BEA WebLogic® Integration

Using Integration Controls

Version 10.2
Document Revised: March 2008

Contents

File Control

Http Control

Message Broker Controls

MQSeries Control

Process Control

RosettaNet Control

Service Broker Control

ALSB Control

TIBCO Rendezvous Control

TPM Control

WLI JMS Control

Worklist Controls

WLI Timer Control

XML MetaData Cache Control

Using Control Factories

Using Message Attachments

CHAPTER 1

Using Controls in Business Processes

When you access a resource through a control, your interaction with the resource is greatly simplified; the underlying control implementation takes care of most of the details for you. You add an instance of a control to your business process project and then invoke its methods. All controls expose Java interfaces that can be invoked directly from your business process.

Designing the business process interactions with resources via controls includes:

- Adding Control Nodes to Your Business Process
- Designing the Communications for Control Nodes
- Using Integration Controls in Web Services or Page Flows

Adding Control Nodes to Your Business Process

You add **Control** nodes to your business process to represent points in the business process at which you design interactions with resources via controls:

- **Control Send** nodes represent points in business processes at which the business processes send messages to resources via controls.
- **Control Receive** nodes represent points in business processes at which the business processes receive asynchronous messages from resources via controls. Business processes wait at these nodes until they receive a message from the specified control.
- **Control Send with Return** nodes handle synchronous exchange of messages between business process and resources via controls.

To learn how to add **Control** nodes to your business processes, see Create Control Nodes in Your Business Process.

Designing the Communications for Control Nodes

Node builders provide task-driven interfaces that allow you to specify the logic required at the nodes in your business process. Control nodes provide control-specific node builders. The tasks you must complete to design the interaction with your resource depend on which control you use and the methods it exposes.

Designing the communications between your business process and resources includes adding instances of controls to your business process project, then designing the interaction with the controls at the appropriate point in the business process. To learn how, see:

- Adding Instances of Controls to Your Business Process Project
- Configuring Control Nodes

To help you specify the communication with a given control, customized interfaces are provided for controls. To learn about specific controls, see the following topics:

- Integration controls:
  - Dynamic Transformation Control
  - ebXML Control
  - Email Control
  - File Control
  - Http Control
  - Message Broker Controls
  - MQSeries Control
  - Process Control
  - RosettaNet Control
  - Service Broker Control
  - ALSB Control
  - TPM Control
  - WLI JMS Control

  - Worklist Controls
  - WLI Timer Control
  - XML MetaData Cache Control
- Additional built-in controls:
  - JDBC Control
  - EJB Control
  - JMS Control
  - Timer Control
  - Service Control

Using Integration Controls in Web Services or Page Flows

You can use a subset of the integration controls in web services and page flows. If you are licensed to use WebLogic Integration, you can use the following integration controls in a web service (JWS) or page flow (JPF): ALSB, Email, File, Process, Task, and Task Worker controls.

CHAPTER 2

Controls and Transactions

Business processes in WebLogic Integration are transactional in nature. Every step of a process is executed within the context of a JTA transaction. To learn about how transactions work within a business process, see Transaction Boundaries.

Some integration controls are transactional. This means that the control is able to participate in transactions within a business process. Whether or not a control is transactional depends on both the underlying resource and the specific control implementation. Also, transactional behavior differs depending on whether the control call is synchronous or asynchronous. To learn about synchronous or asynchronous operations in business processes, see Building Synchronous and Asynchronous Business Processes.

For synchronous control calls:

- If the control and associated resource are transactional, the resource participates in the current process transaction
- If the control and associated resource are not transactional, changes to the resource occur outside the scope of the current transaction and changes are not rolled back in case of failure

For asynchronous control calls:

- The process transaction is never propagated to the resource
- Asynchronous control calls are buffered by default
- Asynchronous call to the resource are not enqueued until the transaction is committed
- On rollback, asynchronous messages are de-queued

The Process control is a special case, since it involves processes calling subprocesses.

For synchronous operations:

- The transaction is always propagated to the subprocess
- An un-handled exception in a subprocess causes the shared transaction to be marked as rollback only. In this case, both the subprocess and the calling process are rolled back.
- Setting the process property `onSyncFailure=rethrow` on the subprocess overrides this behavior and results in the following:
  - Failure does not force a rollback
  - Subprocess throws an exception
  - Calling process catches the exception, just as with any other control exception

For asynchronous operations

- The transaction is not propagated to the subprocess
- The message is buffered on the subprocess' queue
- The subprocess runs in its own transaction
- The control call is successful if the message is properly enqueued on the subprocess' queue
- Failure of the subprocess is not communicated to the calling process. For example, an unhandled exception causes the subprocess to fail but the caller process is not notified

The following integration controls are transactional:

- ALSB
- ebXML
- Message Broker
- MQSeries
- Process (see the previously listed qualifications)
- RosettaNet
- WLI JMS
- Worklist

The following integration controls are not transactional:

- File
- Email
- Http
- Service Broker
- TPM

Good Practice in Creating Web Service Controls for a Business Process Application

When you call Web Service controls asynchronously from business processes, it is recommended that you buffer the asynchronous call. After creating the Web Service control, specify that the asynchronous calls from the business process to the control are buffered. By doing so, you ensure that the message sent from the business process to the Web service is enqueued. An asynchronous call to a resource marks the boundary of a transaction in your business process; a call to a resource is not enqueued until the transaction is committed. In other words, by buffering the call to the resource, you ensure that the transaction is committed before any response from the resource is attempted. If you do not buffer the call, your business process must wait for the HTTP acknowledgement to occur before the transaction is committed, leaving open the possibility that the resource attempts to respond to the business process before the HTTP acknowledgement occurs.

For an example of buffered asynchronous calls to Web Services, see how the taxCalculation, priceProcessor, and availProcessor Web Service controls are used in Tutorial: Building Your First Business Process.

Dynamic Transformation Control

Note: The Dynamic Transformation control is available in BEA WorkSpace Studio only for licensed users of WebLogic Integration.

A Dynamic Transformation control provides a business process with the ability to dynamically select and implement a transformation during run time. Specifically, it provides the ability to choose which XQuery, XSLT, or MFL file is invoked at run time. For example, if you have an integration hub that receives documents from different regional offices, you can use the Dynamic Transformation control to perform different transformations based on the area code of each regional office.

For general information about using controls, see Chapter 1, "Using Controls in Business Processes."

Topics Included in This Section

Overview: Dynamic Transformation Control

A Dynamic Transformation control provides a business process with the ability to dynamically select which transformation is invoked at run time.

This control provides eight base methods and the ability to create custom methods for selecting which file to use for your transformation. You can base the selection on the method context or the content of the message. The dynamic selection of the transformation is done through setting URIs that point to different transformation files. The available types of transformations are as follows:

- XQuery—contained in `.xq` files
- XSL—contained in `.xsl` files
- MFL—contained in `.mfl` files

The Dynamic Transformation control uses transformations that have already been created, such as those created with the Transformation control. Be sure to test your transformation before creating a Dynamic Transformation control. To learn more about Transformation controls and transforming data, see Transformation Data Using XQuery.

Before calling a Dynamic Transformation control, in the process of developing your business process, you need to design the logic that determines which transformation to run. You can use a Decision node or other controls, such as the Database or XML Metadata controls, for this purpose.

The Dynamic Transformation control provides the following methods:

- **Dynamic XQuery Transformation**
  - Method with HashMap. Use this base method to select the transformation when the argument type and values are unknown at design time. This method requires that you provide a HashMap with the variables and the types along with the XQuery file. The XML documents can be either typed or untyped. Use a **Perform** node to specify these values, as shown in the following:

```
public void perform() {
_map = new java.util.HashMap();
// The name should be the variable Name and the value the Document
passed
_map.add("CustomerDoc",customerDoc);
}
```

  - Method with XmlObjects In XmlObjects Out. Use this base method to choose which XQuery to use for transforming untyped XML documents to untyped XML documents.
  - Custom method. When you know the input types (arguments) at design time, you can define a custom method to choose which XQuery file to use for typed, untyped, or Java class documents.

- **Dynamic XSLT Transformations**
  - Method with HashMap. Use this base method to select the transformation when the argument type and values are unknown at design time. This method requires that you provide a HashMap with the variables and the types along with the XSL file. The XML documents can be either typed or untyped. Use a Perform node to specify these values. See Dynamic XQuery Transformation.
  - Custom method. When you know the input types (arguments) at design time, you can define a custom method to choose which XSL file to use for untyped documents (XmlObject).

- **MFL**—Five base methods provide the ability to select which MFL is invoked at run time based on the message context. Both binary-to-XML and XML-to-binary transformations are supported.

Note: To learn about the available base methods, see Using the Base Methods in the Dynamic Transformation Control.

Related Topics

Transformation Data Using XQuery

Creating a New Dynamic Transformation Control

This topic describes how to create a new Dynamic Transformation control.

For information on where to add control instances in a business process, see "Adding Instance of Controls to Your Business Process Project".

To Create a New Dynamic Transformation Control

Note: Before creating a Dynamic Transformation control, during design time, you need to create or import and test your transformation files. To learn more, see Guide to Data Transformation.

1. Open your WebLogic Integration application in BEA WorkSpace Studio.

2. In the Package Explorer pane, double-click the business process (Process.java file) to which you want to add the Dynamic Transformation control. The business process is displayed in the **Design** view.

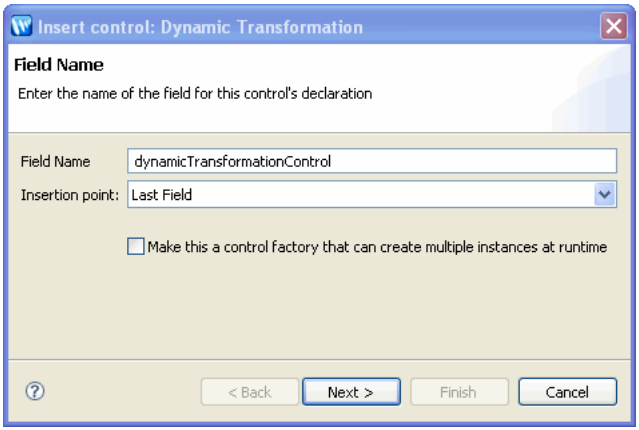
3. Click on the Data Palette and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the **Data Palette** is not visible, from the menu bar, click **Window** > **Show View** > **Data Palette**.

4. Select **Dynamic Transformation**.

   The **Insert Control: Dynamic Transformation** dialog box is displayed (see Figure 3-1).

Figure 3-1 Insert Control: Dynamic Transformation



5. In the **Insert Control: Dynamic Transformation** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new Dynamic Transformation control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point**, from the drop-down list select the insertion node, you want to add the control.

   – Decide whether you want to make this a control factory by selecting or clearing the **Make this a control factory that can create multiple instances at runtime** check box.

   – Click **Next**.

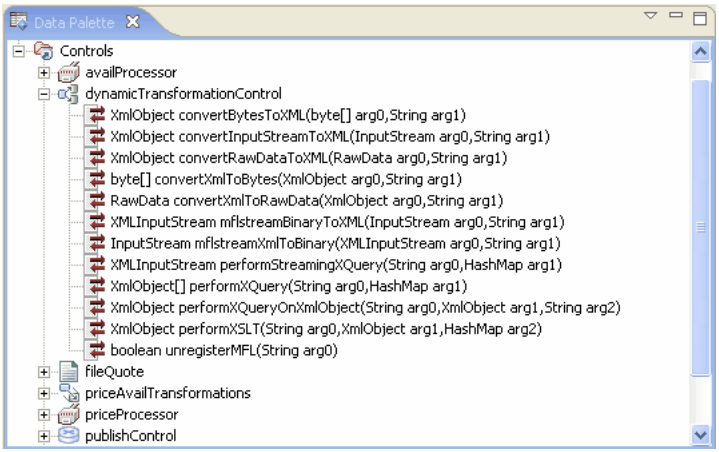
The **Create Control** dialog-box appears.

6. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Finish**.

   A new Dynamic Transformation control and an instance of it are created and the instance of the control is displayed on the **Controls** tab in the Data Palette. The new file is displayed in the Package Explorer pane in BEA WorkSpace Studio. (You can double-click any control file to view or edit it in the Source view.)

7. To display the base methods provided on a Dynamic Transformation control, expand the control instance by clicking the + beside its name on the Data Palette (see Figure 3-2).

Figure 3-2 Control Instances



8. After you create an instance of the Dynamic Transformation control in your business process, you can do the following:
   - Design the interaction of the business process with the Dynamic Transformation control by simply dragging and dropping the base methods from the Data Palette onto the Design view at the point in your business process at which you want to design the interaction. To learn more, see Using the Base Methods in the Dynamic Transformation Control.

– View and edit the properties of the control type or the instance of that control type in the **JPD Configuration** pane. The control type is represented as a Java file in the **Package Explorer** pane and the instance is represented in the **Data Palette**. To learn how to use the **JPD Configuration** pane for specifying properties for control types versus control instances, see Setting Control Properties and Annotations.

Using a Dynamic Transformation Control

This section describes ways to use a Dynamic Transformation control. It contains the following topic:

Using the Base Methods in the Dynamic Transformation Control

Once you have created the Dynamic Transformation control, you can add a control instance to your business process and invoke its base methods from within your application.

To learn about adding an instance of the Dynamic Transformation control to your business process, see Create Control Nodes in Your Business Process.

The following methods are available:

- `XmlObject[] performXQuery(String xQueryName,HashMap inputList)`

  Executes an XQuery and the arguments that it operates on based on message context.

- `XmlObject performXQueryOnXmlObject(String xQueryName,XmlObject inputDoc,String varName)`

  Executes an XQuery that takes an XmlObject in and XmlObject out.

- `XmlObject performXSLT(String xslTemplate,XmlObject source,HashMap inputList)`

  Executes an XSLT and the arguments that it operates on based on message context.

- `XmlObject convertBytesToXML(byte[] data,String mflUrl)`

  Executes MFL transformations.

- `XmlObject convertInputStreamToXML(InputStream iStream,String mflUrl)`

  Executes MFL transformations.

- `XmlObject convertRawDataToXML(RawData rawData, String mflUrl)`

  Executes MFL transformations.

- `byte[] convertXmlToBytes(XmlObject xmlObject,String mflURL)`

  Executes MFL transformations from XML data to non-XML data.

- `RawData convertXmlToRawData(XmlObject xmlObject,String mflURL)`

  Executes MFL transformations from XML data to non-XML data.

  Note: If the xq or mfl resource is required to be accessible from a different application and not your current application, the resource will have to be copied under WebContent of the Web Project of your current application.

The `@DynamicTransformationControl.Ddtf`, `@DynamicTransformationControl.Xquery`, `@DynamicTransformationControl.Xsl` annotations control the behavior of these methods. Additional information is also available in the Interface DynamicTransformation.

Related Topics

Creating Schema Projects

Example: Dynamic Transformation Control

To see an example of using a Dynamic Transformation control in a business process, see Solution Samples.

ebXML Control

Note: The ebXML control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

The ebXML protocol (Electronic Business using eXtensible Markup Language) is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. It is sponsored by UN/CEFACT and OASIS. To learn about ebXML, see `http://www.ebXML.org`.

The ebXML control enables WebLogic Workshop business processes to exchange business messages and data with trading partners via ebXML. The ebXML control supports both the ebXML 1.0 and ebXML 2.0 messaging services. You use ebXML controls in *initiator* business processes to manage the exchange of ebXML business messages with participants. For an introduction to ebXML solutions, see *Introducing Trading Partner Integration*.

Topics Included in This Section

Overview: ebXML Control
Describes the ebXML control.

Creating an ebXML Control
Describes how to create and configure a ebXML control.

Using an ebXML Control
Describes how to use an ebXML control in a business process.

Example: ebXML Control
Provides links to ebXML examples.

Related Topics

Introducing Trading Partner Integration

Trading Partner Management

Interface EBXMLControl

Building ebXML Participant Business Processes

Annotation Type EbXML

Annotation Type EbXML method

Overview: ebXML Control

You use ebXML controls in *initiator* business processes to exchange ebXML business messages with participants. The ebXML control provides methods for sending and receiving business messages, as described in Interface EBXMLControl. Callbacks handle ebXML messages, acknowledgements, and errors received from the participant.

You should *not* use ebXML controls in participant business processes to respond to incoming messages. Instead, you use **Client Request** nodes to handle incoming business messages from the initiator and **Client Response** nodes to handle outgoing business messages to the initiator. To learn about building participant business processes that use ebXML, see Building ebXML Participant Business Processes. To learn about designing business processes that use ebXML, see *Introducing Trading Partner Integration*.

At run-time, the ebXML control relies on trading partner and service information stored in the TPM repository. To learn about the TPM repository, see *Introducing Trading Partner Integration*.

To learn about adding or updating information in the TPM repository, see Trading Partner Management in *Using the WebLogic Integration Administration Console*.

Creating an ebXML Control

This topic describes how to create a new ebXML control. Each ebXML control instance represents a single ebXML conversation. For each separate ebXML conversation in a business process, you must add a *separate* ebXML control instance.

To create a new ebXML control

1. In the **Package Explorer** pane, double-click the business process to which you want to add the ebXML control. The business process is displayed in the **Design** view.

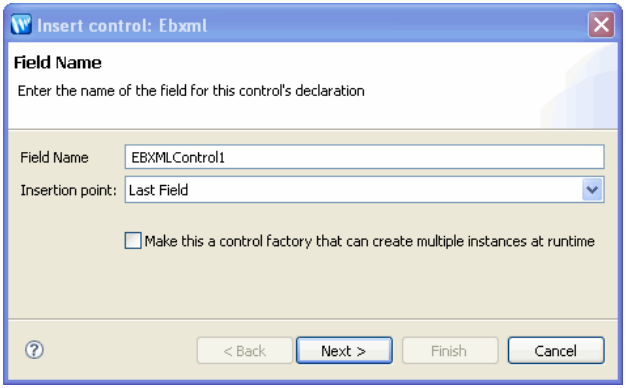
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the **Data Palette** view is not visible in BEA WorkSpace Studio, click **Window** > **Show View** > **Data Palette** from the menu bar.

3. Select **Ebxml**.

   The **Insert control: Ebxml** dialog box appears as shown in Figure 4-1.

Figure 4-1 Insert control: Ebxml



4. In the **Insert control: Ebxml** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new ebXML control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point:** from the drop-down list select the point where you want the field name to be inserted in the process file.

– Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

– Click **Next**.

The **Create Control** wizard appears.

5. In the **Create Control** wizard enter the following details:

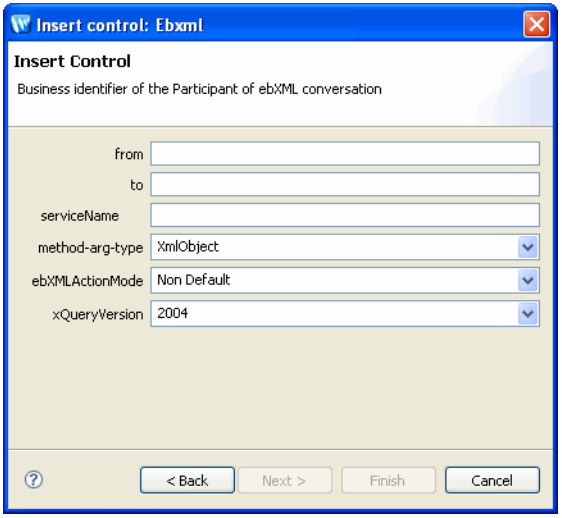
   – In the **Name** field, type the name of your new control extension file.

   – Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

   – Click **Next**.

   The **Insert control: Ebxml** dialog box appears (see Figure 4-2).

Figure 4-2 Ebxml Wizard



6. In the **Insert control: Ebxml** dialog box, specify the following information as shown in Table 4-1.

Table 4-1 Insert Control Properties

Field	Description
`from`	Optional. Business ID for the initiator in this conversation. One of the following values: • **Empty**—Uses the default trading partner. • **Static Value**—Business ID of the initiating trading partner. The specified business ID must be configured in the TPM repository. To specify the initiator business ID dynamically, use selectors or use the `setProperties` method in a **Control Send** node, as described in Dynamically Specifying Business IDs. You can also obtain this value by using XQuery selectors on process variables or method parameters in an incoming message. **Note:** If the value of the "from" trading partner is left unspecified via the to attribute, to-selector attribute, or via setProperties, the system will use the ID of the local trading partner marked as "default" in the TPM console.
`to`	Optional. Business ID for the participant in this conversation. One of the following values • **Empty**—Uses the default trading partner. • **Static Value**—Business ID of the participating trading partner. The specified business ID must be configured in the TPM repository. To specify the participant business ID dynamically, use selectors or use the `setProperties` method in a **Control Send** node, as described in Dynamically Specifying Business IDs. You can also obtain this value by using XQuery selectors on process variables or method parameters in an incoming message.
`serviceName`	Required. Name of an ebXML service. For initiator and participant business processes that participate in the same conversation, the settings for **`ebxml-service-name`** must be identical. This service name corresponds to the `eb:Service` entry in the ebXML message envelope.

Table 4-1 Insert Control Properties

`method-arg-type`	Required. Type of attachment. One of the following values: • `XmlObject`—Default. Represents data in untyped XML format. The XML data is not specified at design time. • `XmlObject[]`—Array containing one or more XmlObject elements. **Note:** The `XmlObject[]` option is not available from the drop-down menu on the control wizard window. It has to be specified in source view, see Specifying XmlObject and RawData Array Payloads. • `RawData`—Represents any non-XML structured or unstructured data for which no MFL file (and therefore no known schema) exists. • `RawData[]`—Array containing one or more RawData elements. **Note:** The `RawData[]` option is not available from the drop-down menu on the control wizard window. It has to be specified in source view, see Specifying XmlObject and RawData Array Payloads. • `MessageAttachment[]`—Array containing one or more parts of an ebXML business message. Message parts can be untyped XML data (`XmlObject` data type) or non-XML data (`RawData` data type). Used when sending different kinds of payloads (XML and non-XML) in the same message. The actual number of message parts might not be known until processed. To learn about working with `MessageAttachment` objects, see Using Message Attachments. To learn more about data types, see Working with Data Types.

