Copy the component and other files to the directory in which the containing module resides, and either wire the component in WebSphere Integration Developer or generate an enterprise archive (EAR) file using the `serviceDeploy` command.

Related concepts

“Runtime rules used for Java to Service Data Objects conversion”

To correctly override generated code, or to determine possible run time exceptions related to Java to Service Data Object (SDO) conversions, an understanding of the rules involved is important. The majority of the conversions are straightforward, but there are some complex cases that the run time provides the best possibility when it converts the generated code.

Related reference

 [Java to XML conversion](#)

The system generates XML based on Java types using predefined rules.

 [genMapper command](#)

Use the `genMapper` command to generate a component that bridges as Service Component Architecture (SCA) reference to a Java interface.

Runtime rules used for Java to Service Data Objects conversion

To correctly override generated code, or to determine possible run time exceptions related to Java to Service Data Object (SDO) conversions, an understanding of the rules involved is important. The majority of the conversions are straightforward, but there are some complex cases that the run time provides the best possibility when it converts the generated code.

Basic types and classes

The run time performs a straightforward conversion between Service Data Objects and basic Java types and classes. Basic types and classes include:

- Char or java.lang.Character
- Boolean
- Java.lang.Boolean
- Byte or java.lang.Byte
- Short or java.lang.Short
- Int or java.lang.Integer
- Long or java.lang.Long
- Float or java.lang.Float
- Double or java.lang.Double
- Java.lang.String
- Java.math.BigInteger
- Java.math.BigDecimal
- Java.util.Calendar
- Java.util.Date
- Java.xml.namespace.QName
- Java.net.URI
- Byte[]

User-defined Java classes and arrays

When converting from a Java class or array to an SDO, the run time creates a data object that has a URI that is generated by inverting the package name of the Java type and has a type equal to the name of the Java class. For example, the Java class com.ibm.xsd.Customer is converted to an SDO and URI `http://xsd.ibm.com` with type Customer. The run time then inspects the contents of the Java class members and assigns the values to properties in the SDO.

When converting from an SDO to a Java type, the run time generates the package name by inverting the URI and the name of the type equals the type of the SDO. For example, the data object with type Customer and URI `http://xsd.ibm.com` generates an instance of the Java package com.ibm.xsd.Customer. The run time then extracts values from the properties of the SDO and assign those properties to fields in the instance of the Java class.

When the Java class is a user-defined interface, you must override the generated code and provide a concrete class that the run time can instantiate. If the run time cannot create the concrete class, an exception occurs.

Java.lang.Object

When a Java type is java.lang.Object the generated type is xsd:anyType. A module can invoke this interface with any SDO. The run time attempts to instantiate a concrete class the same way it does for user-defined Java classes and arrays, if the run time can find that class. Otherwise, the run time passes the SDO to the Java interface.

Even if the method returns a java.lang.Object type, the run time converts to an SDO only if the method returns a concrete type. The run time uses a similar conversion to that for converting user-defined Java classes and arrays to SDOs, as described by the next paragraph.

When converting from a Java class or array to an SDO, the run time creates a data object that has a URI that is generated by inverting the package name of the Java type and has a type equal to the name of the Java class. For example, the Java class `com.ibm.xsd.Customer` is converted to an SDO and URI `http://xsd.ibm.com` with type `Customer`. The run time then inspects the contents of the Java class members and assigns the values to properties in the SDO.

In either case, if the run time is unable to complete the conversion an exception occurs.

Java.util container classes

When converting to a concrete Java container class such as `Vector`, `HashMap`, `HashSet` and the like, the run time instantiates the appropriate container class. The run time uses a method similar to that used for user-defined Java classes and arrays to populate the container class. If the run time cannot locate a concrete Java class, the run time populates the container class with the SDO.

When converting concrete Java container classes to SDOs, the run time uses the generated schemas shown in “Java to XML conversion.”

Java.util interfaces

For certain container interfaces in the `java.util` package, the run time instantiates the following concrete classes:

Table 1. WSDL type to Java class conversion

Interface	Default concrete classes
Collection	HashSet
Map	HashMap
List	ArrayList
Set	HashSet

Related tasks

“Overriding the generated Service Component Architecture implementation” on page 15

Sometimes, the conversion the system creates between a Java code and a Service Data Object (SDO) may not meet your needs. Use this procedure to replace the default Service Component Architecture (SCA) class implementation with your own.

“Overriding a Service Data Object to Java conversion” on page 16

Sometimes, the conversion the system creates between a Service Data Object (SDO) and a Java type object may not meet your needs. Use this procedure to replace the default implementation with your own.

Related reference



Java to XML conversion

The system generates XML based on Java types using predefined rules.



genMapper command

Use the `genMapper` command to generate a component that bridges as Service Component Architecture (SCA) reference to a Java interface.

Chapter 2. Developing client applications for business processes and tasks

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

About this task

These clients can be Enterprise JavaBeans™ (EJB) clients, Web service clients, or Web clients that exploit the Business Process Choreographer Explorer JavaServer Faces (JSF) components. Business Process Choreographer provides Enterprise JavaBeans (EJB) APIs and interfaces for Web services for you to develop these clients. The EJB API can be accessed by any Java application, including another EJB application. The interfaces for Web services can be accessed from either Java environments or Microsoft® .Net environments.

Developing EJB client applications for business processes and human tasks

The EJB APIs provide a set of generic methods for developing EJB client applications for working with the business processes and human tasks that are installed on a WebSphere Process Server.

About this task

With these Enterprise JavaBeans (EJB) APIs, you can create client applications to do the following:

- Manage the life cycle of processes and tasks from starting them through to deleting them when they complete
- Repair activities and processes
- Manage and distribute the workload over members of a work group

The EJB APIs are 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 EJB APIs, see the Javadoc in the `com.ibm.bpe.api` package and the `com.ibm.task.api` package.

The following steps provide an overview of the actions you need to take to develop an EJB client application.

Procedure

1. Decide on the functionality that the application is to provide.
2. Decide which of the session beans that 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.
The users of your application must be assigned the appropriate authorization roles to call the methods that you include in your application, and to view the objects and the attributes of these objects that these methods return. When an instance of the appropriate session bean is created, WebSphere Application Server associates a context with the instance. The context contains information about the caller's principal ID, group membership list, and roles. This information is used to check the caller's authorization for each call.
The Javadoc contains authorization information for each of the methods.
4. Decide how to render the application.
The EJB APIs can be called locally or remotely.
5. Develop the application.
 - a. Access the EJB API.
 - b. Use the EJB API to interact with processes or tasks.
 - Query the data.
 - Work with the data.

Accessing the EJB APIs

The Enterprise JavaBeans (EJB) APIs are provided as two stateless session enterprise beans. Business process applications and task applications access the appropriate session enterprise bean through the home interface of the bean.

About this task

The `BusinessFlowManagerService` interface provides the methods for business process applications, and the `HumanTaskManagerService` interface provides the methods for task-based applications. The application can be any Java application, including another Enterprise JavaBeans (EJB) application.

Accessing the remote interface of the session bean

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

About this task

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

Procedure

1. Add a reference to the remote interface of the 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 `<install_root>/ProcessChoreographer/client/bpe137650.jar` file with the enterprise archive (EAR) file of your application.
- b. For task applications, package the `<install_root>/ProcessChoreographer/client/task137650.jar` file with the EAR file of your application.
- c. Set the **Classpath** 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.
- d. If you use complex data types in your business process or human task and your client does not run in an EJB application or a Web application, package the corresponding XSD or WSDL files with the EAR file of your application.

3. Locate the remote home interface of the session bean through the Java Naming and Directory Interface (JNDI).

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 remote 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();
```

Access to the session bean does not guarantee that the caller can perform all of the actions provided by the bean; the caller must also be authorized for these actions. When an instance of the session bean is created, a context is associated with the instance of the session bean. The context contains the caller's principal ID, group membership list, and indicates whether the caller has one of the Business Process Choreographer J2EE roles. The context is used to check the caller's authorization for each call, even when global security is not set. If global security is not set, the caller's principal ID has the value UNAUTHENTICATED.

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();
```

Tip: To prevent database lock conflicts, avoid running statements similar to the following in parallel:

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

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

The `getActivityInstance` method and other read operations set a read lock. In this example, a read lock on the activity instance is upgraded to an update lock on the activity instance. This can result in a database deadlock when these transactions are run in parallel.

Example

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 interface of the session bean

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

About this task

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

Procedure

1. Add a reference to the local interface of the 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. Locate the local home interface of the session bean through the Java Naming and Directory Interface (JNDI).

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 BusinessFlowManager bean

```

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

The local home interface of the 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 session bean.

The following example shows this step for a process application:

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

Access to the session bean does not guarantee that the caller can perform all of the actions provided by the bean; the caller must also be authorized for these actions. When an instance of the session bean is created, a context is associated with the instance of the session bean. The context contains the caller's principal ID, group membership list, and indicates whether the caller has one of the Business Process Choreographer J2EE roles. The context is used to check the caller's authorization for each call, even when global security is not set. If global security is not set, the caller's principal ID has the value UNAUTHENTICATED.

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();
```

Tip: To prevent database deadlocks, avoid running statements similar to the following in parallel:

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

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

The getActivityInstance method and other read operations set a read lock. In this example, a read lock on the activity instance is upgraded to an update lock on the activity instance. This can result in a database deadlock when these transactions are run in parallel

Example

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

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

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

Querying business-process and task-related objects

The client applications work with 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.

About this task

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 method or the queryAll method of the service API to retrieve stored information about business processes and tasks.

The query method can be called by all users, and it returns the properties of the objects for which work items exist. The queryAll method can be called only by users who have one of the following J2EE roles: BPESystemAdministrator, TaskSystemAdministrator, BPESystemMonitor, or TaskSystemMonitor. This method returns the properties of all the objects that are stored in the database.

All API queries are mapped to SQL queries. The form of the resulting SQL query depends on the following aspects:

- Whether the query was invoked by someone with one of the J2EE roles.
- The objects that are queried. Predefined database views are provided for you to query the object properties.
- The insertion of a from clause, join conditions, and user-specific conditions for access control.

You can include both custom properties and variable properties in queries. If you include several custom properties or variable properties in your query, this results

in self-joins on the corresponding database table. Depending on your database system, these query() calls might have performance implications.

You can also store queries in the Business Process Choreographer database using the createStoredQuery method. You provide the query criteria when you define the stored query. The criteria are applied dynamically when the stored query runs, that is, the data is assembled at runtime. If the stored query contains parameters, these are also resolved when the query runs.

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.

Syntax of the API query method:

The syntax of the Business Process Choreographer API queries is similar to SQL queries. A query can include a select clause, a where clause, an order-by clause, a skip-tuples parameter, a threshold parameter and a time-zone parameter.

The syntax of the query depends on the object type. The following table shows the syntax for each of the different object types.

Table 2.

Object	Syntax
Process template	ProcessTemplateData[] queryProcessTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);
Task template	TaskTemplate[] queryTaskTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);
Business-process and task-related data	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);

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 similar to the syntax of an SQL SELECT clause; use commas to separate parts of the clause. Each part of the clause must specify a column from one of the predefined views. The columns must be fully specified by view name and column name. The columns returned in the QueryResultSet object appear in the same order as the columns specified in the select clause.

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

To select the properties of multiple name-value pairs, such as custom properties and properties of variables that can be queried, add a one-digit counter to the view name. This counter can take the values 1 through 9.

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.
- "QUERY_PROPERTY1.STRING_VALUE, QUERY_PROPERTY2.INT_VALUE"
Gets the values of the properties of variables that can be queried. These parts 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.

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 similar to the syntax of an SQL WHERE clause. You do not need to explicitly add an SQL from clause or join predicates to the API 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
The LIKE operation supports the wildcard characters that are defined for the queried database.
- Set operation: IN

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`.
- If the value of the property in the comparison statement contains single quotation marks (`'`), double the quotation marks, for example, `"TASK_CPROP.STRING_VALUE='d''automatisation'"`.
- Refer to properties of multiple 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' AND
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.

You can specify a list of columns from the views by which the result is sorted. These columns must be fully qualified by the name of the view and the column. It is a best practice to specify columns that are in the select clause.

The order-by clause syntax is similar to the syntax of an SQL order-by clause; use commas to separate each part of the clause. You can also specify ASC to sort the

columns in ascending order, and DESC to sort the columns in descending order. If you do not want to sort the query result set, you must specify null for the order-by clause.

Sort criteria are applied on the server, that is, the locale of the server is used for sorting. If you specify more than one column, the query result set is ordered by the values of the first column, then by the values of the second column, and so on. You cannot specify the columns in the order-by clause by position as you can with an SQL query.

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 from the beginning of the query result set 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, for example, to retrieve the first 20 items, then the next 20 items, and so on.

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.

Because query result sets in production scenarios can contain thousands or even millions of items, it is a best practice to always specify a threshold. The threshold parameter can be useful, for example, in a graphical user interface where only a small number of items should be displayed at one time. 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')",
 (String)null,
 (Integer)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')",
              (String)null, (Integer)null, (TimeZone)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.

Parameters in stored queries:

A stored query is a query that is stored in the database and identified by a name. The qualifying tuples are assembled dynamically when the query is run. To make stored queries reusable, you can use parameters in the query definition that are resolved at runtime.

For example, you have defined custom properties to store customer names. You can define queries to return the tasks that are associated with a particular customer, ACME Co. To query this information, the where clause in your query might look similar to the following example:

```
String whereClause =
    "TASK.STATE = TASK.STATE.STATE_READY
    AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
    AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = 'ACME Co.'";
```

To make this query reusable so that you can also search for the customer, BCME Ltd, you can use parameters for the values of the custom property. If you add parameters to the task query, it might look similar to the following example:

```
String whereClause =
    "TASK.STATE = TASK.STATE.STATE_READY
    AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
    AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = '@param1'";
```

The @param1 parameter is resolved at runtime from the list of parameters that is passed to the query method. The following rules apply to the use of parameters in queries:

- Parameters can only be used in the where clause.
- Parameters are strings.

- Parameters are replaced at runtime using string replacement. If you need special characters you must specify these in the where clause or passed-in at runtime as part of the parameter.
- Parameter names consist of the string @param concatenated with an integer number. The lowest number is 1, which points to the first item in the list of parameters that is passed to the query API at runtime.
- A parameter can be used multiple times within a where clause; all occurrences of the parameter are replaced by the same value.

Related tasks

“Managing stored queries” on page 58

Stored queries provide a way to save queries that are run often. The stored query can be either a query that is available to all users (public query), or a query that belongs to a specific user (private query).

Query results:

A query result set contains the results of a query.

The elements of the result set are properties of the objects that satisfy the where clause given by the caller, and that the caller is authorized to see. You can read elements in a relative fashion using the API 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
OID	getOID
Timestamp	getTimestamp getString getTimestampAsLong
Integer	getInteger getShort getLong getString getBoolean
Boolean	getBoolean getShort getInteger getLong getString
byte[]	getBinary

Example:

The following query is run:

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

User-specific access conditions:

User-specific access conditions are added when the SQL SELECT statement is generated from the API query. These conditions guarantee that only those objects are returned to the caller that satisfy the condition specified by the caller and to which the caller is authorized.

The access condition that is added depends on whether the user is a system administrator.

Queries invoked by users who are not system administrators

The generated SQL WHERE clause combines the API where clause with an access control condition that is specific to the user. The query retrieves only those objects that the user is authorized to access, that is, only those objects for which the user has a work item. A work item represents the assignment of a user or user group to an authorization role of a business object, such as a task or process. If, for example, the user, John Smith, is a member of the potential owners role of a given task, a work item object exists that represents this relationship.

For example, if a user, who is not a system administrator, queries tasks, the following access condition is added to the WHERE clause if group work items are not enabled:

```

FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'user'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true )

```

So, if John Smith wants to get a list of tasks for which he is the potential owner, the API where clause might look as follows:

```
"WORK_ITEM.REASON == WORK_ITEM.REASON.REASON_POTENTIAL_OWNER"
```

This API where clause results in the following access condition in the SQL statement:

```

FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'JohnSmith'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
AND WI.REASON = 1

```

This also means that if John Smith wants to see the activities and tasks for which he is a process reader or a process administrator and for which he does not have a work item, then a property from the PROCESS_INSTANCE view must be added to the select, where, or order-by clause of the query, for example, PROCESS_INSTANCE.PIID.

If group work items are enabled, an additional access condition is added to the WHERE clause that allows a user to access objects that the group has access to.

Queries invoked by system administrators

System administrators can invoke the query method to retrieve objects that have associated work items. In this case, a join with the WORK_ITEM view is added to the generated SQL query, but no access control condition for the WORK_ITEM.OWNER_ID.

In this case, the SQL query for tasks contains the following:

```

FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID

```

queryAll queries

This type of query can be invoked only by system administrators or system monitors. Neither conditions for access control nor a join to the WORK_ITEM view are added. This type of query returns all of the data for all of the objects.

Examples of the query and queryAll methods:

These examples show the syntax of various typical API queries and the associated SQL statements that are generated when the query is processed.

Example: querying tasks in the ready state:

This example shows how to use the query method to retrieve tasks that the logged-on user can work with.

John Smith wants to get a list of the tasks that have been assigned to him. For a user to be able to work on a task, the task must be in the ready state. The logged-on user must also have a potential owner work item for the task. The following code snippet shows the query method call for this query:

```

query( "DISTINCT TASK.TKIID",
      "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
      AND " +
      "TASK.STATE = TASK.STATE.STATE_READY AND " +
      "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
      (String)null, (String)null, (Integer)null, (TimeZone)null )

```

The following actions are taken when the SQL SELECT statement is generated:

- A condition for access control is added to the where clause. This example assumes that group work items are not enabled.
- Constants, such as TASK.STATE.STATE_READY, are replaced by their numeric values.
- A FROM clause and join conditions are added.

The following code snippet shows the SQL statement that is generated from the API query:

```

SELECT DISTINCT TASK.TKIID
FROM   TASK TA, WORK_ITEM WI,
WHERE  WI.OBJECT_ID = TA.TKIID
AND    TA.KIND IN ( 101, 105 )
AND    TA.STATE = 2
AND    WI.REASON = 1
AND    ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )

```

To restrict the API query to tasks for a specific process, for example, sampleProcess, the query looks as follows:

```

query( "DISTINCT TASK.TKIID",
      "PROCESS_TEMPLATE.NAME = 'sampleProcess' AND "+
      "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
      AND " +
      "TASK.STATE = TASK.STATE.STATE_READY AND " +
      "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
      (String)null, (String)null, (Integer)null, (TimeZone)null )

```

Example: querying tasks in the claimed state:

This example shows how to use the query method to retrieve tasks that the logged-on user has claimed.

The user, John Smith, wants to search for tasks that he has claimed and are still in the claimed state. The condition that specifies "claimed by John Smith" is TASK.OWNER = 'JohnSmith'. The following code snippet shows the query method call for the query:

```

query( "DISTINCT TASK.TKIID",
      "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
      "TASK.OWNER = 'JohnSmith'",
      (String)null, (String)null, (Integer)null, (TimeZone)null )

```

The following code snippet shows the SQL statement that is generated from the API query:

```

SELECT DISTINCT TASK.TKIID
FROM   TASK TA, WORK_ITEM WI,
WHERE  WI.OBJECT_ID = TA.TKIID
AND    TA.STATE = 8
AND    TA.OWNER = 'JohnSmith'
AND    ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )

```

When a task is claimed, work items are created for the owner of the task. So, an alternative way of forming the query for John Smith's claimed tasks is to add the following condition to the query instead of using `TASK.OWNER = 'JohnSmith'`:

```
WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER
```

The query then looks like the following code snippet:

```
query( "DISTINCT TASK.TKIID",
      "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
      "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER",
      (String)null, (String)null, (Integer)null, (TimeZone)null )
```

The following actions are taken when the SQL SELECT statement is generated:

- A condition for access control is added to the where clause. This example assumes that group work items are not enabled.
- Constants, such as `TASK.STATE.STATE_READY`, are replaced by their numeric values.
- A FROM clause and join conditions are added.

The following code snippet shows the SQL statement that is generated from the API query:

```
SELECT DISTINCT TASK.TKIID
FROM   TASK TA, WORK_ITEM WI,
WHERE  WI.OBJECT_ID = TA.TKIID
AND    TA.STATE = 8
AND    WI.REASON = 4
AND    ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

John is about to go on vacation so his team lead, Anne Grant, wants to check on his current work load. Anne has system administrator rights. The query she invokes is the same as the one John invoked. However, the SQL statement that is generated is different because Anne is an administrator. The following code snippet shows the generated SQL statement:

```
SELECT DISTINCT TASK.TKIID
FROM   TASK TA, WORK_ITEM WI,
WHERE  TA.TKIID = WI.OBJECT_ID =
AND    TA.STATE = 8
AND    TA.OWNER = 'JohnSmith')
```

Because Anne is an administrator, an access control condition is not added to the WHERE clause.

Example: querying escalations:

This example shows how to use the query method to retrieve escalations for the logged-on user.

When a task is escalated, and escalation receiver work item is created. The user, Mary Jones wants to see a list of tasks that have been escalated to her. The following code snippet shows the query method call for the query:

```
query( "DISTINCT ESCALATION.ESIID, ESCALATION.TKIID",
      "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_ESCALATION_RECEIVER",
      (String)null, (String)null, (Integer)null, (TimeZone)null )
```

The following actions are taken when the SQL SELECT statement is generated:

- A condition for access control is added to the where clause. This example assumes that group work items are not enabled.

- Constants, such as TASK.STATE.STATE_READY, are replaced by their numeric values.
- A FROM clause and join conditions are added.

The following code snippet shows the SQL statement that is generated from the API query:

```
SELECT DISTINCT ESCALATION.ESIID, ESCALATION.TKIID
FROM   ESCALATION ESC, WORK_ITEM WI
WHERE  ESC.ESIID = WI.OBJECT_ID
AND    WI.REASON = 10
AND
( WI.OWNER_ID = 'MaryJones' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

Example: using the queryAll method:

This example shows how to use the queryAll method to retrieve all of the activities that belong to a process template.

The queryAll method is available only to users with system administrator or system monitor rights. The following code snippet shows the queryAll method call for the query to retrieve all of the activities that belong to the process template, sampleProcess:

```
queryAll( "DISTINCT ACTIVITY.AIID",
         "PROCESS_TEMPLATE.NAME = 'sampleProcess'",
         (String)null, (String)null, (Integer)null, (TimeZone)null )
```

The following code snippet shows the SQL query that is generated from the API query:

```
SELECT DISTINCT ACTIVITY.AIID
FROM   ACTIVITY AI, PROCESS_TEMPLATE PT
WHERE  AI.PTID = PT.PTID
AND    PT.NAME = 'sampleProcess'
```

Because the call is invoked by an administrator, an access control condition is not added to the generated SQL statement. A join with the WORK_ITEM view is also not added. This means that the query retrieves all of the activities for the process template, including those activities without work items.

Example: including query properties in a query:

This example shows how to use the query method to retrieve tasks that belong to a business process. The process has query properties defined for it that you want to include in the search.

For example, you want to search for all of the human tasks in the ready state that belong to a business process. The process has a query property, **customerID**, with the value CID_12345, and a namespace. The following code snippet shows the query method call for the query:

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
        PROCESS_INSTANCE.NAME",
        " QUERY_PROPERTY.NAME = 'customerID' AND " +
        " QUERY_PROPERTY.STRING_VALUE = 'CID_12345' AND " +
        " QUERY_PROPERTY.NAMESPACE =
        'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
        " TASK.KIND IN
        ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
        " TASK.STATE = TASK.STATE.STATE_READY ",
        (String)null, (String)null, (Integer)null, (TimeZone)null );
```


If you now want to add a second query property to the query, for example, **Priority**, with a given namespace, the query method call for the query looks as follows:

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
        PROCESS_INSTANCE.NAME",
        " QUERY_PROPERTY1.NAME = 'customerID' AND " +
        " QUERY_PROPERTY1.STRING_VALUE = 'CID_12345' AND " +
        " QUERY_PROPERTY1.NAMESPACE =
          'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
        " QUERY_PROPERTY2.NAME = 'Priority' AND " +
        " QUERY_PROPERTY2.NAMESPACE =
          'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
        " TASK.KIND IN
          ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
        " TASK.STATE = TASK.STATE.STATE_READY ",
        (String)null, (String)null, (Integer)null, (TimeZone)null );
```

If you add more than one query property to the query, you must number each of the properties that you add as shown in the code snippet. However, querying custom properties affects performance; performance decreases with the number of custom properties in the query.

Example: including custom properties in a query:

This example shows how to use the query method to retrieve tasks that have custom properties.

For example, you want to search for all of the human tasks in the ready state that have a custom property, **customerID**, with the value CID_12345. The following code snippet shows the query method call for the query:

```
query ( " DISTINCT TASK.TKIID ",
        " TASK_CPROP.NAME = 'customerID' AND " +
        " TASK_CPROP.STRING_VALUE = 'CID_12345' AND " +
        " TASK.KIND IN
          ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
        " TASK.STATE = TASK.STATE.STATE_READY ",
        (String)null, (String)null, (Integer)null, (TimeZone)null );
```

If you now want to retrieve the tasks and their custom properties, the query method call for the query looks as follows:

```
query ( " DISTINCT TASK.TKIID, TASK_CPROP.NAME, TASK_CPROP.STRING_VALUE",
        " TASK.KIND IN
          ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
        " TASK.STATE = TASK.STATE.STATE_READY ",
        (String)null, (String)null, (Integer)null, (TimeZone)null );
```

The SQL statement that is generated from this API query is shown in the following code snippet:

```
SELECT DISTINCT TA.TKIID , TACP.NAME , TACP.STRING_VALUE
FROM TASK TA LEFT JOIN TASK_CPROP TACP ON (TA.TKIID = TACP.TKIID),
WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND TA.KIND IN ( 101, 105 )
AND TA.STATE = 2
AND (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID IS NULL AND WI.EVERYBODY = 1 )
```

This SQL statement contains an outer join between the TASK view and the TASK_CPROP view. This means that tasks that satisfy the WHERE clause are retrieved even if they do not have any custom properties.

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 the predefined 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.

Note: The views might contain columns that are not described. These columns are for internal use only.

ACTIVITY view:

Use this predefined database view for queries on activities.

Table 3. 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 (21) KIND_RECEIVE (23) KIND_REPLY (24) KIND_THROW (25) KIND_RETHROW (46) KIND_TERMINATE (26) KIND_WAIT (27) KIND_COMPENSATE (29) KIND_SEQUENCE (30) KIND_EMPTY (3) KIND_SWITCH (32) KIND_WHILE (34) KIND_PICK (36) KIND_FLOW (38) KIND_SCOPE (40) KIND_SCRIPT (42) KIND_STAFF (43) KIND_ASSIGN (44) KIND_CUSTOM (45) KIND_FOR_EACH_PARALLEL (49) KIND_FOR_EACH_SERIAL (47)
COMPLETED	Timestamp	The time the activity is completed.
ACTIVATED	Timestamp	The time the activity is activated.
FIRST_ACTIVATED	Timestamp	The time at which the activity was activated for the first time.
STARTED	Timestamp	The time the activity is started.

Table 3. Columns in the ACTIVITY view (continued)

Column name	Type	Comments
STATE	Integer	The state of the activity. Possible values are: STATE_INACTIVE (1) STATE_READY (2) STATE_RUNNING (3) STATE_PROCESSING_UNDO (14) STATE_SKIPPED (4) STATE_FINISHED (5) STATE_FAILED (6) STATE_TERMINATED (7) STATE_CLAIMED (8) STATE_TERMINATING (9) STATE_FAILING (10) STATE_WAITING (11) STATE_EXPIRED (12) STATE_STOPPED (13)
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. Possible values are: TRUE The activity is business relevant. You can view the activity status in Business Process Choreographer Explorer. FALSE The activity is not business relevant.
EXPIRES	Timestamp	The date and time when the activity is due to expire. If the activity has expired, the date and time when this event occurred.

ACTIVITY_ATTRIBUTE view:

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

Table 4. 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 5. 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 instance that is 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 6. 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 policy of the component. If this attribute is set to TRUE, the work item assignments for the task can be modified. This means that work items can be created, deleted, or transferred.

Table 6. Columns in the APPLICATION_COMP view (continued)

Column name	Type	Comments
SUPPORT_FOLLOW_ON	Boolean	The default follow-on task policy of the component. If this attribute is set to TRUE, follow-on tasks can be created for tasks. This value can be overwritten by a definition in the task template or the task.
SUPPORT_SUB_TASK	Boolean	The default subtask policy of the component. If this attribute is set to TRUE, subtasks can be created for tasks. This value can be overwritten by a definition in the task template or the task.

ESCALATION view:

Use this predefined database view to query data for escalations.

Table 7. 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 (1) Creates a work item for each escalation receiver. ACTION_SEND_EMAIL (2) Sends an e-mail to each escalation receiver. ACTION_CREATE_EVENT (3) 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 (2) Specifies that the human or participating task is ready to be claimed. ACTIVATION_STATE_RUNNING (3) Specifies that the originating task is started and running. ACTIVATION_STATE_CLAIMED (8) Specifies that the task is claimed. ACTIVATION_STATE_WAITING_FOR_SUBTASK (20) Specifies that the task is waiting for the completion of subtasks.
ACTIVATION_TIME	Timestamp	The time when the escalation is activated.

