WebSphere® Process Server for z/OS

IBM

Version 6.0.1

Developing and Deploying Modules

Note

Before using this information, be sure to read the general information in “Notices” on page 121.

23 June 2006

This edition applies to version 6, release 0, modification 1 of WebSphere Process Server for z/OS (product number 5655-N53) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

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

Contents

Developing and deploying modules

Developing and deploying modules are fundamental tasks.

WebSphere Process Server documentation (in PDF format)

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

Overview of developing modules

A module is a basic deployment unit for a WebSphere Process Server application. A module contains one or more component libraries and staging modules used by the application. A component may reference other 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.

6.0.1+ 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 for z/OS 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.

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

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.

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.

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

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.

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 4.

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);
}
```

Invoke the service.

Invoking components

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

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

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.

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;
 }
}
```

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

Dynamically invoking a component:

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

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

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

interface this way.

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

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

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

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/wsdl/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);
```

Considerations when invoking services on different servers

One of the benefits of Service Oriented Architecture is the ability for consumers to use services that already exist in other service modules. To balance the workload equitably, you may install applications on different servers within a cell and those applications may reside on different physical servers.

One of the advantages of WebSphere Process Server is the ability to distribute the application workload across multiple servers in a cell. This distribution allows for better workload balancing amongst the various servers within the cell and maximizes the maintainability of the computing resources because there is only one copy of an application or service within the server. Thus, an application on server A may require a service installed in server B within the cell. To use services in this manner, you must configure communications between the servers. The type of configuration you perform depends on whether the calling service component invokes the service asynchronously or synchronously.

Related topics describe how to configure the systems for both asynchronous and synchronous invocations.

Configuring servers to invoke services asynchronously:

To enable service components on different servers to communicate, you have to configure the servers similarly. This topic describes the configuration you perform to enable the communication for applications that asynchronously invoke services on a different server.

The task assumes that you have already installed WebSphere Process Server on the systems for which you are configuring the communications but have not yet installed the applications involved. You are using an administrative console that can examine and change the configuration for both servers involved.

Before installing an application that requires the services of a service component installed on another system, you must configure the systems so they can communicate the requests. For service modules that use asynchronous invocations, the process involves foreign buses and Service Integration Bus (SIBus) mediations.

Note: For the purposes of this task, the invoking service module resides on system A and the target resides on system B.

For the purposes of this task, Figure 1 on page 9 contains the information to use in the configuration.

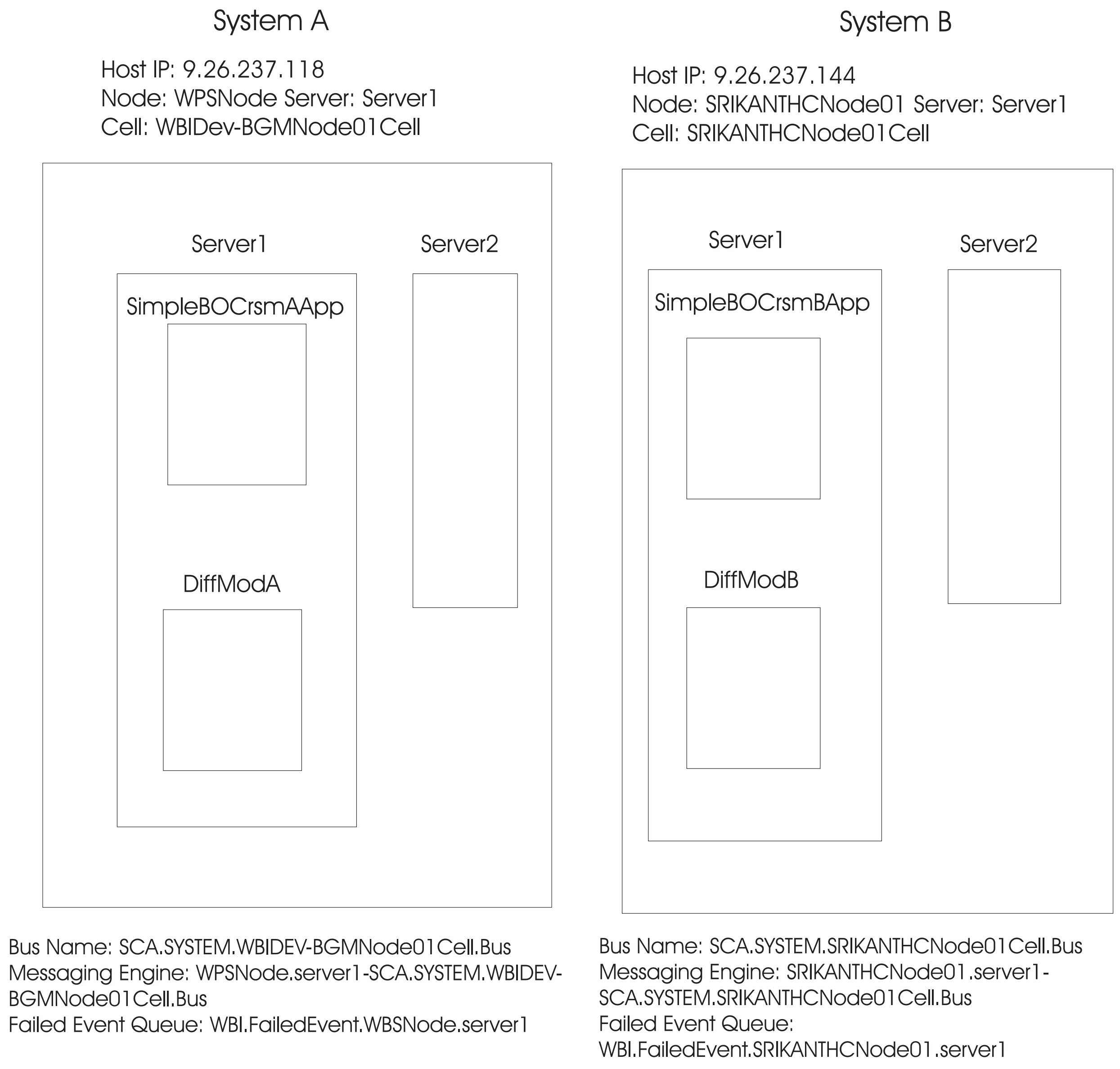

System A
Host IP: 9.26.237.118
Node: WPSNode Server: Server1
Cell: WBIDev-BGMNode01Cell
Server1
Server2
SimpleBOCrsmAApp
DiffModA
Bus Name: SCA.SYSTEM.WBIDEV-BGMNode01Cell.Bus
Messaging Engine: WPSNode.server1-SCA.SYSTEM.WBIDEV-BGMNode01Cell.Bus
Failed Event Queue: WBI.FailedEvent.WBSNode.server1
System B
Host IP: 9.26.237.144
Node: SRIKANTHCNode01 Server: Server1
Cell: SRIKANTHCNode01Cell
Server1
Server2
SimpleBOCrsmBApp
DiffModB
Bus Name: SCA.SYSTEM.SRIKANTHCNode01Cell.Bus
Messaging Engine: SRIKANTHCNode01.server1-SCA.SYSTEM.SRIKANTHCNode01Cell.Bus
Failed Event Queue: WBI.FailedEvent.SRIKANTHCNode01.server1

Figure 1. Invoking a service on a different system

Note: For simplicity, only the servers involved in this communication in each cell is shown and each server resides on a different physical machine.

1. Collect information about each server involved in the communication. You will need the following information for both the originator and target servers:
   - Host IP address
   - Cell
   - Node
   - Server
   - Bus name
   - Messaging engine
   - Failed Event Queue name
2. Install the applications.

3. Create a foreign bus on each server pointing to the other server and set the routing definition type to `Direct, service integration bus link`.

   See "Adding a foreign bus" in the WebSphere Application Server Network Deployment, version 6 information center for more information.

   From the example, the foreign bus on System A would be SCA.SYSTEM.SRIKANTHCNode01Cell.Bus. The foreign bus on System B would be SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus.