Table 4-1 Insert Control Properties

`ebxmlActionMode`	Action mode for this ebXML control. Determines the value specified in the `eb:Action` element in the message header of the ebXML message, which becomes important in cases where multiple message exchanges occur within the same conversation. One of the following values: • `default`—Sets the `eb:Action` element to `SendMessage` (default name). • `non-default`—Sets the `eb:Action` element to the name of the method (on the ebXML control) that sends the message in the initiator business process. For sending a message from the initiator to the participant, this name must match the method name of the **Client Request** node in the corresponding participant business process. For sending a message from the participant to the initiator, the method name in the callback interface for the client callback node in the participant business process must match the method name (on the ebXML control) in the control callback interface in the initiator business process. Using `non-default` is recommended to ensure recovery and high availability. If unspecified, the `ebxml-action-mode` is set to `non-default`.
`xQueryVersion`	Specify the version of the XQuery used.

7. Click **Finish**.

   An ebXML control instance is displayed in the **Controls** tab.

After you create the control file, the name of the control file becomes available as a service on the Services tab in the BEA WorkSpace Studio.

Specifying XmlObject and RawData Array Payloads

The `XmlObject[]` and `RawData[]` payload options are only available in source view. You can configure your ebXML control to use these options after you have created it.

To Specify the Payload in Source View

1. Open your control definition file. You can do this by double-clicking on the file in the **Package Explorer** pane.

2. Click the **Source** view tab.
3. In the `request` and `response` methods, change the payload specified to the payload type that you want to use.

   The following restrictions apply to payload specifications:

   – If an array of any type is used, an argument of the same type cannot follow that array in the argument list. In other words, an array must be the last argument specified of that type.

   – If a `MessageAttachment[]` type is one of your arguments, no other array (including a `MessageAttachment[])` is allowed in the argument list.
4. After you have applied your changes, save and close your control definitions file.

Note: The order of arguments which you used in the control definition file and the order of the arguments in the node on the participant business process which is listening for your message must match.

To learn more about the `request` and `response` methods, see Interface EBXMLControl.

Using an ebXML Control

All WebLogic Workshop controls follow a consistent model. Many aspects of using ebXML controls are identical or similar to using other WebLogic Workshop controls.

After you have added an ebXML control to an initiator business process, you can use methods on the control to exchange ebXML messages with participant trading partners. In the Design View, you expand the node for the ebXML control in the Data Palette to expose its methods, and then drag and drop the methods you want onto the business process. Common tasks include:

- Sending Messages to Participants
- Handling Responses from Participants
- Dynamically Specifying Business IDs

To learn more about these methods, see Interface ebXML Control.

Sending Messages to Participants

To send an ebXML message to a participant, you use a send message method in a **Control Send** node. By default, the control file instance includes a generated send method named `request`. To add the **Control Send** node to a business process, you drag this method from the Data Palette onto

the business process. For business processes that involve multiple round-trips, you need to create a separate **Control Send** node for each operation that involves sending an ebXML message to the participant.

Note: The default return type for the `request` method is void. However, you can also specify the return type to be `XmlObject`. If you use `XmlObject` as the return type, the content the `XmlObject` is the ebXML envelope data.

After creating the **Control Send** node, you need to specify the payload parts and their Java data types (see Table 4-2).

Table 4-2 Data Types

Type	Description
`XmlObject`	Data in untyped XML format.
`XmlObject[]`	An array containing one or more `XmlObject` elements.
`RawData`	Any non-XML structured or unstructured data for which no MFL file (and therefore no known schema) exists.
`RawData[]`	An array containing one or more `RawData` elements
`MessageAttachment []`	Array containing one or more parts of an ebXML business message. Message parts can be untyped XML data (`XmlObject` data type) or non-XML data (`RawData` data type). Used when sending different kinds of payloads (XML and non-XML) in the same message. The actual number of message parts might not be known until processed. To learn about working with `MessageAttachment[]` objects, see Using Message Attachments.

Attachments can also be typed XML or typed MFL data as long as you specify the corresponding XML Bean or MFL class name in the parameter.

If you use arrays as attachment type, certain restrictions apply to the order of your arguments. For more informations, see Specifying XmlObject and RawData Array Payloads.

You can specify business IDs statically (using the @EBXMLControl.EBXML Annotation) or dynamically. To learn about specifying business IDs dynamically, see Dynamically Specifying Business IDs.

Handling Responses from Participants

Participants can respond to initiator requests in the following ways:

- acknowledge that the request was received
- reply to the request
- notify that an error occurred

To handle responses from participants, initiator business processes use the following callback methods (see Table 4-3).

Table 4-3 Callback methods

Method Name	Description
`onAck`	Handles the acknowledgement of the message receipt from the participant.
`onError`	Handles an error sent by the participant.
`response`	Handles the message reply sent by the participant.

To receive an ebXML message from a participant, you use the appropriate method. To add the method to a business process, you drag it from the **Data Palette** onto the business process, which creates a **Control Receive** node. For business processes that involve multiple round-trips, you need to create a separate **Control Receive** node for each operation that involves receiving an ebXML message from the participant.

For the `response` method, if you specify `non-default` in the `ebxml-action-node`, you can rename the **Control Receive** node to make it more descriptive, such as `getInvoice`. However, if you specify `default` in the `ebxml-action-node`, you must use the default name (`onMessage`) and the business process can have only one `onMessage` **Control Receive** node.

For the `response` method, after creating the **Control Receive** node, you need to specify the payload parts and their Java data type for the incoming message. To learn about valid data types, see Sending Messages to Participants.

The `onError` and `onAck` methods are system-level methods. Both use the `EnvelopeDocument` argument, which will contain an ebXML envelope when the message is received. As they are system-level methods, these arguments are not seen in the default control but you can drag them onto the business process from the **Data Palette**. If your application contains a schema project

that includes the `envelope.xsd` file, and if the schema is already built, you can extract the values you want by creating a query (in the XQuery language) using the mapper functionality of WebLogic Workshop. To learn about creating queries with the mapper functionality, see Transforming Data Using XQuery.

You can retrieve the message envelope of an incoming ebXML message by using the `envelope` annotation from the `com.bea.control.EBXMLControl.EbXMLMethod` based annotations. To learn more about the `envelope` annotation, see EBXMLControl.EbxmlMethod Annotation.

Dynamically Specifying Business IDs

The ebXML control adds the capability of dynamically binding business IDs for the initiator (`from` property) and the participant (`to` property) of the control. Dynamic binding of properties can be achieved the following ways:

- Using selectors
- Using the `setProperties()` method

Order of Precedence

The hierarchy of property settings is as follows, starting with the approach having the highest precedence:

1. Properties dynamically bound using selectors (see EBXMLControl.EbxmlMethod Annotation) and the `DynamicProperties.xml` file
2. Properties set using the `setProperties()` method
3. Properties set at the control instance level using the Annotation EBXMLControl.EbXML in the business process.
4. Properties set at control class level using Annotation EBXMLControl.EbXML in the Dynamic selectors have a higher precedence than static selectors.

Using Selectors

Using a dynamic selector, ebXML controls allow you to decide at run time which one of multiple trading partners to send a business message to. When you specify a dynamic selector, you build and test an XQuery that retrieves the business ID you need.

To use a dynamic selector

1. In the **Data Palette**, select the ebXML control.

2. In the **Properties** pane, locate the **from-selector** or **to-selector** attributes.
3. Select an element from the schema to associate it with the control. The resulting query appears in the XQuery area.

Using setProperties

The `setProperties` method accepts an `ebXMLPropertiesDocument` parameter. The `ebXMLPropertiesDocument` type is an XML Beans class that is generated out of the corresponding schema element defined in `DynamicProperties.xsd`. The `DynamicProperties.xsd` file is located in the schemas/system folder, or in the utility folder of the Process Applications.

If your application contains a schema project that includes the `DynamicProperties.xsd` file, and if the schema is already built, you can extract the values you want by creating a query (in the XQuery language) using the mapper functionality of WebLogic Workshop. To learn about creating queries with the mapper functionality, see Transforming Data Using XQuery.

To set business IDs dynamically using the setProperties method

1. Verify that your application contains a schema project that includes the `DynamicProperties.xsd` file, and that the schema is already built.
2. Create a **Control Send** node in a business process.
3. From the Data Palette, drag the `setProperties` method and drop it onto the **Control Send** node.
4. In the **Send Data** tab, select **Transformation**, specify variables that contain the `to` and `from` values, and then create a transformation to map them to the corresponding elements in `ebXMLPropertiesDocument`.

   To display the current property settings, use the `getProperties()` method.

Example: ebXML Control

For examples of how to use the ebXML control, see Tutorial: Building ebXML Solutions.

Email Control

Note: The Email control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

The Email control enables WebLogic Integration business processes to send e-mail to a specific destination. To receive e-mail, you must use the Email Event Generator. Use the WebLogic Integration Administration Console to create and manage event generators. To learn about creating and managing event generators, see Event Generators in *Using the WebLogic Integration Administration Console.*

For information on how to add control instances to business processes, see Using Controls in Business Processes.

Topics Included in This Section

Sample Email Messages
Provides sample e-mail messages with different formats.

Overview: Email Control

The Email control enables BEA WorkSpace Studio web services and business processes to send e-mail to a specific destination. The body of the e-mail message can be text (plain, HTML, or XML) or can be an XML object. The control is customizable, allowing you to specify e-mail transmission properties in an annotation or to use dynamic properties passed as an XML variable.

The Email control is flexible, allowing you to send a variety of content types and various combinations of body and attachments. For examples of e-mail messages that can be sent using the Email control, see Sample Email Messages.

Related Topics

EmailControl Interface

Email Control Annotations

Configuring an Email Control

When you add an Email control to your business process, you can use an existing Email control extension file (`.java`) or create a new one. Depending on the data type of the message body you select, the `.java` file includes one of the following `sendEmail` utility methods. (Note the different body types in the two methods.) You can specify the values for the fields as class annotations in the `.java` file.

Message Body with Data Type Text

```
@EmailControl.send to="{to}"
cc="{cc}"
bcc="{bcc}"
subject="{subject}"
body="{body}"
attachments="{attachments}"
content-type="text/plain"
void sendEmail(String to, String cc, String bcc, String subject,
               String body, String attachments);
```

Message Body with Data Type XML Object

```
@EmailControl.send to="{to}"
cc="{cc}"
bcc="{bcc}"
subject="{subject}"
body="{body}"
attachments="{attachments}"
content-type="text/xml"
void sendEmail(String to, String cc, String bcc, String subject,
               XmlObject body, String attachments);
```

Customizing an Email Control

Depending on the needs of your application, you can customize the base control. When extending the base control, you can add a method that specifies e-mail transmission properties in the annotation. The customized method does not require the user to supply as many parameters.

```
A custom Email control.
@EmailControl
smtp-address = "smtp.myorg.com:25"
from-address = "joe.user@myorg.com"
from-name = "Joe User"
reply-to-address = "reply@myorg.com"
reply-to-name = "Customer Service"
header-encoding=""
username=""
password=""
public interface MyEmailControl extends
EmailControl,com.bea.control.ControlExtension
@EmailControl.send to="{to}"
subject="Thanks for your order"
body="{body}"
attachments="/weblogic/samples/order.txt"
    public void sendOrderConfirmation(String to,
                                String body);
```

Using Dynamic Properties for an Email Control

You can override class-level annotations for an Email control by using dynamic properties. To use dynamic properties, pass an XML variable that conforms to the control's dynamic-property schema to the control's `setProperties()` method. You can retrieve the current property settings using the `getProperties()` method.

The `setProperties()` method accepts an EmailControlPropertiesDocument parameter. The EmailControlPropertiesDocument type is an XML Beans class that is generated out of the corresponding schema element defined in `DynamicProperties.xsd`. The `DynamicProperties.xsd` file is located in the system folder of New Process Applications or in the system folder of the Schemas project.

The following is an example of an XML variable used to set dynamic properties:

```
<EmailControlProperties>
    <smtp-address>myorg.mymailserver.com:25</smtp-address>
    <from-name>Joe User</from-name>
    <from-address>joe.user@myorg.com</from-address>
    <reply-to-address>reply@myorg.com</reply-to-address>
    <reply-to-name>Joe User</reply-to-name>
</EmailControlProperties>
```

Creating a New Email Control

This topic describes how to create a new Email control.

To learn about WorkSpace Studio controls, see Using Controls in Business Processes.

To create a new Email control:

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the Email control. The business process is displayed in the **Design** view.

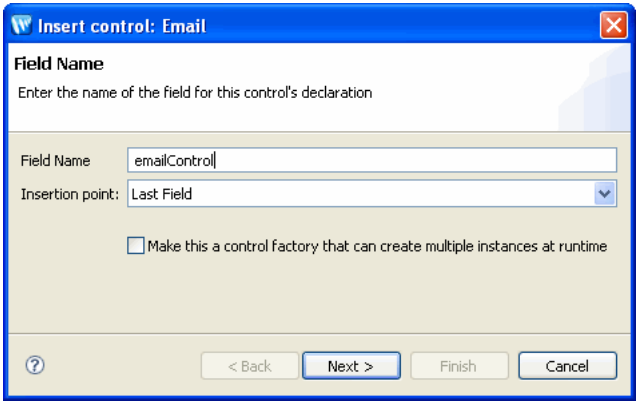
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the **Data Palette** view is not visible in BEA WorkSpace Studio, click **Window** > **Show View** > **Data Palette** from the menu bar.

3. Select **Email**.

   The **Insert control: Email** dialog box appears.

Figure 5-1 Insert control: Email



4. In the **Insert control: Email** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new Email control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

     The **Create Control** dialog-box appears.
5. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the class name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.
6. In the **Insert control- Email** dialog box enter the following name and address parameters:
   - **SMTP host**—The address of the SMTP server in *`host:port`* or *`host`* form. If the port is not specified, the standard SMTP port of 25 is used.
   - **From address**—The originating e-mail address

  - **From name**—The Display name for the originating e-mail address
  - **Body type**—Select the type of data contained in the message body.
  - Click **Finish**.

If you need to specify reply information (name and address) or SMTP authentication parameters (username and password or password alias), assign values to the following optional parameters using the Properties pane:

- **reply-to-address**—The e-mail address to reply to
- **reply-to-name**—The display name for the reply to address
- **header-encoding**—A string specifying the encoding to be used for the mail headers as specified by `from-name`, `reply-to-name`, `to`, `bc`, `bcc`, `subject`, and `attachments`. If no header encoding is specified, the system default encoding is used.
- **username**—The username for servers that require authentication to send.
- **password**—The password associated with the `smtp-username`.
- **password-alias**—The password alias associated with the `smtp-username`. The alias is used to look up the password in the password store. This attribute is mutually exclusive with the `smtp-password` attribute.

Note: If the Properties pane is not visible in BEA WorkSpace Studio, choose **Window** > **Show View** > **Properties** from the menu bar.

Email Control Methods

To learn about the methods available on the Email control, see the EmailControl Interface.

Sample Email Messages

The following samples show what types of messages can be sent using the Email control.

Example 1: HTML Body, No Attachments

If the supplied String body is an HTML document, you can set the content-type annotation attribute to generate the following e-mail.

```
To: user@myorg.com
Subject: Thanks for your order
Content-Type: text/html
```

```
<html>
<head>
<title>Thanks for your order</title>
...
```

Example 2: Body with Attachments

For a message body with attachments, the Email Control generates a `multipart/mixed` message with the message body as the first part. Attachments are added as MIME parts with content types in accordance with their file name suffix. Table 5-1 lists commonly used file suffixes.

Table 5-1 File Suffixes

Suffix	Content-Type
.doc	application/msword
.gif	image/gif
.html	text/html
.jar	application/java-archive
.jpg	image/jpeg
.pdf	application/pdf
.txt	text/plain
.xls	application/msexcel
.xml	application/xml or text/xml
.zip	application/x-zip-compressed

Attachments with unknown extensions receive the `application/octet-stream` MIME type. The Email control also base64 encodes attachments which include binary data, as shown in the following example:

```
To: user@myorg.com
Subject: Thanks for your order
Content-Type: multipart/mixed;
boundary="------------F141E40DDE2763DF92513DD4"
```

```
------------F141E40DDE2763DF92513DD4
Content-type: text/plain; charset=us-ascii

Dear Sir,

Please see the attached diagram and brochure.

Thanks,
Customer Service

------------F141E40DDE2763DF92513DD4
Content-type: image/jpeg;
        name="picture.jpg"
Content-Disposition: attachment; filename="picture.jpg"
Content-transfer-encoding: base64

/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAPAAA/+4ADkFkb2JlA
...

------------F141E40DDE2763DF92513DD4
Content-Type: application/pdf;
        name="brochure.pdf"
Content-Transfer-Encoding: base64
Content-Disposition: inline;
        filename="brochure.pdf"

JVBERi0xLjIgDSXi48/TDQogDTEwIDAgb2JqDTw8DS9MZW5ndGggMTEgMCBSDS9Ga
...

------------F141E40DDE2763DF92513DD4
```

Example 3: No Body, One Attachment

An Email control send action with no body and one attachment does not generate an `multipart/mixed` message. This supports interchange scenarios that require the XML document to be in the message body.

```
To: inbox@myorg.com
Subject: new XML order
Content-Type: application/xml

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

Exceptions and Errors

You can use an exception handler to catch and deal with any exceptions that are thrown by the Email control.

If one or more of the `To` or `cc` recipients is determined to be invalid by the local mail server, an exception may be thrown immediately. However, if the invalid recipients can only be detected by the destination mail server, this is out of the scope of the Email control. We recommend that the `From` address be a mailbox for handling messages bounced back to the sender.

If one or more of the attachment file names is not found, an exception is thrown.

File Control

Note: The File control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

A File control makes it easy to read, write, or append to a file in a file system. The topics in this section describe how to work with the File control. For information on how to add control instances to business processes, see Using Controls in Business Processes.

Topics Included in This Section

Overview: File Control
Provides an overview of the File control.

Creating a New File Control
Describes how to create a new File control using the BEA WorkSpace Studio graphical design interface.

Using a File Control

Describes how to use a File control in your business processes. Describes the default methods and the methods you can customize.

Using File Control for SFTP
Describes how to use SSH File Transfer Protocol (SFTP) with your File control.

Service Provider Interface

Provides an example of a File control in the context of a business process.

Overview: File Control

A File control makes it easy to read, write, or append to a file in a file system. The files can be one of the following types: XmlObject, RawData (binary), or String. When creating a File control, select the file type that matches the files present in the specified directory.

In addition, the File control supports file manipulation operations such as copy, rename, and delete. You can also retrieve a list of the files stored in the specified directory.

Creating a New File Control

A File control performs an operation on a file. Each File control is customized to perform certain operations.

This topic describes how to create a new File control and provides an example of the File control's declaration in the java file.

For information on how to add control instances to business processes, see Using Controls in Business Processes.

Creating a New File Control

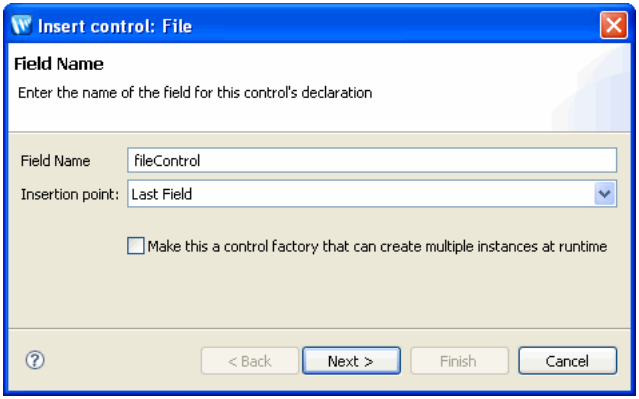
You can create a new File control and add it to your business process. To define a new File control:

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the File control. The business process is displayed in the **Design** view.
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.
3. Select **File**.

   The **Insert control: File** dialog box appears.

Figure 6-1 Insert Control: File



4. In the **Insert control: File** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new File control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

   The **Create Control** dialog-box appears.

5. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.

     The **Insert control -File** dialog-box appears.

6. In the **Insert control -File** dialog-box enter the following.
   - In the **Directory Name** field, enter the name of the directory where the File control looks for files. Alternatively, you can click the **Browse** button to locate a directory on your hard disk.

A directory name is the absolute path name for the directory; it includes the drive specification as well as the path specification. For example, the following are valid directory names:

```
C:\directory (Windows)
```

```
/directory (Unix)
```

```
\\servername\sharename\directory (Win32 UNC)
```

You can also enter a period (`.`), which specifies the current working directory. When you enter a forward slash (`/`) in the **Directory Name** field, it is interpreted as follows:

- UNIX systems—the root directory
- Windows systems—the root of the user directory (for example, `C:` if the user directory is `C:\bea`).

The **Directory Name** field is required. Leaving the **Directory Name** field empty results in an error.

Note: When writing files locally, if the specified directory does not already exist, it is created and the file is written into the new directory.

– In the **File name filter** field, enter the file name filter, either a file name or file mask. Use file names for read, write and append operations. If the file name field contains a wild-card character, such as an asterisk (`*`), it is treated as a file mask. A wild-card character is specified to get the list of files in a directory. Wild-card characters are not valid for any other operation.

  The **File name filter** field is optional when inserting a control, but this property must then be set dynamically before performing a file operation.

– Select the type of data contained in the file using the **Type of Data** menu. The file type indicates the type of files present in the directory specified in the **Directory Name** field. Based on this type, appropriate methods (such as `write(String` *`data`*`)` or `write(XmlObject` *`data`*`)` or `write(RawData` *`data`*`)`) are generated for the File control. For example, if the directory contains XML documents, the type should be set to XmlObject so that read/write methods generated for the control will accept XmlObject variables. The same is true for RawData and String types.

– If you are operating on a file of type String or XmlObject, you can optionally specify the character set encoding by entering the character set code in the **Encoding** field. This option can not be used with the large files option.

– If the specified directory contains files you want to read one line at a time, select the button labeled **The directory contains large files to be processed**. The resulting `readLine()` method is created with support for large files.

You can define a line by specifying either its record size or a delimiter string:

- If you enter a record size in the **Record Size** field, the file is read that number of bytes at a time.
- If you are operating on a file of type String and you enter characters in the **Delimiter string** field, the file is read by record with each record defined as being terminated by that delimiter. Click **Read file content including delimiter string** to include the delimiter in the record.

  WARNING: If the specified delimiter string does not exist in a file being processed, application behavior is unpredictable.

  If no record size or string delimiter is specified, the file is processed one line at a time. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed. This style of file processing can be used with any size file.

  Note: You cannot define a line by specifying both a record size and a string delimiter.

– Click **Finish**.

File Control Methods

To learn about the methods available on the File control, see the Interface FileControl.

Example: File Control Declaration

When you create a new File control, its declaration appears in the java file. The following code snippet is an example of the File Control declaration:

```
package requestquote;
import com.bea.control.FileControl;
import org.apache.beehive.controls.api.bean.ControlExtension;
import
com.bea.wli.control.dynamicProperties.FileControlPropertiesDocument;
/*
 * A custom File control.
 */
@ControlExtension
@FileControl.FileInfo(directoryName = "C:/bea/")
```

```
public interface FileCntrl extends com.bea.control.FileControl {


@FileControl.IOOperation(ioType = FileIOType.READLINE, delimiterString =
"")
String readline();


static final long serialVersionUID = 1L;
public FileCntrl create();
}
```

The actual attributes that are present on the `@FileControl.FileInfo` and `com.bea.control.FileControl.IOOperation` annotations depend on the values you entered in the Insert Control dialog.

The `@FileControl`annotation controls the behavior of the File control. All of the attributes of the `@FileControl` annotation are optional and have default values.

To learn more, see FileControl Annotation.

The File control, named `TaxControlFile` in the example above, is declared as an extension of `FileControl`. The `com.bea.control.FileControl.IOOperation` annotations indicate that the file operation is readline (read `tax_file.txt` record by record) and specifies the record size.

Using a File Control

A File control performs operations on a file such as reading a file, writing a file, and appending data to a file. You can also use the File control to copy, rename, and delete files.

You usually configure a separate File control for each file you want to manipulate. You can specify settings for a File control in several different ways. One way is to set the File control's properties in Design view. Another way is to call the `setProperties` method of the `FileControl` interface. You can change File control configuration properties dynamically. To get the current property settings, use the `getProperties()` method.

The following sections describe how to configure the File control.

Setting Default File Control Behavior

You can specify the behavior of a File control in Design View by setting the control's properties in the Properties pane. These properties correspond to attributes of the `@FileControl` and

`@FileControl.Operation` annotations, which identify the File control in your code. The following attributes specify class- and method-level configuration attributes for the File control.

Table 6-1 Configuration

Annotation	Attribute	Description
`@FileControl.FileInfo`	`createMode`	Specifies whether a file is overwritten or renamed when a new file of the same name is created. **Note:** When you use `create-mode="rename=old"` to rename a file, make sure that you mention the `suffix-name` and the `suffix-type` attributes for the new file name. If the suffix attributes are not indicated, then the File control overwrites the old file, instead of renaming it.
	`directory-name`	The absolute path name for the directory. (When writing files locally, if the specified directory does not already exist, it is created and the file is written into the new directory.)
	`file-mask`	Either a file name or a file mask.
	`suffix-name`	Suffix to be used with a timestamp or incrementing index for creating file names.
	`suffix-type`	Specifies whether a timestamp or an incrementing index should be used as a suffix for file names.

Table 6-1 Configuration

Annotation	Attribute	Description
`FileControl.FTP`	*`hostName`*	Name of the FTP host, for example, `ftp://ftp.bea.com`.
	`localDirectory`	Directory used for transferring files between the remote file system and the local file system. When reading a remote file, the file is copied from the remote system to the local directory and then read. Similarly, when writing to a remote file system, the file is written to the local directory and then copied to the remote system. **Note:** This is a temporary working directory for the File control. It should not be treated as a user-archive directory. The contents of `ftp-local-directory` are deleted after the FTP operation is performed.
	`password`	FTP user's password. If you specify this attribute, you cannot specify the *`ftp-password-alias`* attribute.
	`passwordAlias`	Alias for a user's password. The alias is used to look up a password in a password store. If you specify this attribute, you cannot specify the *`ftp-password`* attribute.

Table 6-1 Configuration

Annotation	Attribute	Description
	userName	Name of the FTP user.
@FileControl.IOOperation	*encoding*	Character set encoding of the file.
	fileContent	Contents of the identified variable which will be written to the file.
	ioType	Type of file operation (read, readline, write, or append).
	recordSize	Size of an individual record (in bytes) within a file to be processed record by record.

For information on `FileControl.SFtp` annotation, see "File Control Annotations for SFTP".

When you use the binary files for transferring using `FtpToLocal()` of file control, place the annotation listed below in your Source view.

```
@Retention(RetentionPolicy.RUNTIME)
      @Target({ElementType.TYPE, ElementType.METHOD})
      @PropertySet(
              prefix = 'FileTransferMode',
              externalConfig = false,
              optional = true,
              hasSetters = false
              )
      public @interface FileTransferMode
      {
          /**
           * This option specifes SFTP properties as a list of name/value
  pairs
           */
          @FeatureInfo(shortDescription = 'File Transfer Mode when using
```

```
SFTP/FTP')
        TransferMode value() default TransferMode.BINARY;
    }
```

Note: If you dont use the above annotation, the binary files will get corrupted. The above procedure should also be followed while using SFTP.

To learn more about specifying default File control behavior with attributes of the `@FileControl` annotation, see FileControl Annotation.

Using Methods of the FileControl Interface

Once you have declared and configured a File control, you can invoke its methods from within your application to perform file operations and to change its configuration. For complete information on each method, see the Interface FileControl.

Use the following methods of the FileControl interface to perform file operations and reconfigure the File control.

Table 6-2 Methods

Method	Description
setProperties	Sets the properties for the control
getProperties	Gets the properties for the control
getFiles	Returns the `FileControlFileListDocument` XML Beans document defined in `DynamicProperties.xsd`
rename	Renames the current file
delete	Deletes the current file
copy	Copies the current file to a different location
reset	Reset the control by closing any operations in progress, such as `readLine`, `readRecord` and `append`.

The File control does not provide callbacks to wait for a file to appear. If the business process needs to wait for a file to appear, use the File Event Generator functionality. The business process can use the Message Broker Subscribe control to subscribe to a channel if it is interested in processing the files in a given directory. A File Event Generator is then configured so that when

a file appears in that directory, it publishes a message to the associated channel containing the contents of the file.

Error Handling When Reading Files

The File control invokes an error handler when exceptions are encountered in `read()` methods. (Exceptions can occur when the contents of the file are invalid.) The error handler moves the file to an error directory. However, if the error directory is not configured, the error handler throws the following exception: File or Directory does not exist. To ensure that useful information about the exception is available, the exception thrown by the error handler is logged and appears on the WebLogic Server Console and the original exception is rethrown.

Using File Control for SFTP

SSH File Transfer Protocol (SFTP) is a communication protocol that provides secure and reliable file transfer capabilities. It is generally used with the SSH-2 protocol.

File control provides support to read, write, and append to a file on the file system or the FTP server. It supports copying, renaming, and deleting files on the file system and FTP server. In addition, you can manipulate files on the SFTP server using file control.

SFTP includes a Service Provider Interface (SPI), which allows third-party client implementations of SFTP to plug in to the file event generator and the file control. The default SPI implementation that is provided with WLI 10.2 uses the SFTP Client API (J2SSH) from SSHTools. It supports both RSA and DSA key pairs.

The Default SFTP SPI implementation using J2SSH supports the following authentication methods:

- Password Authentication – In this method, users are authenticated based on the credentials (username/password pair) provided by them. The username/password pair is sent over an encrypted connection to the server, and the credentials are validated.
- Host-based Authentication – In this method, users are authenticated based on the common private host key. You must configure the SFTP server with all the managed servers and their public keys within the cluster as the request for connection or authentication can originate from any of the managed servers.
- Public Key Authentication – In this method, users are authenticated based on their private key. In a clustered environment, the private key that is configured to connect to the SFTP server must be accessible from all the managed servers in the cluster.

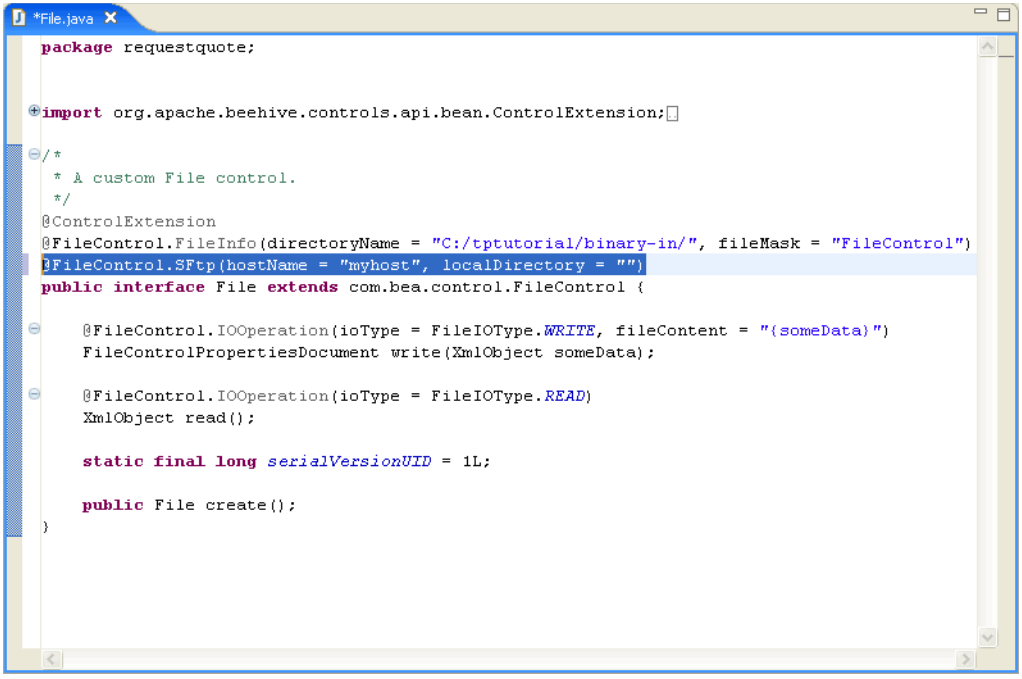
Note: SPI can be extended to support any other authentication method.

Configuring File Control for SFTP

To configure file control for SFTP, complete the following steps:

1. Create a file Control, as described in Creating a New File Control.
2. Right-click the file control, and select Edit.
3. Add `@FileControl.SFtp` to the code in the Source view, and add the annotations, as required (see Figure 6-2).

Figure 6-2 Adding SFTP to File Control



File Control Annotations for SFTP

File control supports the following annotations:

SFtpCustomProperty- You can use this annotation to specify any additional properties for authentication and to define a name/value pair. The annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
@PropertySet(
        prefix = "SFtpCustomProperty",
        externalConfig = false,
        optional = true,
        hasSetters = false
        )
public @interface SFtpCustomProperty
{
        @FeatureInfo(shortDescription = "SFTP property name")
        @AnnotationMemberTypes.Text
        String name() default Constants.ANNOTATION_VAL_NOT_SPECIFIED;

        @FeatureInfo(shortDescription = "SFTP property value")
        @AnnotationMemberTypes.Text
        String value() default Constants.ANNOTATION_VAL_NOT_SPECIFIED;
}
```

The annotation attributes are as follows:

- name: The name of the property used for connecting or authenticating with the SFTP server.
- value: The value for the property used for connecting or authenticating with the SFTP server.

  Note: This annotation must be used only when the `@SFtp` annotation is used.

- **SFtp-** You can use this annotation to specify the properties of the SFTP server so that file control can interact with the SFTP server using the properties in this annotation. This annotation specifies the authentication method used to connect to the SFTP server, and the local directory in which files downloaded from the SFTP server are copied. The annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
```

```
@PropertySet(
            prefix = "SFtp",
            externalConfig = false,
            optional = true,
            hasSetters = false
            )
public @interface SFtp {
        @FeatureInfo(shortDescription="specifies the name of the SFTP
host, for example, sftp://sftp.bea.com")
        @AnnotationMemberTypes.Text
        String hostName() ;


       @FeatureInfo(shortDescription = "specifies the SFTP server port
number")
        @AnnotationMemberTypes.Int
        int port() default 22;


        @FeatureInfo(shortDescription = "specifies the authentication
method to use for authenticating to SFTP server")
        SFtpAuthMethod authMethod() default
SFtpAuthMethod.PASSWORD_BASED_AUTHENTICATION;


        @FeatureInfo(shortDescription="specifies the login name of the
SFTP user")
        @AnnotationMemberTypes.Text
       String userName()default Constants.ANNOTATION_VAL_NOT_SPECIFIED;


       @FeatureInfo(shortDescription="the SFTP user's password in case
of PASSWORD_BASED_AUTHENTICATION")
        @AnnotationMemberTypes.Text
      String password() default Constants.ANNOTATION_VAL_NOT_SPECIFIED;


        @FeatureInfo(shortDescription="alias for a user's password")
```

```
        @AnnotationMemberTypes.Optional

        @AnnotationMemberTypes.Text

        String passwordAlias()default
Constants.ANNOTATION_VAL_NOT_SPECIFIED;


        @FeatureInfo(shortDescription = "path to the user/host private
key file")

        @AnnotationMemberTypes.Text

        String privateKey() default
Constants.ANNOTATION_VAL_NOT_SPECIFIED;


       @FeatureInfo(shortDescription = "Passphrase for the private key")

        @AnnotationMemberTypes.Text

        String passphrase() default
Constants.ANNOTATION_VAL_NOT_SPECIFIED;


        @FeatureInfo(shortDescription = "alias for the private key's
passphrase")

        @AnnotationMemberTypes.Text

        String passphraseAlias() default
Constants.ANNOTATION_VAL_NOT_SPECIFIED;


       @FeatureInfo(shortDescription = "additional SFTP properties as a
list of name/value pairs")

        @AnnotationMemberTypes.Optional

        SFtpCustomProperties customProperties()default
@SFtpCustomProperties({});


       @FeatureInfo(shortDescription="local directory when transferring
files between remote file system and the local directory")

        @AnnotationMemberTypes.Text

        String localDirectory() ;