Table 7. Columns in the ESCALATION view (continued)

Column name	Type	Comments
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 (8) Specifies that the task is claimed. AT_LEAST_EXPECTED_STATE_ENDED (20) Specifies that the task is in a final state (FINISHED, FAILED, TERMINATED or EXPIRED). AT_LEAST_EXPECTED_STATE_SUBTASKS_COMPLETED (21) 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.
INCREASE_PRIORITY	Integer	Indicates how the task priority will be increased. Possible values are: INCREASE_PRIORITY_NO (1) The task priority is not increased. INCREASE_PRIORITY_ONCE (2) The task priority is increased once by one. INCREASE_PRIORITY_REPEATED (3) 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 (1) STATE_WAITING (2) STATE_ESCALATED (3) STATE_SUPERFLUOUS (4)
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 8. 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 9. 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.

ESC_TEMPL view:

Use this predefined database view to query data for escalation templates.

Table 10. Columns in the ESC_TEMPL view

Column name	Type	Comments
ESTID	String	The ID of the escalation template.
ACTION	Integer	The action triggered by the escalation. Possible values are: ACTION_CREATE_WORK_ITEM (1) Creates a work item for each escalation receiver. ACTION_SEND_EMAIL (2) Sends an e-mail to each escalation receiver. ACTION_CREATE_EVENT (3) 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 (2) Specifies that the human or participating task is ready to be claimed. ACTIVATION_STATE_RUNNING (3) Specifies that the originating task is started and running. ACTIVATION_STATE_CLAIMED (8) Specifies that the task is claimed. ACTIVATION_STATE_WAITING_FOR_SUBTASK (20) Specifies that the task is waiting for the completion of subtasks.

Table 10. Columns in the ESC_TEMPL view (continued)

Column name	Type	Comments
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 (8) Specifies that the task is claimed. AT_LEAST_EXPECTED_STATE_ENDED (20) Specifies that the task is in a final state (FINISHED, FAILED, TERMINATED or EXPIRED). AT_LEAST_EXPECTED_STATE_SUBTASKS_COMPLETED (21) Specifies that all of the subtasks of the task are complete.
CONTAINMENT_CTX_ID	String	If the escalation template belongs to an inline task template, the containment context is the process template. If the escalation template context belongs to a stand-alone task template, the containment context is the task template.
FIRST_ESTID	String	The ID of the first escalation template in a chain of escalation templates.
INCREASE_PRIORITY	Integer	Indicates how the task priority will be increased. Possible values are: INCREASE_PRIORITY_NO (1) The task priority is not increased. INCREASE_PRIORITY_ONCE (2) The task priority is increased once by one. INCREASE_PRIORITY_REPEATED (3) The task priority is increased by one each time the escalation repeats.
NAME	String	The name of the escalation template.
PREVIOUS_ESTID	String	The ID of the previous escalation template in a chain of escalation templates.
TKTID	String	The task template ID to which the escalation template belongs.

ESC_TEMPL_CPROP view:

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

Table 11. Columns in the ESC_TEMPL_CPROP view

Column name	Type	Comments
ESTID	String	The ID of the escalation template.
NAME	String	The name of the property.
TKTID	String	The task template ID to which the escalation template belongs.

Table 11. Columns in the ESC_TEMPL_CPROP view (continued)

Column name	Type	Comments
DATA_TYPE	String	The type of the class for non-string custom properties.
VALUE	String	The value for custom properties of type String.

ESC_TEMPL_DESC view:

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

Table 12. Columns in the ESC_TEMPL_DESC view

Column name	Type	Comments
ESTID	String	The ID of the escalation template.
LOCALE	String	The name of the locale associated with the description or display name.
TKTID	String	The task template ID to which the escalation template belongs.
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 13. 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 14. 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.

Table 14. Columns in the *PROCESS_INSTANCE* view (continued)

Column name	Type	Comments
STATE	Integer	The state of the process instance. Possible values are: STATE_READY (1) STATE_RUNNING (2) STATE_FINISHED (3) STATE_COMPENSATING (4) STATE_INDOUBT (10) STATE_FAILED (5) STATE_TERMINATED (6) STATE_COMPENSATED (7) STATE_COMPENSATION_FAILED (12) STATE_TERMINATING (8) STATE_FAILING (9) STATE_SUSPENDED (11)
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_PIID	ID	The ID of the parent process instance.
PARENT_NAME	String	The name of the parent process instance.
TOP_LEVEL_PIID	ID	The process instance ID of the top-level process instance. If there is no top-level process instance, this is the process instance ID of the current 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.
TEMPLATE_DESCR	String	Description of the associated process template.
RESUMES	Timestamp	The time when the process instance is to be resumed automatically.

PROCESS_TEMPLATE view:

Use this predefined database view for queries on process templates.

Table 15. 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.

Table 15. Columns in the *PROCESS_TEMPLATE* view (continued)

Column name	Type	Comments
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 (1) STATE_STOPPED (2)
EXECUTION_MODE	Integer	Specifies how process instances that are derived from this process template can be run. Possible values are: EXECUTION_MODE_MICROFLOW (1) EXECUTION_MODE_LONG_RUNNING (2)
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 (2) COMP_SPHERE_SUPPORTS (4)
DISPLAY_NAME	String	The descriptive name of the process.

QUERY_PROPERTY view:

Use this predefined database view for queries on process-level variables.

Table 16. Columns in the *QUERY_PROPERTY* view

Column name	Type	Comments
PIID	ID	The process instance ID.
VARIABLE_NAME	String	The name of the process-level variable.
NAME	String	The name of the query property.
NAMESPACE	String	The namespace of the query property.
GENERIC_VALUE	String	A string representation for property types that do not map to one of the defined types: STRING_VALUE, NUMBER_VALUE, DECIMAL_VALUE, or TIMESTAMP_VALUE.
STRING_VALUE	String	If a property type is mapped to a string type, this is the value of the string.

Table 16. Columns in the QUERY_PROPERTY view (continued)

Column name	Type	Comments
NUMBER_VALUE	Integer	If a property type is mapped to an integer type, this is the value of the integer.
DECIMAL_VALUE	Decimal	If a property type is mapped to a floating point type, this is the value of the decimal.
TIMESTAMP_VALUE	Timestamp	If a property type is mapped to a timestamp type, this is the value of the timestamp.

TASK view:

Use this predefined database view for queries on task objects.

Table 17. 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.

Table 17. Columns in the TASK view (continued)

Column name	Type	Comments
FOLLOW_ON_TKIID	ID	The ID of the instance of the follow-on task.
HIERARCHY_POSITION	Integer	Possible values are: HIERARCHY_POSITION_TOP_TASK (0) The top-level task in the task hierarchy. HIERARCHY_POSITION_SUB_TASK (1) The task is a subtask in the task hierarchy. HIERARCHY_POSITION_FOLLOW_ON_TASK (2) The task is a follow-on task in the task hierarchy.
IS_AD_HOC	Boolean	Indicates whether this task was created dynamically at runtime or from a task template.
IS_ESCALATED	Boolean	Indicates whether an escalation of this task has occurred.
IS_INLINE	Boolean	Indicates whether the task is an inline task in a business process.
IS_WAIT_FOR_SUB_TK	Boolean	Indicates whether the parent task is waiting for a subtask to reach an end state.
KIND	Integer	The kind of task. Possible values are: KIND_HUMAN (101) States that the task is a <i>collaboration task</i> that is created and processed by a human. KIND_WPC_STAFF_ACTIVITY (102) States that the task is a human task that is a staff activity of a WebSphere Business Integration Server Foundation, version 5 business process. KIND_ORIGINATING (103) States that the task is an <i>invocation task</i> that supports person-to-computer interactions, which enables people to create, initiate, and start services. KIND_PARTICIPATING (105) States that the task is a <i>to-do task</i> that supports computer-to-person interactions, which enable a person to implement a service. KIND_ADMINISTRATIVE (106) States that the task is an administration 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.

Table 17. Columns in the TASK view (continued)

Column name	Type	Comments
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.
RESUMES	Timestamp	The time when the task is to be resumed automatically.
STARTED	Timestamp	The time when the task was started (STATE_RUNNING, STATE_CLAIMED).
STARTER	String	The principal ID of the task starter.
STATE	Integer	The state of the task. Possible values are: STATE_READY (2) States that the task is ready to be claimed. STATE_RUNNING (3) States that the task is started and running. STATE_FINISHED (5) States that the task finished successfully. STATE_FAILED (6) States that the task did not finish successfully. STATE_TERMINATED (7) States that the task has been terminated because of an external or internal request. STATE_CLAIMED (8) States that the task is claimed. STATE_EXPIRED (12) States that the task ended because it exceeded its specified duration. STATE_FORWARDED (101) 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 through creating, deleting, or transferring work items.
SUPPORT_FOLLOW_ON	Boolean	Indicates whether this task supports the creation of follow-on tasks.
SUPPORT_SUB_TASK	Boolean	Indicates whether this task supports the creation of subtasks.
SUSPENDED	Boolean	Indicates whether the task is suspended.
TKTID	ID	The task template ID.
TOP_TKIID	ID	The top parent task instance ID if this is a 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 18. 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 19. 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 20. 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.

Table 20. Columns in the TASK_TEMPL view (continued)

Column name	Type	Comments
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.
DEFINITION_NAME	String	The name of the task template definition in the Task Execution Language (TEL) file.
DEFINITION_NS	String	The namespace of the task template definition in the TEL file.
IS_AD_HOC	Boolean	Indicates whether this task template was created dynamically at runtime or when the task was deployed as part of an EAR file.
IS_INLINE	Boolean	Indicates whether this task template is modeled as a task within a business process.
KIND	Integer	The kind of tasks that are derived from this task template. Possible values are: KIND_HUMAN (101) States that the task is a <i>collaboration task</i> that is created and processed by a human. KIND_ORIGINATING (103) States that the task is an <i>invocation task</i> that supports person-to-computer interactions, which enables people to create, initiate, and start services. KIND_PARTICIPATING (105) States that the task is a <i>to-do task</i> that supports computer-to-person interactions, which enable a person to implement a service. KIND_ADMINISTRATIVE (106) States that the task is an administration 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.

Table 20. Columns in the TASK_TEMPL view (continued)

Column name	Type	Comments
STATE	Integer	The state of the task template. Possible values are: STATE_STARTED (1) Specifies that the task template is available for creating task instances. STATE_STOPPED (2) 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 using creation, deletion, or transfer of work items.
SUPPORT_FOLLOW_ON	Boolean	Indicates whether the task template supports the creation of follow-on tasks.
SUPPORT_SUB_TASK	Boolean	Indicates whether the task template supports the creation of subtasks.
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 21. 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.
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 22. 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 23. 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.
OBJECT_TYPE	Integer	<p>The type of the associated object. Possible values are:</p> <p>OBJECT_TYPE_ACTIVITY (1) Specifies that the work item was created for an activity.</p> <p>OBJECT_TYPE_PROCESS_INSTANCE (3) Specifies that the work item was created for a process instance.</p> <p>OBJECT_TYPE_TASK_INSTANCE (5) Specifies that the work item was created for a task.</p> <p>OBJECT_TYPE_TASK_TEMPLATE (6) Specifies that the work item was created for a task template.</p> <p>OBJECT_TYPE_ESCALATION_INSTANCE (7) Specifies that the work item was created for an escalation instance.</p> <p>OBJECT_TYPE_APPLICATION_COMPONENT (9) Specifies that the work item was created for an application component.</p>
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.

Table 23. Columns in the WORK_ITEM view (continued)

Column name	Type	Comments
REASON	Integer	The reason for the assignment of the work item. Possible values are: REASON_POTENTIAL_STARTER (5) REASON_POTENTIAL_INSTANCE_CREATOR (11) REASON_POTENTIAL_STARTER (1) REASON_EDITOR (2) REASON_READER (3) REASON_ORIGINATOR (9) REASON_OWNER (4) REASON_STARTER (6) REASON_ESCALATION_RECEIVER (10) REASON_ADMINISTRATOR (7)
CREATION_TIME	Timestamp	The date and time when the work item was created.

Filtering data using variables in queries

A query result returns the objects that match the query criteria. You might want to filter these results on the values of variables.

About this task

You can define variables that are used by a process at runtime in its process model. For these variables, you declare which parts can be queried.

For example, John Smith, calls his insurance company's service number to find out the progress of his insurance claim for his damaged car. The claims administrator uses the customer ID to the find the claim.

Procedure

1. Optional: List the properties of the variables in a process that can be queried.

Use the process template ID to identify the process. You can skip this step if you know which variables can be queried.

```
List variableProperties = process.getQueryProperties(ptid);
for (int i = 0; i < variableProperties.size(); i++)
{
    QueryProperty queryData = (QueryProperty)variableProperties.get(i);
    String variableName = queryData.getVariableName();
    String name          = queryData.getName();
    int mappedType      = queryData.getMappedType();
    ...
}
```

2. List the process instances with variables that match the filter criteria.

For this process, the customer ID is modeled as part of the variable customerClaim that can be queried. You can therefore use the customer's ID to find the claim.

```
QueryResultSet result = process.query
("PROCESS_INSTANCE.NAME, QUERY_PROPERTY.STRING_VALUE",
"QUERY_PROPERTY.VARIABLE_NAME = 'customerClaim' AND " +
"QUERY_PROPERTY.NAME = 'customerID' AND " +
"QUERY_PROPERTY.STRING_VALUE like 'Smith%'",
(String)null, (Integer)null,
(Integer)null, (TimeZone)null );
```

This action returns a query result set that contains the process instance names and the values of the customer IDs for customers whose IDs start with Smith.

Managing stored queries

Stored queries provide a way to save queries that are run often. The stored query can be either a query that is available to all users (public query), or a query that belongs to a specific user (private query).

About this task

A stored query is a query that is stored in the database and identified by a name. A private and a public stored query can have the same name; private stored queries from different owners can also have the same name.

You can have stored queries for business process objects, task objects, or a combination of these two object types.

Related concepts

“Parameters in stored queries” on page 32

A stored query is a query that is stored in the database and identified by a name. The qualifying tuples are assembled dynamically when the query is run. To make stored queries reusable, you can use parameters in the query definition that are resolved at runtime.

Managing public stored queries:

Public stored queries are created by the system administrator. These queries are available to all users.

About this task

As the system administrator, you can create, view, and delete public stored queries. If you do not specify a user ID in the API call, it is assumed that the stored query is a public stored query.

Procedure

1. Create a public stored query.

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

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

The result of the stored query is 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. List the names of the available public stored queries.

The following code snippet shows how to limit the list of returned queries to just the public queries.

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

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

A stored private query can have the same name as a stored public query. If these names are the same, the private stored query is returned. The following code snippet shows how to return only the public query with the specified name. If you want to run this query for task-based objects, specify `StoredQuery` as the returned object type instead of `StoredQueryData`.

```
StoredQueryData storedQuery = process.getStoredQuery
    (StoredQueryData.KIND_PUBLIC, "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

5. Delete a public stored query.

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

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

Managing private stored queries for other users:

Private queries can be created by any user. These queries are available only to the owner of the query and the system administrator.

About this task

As the system administrator, you can manage private stored queries that belong to a specific user.

Procedure

1. Create a private stored query for the user ID Smith.

For example, the following code snippet creates a stored query for process instances and saves it with the name `CustomerOrdersStartingWithA` for the user ID Smith.

```
process.createStoredQuery("Smith", "CustomerOrdersStartingWithA",
    "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
    "PROCESS_INSTANCE.NAME LIKE 'A%'",
    "PROCESS_INSTANCE.NAME",
    (Integer)null, (TimeZone)null,
    (List)null, (String)null);
```

The result of the stored query is 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
    ("Smith", "CustomerOrdersStartingWithA",
    (Integer)null, (Integer)null, (List)null);
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. Get a list of the names of the private queries that belong to a specific user.

For example, the following code snippet shows how to get a list of private queries that belongs to the user Smith.

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

4. View the details of a specific query.

The following code snippet shows how to view the details of the `CustomerOrdersStartingWithA` query that is owned by the user Smith.

```

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

```

5. Delete a private stored query.

The following code snippet shows how to delete a private query that is owned by the user Smith.

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

Working with your private stored queries:

If you are not a system administrator, you can create, run, and delete your own private stored queries. You can also use the public stored queries that the system administrator created.

Procedure

1. Create a private stored query.

For example, the following code snippet creates a stored query for process instances and saves it with a specific name. If a user ID is not specified, it is assumed that the stored query is a private stored query for the logged-on user.

```
process.createStoredQuery("CustomerOrdersStartingWithA",
    "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
    "PROCESS_INSTANCE.NAME LIKE 'A%'",
    "PROCESS_INSTANCE.NAME",
    (Integer)null, (TimeZone)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. Get a list of the names of the stored queries that the logged-on user can access.

The following code snippet shows how to get both the public and the private stored queries that the user can access.

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

4. View the details of a specific query.

The following code snippet shows how to view the details of the CustomerOrdersStartingWithA query that is owned by the user Smith.

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

5. Delete a private stored query.

The following code snippet shows how to delete a private stored query.

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

Developing applications for business processes

A business process is a set of business-related activities that are invoked in a specific sequence to achieve a business goal. Examples are provided that show how you might develop applications for typical actions on processes.

About this task

A business process can be either a microflow or a long-running process:

- Microflows are short running business processes that are 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. The use of certain constructs in a process causes interruptions in the process flow, for example, invoking a human task, invoking a service using an synchronous binding, or using timer-driven activities.

Parallel branches of the process are usually navigated asynchronously, that is, activities in parallel branches are executed concurrently. Depending on the type and the transaction setting of the activity, an activity can be run in its own transaction.

Required roles for actions on process instances

Access to the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on a process. The caller must be logged on to the client application with a role that is authorized to perform the action.

The following table shows the actions on a process instance that a specific role can take.

Action	Caller's principal role		
	Reader	Starter	Administrator
createMessage	x	x	x
createWorkItem			x
delete			x
deleteWorkItem			x
forceTerminate			x
getActiveEventHandlers	x		x
getActivityInstance	x		x
getAllActivities	x		x
getAllWorkItems	x		x
getClientUISettings	x	x	x
getCustomProperties	x	x	x
getCustomProperty	x	x	x
getCustomPropertyName	x	x	x
getFaultMessage	x	x	x
getInputClientUISettings	x	x	x
getInputMessage	x	x	x
getOutputClientUISettings	x	x	x
getOutputMessage	x	x	x
getProcessInstance	x	x	x

Action	Caller's principal role		
	Reader	Starter	Administrator
getVariable	x	x	x
getWaitingActivities	x	x	x
getWorkItems	x		x
restart			x
resume			x
setCustomProperty		x	x
setVariable			x
suspend			x
transferWorkItem			x

Required roles for actions on business-process activities

Access to the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on an activity. The caller must be logged on to the client application with a role that is authorized to perform the action.

The following table shows the actions on an activity instance that a specific role can take.

Action	Caller's principal role				
	Reader	Editor	Potential owner	Owner	Administrator
cancelClaim				x	x
claim			x		x
complete				x	x
createMessage	x	x	x	x	x
createWorkItem					x
deleteWorkItem					x
forceComplete					x
forceRetry					x
getActivityInstance	x	x	x	x	x
getAllWorkItems	x	x	x	x	x
getClientUISettings	x	x	x	x	x
getCustomProperties	x	x	x	x	x
getCustomProperty	x	x	x	x	x
getCustomPropertyNames	x	x	x	x	x
getFaultMessage	x	x	x	x	x
getFaultNames	x	x	x	x	x
getInputMessage	x	x	x	x	x
getOutputMessage	x	x	x	x	x
getVariable	x	x	x	x	x
getVariableNames	x	x	x	x	x
getInputVariableNames	x	x	x	x	x
getOutputVariableNames	x	x	x	x	x

Action	Caller's principal role				
	Reader	Editor	Potential owner	Owner	Administrator
getWorkItems	x	x	x	x	x
setCustomProperty		x		x	x
setFaultMessage		x		x	x
setOutputMessage		x		x	x
setVariable					x
transferWorkItem				x To potential owners or administrators only	x

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. Various actions can be taken on the process instance to manage its life cycle.

About this task

Examples are provided that show how you might develop applications for the following typical life-cycle actions on processes.

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.

About this task

Examples are provided that show how you might develop applications for typical scenarios for starting microflows and long-running processes.

Running a microflow that contains a unique starting service:

A microflow can be started by a receive activity or a pick activity. The starting service is unique if the microflow starts with a receive activity or when the pick activity has only one onMessage definition.

About this task

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 passing the process template name as a parameter in the call.

If the microflow is a one-way operation, use the sendMessage method to run the process. This method is not covered in this example.

Procedure

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 =
    PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
"PROCESS_TEMPLATE.NAME",
    new Integer(50),
    (TimeZone)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.getInputMessageType());
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. The starting service is not unique if the microflow starts with a pick activity that has multiple *onMessage* definitions.

About this task

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 passing the ID of the starting service in the call.

If the microflow is a one-way operation, use the *sendMessage* method to run the process. This method is not covered in this example.

Procedure

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 =
    PROCESS_TEMPLATE.EXECUTION_MODE.EXECUTION_MODE_MICROFLOW",
"PROCESS_TEMPLATE.NAME",
    new Integer(50),
    (TimeZone)null);

```

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

2. Determine the starting service to be called.

This example uses the first template that is found.

```

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.getInputMessageType());

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.

Procedure

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 =
PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
"PROCESS_TEMPLATE.NAME",
new Integer(50),
(TimeZone)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.getInputMessageType());
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 human task, 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 or pick activities, or a pick activity that has multiple onMessage definitions, then you must identify the service to be called.

Procedure

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 =
PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
"PROCESS_TEMPLATE.NAME",
new Integer(50),
(TimeZone)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.getInputMessageType());

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.sendMessage(activity.getServiceTemplateID(),
    activity.getActivityTemplateID(),
    input);
```

This action creates an instance 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 human task, receive, or pick activities, work items are created for the potential owners.

Suspending and resuming a business process:

You can suspend long-running, top-level process instance while it is running and resume it again to complete it.

Before you begin

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.

About this task

You might want to suspend a process instance, 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. You might also want to suspend a process to fix a problem that is causing the process instance to fail, and then resume it again when the problem is fixed.

Procedure

1. Get the running process, CustomerOrder, 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. The process instance is put into the suspended state. Subprocesses with the autonomy attribute set to `child` are also suspended if they are in the `running`, `failing`, `terminating`, or `compensating` state. Inline tasks that are associated with this process instance are also suspended; stand-alone tasks associated with this process instance are not suspended.

In this state, activities that are started can still be finished but no new activities are activated, for example, a human task activity in the `claimed` state can be completed.

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.

Before you begin

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

About this task

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.

If the process has more than one receive activity or pick activity (also known as a receive choice activity) that can create the process instance, all of the messages that belong to these activities are used to restart the process instance. If any of these activities implement a request-response operation, the response is sent again when the associated reply activity is navigated.

Procedure

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 top-level process instance that is known to be in an unrecoverable

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.

Procedure

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 of the process is run as if a fault had occurred on the top-level scope. If you terminate the process instance without compensation, the process instance is terminated immediately without waiting for activities, to-do tasks, or inline invocation tasks to end normally.

Applications that are started by the process and standalone tasks that are related to the process are not terminated by the force terminate request. If these applications are to be terminated, you must add statements to your process application that explicitly terminate the applications started by the process.

Deleting process instances:

Completed process 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. However, stored process instance data does not only impact disk space and performance but also prevents process instances that use the same correlation set values from being created. Therefore, you should regularly delete process instance data from the database.

About this task

To delete a process instance, you need process administrator rights and the process instance must be a top-level process instance.

The following example shows how to delete all of the finished process instances.

Procedure

1. List the process instances that are finished.

```
QueryResultSet result =  
    process.query("DISTINCT PROCESS_INSTANCE.PIID",  
                "PROCESS_INSTANCE.STATE =  
                PROCESS_INSTANCE.STATE.STATE_FINISHED",  
                (String)null, (Integer)null, (TimeZone)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 and its inline tasks from the database.

Processing human task activities

Human task activities in business processes are assigned to various people in your organization through work items. When a process is started, work items are created for the potential owners.

About this task

When a human task activity is activated, both an activity instance and an associated to-do task are created. Handling of the human task activity and the work item management is delegated to Human Task Manager. Any state change of the activity instance is reflected in the task instance and vice versa.

A potential owner claims the activity. This person is responsible for providing the relevant information and completing the activity.

Procedure

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

```
QueryResultSet result =
    process.query("ACTIVITY.AIID",
        "ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
        ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
        WORK_ITEM.REASON =
            WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
        (String)null, (Integer)null, (TimeZone)null);
```

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

2. Claim the activity to be worked on:

```
if (result.size() > 0)
{
    result.first();
    AIID aaid = (AIID) result.getOID(1);
    ClientObjectWrapper input = process.claim(aaid);
    DataObject activityInput = null ;
    if ( input.getObject() != null && input.getObject() instanceof DataObject )
    {
        activityInput = (DataObject)input.getObject();
        // read the values
        ...
    }
}
```

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

3. When work on the activity is finished, complete the activity. The activity can be completed either successfully or with a fault message. If the activity is successful, an output message is passed. If the activity is unsuccessful, the activity is put into the failed or stopped state and a fault message is passed.

You must create the appropriate messages for these actions. When you create the message, you must specify the message type name so that the message definition is contained.

- a. To complete the activity successfully, create an output message.

```
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);
}

//complete the activity
process.complete(aiid, output);
```

This action sets an output message that contains the order number.

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

```
//retrieve the faults modeled for the human task activity
List faultNames = process.getFaultNames(aiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
    process.createMessage(aiid, 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);
}

process.complete(aiid, (String) faultNames.get(0), myFault);
```

This action sets the activity in either the failed or the stopped state. If the **continueOnError** parameter for the activity in the process model is set to true, the activity is put into the failed state and the navigation continues. If the **continueOnError** parameter is set to false and the fault is not caught on the surrounding scope, the activity is put into the stopped state. In this state the activity can be repaired using force complete or force retry.

Processing a single person workflow

Some workflows are performed by only one person, for example, ordering books from an online bookstore. This type of workflow has no parallel paths. The `completeAndClaimSuccessor` API supports the processing of this type of workflow.

About this task

In an online bookstore, the purchaser completes a sequence of actions to order a book. This sequence of actions can be implemented as a series of human task activities (to-do tasks). If the purchaser decides to order several books, this is equivalent to claiming the next human task activity. This type of workflow is also known as *page flow* because user interface definitions are associated with the activities to control the flow of the dialogs in the user interface.

The `completeAndClaimSuccessor` API completes a human task activity and claims the next one in the same process instance for the logged-on person. It returns information about the next claimed activity, including the input message to be worked on. Because the next activity is made available within the same transaction of the activity that completed, the transactional behavior of all the human task activities in the process model must be set to `participates`.

Compare this example with the example that uses both the Business Flow Manager API and the Human Task Manager API.

Procedure

1. Claim the first activity in the sequence of activities.

```
//
//Query the list of activities that can be claimed by the logged-on user
//
QueryResultSet result =
    process.query("ACTIVITY.AIID",
        "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
        ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
        ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
        WORK_ITEM.REASON =
            WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
        (String)null, (Integer)null, (TimeZone)null);

...
//
//Claim the first activity
//
if (result.size() > 0)
{
    result.first();
    AIID aaid = (AIID) result.getOID(1);
    ClientObjectWrapper input = process.claim(aaid);
    DataObject activityInput = null ;
    if ( input.getObject() != null && input.getObject() instanceof DataObject )
    {
        activityInput = (DataObject)input.getObject();
        // read the values
        ...
    }
}
```

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

2. When work on the activity is finished, complete the activity, and claim the next activity.

To complete the activity, an output message is passed. When you create the output message, you must specify the message type name so that the message definition is contained.

```
ActivityInstanceData activity = process.getActivityInstance(aaid);
ClientObjectWrapper output =
    process.createMessage(aaid, 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);
}

//complete the activity and claim the next one
CompleteAndClaimSuccessorResult successor =
    process.completeAndClaimSuccessor(aaid, output);
```

This action sets an output message that contains the order number and claims the next activity in the sequence. If `AutoClaim` is set for successor activities and if there are multiple paths that can be followed, all of the successor activities are claimed and a random activity is returned as the next activity. If there are no more successor activities that can be assigned to this user, `Null` is returned.

If the process contains parallel paths that can be followed and these paths contain human task activities for which the logged-on user is a potential owner of more than one of these activities, a random activity is claimed automatically and returned as the next activity.

3. Work on the next activity.

```
String name = successor.getActivityName();

ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject() !=
    null && nextInput.getObject() instanceof DataObject )
{
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
}

aaid = successor.getAIID();
```

4. Continue with step 2 to complete the activity.

Related tasks

“Processing a single person workflow that includes human tasks” on page 100
Some workflows are performed by only one person, for example, ordering books from an online bookstore. This example shows how to implement the sequence of actions for ordering the book as a series of human task activities (to-do tasks). Both the Business Flow Manager and the Human Task Manager APIs are used to process the workflow.

Sending a message to a waiting activity

You can use inbound message activities (receive activities, `onMessage` in pick activities, `onEvent` in event handlers) 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.

About this task

You can use originating tasks to send the message to the activity.