4. Set up an SIB mediation link on each server pointing to the messaging engine on the other server.

   See "Adding a service integration bus link" in the WebSphere Application Server Network Deployment, version 6 information center for more information.

   From the example, the SIB mediation link on System A would be:

   ```
   SIB Link: TestCrossCell
   Remote ME: SRIKANTHCNode01.server1-SCA.SYSTEM.SRIKANTHCNode01Cell.Bus
   Bootstrap: 9.26.237.144:7277:BootstrapBasicMessaging
   ```

   The SIB mediation link on System B would be:

   ```
   SIB Link: TestCrossCell
   Remote ME: WPSNode.server1.SCA.SYSTEM.WBIDev-BGMNode01.Cell.Bus
   Bootstrap: 9.26.237.118:7276:BootstrapBasicMessaging
   ```

   Attention: The port number in the bootstrap is the SIB endpoint address port. If you enabled security, you must use the secure SIB endpoint address port.
5. Synchronize the SIB mediation links by restarting the servers.

   You should see messages similar to: `[8/24/05 11:00:09:741 PDT] 00000086 SibMessage I [SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus:WPSNode.server1-SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus] CWSIP0382I: messaging engine 2D7333574B0CD70B responded to subscription request, Publish Subscribe topology now consistent.`
6. Display the destinations for each service module.
7. Modify the forwarding path of outgoing destinations of the invoking service module that must be wired to targets on the other system.

   The destination to wire will have `importlink` in the destination name, for example on System A the destination would be sca/SimpleBoCrsmA/importlink/test/sca/cros/simple/custinfo/CustomerInfo. Modify the path by prefixing the foreign bus name to the destination name. From the example, the foreign bus name for the second system is SCA.SYSTEM.SRIKANTHCNode01Cell.Bus. The result is

   ```
   SCA.SYSTEM.SRIKANTHCNode01Cell.Bus:sca/SimpleBoCrsmA/importlink/
   test/sca/cros/simple/custinfo/CustomerInfo
   ```
8. Create two destinations on the target server and configure them to point back to the invoking service module on the other server.

   From the example, on System B you would create:

   ```
   sca/SimpleBOCrsmA/import/test/sca/cros/simple/custinfo/CustomerInfo
   sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer
   ```

   Then set the forwarding paths to point to their counterparts on invoking server. This would look like:

   ```
   SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus:
   sca/SimpleBOCrsmA/import/test/sca/cros/simple/custinfo/CustomerInfo
   SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus:
   sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer
   ```

   a. Set the exception destination to the Failed Event queue for both of the destinations you created.

From the example, the value would be:`WBI.FailedEventSRIKANTHCNode01.server1`.

9. **Optional:** Add sender roles to the foreign buses, if you enabled security on the systems. Make sure to define the user each application uses on both systems from the operating system command prompt. The command to add the role is:

   ```
   wsadmin $AdminTask addUserToForeignBusRole -bus busName
     -foreignBus foreignBusName -role roleName -user userName
   ```

   Where:

   busName
   : Is the name of the bus on the system you enter the command.

   foreignBusName
   : Is the foreign bus to which you are adding the user.

   userName
   : Is the userid to add to the foreign bus.

Start the applications.

Configuring servers to invoke services synchronously:

When a service component invokes another service component synchronously, you must configure the invoking service component to point to the system running the target so the target service can communicate results to the invoking service component.

The task assumes that you have already installed WebSphere Process Server on the systems for which you are configuring the communications but have not yet installed the applications involved. You are using an administrative console that can examine and change the configuration for both servers involved.

A service component invoking another service synchronously can communicate with the target only by configuring the export Java Naming and Directory Interface (JNDI) name on the target system to a JNDI name on the invoking system.

Note: For the purposes of this task, the invoking service module resides on system A and the target resides on system B.

For the purposes of this task, Figure 2 on page 12 contains the information to use in the configuration.

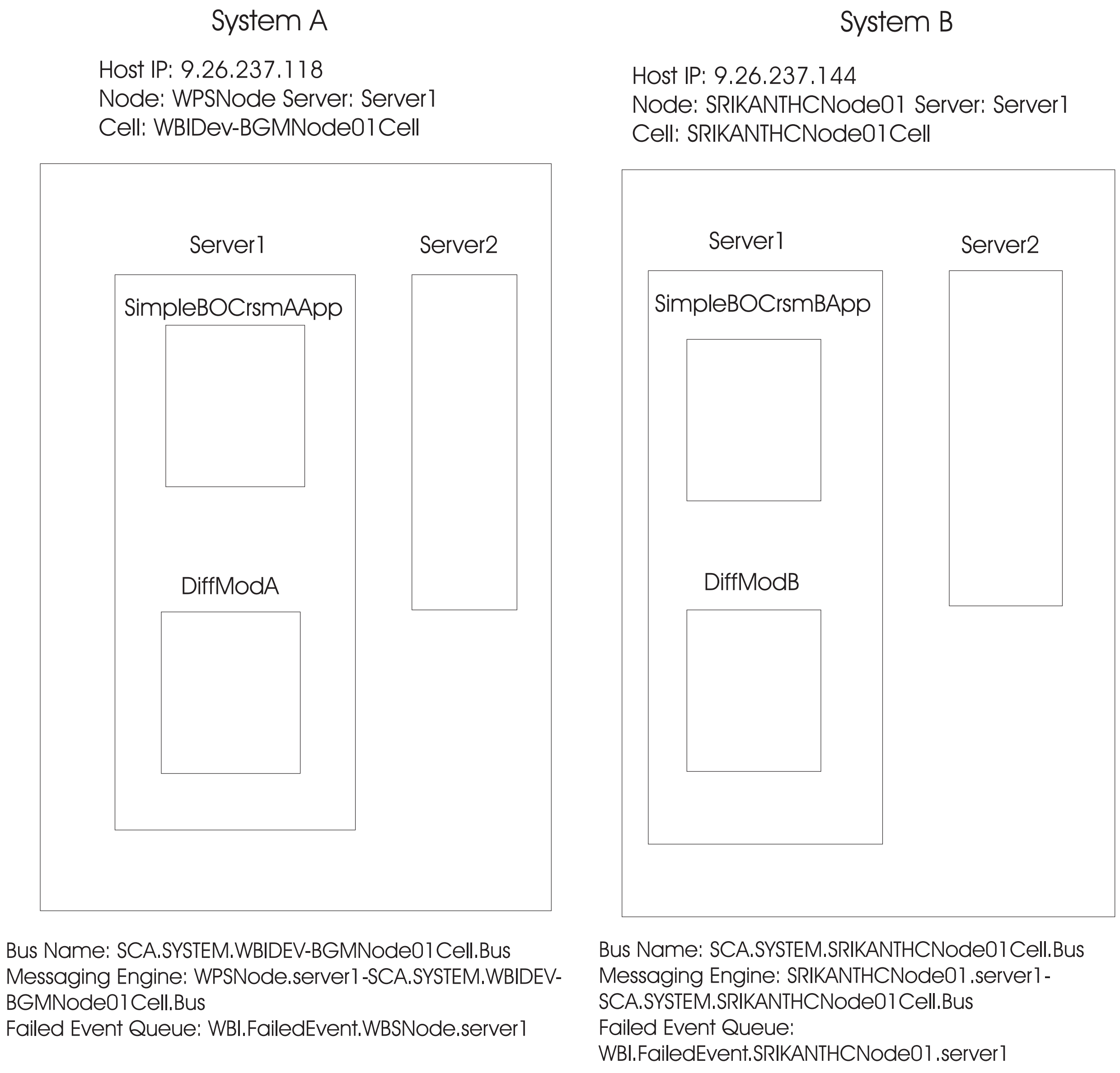

System A
Host IP: 9.26.237.118
Node: WPSNode Server: Server1
Cell: WBIDev-BGMNode01Cell
Server1
Server2
SimpleBOCrsmAApp
DiffModA
Bus Name: SCA.SYSTEM.WBIDEV-BGMNode01Cell.Bus
Messaging Engine: WPSNode.server1-SCA.SYSTEM.WBIDEV-BGMNode01Cell.Bus
Failed Event Queue: WBI.FailedEvent.WBSNode.server1
System B
Host IP: 9.26.237.144
Node: SRIKANTHCNode01 Server: Server1
Cell: SRIKANTHCNode01Cell
Server1
Server2
SimpleBOCrsmBApp
DiffModB
Bus Name: SCA.SYSTEM.SRIKANTHCNode01Cell.Bus
Messaging Engine: SRIKANTHCNode01.server1-SCA.SYSTEM.SRIKANTHCNode01Cell.Bus
Failed Event Queue: WBI.FailedEvent.SRIKANTHCNode01.server1

Figure 2. Invoking a service on a different system

Note: For simplicity, only the servers involved in this communication in each cell is shown and each server resides on a different physical machine.

1. Install the applications on each server.
2. Create a new namespace binding on the invoking system (System A, in the example) pointing to the export on the target system.

   On the **Name Space Bindings** panel, select a scope of Cell and click **Apply**. With the changed scope, click **New** in the display to create the new binding.

   In the wizard, specify the following (the values are appropriate for the example configuration):

   a. Binding type is CORBA
   b. The basic properties are:
      - Binding identifier is a unique string, in our example: `sca_import_test_sca_cross_simple_custinfo_CustomerInfo`

- Name in Name space is the JNDI name of the enterprise Java bean (EJB) you are invoking on the target system, for example:

  ```
  sca/SimpleBOCrsmB/export/test/sca/cros/simple/custinfo/CustomerInfo
  ```

  This names the export interface on the target system.
- Corbaname URL is the IP address and port number of the naming service on the target system, for example:

  ```
  corbaname:iiop:9.26.237.144:2809/NameServiceServerRoot#sca/
    impleBOCrsmB/export/test/sca/cros/simple/custinfo/CustomerInfo
  ```

  When finished, click **Next** and verify the values on the **Summary** page. After verifying, click **Finish**.

  Your system displays your new binding.
3. Save your changes by clicking **Save**.

Start the applications. The service component on System A can now synchronously invoke the service on System B.

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.

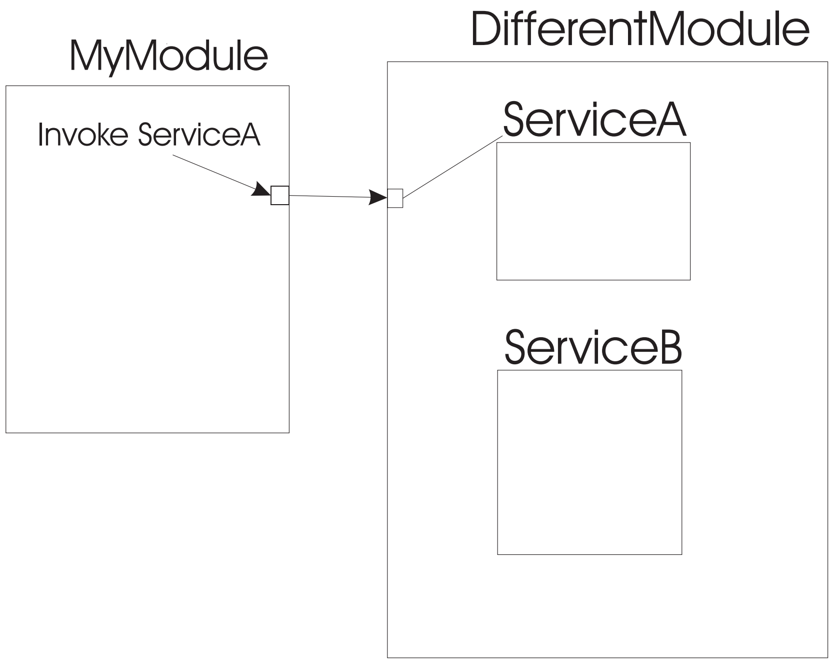


Figure 3. Simple invocation model

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 . APPA, APPB, and APPC all invoke CalculateFinalCost.

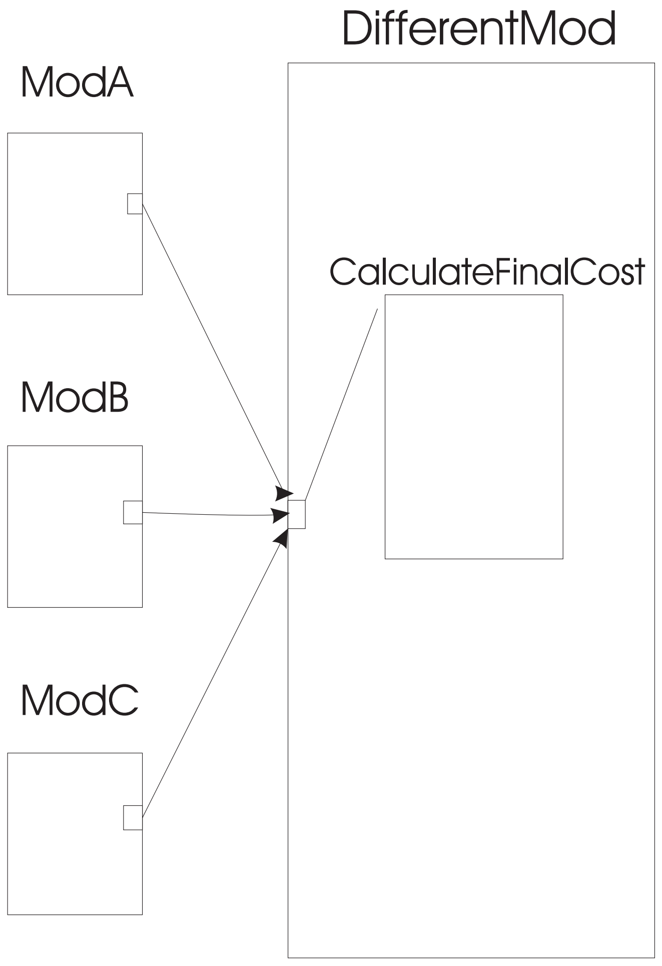


Figure 4. Multiple applications invoking a single service

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).

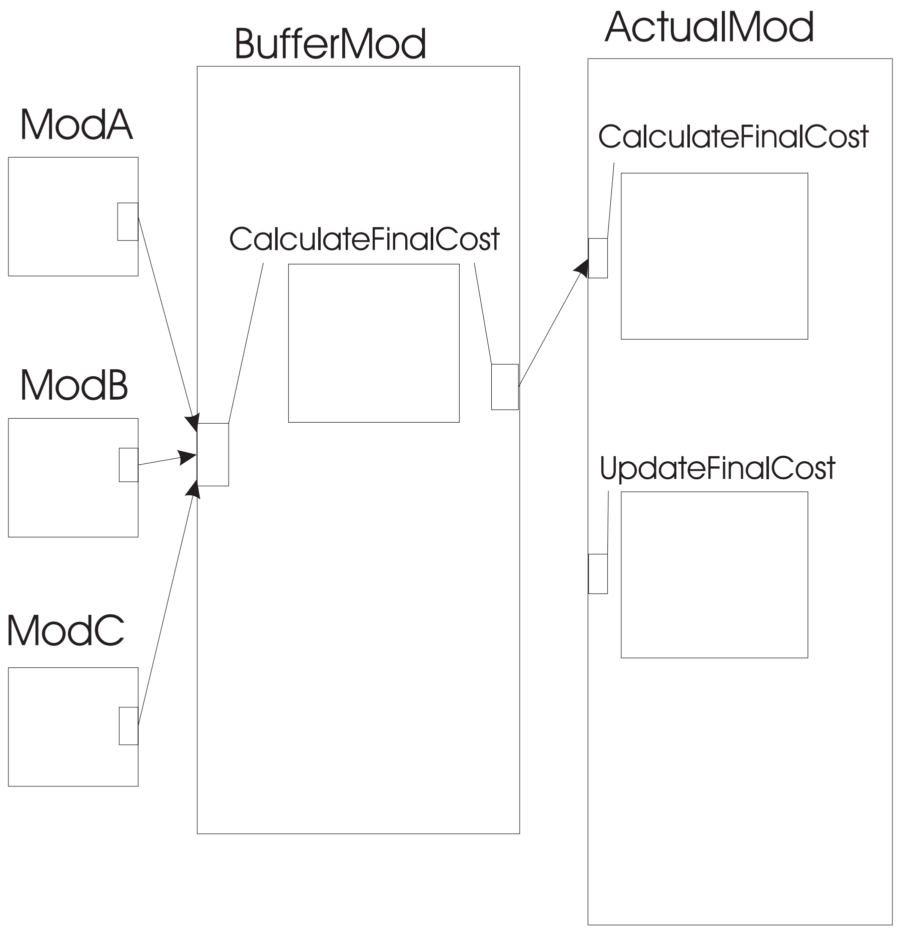


Figure 5. Isolated invocation model invoking UpdateCalculateFinal

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).

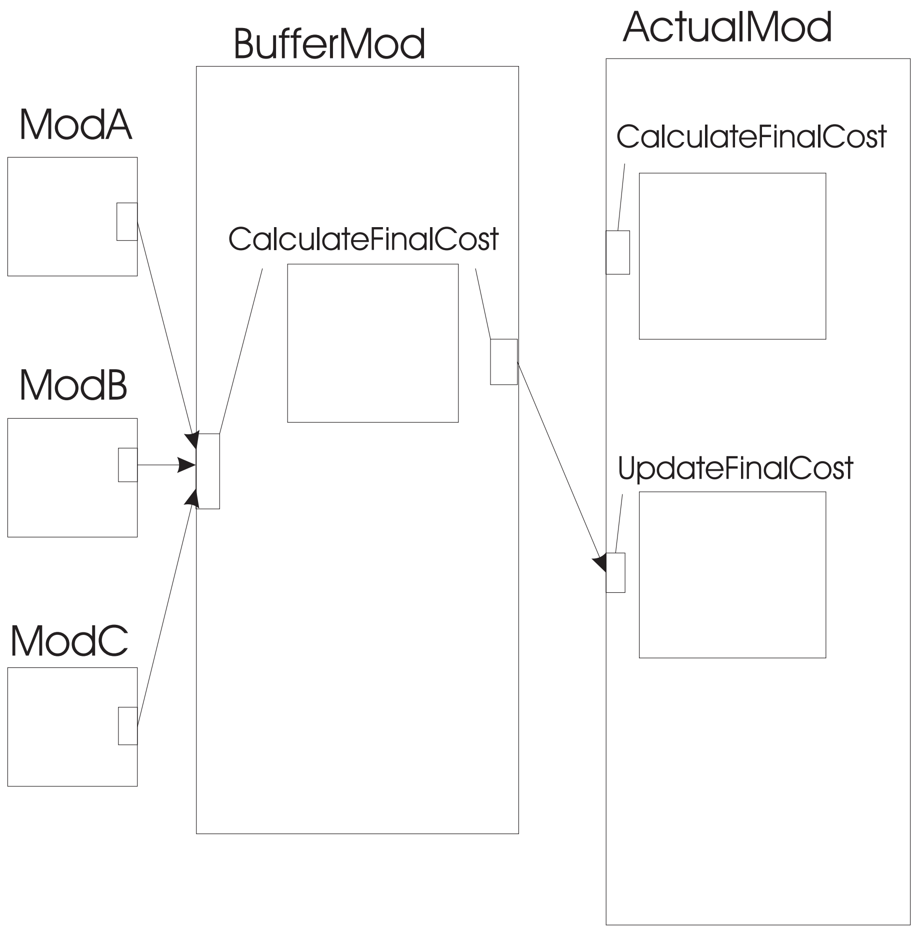


Figure 6. Isolated invocation model invoking UpdatedCalculateFinal

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.

Developing applications for business processes and tasks

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

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

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

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

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

   Examples for typical business process and human task functionality are provided.
2. Decide which of the Business Choreographer APIs you are going to use.

   Depending on the scenarios that you want to implement with your application, you can use one, or both, of the session beans.
3. Determine the authorization authorities needed by users of the application.

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

   The Javadoc contains authorization information for each of the methods. Choose the methods that best fit the users of your application.
4. Decide how to render the application.

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

Accessing the generic APIs

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

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

You can access the generic APIs using either the remote session bean or the local session bean.

Accessing the remote session bean

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

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

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

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

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

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

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

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

2. Package the generated stubs with your application.

   If your application runs on a different Java Virtual Machine (JVM) from the one where the BPEContainer application or the TaskContainer application runs, complete the following actions:

   a. For process applications, package the `<`*`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 **Class-Path** parameter in the manifest file of the application module to include the JAR file. The application module can be a J2EE application, a Web application, or an EJB application.

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

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

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

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

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

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

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

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

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

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

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

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

   // Begin a transaction
   transaction.begin();

     // Applications calls ...

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

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

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

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

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

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

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

Accessing the local session bean

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

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

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

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

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

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

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

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

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

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

  // Lookup the local home interface of the LocalBusinessFlowManager bean

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

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

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

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

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

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

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

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

   // Begin a transaction
   transaction.begin();

     // Applications calls ...

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

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

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

  // Lookup the local home interface of the LocalHumanTaskManager bean

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

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

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

Developing applications for business processes

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

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

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

Authorization roles for business processes

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.

A role is a group of employees who share the same level of authority. Java 2 Platform, Enterprise Edition (J2EE) roles are set up when the business process container is configured. Instance-based roles are assigned to processes and activities when the process is modeled. Role-based authorization requires that global security is enabled in WebSphere Application Server.

J2EE roles

The following J2EE roles are supported:

- J2EE BPESystemAdministrator. Users assigned to this role have all privileges.
- J2EE BPESystemMonitor. Users assigned to this role can view the properties of all business process objects.

You can use the administrative console to change the assignment of users and groups to these roles.

Setting up Roles using RACF security: These RACF permissions apply when the following security fields are specified:

- **com.ibm.security.SAF.authorization= true**

```
RDEFINE EJBROLE BPESystemAdministrator  UACC(NONE)
PERMIT BPESystemAdministrator CLASS(EJBROLE)  ID(userid) ACCESS(READ)
```

```
RDEFINE EJBROLE BPESystemMonitor  UACC(NONE)
PERMIT BPESystemMonitor  CLASS(EJBROLE)  ID(userid) ACCESS(READ)
```

- **com.ibm.security.SAF.delegation= true**

  ```
  RDEFINE EJBROLE JMSAPIUser  UACC(NONE) APPLDATA(' userid')
  ```

You can use Security Authorization Facility (SAF)-based authorization (for example, using the RACF EJBROLE profile) to control access by a client to Java 2 Platform, Enterprise Edition (J2EE) roles in EJB and Enterprise applications, including the Business process container. For more information on using SAF, see System Authorization Facility for role-based authorization in the WebSphere Application Server for z/OS information center.

Instance-based roles

A process instance or an activity is not assigned directly to a staff member in the process model, instead it is assigned to one of the available roles. Any staff member that is assigned to an instance-based role can perform the actions for that role. The association of users to instance-based roles is determined at runtime using staff resolution.

The following instance-based roles are supported:

- For processes: reader, starter, administrator
- For activities: reader, editor, potential owner, owner, administrator

These roles are authorized to perform the following actions:

Role	Authorized actions
Activity reader	View the properties of the associated activity instance, and its input and output messages.
Activity editor	Actions that are authorized for the activity reader, and write access to messages and other data associated with the activity.
Potential activity owner	Actions that are authorized for the activity reader. Members of this role can claim the activity, and send messages to receive or pick activities.
Activity owner	Work on and complete an activity. Members of this role can transfer owned work items to an administrator or a potential owner.
Activity administrator	Repair activities that are stopped due to unexpected errors, and force terminate long-running activities.
Process starter	View the properties of the associated process instance, and its input and output messages.
Process reader	View the properties of the associated process instance, its input and output messages, and everything that the activity reader supports for all of the contained activities but not those of the subprocesses.
Process administrator	Members of this role can administer process instances and intervene in a process that has started; create, delete, and transfer work items. Members of this role also have activity administrator authorization.

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

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

Required roles for actions on process instances:

Access to the LocalBusinessFlowManager or the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on a process; the caller must also be authorized to perform the action. The following 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
getActiveHandlers	x	x	x
getAllActivities	x		x
getAllWorkItems	x		x
getClientUISettings	x		x
getCustomProperties	x	x	x
getCustomProperty	x	x	x
getCustomPropertyNames	x	x	x
getFaultMessage	x	x	x
getInputClientUISettings	x		x
getInputMessage	x	x	x
getOutputClientUISettings	x		x
getOutputMessage	x	x	x
getProcessInstance	x	x	x
getVariable	x	x	x
getWaitingActivities	x	x	x
getWorkItems	x		x
resume			x
restart			x
setCustomProperty		x	x
setVariable			x
suspend			x
transferWorkItem			x

Required roles for actions on business-process activities:

Access to the LocalBusinessFlowManager or the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on an activity; the caller must also be authorized to perform the action. The following 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
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
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

Starting business processes

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

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

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.

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),
     null);
   ```

   The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started by the call method.
2. Start the process with an input message of the appropriate type.

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

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

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

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

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

A microflow can be started by a receive activity or a pick activity. The starting service is not unique if the microflow starts with a pick activity that has multiple onMessage definitions.

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.

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),
     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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Determine the starting service to be called.