    }
```

The following annotation attributes appear in the **Properties pane** of BEA WorkSpace Studio (see Figure 6-3):

Figure 6-3 Properties Pane

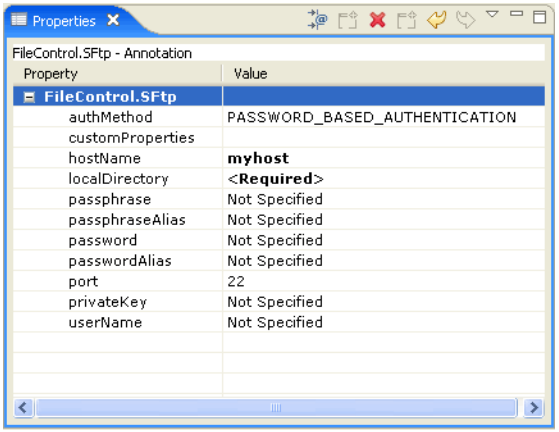


Table 6-3 SFTP Configuration

Annotation	Attribute	Description
`FileControl.SFtp`	`authMethod`	Specifies the authentication method for connecting to the SFTP server. The supported authentication methods are password based, host based, public key based and other. The default authentication method is password based.

Table 6-3 SFTP Configuration

Annotation	Attribute	Description
	`customProperties`	Represents any additional properties required for authenticating with the SFTP server. You can specify these as a list of name/value pairs by using the annotation `@SFtpCustomProperties`. This attribute enables you to use any other authentication method that is not provided in WLI 10.2, if `authMethod` is set to `OTHER_AUTHENTICATION_METHOD`. If necessary, you can specify additional properties when `authMethod` is set to either `HOST_BASED_AUTHENTICATION` or `PUBLIC_KEY_BASED_AUTHENTICATION`.
	`hostName`	The name or IP address of the SFTP server that the file event generator or control communicates with.
	`localDirectory`	The path to the directory on the local file system in which files downloaded from the SFTP server are copied.

Table 6-3 SFTP Configuration

Annotation	Attribute	Description
	passphrase	The pass-phrase for the private key if the key is protected with a pass-phrase. If this attribute is specified, then passphraseAlias must not be specified. The attribute is required only when authMethod is set to either HOST_BASED_AUTHENTICATION or PUBLIC_KEY_BASED_AUTHENTICATION.
	passphraseAlias	The password alias for the pass-phrase if the private key is protected with a pass-phrase. If this attribute is specified, then the passphrase attribute must not be specified. The attribute is required only when the authMethod is set to either HOST_BASED_AUTHENTICATION or PUBLIC_KEY_BASED_AUTHENTICATION.
	password	The password used for authenticating with the SFTP server. If this attribute is specified, then passwordAlias must not be specified. It is required only when authMethod is set to PASSWORD_BASED_AUTHENTICATION.

Table 6-3 SFTP Configuration

Annotation	Attribute	Description
	passwordAlias	The password alias for authenticating with the SFTP server. If this attribute is specified, the password must not be specified. It is required only when authMethod is set to PASSWORD_BASED_AUTHENTICATION.
	port	The port number on which the SFTP daemon is running. The default port number is 22.
	privateKey	The path to the private key file for authenticating with the SFTP server. If the authMethod attribute is set to HOST_BASED_AUTHENTICATION, then this attribute represents the host's private key. If authMethod is set to PUBLIC_KEY_BASED_AUTHENTICATION, then this attribute represents the private key of the users. The attribute is required only when the authMethod is set to either HOST_BASED_AUTHENTICATION or PUBLIC_KEY_BASED_AUTHENTICATION.

Table 6-3 SFTP Configuration

Annotation	Attribute	Description
	`userName`	The user name used for authenticating with the SFTP server. This attribute is required in all the authentication methods that are supported in WLI 10.2.

Service Provider Interface

WLI provides a framework for SFTP, and you can plug in your own libraries. To install SSHTools, do the following:

1. Download `J2SSH 0.2.9`, from the following location: http://sourceforge.net/projects/sshtools to your local directory.
2. Copy the `j2ssh-core-0.2.9.jar` and `j2ssh-common-0.2.9.jar` to the following directory: *`BEA_HOME`*`\wli_10.2\lib` directory.

   Note: In this pathname, *`BEA_HOME`* represents the directory in which you installed WebLogic Integration, typically `C:\bea\wli_10.2`.

SPI includes the following enumeration types:

SFtpVersion

This enumeration represents the SSH version and defines the following constants:

- `SSH_VERSION_1: SSH Version 1`
- `SSH_VERSION_2: SSH Version 2`
- `SSH_VERSION_OTHER: Any other version of SSH`

SFtpAuthMethod

This enumeration represents the authentication method to be used with the SFTP server. It defines the following constants:

- `PASSWORD_BASED_AUTHENTICATION`: This constant defines the password-based authentication. The user must provide the username/password pair to use this authentication method.

- `HOST_BASED_AUTHENTICATION`: This constant defines the host-based authentication. The user must provide the private key of the client computer, that is trying to connect to the SFTP server.
- `PUBLIC_KEY_BASED_AUTHENTICATION`: This constant defines the public key based authentication. The user must provide the private key of the user who is trying to connect to the SFTP server.
- `OTHER_AUTHENTICATION_METHOD`: This constant defines any other authentication method, which is not supported by WLI 10.2.

SFtpClientException

This class represents the SFTP client exception, a generic exception category that must be thrown from the SPI implementation. When implementing SPI, you can throw any exception by wrapping it `SFtpClientException` and throwing it to the upper level. The interface of this exception is as follows:

- `public SFtpClientException()`: Default constructor
- `public SFtpClientException(String message)`: Constructs the SFTP client exception with the specified detail message
- `public SFtpClientException(String message, Throwable cause)`: Constructs the SFTP client exception with the specified detail message and the original exception
- `public SFtpClientException(Throwable cause)`: Constructs the SFTP client exception given the original exception

SFtpFile

This enumeration represents the attributes of the file on the SFTP server. Table 6-4, lists the interfaces in this enumeration.

Table 6-4 SFtp File Interface

Interface	Description
`public SFtpFile()`	Default constructor.
`public SFtpFile(String absolutePath)`	Constructs the `SFtpFile` object based on the path of the file. If the absolute path contains the file name, the file name is retrieved.

Table 6-4 SFtp File Interface

Interface	Description
`public void setAbsolutePath(String absolutePath)`	Sets the absolute path for the file represented by this `SFtpFile` object. This method retrieves the file name from the given absolute path if the file name is present.
`public String getAbsolutePath()`	Returns the absolute path of the file represented by this object.
`public String getFilename()`	Returns the name of the file represented by this object.
`public void setFileName(String filename)`	Sets the name of the file represented by this object.
`public void setParent(String parent)`	Sets the directory in which the file resides on the remote server.
`public String getParent()`	Returns the directory in which the file resides on the remote server.
`public void setSize(long size)`	Sets the size of the file represented by this object.
`public long size()`	Returns the size of the file represented by this object.
`public void setModifiedTime(long time)`	Sets the modified time of the file represented by this object.
`public long getModifiedTime()`	Returns the modified time of the file represented by this object.
`public void setFile(boolean isFile)`	Sets a flag to determine whether the file is a disk file.

Table 6-4 SFtp File Interface

Interface	Description
`public boolean isFile()`	Returns a value to indicate whether the file is a disk file.
`public boolean isDirectory()`	Returns a value to indicate whether the file represented by this object is a directory.
`public void setDirectory( boolean isDirectory)`	Sets a flag to determine whether the file is a directory.
`public boolean canRead()`	Returns a value to indicate whether the file represented by this object can be read.
`public void setRead(boole an canRead)`	Sets a flag to determine whether the file represented by this object can be read.
`public boolean canWrite()`	Returns a value to indicate whether the file represented by this object can be written.
`public void setWrite(bool ean canWrite)`	Sets a flag to determine whether the file represented by this object can be written.
`public boolean isLink()`	Returns a value to indicate whether the file represented by this object is a link.
`public void setLink(boole an isLink)`	Sets a flag to determine whether the file represented by this object is a link.
`public boolean isSocket()`	Returns a value to indicate whether the file represented by this object is a socket.

Table 6-4 SFtp File Interface

Interface	Description
`public void setSocket(boolean isSocket)`	Sets a flag to determine whether the file represented by this object is a socket.
`public boolean isOpen()`	Returns a value to indicate whether the file represented by this object is opened.
`public void setOpen(boolean isOpen)`	Sets a flag to determine whether the file represented by this object is opened.
`public void setStageFilepath(String filepath)`	Sets the path whether the file is staged: that is, the path where the file is copied to the local file system.
`public String getStageFilepath()`	Returns the staging path for the file. This path is the location where the file is copied locally.
`public boolean equals(Object object)`	Checks whether two SFTP files are equal.

SFtpClient

This enumeration represents the SFTP client interface that users must implement to plug in any third-party SFTP client implementation to file event generator and file control. This interface provides methods for connecting and authenticating with the SFTP server, retrieving files, listing

files, renaming or deleting files, and transferring files to the SFTP server. The interface is described in Table 6-5:

Table 6-5 SFtp Client Interface

Interface	Description
`public void setAuthenticationMethod(SFtpAuthMethod authMethod)`	Sets the authentication method the client uses to connect to the SFTP server. For the list of authentication methods supported by the SPI, see SFtpAuthMethod.
`public SFtpAuthMethod getAuthenticationMethod()`	Returns the authentication method used by the client to connect to the SFTP server.
`public void setSshVersion(SFtpVersion sshVersion)`	Sets the SSH version that client uses to connect to the SFTP server. See SFtpVersion, for the list of SSH versions supported by the SPI.
`public SFtpVersion getSshVersion()`	Returns the SSH version the client uses to connect to the SFTP server.
`public void setAcceptUnknownHostKeys(boolean accept)`	Sets whether public keys from unknown SFTP servers are accepted. You can use this method to override the default behavior of accepting the public keys from unknown SFTP servers and updating known host files.
`public boolean acceptUnknownHostKeys()`	Accepts public keys from unknown SFTP servers and to update the known host's file.
`public void connect() throws SFtpClientException`	Connects to the SFTP server, based on the authentication method configured. If any exception is thrown by the actual SFTP client API (for example J2SSH), that exception is wrapped in `SFtpClientException` and thrown to the caller.

Table 6-5 SFtp Client Interface

Interface	Description
`public void setUserName(String userName)`	Sets the username for authenticating with the SFTP server. The username is required for `PASSWORD_BASED_AUTHENTICATION`, and it is also required for `HOST_BASED_AUTHENTICATION` and `PUBLIC_KEY_BASED_AUTHENTICATION`.
`public String getUserName()`	Returns the user name the client uses for authenticating with the SFTP server.
`public String getHostname()`	Returns the host name of the SFTP server to which the client is trying to connect.
`public int getPort()`	Returns the port number on which the SFTP daemon is running.
`public void setPassword(char[] pwd)`	Sets the password used for authentication when the authentication method is set to `PASSWORD_BASED_AUTHENTICATION`.
`public void setPassPhrase(char[] phrase)`	Sets the pass phrase for the private key file if the private key is protected by a password. This method is used only when the authentication method is either `HOST_BASED_AUTHENTICATION` or `PUBLIC_KEY_BASED_AUTHENTICATION`.
`public void setPrivateKeyFile(String path)`	Sets the path of the private key file used for authentication when the authentication method is either `HOST_BASED_AUTHENTICATION` or `PUBLIC_KEY_BASED_AUTHENTICATION`.
`public void setCustomProperties(HashMap<String, Object> props)`	Sets any additional properties that users may specify when authenticating with the SFTP server. It provides a way to extend the SPI if the third-party SFTP client requires any additional properties.
`public HashMap<String, Object> getCustomProperties()`	Returns the additional properties that users may use when authenticating with the SFTP server.

Table 6-5 SFtp Client Interface

Interface	Description
`public String pwd() throws SFtpClientException`	Returns the absolute path name of the current remote working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public void cd(String dir) throws SFtpClientException`	Changes the working directory on the SFTP server. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public void mkdir(String dir) throws SFtpClientException`	Creates a directory on the SFTP server. If the directory already exists, this method fails and an exception is thrown. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public void mkdirs(String dir) throws SFtpClientException`	Creates a directory or a set of directories recursively. For example, consider a scenario where the path to the directory is `/tmp/test/test1/test2`. If the `tmp` directory already exist, then this method creates the directories `test`, `test1`, and `test2`. This method does not fail even if a directory already exists. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public void rm(String path) throws SFtpClientException`	Removes a file on the SFTP server. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public void rmdir(String path) throws SFtpClientException`	Removes a directory on the SFTP server. The path to the directory to be removed can be absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.

Table 6-5 SFtp Client Interface

Interface	Description
`public void rename(String oldPath, String newPath) throws SFtpClientException`	Renames a file on the SFTP server. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public List<SFtpFile> list(String path) throws SFtpClientException`	Returns the list of files available on the SFTP server in the directory at the specified path. If the path given represents a file, then this method returns the details about that file. The user is expected to populate the `SFtpFile` objects and return a list of `SFtpFile` objects. The path to the directory/file to be listed can be absolute or relative to the current working directory. If any exception is thrown, the user must wrap that exception in `SFtpClientException`, and throw the exception back to the caller.
`public List<SFtpFile> list() throws SFtpClientException`	Returns the list of files available in the current working directory on the SFTP server. The user must populate the `SFtpFile` objects and return a list of `SFtpFile` objects. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public abstract InputStream get(String s, SFtpFileTransferMode sftpfiletransfermode) throws SFtpClientException;`	Retrieves the file on the SFTP server as a stream. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.

Table 6-5 SFtp Client Interface

Interface	Description
`public abstract void get(String s, String s1, SFtpFileTransferMode sftpfiletransfermode) throws SFtpClientException;`	Retrieves the file on the SFTP server and copy it to the local system on the local file system. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public abstract void get(String s, OutputStream outputstream, SFtpFileTransferMode sftpfiletransfermode) throws SFtpClientException;`	Retrieves a file on the SFTP server and copies the contents of the file to an output stream. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public abstract void put(String s, String s1, SFtpFileTransferMode sftpfiletransfermode) throws SFtpClientException;`	Copies a file on the local file system to the remote SFTP server. The path to the file on the SFTP server be a file or a directory. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.

Table 6-5 SFtp Client Interface

Interface	Description
`public abstract void put(InputStream inputstream, String s, SFtpFileTransferMode sftpfiletransfermode) throws SFtpClientException;`	Copies the file contents from a stream on the remote SFTP server on the SFTP server. The path to the file on the SFTP server can be a file or a directory. The path can be either absolute or relative to the current working directory. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.
`public void disconnect() throws SFtpClientException`	Disconnects from the SFTP server and to clean up any resources held by this client. If any exception is thrown, the exception must be wrapped in `SFtpClientException`, and the exception is thrown back to the caller.

In addition to implementing the above methods, the user must define the following constructors in their implementation of `SFtpClient`:

- `public XXXSFtpClient (String hostname)`: Constructs the SFTP client object, when the host name with a default port number of 22 is provided.
- `Public XXXSFtpClient(String hostname, int portnumber)`: Constructs the SFTP client object, after the host name and port number are provided.

AbstractSFtpClient

This class implements the `SFtpClient` interface and provides a default implementation for all the methods in the `SFtpClient` interface. Although many of these implemented methods do not require any modifications, this class allows you to set properties, such as user name, password, private key, pass-phrase, authentication method, and custom properties.

SFtpClientFactory

This class represents the factory class for `SFtpClient`. You can create instances of `SFtpClient` by using this class. You must implement this class for adding any third-party SFTP client

implementation to file event generator and file control by using the WLI console. For more information, see "Configuring SPI". If the implementation class is not configured, then the default implementation of the `SFtpClientFactory` is used. The default implementation is provided by `com.bea.wli.sftp.j2ssh.impl.J2SSHSFtpClientFactory` and relies on the SFTP client implementation (J2SSH) from SSHTools.

- `public static SFtpClientFactory newInstance(String factoryName)` throws `SFtpClientException`: This static method is used to create a new instance of `SFtpClientFactory`. The new instance is created based on the SFTP client factory class name that is configured through the WebLogic Integration administration console. If this property is not configured, then the default implementation of the SFTP client factory `com.bea.wli.sftp.j2ssh.impl.J2SSHSFtpClientFactory` is returned.
- `public abstract SFtpClient createSFtpClient(String hostName)`: This method is used to create an instance of the `SFtpClient` given the host name and the default port number of 22.
- `public abstract SFtpClient createSFtpClient(String hostName, int portNumber)`: This method is used to create a new instance of the `SFtpClient`, when the host name and port number are provided.

Configuring SPI

To plug in third-party SFTP client implementation using SPI, complete the following steps:

1. Open a web browser, such as Internet Explorer, and enter following URL in the address bar of the browser: `http://adminserver:port/wliconsole`.

   Note: Here `adminserver` is the host name or IP address of the WebLogic Integration Administrative server, and port is the server's listening port. For example, you can type the following to open the WebLogic Integration administration console: http://localhost:7001/wliconsole.

2. Enter the username and password in the Console window.

   The WebLogic Integration Administration Console home page is displayed.

   Note: The WebLogic Integration administration console is password protected. You must create a WLI domain using the Configuration Wizard before you start the server. For more information about creating a domain using Configuration Wizard, see *Domain Configuration Wizard Guide*.

3. Click **System Configuration** on the left pane.
4. Click **Configure** on the left pane under **SFTP**.

The **Current SFTP Settings** page is displayed.

5. Click **Configure**, on the **Current SFTP Settings** page.
6. Enter the following properties:
   - **SFTP Client Factory**: Specify the class name that represents the third-party implementation of the SFTP client factory. If this property is not specified, then the default implementation is used.
   - **Accept Server Keys**: Set whether the public key sent by an unknown SFTP server during handshake is accepted. By default, the implementation accepts the keys from the unknown SFTP server, updates the known hosts file with this entry, and connects to that SFTP server. If this behavior is turned off, then connection to any SFTP server is established only if there is an entry for that server in the known hosts file.

Adding the SPI Implementation to the Server Class Path

To add the SPI implementation to the server class path, complete the following steps:

1. Edit the **`setDomainEnv.cmd\sh`** file located in `\BEA_Home\wli_10.2\DOMAIN_HOME/bin.`
2. Add the jar file containing the SPI implementation to `POST_CLASSPATH`.

SFTP Reference Implementation

An SFTP reference implementation is available with WLI 10.2, which is available in the `\BEA_Home\wli_10.2\samples\sftp_ref_impl` directory, and a default build script is provided for building the source file. This reference implementation is based on the J2SSH (from SSH Tools).

Build an SFTP Reference Implementation

To build an SFTP reference implementation, complete the following steps:

1. Run the **commEnv.cmd** script, which is located in the `\BEA_Home\wlserver_10.0\commom\bin` directory.
2. Change the directory path to `\BEA_Home\wli_10.2\samples\sftp_ref_impl`.
3. Run **`ant clean`** to delete the generated artifacts.

4. To build the jar file, run **`ant deploy`**. The jar file is created in `\BEA_Home\wli_10.2\samples\sftp_ref_impl\build\ar`.
5. To re-deploy the jar file, run **`ant redeploy`**.
6. Plug in the reference implementation, as described in Configuring SPI and Adding the SPI Implementation to the Server Class Path.

Designing an Application to Test the Implementation

Here is an example of how you can design an application to test the SFTP implementation.

1. Create a business process (process.java) with file control.
2. Add `@FileControl.SFtp` to the file control in the **Source** view.
3. Build and deploy the application.

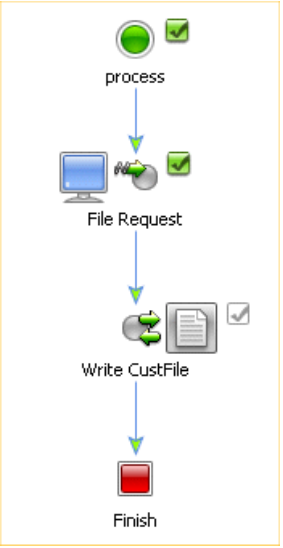
For information about business process, refer *Guide to Building Business Process* and *Tutorial Building your First Business Process*.

Example: File Control

This section provides an example of a File control used in the context of a business process. In this case, the File control instance writes a file to a specified location, triggered by a user request. This example assumes that you have created a new business process containing a client request node.

The business process is shown in the following figure:

Figure 6-4 Example File Control



The business process starts with a client request node, File Request, representing a point in the process at which a client sends a request to a process. In this case, the client invokes the `fileRequest()` method on the process to write a file with information on a new customer to the file system.

Complete the following tasks to design your business process to write the requested file to your file system:

- To Create an Instance of a File Control in Your Project
- To Design a Control Send Node in Your Business Process to Interact With Your File Control

To Create an Instance of a File Control in Your Project

In this scenario, you add one instance of the File control to your business process.

1. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.
2. Choose **File**.

   The **Insert Control: File** dialog box is displayed.

3. In the **Insert Control: File** dialog box do the following:, enter **myFile** as the Field Name for this control and click **Next**.
4. Enter **MyFile** in the **Name** field, and click **Next**.
5. In the **Create Control**, enter the following:
   - **Directory Name**—Enter the location in which you want the File control to write the file. You can use any location on your file system. In this case, the directory name is `C:/temp/customers`.
   - **Type of Data**—Select **XmlObject** from the drop-down list.
6. Click **Finish**.

   An instance of a File control, named **myFile**, is created in your project and displayed in the Data Palette.
7. Select **File > Save** to save your work.

To Design a Control Send Node in Your Business Process to Interact With Your File Control

1. Expand the **myFile** control instance in the **Data Palette**. Then click the following method:

   ```
   FileControlPropertiesDocument write(XmlObject someData)
   ```
2. Drag the method from the **Data Palette** and drop it on your **FileWrite** business process in the **Design view**, placing it immediately after the **File Request** node. The node is named **write** by default.
3. Rename the node, replacing **write** with **Write CustFile**.
4. Double-click the **Write CustFile** node. Its node builder opens on the **General Settings** tab.
5. Confirm that **myFile** is displayed in the **Control** field and that the following method is selected in the **Method** field:

   ```
   FileControlPropertiesDocument write(com.bea.xml.XmlObject someData)
   ```
6. Click **Send Data** to open the second tab in the node builder. The **Method Expects** field is populated with the data type expected by the `write`() method: `XmlObject someData`.
7. In the **Select variables to assign** field, click the arrow to display the list of variables in your project. Then choose **requestCustFile(InputDocument)**. If the variable does not already exist, you can choose **Create new variable...** to create it now.
8. Click **Close**.

9. Double click on the client request node (**File Request**) to open the node builder.
10. Click Receive Data to open the second tab on the node builder. The **Method Expects** field is populated with the data type expected, in this case `InputDocument CustFile`. In the **Select variables to assign** field, click the arrow to display the list of variables in your project. Then choose **requestCustFile(InputDocument)**.
11. Click **Close**.

This step completes the design of your File control node.

At run time, pass a variable of type XmlObject to the Client Request method. The customer document is written to your file system in the location specified.

Http Control

Note: The Http control is available in BEA WorkSpace Studio only for licensed users of WebLogic Integration.

Hyper-Text Transfer Protocol (HTTP) is the globally-accepted method of communicating web pages across the internet. It is a stateless, application-level protocol. The currently defined version of HTTP is 1.1. HTTP protocol is a synchronous protocol, that is, each request message sent from the client to a server is followed by a response message returned from the server to the client.

The Http control's purpose is to provide outgoing HTTP access to BEA WorkSpace Studio clients. The Http control complements the other controls provided in WebLogic Integration and can be used with BEA WorkSpace Studio and business processes to work with HTTP requests and process responses. The Http control is built using the features of the WebLogic Platform control architecture. The Http control source file is a wrapper around the Jakarta Commons *HttpClient* package. The Http control conforms to HTTP/1.1 specific features.

The Http control supports two types of request methods for data transfer, namely *Get* and *Post*. By using the Get mode, you can send your business data along with the URL. By using Post mode, you can send large amount of information like Binary, XML and String documents to the server within the body of the request.

You can specify Http control properties in an annotation, or pass dynamic properties via an XML variable. Inbound HTTP requests can be processed with the HTTP event generator. The HTTP

event generator is a servlet which takes an HTTP request, checks for the content type and then publishes the message to the message broker. For more information on the HTTP event generator, see The HTTP Event Generator.

Using the Http control, you can send an HTTP or HTTPS (Secure HTTP) request to a URL and receive specific HTTP response header and body data, as follows:

- Send Business data using HTTP Get and receive the HTTP response code and the message corresponding to the response code in an XML document.
- Set HTTP header values for the HTTP Post mode.
- Send Binary, XML, and String type data using HTTP Post and receive the HTTP response code and the message corresponding to the response code in an XML document.
- Configure cookies for both the HTTP Get and HTTP Post modes.
- Communicate via a secure HTTP (HTTPS) connection with both client-side and server-side authentication enabled.
- Use a proxy server for sending an HTTP or HTTPS request.
- Receive response headers in an XML document conforming to a pre-defined schema.
- Receive response body data of type Binary, XML or String.
- Receive cookies in an XML document conforming to a pre-defined schema.

Topics Included in This Section

Creating a New Http Control
Describes how to create a new Http control

Using the Http Control in a Business Process
Describes how to create a new Http control and use it in a business process.

Specifying Http Control Properties
Describes Http control properties and the method to specify and edit these properties.

Using HTTP Methods to Set Properties
Describes the various HTTP methods used to specify header properties, cookies, and so on.

Logging Debug Messages and Exceptions
Describes the method used to log debug messages.

Creating a New Http Control

This topic describes how to create a new Http control.

Creating a New Http Control

You can create a new Http control and add it to your business process. To define a new Http control:

1. In the Package Explorer pane, double-click the business process (Process.java file) to which you want to add the Http control. The business process is displayed in the Design view.

2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **Http**.

   The **Insert control: Http** dialog box appears.

4. In the **Insert control: Http** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new HTTP control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.

   – Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

   – Click **Next**.

     The **Create Control** dialog-box appears.

5. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.

     The **Insert control: HTTP** dialog-box appears.
6. In the **Insert control: HTTP** dialog box enter the following details:
   - In **the URL field**, specify the target URL for your Http control, for example, `http://www.bea.com`, `https://www.verisign.com` or `http://localhost:7001/console`.
   - Select the **HTTP mode** that you want to use. You can select either the Get, or the Post mode.
   - From the **Sending Body Data Type** drop-down list, select the data type. You can send your data as an XML object, String, or byte stream. This option is applicable only to the HTTP Post mode.
   - From the **Receiving Body Data Type** drop-down list, select the data type in which you want to receive data. You can choose to receive data in a different format. For example, if you select the Byte data type for sending data and you want to receive the data as an XML object, you can do it.
   - Click **Finish**. Alternatively, you may create a Http control java file manually. For example, you may copy an existing Http control java file and modify the copy.

The Java file for the Http Control

When you create a new Http control, you create a new java control file in your project. The following is an example of a java control file

```
package requestquote;

import org.apache.beehive.controls.api.bean.ControlExtension;

import com.bea.wli.control.httpParameter.ParametersDocument;

import com.bea.wli.control.httpResponse.ResponseDocument;

/*

 * A custom Http control.
```

```
 */
@ControlExtension
@com.bea.control.HttpControl.EndPoint(url = "http://www.bea.com")
public interface HttpControl extends com.bea.control.HttpControl {
       @MethodType(method = METHOD.GET)
       public ResponseDocument get(ParametersDocument parameters, String
charset);


       @MethodType(method = METHOD.GETRESPONSE)
       public byte[] getResponseData();


       static final long serialVersionUID = 1L;


       public HttpControl create();
}
```

The contents of the Http control's java file depend on the selections made in the Insert Http dialog. The given example was generated in response to selection of byte[] from the Body Type drop-down list.

Using the Http Control in a Business Process

The business process starts with a client request node, representing a point in the process at which a client sends a request to a process. In this case, the client invokes the `setProperties` method on the process to specify a dynamic property for your Http control.

Complete the following tasks to design your business process to send and receive data using your Http control, using a dynamic property setting that specifies the target URL to send and receive data.

- Create an instance of the Http control, and call it MyHttpControl. Use the steps provided in Creating a New Http Control.
- Your new Http control will be visible under the Controls tab in the Data Palette. Expand MyHttpControl to see the Http methods that you can use in your business process.

- Design a Control Send Node in your business process and specify a dynamic property to be used during run time.

To Design a Control Send Node in Your Business Process

1. Expand the **MyHttpControl** control instance in the **Data Palette**. Then click the following method:

   ```
   setProperties(HttpControlPropertiesDocument propsDoc)
   ```

2. Drag the method from the Data Palette and drop it on your business process in the Design view, placing it immediately after the **Client Request** node.

3. Double-click the **SetProperties** node. Its node builder opens on the **General Settings** tab.

4. Confirm that **MyHttpControl** is displayed in the **Control** field and that the following method is selected in the **Method** field:

   ```
   setProperties(HttpControlPropertiesDocument propsDoc)
   ```

5. Click **Send Data** to open the second tab in the node builder. The **Control Expects** field is populated with the data type expected by the `setProperties` method: `HttpControlPropertiesDocument`.

6. In the **Select variables to assign** field, choose **Create new variable...** using the name `dynamicprop`. Close the window.

7. Double click on the client request node to open the node builder.

8. Open the General Settings tab of the node builder and create a variable of type `com.bea.wli.control.dynamicProperties.HttpControlPropertiesDocument`.

9. Open the Receive Data tab. The Client Sends field in this tab populated with the variables that have been created in the General Settings tab, in this case, `HttpControlPropertiesDocument x0`. In the Select variables to assign field, click the arrow to display the list of variables in your project and choose `dynamicprop` as the variable to assign.

This step completes the design of your Http control node.

At run time, the dynamic property that you defined will override the static property defined using the Property Editor.

Specifying Http Control Properties

Most aspects of a Http control can be configured from the Properties pane in Design View. You can also specify run-time properties that define the way your Http control is used during run time. For more information on how to use run time, or dynamic properties, see Setting Dynamic Http Control Properties.

You can define the control properties in the Properties pane, or, you can change the properties in the Source view of the Http control's java file. For more information on the java file for the Http control, see The Java file for the Http Control.

When you modify properties for your Http control using the Properties pane, your changes are reflected in the Source view of the control's java file, and vice versa. However, the properties that you specify during run time override the properties set using the Property Editor in the Design view. For more information on setting properties, see Using HTTP Methods to Set Properties.

Using HTTP Methods to Set Properties

You can specify the behavior of an Http control in Design View by setting the control’s properties in the Properties pane. The following attributes specify class- and method-level configuration attributes for the Http control.

This topic defines the various HTTP methods that you can use to specify properties. Each method is described briefly in Http Control MethodsTable 7-1, and detailed in subsequent sections that are referenced to the methods outlined in the table.

You can use the following methods with the Http control:

Table 7-1 Http Control Methods

Purpose of Method	Description	Method
Setting Dynamic Http Control Properties.	This method sets the Http control properties at run time. Dynamic properties always override the static properties set in the Properties pane.	`void setProperties(HttpControlPropertiesDocument propsDoc)`
Setting Connection Time-out.	This method sets the connection time out for an HTTP request. Set this property to define the maximum time you want your Http control to establish a connection. A time-out value of zero (zero is the default value) indicates that the connection time-out has not been used.	`void setConnectionTimeOut(int timeoutInMilliSeconds)`
Setting Connection Retry Count.	This method defines the number of times your Http control will try to establish connection with the target.	`void setConnectionRetrycount(int retryCount)`
Setting Cookie	This method allows you to set cookies for your Http control	`void setCookies(CookiesDocument cookies)`
Configuring Proxy Settings.	This method allows you to specify proxy settings such as String host, initial port, String user name, and String password.	`void setProxyConfig setProxyConfig (String host, int port, String userName, String password)`
Configuring Server-side SSL	This property allows you to configure server-side Secure Socket Layer authentication process.	`void setServerSideSSL(String trustStoreLocation, boolean hostVerificationFlag)`
Configuring Client-side SSL	This property allows you to set client-side authentication.	`void setClientSideSSL(String keyStoreType, String keyStoreLocation, String keyStorePassword, String keyPassword)`

Table 7-1 Http Control Methods (Continued)

Purpose of Method	Description	Method
Sending an HTTP Get Request	This method allows you to send an HTTP request using the HTTP Get mode and receive the HTTP response code from the server.	`ResponseDocument sendDataAsHttpGet(ParameterDocument parameters,String charset)`
Setting Headers for HTTP Post	This method allows you to set the header properties for the HTTP Post mode.	`void setHeadersForHttpPost(HeaderDocument headers)`

Sending Data as HTTP Post	This method allows you to send body data as HTTP Post and receive the response code.
	Depending on the body data type that you select while configuring the Http control, the appropriate method is populated in the java file.
		`ResponseDocument sendDataAsHttpPost(String bodyData)`
		`ResponseDocument sendDataAsHttpPost (XmlObject bodyData)`
		`ResponseDocument sendDataAsHttpPost (byte[] bodyData)`

Recieving HTTP Response Headers	This method allows you to get the headers of an HTTP response.	`HeadersDocument getResponseHeaders()`
Recieving Cookies From the Server	This method allows you to receive cookies from an HTTP response.	`CookiesDocument getCookies()`

Recieving HTTP Body Data	Depending on the body data type that you select while configuring the Http control, the appropriate method is populated in the java file.	`String getResponseBodyAsString()`
		`XmlObject getResponseBodyAsXML()`
		`byte[] getResponseBodyAsBytes()`

Setting Dynamic Http Control Properties

Method: `void setProperties(HttpControlPropertiesDocument propsDoc)`

To use dynamic properties, pass an XML variable that conforms to the Http control's dynamic-property schema to the Http control's setProperties() method.

Example of an XML Variable to Set Dynamic Properties