Procedure

1. List the activity service templates that are waiting for a message from the logged-on user in a process instance with a specific process instance ID.

```
ActivityServiceTemplateData[] services = process.getWaitingActivities(piid);
```

2. Send a message to the first waiting service.

It is assumed that the first service is the one that you want serve. The caller must be a potential starter of the activity that receives the message, or an administrator of the process instance.

```
VTID vtid = services[0].getServiceTemplateID();
ATID atid = services[0].getActivityTemplateID();
String inputType = services[0].getInputMessageType();

// create a message for the service to be called
ClientObjectWrapper message =
    process.createMessage(vtid,atid,inputMessageType);
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.

About this task

You can invoke an event handler any number of times as long as the corresponding scope is running. 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.

Procedure

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.

This example uses the first event handler that is found.

```

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.getInputMessageType());

    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 process can expose Web services operations that are modeled as Web Services Description Language (WSDL) one-way or request-response operations. The results of long-running processes with one-way interfaces cannot be retrieved using the `getOutputMessage` method, because the process has no output. However, you can query the contents of variables, instead.

About this task

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.

Procedure

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",
     (String)null, (Integer)null, (TimeZone)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");
    }
}

```

Repairing activities

A long-running process can contain activities that are also long running. These activities might encounter uncaught errors and go into the stopped state. An activity in the running state might also appear to be not responding. In both of these cases, a process administrator can act on the activity in a number of ways so that the navigation of the process can continue.

About this task

The Business Process Choreographer API provides the `forceRetry` and `forceComplete` methods for repairing activities. Examples are provided that show how you might add repair actions for activities to your applications.

Forcing the completion of an activity: About this task

Activities in long-running processes can sometimes encounter faults. If these faults are not caught by a fault handler 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.

You can also force the completion of activities in the running state if, for example, an activity is not responding.

Additional requirements exist for certain types of activities.

Human task 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.

Procedure

1. List the stopped activities in the stopped state.

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

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

2. Complete the activity, for example, a stopped human task activity.

In this example, an output message is passed.

```
if (result.size() > 0)
{
    result.first();
    AIID aaid = (AIID) result.getOID(1);
    ActivityInstanceData activity = process.getActivityInstance(aaid);
    ClientObjectWrapper output =
        process.createMessage(aaid, activity.getOutputMessageType());
    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(aaid, output, continueOnError);
}
```

This action completes the activity. If an error occurs, the **continueOnError** parameter determines the action to be taken if a fault is provided with the forceComplete request.

In the example, **continueOnError** is true. This value means that if a fault is provided, 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: About this task

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.

Procedure

1. List the stopped activities.

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

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

2. Retry the execution of the activity, for example, a stopped human task activity.

```
if (result.size() > 0)
{
    result.first();
    AIID aaid = (AIID) result.getOID(1);
    ActivityInstanceData activity = process.getActivityInstance(aaid);
    ClientObjectWrapper input =
        process.createMessage(aaid, activity.getOutputMessageType());
    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(aaid, input, continueOnError);
}
```

This action retries the activity. If an error occurs, the **continueOnError** parameter determines the action to be taken if an error occurs during processing of the forceRetry request.

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 a fault handler on the process level is run before the process state ends in the failed state.

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, `sendMessage()`. The execution of the process instance is driven automatically by the server.

Table 24. API methods for process templates

Method	Description
<code>getProcessTemplate</code>	Retrieves the specified process template.
<code>queryProcessTemplates</code>	Retrieves process templates that are stored in the database.

Process instances

The following API methods are related to starting process instances.

Table 25. API methods are related to starting process instances

Method	Description
<code>call</code>	Creates and runs a microflow.
<code>callWithReplyContext</code>	Creates and runs a microflow with a unique starting service or a long-running process with a unique starting service from the specified process template. The call waits asynchronously for the result.
<code>callWithUISettings</code>	Creates and runs a microflow and returns the output message and the client user interface (UI) settings.
<code>initiate</code>	Creates a process instance and initiates processing of the process instance. Use this method for long-running processes. You can also use this method for microflows that you want to fire and forget.
<code>sendMessage</code>	Sends the specified message to the specified activity service and process instance. If a process instance with the same correlation set values does not exist, it is created. The process can have either unique or non-unique starting services.
<code>getStartActivities</code>	Returns information about the activities that can start a process instance from the specified process template.
<code>getActivityServiceTemplate</code>	Retrieves the specified activity service template.

Table 26. 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, or terminated state.
forceTerminate	Terminates the specified top-level process instance, its subprocesses with child autonomy, and its running, claimed, or waiting activities.
delete	Deletes the specified top-level process instance and its subprocesses with child autonomy.
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 `continueOnError` 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 `continueOnError` 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 working with and repairing activities.

Table 27. 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.
completeAndClaimSuccessor	Completes the activity instance and claims the next one in the same process instance for the logged-on person.
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 28. API methods for variables and custom properties

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

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 typical applications for human tasks are provided.

About this task

For more information on the Human Task Manager API, see the Javadoc in the `com.ibm.task.api` package.

Starting an invocation task that invokes a synchronous interface

An invocation task is associated with a Service Component Architecture (SCA) component. When the task is started, it invokes the SCA component. Start an invocation task synchronously only if the associated SCA component can be called synchronously.

About this task

Such an SCA component can, for example, be implemented as a microflow or as a simple Java class.

This scenario creates an instance of a task template and passes some customer data. The task remains in the running state until the two-way operation returns. The result of the task, `OrderNo`, is returned to the caller.

Procedure

1. Optional: List the task templates to find the name of the invocation 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),
 (TimeZone)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 the task synchronously.

For a task to run synchronously, it must be a two-way operation. The example uses the `createAndCallTask` method to create and run the task.

```

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 invocation task that invokes an asynchronous interface

An invocation task is associated with a Service Component Architecture (SCA) component. When the task is started, it invokes the SCA component. Start an invocation task asynchronously only if the associated SCA component can be called asynchronously.

About this task

Such an SCA component can, for example, be implemented as a long-running process or a one-way operation.

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

Procedure

1. Optional: List the task templates to find the name of the invocation 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),
 (TimeZone)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.

The example uses the `createAndStartTask` method to create and run the task.

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

Creating and starting a task instance

This scenario shows how to create an instance of a task template that defines a collaboration task (also known as a *human task* in the API) and start the task instance.

Procedure

1. Optional: List the task templates to find the name of the collaboration 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.HUMAN",
 "TASK_TEMPL.NAME",
 new Integer(50),
 (TimeZone)null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted task 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 and start the collaboration task; a reply handler is not specified in this example.

The example uses the `createAndStartTask` method to create and start the task.

```
TKIID tkiid = task.createAndStartTask( template.getName(),
                                       template.getNamespace(),
                                       input,
                                       (ReplyHandlerWrapper)null);
```

Work items are created for the people concerned with the task instance. For example, a potential owner can claim the new task instance.

4. Claim the task instance.

```
ClientObjectWrapper input2 = task.claim(tkiid);
DataObject taskInput = null ;
if ( input2.getObject() != null && input2.getObject() instanceof DataObject )
{
    taskInput = (DataObject)input2.getObject();
    // read the values
    ...
}
```

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

Processing to-do tasks or collaboration tasks

To-do tasks (also known as *participating tasks* in the API) or collaboration tasks (also known as *human tasks* in the API) are assigned to various people in your organization through work items. To-do tasks and their associated work items are created, for example, when a process navigates to a human task activity.

About this task

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.

Procedure

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",
              (String)null, (Integer)null, (TimeZone)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.createFaultMessage(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.

Suspending and resuming a task instance

You can suspend collaboration task instances (also known as *human tasks* in the API) or to-do task instances (also known as *participating tasks* in the API).

Before you begin

The task instance can be in the ready or claimed state. It can be escalated. The caller must be the owner, originator, or administrator of the task instance.

About this task

You can suspend a task instance while it is running. You might want to do this, for example, so that you can gather information that is needed to complete the task. When the information is available, you can resume the task instance.

Procedure

1. Get a list of tasks that are claimed by the logged-on user.

```

QueryResultSet result = task.query("DISTINCT TASK.TKIID",
    "TASK.STATE = TASK.STATE.STATE_CLAIMED",
    (String)null,
    (Integer)null,
    (TimeZone)null);

```

This action returns a query result set that contains a list of the tasks that are claimed by the logged-on user.

2. Suspend the task instance.

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

This action suspends the specified task instance. The task instance is put into the suspended state.

3. Resume the process instance.

```
task.resume( tkiid );
```

This action puts the task instance into the state it had before it was suspended.

Analyzing the results of a task

A to-do task (also known as a *participating* task in the API) or a collaboration task (also known as a *human task* in the API) runs asynchronously. If a reply handler is specified when the task starts, the output message is automatically returned when the task completes. If a reply handler is not specified, the message must be retrieved explicitly.

About this task

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.

Procedure

Analyze the results of the task.

The example shows how to check the order number of a successfully completed task.

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.NAME = 'CustomerOrder' AND
                                   TASK.STATE = TASK.STATE.STATE_FINISHED",
                                   (String)null, (Integer)null, (TimeZone)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. Because the task instance is terminated immediately, you should terminate a task instance only in exceptional situations.

Procedure

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. This example shows how to delete all of the task instances that are finished and are not automatically deleted.

Procedure

1. List the task instances that are finished.

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

Releasing a claimed task

When a potential owner claims a task, this person is responsible for completing the task. However, sometimes the claimed task must be released so that another potential owner can claim it.

About this task

Sometimes it is necessary for someone with administrator rights to release a claimed task. This situation might occur, for example, when a task must be completed but the owner of the task is absent. The owner of the task can also release a claimed task.

Procedure

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'",
              (String)null, (Integer)null, (TimeZone)null);
```

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

2. Release the claimed task.

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


This action returns the task to the ready state so that it can be claimed by one of the other potential owners. Any output or fault data that is set by the original owner is kept.

Managing work items

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.

About this task

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 human task activity instance, a process instance, or a task instance. The reasons are derived from the role that the user has for the object. An object can have multiple work items because a user can have different roles in association with the object, and a work item is created for each of these roles. For example, a to-do task instance can have an administrator, reader, editor, and owner work item at the same time.

The actions that can be taken to manage work items depend on the role that the user has, for example, an administrator can create, delete and transfer work items, but the task owner can transfer work items only.

- 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'",
                                   (String)null, (Integer)null,
                                   (TimeZone)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'",
                                   (String)null, (Integer)null,
                                   (TimeZone)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',
(String)null, (Integer)null, (TimeZone)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.

Creating task templates and task instances at runtime

You usually use a modeling tool, such as WebSphere Integration Developer to build task templates. You then install the task templates in WebSphere Process Server and create instances from these templates, for example, using Business Process Choreographer Explorer. However, you can also create human or participating task instances or templates at runtime.

About this task

You might want to do this, for example, when the task definition is not available when the application is deployed, the tasks that are part of a workflow are not yet known, or you need a task to cover some ad-hoc collaboration between a group of people.

You can model ad-hoc To-do or Collaboration tasks by creating instances of the `com.ibm.task.api.TaskModel` class, and using them to either create a reusable task template, or directly create a run-once task instance. To create an instance of the `TaskModel` class, a set of factory methods is available in the `com.ibm.task.api.ClientTaskFactory` factory class. Modeling human tasks at runtime is based on the Eclipse Modeling Framework (EMF).

Procedure

1. Create an `org.eclipse.emf.ecore.resource.ResourceSet` using the `createResourceSet` factory method.
2. Optional: If you intend to use complex message types, you can either define them using the `org.eclipse.xsd.XSDFactory` that you can get using the factory method `getXSDFactory()`, or directly import an existing XML schema using the `loadXSDSchema` factory method .

To make the complex types available to the WebSphere Process Server, deploy them as part of an enterprise application.

3. Create or import a Web Services Definition Language (WSDL) definition of the type `javax.wsdl.Definition`.

You can create a new WSDL definition using the `createWSDLDefinition` method. Then you can add it a port type and operation. You can also directly import an existing WSDL definition using the `loadWSDLDefinition` factory method.

4. Create the task definition using the `createTTask` factory method.

If you want to add or manipulate more complex task elements, you can use the `com.ibm.wbit.tel.TaskFactory` class that you can retrieve using the `getTaskFactory` factory method .

5. Create the task model using the createTaskModel factory method, and pass it the resource bundle that you created in the step 1 and which aggregates all other artifacts you created in the meantime.
6. Optional: Validate the model using the TaskModel validate method.

Results

Use one of the Human Task Manager EJB API create methods that have a **TaskModel** parameter to either create a reusable task template, or a run-once task instance.

Creating runtime tasks that use simple Java types:

This example creates a runtime task that uses only simple Java types in its interface, for example, a String object.

About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

Procedure

1. Access the ClientTaskFactory and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. Create the WSDL definition and add the descriptions of your operations.

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
    ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );
```

```
// create a port type
PortType portType = factory.createPortType( definition, "doItPT" );
```

```
// create an operation; the input and output messages are of type String:
// a fault message is not specified
Operation operation = factory.createOperation
    ( definition, portType, "doIt",
      new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
      new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
      (Map)null );
```

3. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```
TTask humanTask = factory.createTTask( resourceSet,
    TTaskKinds.HTASK_LITERAL,
    "TestTask",
    new UTCDate( "2005-01-01T00:00:00" ),
    "http://www.ibm.com/task/test/",
    portType,
    operation );
```

This step initializes the properties of the task model with default values.

4. Modify the properties of your human task model.

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );
```

```
// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();
```

```
// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. Create the runtime task instance or template.

Use the `HumanTaskManagerService` interface to create the task instance or the task template. Because the application uses simple Java types only, you do not need to specify an application name.

- The following snippet creates a task instance:

```
atask.createTask( taskModel, (String)null, "HTM" );
```
- The following snippet creates a task template:

```
task.createTaskTemplate( taskModel, (String)null );
```

Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

Creating runtime tasks that use complex types:

This example creates a runtime task that uses complex types in its interface. The complex types are already defined, that is, the local file system on the client has XSD files that contain the description of the complex types.

About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

Procedure

1. Access the `ClientTaskFactory` and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. Add the XSD definitions of your complex types to the resource set so that they are available when you define your operations.

The files are located relative to the location where the code is executed.

```
factory.loadXSDSchema( resourceSet, "InputB0.xsd" );
factory.loadXSDSchema( resourceSet, "OutputB0.xsd" );
```

3. Create the WSDL definition and add the descriptions of your operations.

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
    ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );
```

```

// create a port type
PortType portType = factory.createPortType( definition, "doItPT" );

// create an operation; the input message is an InputBO and
// the output message an OutputBO;
// a fault message is not specified
Operation operation = factory.createOperation
    ( definition, portType, "doIt",
      new QName( "http://Input", "InputBO" ),
      new QName( "http://Output", "OutputBO" ),
      (Map)null );

```

4. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```

TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );

```

This step initializes the properties of the task model with default values.

5. Modify the properties of your human task model.

```

// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);

```

6. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

7. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

8. Create the runtime task instance or template.

Use the HumanTaskManagerService interface to create the task instance or the task template. You must provide an application name that contains the data type definitions so that they can be accessed. The application must also contain a dummy task or process so that the application is loaded by Business Process Choreographer.

- The following snippet creates a task instance:

```
task.createTask( taskModel, "BOapplication", "HTM" );
```
- The following snippet creates a task template:

```
task.createTaskTemplate( taskModel, "BOapplication" );
```

Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

Creating runtime tasks that use an existing interface:

This example creates a runtime task that uses an interface that is already defined, that is, the local file system on the client has a file that contains the description of the interface.

About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

Procedure

1. Access the ClientTaskFactory and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. Access the WSDL definition and the descriptions of your operations.

The interface description is located relative to the location where the code is executed.

```
Definition definition = factory.loadWSDLDefinition(
    resourceSet, "interface.wsdl" );
PortType portType = definition.getPortType(
    new QName( definition.getTargetNamespace(), "doItPT" ) );
Operation operation = portType.getOperation(
    "doIt", (String)null, (String)null);
```

3. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```
TTask humanTask = factory.createTTask( resourceSet,
    TTaskKinds.HTASK_LITERAL,
    "TestTask",
    new UTCDate( "2005-01-01T00:00:00" ),
    "http://www.ibm.com/task/test/",
    portType,
    operation );
```

This step initializes the properties of the task model with default values.

4. Modify the properties of your human task model.

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. Create the runtime task instance or template.

Use the `HumanTaskManagerService` interface to create the task instance or the task template. You must provide an application name that contains the data type definitions so that they can be accessed. The application must also contain a dummy task or process so that the application is loaded by Business Process Choreographer.

- The following snippet creates a task instance:

```
task.createTask( taskModel, "B0application", "HTM" );
```

- The following snippet creates a task template:

```
task.createTaskTemplate( taskModel, "B0application" );
```

Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

Creating runtime tasks that use an interface from the calling application:

This example creates a runtime task that uses an interface that is part of the calling application. For example, the runtime task is created in a Java snippet of a business process and uses an interface from the process application.

About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

Procedure

1. Access the `ClientTaskFactory` and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
```

```
// specify the context class loader so that following resources are found
ResourceSet resourceSet = factory.createResourceSet
    ( Thread.currentThread().getContextClassLoader() );
```

2. Access the WSDL definition and the descriptions of your operations.

Specify the path within the containing package JAR file.

```
Definition definition = factory.loadWSDLDefinition( resourceSet,
    "com/ibm/workflow/metaflow/interface.wsdl" );
PortType portType = definition.getPortType(
    new QName( definition.getTargetNamespace(), "doItPT" ) );
Operation operation = portType.getOperation
    ("doIt", (String)null, (String)null);
```

3. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (`UTCDate`) is not required.

```
TTask humanTask = factory.createTTask( resourceSet,
    TTaskKinds.HTASK_LITERAL,
    "TestTask",
    new UTCDate( "2005-01-01T00:00:00" ),
    "http://www.ibm.com/task/test/",
    portType,
    operation );
```

This step initializes the properties of the task model with default values.

4. Modify the properties of your human task model.

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. Create the runtime task instance or template.

Use the HumanTaskManagerService interface to create the task instance or the task template. You must provide an application name that contains the data type definitions so that they can be accessed.

- The following snippet creates a task instance:
task.createTask(taskModel, "WorkflowApplication", "HTM");
- The following snippet creates a task template:
task.createTaskTemplate(taskModel, "WorkflowApplication");

Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

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 29. API methods for task templates

Method	Description
getTaskTemplate	Retrieves the specified task template.

Table 29. API methods for task templates (continued)

Method	Description
createAndCallTask	Creates and runs a task instance from the specified task template and waits synchronously 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. 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 30. API methods for task instances

Method	Description
getTask	Retrieves a task instance; the task instance can be in any state.
callTask	Starts an invocation task synchronously.
startTask	Starts a task that has already been created.
suspend	Suspends the collaboration or to-do task.
resume	Resumes the collaboration or to-do task.
terminate	Terminates the specified task instance. If an invocation task is terminated, this action has no impact on the invoked service.
delete	Deletes the specified task instance.
claim	Claims the task for processing.
update	Updates the task instance.
complete	Completes the task instance.
cancelClaim	Releases a claimed task instance so that it can be worked on by another potential owner.
createWorkItem	Creates a work item for the task instance.
transferWorkItem	Transfers the work item to a specified owner.
deleteWorkItem	Deletes the work item.

Escalations

The following methods are available to work with escalations.

Table 31. API methods for working with escalations

Method	Description
getEscalation	Retrieves the specified escalation instance.

Custom properties

Tasks, task templates, and escalations can all have custom properties. The interface provides a get and a set method to retrieve and set values for custom properties. You can also associate named properties with, and retrieve named properties from task instances. Custom property names and values must be of the `java.lang.String` type. The following methods are valid for tasks, task templates, and escalations.

Table 32. API methods for variables and custom properties

Method	Description
getCustomProperty	Retrieves the named custom property of the specified task instance.
getCustomProperties	Retrieves the 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:

The actions that can be carried out on a task depend on whether the task is a to-do task, a collaboration task, an invocation task, or an administration task.

You cannot use all of the actions provided by the `HumanTaskManager` interface for all kinds of tasks. The following table shows the actions that you can carry out on each kind of task.

Action	Kind of task			
	To-do task	Collaboration task	Invocation task	Administration task
callTask			X	
cancelClaim	X	X ¹		
claim	X	X ¹		
complete	X	X ¹		X
completeWithFollowOnTask ⁴	X	X ¹		
completeWithFollowOnTask ⁵		X ³	X ³	
createFaultMessage	X	X	X	X
createInputMessage	X	X	X	X
createOutputMessage	X	X	X	X
createWorkItem	X	X ¹	X	X

Action	Kind of task			
	To-do task	Collaboration task	Invocation task	Administration task
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
getFaultNames	X	X ¹		
getFaultMessage	X	X ¹	X	
getInputMessage	X	X ¹	X	
getOutputMessage	X	X ¹	X	
getUsersInRole	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		
startTask	X ¹	X ¹	X	X
startTaskAsSubtask ⁶	X	X ¹		
startTaskAsSubtask ⁷		X ³	X ³	
suspend	X	X ¹		
suspendWithCancelClaim	X	X ¹		
terminate	X ¹	X ¹	X ¹	
transferWorkItem	X	X ¹	X	X
update	X	X ¹	X	X

Notes:

1. For stand-alone tasks, ad-hoc tasks, and task templates only
2. For stand-alone tasks, inline tasks in business processes, and ad-hoc tasks only
3. For stand-alone tasks and ad-hoc tasks only
4. The tasks kinds that can have follow-on tasks
5. The task kinds that can be used as follow-on tasks
6. The tasks kinds that can have subtasks
7. The task kinds that can be used as subtasks

Developing applications for business processes and human tasks

People are involved in most business process scenarios. For example, a business process requires people interaction when the process is started or administered, or when human task activities are performed. To support these scenarios, you need to use both the Business Flow Manager API and the Human Task Manager API.

About this task

To involve people in business process scenarios, you can include the following task kinds in the business process:

- An inline invocation task (also known as an *originating task* in the API).
You can provide an invocation task for every receive activity, for each onMessage element of a pick activity, and for each onEvent element of an event handler. This task then controls who is authorized to start a process or communicate with a running process instance.
- An administration task.
You can provide an administration task to specify who is authorized to administer the process or perform administrative operations on the failed activities of the process.
- A to-do task (also known as a *participating task* in the API).
A to-do task implements a human task activity. This type of activity allows you to involve people in the process.

Human task activities in the business process represent the to-do tasks that people perform in the business process scenario. You can use both the Business Flow Manager API and the Human Task Manager API to realize these scenarios:

- The business process is the container for all of the activities that belong to the process, including the human task activities that are represented by to-do tasks. When a process instance is created, a unique object ID (PIID) is assigned.
- When a human task activity is activated during the execution of the process instance, an activity instance is created, which is identified by its unique object ID (AIID). At the same time, an inline to-do task instance is also created, which is identified by its object ID (TKIID). The relationship of the human task activity to the task instance is achieved by using the object IDs:
 - The to-do task ID of the activity instance is set to the TKIID of the associated to-do task.
 - The containment context ID of the task instance is set to the PIID of the process instance that contains the associated activity instance.
 - The parent context ID of the task instance is set to the AIID of the associated activity instance.
- The life cycles of all inline to-do task instances are managed by the process instance. When the process instance is deleted, then the task instances are also deleted. In other words, all of the tasks that have the containment context ID set to the PIID of the process instance are automatically deleted.

Determining the process templates or activities that can be started

A business process can be started by invoking the call, initiate, or sendMessage methods of the Business Flow Manager API. If the process has only one starting activity, you can use the method signature that requires a process template name as a parameter. If the process has more than one starting activity, you must explicitly identify the starting activity.

About this task

When a business process is modeled, the modeler can decide that only a subset of users can create a process instance from the process template. This is done by associating an inline invocation task to a starting activity of the process and by specifying authorization restrictions on that task. Only the people that are potential starters or administrators of the task are allowed to create an instance of the task, and thus an instance of the process template.

If an inline invocation task is not associated with the starting activity, or if authorization restrictions are not specified for the task, everybody can create a process instance using the starting activity.

A process can have more than one starting activity, each with different people queries for potential starters or administrators. This means that a user can be authorized to start a process using activity A but not using activity B.

Procedure

1. Use the Business Flow Manager API to create a list of the current versions of process templates that are in the started state.

Tip: The `queryProcessTemplates` method excludes only those process templates that are part of applications that are not yet started. So, if you use this method without filtering the results, the method returns all of the versions of the process templates regardless of which state they are in.

```
// current timestamp in UTC format, converted to yyyy-mm-ddThh:mm:ss
String now = (new UTCDate()).toXsdString();
String whereClause = "PROCESS_TEMPLATE.STATE =
PROCESS_TEMPLATE.STATE.STATE_STARTED AND
PROCESS_TEMPLATE.VALID_FROM =
(SELECT MAX(VALID_FROM) FROM PROCESS_TEMPLATE
WHERE NAME=PROCESS_TEMPLATE.NAME AND
VALID_FROM <= TS('" + now + "'))";

ProcessTemplateData[] processTemplates = process.queryProcessTemplates
( whereClause,
"PROCESS_TEMPLATE.NAME",
(Integer)null, (TimeZone)null);
```

The results are sorted by process template name.

2. Create the list of process templates and the list of starting activities for which the user is authorized.

The list of process templates contains those process templates that have a single starting activity. These activities are either not secured or the logged-on user is allowed to start them. Alternatively, you might want to gather the process templates that can be started by at least one of the starting activities.

Tip: A process administrator can also start a process instance. To get a complete list of templates, you also need to read the administration task template that is associated with the process template, and check whether the logged-on user is an administrator.

```
List authorizedProcessTemplates = new ArrayList();
List authorizedActivityServiceTemplates = new ArrayList();
```

3. Determine the starting activities for each of the process templates.

```
for( int i=0; i<processTemplates.length; i++ )
{
    ProcessTemplateData template = processTemplates[i];
    ActivityServiceTemplateData[] startActivities =
        process.getStartActivities(template.getID());
}
```

4. For each starting activity, retrieve the ID of the associated inline invocation task template.

```
for( int j=0; j<startActivities.length; j++ )
{
    ActivityServiceTemplateData activity = startActivities[j];
    TKTID tktid = activity.getTaskTemplateID();
}
```

- a. If an invocation task template does not exist, the process template is not secured by this starting activity.

In this case, everybody can create a process instance using this start activity.

```
boolean isAuthorized = false;
    if ( tktid == null )
    {
        isAuthorized = true;
        authorizedActivityServiceTemplates.add(activity);
    }
```

- b. If an invocation task template exists, use the Human Task Manager API to check the authorization for the logged-on user.

In the example, the logged-on user is Smith. The logged-on user must be a potential starter of the invocation task or an administrator.

```
if ( tktid != null )
{
    isAuthorized =
        task.isUserInRole
            (tkid, "Smith", WorkItem.REASON_POTENTIAL_STARTER) ||
        task.isUserInRole(tktid, "Smith", WorkItem.REASON_ADMINISTRATOR);

    if ( isAuthorized )
    {
        authorizedActivityServiceTemplates.add(activity);
    }
}
```

If the user has the specified role, or if people assignment criteria for the role are not specified, the `isUserInRole` method returns the value `true`.

5. Check whether the process can be started using only the process template name.

```
if ( isAuthorized && startActivities.length == 1 )
{
    authorizedProcessTemplates.add(template);
}
```

6. End the loops.

```
    } // end of loop for each activity service template
} // end of loop for each process template
```

Processing a single person workflow that includes human tasks

Some workflows are performed by only one person, for example, ordering books from an online bookstore. This example shows how to implement the sequence of actions for ordering the book as a series of human task activities (to-do tasks). Both the Business Flow Manager and the Human Task Manager APIs are used to process the workflow.

About this task

In an online bookstore, the purchaser completes a sequence of actions to order a book. This sequence of actions can be implemented as a series of human task activities (to-do tasks). If the purchaser decides to order several books, this is equivalent to claiming the next human task activity. Information about the sequence of tasks is maintained by Business Flow Manager, while the tasks themselves are maintained by Human Task Manager.

Compare this example with the example that uses only the Business Flow Manager API.

Procedure

1. Use the Business Flow Manager API to get the process instance that you want to work on.

In this example, an instance of the CustomerOrder process.

```
ProcessInstanceData processInstance =  
    process.getProcessInstance("CustomerOrder");  
String piid = processInstance.getID().toString();
```

2. Use the Human Task Manager API to query the ready to-do tasks (kind participating) that are part of the specified process instance.

Use the containment context ID of the task to specify the containing process instance. For a single person workflow, the query returns the to-do task that is associated with the first human task activity in the sequence of human task activities.

```
//  
// Query the list of to-do tasks that can be claimed by the logged-on user  
// for the specified process instance  
//  
QueryResultSet result =  
    task.query("DISTINCT TASK.TKIID",  
              "TASK.CONTAINMENT_CTX_ID = ID('" + piid + "') AND  
              TASK.STATE = TASK.STATE.STATE_READY AND  
              TASK.KIND = TASK.KIND.KIND_PARTICIPATING AND  
              WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",  
              (String)null, (Integer)null, (TimeZone)null);
```

3. Claim the to-do task that is returned.

```
if (result.size() > 0)  
{  
    result.first();  
    TKIID tkiid = (TKIID) result.getOID(1);  
    ClientObjectWrapper input = task.claim(tkiid);  
    DataObject activityInput = 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.

4. Determine the human task activity that is associated with the to-do task.

You can use one of the following methods to correlate activities to their tasks.

- The task.getActivityID method:

```
AIID aiid = task.getActivityID(tkiid);
```

- The parent context ID that is part of the task object:

```
AIID aiid = null;  
Task taskInstance = task.getTask(tkiid);  
  
OID oid = taskInstance.getParentContextID();  
if ( oid != null and oid instanceof AIID )  
{  
    aiid = (AIID)oid;  
}
```

5. When work on the task is finished, use the Business Flow Manager API to complete the task and its associated human task activity, and claim the next human task activity in the process instance.