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

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

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

   ```
   ActivityServiceTemplateData activity = startActivities[0];
   //create a message for the service to be called
   ClientObjectWrapper input = process.createMessage
                              (activity.getServiceTemplateID(),
                               activity.getActivityTemplateID(),
                               activity.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   //start the process
   PIID piid = process.initiate(activity.getServiceTemplateID(),
                               activity.getActivityTemplateID(),
                               null,
                               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 staff, receive, or pick activities, work items are created for the potential owners.

Processing staff activities

Staff 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. One of these owners claims the activity. This person is responsible for providing the relevant information and completing the activity.

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",
                      null, null, 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 aiid = (AIID) result.getOID(1);
   ```

```
 ClientObjectWrapper input = process.claim(aiid);
 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 staff 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, the activity is put into the stopped state. In this state the activity can be repaired using force terminate or force retry.

Sending a message to a waiting activity

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

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

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

2. Send a message.

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

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

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

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

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

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

Handling events

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

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.getInputMessageTypeName());

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

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

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

Analyzing the results of a process

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

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

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

Managing the life cycle of a business process

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

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

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

Suspending and resuming a business process:

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

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

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

1. Get the 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.
3. Resume the process instance.
   ```
   process.resume( piid );
   ```
   This action puts the process instance and its subprocesses into the states they had before they were suspended.

Restarting a business process:

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

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

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

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.

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. For example, when an application is invoked and fails, and it does not return to a dormant state.

Because a process instance terminates immediately, without waiting for any outstanding subprocesses or activities, you should terminate a process instance only in exceptional situations.

1. Retrieve the process instance that is to be terminated.

   ```
   ProcessInstanceData processInstance =
          process.getProcessInstance("CustomerOrder");
   ```

2. Terminate the process instance.

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

   To terminate the process instance with compensation:

   ```
   PIID piid = processInstance.getID();
   process.forceTerminate(piid, CompensationBehaviour.INVOKE_COMPENSATION);
   ```

   To terminate the process instance without compensation:

   ```
   PIID piid = processInstance.getID();
   process.forceTerminate(piid);
   ```

   If you terminate the process instance with compensation, the compensation handler defined for the process template is called. If the process template does not have a compensation handler defined, the default compensation handler is called. If you terminated the process instance without compensation, the process instance is terminated immediately without waiting for activities to end normally.

   Applications that are started by the process are not affected 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, or if you want to defer the deletion of processes to off-peak times. However, process instance data that is no longer needed can impact disk space and performance. Therefore, you should regularly delete process instance data. 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.

1. List the process instances that are finished.
   ```
   QueryResultSet result =
        process.query("DISTINCT PROCESS_INSTANCE.PIID",
                      "PROCESS_INSTANCE.STATE =
                              PROCESS_INSTANCE.STATE.STATE_FINISHED",
                       null, null, null);
   ```
   This action returns a query result set that lists process instances that are finished.
2. Delete the process instances that are finished.
   ```
   while (result.next() )
   {
      PIID piid = (PIID) result.getOID(1);
      process.delete(piid);
   }
   ```
   This action deletes the selected process instance from the database.

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, you can repair the activities so that the navigation of the process can continue.

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:

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.

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

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

Invoke activities
: You can also force the completion of invoke activities that call an asynchronous service that is not a subprocess if these activities are in the

running state. You might want to do this, for example, if the asynchronous service is called and it does not respond.

1. List the stopped activities in the stopped state.

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

   This action returns the stopped activities for the CustomerOrder process instance.
2. Complete the activity, for example, a stopped staff activity.

   In this example, an output message is passed.

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

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

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

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

Retrying the execution of a stopped activity:

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

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

1. List the stopped activities.

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

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

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

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

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

   This action retries the activity. If an error occurs, the **continueOnError** parameter determines 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 it eventually reaches 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, initiate(). The execution of the process instance is driven automatically by the server.

Table 1. API methods for process templates

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

Process instances

The following API methods start process instances.

Table 2. API methods for starting process instances

Method	Description
call	Creates and runs a microflow.
callWithReplyContext	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.
callWithUISettings	Creates and runs a microflow and returns the output message and the client user interface (UI) settings.
initiate	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.
sendMessage	Sends the specified message to the specified activity service and process instance. The process instance can be either a microflow or a long-running process. These processes can either unique or non-unique starting services.
getStartActivities	Returns information about the activities that can start a process instance from the specified process template.
getActivityServiceTemplate	Retrieves the specified activity service template.

Table 3. 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 continue-on-error flag is set to false and an unhandled error occurs, the activity is put into the stopped state. A process administrator can then repair the activity. The continue-on-error flag and the associated repair functions can, for example, be used in a long-running process where an invoke activity fails occasionally, but the effort required to model compensation and fault handling is too high.

The following methods are available for working with and repairing activities.

Table 4. API methods for controlling the life cycle of activity instances

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

Variables and custom properties

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

Table 5. API methods for variables and custom properties

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

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.

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

Authorization roles for human tasks

Actions that you can take on human tasks depend on your authorization role. This role can be a J2EE role or an instance-based role.

A role is a group of employees who share the same level of authority. Java 2 Platform, Enterprise Edition (J2EE) roles are set up when the human task container is configured. Instance-based roles are assigned to human tasks and escalations when the task is modeled. Role-based authorization requires that global security is enabled in WebSphere Application Server.

J2EE roles

The following J2EE roles are supported:

- J2EE TaskSystemAdministrator. Users assigned to this role have all privileges.
- J2EE TaskSystemMonitor. Users assigned to this role can view the properties of all of the task objects.

You can use the administrative console to change the assignment of users and groups to these roles.

Setting up Roles using RACF security: These RACF permissions apply when the following security fields are specified:

- **com.ibm.security.SAF.authorization= true**
  ```
  RDEFINE EJBROLE TaskSystemAdministrator  UACC(NONE)
  PERMIT TaskSystemAdministrator CLASS(EJBROLE)  ID(userid) ACCESS(READ)
  ```
  ```
  RDEFINE EJBROLE TaskSystemMonitor  UACC(NONE)
  PERMIT TaskSystemMonitor  CLASS(EJBROLE)  ID(userid) ACCESS(READ)
  ```
- **com.ibm.security.SAF.delegation= true**
  ```
  RDEFINE EJBROLE JMSAPIUser  UACC(NONE) APPLDATA(' userid')
  ```

You can use Security Authorization Facility (SAF)-based authorization (for example, using the RACF EJBROLE profile) to control access by a client to Java 2 Platform, Enterprise Edition (J2EE) roles in EJB and Web applications, including the WebSphere Application Server administrative console application. For more information, see System Authorization Facility for role-based authorization in the WebSphere Application Server for z/OS information center.

Instance-based roles

A task instance or an escalation instance is not assigned directly to a staff member in the task model, instead it is assigned to one of the available roles. Any staff member that is assigned to an instance-based role can perform the actions for that role. The association of users to instance-based roles is determined at runtime using staff resolution.

The following instance-based roles are supported:

- For tasks: potential instance creator, originator, potential starter, starter, potential owner, owner, reader, editor, administrator
- For escalations: escalation receiver

These roles are authorized to perform the following actions:

Role	Authorized actions
Potential instance creator	Members of this role can create an instance of the task. If no potential instance creator is defined for the task template or the application components, then all users are considered to be a member of this role.
Originator	Members of this role have administrative rights until the task starts. When the task starts, the originator has the authority of a reader and can perform some administrative actions, such as suspending and resuming tasks, and transferring work items.
Potential starter	Members of this role can start an existing task instance. If a potential starter is not specified, the originator becomes the potential starter. For inline tasks without a potential starter, the default is everybody.
Starter	Members of this role have the authority of a reader and can perform some administrative actions, such as transferring work items.
Potential owner	Members of this role can claim a task. If no potential owner is defined for the task template or the application components, then all users are considered to be a member of this role.
Owner	Work on and complete a task.
Reader	View the properties of all of the task objects, but cannot work on them.
Editor	Members of this role can work with the content of a task, but cannot claim or complete it
Administrator	Members of this role can administer tasks, task templates, and escalations.
Escalation receiver	Members of this role have the authority of a reader.

Required roles for actions on tasks:

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

Action	Caller's principal role								
	Owner	Pot owner	Starter	Pot starter	Origin	Admin	Editor	Reader	Esc receiver
callTask				X[1]	X[1]	X[1]			
cancelClaim	X					X			
claim		X				X			
complete	X					X			
createFaultMessage	X	X	X	X	X[1]	X	X	X	X
createInputMessage	X	X	X	X	X[1]	X	X	X	X
createOutputMessage	X	X	X	X	X[1]	X	X	X	X
createWorkItem					X[1, 2]	X			
delete					X[3]	X			
deleteWorkItem					X[1, 2]	X			

Action	Caller's principal role								
	Owner	Pot owner	Starter	Pot starter	Origin	Admin	Editor	Reader	Esc receiver
getCustomProperty	X	X	X	X	X[1]	X	X	X	X
getDocumentation	X	X	X	X	X[1]	X	X	X	X
getFaultMessage	X	X	X	X	X[1]	X	X	X	X
getFaultNames	X	X	X	X	X[1]	X	X	X	X
getInputMessage	X	X	X	X	X[1]	X	X	X	X
getOutputMessage	X	X	X	X	X[1]	X	X	X	X
getRoleInfo	X	X	X	X	X[1]	X	X	X	X
getTask	X	X	X	X	X[1]	X	X	X	X
getUISettings	X	X	X	X	X[1]	X	X	X	X
resume	X				X[1]	X			
setCustomProperty	X		X			X	X		
setFaultMessage	X					X	X		
setOutputMessage	X					X	X		
startTask				X	X[1]	X			
suspend	X				X[1]	X			
terminate	X		X[1]			X			
transferWorkItem	X		X		X[1]	X			
update	X		X			X	X		

Notes:

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

Abbreviations:

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

Starting an originating task that invokes a synchronous interface

Originating tasks that invoke a synchronous interface include inline originating tasks in a microflow, stand-alone originating tasks in a microflow, and originating tasks that start, for example, 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.

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

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

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

   The results are sorted by name. The query returns an array containing the first 50 sorted originating templates.
2. Create an input message of the appropriate type.

   ```
   TaskTemplate template = taskTemplates[0];

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

Originating tasks that invoke a synchronous interface include inline originating tasks in a microflow, stand-alone originating tasks in a microflow, and originating tasks that start, for example, a simple Java class.

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

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

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

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

   The results are sorted by name. The query returns an array containing the first 50 sorted originating templates.
2. Create an input message of the appropriate type.

```
TaskTemplate template = taskTemplates[0];

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

3. Create the task and run it asynchronously.

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

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

Creating and starting a task instance

This scenario shows how to create an instance of a task template that defines a human task and start the task instance.

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

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

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

   The results are sorted by name. The query returns an array containing the first 50 sorted human 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 human 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
                                          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 participating or purely human tasks

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

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

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

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

2. Claim the task to be worked on.

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

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

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

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

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

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

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

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

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

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

//create a message of the appropriate type
ClientObjectWrapper myFault =
      task.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 human task instances or participating task instances and resume them again to complete them.

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.

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.

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",
                                   null, null, 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 participating or purely human task 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.

The results of the task are stored in the database only if the task template from which the task instance was derived does not specify automatic deletion of the derived task instances.

Analyze the results of the task.

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",
                                    null, null, null);
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper output = task.getOutputMessage(tkiid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject)
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

Terminating a task instance

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

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

1. Retrieve the task instance to be terminated.
   ```
   Task taskInstance = task.getTask(tkiid);
   ```
2. Terminate the task instance.
   ```
   TKIID tkiid = taskInstance.getID();
   task.terminate(tkiid);
   ```
   The task instance is terminated immediately without waiting for any outstanding tasks.

Deleting task instances

Task instances are only automatically deleted when they complete if this is specified in the associated task template from which the instances are derived. The following example shows how to delete all of the task instances that are finished and are not automatically deleted.

1. List the task instances that are finished.
   ```
   QueryResultSet result =
        task.query("DISTINCT TASK.TKIID",
                   "TASK.STATE = TASK.STATE.STATE_FINISHED",
                    null, null, null);
   ```

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

2. Delete the task instances that are finished.

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

Releasing a claimed task

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

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

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

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

2. Release the claimed task.

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

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

Managing work items

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

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

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'",
                                     null, null, null);
  if ( result.size() > 0 )
  {
  ```

```
  result.first();
  // create the work item
  task.createWorkItem((TKIID)(result.getOID(1)),
                        WorkItem.REASON_ADMINISTRATOR,"Smith");
}
```

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

- Delete a work item.