```
<?xml version="1.0" encoding="UTF-8"?>
<xyz:HttpControlProperties
xmlns:xyz="http://www.bea.com/wli/control/dynamicProperties">
<xyz:URLName>http://localhost:7001/console</xyz:URLName>
</xyz:HttpControlProperties>
```

Schema for Http Control Properties

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://www.bea.com/wli/control/dynamicProperties"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.bea.com/wli/control/dynamicProperties"
elementFormDefault="qualified">
<xs:element name="HttpControlProperties">
<xs:complexType>
<xs:sequence>
<xs:element name="URLName" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Setting Connection Time-out

Method: `setConnectionTimeOut (int timeoutInMilliSeconds)`

This method sets the connection time out for an HTTP request. The connection time-out is maximum time that a control is allowed to establish a connection - the connection fails after this time elapses. The parameter time-out is set in milliseconds. A time-out value of zero (zero is the default value) indicates that the connection time-out has not been used.

Setting Connection Retry Count

Method: `setConnectionRetrycount (int retryCount)`

This method sets the retry count, that is, the number of times your application will retry for the HTTP request. If this value is not specified, then the application will try to connect only once. If a connection is not established in the first try, the second attempt is likely to succeed. It is

recommended that you set this property so that your HTTP requests go through in the second attempt, if not the first one.

Configuring Server-side SSL

Method: `setServerSideSSL (String trustStoreLocation, boolean hostVerificationFlag)`

The Http control provides complete support for HTTP over Secure Sockets Layer (SSL) and Transport Layer Security (TLS), by leveraging the Java Secure Socket Extension (JSSE). JSSE is integrated into JDK1.4, which is shipped along with WebLogic Integration Platform.

When you run this method, the configuration for server-side authentication is set. By default, JSSE uses (`jdk142_04\jre\lib\security\cacerts`) as its Trust Store location, which includes some well-known certificate authorities such as Verisign and CyberTrust. Therefore, you do not need to specify any Trust Store locations for the certificates, which are issued by the certification authority.

Additionally, you can provide a host-name verification flag that ensures that the SSL session's server host-name matches with the host name returned in the server certificates Common Name field of the SubjectDN entry. By default this entry is set to `False`.

For example, if you specify `https://www.verisign.com/` as the URL for authentication, you do not have to specify the Trust Store location, as Verisign is a trusted authority in certificates of JSSE.

To accept self-signed or SSL certificates that are not trusted, you need to import the server certificates into its Trust Store Location. For more information on JSSE, see the Java Secure Socket Extension (JSSE) Reference Guide at the following location:

`http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html`

The following example shows how to create a store, import a server certificate, and to specify the parameters for this method:

1. Run the following Keytool command to create a new Keystore.

   `keytool -genkey -alias` *`aliasname`* `-keyalg rsa -keystore` *`keystore name`*

   The following is an example of the command, including user-input values:

   ```
   keytool -genkey -alias teststore -keyalg rsa -keystore
   c:\teststore.jks
   ```

   For more information, see *Creating a Keystore to Use with JSSE*, at the following location:

http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

2. Launch an HTTPS site to copy the certificate. For example, you can launch the WebLogic Server Console of the localhost or any other machine using the `https://host:port/console` format. When you are prompted for the server certificate, click the **View Certificate** button, navigate to the **Details** tab, and then click **Copy to File**.
3. Import the certificate that you copied to the Keystore that you created in Step 1, using the following command:

   keytool -import -alias *aliascertname* -file *certificatename* -keystore *keystore_name*

   For example:

   keytool -import -alias testcer -file c:\test.cer -keystore c:\teststore.jks
4. In the `setServerSideSSL` method, specify the Trust Store location as `C:\teststore.jks` and the URL to which you send a request as `https://host:port/console`. To verify the host name, set the `host-name` verification flag as `true`.

Configuring Client-side SSL

Method: `setClientSideSSL (String keyStoreType, String keyStoreLocation, String keyStorePassword, String keyPassword)`

This method sets the configuration for client-side authentication. You should use this method when both server-side and client-side authentication are required. Before configuring this method, you must configure Configuring Server-side SSL.

In this method, both the `keyStoreType` and `keyPassword` fields are optional. If you do not specify the keyStoreType, the method uses the default Keystore type (which is specified in the `java.security` file).

For some Keystores, the Keystore password differs from the key password. In such cases, you must specify both the Keystore password and key password.

If you want both server-side and client-side configuration, the server certificate should be in the Client Trust Store. Similarly, the client certificate should be in the Server Trust Store and the client should specify the Keystore location and passwords appropriately.

Configuring Proxy Settings

Method: `setProxyConfig (String host, int port, String userName, String password)`

This method configures parameters for a proxy server. To send an HTTP request using a proxy server, you must properly configure the host, port, user name, and password.

Note: The Http control supports the Basic Scheme protocol. It does not support NTLM protocol. You need to configure your proxy settings accordingly.

Setting Cookie

Method: `setCookies(CookiesDocument cookies)`

The Http control allows you to manually set the cookies sent to the server. To send cookies to the server with an HTTP request, you have to pass a XML variable that conforms to the Http control's cookies document schema.

Example: XML Variable Used to Set Cookies

```
<?xml version="1.0" encoding="UTF-8"?>
<Cookies xmlns="http://www.bea.com/wli/control/HttpCookies">
<Cookie>
<Name>CookieName1</Name>
<Value>CookieValue1</Value>
</Cookie>
<Cookie>
<Name>CookieName2</Name>
<Value>CookieValue2</Value>
</Cookie>
</Cookies>
```

Schema for Setting Cookie

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.bea.com/wli/control/HttpCookies"
elementFormDefault="qualified"
targetNamespace="http://www.bea.com/wli/control/HttpCookies">
<xs:element name="Cookies">
<xs:complexType>
<xs:sequence>
<xs:element name="Cookie" minOccurs="0"
maxOccurs="unbounded">
<xs:complexType>
```

```
<xs:sequence>
<xs:element name="Name" type="xs:string"
minOccurs="0"/>
<xs:element name="Value" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Setting Headers for HTTP Post

Method: `setHeadersForHttpPost (HeadersDocument headers)`

This method sets the request header for an HTTP Post. To set the request header, you have to pass an XML variable that conforms to the Http control's headers document schema. You can overwrite the default header's values by specifying them in the following manner:

`User-agent`, `Content-Type`, and so on.

Example: XML Variable Used to Set the Headers

```
<?xml version="1.0" encoding="UTF-8"?>
<xyz:Headers xmlns:xyz="http://www.bea.com/wli/control/HttpHeaders">
<xyz:Header>
<xyz:name>Content-Type</xyz:name>
<xyz:value>text/*</xyz:value>
</xyz:Header>
<xyz:Header>
<xyz:name>header</xyz:name>
<xyz:value>h1</xyz:value>
</xyz:Header>
</xyz:Headers>
```

Schema for Setting HTTP Post Headers

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
xmlns="http://www.bea.com/wli/control/HttpHeaders"
elementFormDefault="qualified"
targetNamespace="http://www.bea.com/wli/control/HttpHeaders">
<xs:element name="Headers">
<xs:complexType>
<xs:sequence>
<xs:element name="Header" minOccurs="0" maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string" minOccurs="0"/>
<xs:element name="value" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Sending an HTTP Get Request

Method: `ResponseDocument sendDataAsHttpGet(ParametersDocument parameters,String charset)`

Use this method when you want to send an HTTP Get request. The Get request is mostly used for accessing static resources such as HTML documents from a Web Server and also can be used to retrieve dynamic information by using additional parameters in the request URL.

With Get requests, the request parameters are transmitted as a query string appended to the request URL. To include multi-byte character parameters in the URL, the Http control encodes the parameters to the characters as defined by the `charset` field of this method. If you do not specify any character set, then the Http control will send the parameter data URL encoded in "UTF-8". To send the parameters with a URL, you must pass the parameters in an XML variable that conforms to the Http control's parameter document schema.

Example: XML Variable Used to Set Parameters in HTTP Get

```
<?xml version="1.0" encoding="UTF-8"?>
<xyz:Parameters xmlns:xyz="http://www.bea.com/wli/control/HttpParameter">
<xyz:Parameter>
```

```
<xyz:Name>Customer Id</xyz:Name>
<xyz:Value>1000</xyz:Value>
</xyz:Parameter>
<xyz:Parameter>
<xyz:Name>Customer Name</xyz:Name>
<xyz:Value>Robert</xyz:Value>
</xyz:Parameter>
</xyz:Parameters>
```

Schema for Sending Parameters for HTTP Get

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.bea.com/wli/control/HttpParameter"
elementFormDefault="qualified"
targetNamespace="http://www.bea.com/wli/control/HttpParameter">
<xs:element name="Parameters">
<xs:complexType>
<xs:sequence>
<xs:element name="Parameter" minOccurs="0"
maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="Name" type="xs:string"
minOccurs="0"/>
<xs:element name="Value" type="xs:string"
minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Sending Data as HTTP Post

Method: `ResponseDocument sendDataAsHttpPost (String/XmlObject/byte[ ] bodyData)`

Use the HTTP Post method to post data to a server. The Http control allows you to post data of three different data types: String, XmlObject, and byte.

The HTTP Post method returns the HTTP response, that is, the HTTP response code and corresponding message in a ResponseDocument. The schema of the response document is the same as described in Schema for Sending Parameters for HTTP Get.

HTTP Post requests are meant to transmit information that is request-dependent, and are used when you need to send large amounts of information to the server. The Http control allows you to post data of three different data types: String, XmlObject, and Byte stream.

In the HTTP protocol, servers and clients use MIME (Multipurpose Internet Mail Extensions) headers to indicate the type of content present in requests and responses. Http control also uses the MIME header(Content-Type), while transmitting data in body of the requests, to describe the type of data being sent. So while posting String or XmlObject data type, you should set the `Content-Type` header appropriately by using the Http control's `setHeadersForHttpPost()` method. The `Content-Type` header contains a `charset` attribute that indicates the character set of the message body.

If you do not set any `charset` attribute, then the Http control uses the default HTTP encoding (ISO-8859-1) to encode the message.

The following examples provide more information on how to send data using the HTTP Post mode:

Example 1 - Request body with String data-type

To post a string message of encoding `Shift-JIS`, you should set the `charset` attribute in the Content-Type request header, by calling the Http control's `setHeadersForHttpPost` method, as follows:

```
Content-type="text/*; charset=Shift-JIS"
```

Example 2 - Request body with XmlObject data type

While sending request messages of XML data type, you have to set the `charset` attribute in `Content-Type` header appropriately.

If you do not specify the character encoding in the `Content-Type` header, then the Http control uses the default encoding as specified in `rfc3023`.

For example, to post an XML document of encoding `EUC-JP`, you need to set the request type header as follows:

```
Content-Type="text/xml; charset=EUC-JP"
```

If you do not specify any `charset` attribute in the request header, the Http control uses `us-ascii` as default encoding to encode the message.

Note: To avoid garbling of body data while posting String or Xml data types, you should always specify the `charset` attribute in the `Content-Type` header.

The HTTP Post method returns the HTTP response, that is, the HTTP response code and corresponding message in a ResponseDocument. The schema of the response document is the same as described in Schema for Sending Parameters for HTTP Get.

Recieving HTTP Response Headers

Method: `HeadersDocument getResponseHeaders`

Use this method to receive the HTTP response headers. The response headers are returned in an XML variable of a pre-defined schema.

The schema for the response headers is same as request headers schema described in Setting Headers for HTTP Post.

Recieving Cookies From the Server

Method: `CookiesDocument getCookies`

Use this method to receive the cookies from the server. The cookies are returned in an XML document of a pre-defined schema.

The schema for the response cookies is same as the request cookies schema described in Schema for Setting Cookie.

Recieving HTTP Body Data

Method: `String getResponseBodyAsString / XmlObject getResponseBodyAsXml / byte[ ] getResponseBodyAsBytes`

In HTTP, in response to a HTTP request, the server sends the body content that corresponds to the resource specified in the request. If you want to receive the response body data, then you should use this method.

The Http control can return the response data in three different data types: String, XmlObject, and Byte[]. You should set the response data type appropriately, depending upon the response data that you expect from the server. If the response body is not available or cannot be read, the control returns a null value.

Note: While parsing the response body of data type String or XmlObject, the Http control uses the character encoding specified in the `Content-Type` response header. If character encoding is not specified in the `Content-Type` header, the Http control uses the default HTTP content encoding `ISO-8859-1` for String and `US-ASCII` encoding for `XmlObject`.

To avoid garbling of data, you should always set the `charset` attribute in the `Content-Type` response header.

Logging Debug Messages and Exceptions

During run time, the Http control checks for different parameters, null value, and method return types. If validation fails at any point, a control exception is thrown to the Business Process Management (BPM). You can log debug messages, review them, and resolve exceptions if required.

To log debug messages, edit the apache log properties file. You can find the apache log properties file, `apacheLog4jCfg.xml`, in the `BEA_HOME\wlserver_10.0\samples\domains\integration` folder.

To log all the debug statements for `HttpControlImpl` and `HttpResource` class files, add the following lines to the `appacheLog4jCfg.xml` file:

```
<category name="com.bea.control.HttpControl">
<!-- NOTE: DO NOT CHANGE THIS PRIORITY LEVEL W/O WLI DEV APPROVAL -->
<!-- Debug-level log information is frequently the only tool available to
diagnose failures! -->
<priority value="debug"/>
<appender-ref ref="SYSLOGFILE"/>
<appender-ref ref="SYSERRORLOGFILE"/>
</category>
<category name="com.bea.control.HttpResource">
<!-- NOTE: DO NOT CHANGE THIS PRIORITY LEVEL W/O WLI DEV APPROVAL -->
<!-- Debug-level log information is frequently the only tool available to
diagnose failures! -->
<priority value="debug"/>
<appender-ref ref="SYSLOGFILE"/>
<appender-ref ref="SYSERRORLOGFILE" />
</category>
```

All debug statements are logged into `workshop_debug.log` file in the corresponding domain where the application runs.

Http Control Caveats

The following are the known limitations of the Http control:

- The Http control doesn't expose any specific method for posting a multi-part document. However, you can write the code to construct a multi-part message and then convert it into byte stream and use the `sendDataAsHttpPost(byte[] bodyData)` method to post data.
- The Http control does not support Microsoft Proxy Server. This is because Microsoft Proxy Server uses NT Lan Manager (NTLM) authentication, which is proprietary to Microsoft.

The HTTP Event Generator

The HTTP event generator is a servlet that takes an HTTP request, checks for the content type in the HTTP request, and then appropriately publishes the message to the Message Broker.

The HTTP event generator supports two message data types (XML and binary). The data-type is determined from the Content-Type header of the HTTP request, property name, and matching values, as well as other handling criteria are specified in the channel rules of the event generator.

You need to configure event generator channels for different data types, using a Message Broker channel name, which instructs that any HTTP request coming to that servlet will publish the message to that channel.

To learn more, see Event Generators in *Using the WebLogic Integration Administration Console*.

Message Broker Controls

Note: The Message Broker controls are available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

Messaging systems are often used in enterprise applications to communicate with legacy systems, or for communication between software components. A client of a messaging system can send messages to, and receive messages from, any other client.

The Message Broker resource provides a publish and subscribe message-based communication model for WebLogic Integration business processes, and includes a powerful message filtering capability.

The Message Broker provides typed channels, to which messages can be published, and to which services can subscribe to receive messages. You can design a business process to subscribe to specific channels, using XML Beans for type-safe methods.

Subscribers to Message Broker channels can filter messages on the channels using XQuery filters. WebLogic Integration supports a powerful mapping tool that allows you to create XQuery filters for channels. Business processes can filter documents on channels, based on document type or document content. For example, you can design a filter that filters on stock symbol documents, or one that filters on a specific purchase order number.

In addition to business processes that can publish messages to Message Broker channels, WebLogic Integration supports event generators, which can publish external events to message

broker channels. WebLogic Integration provides native event generators, including Email, File, HTTP, JMS, MQ, and Timer event generators. These event generators allow you to start or continue a business process based on events, such as the receipt of email or a new file appearing in a directory. To learn about creating and managing event generators using the WebLogic Integration Administration Console, see Event Generators in *Using the WebLogic Integration Administration Console* at the following URL:
`http://edocs.bea.com/wli/docs102/adminhelp/evntgen.html`

To learn more about channels, see:

- Messge Broker in *Using the WebLogic Integration Administration Console*.
- "Note About Static and Dynamic Subscriptions" in @com.bea.wli.control.broker.MessageBroker.StaticSubscription.

You can customize Message Broker controls in several ways. You may modify the properties of the control. These modifications is described in more detail in the sections that follow.

Topics Included in This Section

Message Broker Publish Control

Two Message Broker controls are available from your business processes: Publish and Subscription. Your business process uses a Publish control to publish messages to Message Broker channels. You bind the Message Broker channel to the Publish control when you declare the control, but it can be overridden dynamically. You can add additional methods to your extension (subclass) of the Message Broker Publish control.

For information on how to add control instances to business processes, see Using Controls in Business Processes.

The following topics provide information about creating and using Message Broker Publish controls:

- Example Code for MB Publish Control

To Create an Instance of a Message Broker Publish Control

1. In the **Package Explorer** pane, double-click the business process to which you want to add the MB Publish control. The business process is displayed in the **Design** view.

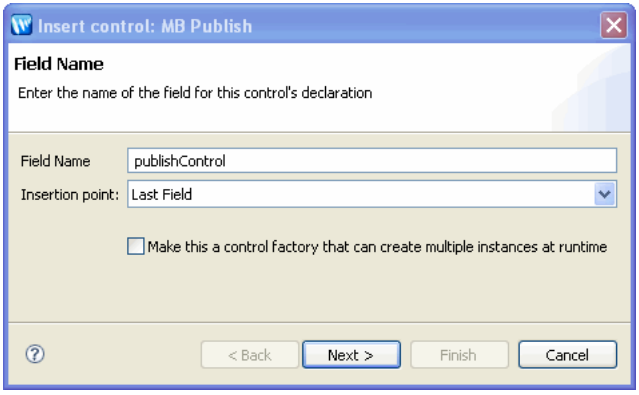
2. Click on the Data Palette and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

Note: If the Data Palette view is not visible in BEA Workshop for WebLogic Platform, click **Window > Show View > Data Palette** from the menu bar.

3. Select **MB Publish**.

   The **Insert Control: MB Publish** dialog box appears (see Figure 8-1).

Figure 8-1 Insert Control: MB Publish



4. In the **Insert Control: MB Publish** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new MB Publish control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

The **Create Control** wizard appears.

5. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.
6. In the **Insert control- MB Publish** dialog-box, enter one of the following details:
   - **Channel name**—Select a channel to which you want your business process to publish.

   Note: If no options are available in the **channel-name** field, you must create a channel file, which defines the channels to which your business process can publish and subscribe.

   - **Message type**—This read-only field displays the type of data published to the specified channel.
   - **Metadata type**—This read-only field displays the metadata type value if `qualifiedMetadataType` is set in the channel definition.
   - Click **Finish.**

   An instance of a MB Publish control is created in your project and displayed in the Data Palette.

Java File for Your MB Publish Control

When you create a new MB Publish control, you create a new Control java file in your project. The following example Control java file is automatically created by the control wizard:

```
package requestquote;

import org.apache.beehive.controls.api.bean.ControlExtension;

/*
 * A custom Publish control.
 */

@ControlExtension

@com.bea.control.PublishControl.ClassPublish(channelName =
"/deadletter/rawData")

public interface MBPublish extends com.bea.control.PublishControl {
```

```
       @com.bea.control.PublishControl.MethodPublish(body = "{value}")
       void publish(com.bea.data.RawData value);
       @com.bea.control.PublishControl.MethodPublish(metadata =
"{metadata}", body = "{value}")
       void publishWithMetadata(org.apache.xmlbeans.XmlObject metadata,
                    com.bea.data.RawData value);

       static final long serialVersionUID = 1L;

       public MBPublish create();
}
```

Using Methods of the MB Publish Interface

This section describes the MB Publish control interface. Use the methods from within your application to publish to Message Broker channels.

MB Publish Control Interface