To complete the human task activity, an output message is passed. When you create the output message, you must specify the message type name so that the message definition is contained.

```

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);
}

//complete the human task activity and its associated to-do task,
// and claim the next human task activity
CompleteAndClaimSuccessorResult successor =
    process.completeAndClaimSuccessor(aiid, output);

```

This action sets an output message that contains the order number and claims the next human task activity in the sequence. If `AutoClaim` is set for successor activities and if there are multiple paths that can be followed, all of the successor activities are claimed and a random activity is returned as the next activity. If there are no more successor activities that can be assigned to this user, `Null` is returned.

If the process contains parallel paths that can be followed and these paths contain human task activities for which the logged-on user is a potential owner of more than one of these activities, a random activity is claimed automatically and returned as the next activity.

6. Work on the next human task activity.

```

ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject() !=
    null && nextInput.getObject() instanceof DataObject )
{
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
}

aiid = successor.getAIID();

```

7. Continue with step 5 to complete the human task activity and to retrieve the next human task activity.

Related tasks

“Processing a single person workflow” on page 71

Some workflows are performed by only one person, for example, ordering books from an online bookstore. This type of workflow has no parallel paths. The `completeAndClaimSuccessor` API supports the processing of this type of workflow.

Handling exceptions and faults

A BPEL process might encounter a fault at different points in the process.

About this task

Business Process Execution Language (BPEL) faults originate from:

- Web service invocations (Web Services Description Language (WSDL) faults)
- Throw activities
- BPEL standard faults that are recognized by Business Process Choreographer

Mechanisms exist to handle these faults. Use one of the following mechanisms to handle faults that are generated by a process instance:

- Pass control to the corresponding fault handlers
- Compensate previous work in the process
- Stop the process and let someone repair the situation (force-retry, force-complete)

A BPEL process can also return faults to a caller of an operation provided by the process. You can model the fault in the process as a reply activity with a fault name and fault data. These faults are returned to the API caller as checked exceptions.

If a BPEL process does not handle a BPEL fault or if an API exception occurs, a runtime exception is returned to the API caller. An example for an API exception is 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

About this task

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

Procedure

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",
        (String)null, (Integer)null, (TimeZone)null);
```

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

2. Read the name of the fault.

```
if (result.size() > 0)
{
    result.first();
    AIID aaid = (AIID) result.getOID(1);
    ClientObjectWrapper faultMessage = process.getFaultMessage(aaid);
    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

About this task

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

Procedure

1. List the human task 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",
        (String)null, (Integer)null, (TimeZone)null);

```

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

2. Read the name of the fault.

```

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

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

```

Developing Web service API client applications

You can develop client applications that access business process applications and human task applications through Web services APIs.

About this task

Client applications can be developed in any Web service client environment, including Java Web services and Microsoft .NET.

Introduction: Web services

Web services are Web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with client applications. Web services allow the use of a language- and environment-neutral programming model.

Web services use the following core technologies:

- XML (Extensible Markup Language). XML solves the problem of data independence. You use it to describe data, and also to map that data into and out of any application or programming language

- WSDL (Web Services Description Language). You use this XML-based language to create a description of an underlying application. It is this description that turns an application into a Web service, by acting as the interface between the underlying application and other Web-enabled applications.
- SOAP (Simple Object Access Protocol). SOAP is the core communications protocol for the Web, and most Web services use this protocol to talk to each other.

Web service components and sequence of control

A number of client-side and server-side components participate in the sequence of control that represents a Web service request and response.

A typical sequence of control is as follows.

1. On the client side:
 - a. A client application (provided by the user) issues a request for a Web service.
 - b. A proxy client (also provided by the user, but which can be automatically generated using client-side utilities) wraps the service request in a SOAP request envelope.
 - c. The client-side development infrastructure forwards the request to a URL defined as the Web service's endpoint.
2. The network transmits the request to the Web service endpoint using HTTP or HTTPS.
3. On the server side:
 - a. The generic Web services API receives and decodes the request.
 - b. The request is either handled directly by the generic Business Flow Manager or Human Task Manager component, or forwarded to the specified business process or human task.
 - c. The returned data is wrapped in a SOAP response envelope.
4. The network transmits the response to the client-side environment using HTTP or HTTPS.
5. Back on the client side:
 - a. The client-side development infrastructure unwraps the SOAP response envelope.
 - b. The proxy client extracts the data from the SOAP response and passes it to the client application.
 - c. The client application processes the returned data as necessary.

Overview of the Web services APIs

Web services APIs allow you to develop client applications that use Web services to access business processes and human tasks running in the Business Process Choreographer environment.

The Business Process Choreographer Web services API provides two separate Web service interfaces (WSDL port types):

- The Business Flow Manager API. Allows client applications to interact with microflows and long-running processes, for example:
 - Create process templates and process instances
 - Claim existing processes
 - Query a process by its ID

Refer to “Developing applications for business processes” on page 61 for a complete list of possible actions.

- The Human Task Manager API. Allows client applications to:
 - Create and start tasks
 - Claim existing tasks
 - Complete tasks
 - Query a task by its ID
 - Query a collection of tasks.

Refer to “Developing applications for human tasks” on page 80 for a complete list of possible actions.

Client applications can use either or both of the Web service interfaces.

Example

The following is a possible outline for a client application that accesses the Human Task Manager Web services API to process a participating human task:

1. The client application issues a query Web service call to the WebSphere Process Server requesting a list of participating tasks to be worked on by a user.
2. The list of participating tasks is returned in a SOAP/HTTP response envelope.
3. The client application then issues a claim Web service call to claim one of the participating tasks.
4. The WebSphere Process Server returns the task’s input message.
5. The client application issues a complete Web service call to complete the task with an output or fault message.

Requirements for business processes and human tasks

Business processes and human tasks developed with the WebSphere Integration Developer to run on the Business Process Choreographer must conform to specific rules to be accessible through the Web services APIs.

The requirements are:

1. The interfaces of business processes and human tasks must be defined using the “document/literal wrapped” style defined in the Java API for XML-based RPC (JAX-RPC 1.1) specification. This is the default style for all business processes and human tasks developed with the WID.
2. Fault messages exposed by business processes and human tasks for Web service operations must comprise a single WSDL message part defined with an XML Schema element. For example:

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

Related information

 [Java API for XML based RPC \(JAX-RPC\) downloads page](#)

 [Which style of WSDL should I use?](#)

Developing client applications

The client application development process consists of a number of steps.

Procedure

1. Decide which Web services API your client application needs to use: the Business Flow Manager API, Human Task Manager API, or both.
2. Export the necessary files from the WebSphere Process Server environment. Alternatively, you can copy the files from the WebSphere Process Server client CD.
3. In your chosen client application development environment, generate a *proxy client* using the exported artifacts.
4. Optional: Generate *helper classes*. Helper classes are required if your client application interacts directly with concrete processes or tasks on the WebSphere server. They are not, however, necessary if your client application is only going to perform generic tasks such as issuing queries.
5. Develop the code for your client application.
6. Add any necessary security mechanisms to your client application.

Copying artifacts

A number of artifacts must be copied from the WebSphere environment to help in the creation of client applications.

There are two ways to obtain these artifacts:

- Publish and export them from the WebSphere Process Server environment.
- Copy files from the WebSphere Process Server client CD.

Publishing and exporting artifacts from the server environment

Before you can develop client applications to access the Web services APIs, you must publish and export a number of artifacts from the WebSphere server environment.

About this task

The artifacts to be exported are:

- Web Service Definition Language (WSDL) files describing the port types and operations that make up the Web services APIs.
- XML Schema Definition (XSD) files containing definitions of data types referenced by services and methods in the WSDL files.
- Additional WSDL and XSD files describing business objects. Business objects describe concrete business processes or human tasks running on the WebSphere server. These additional files are only required if your client application needs to interact directly with the concrete business processes or human tasks through the Web services APIs. They are not necessary if your client application is only going to perform generic tasks, such as issuing queries.

After these artifacts are published, you need to copy them to your client programming environment, where they are used to generate a proxy client and helper classes.

Specifying the Web service endpoint address:

The Web service endpoint address is the URL that a client application must specify to access the Web services APIs. The endpoint address is written into the WSDL file that you export to generate a proxy client for your client application.

About this task

The Web service endpoint address to use depends on your WebSphere server configuration:

- Scenario 1. A single WebSphere server. The WebSphere endpoint address to specify is the host name and port number of the server, for example **host1:9080**.
- Scenario 2. A WebSphere cluster composed of several servers. The WebSphere endpoint address to specify is the host name and port of the server that is hosting the Web services APIs, for example, **host2:9081**.
- Scenario 3. A Web server is used as a front end. The WebSphere endpoint address to specify is the host name and port of the Web server, for example: **host:80**.

By default, the Web service endpoint address takes the form *protocol://host:port/context_root/fixed_path*. Where:

- *protocol*. The communications protocol to be used between the client application and the WebSphere server. The default protocol is HTTP. You can instead choose to use the more secure HTTPS (HTTP over SSL) protocol. It is recommended to use HTTPS.
- *host:port*. The host name and port number used to access the machine that is hosting the Web services APIs. These values vary depending on the WebSphere server configuration; for example, whether your client application is to access the application directly or through a Web server front end.
- *context_root*. You are free to choose any value for the context root. The value you choose must, however, be unique within each WebSphere cell. The default value uses a "node_server/cluster" suffix that eliminates the risk of naming conflicts.
- *fixed_path* is either `/sca/com/ibm/bpe/api/BFMWS` (for the Business Flow Manager API) or `/sca/com/ibm/task/api/HTMWS` (for the Human Task Manager API) and cannot be modified.

The Web service endpoint address is initially specified when configuring the business process container or human task container:

Procedure

1. Log on to the administrative console with a user ID with administrator rights.
2. Choose **Applications** → **SCA modules**.

Note: You can also select **Applications** → **Enterprise applications** to display a list of all available enterprise applications.

3. Select **BPEContainer** (for the business process container) or **TaskContainer** (for the human task container) from the list of SCA modules or applications.
4. Choose **Provide HTTP endpoint URL information** from the list of **Additional properties**.
5. Select one of the default prefixes from the list, or enter a custom prefix. Use a prefix from the default prefix list if your client applications are to connect directly to the application server hosting the Web services API. Otherwise, specify a custom prefix.
6. Click **Apply** to copy the selected prefix to the SCA module.
7. Click **OK**. The URL information is saved to your workspace.

Results

You can view the current value in the administrative console (for example, for the business process container: **Enterprise Applications** → **BPEContainer** → **View Deployment Descriptor**).

In the exported WSDL file, the `location` attribute of the `soap:address` element contains the specified Web services endpoint address. For example:

```
<wsdl:service name="BFMWSservice">
  <wsdl:port name="BFMWSport" binding="this:BFMWSbinding">
    <soap:address location=
      "https://myserver:9080/WebServicesAPIs/sca/com/ibm/bpe/api/BFMWS"/>
  </wsdl:port>
</wsdl:service>
```

Related concepts

“Adding security (Java Web services)” on page 120

You must secure Web service communications by implementing security mechanisms in your client application.

Related tasks

“Adding security (.NET)” on page 128

You can secure Web service communications by integrating security mechanisms into your client application.

Publishing WSDL files:

A Web Service Definition Language (WSDL) file contains a detailed description of all the operations available with a Web services API. Separate WSDL files are available for the Business Flow Manager and Human Task Manager Web services APIs. You must first publish these WSDL files then copy them from the WebSphere environment to your development environment, where they are used to generate a proxy client.

Before you begin

Before publishing the WSDL files, be sure to specify the correct Web services endpoint address. This is the URL that your client application uses to access the Web services APIs.

About this task

You only need to publish WSDL files once.

Note: If you have the WebSphere Process Server client CD, you can copy the files directly from there to your client programming environment instead.

Publishing the business process WSDL:

Use the administrative console to publish the WSDL file.

Procedure

1. Log on to the administrative console with a user ID with administrator rights.
2. Select **Applications** → **SCA modules**

Note: You can also select **Applications** → **Enterprise applications** to display a list of all available enterprise applications.

3. Choose the **BPEContainer** application from the list of SCA modules or applications.
4. Select **Publish WSDL files** from the list of **Additional properties**

5. Click on the zip file in the list.
6. On the File Download window that appears, click **Save**.
7. Browse to a local folder and click **Save**.

Results

The exported zip file is named BPEContainer_WSDLFiles.zip. The zip file contains a WSDL file that describes the Web services, and any XSD files referenced from within the WSDL file.

Publishing the human task WSDL:

Use the administrative console to publish the WSDL file.

Procedure

1. Log on to the administrative console with a user ID with administrator rights.
2. Select **Applications** → **SCA modules**

Note: You can also select **Applications** → **Enterprise applications** to display a list of all available enterprise applications.

3. Choose the **TaskContainer** application from the list of SCA modules or applications.
4. Select **Publish WSDL files** from the list of **Additional properties**
5. Click on the zip file in the list.
6. On the File Download window that appears, click **Save**.
7. Browse to a local folder and click **Save**.

Results

The exported zip file is named TaskContainer_WSDLFiles.zip. The zip file contains a WSDL file that describes the Web services, and any XSD files referenced from within the WSDL file.

Exporting business objects:

Business processes and human tasks have well-defined interfaces that allow them to be accessed externally as Web services. If these interfaces reference business objects, you need to export the interface definitions and business objects to your client programming environment.

About this task

This procedure must be repeated for each business object that your client application needs to interact with.

In WebSphere Process Server, business objects define the format of request, response and fault messages that interact with business processes or human tasks. These messages can also contain definitions of complex data types.

For example, to create and start a human task, the following items of information must be passed to the task interface:

- The task template name
- The task template namespace

- An input message, containing formatted business data
- A response wrapper for returning the response message
- A fault message for returning faults and exceptions

These items are encapsulated within a single business object. All operations of the Web service interface are modeled as a "document/literal wrapped" operation. Input and output parameters for these operations are encapsulated in wrapper documents. Other business objects define the corresponding response and fault message formats.

In order to create and start the business process or human task through a Web service, these wrapper objects must be made available to the client application on the client side.

This is achieved by exporting the business objects from the WebSphere environment as Web Service Definition Language (WSDL) and XML Schema Definition (XSD) files, importing the data type definitions into your client programming environment, then converting them to helper classes for use by the client application.

Procedure

1. Launch the WebSphere Integration Developer Workspace if it is not already running.
2. Select the Library module containing the business objects to be exported. A Library module is a compressed file that contains the necessary business objects.
3. Export the Library module.
4. Copy the exported files to your client application development environment.

Example

Assume a business process exposes the following Web service operation:

```
<wsdl:operation name="updateCustomer">
  <wsdl:input message="tns:updateCustomerRequestMsg"
    name="updateCustomerRequest"/>
  <wsdl:output message="tns:updateCustomerResponseMsg"
    name="updateCustomerResponse"/>
  <wsdl:fault message="tns:updateCustomerFaultMsg"
    name="updateCustomerFault"/>
</wsdl:operation>
```

with the WSDL messages defined as:

```
<wsdl:message name="updateCustomerRequestMsg">
  <wsdl:part element="types:updateCustomer"
    name="updateCustomerParameters"/>
</wsdl:message>
<wsdl:message name="updateCustomerResponseMsg">
  <wsdl:part element="types:updateCustomerResponse"
    name="updateCustomerResult"/>
</wsdl:message>
<wsdl:message name="updateCustomerFaultMsg">
  <wsdl:part element="types:updateCustomerFault"
    name="updateCustomerFault"/>
</wsdl:message>
```

The *concrete* customer-defined elements `types:updateCustomer`, `types:updateCustomerResponse`, and `types:updateCustomerFault` must be passed to

and received from the Web services APIs using <xsd:any> parameters in all *generic* operations (call, sendMessage, and so on) performed by the client application. These customer-defined elements are created, serialized and deserialized on the client application side using helper classes that are generated using the exported XSD files.

Related tasks

“Creating helper classes for BPEL processes (.NET)” on page 125
Certain Web services API operations require client applications to use “document/literal” style wrapped elements. Client applications require helper classes to help them generate the necessary wrapper elements.

Using files on the client CD

As an alternative to exporting artifacts from the WebSphere server environment, you can copy the files necessary for generating a client application from the WebSphere Process Server client CD.

In this case, you must manually modify the default Web services endpoint address of the Business Flow Manager API or Human Task Manager API.

If the client application is to access both APIs, you must edit the default endpoint address for both APIs.

Copying files from the client CD:

The files necessary to access the Web services APIs are available on the WebSphere Process Server client CD.

Procedure

1. Access the client CD and browse to the ProcessChoreographer\client directory.
2. Copy the necessary files to your client application development environment.

For the Business Flow Manager API, copy:

BFMWS.wsdl

Describes the Web services available in the Business Flow Manager Web services API. This file contains the endpoint address.

BFMIF.wsdl

Describes the parameters and data type of each Web service in the Business Flow Manager Web services API.

BFMIF.xsd

Describes data types used in the Business Flow Manager Web services API.

BPCGEN.xsd

Contains data types that are common between the Business Flow Manager and Human Task Manager Web services APIs.

For the Human Task Manager API, copy:

HTMWS.wsdl

Describes the Web services available in the Human Task Manager Web services API. This file contains the endpoint address.

HTMIF.wsdl

Describes the parameters and data type of each Web service in the Human Task Manager Web services API.

HTMIF.xsd

Describes data types used in the Human Task Manager Web services API.

BPCGEN.xsd

Contains data types that are common between the Business Flow Manager and Human Task Manager Web services APIs.

Note: The BPCGen.xsd file is common to both APIs.

After you copy the files, you must manually change the Web services API endpoint address the BFMWS.wsdl or HTMWS.wsdl files to that of the WebSphere application server that is hosting the Web services APIs.

Manually changing the Web service endpoint address:

If you copy files from the client CD, you must change the default Web service endpoint address specified in WSDL files to that of the server that is hosting the Web services APIs.

About this task

You can use the administrative console to set the Web service endpoint address before exporting the WSDL files. If, however, you copy the WSDL files from the WebSphere Process Server client CD, you must modify the default Web service endpoint address manually.

The Web service endpoint address to use depends on your WebSphere server configuration:

- Scenario 1. There is a single WebSphere server. The WebSphere endpoint address to specify is the host name and port number of the server, for example **host1:9080**.
- Scenario 2. A WebSphere cluster composed of several servers. The WebSphere endpoint address to specify is the host name and port of the server that is hosting the Web services APIs, for example, **host2:9081**.
- Scenario 3. A Web server is used as a front end. The WebSphere endpoint address to specify is the host name and port of the Web server, for example: **host:80**.

Changing the Business Flow Manager API endpoint:

If you copy the Business Flow Manager API files from the WebSphere Process Server client CD, you must manually edit the default endpoint address.

Procedure

1. Navigate to the directory containing the files copied from the client CD.
2. Open the BFMWS.wsdl file in a text editor or XML editor.
3. Locate the soap:address element (towards the bottom of the file).
4. Modify the value of the location attribute with the HTTP URL of the server on which the Web service API is running. To do this:
 - a. Optionally, replace http with https to use the more secure HTTPS protocol.
 - b. Replace *localhost* with the host name or IP address of the Web services APIs server endpoint address.
 - c. Replace *9080* with the port number of the application server.

- d. Replace *BPEContainer_N1_server1* with the context root of the application running the Web services API. The default context root is composed of:
 - *BPEContainer*. The application name.
 - *N1*. The node name.
 - *server1*. The server name.
- e. Do not modify the fixed portion of the URL (*/sca/com/ibm/bpe/api/BFMWS*).

For example, if the application is running on the server **s1.n1.ibm.com** and the server is accepting SOAP/HTTP requests at port **9080**, modify the `soap:address` element as follows:

```
<soap:address location="http://s1.n1.ibm.com:9080/
                    BPEContainer_N1_server1/sca/com/ibm/bpe/api/BFMWS"/>
```

Related concepts

“Adding security (Java Web services)” on page 120

You must secure Web service communications by implementing security mechanisms in your client application.

Related tasks

“Adding security (.NET)” on page 128

You can secure Web service communications by integrating security mechanisms into your client application.

Changing the Human Task Manager API endpoint:

If you copy the Human Task Manager API files from the WebSphere Process Server client CD, you must manually edit the default endpoint address.

Procedure

1. Navigate to the directory containing the files copied from the client CD.
2. Open the `HTMWS.wsdl` file in a text editor or XML editor.
3. Locate the `soap:address` element (towards the bottom of the file).
4. Modify the value of the `location` attribute with the correct endpoint address. To do this:
 - a. Optionally, replace `http` with `https` to use the more secure HTTPS protocol.
 - b. Replace `localhost` with the host name or IP address of the Web services API server’s endpoint address.
 - c. Replace `9080` with the port number of the application server.
 - d. Replace *HTMContainer_N1_server1* with the context root of the application running the Web services API. The default context root is composed of:
 - *HTMContainer*. The application name.
 - *N1*. The node name.
 - *server1*. The server name.
 - e. Do not modify the fixed portion of the URL (*/sca/com/ibm/task/api/HTMWS*).

For example, if the application is running on the server **s1.n1.ibm.com** and the server is accepting SOAP/HTTPS requests at port **9081**, modify the `soap:address` element as follows:

```
<soap:address location="https://s1.n1.ibm.com:9081/
                    HTMContainer_N1_server1/sca/com/ibm/task/api/HTMWS"/>
```

Related concepts

“Adding security (Java Web services)” on page 120

You must secure Web service communications by implementing security mechanisms in your client application.

Related tasks

“Adding security (.NET)” on page 128

You can secure Web service communications by integrating security mechanisms into your client application.

Developing client applications in the Java Web services environment

You can use any Java-based development environment compatible with Java Web services to develop client applications for the Web services APIs.

Generating a proxy client (Java Web services)

Java Web service client applications use a *proxy client* to interact with the Web services APIs.

About this task

A proxy client for Java Web services contains a number of Java Bean classes that the client application calls to perform Web service requests. The proxy client handles the assembly of service parameters into SOAP messages, sends SOAP messages to the Web service over HTTP, receives responses from the Web service, and passes any returned data to the client application.

Basically, therefore, a proxy client allows a client application to call a Web service as if it were a local function.

Note: You only need to generate a proxy client once. All client applications accessing the same Web services API can then use the same proxy client.

In the IBM® Web services environment, there are two ways to generate a proxy client:

- Using Rational® Application Developer or WebSphere Integration Developer integrated development environments.
- Using the WSDL2Java command-line tool.

Other Java Web services development environments usually include either the WSDL2Java tool or proprietary client application generation facilities.

Using Rational Application Developer to generate a proxy client:

The Rational Application Developer integrated development environment allows you to generate a proxy client for your client application.

Before you begin

Before generating a proxy client, you must have previously exported the WSDL files that describe the business process or human task Web services interfaces from the WebSphere environment (or the WebSphere Process Server client CD) and copied them to your client programming environment.

Procedure

1. Add the appropriate WSDL file to your project:
 - For business processes:

- a. Unzip the exported file
BPEContainer_nodename_servername_WSDLFiles.zip to a temporary directory.
- b. Import the subdirectory META-INF from the unzipped directory
BPEContainer_nodename_servername.ear/b.jar.
- For human tasks:
 - a. Unzip the exported file
TaskContainer_nodename_servername_WSDLFiles.zip to a temporary directory.
 - b. Import the subdirectory META-INF from the unzipped directory
TaskContainer_nodename_servername.ear/h.jar.

A new directory wsdl and subdirectory structure are created in your project.

2. Modify the Web Service wizard properties:
 - a. In Rational Application Developer, choose **Preferences** → **Web services** → **Code generation** → **IBM WebSphere runtime**.
 - b. Select the **Generate Java from WSDL using the no wrapped style** option.

Note: If you cannot select the **Web services** option in the **Preferences** menu, you must first enable the required capabilities as follows: **Window** → **Preferences** → **Workbench** → **Capabilities**. Click on **Web Service Developer** and click **OK**. Then reopen the Preferences window and change the **Code Generation** option.

3. Select the BFMWS.WSDL or HTMWWS.WSDL file located in the newly-created wsdl directory.
4. Right-click and choose **Web services** → **Generate client**.
Before continuing with the remaining steps, ensure that the server has started.
5. On the Web Services window, click **Next** to accept all defaults.
6. On the Web Service Selection window, click **Next** to accept all defaults.
7. On the Client Environment Configuration window:
 - a. Click **Edit** and change the Web service runtime option to IBM WebSphere
 - b. Change the J2EE Version option to 1.4.
 - c. Click **OK**.
 - d. Click **Next**.
8. This step is only necessary if you need to generate a Web Services client that includes both Business Process and Human Task Web Services APIs, as there are duplicate methods in both WSDL files.
 - a. On the Web Service Proxy window, select Define custom mapping for namespace to package then click **OK**.
 - b. On the Web Service Client namespace to package mapping window, add the following namespaces and package:
For BFMWS.wsdl:

Namespace	Package
http://www.ibm.com/xmlns/prod/websphere/business-process/types/6.0	com.ibm.sca.bpe
http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0	com.ibm.sca.bpe
http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0/Binding	com.ibm.sca.bpe

Namespace	Package
http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/6.0	com.ibm.sca.bpe

For HTMWS.wsdl:

Namespace	Package
http://www.ibm.com/xmlns/prod/websphere/human-task/types/6.0	com.ibm.sca.task
http://www.ibm.com/xmlns/prod/websphere/human-task/services/6.0	com.ibm.sca.task
http://www.ibm.com/xmlns/prod/websphere/human-task/services/6.0/Binding	com.ibm.sca.task
http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/6.0	com.ibm.sca.task

If asked to confirm overwriting, click **YesToAll**.

9. Click **Finish**.

Results

A proxy client, made up of a number of proxy, locator and helper Java classes, is generated and added to your project. The deployment descriptor is also updated.

Using WSDL2Java to generate a proxy client:

WSDL2Java is a command-line tool that generates a proxy client. A proxy client make it easier to program client applications.

Before you begin

Before generating a proxy client, you must have previously exported the WSDL files that describe the business process or human task Web services APIs from the WebSphere environment (or the WebSphere Process Server client CD) and copied them to your client programming environment.

About this task

Procedure

1. Use the WSDL2Java tool to generate a proxy client: Type:

```
wsdl2java options WSDLfilepath
```

Where:

- *options* include:

-noWrappedOperations (-w)

Disables the detection of wrapped operations. Java beans for request and response messages are generated.

Note: This is not the default value.

-role (-r)

Specify the value **client** to generate files and binding files for client-side development.

-container (-c)

The client-side container to use. Valid arguments include:

client A client container

ejb An Enterprise JavaBeans (EJB) container.

none No container

web A Web container

-output (-o)

The folder in which to store the generated files.

For a complete list of WSDL2Java parameters, use the **-help** command line switch, or refer to the online help for the WSDL2Java tool in the WID/RAD.

- *WSDLfilepath* is the path and filename of the WSDL file that you exported from WebSphere environment or copied from the client CD.

The following example generates a proxy client for the Human Task Activities Web services API:

```
call wsd12java.bat -r client -c client -noWrappedOperations
                    -output c:\ws\proxyClient c:\ws\bin\HTMWS.wsd1
```

2. Include the generated class files in your project.

Creating helper classes for BPEL processes (Java Web services)

Business objects referenced in concrete API requests (for example, `sendMessage`, or `call`) require client applications to use "document/literal wrapped" style elements. Client applications require helper classes to help them generate the necessary wrapper elements.

Before you begin

To create helper classes, you must have exported the WSDL file of the Web services API from the WebSphere Process Server environment.

About this task

The `call()` and `sendMessage()` operations of the Web services APIs allow interaction with BPEL processes on the WebSphere Process Server. The input message of the `call()` operation expects the document/literal wrapper of the process input message to be provided.

There are a number of possible techniques for generating helper classes for a BPEL process or human task, including:

1. Use the `SoapElement` object.

In the Rational Application Developer environment available in WebSphere Integration Developer, the Web service engine supports JAX-RPC 1.1. In JAX-RPC 1.1, the `SoapElement` object extends a Document Object Model (DOM) element, so it is possible to use the DOM API to create, read, load, and save SOAP messages.

For example, assume the WSDL file contains the following input message for a workflow process or human task:

```
<xsd:element name="operation1">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="input1" nillable="true" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```


The WSDL file is created when you develop a process or human task module. To create the corresponding SOAP message in your client application using the DOM API:

```
SOAPFactory soapfactoryinstance = SOAPFactory.newInstance();
SOAPElement soapmessage = soapfactoryinstance.createElement
    ("operation1", namespaceprefix, interfaceURI);
SOAPElement inutelement = soapfactoryinstance.createElement("input1");
inutelement.addTextNode( message value);
soapmessage.addChildElement(oututelement);
```


The following example shows how to create input parameters for the sendMessage operation in your client application:

```
SendMessage inWsend = new SendMessage();
inWsend.setProcessTemplateName(processtemplatename);
inWsend.setPortType(porttype);
inWsend.setOperation(operationname);
inWsend.set_any(soapmessage);
```

2. Use the WebSphere Custom Data Binding feature.

This technique is described in the following developerWorks articles:

- How to choose a custom mapping technology for Web services
- Developing Web Services with EMF SDOs for complex XML schema

 [Interoperability With Patterns and Strategies for Document-Based Web Services](#)

 [Web Services support for Schema/WSDL\(s\) containing optional JAX-RPC 1.0/1.1 XML Schema Types](#)

Creating a client application (Java Web services)

A client application sends requests to and receives responses from the Web services APIs. By using a proxy client to manage communications and helper classes to format complex data types, a client application can invoke Web service methods as if they were local functions.