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

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

- Transfer a work item.

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

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

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. 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.

1. **Optional:** If your interfaces contain types that are not simple Java types, create or identify an application that contains the data types that are used by the runtime task or template.

   The runtime task or task template runs in the context of the application and gets access to the data types. Ensure that your application also contains a task or process definition so that the application is loaded by Business Process Choreographer. These tasks or processes can be dummy tasks or processes.
2. Create a task model.

   The model refers to the data types in the application identified in step 1.

3. Validate the task model.
4. Create the task template or the task instance.

   Use the HumanTaskManagerService interface to complete this action. If your interfaces contain types other than simple Java types, specify the name of the application that contains the data type definitions when you create your task instance or template.

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.

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

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" ),
             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:

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

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

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.

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

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, "InputBO.xsd" );
   factory.loadXSDSchema( resourceSet, "OutputBO.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" ),
             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" );
     ```

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.

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

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", null, 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, "BOapplication", "HTM" );
     ```
   - The following snippet creates a task template:

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

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.

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

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", null, 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" );
```

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 6. API methods for task templates

Method	Description
getTaskTemplate	Retrieves the specified task template.
createAndCallTask	Creates and runs a task instance from the specified task template and waits 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 7. API methods for task instances

Method	Description
getTask	Retrieves a task instance; the task instance can be in any state.
callTask	Starts an originating task synchronously.
startTask	Starts a task that has already been created.
suspend	Suspends the human or participating task.
resume	Resumes the human or participating task.

Table 7. API methods for task instances (continued)

Method	Description
terminate	Terminates the specified task instance. If an originating task is terminated, this action has no impact on the invoked service.
delete	Deletes the specified task instance.
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 8. API methods for working with escalations

Method	Description
getEscalation	Retrieves the specified escalation instance.

Variables and custom properties

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

Table 9. API methods for variables and custom properties

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

Allowed actions for tasks:

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

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

Action	Kind of task			
	Participating task	Human task	Originating task	Administrative task
callTask			X[1]	
cancelClaim	X	X[1]		
claim	X	X[1]		
complete	X	X[1]		X
createFaultMessage	X	X	X	X
createInputMessage	X	X	X	X
createOutputMessage	X	X	X	X
createWorkItem	X	X[1]	X	X
delete	X[1]	X[1]	X	X[1]
deleteWorkItem	X	X[1]	X	X
getCustomProperty	X	X[1]	X	X
getDocumentation	X	X[1]	X	X
getFaultMessage	X	X[1]	X	
getInputMessage	X	X[1]	X	
getOutputMessage	X	X[1]	X	
getRoleInfo	X	X[1]	X	X
getTask	X	X[1]	X	X
getUISettings	X	X[1]	X	X
resume	X	X[1]		
setCustomProperty	X	X[1]	X	X
setFaultMessage	X	X[1]		
setOutputMessage	X	X[1]		
startTask	X[1]	X[1]	X	X
suspend	X	X[1]		
terminate	X[1]	X[1]	X[1]	
transferWorkItem	X	X[1]	X	X
updateInactiveTask	X[2]	X[3]	X[2]	X[2]
updateTask	X	X[1]	X	X

Notes:

1. For stand-alone and runtime tasks and task templates only
2. For stand-alone tasks, inline tasks in business processes, and runtime tasks only
3. For stand-alone tasks and runtime tasks only

Querying business-process and task-related objects

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

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

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

Queries on business-process and task-related objects

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

Predefined database views are provided for you to query the object properties. For process templates, 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 is made up of:
- Select clause
- Where clause
- Order-by clause
- Skip-tuples parameter
- Threshold parameter
- Time-zone parameter

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

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

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

The query interface also contains a queryAll method. You can use this method to retrieve all of the relevant data about an object, for example, for monitoring purposes. The caller of the queryAll method must have 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.

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

Select clause:

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

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

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

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

Examples of select clauses

- ″WORK_ITEM.OBJECT_TYPE, WORK_ITEM.REASON″

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

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

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

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

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

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

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

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

Where clause:

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

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

The where-clause syntax supports:
- Keywords: AND, OR, NOT
- Comparison operators: =, <=, <, <>, >,>=, LIKE
- Set operation: IN

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

The following rules also apply:
- Specify object ID constants as `ID('string-rep-of-oid')`.
- Specify binary constants as `BIN('UTF-8 string')`.
- Use symbolic constants instead of integer enumerations. For example, instead of specifying an activity state expression `ACTIVITY.STATE=2`, specify `ACTIVITY.STATE=ACTIVITY.STATE.STATE_READY`.
- 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 name-value pairs, such as custom properties, by adding a one-digit suffix to the view name. For example: `"TASK_CPROP1.NAME='prop1' AND "TASK_CPROP2.NAME='prop2'"`
- Specify time-stamp constants as `TS('yyyy-mm-ddThh:mm:ss')`. To refer to the current date, specify `CURRENT_DATE` as the timestamp.

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

Examples of where clauses
- Comparing an object ID with an existing ID

  ```
  "WORK_ITEM.WIID = ID('_WI:800c00ed.df8d7e7c.feffff80.38')"
  ```

  This type of where clause is usually created dynamically with an existing object ID from a previous call. If this object ID is stored in a *wiid1* variable, the clause can be constructed as:

```
"WORK_ITEM.WIID = ID('" + wiid1.toString() + "')"
```

- Using time stamps

  ```
  "ACTIVITY.STARTED >= TS('2002-06-1T16.00.00')"
  ```
- Using symbolic constants

  ```
  "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER"
  ```
- Using Boolean values true and false

  ```
  "ACTIVITY.BUSINESS_RELEVANCE = TRUE"
  ```
- Using custom properties

  ```
  "TASK_CPROP1.NAME = 'prop1' AND " TASK_CPROP1.STRING_VALUE = 'v1' OR
   TASK_CPROP2.NAME = 'prop2' AND " TASK_CPROP2.STRING_VALUE = 'v2'"
  ```

Order-by clause:

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

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

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

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

Examples of order-by clauses

- ″PROCESS_TEMPLATE.NAME″

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

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

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

Skip-tuples parameter:

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

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

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

Example of a skip-tuples parameter

- new Integer(5)

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

Threshold parameter:

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

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

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

Example of a threshold parameter

- new Integer(50)

  Specifies that 50 qualifying tuples are to be returned.

Timezone parameter:

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

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

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

Examples of time-zone parameters

-
  ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 null,
                 null,
                 java.util.TimeZone.getDefault() );
  ```

  Returns object IDs for activities that started later than 17:40 local time on 1 January 2005.