```
package com.bea.control;

import
com.bea.wli.control.dynamicProperties.PublishControlPropertiesDocument;
import org.w3c.dom.Element;
import weblogic.jws.control.Control;

/**
 * Message Broker Publish control base interface
 */
public interface PublishControl extends Control {
    /**
     * Temporarily sets the message headers to use in the next publish
operation
```

```
     * @param headers headers to set
     */
    void setOutputHeaders(Element[] headers);
    /**
     * Sets the dynamic properties for the control
     * @param props the dynamic properties for the control
     */
     void setProperties(PublishControlPropertiesDocument props);
    /**
     * Sets the dynamic properties for the control
     * @return the current properties for the control
     */
     PublishControlPropertiesDocument getProperties();
}
```

The `PublishControlPropertiesDocument` XML Bean is defined in `DynamicProperties.xsd` which is located in the Schemas folder of each process application.

To learn more about the methods available on the MB Publish control, see the Interface PublishControl Javadoc.

Method Attributes

The following method attributes determine the behavior of the MB Publish control.

Class attributes include:

`channel-name`
: The name of the Message Broker channel to which the MB Publish control publishes messages.

`message-metadata`
: By default, this XML header is included in messages published with this control. Valid values include a string containing XML.

Method attributes include:

`message-metadata`
: XML header to include in messages published with the control method to which it is associated. Valid values include a string containing XML, or a method parameter in curly braces. For example: {*`parameter1`*}.

message-body

Valid values include a string containing text that is used as the message body in the published message, or a method parameter in curly braces. For example: {*parameter2*}.

Example Code for MB Publish Control

The Publish control allows you to override class-level annotations with dynamic properties. To do so, use an XML variable that conforms to the control's dynamic property schema.

The following is an example of an XML variable you can use to specify the dynamic properties:

```
<PublishControlProperties>
    <channel-name>potopic</channel-name>
    <message-metadata>
        <custom-header>ACME Corp</custom-header>
    <message-metadata>
</PublishControlProperties>
```

The XML Schema for the MB Publish control dynamic properties is shown in the following listing. You can obtain this schema by adding the WLI Schemas project template to you application. You can get and set these properties using the `getProperties` and `setProperties` methods.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.bea.com/wli/control/dynamicProperties"
xmlns="http://www.bea.com/wli/control/dynamicProperties"
elementFormDefault="qualified">
    <xs:element name="PublishControlProperties">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="channel-name" type="xs:string"
                            minOccurs="0" maxOccurs="1" />
                <xs:element name="message-metadata" type="header"
                            minOccurs="0" maxOccurs="1" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

<!-- The following complex-type represents any arbitrary sequence of XML content
-->

    <xs:complexType name="header">
        <xs:sequence>
            <xs:any namespace="##other" minOccurs="0"
                    maxOccurs="unbounded" processContents="lax" />
        </xs:sequence>
```

```
    </xs:complexType>
</xs:schema>
```

Example Code

MB Publish controls must be extended. The following is an example of how to code a MB Publish control in your business process.

```
/*
 * @com.bea.control.PublishControl.ClassPublish
channel-name="/controls/potopic"
 */
public interface MBPublish extends com.bea.control.PublishControl{
    /**
     * @com.bea.control.PublishControl.MethodPublish
     *      message-metadata="<custom-header>ACME Corp</custom-header>"
     *      message-body="{myMsgToSend}"
     */
    void publishPO(XmlObject myMsgToSend);
}


/*
 * org.apache.beehive.controls.api.bean.Control
 */
private MyPublishControl pubCtrl;

// publish a message
void sendIt(XmlObject myMsgToSend) {
    pubCtrl.publishPO(myMsgToSend);
}
```

Message Broker Subscription Control

Two Message Broker controls are available from your business processes: Publish and Subscription. Your business process uses a Subscription control to dynamically subscribe to channels and receive messages. You bind the channel and optionally, an XQuery expression for filtering messages, when you create an instance of the control for your business process. The bindings cannot be overridden dynamically.

The Subscription control interface includes methods that allow your business process to subscribe to and unsubscribe from the bound Message Broker channel.

Subscribe operations are part of the larger XA transaction, as with other business process operations. This allows subscribe operations to be rolled back if the business process operation fails. Because a subscription is in a transaction, you have to commit the transaction to make it durable. If you're doing non-transactional work, that is, if a subscribe operation must be committed before performing an action that might trigger a return message, use `<transaction/>` blocks in the flow to commit the current business process state, including the subscription.

For information on how to add control instances to business processes, see Using Controls in Business Processes.

The following topics provide information about creating and using Message Broker Subscription controls:

- To Create an Instance of a Message Broker Subscription Control
- Using Methods of the MB Subscription Interface
- Example Code for MB Subscription Control
- Note About Static and Dynamic Subscriptions to Message Broker Channels

To Create an Instance of a Message Broker Subscription Control

1. In the **Package Explorer** pane, double-click the business process to which you want to add the MB Subscription control. The business process is displayed in the **Design** view.
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **MB Subscription**.

   The **Insert Control:MB Subscription** dialog box appears.

Figure 8-2 Insert Control: MB Subscription

4. In the **Insert Control:MB Subscription** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new MB Subscription control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

     The **Create Control** dialog-box appears.
5. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.
6. In the **Insert Control:MB Subscription** dialog box enter the following details:
   - **Channel-name**—Select a channel to which you want your business process to subscribe.

   Note: If no options are available in the **channel-name** field, you must create a channel file, which defines the channels to which your business process can publish and subscribe.

– **Message type**—This read-only field displays the type of data received from the specified channel: **String**, **XmlObject**, **RawData**.

– **Metadata type**—This read-only field displays the metadata type value if `qualifiedMetadataType` is set in the channel definition.

– Select the **This subscription will be filtered** check box if you want to subscribe using filter values.

Note: If you choose the option "this subscription" will be filtered, it's mandatory to provide the filter value in SubscriptionControl annotation

– Click **Finish**.

An instance of a MB Subscription control is created in your project and displayed in the **Controls** tab.

The control declaration is written to your Process.java file.

```
/**
  * org.apache.beehive.controls.api.bean.Control
  */
  private processes.mbSubscribe mbSubscribe;
```

Note: If a JPD has any in-line XQueries, while upgrading the JPD, either the XQueries are upgraded, or the JPD is marked with @XQuery.Version=2002. If the JPD does not have any XQueries, the version is not set to xquery=2004. If this JPD refers to a subscription control that has XQuery set, the upgraded JPD version is xquery=2004, and the upgraded subscription control version is xquery=2002. The XQuery prolog of the JPD is appended to any XQueries in subscription controls used in the JPD while generating the compiled artifacts (`wli-subscriptions.xml`). The version incompatibility makes the generated XQuery invalid, as the XQuery is XQ2002 and its prolog is XQ2004.

The only solution is to ascertain that all xqueries in an upgraded application are either version XQ2002 or XQ2004. The XQuery functions and namespaces declared in an XQuery prolog in the JPD are not accessible to the XQueries in subscription control. This is an unsupported case when mixing two different XQuery versions in the same application.

Java File for Your MB Subscription Control

When you create a new MB Subscription control, you create a new Control java file in your project. The following example Control java file is automatically created by the control wizard:

```
import com.bea.control.SubscriptionControl;
import com.bea.data.RawData;
import com.bea.xml.XmlObject;

package requestquote;

import com.bea.control.SubscriptionControl;

import org.apache.beehive.controls.api.bean.ControlExtension;

import org.apache.beehive.controls.api.events.EventSet;

/*

 * A custom Subscribe control.

 */

@ControlExtension

@SubscriptionControl.ClassSubscription(channelName = "/deadletter/string")

public interface MBSubscription extends com.bea.control.SubscriptionControl
{

       @EventSet

       interface Callback {

              @SubscriptionControl.SubscriptionCallback(body =
"{message}")

              void onMessage(String message);

       }


       static final long serialVersionUID = 1L;


       public MBSubscription create();

}
```

You must select the **This subscription will be filtered** check box to ensure that the `subscribeWithFilterValue()` method in included in the Control java file. The `onMethod` method on the Calback interface uses the message type defined in the channel file.

Using Methods of the MB Subscription Interface

This section describes the MB Subscription control interface.The methods you can use to subscribe to Message Broker channels are available from within your application.

Class Interface

```
package com.bea.control;

import weblogic.jws.control.Control;

/**
 * Message Broker Subscription control base interface
 */

public interface SubscriptionControl extends Control
{

/**
     * Subscribes the control to the message broker. If the subscription
     * uses a filter expression, then the default filter value will be
     * used. If no default filter value is defined in the annotations,
     * then a <tt>null</tt> filter value will be used, meaning that any
     * filter result will trigger a callback.
     */

    void subscribe();

    /**
     * Unsubscribes the control from the message broker, stopping
     * further events (messages) from being delivered to the control.
     */

    void unsubscribe();


    interface Callback {
        /**

       * Internal callback method used to implement user-defined callbacks.
         * JPDs cannot and should not attempt to implement this callback
method.
         *
         * @param msg  the message that triggered the subscription
```

```
         * @throws Exception
         *
        void _internalCallback(Object msg) throws Exception;
         */
    }
}
```

Note: If the subscription uses a filter, you must define custom subscription methods to specify the filter value to be matched at run time.

The Subscription control does not define callback methods for you. You must define a custom callback to specify how the business process expects to receive the event messages. (Event messages can be XML, raw data, or string.)

To learn more about the methods available on the MB Subscription control, see the Interface SubscriptionControl Javadoc.

Method Attributes

This section describes the class and method attributes supported for the Subscription control.

Class attributes include:

channel-name
: The name of the Message Broker channel to which the control subscribes. This is a required class-level annotation that cannot be overridden.

xquery
: The XQuery expression that is evaluated for each message published to a subscribed channel. Messages that do not satisfy this expression are not dispatched to a subscribing business process. This is an optional class-level annotation that cannot be overridden.

Method attributes include:

filter-value-match
: The *filter-value* that the XQuery expression results must match for the message to be dispatched to a subscribing business process. This is an optional method-level annotation. Valid values for the *filter-value-match* annotation include a string constant that is compared directly to the XQuery results, or a method parameter in curly braces. For example: `{parameter1}`

Callback method attributes include:

message-metadata

The name of a parameter in the callback method that receives the metadata from the message that triggered the subscription. This parameter must be of type `XmlObject` (or a typed XML Bean class).

message-body

The name of a parameter in the callback method that receives the body from the message that triggered the subscription. This parameter must be of type `XmlObject` (or a typed XML Bean class), `String`, `RawData`, or a non-XML MFL class (a subclass of `MflObject`).

Example Code for MB Subscription Control

MB Subscription controls must be extended. The following is an example of a MB Subscription control file.

```
package requestquote;

import org.apache.beehive.controls.api.bean.ControlExtension;

import org.apache.beehive.controls.api.events.EventSet;

import org.apache.xmlbeans.XmlObject;

/*

 * A custom Subscribe control.

 */

@ControlExtension

@SubscriptionControl.ClassSubscription(channelName = "/deadletter/xml")

public interface SubscriptionControl extends

              com.bea.control.SubscriptionControl {


       @SubscriptionControl.MethodSubscription(filterValueMatch =
"{value}")

       void subscribeWithFilterValue(String value);


       @EventSet

       interface Callback {
```

```
            @SubscriptionControl.SubscriptionCallback(body =
"{message}", metadata = "{metadata}")
            void onMessage(XmlObject message, XmlObject metadata);
      }

      static final long serialVersionUID = 1L;

      public SubscriptionControl create();
}
```

Note About Static and Dynamic Subscriptions to Message Broker Channels

In addition to the dynamic subscriptions you design at **Control** nodes in your business process, you can design static subscriptions at **Start** nodes to receive messages from Message Broker channels.

To learn how to design static subscriptions to Message Broker channels at business process Start nodes, see Designing Start Nodes.

Using Event Generators to Publish to Message Broker Channels

Event generators publish messages to Message Broker channels. WebLogic Integration supports the following event generators:

- Email event generators
- File event generators
- HTTP event generators
- JMS event generators
- MQSeries event generators
- Timer event generators

MQSeries Control

Note: The MQSeries control is available in BEA WorkSpace Studio only for licensed users of WebLogic Integration.

MQSeries is a middleware product from IBM that runs on multiple platforms. It enables message transfer between applications; the sending application PUTs a message on a Queue, and the receiving application GETs the message from the Queue. The sending and receiving applications do not have to be on the same platform, and can execute at different times. MQSeries manages all the storage, logging and communications details required to guarantee delivery of the message to the destination queue.

Disclaimer

Use of the MQSeries control and event generator with BEA WebLogic Integration in no manner confers or grants the right to use MQSeries control including "dynamic libraries". In order to use such IBM products, the user of the MQSeries control and event generator must obtain a valid license from IBM.

Topics Included in This Section

Overview: MQSeries Control

Describes the function of the MQSeries control within WebLogic Integration.

Overview: MQSeries Control

The MQSeries control enables WebLogic Integration business processes to send and receive messages using MQSeries queues. Using the MQSeries control, you can send and receive Binary, XML, and String messages. You can specify MQSeries control properties while configuring the MQSeries control or dynamically at run time. You can also set the transaction boundaries for the MQSeries business operations.

The MQSeries control complements the other controls provided in WebLogic Integration, and can be used with other WebLogic Integration business processes. For information on how to add control instances to business processes, see Using Controls in Business Processes.

The MQSeries Event Generator can be used for polling specific MQSeries queues for incoming messages. For more information, see Using the MQSeries Event Generator.

Prerequisites to Adding an MQSeries Control

Before adding the MQSeries control to the BEA WorkSpace Studio, you must complete the following tasks:

1. Install the WebSphere MQSeries client on your machine.
2. Add the `com.ibm.mq.jar` file from the MQSeries client installation to the system environment CLASSPATH variable.
3. Optionally, enable MQSeries control logging by adding the following code to the `apacheLog4jCfg.xml` file:

   ```
   <category name=”com.bea.control.MQControl”>
   <!-- NOTE: DO NOT CHANGE THIS PRIORITY LEVEL W/O WLI DEV APPROVAL -->
   <!-- Debug-level log information is frequently the only tool available
   to diagnose failures! -->
   <priority value="warn"/>
   <appender-ref ref="SYSLOGFILE"/>
   <appender-ref ref="SYSERRORLOGFILE" />
   </category>
   ```

   The MQSeries control uses the Workshop debugger for logging messages.

   Note: To change the log level, refer to the control documents.
4. In BEA WorkSpace Studio, import the `com.ibm.mq.jar` file from the MQSeries client installation into the Libraries folder of the application where the MQSeries control is used.

Now you can add a new MQSeries control to send and receive messages.

Creating and Configuring a New Instance of the MQSeries Control

You can create and configure a new instance of the MQSeries control and add it to your business process. This topic includes the following sections:

To Add a New MQSeries Control

To add a new MQSeries control to WebLogic Integration, perform the following steps:

1. In the Package Explorer pane, double-click the business process (Process.java file) to which you want to add the MQ Series control. The business process is displayed in the Design view.

2. Click on the Data Palette and from the drop-down list choose Integration Controls to display the list of controls used for integrating applications.

Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **MQSeries Control**.

   The **Insert Control: MQSeries** dialog box is displayed (see Figure 9-1).

Figure 9-1 Insert Control: MQSeries

Note: If you are creating the control for the first time, the **Locate the MQ Series jar file** dialog box will is displayed. Browse for the **com.ibm.mq.jar** file located in the MQ series installation folder and click **Open**.

4. In the **Insert control: MQSeries** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new MQSeries control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.
5. In the **Create Control** wizard enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.

     The **Insert control: MQSeries** dialog-box is displayed.
6. Configure the following settings in the **Insert control: MQSeries** dialog-box as mentioned below:

– To configure the general settings for your MQSeries control, see To Specify MQSeries Control General Settings

– To specify connection settings for your MQSeries control, see To Specify MQSeries Control Connection Settings

– To specify connection settings for your MQSeries control, see To Specify MQSeries Control Authorization Settings

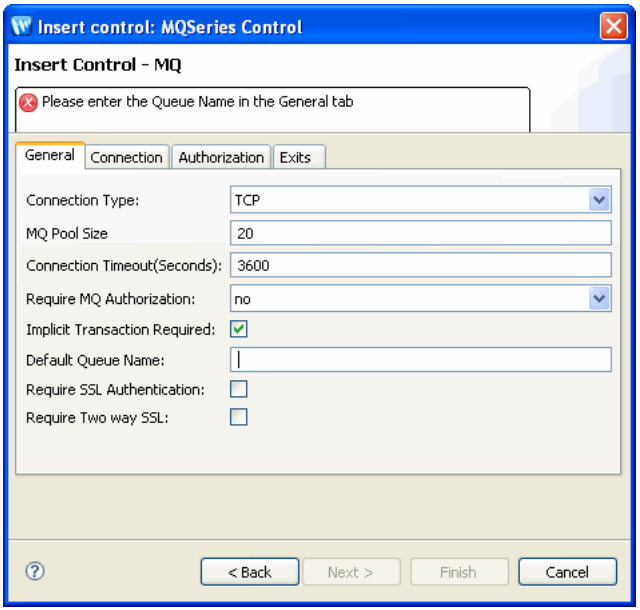
– To use the MQSeries exit implementation, see To Specify MQSeries Control Exits

7. Click **Finish**.

To Specify MQSeries Control General Settings

To specify connection settings for the MQSeries control, perform the following tasks (see Figure 9-2):

Figure 9-2 General



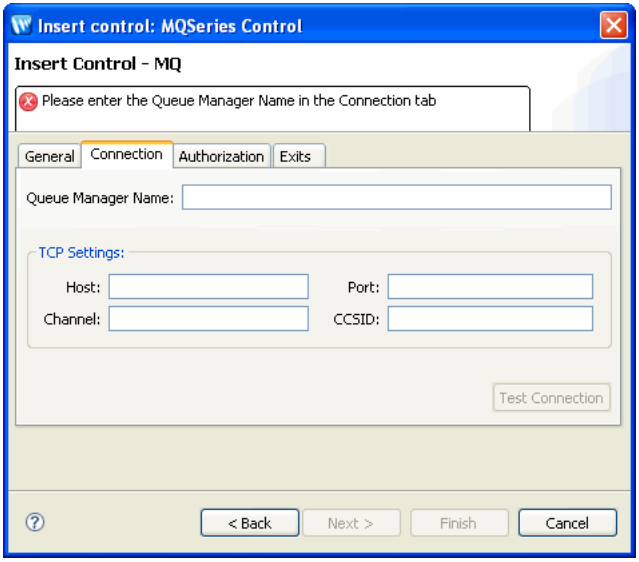
1. From the **Connection Type** drop-down list, select the type of connection that you want to establish; a Bindings or TCP type connection. Using the Bindings connection type, you can only get a connection to queue managers on the local system. Using the TCP connection type, you can also get connections to remote queue managers.

2. In the **MQ Pool Size** text box, enter the number of MQSeries connections to be maintained in the MQSeries connection pool.
3. In the **Connection Time-out (Seconds)** field, enter the number of seconds after which the connection should time out.
4. From the **Require MQ Authorization** drop-down list, select either **Yes** or **No**. MQ authorization is applicable only in TCP mode. To get MQ authorization, you must enter the MQSeries user name and password in the **Authorization** tab.
5. The **Implicit Transaction Required** option is selected by default. When selected, the MQSeries control handles transactions implicitly for each Put and Get individually, without having to set an explicit transaction boundary. When this option is not selected, you must explicitly set the transaction boundaries. For more information, see Understanding Transaction Management.
6. In the **Default Queue Name** field, enter the default queue name to be used by the MQSeries control for sending and receiving messages.
7. Select **Require SSL Authentication** if you want to enable server-side authentication using SSL (one way) for this instance of the MQSeries control.
8. Select **Require Two way SSL** if you want to enable client-side authentication also using SSL (two way) for this instance of the MQSeries control.

To Specify MQSeries Control Connection Settings

To specify TCP/IP settings for the MQSeries control, perform the following tasks, in the **Connection** tab (see Figure 9-3):

Figure 9-3 Connection



1. In the **Connection** tab, in the **Queue Manager Name** field, enter the name of the Queue Manager for which the connection is being obtained.

   Note: Specify TCP/IP settings only if you have set your connection type to TCP.

2. In the **Host** field, enter the name of the host machine containing the queue manager to which you want to connect.

3. In the **Port** field, enter the port number on which the queue manager is available for connection.

4. In the **Channel** field, enter the MQSeries server connection channel configured in the queue manager.

5. In the **CCSID** field, enter the Coded Character Set to be used when a connection is established. The CCSID is used mainly for internationalization support.

6. Click the **Test Connection** button to validate the values entered, and to check that you can connect to the queue manager.

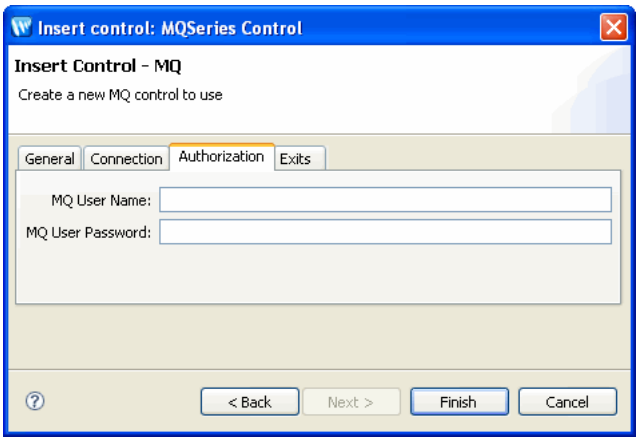
   Note: The TCP Setting of Connection tab is enabled only when the TCP connection mode is selected

WARNING: Do not click the **Test Connection** button when selecting the SSL option for your control. Clicking this button will cause your connection to fail.