Before you begin

Before starting to create a client application, generate the proxy client and any necessary helper classes.

About this task

You can develop client applications using any Web services-compatible development tool, for example IBM Rational Application Developer (RAD). You can build any type of Web services application to call the Web services APIs.

Procedure

1. Create a new client application project.
2. Generate the proxy client and add the Java helper classes to your project.
3. Code your client application.
4. Build the project.
5. Run the client application.

The following example shows how to use the Business Flow Manager Web service API.

```

// create the proxy
    BFMIFProxy proxy = new BFMIFProxy();
// prepare the input data for the operation
    GetProcessTemplate iw = new GetProcessTemplate();
    iw.setIdentifier(your_process_template_name);

// invoke the operation
    GetProcessTemplateResponse ow = proxy.getProcessTemplate(iw);

// process output of the operation
    ProcessTemplateType ptd = ow.getProcessTemplate();
    System.out.println("getName= " + ptd.getName());
    System.out.println("getPtid= " + ptd.getPtid());

```

Related tasks

“Generating a proxy client (Java Web services)” on page 115

Java Web service client applications use a *proxy client* to interact with the Web services APIs.

“Creating helper classes for BPEL processes (Java Web services)” on page 118
Business objects referenced in concrete API requests (for example, `sendMessage`, or `call`) require client applications to use “document/literal wrapped” style elements. Client applications require helper classes to help them generate the necessary wrapper elements.

Adding security (Java Web services)

You must secure Web service communications by implementing security mechanisms in your client application.

WebSphere Application Server currently supports the following security mechanisms for the Web services APIs:


- The user name token
- Lightweight Third Party Authentication (LTPA)

Related concepts

Authorization roles for business processes

A role is a set of people who share the same level of authorization. Actions that you can take on business processes depend on your authorization role. This role can be a J2EE role or an instance-based role.

Related information

 http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r1mx/topic/com.ibm.websphere.bpc.610.doc/doc/bpc/c6task_auth.html

Implementing the user name token:

The user name token security mechanism provides user name and password credentials.

About this task

With the user name token security mechanism, you can choose to implement various *callback handlers*. Depending on your choice:

- You are prompted to supply a user name and password each time you run the client application.
- The user name and password are written into the deployment descriptor.

In either case, the supplied user name and password must match those of an authorized role in the corresponding business process container or human task container.

The user name and password are encapsulated in the request message envelope, and so appear "in clear" in the SOAP message header. It is therefore strongly recommended that you configure the client application to use the HTTPS (HTTP over SSL) communications protocol. All communications are then encrypted. You can select the HTTPS communications protocol when you specify the Web service API's endpoint URL address.

To define a user name token:

Procedure

1. Create a security token:
 - a. Open the **Deployment Editor** of your module
 - b. Click the **WS Extension** tab.
 - c. Under **Service References**, the following Web Service References may be listed:
 - service/BFMWSService for business processes
 - service/HTMWSService for human tasksWhich are listed depends on whether BFMWS.wsdl (for business process), HTMWS.wsdl (for human tasks), or both, were added when generating the proxy client.
 - d. For both service references:
 - 1) Select one of the **Service References**.
 - 2) Expand the **Request Generator Configuration** section.
 - 3) Expand the **Security Token** subsection.
 - 4) Click **Add**. The Security Token window opens.
 - 5) In the **Name** field, type a name for the new security token: **UserNameTokenBFM** or **UserNameTokenHTM** .
 - 6) In the **Token type** drop-down list, select **Username**. (The **Local name** field is automatically populated with a default value.)
 - 7) Leave the **URI** field blank. No URI value is required for a user name token.
 - 8) Click **OK**.
2. Create a token generator:
 - a. Open the **Deployment Editor** of your module
 - b. Click on the **WS Binding** tab
 - c. Under **Service References**, the same Web Service References are listed as in the previous step:
 - service/BFMWSService for business processes
 - service/HTMWSService for human tasks
 - d. For both service references:
 - 1) Select one of the **Service References**.
 - 2) Expand the **Security Request Generator Binding Configuration** section.
 - 3) Expand the **Token Generator** subsection.
 - 4) Click **Add**. The Token Generator window opens.

- 5) In the **Name** field, type a name for the new token generator, such as "UserNameTokenGeneratorBFM" or "UserNameTokenGeneratorHTM".
- 6) In the **Token generator class** field, ensure that the following token generator class is selected:
com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator.
- 7) In the **Security token** drop-down list, select the appropriate security token that you created earlier.
- 8) Select the **Use value type** check box.
- 9) In the **Value type** field, select **Username Token**. (The **Local name** field is automatically populated to reflect your choice of **Username Token**.)
- 10) In the **Call back handler** field, type either "com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler" (which prompts for the user name and password when you run the client application) or "com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler".
- 11) If you choose **NonPromptCallbackHandler**, you must specify a valid user name and password in the corresponding field of the deployment descriptor.
- 12) Click **OK**.

Related tasks

"Specifying the Web service endpoint address" on page 107

The Web service endpoint address is the URL that a client application must specify to access the Web services APIs. The endpoint address is written into the WSDL file that you export to generate a proxy client for your client application.

Related information

 IBM WebSphere Developer Technical Journal: Web services security with WebSphere Application Server V6

Implementing the LTPA security mechanism:

The Lightweight Third Party Authentication (LTPA) security mechanism can be used when the client application is running within a previously established security context.

About this task

The LTPA security mechanism is only available if your client application is running in a secure environment in which a security context has already been established. For example, if your client application is running in an Enterprise JavaBeans (EJB) container, then the EJB client must log in before being able to invoke the client application. A security context is then established. If the EJB client application then invokes a Web service, the LTPA callback handler retrieves the LTPA token from the security context and adds it to the SOAP request message. On the server side, the LTPA token is handled by the LTPA mechanism.

To implement the LTPA security mechanism:

Procedure

1. In the Rational Application Developer environment available in WebSphere Integration Developer, choose **WS Binding** → **Security Request Generator Binding Configuration** → **Token Generator**.

2. Create a security token:
 - a. Open the **Deployment Editor** of your module
 - b. Click the **WS Extension** tab.
 - c. Under **Service References**, the following **Web Service References** may be listed:
 - service/BFMWSService for business processes
 - service/HTMWSService for human tasks

Which are listed depends on whether BFMWS.wsdl (for business process), HTMWS.wsdl (for human tasks), or both, were added when generating the proxy client.
 - d. For both service references:
 - 1) Select one of the **Service References**.
 - 2) Expand the **Request Generator Configuration** section.
 - 3) Expand the **Security Token** subsection.
 - 4) Click **Add**. The Security Token window opens.
 - 5) In the **Name** field, type a name for the new security token: **LTPATokenBFM** or **LTPATokenHTM** .
 - 6) In the **Token type** drop-down list, select **LTPAToken**. (The **URI** and **Local name** fields are automatically populated with default values.)
 - 7) Click **OK**.
3. Create a token generator:
 - a. Open the **Deployment Editor** of your module
 - b. Click on the **WS Binding** tab
 - c. Under **Service References**, the same Web Service References are listed as in the previous step:
 - service/BFMWSService for business processes
 - service/HTMWSService for human tasks
 - d. For both service references:
 - 1) Select one of the **Service References**.
 - 2) Expand the **Security Request Generator Binding Configuration** section.
 - 3) Expand the **Token Generator** subsection.
 - 4) Click **Add**. The Token Generator window opens.
 - 5) In the **Name** field, type a name for the new token generator, such as "LTPATokenGeneratorBFM" or "LTPATokenGeneratorHTM".
 - 6) In the **Token generator class** field, ensure that the following token generator class is selected:
com.ibm.wsspi.wssecurity.token.LTPATokenGenerator.
 - 7) In the **Security token** drop-down list, select the appropriate security token that you created earlier.
 - 8) Select the **Use value type** check box.
 - 9) In the **Value type** field, select **LTPAToken**. (The **URI** and **Local name** fields are automatically populated to reflect your choice of **LTPA Token**.)
 - 10) In the **Call back handler** field, type either "com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler".
 - 11) Click **OK**.

Results

At runtime, the `LTPATokenCallbackHandler` retrieves the LTPA token from the existing security context and adds it to the SOAP request message.

Adding transaction support (Java Web services)

Java Web service client applications can be configured to allow server-side request processing to participate in the client's transaction, by passing a client application context as part of the service request. This atomic transaction support is defined in the Web Services-Atomic Transaction (WS-AT) specification.

About this task

WebSphere Application Server runs each Web services API request as a separate atomic transaction. Client applications can be configured to use transaction support in one of the following ways:

- Participate in the transaction. Server-side request processing is performed within the client application transaction context. Then, if the server encounters a problem while the Web services API request is running and rolls back, the client application's request is also rolled back.
- Not use transaction support. WebSphere Application Server still creates a new transaction in which to run the request, but server-side request processing is not performed with the client application transaction context.

Developing client applications in the .NET environment

Microsoft .NET offers a powerful development environment in which to connect applications through Web services.

Generating a proxy client (.NET)

.NET client applications use a *proxy client* to interact with the Web service APIs. A proxy client shields client applications from the complexity of the Web service messaging protocol.

Before you begin

To create a proxy client, you must first export a number of WSDL files from the WebSphere environment and copy them to your client programming environment.

Note: If you have the WebSphere Process Server client CD, you can copy the files from there instead.

About this task

A proxy client comprises a set of C# bean classes. Each class contains all the methods and objects exposed by a single Web service. The service methods handle the assembly of parameters into complete SOAP messages, send SOAP messages to the Web service over HTTP, receives responses from the Web service, and handle any returned data.

Note: You only need to generate a proxy client once. All client applications accessing the Web services APIs can then use the same proxy client.

Procedure

1. Use the WSDL command to generate a proxy client: Type:

```
wSDL options WSDLfilepath
```

Where:

- *options* include:

/language

Allows you to specify the language used to create the proxy class. The default is C#. You can also specify **VB** (Visual Basic), **JS** (JScript), or **VJS** (Visual J#) as the language argument.

/output

The name of the output file, with the appropriate suffix. For example, proxy.cs

/protocol

The protocol implemented in the proxy class. **SOAP** is the default setting.

For a complete list of **WSDL.exe** parameters, use the */?* command line switch, or refer to the online help for the WSDL tool in Visual Studio.

- *WSDLfilepath* is the path and filename of the WSDL file that you exported from the WebSphere environment or copied from the client CD.

The following example generates a proxy client for the Human Task Manager Web services API:

```
wsd1 /language:cs /output:proxycient.cs c:\ws\bin\HTMWS.wsdl
```

2. Compile the proxy client as a Dynamic Link Library (DLL) file.

Creating helper classes for BPEL processes (.NET)

Certain Web services API operations require client applications to use "document/literal" style wrapped elements. Client applications require helper classes to help them generate the necessary wrapper elements.

Before you begin

To create helper classes, you must have exported the WSDL file of the Web services API from the WebSphere Process Server environment.

About this task

The call() and sendMessage() operations of the Web services APIs cause BPEL processes to be launched within WebSphere Process Server. The input message of the call() operation expects the document/literal wrapper of the BPEL process input message to be provided. To generate the necessary beans and classes for the BPEL process, copy the <wsdl:types> element into a new XSD file, then use the xsd.exe tool to generate helper classes.

Procedure

1. If you have not already done so, export the WSDL file of the BPEL process interface from WebSphere Integration Developer.
2. Open the WSDL file in a text editor or XML editor.
3. Copy the contents of all child elements of the <wsdl:types> element and paste it into a new, skeleton, XSD file.
4. Run the xsd.exe tool on the XSD file:

```
call xsd.exe file.xsd /classes /o
```

Where:

file.xsd

The XML Schema Definition file to convert.

/classes (/c)

Generate helper classes that correspond to the contents of the specified XSD file or files.

/output (/o)

Specify the output directory for generated files. If this directory is omitted, the default is the current directory.

For example:

call xsd.exe ProcessCustomer.xsd /classes /output:c:\temp

5. Add the class file that is generated to your client application. If you are using Visual Studio, for example, you can do this using the **Project** → **Add Existing Item** menu option.

If the ProcessCustomer.wsdl file contains the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:bons1="http://com/ibm/bpe/unittest/sca"
  xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  name="ProcessCustomer"
  targetNamespace="http://ProcessTypes/bpel/ProcessCustomer">
  <wsdl:types>
    <xsd:schema targetNamespace="http://ProcessTypes/bpel/ProcessCustomer"
      xmlns:bons1="http://com/ibm/bpe/unittest/sca"
      xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:import namespace="http://com/ibm/bpe/unittest/sca"
        schemaLocation="xsd-includes/http.com.ibm.bpe.unittest.sca.xsd"/>
      <xsd:element name="doit">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="input1" nillable="true" type="bons1:Customer"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="doitResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="output1" nillable="true" type="bons1:Customer"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="doitRequestMsg">
    <wsdl:part element="tns:doit" name="doitParameters"/>
  </wsdl:message>
  <wsdl:message name="doitResponseMsg">
    <wsdl:part element="tns:doitResponse" name="doitResult"/>
  </wsdl:message>
  <wsdl:portType name="ProcessCustomer">
    <wsdl:operation name="doit">
      <wsdl:input message="tns:doitRequestMsg" name="doitRequest"/>
      <wsdl:output message="tns:doitResponseMsg" name="doitResponse"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

The resulting XSD file contains:

```
<xsd:schema xmlns:bons1="http://com/ibm/bpe/unittest/sca"
  xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ProcessTypes/bpel/ProcessCustomer">
```



```

<xsd:import namespace="http://com/ibm/bpe/unittest/sca"
             schemaLocation="Customer.xsd"/>
<xsd:element name="doit">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="input1" type="bons1:Customer" nillable="true"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="doitResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="output1" type="bons1:Customer" nillable="true"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>

```

Related information



Microsoft documentation for the XML Schema Definition Tool (XSD.EXE)

Creating a client application (.NET)

A client application sends requests to and receives responses from the Web services APIs. By using a proxy client to manage communications and helper classes to format complex data types, a client application can invoke Web service methods as if they were local functions.

Before you begin

Before starting to create a client application, generate the proxy client and any necessary helper classes.

About this task

You can develop .NET client applications using any .NET-compatible development tool, for example, Visual Studio .NET. You can build any type of .NET application to call the generic Web service APIs.

Procedure

1. Create a new client application project. For example, create a **WinFX Windows® Application** in Visual Studio.
2. In the project options, add a reference to the Dynamic Link Library (DLL) file of the proxy client. Add all of the helper classes that contain business object definitions to your project. In Visual Studio, for example, you can do this using the **Project** → **Add existing item** option.
3. Create a proxy client object. For example:

```

HTMClient.HTMReference.HumanTaskManagerComponent1Export_HumanTaskManagerHttpService service =
    new HTMClient.HTMReference.HumanTaskManagerComponent1Export_HumanTaskManagerHttpService();

```

4. Declare any business object data types used in messages to be sent to or received from the Web service. For example:

```

HTMClient.HTMReference.TKIID id = new HTMClient.HTMReference.TKIID();

ClipBG bg = new ClipBG();
Clip clip = new Clip();

```

5. Call specific Web service functions and specify any required parameters. For example, to create and start a human task:

```

HTMClient.HTMReference.createAndStartTask task =
    new HTMClient.HTMReference.createAndStartTask();
HTMClient.HTMReference.StartTask sTask = new HTMClient.HTMReference.StartTask();

sTask.taskName = "SimpleTask";
sTask.taskNamespace = "http://myProcess/com/acme/task";
sTask.inputMessage = bg;
task.inputTask = sTask;

```

```
id = service.createAndStartTask(task).outputTask;
```

6. Remote processes and tasks are identified with persistent IDs (*id* in the example in the previous step). For example, to claim a previously created human task:

```

HTMClient.HTMReference.claimTask claim = new HTMClient.HTMReference.claimTask();
claim.inputTask = id;

```

Related tasks

“Generating a proxy client (.NET)” on page 124

.NET client applications use a *proxy client* to interact with the Web service APIs. A proxy client shields client applications from the complexity of the Web service messaging protocol.

“Creating helper classes for BPEL processes (.NET)” on page 125

Certain Web services API operations require client applications to use “document/literal” style wrapped elements. Client applications require helper classes to help them generate the necessary wrapper elements.

Adding security (.NET)

You can secure Web service communications by integrating security mechanisms into your client application.

About this task

These security mechanisms can include user name token (user name and password), or custom binary and XML-based security tokens.

Procedure

1. Download and install the Web Services Enhancements (WSE) 2.0 SP3 for Microsoft .NET. This is available from:

<http://www.microsoft.com/downloads/details.aspx?familyid=1ba1f631-c3e7-420a-bc1e-ef18bab66122&displaylang=en>

2. Modify the generated proxy client code as follows.

Change:

```

public class Export1_MyMicroflowHttpService : System.Web.Services.Protocols.SoapHttpClientProtocol {
    To:
public class Export1_MyMicroflowHttpService : Microsoft.Web.Services2.WebServicesClientProtocol {

```

Note: These modifications are lost if you regenerate the proxy client by running the WSDL.exe tool.

3. Modify the client application code by adding the following lines at the top of the file:

```

using System.Web.Services.Protocols;
using Microsoft.Web.Services2;
using Microsoft.Web.Services2.Security.Tokens;
...

```

4. Add code to implement the desired security mechanism. For example, the following code adds user name and password protection:

```

string user = "U1";
string pwd = "password";
UsernameToken token =
    new UsernameToken(user, pwd, PasswordOption.SendPlainText);

me._proxy.RequestSoapContext.Security.Tokens.Clear();
me._proxy.RequestSoapContext.Security.Tokens.Add(token);

```

Querying business-process and task-related objects

You can use the Web services APIs to query business-process and task-related objects in the Business Process Choreographer database to retrieve specific properties of these objects.

About this task

The Business Process Choreographer database stores template (model) and instance (runtime) data for managing business processes and tasks.

Through the Web services APIs, client applications can issue queries to retrieve information from the database about business processes and tasks.

Client applications can issue a one-off query to retrieve a specific property of an object. Queries that you use often can be saved. These stored queries can then be retrieved and used by your client application.

Queries on business-process and task-related objects

Use the query interface of the Web services APIs to obtain information about business processes and tasks.

Client applications use an SQL-like syntax to query the database.

Example for Java Web services

```

string processTemplateName = "ProcessCustomerLR";
query query1 = new query();
query1.selectClause = "DISTINCT PROCESS_INSTANCE.STARTED, PROCESS_INSTANCE.PIID";
query1.whereClause =
    "PROCESS_INSTANCE.TEMPLATE_NAME = '" + processTemplateName + "'";
query1.orderByClause = "PROCESS_INSTANCE.STARTED";
query1.threshold = null;
query1.timeZone = "UTC"; query1.skipTuples = null;
queryResponse queryResponse1 = proxy.query(query1);

```

Information retrieved from the database is returned through the Web services APIs as a *query result set*.

For example:

```

QueryResultSetType queryResultSet = queryResponse1.queryResultSet;
if (queryResultSet != null) {
    Console.WriteLine("--> QueryResultSetType");
    Console.WriteLine(" . size= " + queryResultSet.size);
    Console.WriteLine(" . numberColumns= " + queryResultSet.numberColumns);
    string indent = " . ";

    // -- the query column info
    QueryColumnInfoType[] queryColumnInfo = queryResultSet.QueryColumnInfo;
    if (queryColumnInfo.Length > 0) {
        Console.WriteLine();
        Console.WriteLine(" . QueryColumnInfoType size= " + queryColumnInfo.Length);
        Console.WriteLine(" | tableName ");
        for (int i = 0; i < queryColumnInfo.Length; i++) {

```

```

        Console.WriteLine( " | " + queryColumnInfo[i].tableName.PadLeft(20) );
    }
    Console.WriteLine();
    Console.WriteLine( " | columnName ");
    for (int i = 0; i < queryColumnInfo.Length ; i++) {
        Console.WriteLine( " | " + queryColumnInfo[i].columnName.PadLeft(20) );
    }
    Console.WriteLine();
    Console.WriteLine( " | data type ");
    for (int i = 0; i < queryColumnInfo.Length ; i++) {
        QueryColumnInfoType tt = queryColumnInfo[i].type;
        Console.WriteLine( " | " + tt.ToString());
    }
    Console.WriteLine();
}
else {
    Console.WriteLine("--> queryColumnInfo= <null>");
}

// - the query result values
string[][] result = queryResultSet.result;
if (result !=null) {
    Console.WriteLine();
    Console.WriteLine("= . result size= " + result.Length);
    for (int i = 0; i &lt; result.Length; i++) {
        Console.WriteLine(indent + i );
        string[] row = result[i];
        for (int j = 0; j &lt; row.Length; j++ ) {
            Console.WriteLine(" | " + row[j]);
        }
        Console.WriteLine();
    }
}
else {
    Console.WriteLine("--> result= <null>");
}
}
else {
    Console.WriteLine("--> QueryResultSetType= <null>");
}
}

```

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

Predefined database views are provided for you to query the object properties. For process templates, the query function has the following syntax:

```

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

```

For task templates, the query function has the following syntax:

```

TaskTemplate[] queryTaskTemplates
    (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 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 one of the following Java 2 Platform, Enterprise Edition (J2EE) roles: BPESystemAdministrator, BPESystemMonitor, TaskSystemAdministrator, or TaskSystemMonitor. Authorization checking using the corresponding work item of the object is not applied.

Example for .NET

```

ProcessTemplateType[] templates = null;

try {
    queryProcessTemplates iw = new queryProcessTemplates();
    iw.whereClause = "PROCESS_TEMPLATE.STATE=PROCESS_TEMPLATE.STATE.STATE_STARTED";
    iw.orderByClause = null;
    iw.threshold = null;
    iw.timeZone = null;

    Console.WriteLine("--> queryProcessTemplates ... ");
    Console.WriteLine("--> query: WHERE " + iw.whereClause + " ORDER BY " +
        iw.orderByClause + " THRESHOLD " + iw.threshold + " TIMEZONE" + iw.timeZone);

    templates = proxy.queryProcessTemplates(iw);

    if (templates.Length < 1) {
        Console.WriteLine("--> No templates found :-(");
    }
    else {
        for (int i = 0; i < templates.Length ; i++) {
            Console.Write("--> found template with ptid: " + templates[i].ptid);
            Console.WriteLine(" and name: " + templates[i].name);
            /* ... other properties of ProcessTemplateType ... */
        }
    }
}
catch (Exception e) {
    Console.WriteLine("exception= " + e);
}

```

Query parameters:

Each query must specify a number of SQL-like clauses and parameters.

A query is made up of:

- Select clause
- Where clause
- Order-by clause
- Skip-tuples parameter
- Threshold parameter
- Time-zone parameter

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 query function of the Web services APIs to query this data.

Managing stored queries

Stored queries provide a way to save queries that are run often. The stored query can be either a query that is available to all users (public query), or a query that belongs to a specific user (private query).

About this task

A stored query is a query that is stored in the database and identified by a name. A private and a public stored query can have the same name; private stored queries from different owners can also have the same name.

You can have stored queries for business process objects, task objects, or a combination of these two object types.

Managing public stored queries

Public stored queries are created by the system administrator. These queries are available to all users.

Managing private stored queries for other users

Private queries can be created by any user. These queries are available only to the owner of the query and the system administrator.

Working with your private stored queries

If you are not a system administrator, you can create, run, and delete your own private stored queries. You can also use the public stored queries that the system administrator created.

Developing JMS client applications

You can develop client applications that access business process applications through the Java Messaging Service (JMS) API.

About this task

Introduction to JMS

WebSphere Process Server Version 6.1 supports asynchronous messaging, based on the Java Messaging Service (JMS) programming interface, as a method of communication.

JMS provides a common way for Java clients (client applications or J2EE applications) to create, send, receive, and read requests as JMS messages.

JMS is an asynchronous message-based interface that:

- Uses either **point-to-point** or **publish/subscribe messaging**. Message-based frameworks can push information to other applications without their requesting

it explicitly. The same information can be delivered to many subscribers in parallel. The Business Process Choreographer's JMS interface supports point-to-point messaging only.

- Offers **rhythm independence**. JMS frameworks function asynchronously, but are also able to simulate a synchronous request/response mode. This allows source and target systems to work simultaneously without having to wait for each other. This ability is extremely useful for the Business Process Choreographer, as it provides the ability to interact asynchronously with long-running business processes.
- Supports **transactions**. Transactions enable client applications to handle groups of messages sent or received as a single atomic unit. JMS transactions run within the server's transaction. For the Business Process Choreographer's JMS interface, you typically send and receive a single message for each transaction.
- **Guarantees information delivery**. JMS frameworks can manage messages in transactional mode and ensure message delivery (though without any guarantee of timeliness of delivery). For the Business Process Choreographer, this reliable message delivery capability is particularly important because it is dealing with business processes.
- Ensures **interoperability between heterogeneous frameworks**. The source and target applications can operate in heterogeneous environments without having to handle problems of communication and execution related to their respective frameworks.
- **Makes exchanges more fluid**. Switching to message mode allows finer-grained information to be exchanged.

Requirements for business processes

Business processes developed with the WebSphere Integration Developer to run on the Business Process Choreographer must conform to specific rules to be accessible through the JMS API.

The requirements are:

1. The interfaces of business processes must be defined using the "document/literal wrapped" style defined in the Java API for XML-based RPC (JAX-RPC 1.1) specification. This is the default style for all business processes and human tasks developed with the WebSphere Integration Developer.
2. Fault messages exposed by business processes and human tasks for Web service operations must comprise a single WSDL message part defined with an XML Schema element. For example:

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

Related information

 [Java API for XML based RPC \(JAX-RPC\) downloads page](#)

 [Which style of WSDL should I use?](#)

Accessing the JMS interface

To send and receive messages through the JMS interface, an application must first create a connection to the BPC.cellname.Bus, create a session, then generate message producers and consumers.

About this task

The process server accepts Java Message Service (JMS) messages that follow the point-to-point paradigm. An application that sends or receives JMS messages must perform the following actions.

The following example assumes that the JMS client is executed in a managed environment (EJB, application client, or Web client container). If you want to execute the JMS client in a J2SE environment, refer to "IBM Client for JMS on J2SE with IBM WebSphere Application Server" at <http://www-1.ibm.com/support/docview.wss?uid=swg24012804>.

Procedure

1. Create a connection to the BPC.*cellname*.Bus. No preconfigured connection factory exists for a client application's requests: a client application can either use the JMS API's ReplyConnectionFactory or create its own connection factory, in which case it can use Java Naming and Directory Interface (JNDI) lookup to retrieve the connection factory. The JNDI-lookup name must be the same as the name specified when configuring the Business Process Choreographer's external request queue. The following example assumes the client application creates its own connection factory named "jms/clientCF".

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

// Look up the connection factory.
// Create a connection factory that connects to the BPC bus.
// Call it, for example, "jms/clientCF".
// Also configure an appropriate authentication alias.
ConnectionFactory connectionFactory =
    (ConnectionFactory)initialContext.lookup("jms/clientCF");
```

```
// Create the connection.
Connection connection = connectionFactory.createConnection();
```

2. Create a session so that message producers and consumers can be created.

```
// Create a transaction session using auto-acknowledgement.
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

3. Create a message producer to send messages. The JNDI-lookup name must be the same as the name specified when configuring the Business Process Choreographer's external request queue.

```
// Look up the destination of the Business Process Choreographer input queue to
// send messages to.
Queue sendQueue = (Queue) initialContext.lookup("jms/BFMJMSAPIQueue");
```

```
// Create a message producer.
MessageProducer producer = session.createProducer(sendQueue);
```

4. Create a message consumer to receive replies. The JNDI-lookup name of the reply destination can specify a user-defined destination, but it can also specify the default (Business Process Choreographer-defined) reply destination jms/BFMJMSReplyQueue. In both cases, the reply destination must lie on the BPC.<cellname>.Bus.

```
// Look up the destination of the reply queue.
Queue replyQueue = (Queue) initialContext.lookup("jms/BFMJMSReplyQueue");
```

```
// Create a message consumer.
MessageConsumer consumer = session.createConsumer(replyQueue);
```

5. Send a message.

```
// Start the connection.
connection.start();
```

```
// Create a message - see the task descriptions for examples - and send it.
```



```

// This method is defined elsewhere ...
String payload = createXMLDocumentForRequest();
TextMessage requestMessage = session.createTextMessage(payload);

// Set mandatory JMS header.
// targetFunctionName is the operation name of JMS API
// (for example, getProcessTemplate, sendMessage)
requestMessage.setStringProperty("TargetFunctionName", targetFunctionName);

// Set the reply queue; this is mandatory if the replyQueue
// is not the default queue (as it is in this example).
requestMessage.setJMSReplyTo(replyQueue);

// Send the message.
producer.send(requestMessage);

// Get the message ID.
String jmsMessageID = requestMessage.getJMSMessageID();

session.commit();

```

6. Receive the reply.

```

// Receive the reply message and analyse the reply.
TextMessage replyMessage = (TextMessage) consumer.receive();

// Get the payload.
String payload = replyMessage.getText();

session.commit();

```

7. Close the connection and free the resources.