-
  ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 null, null, null);
  ```

  Return object IDs for activities that started later than 17:40 UTC on 1 January 2005. This specification is, for example, 6 hours earlier in Eastern Standard Time.

Query results:

A query result set contains the results of a query.

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

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

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

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

Example:

The following query is run:

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

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

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

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

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

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

Managing stored queries

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

1. Create a stored query.

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

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

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

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

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

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

   ```
   String[] storedQuery = process.getStoredQueryNames();
   ```
4. **Optional:** Check the query defined by a specific stored query.

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

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

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

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

Predefined database views are provided for business-process and human-task objects. Use these views when you query reference data for these objects. When you use these views, you do not need to explicitly add join predicates for view columns, these constructs are added automatically for you. You can use the generic query function of the service API (BusinessFlowManagerService or

HumanTaskManagerService) to query this data. You can also use the corresponding method of the HumanTaskManagerDelegate API or your predefined queries provided by your implementations of the ExecutableQuery interface.

ACTIVITY view:

Use this predefined database view for queries on activities.

Table 10. Columns in the ACTIVITY view

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

Table 10. Columns in the ACTIVITY view (continued)

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

ACTIVITY_ATTRIBUTE view:

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

Table 11. 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 12. Columns in the ACTIVITY_SERVICE view

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

APPLICATION_COMP view:

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

Table 13. Columns in the APPLICATION_COMP view

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

ESCALATION view:

Use this predefined database view to query data for escalations.

Table 14. Columns in the ESCALATION view

Column name	Type	Comments
ESIID	String	The ID of the escalation instance.
ACTION	Integer	The action triggered by the escalation. Possible values are: **ACTION_CREATE_WORK_ITEM** Creates a work item for each escalation receiver. **ACTION_SEND_EMAIL** Sends an e-mail to each escalation receiver. **ACTION_CREATE_EVENT** Creates and publishes an event.
ACTIVATION_STATE	Integer	An escalation instance is created if the corresponding task reaches one of the following states: **ACTIVATION_STATE_READY** Specifies that the human or participating task is ready to be claimed. **ACTIVATION_STATE_RUNNING** Specifies that the originating task is started and running. **ACTIVATION_STATE_CLAIMED** Specifies that the task is claimed.
ACTIVATION_TIME	Timestamp	The time when the escalation is activated.
AT_LEAST_ EXP_STATE	Integer	The state of the task that is expected by the escalation. If a timeout occurs, the task state is compared with the value of this attribute. Possible values are: **AT_LEAST_EXPECTED_STATE_CLAIMED** Specifies that the task is claimed. **AT_LEAST_EXPECTED_STATE_ENDED** Specifies that the task is in a final state (FINISHED, FAILED, TERMINATED or EXPIRED).
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** The task priority is not increased. **INCREASE_PRIORITY_ONCE** The task priority is increased once by one. **INCREASE_PRIORITY_REPEATED** The task priority is increased by one each time the escalation repeats.
NAME	String	The name of the escalation.

Table 14. Columns in the ESCALATION view (continued)

Column name	Type	Comments
STATE	Integer	The state of the escalation. Possible values are: STATE_INACTIVE STATE_WAITING STATE_ESCALATED STATE_SUPERFLUOUS
TKIID	String	The task instance ID to which the escalation belongs.

ESCALATION_CPROP view:

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

Table 15. Columns in the ESCALATION_CPROP view

Column name	Type	Comments
ESIID	String	The escalation ID.
NAME	String	The name of the property.
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 16. Columns in the ESCALATION_DESC view

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

PROCESS_ATTRIBUTE view:

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

Table 17. 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 18. Columns in the PROCESS_INSTANCE view

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

PROCESS_TEMPLATE view:

Use this predefined database view for queries on process templates.

Table 19. Columns in the PROCESS_TEMPLATE view

Column name	Type	Comments
PTID	ID	The process template ID.
NAME	String	The name of the process template.

Table 19. Columns in the PROCESS_TEMPLATE view (continued)

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

TASK view:

Use this predefined database view for queries on task objects.

Table 20. 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.

Table 20. Columns in the TASK view (continued)

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

Table 20. Columns in the TASK view (continued)

Column name	Type	Comments
KIND	Integer	The kind of task. Possible values are: **KIND_HUMAN** States that the task is created and processed by a human. **KIND_WPC_STAFF_ACTIVITY** States that the task is a human task that is a staff activity of a WebSphere Business Integration Server Foundation, version 5 business process. **KIND_ORIGINATING** States that the task supports person-to-computer interactions, which enables people to create, initiate, and start services. **KIND_PARTICIPATING** States that the task supports computer-to-person interactions, which enable a person to implement a service. **KIND_ADMINISTRATIVE** States that the task is an administrative task.
LAST_MODIFIED	Timestamp	The time when the task was last modified.
LAST_STATE_ CHANGE	Timestamp	The time when the state of the task was last modified.
NAME	String	The name of the task.
NAME_SPACE	String	The namespace that is used to categorize the task.
ORIGINATOR	String	The principal ID of the task originator.
OWNER	String	The principal ID of the task owner.
PARENT_ CONTEXT_ID	String	The parent context for this task. This attribute provides a key to the corresponding context in the calling application component. The parent context is set by the application component that creates the task.
PRIORITY	Integer	The priority of the task.
STARTED	Timestamp	The time when the task was started (STATE_RUNNING, STATE_CLAIMED).
STARTER	String	The principal ID of the task starter.

Table 20. Columns in the TASK view (continued)

Column name	Type	Comments
STATE	Integer	The state of the task. Possible values are: **STATE_READY** States that the task is ready to be claimed. **STATE_RUNNING** States that the task is started and running. **STATE_FINISHED** States that the task finished successfully. **STATE_FAILED** States that the task did not finish successfully. **STATE_TERMINATED** States that the task has been terminated because of an external or internal request. **STATE_CLAIMED** States that the task is claimed. **STATE_EXPIRED** States that the task ended because it exceeded its specified duration.
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.
SUSPENDED	Boolean	Indicates whether the task is suspended.
TKTID	String	The task template ID.
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 21. Columns in the TASK_CPROP view

Column name	Type	Comments
TKIID	String	The task instance ID.
NAME	String	The name of the property.
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 22. 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 23. Columns in the TASK_TEMPL view

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

Table 23. Columns in the TASK_TEMPL view (continued)

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

Column name	Type	Comments
TKTID	String	The task template ID.
NAME	String	The name of the property.
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 25. 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 26. 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	The type of the associated object. Possible values are: **OBJECT_TYPE_ACTIVITY** Specifies that the work item was created for an activity. **OBJECT_TYPE_PROCESS_INSTANCE** Specifies that the work item was created for a process instance. **OBJECT_TYPE_TASK_INSTANCE** Specifies that the work item was created for a task. **OBJECT_TYPE_TASK_TEMPLATE** Specifies that the work item was created for a task template. **OBJECT_TYPE_ESCALATION_ INSTANCE** Specifies that the work item was created for an escalation instance. **OBJECT_TYPE_APPLICATION_ COMPONENT** Specifies that the work item was created for an application component.
OBJECT_ID	ID	The ID of the associated object, for example, the associated process or task.

Table 26. Columns in the WORK_ITEM view (continued)

Column name	Type	Comments
ASSOC_OBJECT_TYPE	Integer	The type of the object referenced by the ASSOC_OID attribute, for example, task, process, or external objects. Use the values for the OBJECT_TYPE attribute.
ASSOC_OID	ID	The ID of the object associated object with the work item. For example, the process instance ID (PIID) of the process instance containing the activity instance for which this work item was created.
REASON	Integer	The reason for the assignment of the work item. Possible values are: REASON_POTENTIAL_STARTER REASON_POTENTIAL_INSTANCE_ CREATOR REASON_POTENTIAL_OWNER REASON_EDITOR REASON_READER REASON_ORIGINATOR REASON_OWNER REASON_STARTER REASON_ESCALATION_RECEIVER REASON_ADMINISTRATOR
CREATION_TIME	Timestamp	The date and time when the work item was created.

Handling exceptions and faults

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

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

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

Handling API exceptions

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

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

Checking which fault is set for an activity

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

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

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

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

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

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

Checking which fault occurred for a stopped invoke activity

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

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

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

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

   This is the local part of the fault queue name.

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

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

Developing Web applications for business processes and human tasks, using JSF components

Business Process Choreographer Explorer provides several JavaServer Faces (JSF) components. You can extend and integrate these components to add business-process and human-task functionality to Web applications.

You can use WebSphere Integration Developer to build your Web application.

1. Create a dynamic project and change the Web Project Features properties of the Web project to include the Faces 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

   These files are in the *install_root*/ProcessChoreographer/client directory.
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 libraries 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.

e. Implement the custom code that you need to support the JSF components.

5. Deploy the application.

   Map the EJB references to the Java Naming and Directory Interface (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 27. 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

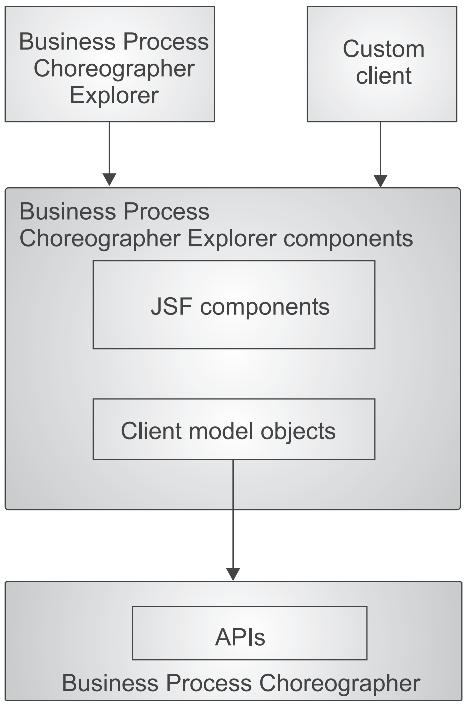
Your deployed Web application contains the functionality provided by the Business Process Choreographer Explorer components.

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 imbed these elements in Web applications. These 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.


Business Process Choreographer Explorer
Custom client
Business Process Choreographer Explorer components
JSF components
Client model objects
APIs
Business Process Choreographer

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 process 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 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.

List handling in the List component:

Every instance of the List component is associated with an instance of the com.ibm.bpe.jsf.handler.BPCListHandler class.

This list handler tracks which items in the associated list are selected and it provides a notification mechanism. 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. Business Process Choreographer Explorer uses this notification mechanism to associate the list entries with the details pages for the different kinds of items. The trigger for a notification event is typically one of the properties of the items that is displayed in the current list.

To exploit the notification mechanism, set the value of the action attribute of the `bpe:column` tag for the property to the JSF navigation target your application is to continue with when the notification event is triggered. The List component renders the entry in the column as a JSF command link. If the link is triggered, the object that represents the entry in the list is determined, and it is passed to all of the registered item listeners. You can register implementations of this interface in the configuration file of your JavaServer Faces (JSF) application.

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()`.

Error handling

You can take advantage of the error handling functions provided by the BPCListHandler class in the following error situations.

- 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. Exploiters 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. When the list is rendered again, the list handler displays the error messages for the qualifying list entries in a separate column. 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.

CommandBar component:

Use the CommandBar component to integrate action buttons in 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.

How commands are processed

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.bpe.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.

Error handling

A command is triggered only if one of the following conditions are true:
- An exception is not thrown
- If an exception is thrown, it is an ErrorsInCommandException exception

There are several ways in which you can implement error handling in the CommandBar component:
- You can decide not to use any of the features of the CommandBar component. If, for example, you want to display the errors on a page that is specific to the selected command, the implementation of the command can catch the exceptions that occur and propagate them to a page bean that is used for the error page. You can make the page bean available to the command implementation using the context attribute of the `bpe:commandbar` tag. After the exception is set on the page bean, the command returns the string of the JSF navigation rule that is defined for the error page.
- If you want to display an error message below the command-bar section in the user interface, create an exception class that implements the com.ibm.bpc.clientcore.exception.CommandBarMessage marker interface. This interface provides a message catalog of error messages.
- If the command operates on a list of items, you might want to track the success of the command for each of the items in the list. To track the errors, map each exception to the item for which the operation failed. The CommandBar component can pass a map, which contains the identifiers as keys and the exceptions as values, to the model object that is defined for the CommandBar component.

  For this mechanism to work, the model object must implement the com.ibm.bpe.jsf.handler.ErrorHandler interface and the command must throw a com.ibm.bpc.clientcore.exception.ErrorsInCommandException exception. The CommandBar component then passes the map contained in the exception to the error handler. The action method is triggered although an error occurred, and the current view is refreshed. The Business Process Choreographer Explorer application makes use of this method to display exceptions in lists.
- If you throw a ClientException exception that does not implement the CommandBarMessage interface and the exception is not an ErrorsInCommandException, the CommandBar component propagates the exception to the BPCError error bean that is defined in the configuration file of your application. The error processing continues with the error navigation target.

Utilities provided by the Business Process Choreographer Explorer JSF components:

The JavaServer Faces (JSF) components provide utilities for user-specific time zone information and error handling.

User-specific time zone information

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();
}
```

ErrorBean interface for error handling

Sometimes, the JSF components exploit a predefined managed bean, BPCError, for error handling. This bean implements the com.ibm.bpc.clientcore.util.ErrorBean interface. In error situations that trigger the error page, the exception is set on the error bean. 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 thrown as a ClientException error by the execute method of a command
- If a ClientException error is thrown 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();
}
```

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.

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>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>
```

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.

To provide a connection to Human Task Manager, the TaskPool managed bean is implemented using the htmConnection managed bean.

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 MyTaskQuery 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)",
      null,
      null,
      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.

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.

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 provides the specific object for each column.

Table 28. bpe:list attributes

Attribute	Required	Description
model	yes	A value binding for a managed bean of the com.ibm.bpe.jsf.handler.BPCListHandler class.

Table 28. bpe:list attributes (continued)

Attribute	Required	Description
styleClass	no	The cascading style sheet (CSS) style for rendering the overall table containing titles, rows, and paging buttons.
headerStyleClass	no	The CSS style class for rendering the table header.
cellStyleClass	no	The CSS style class for rendering individual table cells.
buttonStyleClass	no	The CSS style class for rendering the buttons in the footer area.
rowClasses	no	The CSS style class for rendering the rows in the table.
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.
checkbox	no	Determines whether the check box for selecting multiple items is rendered. The attribute has the value `true` or `false`.

Table 29. bpe:column attributes

Attribute	Required	Description
name	yes	The name of the object property that is shown in this column. This name must correspond to a named property as defined in the corresponding client model class.
action	no	If this attribute is specified as an outcome string, it defines an outcome used by the JavaServer Faces (JSF) navigation handler to determine the next page. If this attribute is specified as a method binding (#{.....}), the method to be called has the signature `String method()` and its return value is used by the JSF navigation handler to determine the next page.
label	no	The label displayed in the header of the column or the cell of the table header row. If this attribute is not set, a default label is provided by the client model class.
converterID	no	The ID used to register the converter in the JSF configuration file. If a converter ID is not specified, the implementation of the objects displayed in the list can contain a definition of a converter for the current property. The list component uses this converter.

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.