To Specify MQSeries Control Authorization Settings

In the **Authorization** tab (see Figure 9-4), specify user name and password for MQ authorization, perform the following tasks:

Figure 9-4 Authorization

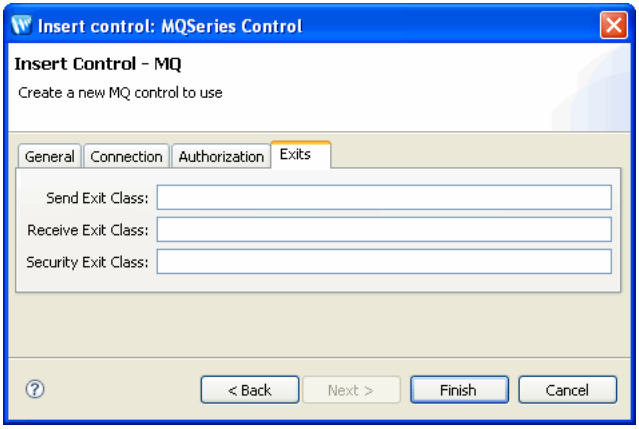


1. In the **MQ User Name field**, enter your MQ user name.
2. In the **MQ User Password** field, enter your MQ password.

To Specify MQSeries Control Exits

In the **Exits** tab (see Figure 9-5), specify MQSeries control exits, perform the following tasks:

Figure 9-5 Exits



1. In the **Exits** tab, in the **Send Exit Class** field, enter the fully qualified name of the class implementing the MQSeries MQSendExit interface.
2. In the **Receive Exit Class** field, enter the fully qualified name of the class implementing the MQSeries MQReceiveExit interface.
3. In the **Security Exit Class** field, enter the fully qualified name of the class implementing the MQSeries MQSecurityExit interface.

   For more information, see Using the MQSeries Control Exit Implementation.

   Note: The **Exits** tab is enabled only when TCP connection mode is selected. The fields in this tab are not mandatory.

The Control File for an MQSeries Control

When you create a new instance of the MQSeries control, you create a new Control file in your project. The following is a sample control file for an MQSeries control:

```
package requestquote;

import org.apache.beehive.controls.api.bean.ControlExtension;

/*
 * A custom MQ control.
 */
@ControlExtension
```

```
@com.bea.control.MQControl.Connection(type =
com.bea.control.MQControl.ConnectionType.TCP, QueueManager = "Manager",
authorization = false, implicitTransaction = true)

@com.bea.control.MQControl.ConnectionPool(poolSize = 20, timeout = 3600)

@com.bea.control.MQControl.TCPSettings(host = "10.12.45.89", port = "1234",
channel = "Host", ccsid = "1234", user = "", password = "", sendExit = "",
receiveExit = "", securityExit = "")

@com.bea.control.MQControl.SSLSettings(sslRequired = false, twoSSLRequired
= false)

public interface MQSeriesControl extends com.bea.control.MQControl {


       static final long serialVersionUID = 1L;


       public MQSeriesControl create();

}
```

The contents of the MQSeries control file depend on the selections made in the Insert MQSeries dialog. The example above was generated based on selecting a TCP connection type.

Using the MQSeries Control Exit Implementation

The MQSeries control allows you to create your own send, receive, and security exits.

To implement an Exit, you must define a new Java class that implements the appropriate interface. Three exit interfaces are defined in the WebSphere MQ package:

- MQSendExit

  The MQSeries MQSendExit interface allows you to examine and change the data sent to the queue manager by the WebSphere MQ Client for Java.

- MQReceiveExit

  The MQSeries MQReceiveExit interface allows you to examine and change the data received from the queue manager by the WebSphere MQ Client for Java.

- MQSecurityExit

The MQSeries MQSecurityExit interface allows you to customize the security settings for connecting to a queue manager.

Notes: User Exits are supported for TCP connections only; they are not supported for bindings connections.

User Exits are used to modify the data that is transmitted between the MQSeries queue manager and the MQSeries client application. This data is in the form of MQSeries headers and does not involve the contents of the actual message being put and received from the queue.

Implementing MQSeries Exits

To implement MQSeries Exits, perform the following tasks:

1. Create the Java class that implements the `com.ibm.mq.MQSendExit`, `com.ibm.mq.MQReceiveExit`, and `com.ibm.mq.MQSecurity` interfaces for the send, receive, and security exits, as shown in the following example:

```
package com.bea.UserExit;
import com.ibm.mq.*;
public class MQUserExit implements MQSendExit, MQReceiveExit,
MQSecurityExit {
public MQUserExit()
{
}
public byte[] sendExit(MQChannelExit channelExit,MQChannelDefinition
channelDefnition,byte[] agentBuffer)
{
return agentBuffer;
}
public byte[] receiveExit(MQChannelExit channelExit,MQChannelDefinition
channelDefnition,byte[] agentBuffer)
{
return agentBuffer;
}
public byte[] securityExit(MQChannelExit
channelExit,MQChannelDefinition channelDefnition,byte[] agentBuffer)
{
return agentBuffer;
}
}
```

   You may implement these interfaces in a single class or in separate classes, as required.

For an MQSendExit, the `agentBuffer` parameter contains the data to be sent. For an MQReceiveExit or an MQSecurityExit, the `agentBuffer` parameter contains the data just received.

For the MQSendExit and the MQSecurityExit, your exit code should return the byte array that you want to send to the server. For a Receive exit, your exit code must return the modified data that you want WebSphere MQ Client for Java to interpret.

2. Bundle the given class in a Jar file, for example, *mquserexits.jar.*

3. Place the Jar file in the WebLogic classpath. Edit the `setDomainEnv.cmd` file located in the WebLogic domain directory to append the Jar file name to the CLASSPATH. To do this, find the following code in the `setDomainEnv.cmd` file:

   ```
   set Pre_CLASSPATH=
   ```

   and append the following line to it:

   ```
   ;%EXIT_DIR%\mquserexits.jar
   ```

   Before you append the code containing the Jar file name to the CLASSPATH, you can define the directory in which the Jar file resides, as follows:

   ```
   set EXIT_DIR=D:\UserExits
   ```

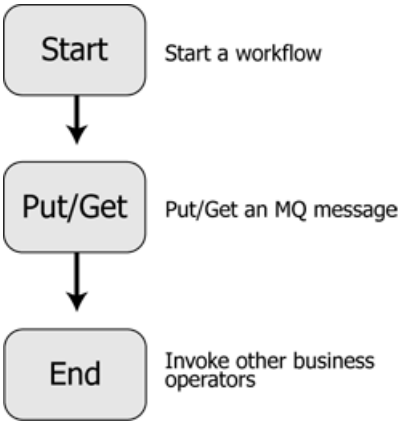
Understanding Transaction Management

Two modes of transaction management are supported by the MQSeries control. They both use the underlying MQSeries syncpoint feature. The two transaction management modes are:

- Implicit Transaction Management
- Explicit Transaction Management

Implicit Transaction Management

Implicit transaction management is selected by default. When this mode is on, the MQSeries control handles the transaction for each MQSeries Get or Put function. The following diagram describes how an implicit transaction is handled by the MQSeries control.



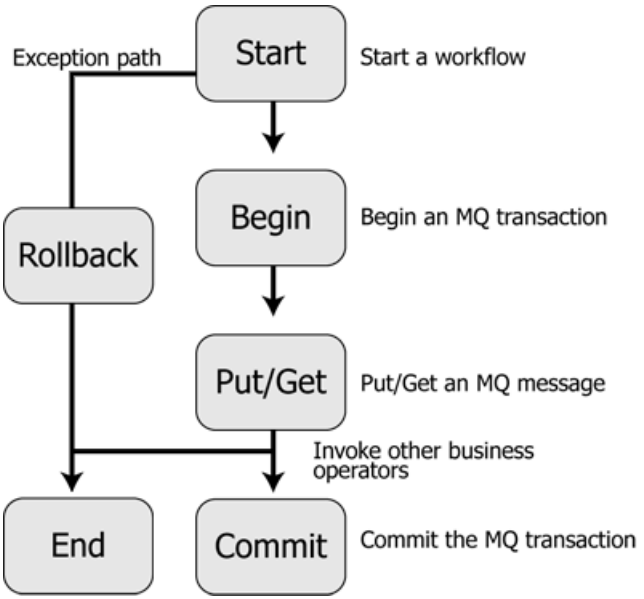
Using implicit transaction management prevents you from grouping several Get and Put functions together as a part of a transactional unit. Each Get and Put are handled individually within a transaction boundary.

Explicit Transaction Management

Explicit transaction management is enabled when you choose not to use implicit transaction management while configuring the MQSeries control. In the explicit transaction mode, you must set the transaction boundaries explicitly, using the Begin and Commit (or Rollback) MQSeries control functions.

Figure 9-6 describes the process of creating a workflow using explicit transaction management.

Figure 9-6 Workflow using Explicit Transaction Management



Using Message Descriptors

A Message Descriptor is an attribute representing a property of the message that is either being sent or received. Message properties can be the type of message, the message ID, and the message priority. For a detailed list of all the message descriptors supported by the MQSeries control, see Table 9-1, “Elements of the MQMDHeaders XML document”.

Using the MQSeries control you can set Message Descriptors for each message while sending the message using the `putMessage` function. You can also get the message descriptors of the messages retrieved from the queue. This facility is supported using the MQMDHeaders document which is provided as an input to the `putMessage` and `getMessage` functions. The MQMDHeaders document is represented using an XMLBean that conforms to the MQMDHeaders schema present in the `jpdpublic.jar` file, located at `BEA_Home\wli_10.2\lib`.

The following elements of the MQMDHeaders XML document can be set as part of the MQMD parameters:

Table 9-1 Elements of the MQMDHeaders XML document

Element Name	Description	Permissible Values	Relevance
MessageType	Type of message	8-Datagram 1-Request 2-Reply Other positive integers are also accepted if they are within the Application or System defined ranges specified by MQSeries.	Put Request, Put Response, Get Response
MessageId	Id of message	Hexadecimal string	Put Request, Put Response, Get Request, Get Response
CorrelationId	Correlation Id of the message	Hexadecimal string	Put Request, Put Response, Get Request, Get Response
GroupMessage	This element is required to send and receive group messages.		Put Request, Put Response, Get Request, Get Response
GroupId	Group Id of the message	Hexadecimal string	Put Request, Put Response, Get Request, Get Response
Priority	Message priority	0-9	Put Request, Put Response, Get Response
Format	Message format	String values representing valid built-in MQSeries formats or user-defined formats. The string values are present in `MQC.MQFMT_*`.	Put Request, Put Response, Get Response
CharacterSet	Character Set of the message	Valid MQSeries Character set	Put Request, Put Response, Get Response

Table 9-1 Elements of the MQMDHeaders XML document (Continued)

Element Name	Description	Permissible Values	Relevance
Persistence	Persistence property of the message	0- non-persistent message. 1-persistent message	Put Request, Put Response, Get Response
Segmentation	Segmentation property of the message	0- segmentation not allowed. 1- segmentation allowed.	Put Request
Expiry	Message expiration	A positive integer or -1 (for unlimited expiration)	Get Request, Put Response, Get Response
UserId	User Id of the message	A string	Put Request, Put Response, Get Response
MessageSequenceNumber	Message Sequence Number of the message	A positive integer, not 0	Put Request, Put Response, Get Request, Get Response
GroupOptions	use this element in the Put Request only if the message being Put is a group message. In the Get Response, this element appears only if the message retrieved is a group message.		Put Request, Get Response
IsLastMessage	Identifies the last message of a group message. This element accepts boolean values.	True or False	Put Request, Get Response
ReportOptions	Identifies the report options to be set while sending a message.		Put Request

Table 9-1 Elements of the MQMDHeaders XML document (Continued)

Element Name	Description	Permissible Values	Relevance
COA	Confirmation on Arrival. COA Report options COA - the COA report without any original message data. COAWithData-the COA report with the first 100 bytes of the original message. COAWithFullData - the COA report with all the original message data.	COA, COAWithData, COAWithFullData, None	Put Request
COD	Confirmation of Delivery COD Report options COD - the COD report without any data of the original message. CODWithData - the COD report with the first 100 bytes of the original message. CODWithFullData - the COD report with all the original message data.	COD, CODWithData, CODWithFullData, None	Put Request

Table 9-1 Elements of the MQMDHeaders XML document (Continued)

Element Name	Description	Permissible Values	Relevance
Exception	Exception Report options Exception - the Exception report without any original message data. ExceptionWithData - the Exception report with the first 100 bytes of the original message. ExceptionWithFullData - the Exception report with all the original message data.	Exception, ExceptionWithData, ExceptionWithFullData, None	Put Request
Expiration	Expiration Report options Expiration - the Expiration report without any original message data. ExpirationWithData - the Expiration report with the first 100 bytes of the original message. ExpirationWithFullData - The expiration report with all the original message data.	Expiration, ExpirationWithData, ExpirationWithFullData, None	Put Request
Feedback	Message feedback	A positive integer	Put Request, Put Response, Get Response
ReplyToQueueName	The queue to which the reports or the reply (used only for a request message) should be sent.	String representing a valid queue name	Put Request, Put Response, Get Response
ReplyToQueueManager	The queue manager containing the reply to queue.	String representing a valid queue manager name	Put Request, Put Response, Get Response

Table 9-1 Elements of the MQMDHeaders XML document (Continued)

Element Name	Description	Permissible Values	Relevance
WaitInterval	The lapse time (in milliseconds) before receiving a message.	A positive integer. -1 for unlimited wait interval	Get Request
ApplicationIdData		String value	Put Request, Put Response and Get Response.
ApplicationOriginData		String value	Put Request, Put Response and Get Response.
PutApplType	Put application type of the message	Positive integer value	Put Request, Put Response and Get Response.
PutApplName	Put application name of the message	String value	Put Request, Put Response and Get Response.
PutDateTime	Put date and time of the message	String value	Put Response and Get Response
AccountingToken	Accounting information for the message	Byte array	Put Request, Put Response and Get Response.
Version	Version information of the message descriptor	2 or 1	Put Request, Put Response and Get Response.

Table 9-1 Elements of the MQMDHeaders XML document (Continued)

Element Name	Description	Permissible Values	Relevance
MessageConsumption	Message consumption option for the `getMessage` function. `Browse` - Retrieve the message from the queue (without deleting the message). `Delete` - Delete the message from the queue after retrieving it.	Browse, Delete	Get Request
MQGMO_CONVERT	Specifies whether data conversion is required for the message during a Get operation. This element must be set to True to retrieve messages of the EBCDIC characterset.	True or False	Get Request

Table 9-2 Attributes of the MQMDHeaders document

Attribute Name	Under Element	Description	Values	Relevance
waitForAllMsgs	GroupMessage	Used while retrieving group messages to specify that no message of the group should be retrieved until all the messages of the group are available in the queue. This attribute is normally specified only while retrieving the first message of the group.	True or False	Get Request and Get Response
logicalOrder	GroupMessage	Used while retrieving group messages to specify that the messages of the group should be retrieved in the order of their Message Sequence Number irrespective of the order in the queue. This option is specified while retrieving all the messages of the group.	True or False	Get Request and Get Response

Schema of the MQMDHeaders Document

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://www.bea.com/wli/control/MQMDHeaders"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.bea.com/wli/control/MQMDHeaders"
elementFormDefault="qualified">
<xs:element name="MQMDHeaders">
     <xs:complexType>
          <xs:sequence>
<xs:element name="MessageType" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="MessageId" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="CorrelationId" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="GroupMessage" minOccurs="0" maxOccurs="1">
<xs:complexType>
     <xs:sequence>
<xs:element name="GroupId" type="xs:string" minOccurs="1" maxOccurs="1"/>
```

```
          </xs:sequence>
<xs:attribute name="waitForAllMsgs" type="xs:boolean" use="optional"/>
<xs:attribute name="logicalOrder" type="xs:boolean" use="optional"/>
     </xs:complexType>
          </xs:element>
<xs:element name="Priority" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="Format" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="CharacterSet" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="Persistence" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="Segmentation" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="Expiry" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="UserId" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="MessageSequenceNumber" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="GroupOptions" minOccurs="0" maxOccurs="1">
     <xs:complexType>
          <xs:sequence>
               <xs:element name="IsLastMessage" type="xs:boolean"
minOccurs="1" maxOccurs="1"/>
          </xs:sequence>
     </xs:complexType>
</xs:element>
<xs:element name="ReportOptions" minOccurs="0" maxOccurs="1">
     <xs:complexType>
     <xs:sequence>
     <xs:element name="COA" type="xs:string" minOccurs="0" maxOccurs="1"/>
     <xs:element name="COD" type="xs:string" minOccurs="0" maxOccurs="1"/>
     <xs:element name="Exception" type="xs:string" minOccurs="0"
maxOccurs="1"/>
     <xs:element name="Expiration" type="xs:string" minOccurs="0"
maxOccurs="1"/>
     </xs:sequence>
     </xs:complexType>
     </xs:element>
```

```
<xs:element name="Feedback" type="xs:int" minOccurs="0" maxOccurs="1"/>
<xs:element name="ReplyToQueueName" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="ReplyToQueueManager" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="WaitInterval" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="ApplicationIdData" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="ApplicationOriginData" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="PutApplType" type="xs:int" minOccurs="0" maxOccurs="1"/>
<xs:element name="PutApplName" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="PutDateTime" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="AccountingToken" type="xs:base64Binary" minOccurs="0"
maxOccurs="1"/>
<xs:element name="Version" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="MessageConsumption" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="MQGMO_CONVERT" type="xs:boolean" minOccurs="0"
maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Sample of an MQMDHeaders Document

The following is a sample MQMDHeaders document that contains most of the message descriptors you can set using the MQSeries control:

```
<?xml version="1.0"?>
<even:MQMDHeaders xmlns:even="http://www.bea.com/wli/control/MQMDHeaders">
<even:MessageType>8</even:MessageType>
<even:MessageId>1111</even:MessageId>
<even:CorrelationId>2222</even:CorrelationId>
<even:GroupMessage>
```

```
     <even:GroupId>3333</even:GroupId>
</even:GroupMessage>
<even:Priority>9</even:Priority>
<even:Format>MQSTR</even:Format>
<even:CharacterSet>819</even:CharacterSet>
<even:Persistence>1</even:Persistence>
<even:Segmentation>1</even:Segmentation>
<even:Expiry>5000</even:Expiry>
<even:UserId>WebLogic</even:UserId>
<even:MessageSequenceNumber>1</even:MessageSequenceNumber>
<even:GroupOptions>
     <even:IsLastMessage>true</even:IsLastMessage>
</even:GroupOptions>
<even:ReportOptions>
     <even:COA>COAWithFullData</even:COA>
     <even:COD>CODWithFullData</even:COD>
     <even:Exception>ExceptionWithFullData</even:Exception>
     <even:Expiration>ExpirationWithFullData</even:Expiration>
</even:ReportOptions>
<even:Feedback>1</even:Feedback>
<even:ReplyToQueueName>trial</even:ReplyToQueueName>
<even:ReplyToQueueManager>QM_itpl_025051</even:ReplyToQueueManager>
<even:ApplicationIdData>App_ID_025051</even:ApplicationIdData>
<even:ApplicationOriginData>Windows_app_025051</even:ApplicationOriginData
> <even:PutApplType>1</even:PutApplType>
<even:PutApplName>MQSeriesClient</even:PutApplName>
<even:Version>2</even:Version>
</even:MQMDHeaders>
```

Using XML Beans to Set the MQMDHeader Element Values

The MQSeries control MQMDHeaders document element values can be set, and the return values can be retrieved, programmatically, using XML beans. The following is an example of setting the MQMDHeader element values prior to calling the `putMessage` function:

```
headers =
com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument.Factory.newInstance();
com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument.MQMDHeaders header =
headers.addNewMQMDHeaders();
```

```
header.setMessageType(MQC.MQMT_DATAGRAM);
header.setPriority(8);
header.setExpiry(5000);
header.setPersistence(MQC.MQPER_PERSISTENT);
header.getReportOptions().setCOA("COA");
header.setReplyToQueueName("ReportQueue");
header.setApplicationIdData("Testing");
header.setApplicationOriginData("AAAA");
header.setPutApplName("Websphere MQ 2");
header.setPutApplType(MQC.MQAT_JAVA);
```

Sending and Receiving Messages

You can send and receive messages with the MQSeries control using the Put and Get functions. Messages can be of the form Bytes, String, or XML data.

Sending Messages

To send a message, select a `putMessage` function based on the data type of the message that you want to send:

- `MQMDHeadersDocument putMessageAsBytes (byte[] message, java.lang.String queue, MQMDHeadersDocument mqmd) throws ResourceException;`
- `MQMDHeadersDocument putMessageAsString (String message, java.lang.String queue, MQMDHeadersDocument mqmd) throws ResourceException;`
- `MQMDHeadersDocument putMessageAsXml (XmlObject message, java.lang.String queue, MQMDHeadersDocument mqmd) throws ResourceException;`

The first parameter that is passed to the function is the message to be put into the queue. The possible types for this parameter are byte[], XmlObject and String for sending Binary, XML and plain text messages respectively.

The second parameter that is passed to the function is the queue to which the message is sent. If a value is not provided at runtime, that is, if the value is null, the default queue name mentioned in the control property is used.

The third parameter that is passed to the function is the XML bean representing the MQMDHeadersDocument provided as an XML document during runtime, which conforms to

the MQMDHeaders schema. The values provided in this document are used for setting the MQMD attributes of the message being sent. If the MQMDHeadersDocument parameter is not provided, or if the parameter is null, the message is put into the queue with the default values for the message descriptors.

The return value of the function is the MQMDHeadersDocument representing the MQMD attributes of the message sent to the queue.

Using the putMessage Function In a Business Process

The following procedure describes how to add any MQSeries control `putMessage` function to a business process.

1. Open the **Client Request** node.
2. In the **General Settings** tab, enter a name for the new method.
3. Click **Add**, then select **MQMDHeadersDocument** from the XML Types list. Enter a name for the variable in the **Name** field. Click **OK** to add your selection to the **Client Request** node. This represents the input MQMDHeaders document for the `putMessage` function.
4. Click **Add** again, then select **String** from the **Java datatype** list. Enter a name for the variable in the **Name** field. Click **OK** to add your selection to the **Client Request** node. This represents the queue name for the `putMessage` function.
5. Click **Add** again, then select **String** from the **Java datatype** list. Enter a name for the variable in the **Name** field. Click **OK** to add your selection to the **Client Request** node. This represents the message for the `putMessage` function.
6. In the **Receive Data** tab, create a new variable for each of the three parameters that you created in the **General Settings** tab of the **Client Request** node. You must provide variable names for all three variables. The variable type is pre-defined, based on the parameters to which you are assigning the variable.
7. Close the **Client Request** node.
8. Drag and drop the `putMessageAsString` function from the **Controls** tab in the **Data Palette** into your business process, just below the **Client Request** node.
9. Open the **Send Data** tab of the `putMessageAsString` function node. From the **Select variables to assign** drop-down list, assign the variables that you created in the **Receive Data** tab of the **Client Request** node, to the corresponding parameter of the `putMessageAsString` function listed in the **Control Expects** column.