```

// Final housekeeping; free the resources.
session.close();
connection.close();

```

Note: It is not necessary to close the connection after each transaction. Once a connection has been started, any number of request and response messages can be exchanged before the connection is closed. The example shows a simple case with a single call within a single business method.

Structure of a Business Process Choreographer JMS message

The header and body of each JMS message must have a predefined structure.

A Java Message Service (JMS) message consists of:

- A message header for message identification and routing information.
- The body (payload) of the message that holds the content.

The Business Process Choreographer supports text message formats only.

Message header

JMS allows clients to access a number of message header fields.

The following header fields can be set by a Business Process Choreographer JMS client:

- **JMSReplyTo**

The destination to send a reply to the request. If this field is not specified in the request message, the reply is sent to the Export interface's default reply destination (an Export is a client interface rendering of a business process

component). This destination can be obtained using `initialContext.lookup("jms/BFMJMSReplyQueue");`

- **TargetFunctionName**

The name of the WSDL operation, for example, "queryProcessTemplates". This field must always be set. Note that the TargetFunctionName specifies the operation of the generic JMS message interface described here. This should not be confused with operations provided by concrete processes or tasks that can be invoked indirectly, for example, using the **call** or **sendMessage** operations.

A Business Process Choreographer client can also access the following header fields:

- **JMSMessageID**

Uniquely identifies a message. Set by the JMS provider when the message is sent. If the client sets the JMSMessageID before sending the message, it is overwritten by the JMS provider. If the ID of the message is required for authentication purposes, the client can retrieve the JMSMessageID after sending the message.

- **JMSCorrelationID**

Links messages. Do not set this field. A Business Process Choreographer reply message contains the JMSMessageID of the request message.

Each response message contains the following JMS header fields:

- **IsBusinessException**

"False" for WSDL output messages, or "true" for WSDL fault messages.

ServiceRuntimeExceptions are not returned to asynchronous client applications. When a severe exception occurs during the processing of a JMS request message, it results in a runtime failure, causing the transaction that is processing this request message to roll back. The JMS request message is then delivered again. If the failure occurs early, during processing of the message as part of the SCA Export (for example, while deserializing the message), retries are attempted up to the maximum number of failed deliveries specified by the SCA Export's receive destination. After the maximum number of failed deliveries is reached, the request message is added to the system exception destination of the Business Process Choreographer bus. If, however, the failure occurs during actual processing of the request by the Business Flow Manager's SCA component, the failed request message is handled by the WebSphere Process Server's failed event management infrastructure, that is, it may end up in the failed event management database if retries do not resolve the exceptional situation.

Message body

The JMS message body is a String containing an XML document representing the document/literal wrapper element of the operation.

A simple example of a valid request message body is:

```
<?xml version="1.0" encoding="UTF-8"?>
<_6:queryProcessTemplates xmlns:_6="http://www.ibm.com/xmlns/prod/
  websphere/business-process/services/6.0">
<whereClause>PROCESS_TEMPLATE.STATE IN (1)</whereClause>
</_6:queryProcessTemplates>
```

Authorization for JMS renderings

To authorize use of the JMS interface, security settings must be enabled in WebSphere Application Server.

When the business process container is installed, the role **JMSAPIUser** must be mapped to a user ID. This user ID is used to issue all JMS API requests. For example, if **JMSAPIUser** is mapped to "User A", all JMS API requests appear to the process engine to originate from "User A".

The **JMSAPIUser** role must be assigned the following authorities:

Request	Required authorization
forceTerminate	Process administrator
sendEvent	Potential activity owner or process administrator

Note: For all other requests, no special authorizations are required.

Special authority is granted to a person with the role of business process administrator. A business process administrator is a special role; it is different from the process administrator of a process instance. A business process administrator has all privileges.

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

no unique ID for: <user ID>

Overview of the JMS API

The JMS message interface (hereafter referred to as the "JMS API") allows you to develop client applications that asynchronously access business processes running in the Business Process Choreographer environment.

The JMS API allows client applications to asynchronously interact with microflows and long-running processes.

The JMS API exposes the same interface as the Web services API, with the following exceptions:

- With the Web services API, the call operation can only be used to invoke microflows. Using the JMS API, however, the call operation can be used to invoke both microflows and long-running processes.
- The following operations are not exposed through the JMS API:
 - The callAsync operation (together with its associated callback operations).
 - The completeAndClaimSuccessor and getParticipatingTask operations

Example - executing a long-running process

For a generic client application to work with long-running processes, the sequence of steps is:

1. Set up the JMS environment, as described in "Accessing the JMS interface" on page 133.
2. Obtain a list of installed process definitions:
 - Send queryProcessTemplates
 - This returns a list of **ProcessTemplate** objects.
3. Obtain a list of start activities (receive or pick with createInstance="yes"):
 - Send getStartActivities.

- This returns a list of **InboundOperationTemplate** objects.
4. Create an input message. This is environment-specific, and may require the use of predeployed, process-specific artifacts.
 5. Create a process instance:
 - Issue a `sendMessage`.

With the JMS API, you may also use the `call` operation for interacting with long-running request-response operations provided by a business process. This operation returns the operation result or fault to the specified reply-to destination, even after a long period of time. Therefore, if you use the `call` operation, you do not need to use the `query` and `getOutputMessage` operations to obtain the process' output or fault message.
 6. Optionally, obtain output messages from the process instances by repeating the following steps:
 - Issue `query` to obtain the finished state of the process instance.
 - Issue `getOutputMessage`.
 7. Optionally, work with additional operations exposed by the process:
 - `getWaitingActivities` or `getActiveEventHandlers` to obtain a list of **InboundOperationTemplate** objects.
 - Create input messages
 - Send messages with `sendMessage`
 8. Optionally, get and set custom properties defined on the process or contained activities with `getCustomProperties` and `setCustomProperties`.
 9. Optionally, finish working with a process instance:
 - Send `delete` and `terminate` to finish working with the long-running process.

Developing JMS applications

JMS client applications must be developed in Java using the Java 2 Enterprise Edition (J2EE) environment.

About this task

JMS client applications exchange request and response messages with the JMS API. To create a request message, the client application fills a JMS `TextMessage` message body with an XML element representing the document/literal wrapper of the corresponding operation.

Copying artifacts

A number of artifacts can be copied from the WebSphere environment to help in the creation of JMS client applications.

Use of these artifacts is only mandatory if the `BOXMLSerializer` is used to create the JMS message body.

There are two ways to obtain these artifacts:

- Publish and export them from the WebSphere Process Server environment.

For WebSphere Process Server 6.1, all client artifacts are to be found in the `install_root\ProcessChoreographer\client` directory. For the JMS API, these artifacts are:

```
BFMIF.wsdl
BFMIF.xsd
BPCGen.xsd
```

- Copy files from the WebSphere Process Server client CD.

Publishing artifacts from the server environment:

To help develop client applications that access the JMS API, you can publish a number of artifacts from the WebSphere server environment.

About this task

For WebSphere Process Server 6.1, all client artifacts are to be found in the *was_home*\ProcessChoreographer\client directory. For the JMS API, these artifacts are:

```
BFMIF.wsdl
BFMIF.xsd
BPCGen.xsd
```

After these artifacts are published, copy them to your client programming environment.

Copying files from the client CD:

The files necessary to access the JMS API are available on the WebSphere Process Server client CD.

Procedure

1. Access the client CD and browse to the ProcessChoreographer\client directory.
2. Copy the necessary files to your client application development environment
For WebSphere Process Server 6.1, all client artifacts are to be found in the \ProcessChoreographer\client directory. For the JMS API, these artifacts are:

```
BFMIF.wsdl
BFMIF.xsd
BPCGen.xsd
```

Checking the response message for business exceptions

JMS client applications must check the message header of all response messages for business exceptions.

About this task

A JMS client application must first check the **IsBusinessException** property in the response message's header.

For example:

```
// receive response message
Message receivedMessage = ((JmsProxy) getToBeInvokedUponObject()).receiveMessage();
String strResponse = ((TextMessage) receivedMessage).getText();

if (receivedMessage.getStringProperty("IsBusinessException") {
    // strResponse is a bussiness fault
    // any api can end w/a processFaultMsg
    // the call api also w/a businessFaultMsg
}
else {
    // strResponse is the output message
}
```

Developing Web applications for business processes and human tasks, using JSF components

Business Process Choreographer provides several JavaServer Faces (JSF) components. You can extend and integrate these components to add business-process and human-task functionality to Web applications.

About this task

You can use WebSphere Integration Developer to build your Web application.

Procedure

1. Create a dynamic project and change the Web Project Features properties to include the JSF base components.

For more information on creating a Web project, go to the information center for WebSphere Integration Developer.

2. Add the prerequisite Business Process Choreographer Explorer Java archive (JAR files).

Add the following files to the WEB-INF/lib directory of your project:

- bpcclientcore.jar
- bfmclientmodel.jar
- htmclientmodel.jar
- bpcjsfcomponents.jar

If you are deploying your Web application on a remote server, also add the following files. These files are needed for remotely accessing the Business Process Choreographer APIs.

- bpe137650.jar
- task137650.jar

In WebSphere Process Server, all of these files are in the following directory:

- On Windows systems: *install_root*\ProcessChoreographer\client
- On UNIX®, Linux®, and i5/OS® systems: *install_root*/ProcessChoreographer/client

3. Add the EJB references that you need to the Web application deployment descriptor, web.xml file.

```
<ejb-ref id="EjbRef_1">
  <ejb-ref-name>ejb/BusinessProcessHome</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>
<ejb-ref id="EjbRef_2">
  <ejb-ref-name>ejb/HumanTaskManagerEJB</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>
<ejb-local-ref id="EjbLocalRef_1">
  <ejb-ref-name>ejb/LocalBusinessProcessHome</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>
<ejb-local-ref id="EjbLocalRef_2">
  <ejb-ref-name>ejb/LocalHumanTaskManagerEJB</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>

```

4. Add the Business Process Choreographer Explorer JSF components to the JSF application.

- a. Add the tag library references that you need for your applications to the JavaServer Pages (JSP) files. Typically, you need the JSF and HTML tag libraries, and the tag library required by the JSF components.

- <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
- <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
- <%@ taglib uri="http://com.ibm.bpe.jsf/taglib" prefix="bpe" %>

- b. Add an <f:view> tag to the body of the JSP page, and an <h:form> tag to the <f:view> tag.

- c. Add the JSF components to the JSP files.

Depending on your application, add the List component, the Details component, the CommandBar component, or the Message component to the JSP files. You can add multiple instances of each component.

- d. Configure the managed beans in the JSF configuration file.

By default, the configuration file is the faces-config.xml file. This file is in the WEB-INF directory of the Web application.

Depending on the component that you add to your JSP file, you also need to add the references to the query and other wrapper objects to the JSF configuration file. To ensure correct error handling, you also need to define both an error bean and a navigation target for the error page in the JSF configuration file.

```

<faces-config>
...
<managed-bean>
  <managed-bean-name>BPCErr</managed-bean-name>
  <managed-bean-class>com.ibm.bpc.clientcore.util.ErrorBeanImpl
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

...
<navigation-rule>
...
<navigation-case>
<description>
The general error page.
</description>
<from-outcome>error</from-outcome>
<to-view-id>/Error.jsp</to-view-id>
</navigation-case>
...
</navigation-rule>
</faces-config>

```

In error situations that trigger the error page, the exception is set on the error bean.

- e. Implement the custom code that you need to support the JSF components.
5. Deploy the application.

If you are deploying the application in a network deployment environment, change the target resource Java Naming and Directory Interface (JNDI) names to values where the Business Flow Manager and Human Task Manager APIs can be found in your cell.

- If your business process containers are configured on another server in the same managed cell, the names have the following structure:

```
cell/nodes/nodename/servers/servername/com/ibm/bpe/api/BusinessManagerHome
cell/nodes/nodename/servers/servername/com/ibm/task/api/HumanTaskManagerHome
```

- If your business process containers are configured on a cluster in the same cell, the names have the following structure:

```
cell/clusters/clustername/com/ibm/bpe/api/BusinessFlowManagerHome
cell/clusters/clustername/com/ibm/task/api/HumanTaskManagerHome
```

Map the EJB references to the JNDI names or manually add the references to the `ibm-web-bnd.xmi` file.

The following table lists the reference bindings and their default mappings.

Table 33. Mapping of the reference bindings to JNDI names

Reference binding	JNDI name	Comments
ejb/BusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	Remote session bean
ejb/LocalBusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	Local session bean
ejb/HumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	Remote session bean
ejb/LocalHumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	Local session bean

Results

Your deployed Web application contains the functionality provided by the Business Process Choreographer Explorer components.

What to do next

If you are using custom JSPs for the process and task messages, you must map the Web modules that are used to deploy the JSPs to the same servers that the custom JSF client is mapped to.

Related concepts

“Error handling in JSF components” on page 144

The JavaServer Faces (JSF) components exploit a predefined managed bean, `BPCError`, for error handling. In error situations that trigger the error page, the exception is set on the error bean.

Related tasks

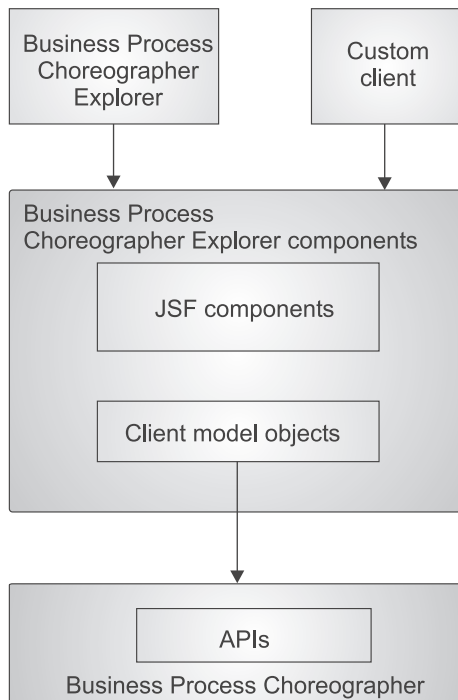
“Accessing the remote interface of the session bean” on page 22

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

Business Process Choreographer Explorer components

The Business Process Choreographer Explorer components are a set of configurable, reusable elements that are based on the JavaServer Faces (JSF) technology. You can embed these elements in Web applications. The Web applications can then access installed business process and human task applications.

The components consist of a set of JSF components and a set of client model objects. The relationship of the components to Business Process Choreographer, Business Process Choreographer Explorer, and other custom clients is shown in the following figure.



JSF components

The Business Process Choreographer Explorer components include the following JSF components. You imbed these JSF components in your JavaServer Pages (JSP) files when you build Web applications for working with business processes and human tasks.

- List component

The List component displays a list of application objects in a table, for example, tasks, activities, process instances, process templates, work items, or escalations. This component has an associated list handler.
- Details component

The Details component displays the properties of tasks, work items, activities, process instances, and process templates. This component has an associated details handler.
- CommandBar component

The CommandBar component displays a bar with buttons. These buttons represent commands that operate on either the object in a details view or the selected objects in a list. These objects are provided by a list handler or a details handler.
- Message component

The Message component displays a message that can contain either a Service Data Object (SDO) or a simple type.

Client model objects

The client model objects are used with the JSF components. The objects implement some of the interfaces of the underlying Business Process Choreographer API and wrap the original object. The client model objects provide national language support for labels and converters for some properties.

Error handling in JSF components

The JavaServer Faces (JSF) components exploit a predefined managed bean, `BPCError`, for error handling. In error situations that trigger the error page, the exception is set on the error bean.

This bean implements the `com.ibm.bpc.clientcore.util.ErrorBean` interface. The error page is displayed in the following situations:

- If an error occurs during the execution of a query that is defined for a list handler, and the error is generated as a `ClientException` error by the `execute` method of a command
- If a `ClientException` error is generated by the `execute` method of a command and this error is not an `ErrorsInCommandException` error nor does it implement the `CommandBarMessage` interface
- If an error message is displayed in the component, and you follow the hyperlink for the message

A default implementation of the `com.ibm.bpc.clientcore.util.ErrorBeanImpl` interface is available.

The interface is defined as follows:

```
public interface ErrorBean {

    public void setException(Exception ex);

    /*
     * This setter method call allows a locale and
     * the exception to be passed. This allows the
     * getExceptionMessage methods to return localized Strings
     */
    public void setException(Exception ex, Locale locale);

    public Exception getException();
    public String getStack();
    public String getNestedExceptionMessage();
    public String getNestedExceptionStack();
    public String getRootExceptionMessage();
    public String getRootExceptionStack();

    /*
     * This method returns the exception message
     * concatenated recursively with the messages of all
     * the nested exceptions.
     */
    public String getAllExceptionMessages();

    /*
     * This method is returns the exception stack
     * concatenated recursively with the stacks of all
     * the nested exceptions.
     */
    public String getAllExceptionStacks();
}
```

Related concepts

“Error handling in the List component” on page 149

When you use the List component to display lists in your JSF application, you can take advantage of the error handling functions provided by the `com.ibm.bpe.jsf.handler.BPCListHandler` class.

Default converters and labels for client model objects

The client model objects implement the corresponding interfaces of the Business Process Choreographer API.

The List component and the Details component operate on any bean. You can display all of the properties of a bean. However, if you want to set the converters and labels that are used for the properties of a bean, you must use either the `column` tag for the List component, or the `property` tag for the Details component. Instead of setting the converters and labels, you can define default converter and labels for the properties by defining the following static methods. You can define the following static methods:

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
    getConverter(String property);
```

The following table shows the client model objects that implement the corresponding Business Flow Manager and Human Task Manager API classes and provide default labels and converter for their properties. This wrapping of the interfaces provides locale-sensitive labels and converters for a set of properties. The following table shows the mapping of the Business Process Choreographer interfaces to the corresponding client model objects.

Table 34. How Business Process Choreographer interfaces are mapped to client model objects

Business Process Choreographer interface	Client model object class
<code>com.ibm.bpe.api.ActivityInstanceData</code>	<code>com.ibm.bpe.clientmodel.bean.ActivityInstanceBean</code>
<code>com.ibm.bpe.api.ActivityServiceTemplateData</code>	<code>com.ibm.bpe.clientmodel.bean.ActivityServiceTemplateBean</code>
<code>com.ibm.bpe.api.ProcessInstanceData</code>	<code>com.ibm.bpe.clientmodel.bean.ProcessInstanceBean</code>
<code>com.ibm.bpe.api.ProcessTemplateData</code>	<code>com.ibm.bpe.clientmodel.bean.ProcessTemplateBean</code>
<code>com.ibm.task.api.Escalation</code>	<code>com.ibm.task.clientmodel.bean.EscalationBean</code>
<code>com.ibm.task.api.Task</code>	<code>com.ibm.task.clientmodel.bean.TaskInstanceBean</code>
<code>com.ibm.task.api.TaskTemplate</code>	<code>com.ibm.task.clientmodel.bean.TaskTemplateBean</code>

Adding the List component to a JSF application

Use the Business Process Choreographer Explorer List component to display a list of client model objects, for example, business process instances or task instances.

Procedure

1. Add the List component to the JavaServer Pages (JSP) file.

Add the `bpe:list` tag to the `h:form` tag. The `bpe:list` tag must include a `model` attribute. Add `bpe:column` tags to the `bpe:list` tag to add the properties of the objects that are to appear in each of the rows in the list.

The following example shows how to add a List component to display task instances.

```

<h:form>

  <bpe:list model="#{TaskPool}">
    <bpe:column name="name" action="taskInstanceDetails" />
    <bpe:column name="state" />
    <bpe:column name="kind" />
    <bpe:column name="owner" />
    <bpe:column name="originator" />
  </bpe:list>

</h:form>

```

The model attribute refers to a managed bean, TaskPool. The managed bean provides the list of Java objects over which the list iterates and then displays in individual rows.

2. Configure the managed bean referred to in the bpe:list tag.

For the List component, this managed bean must be an instance of the com.ibm.bpe.jsf.handler.BPCListHandler class.

The following example shows how to add the TaskPool managed bean to the configuration file.

```

<managed-bean>
  <managed-bean-name>TaskPool</managed-bean-name>
  <managed-bean-class>com.ibm.bpe.jsf.handler.BPCListHandler</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>query</property-name>
    <value>#{TaskPoolQuery}</value>
  </managed-property>
  <managed-property>
    <property-name>type</property-name>
    <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>TaskPoolQuery</managed-bean-name>
  <managed-bean-class>sample.TaskPoolQuery</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>type</property-name>
    <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>htmConnection</managed-bean-name>
  <managed-bean-class>com.ibm.task.clientmodel.HTMConnection</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>jndiName</property-name>
    <value>java:comp/env/ejb/LocalHumanTaskManagerEJB</value>
  </managed-property>
</managed-bean>

```

The example shows that TaskPool has two configurable properties: query and type. The value of the query property refers to another managed bean, TaskPoolQuery. The value of the type property specifies the bean class, the properties of which are shown in the columns of the displayed list. The associated query instance can also have a property type. If a property type is specified, it must be the same as the type specified for the list handler.

You can add any type of query logic to the JSF application as long as the result of the query can be represented as list of strongly-typed beans. For example,

the TaskPoolQuery is implemented using a list of com.ibm.task.clientmodel.bean.TaskInstanceBean objects.

3. Add the custom code for the managed bean that is referred to by the list handler.

The following example shows how to add custom code for the TaskPool managed bean.

```
public class TaskPoolQuery implements Query {

    public List execute throws ClientException {

        // Examine the faces-config file for a managed bean "htmConnection".
        //
        FacesContext ctx = FacesContext.getCurrentInstance();
        Application app = ctx.getApplication();
        ValueBinding htmVb = app.createValueBinding("#{htmConnection}");
        htmConnection = (HTMConnection) htmVb.getValue(ctx);
        HumanTaskManagerService taskService =
            htmConnection.getHumanTaskManagerService();

        // Then call the actual query method on the Human Task Manager service.
        //
        QueryResultSet queryResult = taskService.query(
            "DISTINCT TASK.TKIID, TASK.NAME, TASK.KIND, TASK.STATE, TASK.TYPE,"
            + "TASK.STARTED, TASK.ACTIVATED, TASK.DUE, TASK.EXPIRES, TASK.PRIORITY" ,
            "TASK.KIND IN(101,102,105) AND TASK.STATE IN(2)
            AND WORK ITEM.REASON IN (1)",
            (String)null,
            (Integer)null,
            (TimeZone)null);
        List applicationObjects = transformToTaskList ( queryResult );
        return applicationObjects ;
    }

    private List transformToTaskList(QueryResultSet result) {

        ArrayList array = null;
        int entries = result.size();
        array = new ArrayList( entries );

        // Transforms each row in the QueryResultSet to a task instance beans.
        for (int i = 0; i < entries; i++) {
            result.next();
            array.add( new TaskInstanceBean( result, connection ) );
        }
        return array ;
    }
}
```

The TaskPoolQuery bean queries the properties of the Java objects. This bean must implement the com.ibm.bpc.clientcore.Query interface. When the list handler refreshes its contents, it calls the execute method of the query. The call returns a list of Java objects. The getType method must return the class name of the returned Java objects.

Results

Your JSF application now contains a JavaServer page that displays the properties of the requested list of objects, for example, the state, kind, owner, and originator of the task instances that are available to you.

Related concepts

“User-specific time zone information” on page 149

The JavaServer Faces (JSF) components provide a utility for handling user-specific time zone information in the List component.

Related reference

“List component: Tag definitions” on page 150

The Business Process Choreographer Explorer List component displays a list of objects in a table, for example, tasks, activities, process instances, process templates, work items, and escalations.

How lists are processed

Every instance of the List component is associated with an instance of the `com.ibm.bpe.jsf.handler.BPCListHandler` class.

This list handler tracks the selected items in the associated list and it provides a notification mechanism to associate the list entries with the details pages for the different kinds of items. The list handler is bound to the List component through the **model** attribute of the `bpe:list` tag.

The notification mechanism of the list handler is implemented using the `com.ibm.bpe.jsf.handler.ItemListener` interface. You can register implementations of this interface in the configuration file of your JavaServer Faces (JSF) application.

The notification is triggered when a link in the list is clicked. Links are rendered for all of the columns for which the **action** attribute is set. The value of the **action** attribute is either a JSF navigation target, or a JSF action method that returns a JSF navigation target.

The `BPCListHandler` class also provides a `refreshList` method. You can use this method in JSF method bindings to implement a user interface control for running the query again.

Query implementations

You can use the list handler to display all kinds of objects and their properties. The content of the list that is displayed depends on the list of objects that is returned by the implementation of the `com.ibm.bpc.clientcore.Query` interface that is configured for the list handler. You can set the query either programmatically using the `setQuery` method of the `BPCListHandler` class, or you can configure it in the JSF configuration files of the application.

You can run queries not only against the Business Process Choreographer APIs, but also against any other source of information that is accessible from your application, for example, a content management system or a database. The only requirement is that the result of the query is returned as a `java.util.List` of objects by the `execute` method.

The type of the objects returned must guarantee that the appropriate getter methods are available for all of the properties that are displayed in the columns of the list for which the query is defined. To ensure that the type of the object that is returned fits the list definitions, you can set the value of the `type` property on the `BPCListHandler` instance that is defined in the faces configuration file to the fully qualified class name of the returned objects. You can return this name in the `getType` call of the query implementation. At runtime, the list handler checks that the object types conform to the definitions.

To map error messages to specific entries in a list, the objects returned by the query must implement a method with the signature `public Object getID()`.

Default converters and labels

The items returned by a query must be beans and their class must match the class specified as the type in the definition of the `BPCListHandler` class or `com.ibm.bpc.clientcore.Query` interface. In addition, the List component checks whether the item class or a superclass implements the following methods:

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
    getConverter(String property);
```

If these methods are defined for the beans, the List component uses the label as the default label for the list and the `SimpleConverter` as the default converter for the property. You can overwrite these settings with the **label** and **converterID** attributes of the `bpe:list` tag. For more information, see the Javadoc for the `SimpleConverter` interface and the `ColumnTag` class.

User-specific time zone information

The JavaServer Faces (JSF) components provide a utility for handling user-specific time zone information in the List component.

The `BPCListHandler` class uses the `com.ibm.bpc.clientcore.util.User` interface to get information about the time zone and locale of each user. The List component expects the implementation of the interface to be configured with **user** as the managed-bean name in your JavaServer Faces (JSF) configuration file. If this entry is missing from the configuration file, the time zone in which WebSphere Process Server is running is returned.

The `com.ibm.bpc.clientcore.util.User` interface is defined as follows:

```
public interface User {

    /**
     * The locale used by the client of the user.
     * @return Locale.
     */
    public Locale getLocale();
    /**
     * The time zone used by the client of the user.
     * @return TimeZone.
     */
    public TimeZone getTimeZone();

    /**
     * The name of the user.
     * @return name of the user.
     */
    public String getName();
}
```

Error handling in the List component

When you use the List component to display lists in your JSF application, you can take advantage of the error handling functions provided by the `com.ibm.bpe.jsf.handler.BPCListHandler` class.

Errors that occur when queries are run or commands are executed

If an error occurs during the execution of a query, the `BPCListHandler` class distinguishes between errors that were caused by insufficient access rights and other exceptions. To catch errors due to insufficient access rights, the `rootCause` parameter of the `ClientException` that is thrown by the `execute` method of the query must be a `com.ibm.bpe.api.EngineNotAuthorizedException` or a `com.ibm.task.api.NotAuthorizedException` exception. The List component displays the error message instead of the result of the query.

If the error is not caused by insufficient access rights, the `BPCListHandler` class passes the exception object to the implementation of the `com.ibm.bpc.clientcore.util.ErrorBean` interface that is defined by the `BPCError` key in your JSF application configuration file. When the exception is set, the error navigation target is called.

Errors that occur when working with items that are displayed in a list

The `BPCListHandler` class implements the `com.ibm.bpe.jsf.handler.ErrorHandler` interface. You can provide information about these errors with the `map` parameter of type `java.util.Map` in the `setErrors` method. This map contains identifiers as keys and the exceptions as values. The identifiers must be the values returned by the `getID` method of the object that caused the error. If the map is set and any of the IDs match any of the items displayed in the list, the list handler automatically adds a column containing the error message to the list.

To avoid outdated error messages in the list, reset the errors map. In the following situations, the map is reset automatically:

- The `refreshList` method `BPCListHandler` class is called.
- A new query is set on the `BPCListHandler` class.
- The `CommandBar` component is used to trigger actions on items of the list. The `CommandBar` component uses this mechanism as one of the methods for error handling.

Related concepts

“Error handling in JSF components” on page 144

The JavaServer Faces (JSF) components exploit a predefined managed bean, `BPCError`, for error handling. In error situations that trigger the error page, the exception is set on the error bean.

List component: Tag definitions

The Business Process Choreographer Explorer List component displays a list of objects in a table, for example, tasks, activities, process instances, process templates, work items, and escalations.

The List component consists of the JSF component tags: `bpe:list` and `bpe:column`. The `bpe:column` tag is a subelement of the `bpe:list` tag.

Component class

`com.ibm.bpe.jsf.component.ListComponent`

Example syntax

```
<bpe:list model="{ProcessTemplateList}">
  rows="20"
  styleClass="list"
```



```

        headerStyleClass="listHeader"
        rowClasses="normal">

        <bpe:column name="name" action="processTemplateDetails"/>
        <bpe:column name="validFromTime"/>
        <bpe:column name="executionMode" label="Execution mode"/>
        <bpe:column name="state" converterID="my.state.converter"/>
        <bpe:column name="autoDelete"/>
        <bpe:column name="description"/>

</bpe:list>

```

Tag attributes

The body of the `bpe:list` tag can contain only `bpe:column` tags. When the table is rendered, the List component iterates over the list of application objects and renders all of the columns for each of the objects.

Table 35. *bpe:list* attributes

Attribute	Required	Description
buttonStyleClass	no	The cascading style sheet (CSS) style class for rendering the buttons in the footer area.
cellStyleClass	no	The CSS style class for rendering individual table cells.
checkbox	no	Determines whether the check box for selecting multiple items is rendered. The attribute has a value of either true or false. If the value is set to true, the check box column is rendered.
headerStyleClass	no	The CSS style class for rendering the table header.
model	yes	A value binding for a managed bean of the <code>com.ibm.bpe.jsf.handler.BPCListHandler</code> class.
rows	no	The number of rows that are shown on a page. If the number of items exceeds the number of rows, paging buttons are displayed at the end of the table. Value expressions are not supported for this attribute.
rowClasses	no	The CSS style class for rendering the rows in the table.
selectAll	no	If this attribute is set to true, all of the items in the list are selected by default.
styleClass	no	The CSS style class for rendering the overall table containing titles, rows, and paging buttons.

Table 36. *bpe:column* attributes

Attribute	Required	Description
action	no	If this attribute is specified, a link is rendered in the column. Either a JavaServer Faces action method or the Faces navigation target is triggered when this link is clicked. A JavaServer Faces action method has the following signature: String method().
converterID	no	The Faces converter ID that is used for converting the property value. If this attribute is not set, any Faces converter ID that is provided by the model for this property is used.
label	no	A literal or value binding expression that is used as a label for the header of the column or the cell of the table header row. If this attribute is not set, any label that is provided by the model for this property is used.
name	yes	The name of the property that is displayed in this column.

Adding the Details component to a JSF application

Use the Business Process Choreographer Explorer Details component to display the properties of tasks, work items, activities, process instances, and process templates.

Procedure

1. Add the Details component to the JavaServer Pages (JSP) file.

Add the `bpe:details` tag to the `<h:form>` tag. The `bpe:details` tag must contain a **model** attribute. You can add properties to the Details component with the `bpe:property` tag.

The following example shows how to add a Details component to display some of the properties for a task instance.