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. If the Details component does not contain any properties, all of the properties of the object are displayed.

   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.

Your JSF application now contains a JavaServer page that displays the details of the specified object, for example, the details of a task instance.

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 30. bpe:details attributes

Attribute	Required	Description
model	yes	A value binding for a managed bean of the com.ibm.bpe.jsf.handler.BPCDetailsHandler class.
styleClass	no	The cascading style sheet style class for rendering the HTML element.
columnClasses	no	A list of CSS styles, separated by commas, for rendering columns.
rowClasses	no	A list of CSS styles, separated by commas, for rendering rows.

Table 31. bpe:property attributes

Attribute	Required	Description
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.
label	no	The label for the property. If this attribute is not set, a default label is provided by the client model class.
converterID	no	The ID used to register the converter in the JavaServer Faces (JSF) configuration file.

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.

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.

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 to display some of the properties of a task instance.

```
<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 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 properties of the Java objects and it must implement the ItemProvider interface. This interface is implemented by the BPCListHandler class and the BPCDetailsHandler class. It also includes a custom claim command.

2. **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 have changed the configuration of either of these handlers or you have used a different managed bean, add a managed bean that implements the ItemProvider interface to the JSF configuration file.

3. Add the code that implements the custom commands to the JSF application.

   The following code snippet shows how to write a command class that extends the command bar. This command class (MyClaimCommand) is referred to by the `bpe:command` tag in the JSP file.

   The command checks the preconditions and any other prerequisites, for example, the correct number of selected items. It then retrieves a reference to the human task API, HumanTaskManagerService. The command iterates over the selected objects and tries to process them. The task is claimed through the HumanTaskManagerService API by an ID. If an exception does not occur, the state is updated for the corresponding TaskInstanceBean object. This action avoids retrieving the value of the object from the server again.

```
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. The command must implement the public String execute(List selectedObjects)throws ClientException method. The command returns an outcome 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 throws an exception other than an ErrorsInCommandException exception.

Your JSF application now contains a JavaServer page that implements a customized command bar.

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.bpe.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 32. bpe:commandbar attributes

Attribute	Required	Description
model	yes	A value binding expression to a managed bean that implements the ItemProvider interface. This managed bean is usually the com.ibm.bpe.jsf.handler.BPCListHandler class or the com.ibm.bpe.jsf.handler.BPCDetailsHandler class that is used by the List component or Details component in the same JavaServer Pages (JSP) file as the CommandBar component.
styleClass	no	The cascading style sheet (CSS) style for rendering the bar.
buttonStyleClass	no	The CSS style for rendering the buttons in the command bar.

Table 33. bpe:command attributes

Attribute	Required	Description
commandID	yes	The ID of the command.
commandClass	yes	The command class that is triggered.

Table 33. bpe:command attributes (continued)

Attribute	Required	Description
action	no	A JavaServer Faces (JSF) action method that has the signature: `String method()`. The value that is returned by the action method, or that is directly specified as a literal overrides the target returned by the execute method of the command. The action attribute is not processed if the command throws an exception other than an ErrorsInCommandException exception. If this attribute is specified as an outcome string, it defines an outcome used by the JSF navigation handler to determine the navigation rule and the next page to display. If this attribute is specified as a method binding (#{.....}), the method to be called has the signature `String method()`. Its return value is used by the JSF navigation handler to determine the navigation rule and the next page to display.
label	yes	The label of the button that is rendered in the command bar.
styleClass	no	The CSS style for rendering the button. This style overrides the button style defined for the command bar.
context	no	A value binding expression, which refers to a managed bean. Use this attribute if the command needs to initialize the target page or bean.

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.

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.

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.bpe.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 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.

Your JSF application now contains a JavaServer page that can render data objects and primitive types.

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 34. bpe:form attributes

Attribute	**Required**	**Description**
model	yes	A value binding expression expression that refers to either a `commonj.sdo.DataObject` object or a `com.ibm.bpc.clientcore.MessageWrapper` object.
simplification	no	If this attribute is set to `true`, properties with a cardinality of zero or one are shown. By default, this attribute is set to `false`.

Table 34. bpe:form attributes (continued)

Attribute	Required	Description
readOnly	no	If this attribute is set to `true`, a read-only form is rendered. By default, this attribute is set to `false`.
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.
styleClass4validInput	no	The CSS class name for rendering input that is valid.
styleClass4invalidInput	no	The CSS 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.
buttonStyleClass	no	The CSS style for the buttons that work on arrays or lists.

Mapping of client model objects

The client model objects implement the corresponding interfaces of the Business Process Choreographer API.

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 35. How Business Process Choreographer interfaces are mapped to client model objects

Business Process Choreographer interface	Client model object class
com.ibm.bpe.api.ActivityInstanceData	com.ibm.bpe.clientmodel.bean.ActivityInstanceBean
com.ibm.bpe.api.ActivityServiceTemplateData	com.ibm.bpe.clientmodel.bean.ActivityServiceTemplateBean
com.ibm.bpe.api.ProcessInstanceData	com.ibm.bpe.clientmodel.bean.ProcessInstanceBean
com.ibm.bpe.api.ProcessTemplateData	com.ibm.bpe.clientmodel.bean.ProcessTemplateBean
com.ibm.task.api.Escalation	com.ibm.task.clientmodel.bean.EscalationBean
com.ibm.task.api.Task	com.ibm.task.clientmodel.bean.TaskInstanceBean
com.ibm.task.api.TaskTemplate	com.ibm.task.clientmodel.bean.TaskTemplateBean

Developing event handlers for human task events

You can create plug-ins for human task API events and escalation notification events.

To work with the events that occur when a task is processed, you must specify the event handler name in the task model.

You can create the following types of event handlers for human task events:

Notification event handler
To create an event handler for escalation notifications, you must implement the com.ibm.task.spi.NotificationEventHandlerPlugin interface.

API event handler
To create an event handler for human task events, you must implement the com.ibm.task.spi.APIEventHandlerPlugin interface.

1. Implement the event handler as a JAR file.

   The JAR file requires the following:

   - A class that implements the event handler interface. For example, com.ibm.task.spi.NotificationEventHandlerPlugin for notification events or com.ibm.task.spi.APIEventHandlerPlugin for API events. For example:

     ```
     package com.ibm.task.spi ;

     public interface NotificationEventHandlerPlugin
     {
        public void interFaceMethod(Param param) ;
        :
     }
     ```
   - In the JAR file, you must have a property file in the directory META-INF/services/ directory with the following file name: `com.ibm.task.spi.`*`%identifier%%type%`*`EventHandlerPlugin`, where *%identifier%* is the event handler name that is specified in your model, for example, `MyEventHandler`, and *%type%* is the event type (either `Notification` or `API`).

     The first line of this file that is neither a comment line nor a blank line must specify the name of the plug-in implementation. For example, the `META-INF/services/`
     `com.ibm.task.spi.`**`MyEventHandlerNotification`**`EventHandlerPlugin` file might contain the line

     ```
     myevents.EventHandlerImplementation
     ```
2. Make the JAR file available to applications.
   - If you want the event handler to be used by only one Java 2 Enterprise Edition (J2EE) application, you can include the JAR file in the application EAR file.
   - If you want several applications to use the event handler, consider putting the JAR file in a WebSphere Application Server shared library. You can then explicitly associate the library with the applications that need access to the event handler.

   The event handler handles human task events that occur within the application.

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 7 on page 102 shows how an application contains components and related libraries.

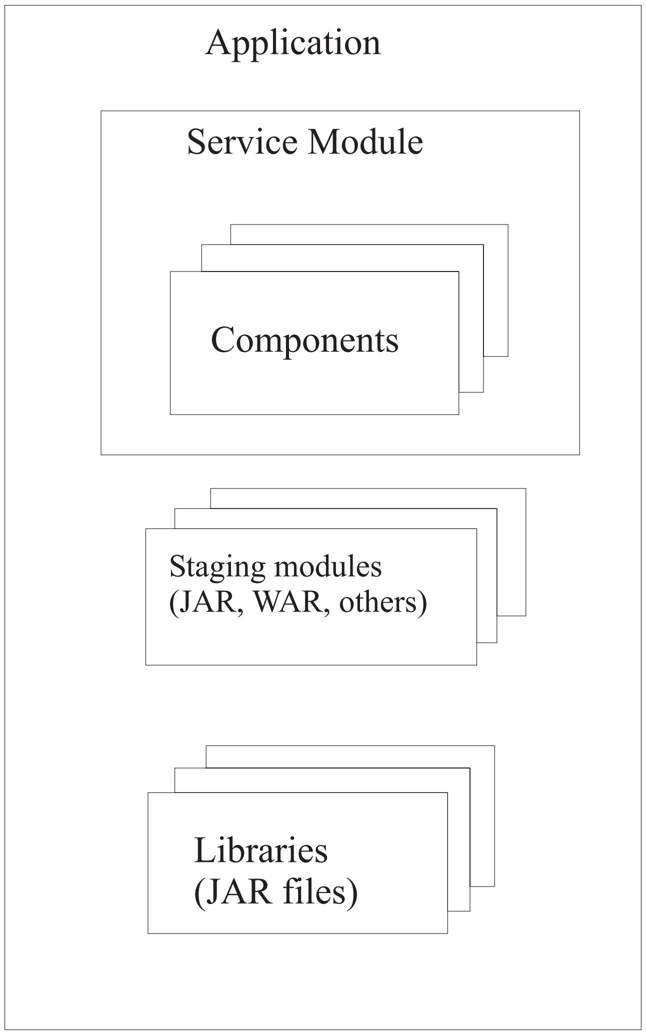


Figure 7. Relationship amongst module, component and library

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 2 Enterprise Edition (J2EE) specification. Details about JAR files can be found in the JAR specification.

In WebSphere Process Server, a JAR file also contains an application, which is the assembled version of the module with all the supporting references and interfaces to any other 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 (see Installing a module on a production server).

Naming conventions for staging modules

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

- *myService*App
- *myService*EJB
- *myService*EJBClient
- *myService*Web

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

Considerations when using libraries

Using libraries provides consistency of business objects and consistency of processing amongst modules because each calling module has its own copy of a specific component. To prevent inconsistencies and failures it is important to make sure that changes to components and business objects used by calling modules are coordinated with all of the calling modules. Update the calling modules by:

1. Copying the module and the latest copy of the libraries to the production server
2. Rebuilding the installable EAR file using the serviceDeploy command
3. Stopping the running application containing the calling module and reinstall it
4. Restarting the application containing the calling module

EAR file overview

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

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

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

Preparing to deploy to a server

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

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

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

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

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

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

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

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

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

   In this step you ensure that any references to artifacts required by a component are available. Names for components should not conflict with the names the serviceDeploy command uses for staging modules. See Naming conventions for staging modules.
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.

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.

Related tasks

Creating a clustered environment

Installing a module on a production server

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

Before deploying a service application to a production server, assemble and test the application on a test server. After testing, export the relevant files as described in Preparing to deploy to a server and bring the files to the production system to deploy. See the information centers for WebSphere Integration Developer and WebSphere Application Server Network Deployment, version 6 for more information.

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

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

   This step defines the module to the server in preparation for installing the application into production.
   a. Locate the JAR file that contains the module to deploy.
   b. Issue the 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.