```
<h:form>

    <bpe:details model="#{TaskInstanceDetails}">
        <bpe:property name="displayName" />
        <bpe:property name="owner" />
        <bpe:property name="kind" />
        <bpe:property name="state" />
        <bpe:property name="escalated" />
        <bpe:property name="suspended" />
        <bpe:property name="originator" />
        <bpe:property name="activationTime" />
        <bpe:property name="expirationTime" />
    </bpe:details>

</h:form>
```

The **model** attribute refers to a managed bean, `TaskInstanceDetails`. The bean provides the properties of the Java object.

2. Configure the managed bean referred to in the `bpe:details` tag.

For the Details component, this managed bean must be an instance of the `com.ibm.bpe.jsf.handler.BPCDetailsHandler` class. This handler class wraps a Java object and exposes its public properties to the details component.

The following example shows how to add the `TaskInstanceDetails` managed bean to the configuration file.

```
<managed-bean>
  <managed-bean-name>TaskInstanceDetails</managed-bean-name>
  <managed-bean-class>com.ibm.bpe.jsf.handler.BPCDetailsHandler</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>type</property-name>
    <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
  </managed-property>
</managed-bean>
```

The example shows that the `TaskInstanceDetails` bean has a configurable type property. The value of the type property specifies the bean class (`com.ibm.task.clientmodel.bean.TaskInstanceBean`), the properties of which are shown in the rows of the displayed details. The bean class can be any JavaBeans class. If the bean provides default converter and property labels, the converter and the label are used for the rendering in the same way as for the List component.

Results

Your JSF application now contains a JavaServer page that displays the details of the specified object, for example, the details of a task instance.

Related reference

“Details component: Tag definitions”

The Business Process Choreographer Explorer Details component displays the properties of tasks, work items, activities, process instances, and process templates.

Details component: Tag definitions

The Business Process Choreographer Explorer Details component displays the properties of tasks, work items, activities, process instances, and process templates.

The Details component consists of the JSF component tags: `bpe:details` and `bpe:property`. The `bpe:property` tag is a subelement of the `bpe:details` tag.

Component class

`com.ibm.bpe.jsf.component.DetailsComponent`

Example syntax

```
<bpe:details model="{MyActivityDetails}">
  <bpe:property name="name"/>
  <bpe:property name="owner"/>
  <bpe:property name="activated"/>
</bpe:details>

<bpe:details model="{MyActivityDetails}" style="style" styleClass="cssStyle">
  style="style"
  styleClass="cssStyle"
</bpe:details>
```

Tag attributes

Use `bpe:property` tags to specify both the subset of attributes that are shown and the order in which these attributes are shown. If the details tag does not contain any attribute tags, it renders all of the available attributes of the model object.

Table 37. *bpe:details* attributes

Attribute	Required	Description
columnClasses	no	A list of cascading style sheet style (CSS) style classes, separated by commas, for rendering columns.
id	no	The JavaServer Faces ID of the component.
model	yes	A value binding for a managed bean of the <code>com.ibm.bpe.jsf.handler.BPCDetailsHandler</code> class.
rowClasses	no	A list of CSS style classes, separated by commas, for rendering rows.
styleClass	no	The CSS class that is used for rendering the HTML element.

Table 38. *bpe:property* attributes

Attribute	Required	Description
converterID	no	The ID used to register the converter in the JavaServer Faces (JSF) configuration file.
label	no	The label for the property. If this attribute is not set, a default label is provided by the client model class.
name	yes	The name of the property to be displayed. This name must correspond to a named property as defined in the corresponding client model class.

Adding the CommandBar component to a JSF application

Use the Business Process Choreographer Explorer CommandBar component to display a bar with buttons. These buttons represent commands that operate on the details view of an object or the selected objects in a list.

About this task

When the user clicks a button in the user interface, the corresponding command is run on the selected objects. You can add and extend the CommandBar component in your JSF application.

Procedure

1. Add the CommandBar component to the JavaServer Pages (JSP) file.

Add the `bpe:commandbar` tag to the `<h:form>` tag. The `bpe:commandbar` tag must contain a model attribute.

The following example shows how to add a CommandBar component that provides refresh and claim commands for a task instance list.

```
<h:form>
  <bpe:commandbar model="#{TaskInstanceList}">
    <bpe:command commandID="Refresh" >
      action="#{TaskInstanceList.refreshList}"
      label="Refresh"/>
    <bpe:command commandID="MyClaimCommand" >
      label="Claim" >
```

```

        commandClass="<customcode>"/>
    </bpe:commandbar>

```

```
</h:form>
```

The **model** attribute refers to a managed bean. This bean must implement the `ItemProvider` interface and provide the selected Java objects. The `CommandBar` component is usually used with either the `List` component or the `Details` component in the same JSP file. Generally, the `model` that is specified in the tag is the same as the `model` that is specified in the `List` component or `Details` component on the same page. So for the `List` component, for example, the `command` acts on the selected items in the list.

In this example, the **model** attribute refers to the `TaskInstanceList` managed bean. This bean provides the selected objects in the task instance list. The bean must implement the `ItemProvider` interface. This interface is implemented by the `BPCListHandler` class and the `BPCDetailsHandler` class.

- Optional: Configure the managed bean that is referred to in the `bpe:commandbar` tag.

If the `CommandBar` **model** attribute refers to a managed bean that is already configured, for example, for a list or details handler, no further configuration is required. If you change the configuration of either of these handlers or you use a different managed bean, add a managed bean that implements the `ItemProvider` interface to the JSF configuration file.

- Add the code that implements the custom commands to the JSF application.

The following code snippet shows how to write a command class that implements the `Command` interface. This command class (`MyClaimCommand`) is referred to by the `bpe:command` tag in the JSP file.

```

public class MyClaimCommand implements Command {

    public String execute(List selectedObjects) throws ClientException {
        if( selectedObjects != null && selectedObjects.size() > 0 ) {
            try {
                // Determine HumanTaskManagerService from an HTMConnection bean.
                // Configure the bean in the faces-config.xml for easy access
                // in the JSF application.
                FacesContext ctx = FacesContext.getCurrentInstance();
                ValueBinding vb =
                    ctx.getApplication().createValueBinding("{htmConnection}");
                HTMConnection htmConnection = (HTMConnection) htmVB.getValue(ctx);
                HumanTaskManagerService htm =
                    htmConnection.getHumanTaskManagerService();

                Iterator iter = selectedObjects.iterator() ;
                while( iter.hasNext() ) {
                    try {
                        TaskInstanceBean task = (TaskInstanceBean) iter.next() ;
                        TKIID tiid = task.getID() ;

                        htm.claim( tiid ) ;
                        task.setState( new Integer(TaskInstanceBean.STATE_CLAIMED) ) ;

                    }
                    catch( Exception e ) {
                        ; // Error while iterating or claiming task instance.
                        // Ignore for better understanding of the sample.
                    }
                }
            }
            catch( Exception e ) {
                ; // Configuration or communication error.
                // Ignore for better understanding of the sample
            }
        }
    }
}

```

```

    }
    return null;
}

// Default implementations
public boolean isMultiSelectEnabled() { return false; }
public boolean[] isApplicable(List itemsOnList) {return null; }
public void setContext(Object targetModel) {; // Not used here }
}

```

The command is processed in the following way:

- a. A command is invoked when a user clicks the corresponding button in the command bar. The `CommandBar` component retrieves the selected items from the item provider that is specified in the **model** attribute and passes the list of selected objects to the `execute` method of the `commandClass` instance.
- b. The **commandClass** attribute refers to a custom command implementation that implements the `Command` interface. This means that the command must implement the `public String execute(List selectedObjects) throws ClientException` method. The command returns a result that is used to determine the next navigation rule for the JSF application.
- c. After the command completes, the `CommandBar` component evaluates the **action** attribute. The **action** attribute can be a static string or a method binding to a JSF action method with the `public String Method()` signature. Use the **action** attribute to override the outcome of a command class or to explicitly specify an outcome for the navigation rules. The **action** attribute is not processed if the command generates an exception other than an `ErrorsInCommandException` exception.
- d. If the **commandClass** attribute does not have a command class specified, the action is immediately called. For example, for the refresh command in the example, the JSF value expression `#{TaskInstanceList.refreshList}` is called instead of a command.

Results

Your JSF application now contains a `JavaServer` page that implements a customized command bar.

Related reference

“`CommandBar` component: Tag definitions” on page 157

The `Business Process Choreographer Explorer CommandBar` component displays a bar with buttons. These buttons operate on the object in a details view or the selected objects in a list.

How commands are processed

Use the `CommandBar` component to add action buttons to your application. The component creates the buttons for the actions in the user interface and handles the events that are created when a button is clicked.

These buttons trigger functions that act on the objects that are returned by a `com.ibm.bpe.jsf.handler.ItemProvider` interface, such as the `BPCListHandler` class, or the `BPCDetailsHandler` class. The `CommandBar` component uses the item provider that is defined by the value of the **model** attribute in the `bpe:commandbar` tag.

When a button in the command-bar section of the application’s user interface is clicked, the associated event is handled by the `CommandBar` component in the following way.

1. The CommandBar component identifies the implementation of the `com.ibm.bpc.clientcore.Command` interface that is specified for the button that generated the event.
2. If the model associated with the CommandBar component implements the `com.ibm.bpc.jsf.handler.ErrorHandler` interface, the `clearErrorMap` method is invoked to remove error messages from previous events.
3. The `getSelectedItems` method of the `ItemProvider` interface is called. The list of items that is returned is passed to the `execute` method of the command, and the command is invoked.
4. The CommandBar component determines the JavaServer Faces (JSF) navigation target. If an **action** attribute is not specified in the `bpe:commandbar` tag, the return value of the `execute` method specifies the navigation target. If the **action** attribute is set to a JSF method binding, the string returned by the method is interpreted as the navigation target. The **action** attribute can also specify an explicit navigation target.

CommandBar component: Tag definitions

The Business Process Choreographer Explorer CommandBar component displays a bar with buttons. These buttons operate on the object in a details view or the selected objects in a list.

The CommandBar component consists of the JSF component tags: `bpe:commandbar` and `bpe:command`. The `bpe:command` tag is a subelement of the `bpe:commandbar` tag.

Component class

`com.ibm.bpc.jsf.component.CommandBarComponent`

Example syntax

```
<bpe:commandbar model="#{TaskInstanceList}">

  <bpe:command
    commandID="Work on"
    label="Work on..."
    commandClass="com.ibm.bpc.explorer.command.WorkOnTaskCommand"
    context="#{TaskInstanceDetailsBean}"/>

  <bpe:command
    commandID="Cancel"
    label="Cancel"
    commandClass="com.ibm.task.clientmodel.command.CancelClaimTaskCommand"
    context="#{TaskInstanceList}"/>

</bpe:commandbar>
```

Tag attributes

Table 39. `bpe:commandbar` attributes

Attribute	Required	Description
<code>buttonStyleClass</code>	no	The cascading style sheet (CSS) style class that is used for rendering the buttons in the command bar.
<code>id</code>	no	The JavaServer Faces ID of the component.

Table 39. *bpe:commandbar* attributes (continued)

Attribute	Required	Description
model	yes	A value binding expression to a managed bean that implements the <code>ItemProvider</code> interface. This managed bean is usually the <code>com.ibm.bpe.jsf.handler.BPCListHandler</code> class or the <code>com.ibm.bpe.jsf.handler.BPCDetailsHandler</code> class that is used by the List component or Details component in the same JavaServer Pages (JSP) file as the <code>CommandBar</code> component.
styleClass	no	The CSS style class that is used for rendering the command bar.

Table 40. *bpe:command* attributes

Attribute	Required	Description
action	no	A JavaServer Faces action method or the Faces navigation target that is to be triggered by the command button. The navigation target that is returned by the action overwrites all other navigation rules. The action is called when either an exception is not thrown or an <code>ErrorsInCommandException</code> exception is thrown by the command.
commandClass	no	The name of the command class. An instance of the class is created by the <code>CommandBar</code> component and run if the command button is selected.
commandID	yes	The ID of the command.
context	no	An object that provides context for commands that are specified using the <code>commandClass</code> attribute. The context object is retrieved when the command bar is first accessed.
immediate	no	Specifies when the command is triggered. If the value of this attribute is true, the command is triggered before the input of the page is processed. The default is false.
label	yes	The label of the button that is rendered in the command bar.
rendered	no	Determines whether a button is rendered. The value of the attribute can be either a Boolean value or a value expression.
styleClass	no	The CSS style class that is used for rendering the button. This style overrides the button style defined for the command bar.

Adding the Message component to a JSF application

Use the Business Process Choreographer Explorer Message component to render data objects and primitive types in a JavaServer Faces (JSF) application.

About this task

If the message type is a primitive type, a label and an input field are rendered. If the message type is a data object, the component traverses the object and renders the elements within the object.

Procedure

1. Add the Message component to the JavaServer Pages (JSP) file.

Add the `bpe:form` tag to the `<h:form>` tag. The `bpe:form` tag must include a `model` attribute.

The following example shows how to add a Message component.

```
<h:form>

    <h:outputText value="Input Message" />
    <bpe:form model="#{MyHandler.inputMessage}" readOnly="true" />

    <h:outputText value="Output Message" />
    <bpe:form model="#{MyHandler.outputMessage}" />

</h:form>
```

The **model** attribute of the Message component refers to a `com.ibm.bpc.clientcore.MessageWrapper` object. This wrapper object wraps either a Service Data Object (SDO) object or a Java primitive type, for example, `int` or `boolean`. In the example, the message is provided by a property of the `MyHandler` managed bean.

2. Configure the managed bean referred to in the `bpe:form` tag.

The following example shows how to add the `MyHandler` managed bean to the configuration file.

```
<managed-bean>
<managed-bean-name>MyHandler</managed-bean-name>
<managed-bean-class>com.ibm.bpc.sample.jsf.MyHandler</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>

    <managed-property>
        <property-name>type</property-name>
        <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
    </managed-property>

</managed-bean>
```

3. Add the custom code to the JSF application.

The following example shows how to implement input and output messages.

```
public class MyHandler implements ItemListener {

    private TaskInstanceBean taskBean;
    private MessageWrapper inputMessage, outputMessage

    /* Listener method, e.g. when a task instance was selected in a list handler.
     * Ensure that the handler is registered in the faces-config.xml or manually.
     */
    public void itemChanged(Object item) {
        if( item instanceof TaskInstanceBean ) {
            taskBean = (TaskInstanceBean) item ;
        }
    }

    /* Get the input message wrapper
     */
    public MessageWrapper getInputMessage() {
        try{
            inputMessage = taskBean.getInputMessageWrapper() ;
        }
        catch( Exception e ) {
```

```

        ; //...ignore errors for simplicity
    }
    return inputMessage;
}

/* Get the output message wrapper
*/
public MessageWrapper getOutputMessage() {
    // Retrieve the message from the bean. If there is no message, create
    // one if the task has been claimed by the user. Ensure that only
    // potential owners or owners can manipulate the output message.
    try{
        outputMessage = taskBean.getOutputMessageWrapper();
        if( outputMessage == null
            && taskBean.getState() == TaskInstanceBean.STATE_CLAIMED ) {
            HumanTaskManagerService htm = getHumanTaskManagerService();
            outputMessage = new MessageWrapperImpl();
            outputMessage.setMessage(
                htm.createOutputMessage( taskBean.getID() ).getObject()
            );
        }
    }
    catch( Exception e ) {
        ; //...ignore errors for simplicity
    }
    return outputMessage;
}
}

```

The MyHandler managed bean implements the `com.ibm.jsf.handler.ItemListener` interface so that it can register itself as an item listener to list handlers. When the user clicks an item in the list, the MyHandler bean is notified in its `itemChanged(Object item)` method about the selected item. The handler checks the item type and then stores a reference to the associated `TaskInstanceBean` object. To use this interface, add an entry to the `itemListener` list in the appropriate list handler in the `faces-config.xml` file.

The MyHandler bean provides the `getInputMessage` and `getOutputMessage` methods. Both of these methods return a `MessageWrapper` object. The methods delegate the calls to the referenced task instance bean. If the task instance bean returns null, for example, because a message is not set, the handler creates and stores a new, empty message. The Message component displays the messages provided by the MyHandler bean.

Results

Your JSF application now contains a JavaServer page that can render data objects and primitive types.

Related reference

“Message component: Tag definitions”

The Business Process Choreographer Explorer Message component renders `commonj.sdo.DataObject` objects and primitive types, such as integers and strings, in a JavaServer Faces (JSF) application.

Message component: Tag definitions

The Business Process Choreographer Explorer Message component renders `commonj.sdo.DataObject` objects and primitive types, such as integers and strings, in a JavaServer Faces (JSF) application.

The Message component consists of the JSF component tag: `bpe:form`.

Component class

com.ibm.bpe.jsf.component.MessageComponent

Example syntax

```
<bpe:form model="#{TaskInstanceDetailsBean.inputMessageWrapper}"
  simplification="true" readOnly="true"
  styleClass4table="messageData"
  styleClass4output="messageDataOutput">
</bpe:form>
```

Tag attributes

Table 41. *bpe:form* attributes

Attribute	Required	Description
id	no	The JavaServer Faces ID of the component.
model	yes	A value binding expression that refers to either a <code>commonj.sdo.DataObject</code> object or a <code>com.ibm.bpc.clientcore.MessageWrapper</code> object.
readOnly	no	If this attribute is set to true, a read-only form is rendered. By default, this attribute is set to false.
simplification	no	If this attribute is set to true, properties that contain simple types and have a cardinality of zero or one are shown. By default, this attribute is set to true.
style4validinput	no	The cascading style sheet (CSS) style for rendering input that is valid.
style4invalidinput	no	The CSS style for rendering input that is not valid.
styleClass4invalidInput	no	The CSS style class name for rendering input that is not valid.
styleClass4output	no	The CSS style class name for rendering the output elements.
styleClass4table	no	The class name of the CSS table style for rendering the tables rendered by the message component.
styleClass4validInput	no	The CSS style class name for rendering input that is valid.

Developing JSP pages for task and process messages

The Business Process Choreographer Explorer interface provides default input and output forms for displaying and entering business data. You can use JSP pages to provide customized input and output forms.

About this task

To include user-defined JavaServer Pages (JSP) pages in the Web client, you must specify them when you model a human task in WebSphere Integration Developer. For example, you can provide JSP pages for a specific task and its input and

output messages, and for a specific user role or all user roles. At runtime, the user-defined JSP pages are included in the user interface to display output data and collect input data.

The customized forms are not self-contained Web pages; they are HTML fragments that Business Process Choreographer Explorer imbeds in an HTML form, for example, fragments for all of the labels and input fields of a message.

When a button is clicked on the page that contains the customized forms, the input is submitted and validated in Business Process Choreographer Explorer. The validation is based on the type of the properties provided and the locale used in the browser. If the input cannot be validated, the same page is shown again and information about the validation errors is provided in the `messageValidationErrors` request attribute. The information is provided as a map that maps the XML Path Expression (XPath) of the properties that are not valid to the validation exceptions that occurred.

To add customized forms to Business Process Choreographer Explorer, complete the following steps using WebSphere Integration Developer.

Procedure

1. Create the customized forms.

The user-defined JSP pages for the input and output forms used in the Web interface need access to the message data. Use Java snippets in a JSP or the JSP execution language to access the message data. Data in the forms is available through the request context.

2. Assign the JSP pages to a task.

Open the human task in the human task editor. In the client settings, specify the location of the user-defined JSP pages and the role to which the customized form applies, for example, administrator. The client settings for Business Process Choreographer Explorer are stored in the task template. At runtime these settings are retrieved with the task template.

3. Package the user-defined JSP pages in a Web archive (WAR file).

You can either include the WAR file in the enterprise archive with the module that contains the tasks or deploy the WAR file separately. If the JSPs are deployed separately, make the JSPs available on the server where the Business Process Choreographer Explorer or the custom client is deployed.

If you are using custom JSPs for the process and task messages, you must map the Web modules that are used to deploy the JSPs to the same servers that the custom JSF client is mapped to.

Results

The customized forms are rendered in Business Process Choreographer Explorer at runtime.

User-defined JSP fragments

The user-defined JavaServer Pages (JSP) fragments are imbedded in an HTML form tag. At runtime, Business Process Choreographer Explorer includes these fragments in the rendered page.

The user-defined JSP fragment for the input message is imbedded before the JSP fragment for the output message.

```

<html....>
  ...
  <form...>
    Input JSP (display task input message)

    Output JSP (display task output message)

  </form>
  ...
</html>

```

Because the user-defined JSP fragments are embedded in an HTML form tag, you can add input elements. The name of the input element must match the XML Path Language (XPath) expression of the data element. It is important to prefix the name of the input element with the provided prefix value:

```

<input id="address"
      type="text"
      name="{prefix}/selectPromotionalGiftResponse/address"
      value="{messageMap['/selectPromotionalGiftResponse/address']}"
      size="60"
      align="left" />

```

The prefix value is provided as a request attribute. The attribute ensures that the input name is unique in the enclosing form. The prefix is generated by Business Process Choreographer Explorer and it should not be changed:

```
String prefix = (String)request.getAttribute("prefix");
```

The prefix element is set only if the message can be edited in the given context. Output data can be displayed in different ways depending on the state of the human task. For example, if the task is in the claimed state, the output data can be modified. However, if the task is in the finished state, the data can be displayed only. In your JSP fragment, you can test whether the prefix element exists and render the message accordingly. The following JSTL statement shows how you might test whether the prefix element is set.

```

...
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...
<c:choose>
  <c:when test="{not empty prefix}">
    <!--Read/write mode-->
  </c:when>
  <c:otherwise>
    <!--Read-only mode-->
  </c:otherwise>
</c:choose>

```

Creating plug-ins to customize human task functionality

Business Process Choreographer provides an event handling infrastructure for events that occur during the processing of human tasks. Plug-in points are also provided so that you can adapt the functionality to your needs. You can use the service provider interfaces (SPIs) to create customized plug-ins for handling events and the processing of staff queries.

About this task

You can create plug-ins for human task API events and escalation notification events. You can also create a plug-in that processes the results that are returned

from people resolution. For example, at peak periods you might want to add users to the result list to help balance the workload.

You can register your plug-ins on different levels, for all tasks on a global level, for the tasks in an application component, for all of the tasks associated with a task template, or for a single task instance.

Creating API event handlers

An API event occurs when an API method manipulates a human task. Use the API event handler plug-in service provider interface (SPI) to create plug-ins to handle the task events sent by the API or the internal events that have equivalent API events.

About this task

Complete the following steps to create an API event handler.

Procedure

1. Write a class that implements the `APIEventHandlerPlugin2` interface or extends the `APIEventHandler` implementation class. This class can invoke the methods of other classes.
 - If you use the `APIEventHandlerPlugin2` interface, you must implement all of the methods of the `APIEventHandlerPlugin2` interface and the `APIEventHandlerPlugin` interface.
 - If you extend the SPI implementation class, overwrite the methods that you need.

This class runs in the context of a Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) application. Ensure that this class and its helper classes follow the EJB specification.

Tip: If you want to call the `HumanTaskManagerService` interface from this class, do not call a method that updates the task that produced the event. This action results in a database deadlock.

2. Assemble the plug-in class and its helper classes into a JAR file.

If the helper classes are used by several J2EE applications, you can package these classes in a separate JAR file that you register as a shared library.

3. Create a service provider configuration file for the plug-in in the `META-INF/services/` directory of your JAR file.

The configuration file provides the mechanism for identifying and loading the plug-in. This file conforms to the Java 2 service provider interface specification.

- a. Create a file with the name `com.ibm.task.spi.plug-in_nameAPIEventHandlerPlugin`, where *plug-in_name* is the name of the plug-in.

For example, if your plug-in is called `Customer` and it implements the `com.ibm.task.spi.APIEventHandlerPlugin` interface, the name of the configuration file is `com.ibm.task.spi.CustomerAPIEventHandlerPlugin`.

- b. In the first line of the file that is neither a comment line nor a blank line, specify the fully qualified name of the plug-in class that you created in step 1.

For example, if your plug-in class is called `MyAPIEventHandler` and it is in the `com.customer.plugins` package, then the first line of the configuration

file must contain the following entry:
`com.customer.plugins.MyAPIEventHandler.`

Results

You have an installable JAR file that contains a plug-in that handles API events and a service provider configuration file that can be used to load the plug-in.

Tip: You only have one `eventHandlerName` property available to register both API event handlers and notification event handlers. If you want to use both an API event handler and a notification event handler, the plug-in implementations must have the same name, for example, `Customer` as the event handler name for the SPI implementation.

You can implement both plug-ins using a single class, or two separate classes. In both cases, you need to create two files in the `META-INF/services/` directory of your JAR file, for example,
`com.ibm.task.spi.CustomerNotificationEventHandlerPlugin` and
`com.ibm.task.spi.CustomerAPIEventHandlerPlugin`.

Package the plug-in implementation and the helper classes in a single JAR file.

What to do next

You now need to install and register the plug-in so that it is available to the human task container at runtime. You can register API event handlers with a task instance, a task template, or an application component.

API event handlers

API events occur when a human task is modified or it changes state. To handle these API events, the event handler is invoked directly before the task is modified (pre-event method) and just before the API call returns (post-event method).

If the pre-event method throws an `ApplicationVetoException` exception, the API action is not performed, the exception is returned to the API caller, and the transaction associated with the event is rolled back. If the pre-event method was triggered by an internal event and an `ApplicationVetoException` exception is thrown, the internal event, such as an automatic claim, is not performed but an exception is not returned to the client application. In this case, an information message is written to the `SystemOut.log` file. If the API method throws an exception during processing, the exception is caught and passed to the post-event method. The exception is passed again to the caller after the post-event method returns.

The following rules apply to pre-event methods:

- Pre-event methods receive the parameters of the associated API method or internal event.
- Pre-event methods can throw an `ApplicationVetoException` exception to prevent processing from continuing.

The following rules apply to post-event methods:

- Post-event methods receive the parameters that were supplied to the API call, and the return value. If an exception is thrown by the API method implementation, the post-event method also receives the exception.
- Post-event methods cannot modify return values.

- Post-event methods cannot throw exceptions; runtime exceptions are logged but they are ignored.

To implement API event handlers, you can use either the `APIEventHandlerPlugin2` interface, which extends the `APIEventHandlerPlugin` interface, or extend the default `com.ibm.task.spi.APIEventHandler` SPI implementation class. If your event handler inherits from the default implementation class, it always implements the most recent version of the SPI. If you upgrade to a newer version of Business Process Choreographer, fewer changes are necessary if you want to exploit new SPI methods.

If you have both a notification event handler and an API event handler, both of these handlers must have the same name because you can register only one event handler name.

Creating notification event handlers

Notification events are produced when human tasks are escalated. Business Process Choreographer provides functionality for handling escalations, such as creating escalation work items or sending e-mails. You can create notification event handlers to customize the way in which escalations are handled.

About this task

To implement notification event handlers, you can use either the `NotificationEventHandlerPlugin` interface, or you can extend the default `com.ibm.task.spi.NotificationEventHandler` service provider interface (SPI) implementation class.

Complete the following steps to create a notification event handler.

Procedure

1. Write a class that implements the `NotificationEventHandlerPlugin` interface or extends the `NotificationEventHandler` implementation class. This class can invoke the methods of other classes.

If you use the `NotificationEventHandlerPlugin` interface, you must implement all of the interface methods. If you extend the SPI implementation class, overwrite the methods that you need.

This class runs in the context of a Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) application. Ensure that this class and its helper classes follow the EJB specification.

The plug-in is invoked with the authority of the `EscalationUser` role. This role is defined when the human task container is configured.

Tip: If you want to call the `HumanTaskManagerService` interface from this class, do not call a method that updates the task or the escalation that produced the event. This action results in a database deadlock.

2. Assemble the plug-in class and its helper classes into a JAR file.

If the helper classes are used by several J2EE applications, you can package these classes in a separate JAR file that you register as a shared library.

3. Create a service provider configuration file for the plug-in in the `META-INF/services/` directory of your JAR file.

The configuration file provides the mechanism for identifying and loading the plug-in. This file conforms to the Java 2 service provider interface specification.

- a. Create a file with the name `com.ibm.task.spi.plug-in_nameNotificationEventHandlerPlugin`, where *plug-in_name* is the name of the plug-in.

For example, if your plug-in is called `HelpDeskRequest` (event handler name) and it implements the `com.ibm.task.spi.NotificationEventHandlerPlugin` interface, the name of the configuration file is `com.ibm.task.spi.HelpDeskRequestNotificationEventHandlerPlugin`.

- b. In the first line of the file that is neither a comment line nor a blank line, specify the fully qualified name of the plug-in class that you created in step 1.

For example, if your plug-in class is called `MyEventHandler` and it is in the `com.customer.plugins` package, then the first line of the configuration file must contain the following entry: `com.customer.plugins.MyEventHandler`.

Results

You have an installable JAR file that contains a plug-in that handles notification events and a service provider configuration file that can be used to load the plug-in. You can register API event handlers with a task instance, a task template, or an application component.

Tip: You only have one `eventName` property available to register both API event handlers and notification event handlers. If you want to use both an API event handler and a notification event handler, the plug-in implementations must have the same name, for example, `Customer` as the event handler name for the SPI implementation.

You can implement both plug-ins using a single class, or two separate classes. In both cases, you need to create two files in the `META-INF/services/` directory of your JAR file, for example, `com.ibm.task.spi.CustomerNotificationEventHandlerPlugin` and `com.ibm.task.spi.CustomerAPIEventHandlerPlugin`.

Package the plug-in implementation and the helper classes in a single JAR file.

What to do next

You now need to install and register the plug-in so that it is available to the human task container at runtime. You can register notification event handlers with a task instance, a task template, or an application component.

Creating plug-ins to post-process people query results

Staff resolution returns a list of the users that are assigned to a specific role, for example, potential owner of a task. You can create a plug-in to change the results of people queries returned by people resolution. For example, to improve workload balancing, you might have a plug-in that removes users from the query result who already have a high workload.

About this task

You can have only one post-processing plug-in; this means that the plug-in must handle the people query results from all tasks. Your plug-in can add or remove users, or change user or group information. It can also change the result type, for example, from a list of users to a group, or to everybody.

Because the plug-in runs after people resolution completes, any rules that you have to preserve confidentiality or security have already been applied. The plug-in receives information about users that have been removed during people resolution (in the HTM_REMOVED_USERS map key). You must ensure that your plug-in uses this context information to preserve any confidentiality or security rules you might have.

To implement post-processing of people query results, you use the `StaffQueryResultPostProcessorPlugin` interface. The interface has methods for modifying the query results for tasks, escalations, task templates, and application components.

Complete the following steps to create a plug-in to post-process people query results.

Procedure

1. Write a class that implements the `StaffQueryResultPostProcessorPlugin` interface.

You must implement all of the interface methods. This class can invoke methods of other classes.

This class runs in the context of a Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) application. Ensure that this class and its helper classes follow the EJB specification.

Tip: If you want to call the `HumanTaskManagerService` interface from this class, do not call a method that updates the task that produced the event. This action results in a database deadlock.

The following example shows how you might change the editor role of a task called `SpecialTask`.