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

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

Creating an installable EAR file using serviceDeploy

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

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

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

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

Deploying applications using ANT tasks

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

This task assumes the following:

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

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

The reference portion of the information center for WebSphere Application Server Network Deployment, version 6 contains a section on application programming interfaces. ANT tasks are described in the package com.ibm.websphere.ant.tasks. For the purpose of this topic, the tasks of interest are ServiceDeploy and InstallApplication.

If you need to install multiple applications concurrently, develop an ANT task before deployment. The ANT task can then deploy and install the applications on the servers without your involvement in the process.

1. Identify the applications to deploy.

2. Create a JAR file for each application.
3. Copy the JAR files to the target servers.
4. Create an ANT task to run the ServiceDeploy command to create the EAR file for each server.
5. Create an ANT task to run the InstallApplication command for each EAR file from step 4 on the applicable servers.
6. Run the ServiceDeploy ANT task to create the EAR file for the applications.
7. Run the InstallApplication ANT task to install the EAR files from step 6.

The applications are correctly deployed on the target servers.

Example of deploying an application unattended

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

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

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

This statement shows how to invoke the ANT task.

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

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

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

Installing business process and human task applications

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

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

Before you install a business process or human task application, make sure that the following conditions are true:

- The servers on which you want to install the application are running.
- In each cluster, at least one server on which you want to install Enterprise JavaBeans modules with processes or tasks is running.

You can install business process and task applications from the administrative console, from the command line, or by running an administrative script, for example. When you run an administrative script to install a business process application or a human task application, a server connection is required. Do not use the `-conntype NONE` option as an installation option.

1. If you are installing an application on a cluster, verify that the application uses the data source that is named after the cluster.

   For example, if the application was generated using the default data source `BPEDB`, change the data source for the application to BPEDB_*cluster_name*, where *cluster_name* is the name of the cluster on which you installed the application.
2. Install the application. For more information, see , which is in the WebSphere Application Server information center.

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

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

Deployment of models

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

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

New versions of a process template or task template have the same name, but a different valid-from attribute. You can deploy many different versions of a process template or task template, each in a different application. However, no two versions of the same process can have the same valid-from date. If you want to install different versions of the same process, specify a different valid-from date for each version. All the different process versions are stored in the database.

If you do not specify a valid-from date, the date is determined as follows:
- For a human task, the valid-from date is the date on which the application was installed.
- For a business process, the valid-from date is the date on which the process was modeled.

When you can install a process application on a cluster in which no servers are running

This topic explains the exceptional circumstances in which you might need to install an application on a cluster that has no running servers.

During the installation of a business process application on a server, the Java Naming and Directory Interface (JNDI) name of the data source of the corresponding business process container must be resolved. You cannot, therefore, install an application without a server connection. In a Network Deployment (ND) environment, this server is the deployment manager.

Restrictions lifted

If you want to install a business process application on a cluster in an ND environment, no server in the cluster need be running if the following conditions are true:

- The required data sources are defined at the cell level.
- The process application does not specify human tasks.

For process applications that have no human tasks, the data source lookup operation is accomplished within the namespace of the deployment manager, when a lookup operation in the namespace of the application server previously failed. If the application is successfully installed, ignore any error messages in the `SystemOut.log` file that indicate a failure of the data source lookup operation within the application server namespace.

When it will work

- The lookup operation within the deployment manager namespace is successful only if the data source JNDI name is defined at the cell level.
- If you use the wizard to configure a business process container or human task container on a stand-alone server, the data source is defined at the server level. The same is true if you use the configuration script `bpeconfig.jacl`, which is provided in the `ProcessChoreographer/sample` directory of your application server installation. In this case, you must define the data source manually at the cell level and use this data source when you install the business process container.
- If you configure a business process container with the wizard on a cluster member, the data source is automatically defined at the cell level. The JNDI name is scoped by the cluster name. The same is true if you use the configuration script `bpeconfig.jacl`, which is provided in the `ProcessChoreographer/sample` directory of your application server installation. In this case, you do not need to change anything manually.

When it will not work

Process applications that contain human tasks require an additional JNDI name lookup operation to locate the staff plug-in provider. Therefore, to help ensure successful installation of such applications, make sure that the cluster includes a running server.

Scoping side effects

A side effect of the name lookup is that if an application server is not running and a data source is defined on its server or node level with the same name as a data source at the cell level, the cell level data source takes precedence. This means that you are using a different data source during deployment and at run time.

Attention: Avoid name clashes. If you define data sources at the cell level manually, use JNDI names that are scoped by the cluster name or server name and node name, for example, `jdbc/BPEDB_`.

Uninstalling business process and human task applications, using the administrative console

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

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

   This action prevents the creation of process and task instances.

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

   b. Select the application that you want to stop.

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

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

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

   f. Click **Stop**.

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

2. Verify that 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 (ND) environment, the deployment manager, all ND-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 no process instances or task instances exist.

   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.

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.

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.

- In an ND environment, the server process is the deployment manager.
- In a stand-alone environment, the server process is the application server.

To ensure that the administrative client automatically connects to the server process, do not use the `-conntype NONE` option as a command option.

Ensure that you delete any process instances or task instances associated with the templates in the applications, for example, using Business Process Choreographer Explorer.

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.

1. Change to the Business Process Choreographer samples directory. Type the following:

   ```
   cd install_root/ProcessChoreographer/sample
   ```
2. Stop the templates and uninstall the corresponding application.

   ```
   install_root/bin/wsadmin -f bpcTemplates.jacl
                               [-user user_name]
                               [-password user password]
                               -uninstall application_name
   ```

   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.

The application is uninstalled.

Installing applications with embedded WebSphere Adapters

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

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

1. Assemble an application with resource adapter archive (RAR) modules in it. See Assembling applications.
2. Install the application following the steps in Installing a new application. In the Map modules to servers step, specify target servers or clusters for each RAR file. Be sure to map all other modules that use the resource adapters defined in the RAR modules to the same targets. Also, specify the Web servers as targets that serve as routers for requests to this application. The plug-in configuration file (plugin-cfg.xml) for each Web server is generated based on the applications that are routed through it.

   Note: When installing a RAR file onto a server, WebSphere Application Server looks for the manifest (MANIFEST.MF) for the connector module. It looks first in the connectorModule.jar file for the RAR file and loads the manifest from the _connectorModule.jar file. If the class path entry is in the manifest from the connectorModule.jar file, then the RAR uses that class path. To ensure that the installed connector module finds the classes and resources that it needs, check the Class path setting for the RAR using the console. For more information, see Resource Adapter settings and WebSphere relational resource adapter settings.

3. Save the changes. Click **Finish > Save**.
4. Create connection factories for the newly installed application
   a. Open the administrative console.
   b. Select the newly installed application Click **Applications > Enterprise Applications >** ***application name***.
   c. Click **Connector Modules** in the Related Items section of the page.
   d. Select the RAR file. Click on ***filename.rar***
   e. Click **Resource adapter** in the Additional Properties section of the page.
   f. Click **J2C Connection Factories** in the Additional Properties section of the page.
   g. Click on an **existing connection factory** to update it, or **New** to create a new one.

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

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

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

Note:

A given native library can only be loaded one time for each instance of the Java virtual machine (JVM). Because each application has its own classloader, separate applications with embedded RAR files cannot both use the same native library. The second application receives an exception when it tries to load the library.

If any application deployed on the application server uses an embedded RAR file that includes native path elements, then you must always ensure that you shut down the application server cleanly, with no outstanding transactions. If the application server does not shut down cleanly it performs recovery upon server restart and loads any required RAR files and native libraries. On completion of recovery, do not attempt any application-related work. Shut down the server and restart it. No further recovery is attempted by the application server on this restart, and normal application processing can proceed.

WebSphere Adapter

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

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

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

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

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

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

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

WebSphere Adapter deployment considerations

The deployment of WebSphere Adapters requires specific options regarding scope.

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

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

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

For embedded WebSphere Adapters:

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

You should not install standalone WebSphere Adapters.

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

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

Installing Standalone WebSphere Adapters

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

You should configure the database before installing the adapter.

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

1. Open the Install RAR file dialog window.

   On the administrative console:

   a. Expand **Resources**

   b. Click **Resource Adapters**

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

   d. Click **Install RAR**

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

   Note: When installing a RAR file using this dialog, the scope you define on the Resource Adapters page has no effect on where the RAR file is installed. You can install RAR files only at the node level. The node on which the file is installed is determined by the scope on the Install RAR page. (The scope you set on the Resource Adapters page determines the scope of the new resource adapters, which you can install at the server, node, or cell level.)

2. Install the RAR file

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

a. Browse to the location of the JCA connector. If the RAR file is on the local workstation select Local Path and browse to find the file. If the RAR file is on a network server, select **Server path** and specify the fully qualified path to the file.
b. Click **Next**
c. Enter the resource adapter name and any other properties needed under General Properties. If you install a J2C Resource Adapter that includes native path elements, consider the following: If you have more than one native path element, and one of the native libraries (native library A) is dependent on another library (native library B), then you must copy native library B to a system directory. Because of limitations on most UNIX systems, an attempt to load a native library does not look in the current directory.
d. Click **OK**.

WebSphere Adapter applications as members of clusters

WebSphere Adapter module applications can be cloned as members of a cluster under certain conditions.

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

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

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

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

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

WebSphere Business Integration Adapter applications as members of clusters

WebSphere Business Integration Adapter module applications can be cloned as members of a cluster under certain conditions.

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

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

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

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

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

Installing EIS applications

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

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

See the subsequent tasks for detailed information.

Deploying an EIS application module to the J2SE platform

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

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

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

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

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

Deploying an EIS application module to the J2EE platform

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

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

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

Table 36. Mapping from bindings to J2EE artifacts

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

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

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

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

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

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

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

Troubleshooting a failed deployment

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

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

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

1. Determine if the application installation failed.

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

   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.

   If the application is not running, the failure occurred when the server attempted to initiate the resources for the application.
   a. Examine the system.out file for messages that will direct you on how to proceed.
   b. Determine if the resources are started.

      Resources that are not started prevent an application from running to protect against lost information. The reasons for a resource not starting include:
      - Bindings are specified incorrectly
      - Resources are not configured correctly
      - Resources are not included in the resource archive (RAR) file

- Web resources not included in the Web services archive (WAR) file

c. Determine if any components are missing.

   The reason for missing a component is an incorrectly built enterprise archive (EAR) file. Make sure that the all of the components required by the module are in the correct folders on the test system on which you built the Java archive (JAR) file. Refer to **Developing and deploying modules > Overview of preparing and installing modules > Preparing to deploy to a server** for 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 118.

   a. Determine if the applications uses any services contained in another application. Make sure that the other application is installed and running.

   b. Determine if the import and export bindings for all services contained in other applications the failing application uses 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 applications specifications when installing an application that contains services. There are occasions when you must delete these specifications before reinstalling the application.

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.

Delete J2C activation specifications when you inadvertently saved a configuration after installing an application that contains services and do not require the specifications.

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 stand alone server or at the **server** scope in a Network 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.**

The system removes selected specifications from the display.

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.

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.

Delete SIBus destinations when you inadvertently saved a configuration after installing an application that contains services and 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.

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.**

The panel displays only the remaining destinations.

Delete the J2C activation specifications related to the module that created these destinations.

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.

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 Websphere Process Server for z/OS version 6.0.1

Printed in USA