```
public StaffQueryResult processStaffQueryResult
    (StaffQueryResult originalStaffQueryResult,
     Task task,
     int role,
     Map context)
{
    StaffQueryResult newStaffQueryResult = originalStaffQueryResult;
    StaffQueryResultFactory staffResultFactory =
        StaffQueryResultFactory.newInstance();
    if (role == com.ibm.task.api.WorkItem.REASON_EDITOR &&
        task.getName() != null &&
        task.getName().equals("SpecialTask"))
    {
        UserData user = staffResultFactory.newUserData
            ("SuperEditor",
             new Locale("en-US"),
             "SuperEditor@company.com");
        ArrayList userList = new ArrayList();
        userList.add(user);

        newStaffQueryResult = staffResultFactory.newStaffQueryResult(userList);
    }
    return(newStaffQueryResult);
}
```

2. Assemble the plug-in class and its helper classes into a JAR file.

If the helper classes are used by several J2EE applications, you can package these classes in a separate JAR file that you register as a shared library.

3. Create a service provider configuration file for the plug-in in the `META-INF/services/` directory of your JAR file.

The configuration file provides the mechanism for identifying and loading the plug-in. This file conforms to the Java 2 service provider interface specification.

- a. Create a file with the name `com.ibm.task.spi.plug-in_nameStaffQueryResultPostProcessorPlugin`, where *plug-in_name* is the name of the plug-in.

For example, if your plug-in is called `MyHandler` and it implements the `com.ibm.task.spi.StaffQueryResultPostProcessorPlugin` interface, the name of the configuration file is

`com.ibm.task.spi.MyHandlerStaffQueryResultPostProcessorPlugin`.

- b. In the first line of the file that is neither a comment line nor a blank line, specify the fully qualified name of the plug-in class that you created in step 1.

For example, if your plug-in class is called `StaffPostProcessor` and it is in the `com.customer.plugins` package, then the first line of the configuration file must contain the following entry:

`com.customer.plugins.StaffPostProcessor`. You have an installable JAR file that contains a plug-in that post processes people query results and a service provider configuration file that can be used to load the plug-in.

4. Install the plug-in.

You can have only one post-processing plug-in for people query results. You must install the plug-in as a shared library.

5. Register the plug-in.

- a. In the administrative console, go to the Custom Properties page of the Human Task Manager (**Application servers** → *server_name* → **Human task container** → **Custom properties**).

- b. Add a custom property with the name **Staff.PostProcessorPlugin**, and a value of the name that you gave to your plug-in, `MyHandler` in this example.

Installing plug-ins

To use a plug-in, you must install the plug-in so that it can be accessed by the task container.

About this task

The way in which you install the plug-in depends on whether the plug-in is to be used by only one Java 2 Enterprise Edition (J2EE) application, or several applications.

Complete one of the following steps to install a plug-in.

- Install a plug-in for use by a single J2EE application.

Add your plug-in JAR file to the application EAR file. In the deployment descriptor editor in WebSphere Integration Developer, install the JAR file for your plug-in as a project utility JAR file for the J2EE application of the main enterprise JavaBeans (EJB) module.

- Install a plug-in for use by several J2EE applications.

Put the JAR file in a WebSphere Application Server shared library and associate the library with the applications that need access to the plug-in. To make the JAR file available in a network deployment environment, distribute the JAR file on each server manually, and then install the shared library once for each cell.

What to do next

You can now register the plug-in.

Registering plug-ins

You can register your plug-ins on different levels in the task container artifact hierarchy. For example, for all tasks on a global level, for the tasks of an application component, for all of the tasks associated with a task template, or for a single task instance.

About this task

When you register multiple plug-ins, scoping is supported. This means that a plug-in that is registered on a lower level of the task container artifact hierarchy, such as a task instance, is used instead of the plug-in that is registered on a higher level, such as a task template or application component. Scoping is supported for all of the hierarchy levels. The task container uses the plug-in that is registered on the lowest level of the hierarchy.

You can register a plug-in in one of the following ways.

- Register the plug-in in the task model.
In the task editor in WebSphere Integration Developer in the Details page of the properties area for the task, specify the name of the event handler in the **Event handler name** field.
- Register the plug-in for ad-hoc tasks or task templates that you create at runtime.
Use the `setEventHandlerName` method of the `TTask` class to register the name of the event handler.
- Change the registered event handler for a task instance at runtime.
Use the `update(Task task)` method to use a different event handler for a task instance at runtime. The caller must have task administrator authority to update this property.
- Register the plug-in on a global level.
In the administration console on the Custom properties page for the human task container, define a custom property for the plug-in. The value of the custom property is the plug-in name.

Part 2. Deploying applications

Chapter 3. 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, version 6 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 5 on page 174 shows how an application contains components and related libraries.

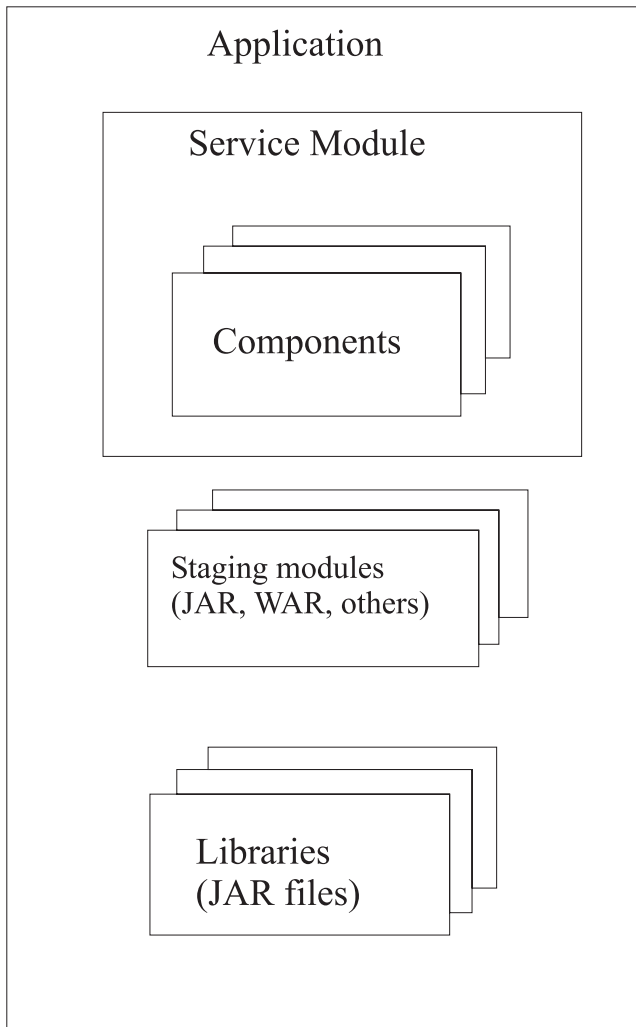


Figure 5. Relationship amongst module, component and library

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 Platform, Enterprise Edition 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 service 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.

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:

- *myServiceApp*

- *myServiceEJB*
- *myServiceEJBClient*
- *myServiceWeb*

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

Related reference

 [serviceDeploy command](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

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.

Related reference

 [serviceDeploy command](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

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 you begin

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

About this task

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.

Procedure

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.
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](#).

Related reference

 [serviceDeploy command](#)

Use the serviceDeploy command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through wsadmin.

Considerations for installing service applications on clusters

Installing a service application on a cluster places additional requirements on you. It is important that you keep these considerations in mind as you install any service applications on a cluster.

Clusters can provide many benefits to your processing environment by providing economies of scale to help you balance request workload across servers and provide a level of availability for clients of the applications. Consider the following before installing an application that contains services on a cluster:

- Will users of the application require the processing power and availability provided by clustering?
If so, clustering is the correct solution. Clustering will increase the availability and capacity of your applications.
- Is the cluster correctly prepared for service applications?
You must configure the cluster correctly before installing and starting the first application that contains a service. Failure to configure the cluster correctly prevents the applications from processing requests correctly.
- Does the cluster have a backup?
You must install the application on the backup cluster also.

Chapter 4. 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 you begin

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* in the Developing and Deploying Modules PDF and bring the files to the production system to deploy. See the information centers for WebSphere Integration Developer and WebSphere Application Server Network Deployment for more information.

Procedure

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 command using the JAR file from the previous step as input.
3. Install the EAR file from step 2. How you install the applications depends on whether you are installing the application on a stand alone server or a server in a cell.

Note: You can either use the administrative console or a script to install the application. See the WebSphere Application Server information center for additional information.

4. Save the configuration. The module is now installed as an application.
5. Start the application.

Results

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

What to do next

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

Related reference

 [serviceDeploy command](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

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 you begin

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](#) for more information.

About this task

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.

Procedure

1. Locate the JAR file that contains the module to deploy.
2. Issue the command using the JAR file from the previous step as input.
This step creates an EAR file.

Note: Perform the following steps at an administrative console.

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

Related reference

 [serviceDeploy command](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

Deploying applications using Apache Ant tasks

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

Before you begin

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 Apache Ant tasks.
- You understand the deployment process.

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

The reference portion of the information center for WebSphere Application Server Network Deployment contains a section on application programming interfaces. Apache 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`.

About this task

If you need to install multiple applications concurrently, develop an Apache Ant task before deployment. The Apache Ant task can then deploy and install the applications on the servers without your involvement in the process.

Procedure

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 Apache Ant task to run the ServiceDeploy command to create the EAR file for each server.
5. Create an Apache Ant task to run the InstallApplication command for each EAR file from step 4 on the applicable servers.
6. Run the ServiceDeploy Apache Ant task to create the EAR file for the applications.
7. Run the InstallApplication Apache Ant task to install the EAR files from step 6.

Results

The applications are correctly deployed on the target servers.

Example of deploying an application unattended

This example shows an Apache 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 Apache Ant task.

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

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

What to do next

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

Related reference

 [serviceDeploy command](#)

Use the serviceDeploy command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through wsadmin.

Chapter 5. Installing business process and human task applications

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

Before you begin

Verify that Business Flow Manager, Human Task Manager, or both are installed and configured for each application server or cluster on which you want to install your application.

About this task

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

Results

After a business process or human task application is installed, all of the business process templates and human task templates are put into the start state. You can create process instances and task instances from these templates.

What to do next

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

Related concepts

“Deployment of business processes and human tasks” on page 184

When WebSphere Integration Developer or service deployment generates the deployment code for your process or task, each process component or task component is mapped to one session enterprise bean. All deployment code is packaged in the enterprise application (EAR) file. Additionally, for each process, a Java class which represents Java code in this process is generated and embedded in the EAR file during installation of the enterprise application. Each new version of a model that is to be deployed must be packaged in a new enterprise application.

“How business process and human task applications are installed in a network deployment environment”

When process templates or human task templates are installed in a network deployment environment, the following actions are performed automatically by the application installation.

How business process and human task applications are installed in a network deployment environment

When process templates or human task templates are installed in a network deployment environment, the following actions are performed automatically by the application installation.

The application is installed asynchronously in stages. Each stage must complete successfully before the following stage can begin.

1. The application installation starts on the deployment manager.
During this stage, the business process templates and human task templates are configured in the WebSphere configuration repository. The application is also validated. If errors occur, they are reported in the System.out file, in the System.err file, or as FFDC entries on the deployment manager.
2. The application installation continues on the node agent.
During this stage, the installation of the application on one application server instance is triggered. This application server instance is either part of, or is, the deployment target. If the deployment target is a cluster with multiple cluster members, the server instance is chosen arbitrarily from the cluster members of this cluster. If errors occur during this stage, they are reported in the SystemOut.log file, in the SystemErr.log file, or as FFDC entries on the node agent.
3. The application runs on the server instance.
During this stage, the process templates and human templates are deployed to the Business Process Choreographer database on the deployment target. If errors occur, they are reported in the System.out file, in the SystemErr.log file, or as FFDC entries on this server instance.

Related tasks

Chapter 5, “Installing business process and human task applications,” on page 183

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

Deployment of business processes and human tasks

When WebSphere Integration Developer or service deployment generates the deployment code for your process or task, each process component or task component is mapped to one session enterprise bean. All deployment code is packaged in the enterprise application (EAR) file. Additionally, for each process, a Java class which represents Java code in this process is generated and embedded in the EAR file during installation of the enterprise application. Each new version of a model that is to be deployed must be packaged in a new enterprise application.

When you install an enterprise application that contains business processes or human tasks, then these are stored as business process templates or human task templates, as appropriate, in the Business Process Choreographer database. 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.

You can deploy many different versions of a process template or task template, each in a different enterprise application. When you install a new enterprise application, the version of the template that is installed is determined as follows:

- If the name of the template and the target namespace do not already exist, a new template is installed
- If the template name and target namespace are the same as those of an existing template, but the valid-from date is different, a new version of an existing template is installed

Note: The template name is derived from the name of the component and not from the business process or human task.

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

- If you use WebSphere Integration Developer, the valid-from date is the date on which the human task or the business process was modeled.
- If you use service deployment, the valid-from date is the date on which the serviceDeploy command was run. Only collaboration tasks get the date on which the application was installed as the valid-from date.

Related tasks

Chapter 5, “Installing business process and human task applications,” on page 183

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

Installing business process and human task applications interactively

You can install an application interactively at runtime using the wsadmin tool and the installInteractive script. You can use this script to change settings that cannot be changed if you use the administrative console to install the application.

About this task

Perform the following steps to install business process applications interactively.

Procedure

1. Start the wsadmin tool.

In the *profile_root/bin* directory, enter wsadmin.

2. Install the application.

At the wsadmin command-line prompt, enter the following command:

```
$AdminApp installInteractive application.ear
```

where *application.ear* is the qualified name of the enterprise archive file that contains your process application. You are prompted through a series of tasks where you can change values for the application.

3. Save the configuration changes.

At the wsadmin command-line prompt, enter the following command:

```
$AdminConfig save
```

You must save your changes to transfer the updates to the master configuration repository. If a scripting process ends and you have not saved your changes, the changes are discarded.

Configuring process application data source and set reference settings

You might need to configure process applications that run SQL statements for the specific database infrastructure. These SQL statements can come from information service activities or they can be statements that you run during process installation or instance startup.

About this task

When you install the application, you can specify the following types of data sources:

- Data sources to run SQL statements during process installation
- Data sources to run SQL statements during the startup of a process instance
- Data sources to run SQL snippet activities

The data source required to run an SQL snippet activity is defined in a BPEL variable of type `tDataSource`. The database schema and table names that are required by an SQL snippet activity are defined in BPEL variables of type `tSetReference`. You can configure the initial values of both of these variables.

You can use the `wsadmin` tool to specify the data sources.

Procedure

1. Install the process application interactively using the `wsadmin` tool.
2. Step through the tasks until you come to the tasks for updating data sources and set references.

Configure these settings for your environment. The following example shows the settings that you can change for each of these tasks.

3. Save your changes.

Example: Updating data sources and set references, using the `wsadmin` tool

In the **Updating data sources** task, you can change data source values for initial variable values and statements that are used during installation of the process or when the process starts. In the **Updating set references** task, you can configure the settings related to the database schema and the table names.

Task [24]: Updating data sources

```
//Change data source values for initial variable values at process start
```

```
Process name: Test
// Name of the process template
Process start or installation time: Process start
// Indicates whether the specified value is evaluated
//at process startup or process installation
Statement or variable: Variable
// Indicates that a data source variable is to be changed
Data source name: MyDataSource
// Name of the variable
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name to jdbc/newName
```

Task [25]: Updating set references

```
// Change set reference values that are used as initial values for BPEL variables
```

```
Process name: Test
// Name of the process template
Variable: SetRef
// The BPEL variable name
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name of the data source of the set reference to jdbc/newName
Schema name: [IISAMPLE]
// The name of the database schema
Schema prefix: []:
// The schema name prefix.
// This setting applies only if the schema name is generated.
Table name: [SETREFTAB]: NEWTABLE
```

```
// Sets the name of the database table to NEWTABLE
Table prefix: []:
// The table name prefix.
// This setting applies only if the prefix name is generated.
```

Uninstalling business process and human task applications, using the administrative console

You can use the administrative console to uninstall applications that contain business processes or human tasks.

Before you begin

To uninstall an application that contains business processes or human tasks, the following prerequisites must be met:

- If the application is installed on a stand-alone server, the server must be running and have access to the Business Process Choreographer database.
- If the application is installed on a cluster, the deployment manager and at least one cluster member must be running. The cluster member have access to the Business Process Choreographer database.
- If the application is installed on managed server, the deployment manager and this server must be running. The server must have access to the Business Process Choreographer database.
- All of the business process templates and human task templates that belong to the application must be in the stopped state.
- There are no instances of business process or human task templates present in any state.
-

For stand-alone server environments that are used as development and unit test environments, the server can be configured to run in development mode. This configuration does not require that the templates be stopped and no instances be present. However, this configuration is not valid for production environments.

About this task

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

Procedure

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

This action prevents the creation of process and task instances.

- a. Click **Applications** → **SCA modules** in the administrative console navigation pane.
- b. Select the module that contains the templates that you want to stop.
- c. Under Additional Properties, click **Business Processes** or **Human Tasks**, or both, as appropriate.
- d. Select all process and task templates by clicking the appropriate check box.
- e. Click **Stop**.

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

2. Verify that the database, at least one application server for each cluster, and the stand-alone server where the application is deployed are running.

In a network deployment environment, the deployment manager, all managed stand-alone application servers, and at least one application server must be running for each cluster where the application is installed.

3. Verify that the application has no business process instances or human task instances.

If necessary, an administrator can use Business Process Choreographer Explorer to delete any process or task 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 or task instances still exist in the application.
 - c. Select again the application that you want to uninstall, and click **Uninstall**.
 - d. Click **Save** to save your changes.

Results

The application is uninstalled.

Uninstalling business process and human task applications, using administrative commands

Administrative commands provide an alternative to the administrative console for uninstalling applications that contain business processes or human tasks.

Before you begin

To uninstall an application that contains business processes or human tasks, the following prerequisites must be met:

- If the application is installed on a stand-alone server, the server must be running and have access to the Business Process Choreographer database.
- If the application is installed on a cluster, the deployment manager and at least one cluster member must be running. The cluster member have access to the Business Process Choreographer database.
- If the application is installed on managed server, the deployment manager and this server must be running. The server must have access to the Business Process Choreographer database.
- All of the business process templates and human task templates that belong to the application must be in the stopped state.
- There are no instances of business process or human task templates present in any state.
-

For stand-alone server environments that are used as development and unit test environments, the server can be configured to run in development mode. This configuration does not require that the templates be stopped and no instances be present. However, this configuration is not valid for production environments.

In addition, if global security is enabled, verify that your user ID has operator authorization.

Ensure that the server process to which the administration client connects is running. To ensure that the administrative client automatically connects to the server process, do not use the `-conntype NONE` option as a command option.

About this task

The following steps describe how to use the `bpcTemplates.jacl` script to uninstall applications that contain business process templates or human task templates. You must stop a template before you can uninstall the application to which it belongs. You can use the `bpcTemplates.jacl` script to stop and uninstall templates in one step.

Before you uninstall applications, you can delete process instances or task instances associated with the templates in the applications, for example, using Business Process Choreographer Explorer. You can also use the `-force` option with the `bpcTemplates.jacl` script to delete any instances associated with the templates, stop the templates, and uninstall them in one step.

CAUTION:

Because the `-force` option deletes all process instance and task instance data, you should use this option with care.

Procedure

1. Change to the Business Process Choreographer samples directory.

On Windows platforms, enter:

```
cd install_root\ProcessChoreographer\admin
```

On Linux, UNIX, and i5/OS platforms, enter:

```
cd install_root/ProcessChoreographer/admin
```

2. Stop the templates and uninstall the corresponding application.

On Windows platforms, enter:

```
install_root\bin\wsadmin -f bpcTemplates.jacl  
                        [-user user_name]  
                        [-password user_password]  
                        -uninstall application_name  
                        [-force]
```

On Linux, UNIX, and i5/OS platforms, enter:

```
install_root/bin/wsadmin -f bpcTemplates.jacl  
                        [-user user_name]  
                        [-password user_password]  
                        -uninstall application_name  
                        [-force]
```

Where:

user_name

If global security is enabled, provide the user ID for authentication.

user_password

If global security is enabled, provide the user password for authentication.

application_name

If global security is enabled, provide the user password for authentication.

Results

The application is uninstalled.

Chapter 6. Installing adapters

Adapters allow your application to communicate with other components of your enterprise information system.

The process you use to install adapters is described in *Configuring and using adapters* in the WebSphere Integration Developer information center.

Chapter 7. Installing EIS applications

An EIS application module can be deployed to a J2EE platform. The deployment 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.

About this task

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

Table 42. 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.	<ul style="list-style-type: none">• ConnectionFactory• ActivationSpec• Destinations
JMS Export	Message Driven Bean provided by the runtime is deployed, resource references generated on the module Session EJB	<ul style="list-style-type: none">• 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.

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.

Before you begin

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

About 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.

Before you begin

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

About this task

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

Table 43. 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.	<ul style="list-style-type: none">• ConnectionFactory• ActivationSpec• Destinations
JMS Export	Message Driven Bean provided by the runtime is deployed, resource references generated on the module Session EJB	<ul style="list-style-type: none">• 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.

Chapter 8. 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.

Before you begin

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.

About this task

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.

Procedure

1. Determine if the application installation failed.

Examine the SystemOut.log 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.

Important: If the installation has failed and the application contains services, you must remove any SIBus destinations or J2C activation specifications created prior to the failure before attempting to reinstall the application. The simplest way to remove these artifacts is to click **Save > Discard all** after the failure. If you inadvertently save the changes, you must manually remove the SIBus destinations and J2C activation specifications (see *Deleting SIBus destinations* and *Deleting J2C activation specifications* in the *Administering* section).

2. If the application is installed correctly, examine it to determine if it started successfully.

If the application did not start successfully, the failure occurred when the server attempted to initiate the resources for the application.

- a. Examine the SystemOut.log file for messages that will direct you on how to proceed.
- b. Determine if resources required by the application are available and/or have started successfully.

Resources that are not started prevent an application from running. This protects 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 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” 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 on page 197.
 - a. Determine if the application uses any services contained in another application. Make sure that the other application is installed and has started successfully.
 - b. Determine if the import and export bindings for devices contained in other applications used by the failing application are configured correctly. Use the administrative console to examine and correct the bindings.
4. Correct the problem and restart the application.

Deleting J2C activation specifications

The system builds J2C application specifications when installing an application that contains services. There are occasions when you must delete these specifications before reinstalling the application.

Before you begin

If you are deleting the specification because of a failed application installation, make sure the module in the Java Naming and Directory Interface (JNDI) name matches the name of the module that failed to install. The second part of the JNDI name is the name of the module that implemented the destination. For example in `sca/SimpleBOCrsmA/ActivationSpec`, **SimpleBOCrsmA** is the module name.

Required security role for this task: When security and role-based authorization are enabled, you must be logged in as administrator or configurator to perform this task.

About this task

Delete J2C activation specifications when you inadvertently saved a configuration after installing an application that contains services and do not require the specifications.

Procedure

1. Locate the activation specification to delete.
The specifications are contained in the resource adapter panel. Navigate to this panel by clicking **Resources > Resource adapters**.
 - a. Locate the **Platform Messaging Component SPI Resource Adapter**.
To locate this adapter, you must be at the **node** scope for a standalone server or at the **server** scope in a deployment environment.
2. Display the J2C activation specifications associated with the Platform Messaging Component SPI Resource Adapter.
Click on the resource adapter name and the next panel displays the associated specifications.

3. Delete all of the specifications with a **JNDI Name** that matches the module name that you are deleting.
 - a. Click the check box next to the appropriate specifications.
 - b. Click **Delete**.

Results

The system removes selected specifications from the display.

What to do next

Save the changes.

Deleting SIBus destinations

SIBus destinations are the connections that make services available to applications. There will be times that you will have to remove destinations.

Before you begin

If you are deleting the destination because of a failed application installation, make sure the module in the destination name matches the name of the module that failed to install. The second part of the destination is the name of the module that implemented the destination. For example in `sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer`, **SimpleBOCrsmA** is the module name.

Required security role for this task: When security and role-based authorization are enabled, you must be logged in as administrator or configurator to perform this task.

About this task

Delete SIBus destinations when you inadvertently saved a configuration after installing an application that contains services or you no longer need the destinations.

Note: This task deletes the destination from the SCA system bus only. You must remove the entries from the application bus also before reinstalling an application that contains services (see Deleting J2C activation specifications in the Administering section of this information center).

Procedure

1. Log into the administrative console.
2. Display the destinations on the SCA system bus.

Navigate to the panel by clicking **Service integration > Buses**
3. Select the SCA system bus destinations.

In the display, click on **SCA.SYSTEM.cellname.Bus**, where *cellname* is the name of the cell that contains the module with the destinations you are deleting.
4. Delete the destinations that contain a module name that matches the module that you are removing.
 - a. Click on the check box next to the pertinent destinations.
 - b. Click **Delete**.

Results

The panel displays only the remaining destinations.

What to do next

Delete the J2C activation specifications related to the module that created these destinations.

Part 3. Appendixes

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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
577 Airport Blvd., Suite 800
Burlingame, CA 94010
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows: (c) (your company name) (year). Portions of

this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

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

IBM, the IBM logo, developerWorks, WebSphere, and z/OS are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

This product includes software developed by the Eclipse Project (<http://www.eclipse.org>).



IBM WebSphere Process Server for Multiplatforms, Version 6.1.0



Printed in USA