10. Open the **Receive Data** tab of the `putMessageAsString` function node. From the **Select variables to assign** drop-down list, create a new variable in which to store the output of the `putMessageAsString` function, the MQMDHeaders document, which represents the attributes of the message that was sent.

You can use similar steps to send messages using the `putMessageAsBytes` or the `putMessageAsXml` functions.

Sending Messages To a Remote Queue Manager

Using the MQSeries control you can add messages to a remote queue managed by a remote queue manager. To do this, you must configure a transmission queue in the queue manager to which the MQSeries control is connected. For more information on how to configure a transmission queue, see the MQSeries documentation on http://www.IBM.com.

To add a message to a remote queue (managed by a remote queue manager) you must drag and drop the following function, before the putMessage call in the workflow:

```
void setRemoteQueueManager(java.lang.String remoteQueueManager);
```

The parameter to this function is the name of the remote queue manager. To set the value for this parameter, in the Design view, open the `remoteQueueManager` node. In the **Send Data** tab, select or create a string variable, then enter the name of your remote queue manager as the default value.

Once you've configured the remote queue manager, the putMessage functions following the `setRemoteQueueManager` function add messages to the remote queue manager.

To revert to the default (local) queue manager to which your control is connected, in the Design view you must drag and drop the `setRemoteQueueManager` again in your workflow. On doing this, a default value, null, is passed as the parameter to this function. This null value or empty string reverts control back to the default queue manager. messages are now automatically added to the local queue.

Receiving Messages

To receive a message, select a `get Message` function based on the data type of the message that you want to receive:

- `byte[] getMessageAsBytes(java.lang.String queue, MQMDHeadersDocument mqmd) throws ResourceException;`
- `String getMessageAsString(java.lang.String queue, MQMDHeadersDocument mqmd) throws ResourceException;`

- `XmlObject getMessageAsXml(java.lang.String queue, MQMDHeadersDocument mqmd) throws ResourceException;`

The first parameter of the function, `java.lang.String queue`, is the name of the queue from which the message is to be received. If a value is not provided at runtime (the value is null) the default queue name mentioned in the control property is used.

The second parameter of the function, `MQMDHeadersDocument`, is an XML bean. This represents the MQMDHeadersDocument provided as an XML document at runtime, which conforms to the MQMDHeaders schema. The values provided in this document are used to retrieve the message corresponding to the MQMD attributes specified in the document. If the MQMDHeadersDocument parameter is not provided, or if the parameter is null, the first message present in the queue is retrieved. If the MQMDHeadersDocument parameter is not null, the MQMD attributes of the message obtained from the queue are updated in this XML bean object itself.

Note: If the MQMDHeadersDocument parameter to the `getMessage` function is null, you must use the `getMQMDHeaders` function after the `getMessage` function in the workflow, to get the MQMD attributes of the message retrieved from the queue. Also, if the MQMDHeadersDocument parameter to the `getMessage` function is null, `Delete` is used as the default MessageConsumption option.

The return value of the function is the message obtained from the queue. The data type of the message depends on the `getMessage` function added. The values may be byte[], XmlObject, or String, depending on whether the message obtained is to be processed as a Binary, XML, or plain text message.

Using the getMessage Function In a Business Process

The following procedure describes how to add a MQSeries control `getMessage` function to a business process.

1. Open the **Client Request** node.
2. In the **General Settings** tab, enter a name for the new method.
3. Click **Add**, then select **MQMDHeadersDocument** from the **XML Types** list. Enter a name for the variable in the **Name** field. Click **OK** to add your selection to the **Client Request** node. This represents the input MQMDHeaders document for the `getMessage` function.
4. Click **Add** again, then select **String** from the **Java datatype** list. Enter a name for the variable in the **Name** field. Click OK to add your selection to the **Client Request** node. This represents the queue name for the `getMessage` function.

5. In the **Receive Data** tab, create a new variable for each of the two parameters that you created in the **General Settings** tab of the **Client Request** node. You must enter variable names for the two variables. The variable type is pre-defined based on the parameters to which they are assigned.

6. Close the **Client Request** node.

7. Drag and drop the `getMessageAsString` function from the **Controls** tab in the **Data Palette** into your business process, just below the **Client Request** node.

8. Open the **Send Data** tab of the `getMessageAsString` function node. From the **Select variables to assign** drop-down list, assign the variables that you created (in the **Receive Data** tab of the **Client Request** node) to the corresponding parameter of the `getMessageAsString` function listed in the **Control Expects** column.

9. Open the **Receive Data** tab of the `getMessageAsString` function node. From the **Select variables to assign** drop-down list, create a new variable in which to store the output of the `getMessageAsString` function. The output is a string representing the message that was retrieved from the queue.

   The Message Descriptor attributes of the message retrieved from the queue are updated in the MQMDHeaders document. This document was provided as input to the `getMessageAsString` function.

You can use a similar procedure to retrieve messages using the `getMessageAsBytes` or the `getMessageAsXml` functions.

Sending Group messages

You can send group messages using the MQSeries control `putMessage` function within a loop. The loop can be created using one of the following process nodes: While Do, Do while, and For Each.

Provide the `GroupOptions` element in the MQMDHeadersDocument. You only provide this element in the input MQMDHeaders XML document if a group message is to be sent.

In the MQMDHeaders document, set the `IsLastMessage` element within `GroupOptions` to `False`, for all messages except the last message. For the last message, the `IsLastMessage` element must be set to `True`.

If you specify a `GroupID` for the first message, then the MQSeries control assigns this ID to the group message. If you do not specify a `GroupID` for the first message, the MQSeries queue manager assigns a group ID to the first message. This ID is returned in the output MQMDHeaders document of the `putMessage` function.

The Group Id assigned to the first message must be used for all the subsequent messages of the group. The `MessageSequenceNumber` of the first message of the group should be 1; the `MessageSequenceNumber` of the second message should be 2, and so on.

Retrieving Group Messages

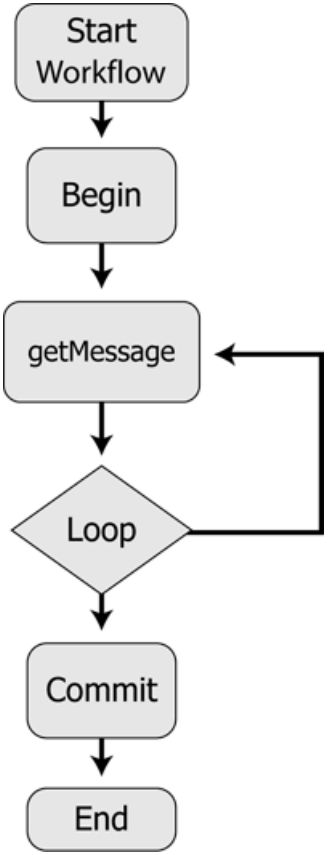
You can retrieve group Messages using the MQSeries control `getMessage` function within a loop. The loop can be created using one of the following process nodes: While Do, Do while, and For Each.

Setting the logicalorder Attribute

You can retrieve group messages using the MQSeries control in a logical order. To configure the MQSeries control to retrieve group messages in a logical order, set the `logicalOrder` attribute of the `GroupMessage` element to `True`.

You can retrieve messages in a logical order only when you use explicit transaction mode. The Figure 9-7 depicts a sample workflow for retrieving group messages in logical order:

Figure 9-7 Sample Workflow



The loop executes until the `IsLastMessage` element within the `GroupOptions` element is set to `True` in the response MQMDHeaders document of the `getMessage` function.

Note: The `GroupOptions` element does not appear in the Get Response MQMDHeaders document if the retrieved message is not a part of a group.

The `logicalOrder` attribute must be set to `True` in each call of the Get service, to retrieve the messages of the group in their logical order (by message sequence number, beginning at one for the first message).

Changing the `logicalOrder` attribute from True to `False` while getting group messages, when its value was `True` in the previous Get service call, changes the logical ordering.

Setting the `logicalOrder` attribute to `False` or not providing this attribute in the Get request document means that the control gets the first message of the group as it appears on the queue irrespective of its message sequence number.

The following is an example of a Get Request MQMDHeaders document for retrieving group messages in logical order, and also waits for all messages in the group:

```
<?xml version="1.0"?>
<even:MQMDHeaders xmlns:even="http://www.bea.com/wli/control/MQMDHeaders">
<even:GroupMessage waitForAllMsgs="true" logicalOrder="true">
<even:GroupId></even:GroupId>
</even:GroupMessage>
<even:MessageConsumption>Delete</even:MessageConsumption>
</even:MQMDHeaders>
```

Setting the waitForAllMsgs Attribute

You can configure the MQSeries control to wait for all messages of the group to be present in the queue before retrieving any message within the group. To configure the MQSeries control to wait for all messages, set the `waitForAllMsgs` attribute of the `GroupMessage` element to `True`.

Note: The `waitForAllMsgs` and the `logicalOrder` attribute are optional and can be set to either `True` or `False`.

You can set the `waitForAllMsgs` to True while retrieving the first message of the group. After you retrieve the first message in the group, you can set this attribute to True again, for retrieving the other messages of the group, provided that you have also set the `logicalOrder` attribute to `True`.

Setting the `waitForAllMsgs` attribute to `False`, or not providing this attribute in the Get Request document means that the control can still get group messages from the queue even when not all of the messages of the group are present in the queue.

Setting the GroupId element

`GroupId` is an optional element within the `GroupMessage` element. Its value may not be provided if the hexadecimal group ID of the group message is not known. When there are multiple group messages present in the queue, the first group message in the queue is retrieved. The `GroupId` value may be specified, if known. If specified and there are multiple group messages in the queue, the group message matching the group ID is retrieved.

Setting the MessageSequenceNumber Element

Group Messages can also be retrieved by specifying the `MessageSequenceNumber` element and the `GroupId`. Messages can be retrieved in this way only if the `logicalOrder` attribute value is `False` or is not provided. When the `MessageSequenceNumber` and the `GroupId` are provided, the message of the group matching the `MessageSequenceNumber` is retrieved. The group messages can still be retrieved in a loop by providing the `GroupId` and incrementing the `MessageSequenceNumber` by one in each Get function call in the loop, the `MessageSequenceNumber` of the first message being one.

Working with MQSeries Message Descriptor Format

Format is a message descriptor attribute. Messages of a particular Format conform to a specific structure which depends on the Format type. For example, CICS, IMS, MQRFH2, and so on. The structure for each built-in MQSeries Format is different and is defined by MQSeries. For more information on MQSeries Formats, see the online MQSeries documentation at the following URL:

`http://www.ibm.com`

Using the MQSeries control you can send messages that correspond to built-in MQSeries formats and user-defined formats. This can only be done using the `putMessageAsBytes` function.

To send a message that conforms to an MQSeries Format, you must add Java code to the business process file. This is shown in the following examples.

Example: Sending a message that conforms to the CICS Format (using the putMessage function)

1. Declare a variable, for example, `putin`, in your business process file, as follows:

   ```
   public com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument putin;
   ```

   This variable represents the input MQMDHeaders document XMLBean variable for the `putMessage` function.

2. Drag and drop the **Perform** node from the **Palette** into the business process, just below the **Client Request** node.

3. Open the **Perform** node in the **Source View** and add the following code.

   ```
   public void perform() throws Exception
   {
   putin.getMQMDHeaders().setFormat(MQC.MQFMT_CICS);
   bytmsg = getCICSHeader();
   }
   ```

```
public byte[] getCICSHeader() throws Exception    {
ByteArrayOutputStream bstream = new ByteArrayOutputStream();
DataOutputStream ostream = new DataOutputStream (bstream);
ostream.writeChars("CIH "); // Struct id

ostream.writeInt(1);      // Version
ostream.writeInt(164);    // StrucLength
ostream.writeInt(273);    // Encoding
ostream.writeInt(819);    // CodedCharSetId
ostream.writeChars("        "); // Format
ostream.writeInt(0);      //Flags
ostream.writeInt(0);      //ReturnCode
ostream.writeInt(0);      //CompCode
ostream.writeInt(0);      //Reason
ostream.writeInt(273);    //UOWControl
ostream.writeInt(-2);     //GetWaitInterval
ostream.writeInt(1);      //LinkType
ostream.writeInt(-1);     //OutputDataLength
ostream.writeInt(0);      //FacilityKeepTime
ostream.writeInt(0);      //ADSDescriptor
ostream.writeInt(0);      //ConversationalTask
ostream.writeInt(0);      //TaskEndStatus
ostream.writeBytes("\0\0\0\0\0\0\0\0");   //Facility
ostream.writeChars("    ");   //Function
ostream.writeChars("    ");   //AbendCode
ostream.writeChars("        ");   //Authenticator
ostream.writeChars("        ");   //Reserved1
ostream.writeChars("        ");   //ReplyToFormat
ostream.writeChars("    ");  //RemoteSysId
ostream.writeChars("    ");  //RemoteTransId
ostream.writeChars("    ");  //TransactionId
ostream.writeChars("    ");  //FacilityLike
ostream.writeChars("    ");  //AttentionId
ostream.writeChars("    ");  //StartCode
ostream.writeChars("    ");  //CancelCode
ostream.writeChars("    ");  //NextTransactionId
ostream.writeChars("        "); //Reserved2
ostream.writeChars("        ");  //Reserved3
ostream.writeChars("HelloWorld");
ostream.flush();
byte[] bArr = bstream.toByteArray();
return bArr;
}
```

This code sets the `Format` element in the input MQMD Headers document of the `putMessage` function to `MQC.MQFMT_CICS`, represented by the String "`MQCICS`".

The `getCICSHeader` function writes the fields present in the CICS header to a byte array output stream and returns an array of bytes. The field values given in this example can be

modified as required. The actual message can be appended to the end of this byte array and can be Put into the MQSeries queue. The byte array can be provided as the first parameter to the `putMessageAsBytes` function, which is added to the process.java file after the Perform node. For more information on the `putMessage` function, see Sending and Receiving Messages.

Example: Sending a message that conforms to the IMS Format (using the putMessage function)

1. Declare a variable, for example, `putin`, in the business process file, as follows:

   ```
   public com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument putin;
   ```

   This variable represents the input MQMDHeaders document XMLBean variable for the `putMessage` function.

2. Drag and drop the **Perform** node from the Node Palette into the business process, just below the **Client Request** node.

3. Open the **Perform** node in the Source view and add the following code.

   ```
   public void perform() throws Exception
   {
   putin.getMQMDHeaders().setFormat(MQC.MQFMT_IMS);
   bytmsg = getIMSHeader();
   }
   public byte[] getIMSHeader() throws Exception    {
   ByteArrayOutputStream bstream = new ByteArrayOutputStream();
   DataOutputStream ostream = new DataOutputStream (bstream);

   ostream.writeBytes("IIH "); // Struct id
   ostream.writeInt(1);      // Version
   ostream.writeInt(84);     // Length
   ostream.writeInt(0);      // Encoding
   ostream.writeInt(0);      // CodedCharacterSet
   ostream.writeBytes("        "); // Format (8 characters)
   ostream.writeInt(0);      // Flags
   ostream.writeBytes("        "); // LTermOverride
   ostream.writeBytes("        "); // MFSMapName
   ostream.writeBytes("        "); // ReplyToFormat
   ostream.writeBytes("        "); // Authenticator
   ostream.writeBytes("\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0");  //
   TransInstanceId
   ostream.writeBytes(" ");  //Transtate
   ostream.writeBytes("1");  // CommitMode
   ostream.writeBytes("F");  // Security Scope
   ostream.writeBytes(" ");  // Resrved
   ostream.writeChars("HelloWorld");
   ostream.flush();
   byte[] bArr = bstream.toByteArray();
   ```

```
return bArr;
}
```

The previous lines of code set the `Format` element in the input MQMD Headers document of the `putMessage` function to `MQC.MQFMT_IMS,` represented by the String "`MQIMS` ".

The `getIMSHeader` function writes the fields present in the IMS header structure to a byte array output stream and returns an array of bytes. The values of the fields given in this example can be modified as required. The actual message can be appended to the end of the byte array and can be Put into the MQSeries queue. The byte array can be provided as the first parameter to the `putMessageAsBytes` function, which is added to the business process file after the Perform node. For more information on the `putMessage` function, see Sending and Receiving Messages.

Example: Sending a message that conforms to the MQRFH2 Format (using the putMessage function)

1. Declare a variable, for example, `putin`, in the business process file of your process project in the application, as follows:

   ```
   public com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument putin;
   ```

   This variable represents the input MQMDHeaders document XMLBean variable for the `putMessage` function.

2. Drag and drop the **Perform** node from the Node Palette into the business process, just below the **Client Request** node.

3. Open the **Perform** node in the Source view and add the following code.

   ```
   public void perform() throws Exception
   {
   putin.getMQMDHeaders().setFormat(MQC.MQFMT_RF_HEADER_2);
   bytmsg = getMQRFH2Header();
   }
   public byte[] getMQRFH2Header() throws Exception    {
   ByteArrayOutputStream bstream = new ByteArrayOutputStream();
   DataOutputStream ostream = new DataOutputStream (bstream);
   String strVariableData =
   "<mcd><Msd>jms_text</Msd></mcd><jms><Dst>someplace</Dst></jms>";
   int iStrucLength = MQC.MQRFH_STRUC_LENGTH_FIXED_2 +
   strVariableData.getBytes().length;
   while(iStrucLength % 4 != 0)
   {
   strVariableData  = strVariableData  + " ";
   iStrucLength = MQC.MQRFH_STRUC_LENGTH_FIXED_2 +
   strVariableData.getBytes().length;
   }
   ```

```
ostream.writeChars(MQC.MQRFH_STRUC_ID);//StrucID
ostream.writeInt(MQC.MQRFH_VERSION_2);//Version
ostream.writeInt(iStrucLength );//StrucLength
ostream.writeInt(273);//Encoding
ostream.writeInt(1208);//CodedCharSetID
ostream.writeChars(MQSTR);//Format
ostream.writeInt(MQC.MQRFH_NO_FLAGS);//Flags
ostream.writeInt(1208);//NameValueCCSID
ostream.writeInt(strVariableData.getBytes().length);//NameValueLength
ostream.writeChars(strVariableData ); //NameValueData
ostream.writeChars("HelloWorld");
ostream.flush();
byte[] bArr = bstream.toByteArray();
return bArr;
}
```

The previous code sets the Format element in the input MQMD Headers document of the `putMessage` function to `MQC.MQFMT_RF_HEADER_2`, represented by the String "`MQHRF2 `".

The `getMQRFH2Header` function writes the fields present in the MQRFH2 header structure to a byte array output stream and returns an array of bytes. The values of the fields given in this example can be modified as required. The actual message can be appended to the end of the byte array and can be Put into the MQSeries queue. The byte array can be provided as the first parameter to the `putMessageAsBytes` function, which is added to the business process file after the Perform node. For more information on the `putMessage` function, see Sending and Receiving Messages.

Setting Dynamic Properties

You can change the MQSeries control properties dynamically at runtime. The MQSeries control properties that you can modify are specified in the MQDynamicProperties document. This document conforms to the MQDynamicProperties schema which is available in the `jpdpublic.jar` file, located at `BEA_Home\wli_10.2\lib`.

To change properties dynamically, perform the following tasks

1. Open the **Client Request** node. In the **General Settings** tab, add a variable of type MQDynamicProperties document.
2. In the **Receive Data** tab, create a new variable for the parameter that you created in the **General Settings** tab of the **Client Request** node by entering a name for the variable. The variable type is already pre-defined based on the parameter to which you are assigning the variable.

3. Drag and drop the **setDynamicProperties** function from the **Controls** tab of the **Data Palette**, into your business process.

4. Open the **Send Data** tab of the **setDynamicProperties** function node. From the **Select variables to assign** drop-down list, assign the variable that you created in the **Receive Data** tab of the **Client Request** node to the corresponding parameter of the **setDynamicProperties** function listed in the **Control Expects** column. All MQSeries Get and Put operations following the `setDynamicProperties` function in the business process use the properties that you specify in the MQDynamicProperties document.

5. While executing your business process at runtime, provide the MQDynamicProperties document as input.

Caution: When you use Explicit Transaction mode, always call the `setDynamicProperties` function before the `Begin` function or after the `Commit` or the `Rollback` functions. If this sequence is not followed, the business process will throw an exception at runtime.

Schema of MQDynamicProperties

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

<xs:schema xmlns="http://www.bea.com/wli/control/MQDynamicProperties"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.bea.com/wli/control/MQDynamicProperties"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="MQDynamicProperties">
<xs:complexType>
<xs:sequence>
<xs:element name="connectionType" type="connType" minOccurs="0"
maxOccurs="1"/>
<xs:element name="queueManager" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="requireAuthorization" type="authType" minOccurs="0"
maxOccurs="1"/>
<xs:element name="host" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="port" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="channel" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="ccsid" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="user" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="password" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="sendExit" type="xs:string" minOccurs="0" maxOccurs="1"/>
```

```
<xs:element name="receiveExit" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="securityExit" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="defaultQueueName" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:element name="implicitTransactionRequired" type="transType"
minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:simpleType name="connType">
<xs:restriction base="xs:string">
<xs:enumeration value="Bindings"></xs:enumeration>
<xs:enumeration value="TCP"></xs:enumeration>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="authType">
<xs:restriction base="xs:string">
<xs:enumeration value="Yes"></xs:enumeration>
<xs:enumeration value="No"></xs:enumeration>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="transType">
<xs:restriction base="xs:string">
<xs:enumeration value="true"></xs:enumeration>
<xs:enumeration value="false"></xs:enumeration>
</xs:restriction>
</xs:simpleType>
</xs:schema>
```

Sample MQDynamicProperties Document

The following is a sample MQDynamicProperties document. You must provide this document at runtime when you execute your business process:

```
<?xml version="1.0"?>
<even:MQDynamicProperties
xmlns:even="http://www.bea.com/wli/control/MQDynamicProperties">
```

```
<even:connectionType>TCP</even:connectionType>
<even:queueManager>newqm</even:queueManager>
<even:requireAuthorization>Yes</even:requireAuthorization>
<even:host>localhost</even:host>
<even:port>1869</even:port>
<even:channel>chn</even:channel>
<even:ccsid>437</even:ccsid>
<even:user>WebLogic</even:user>
<even:password>WebLogic</even:password>
<even:defaultQueueName>errqueue</even:defaultQueueName>
</even:MQDynamicProperties>
```

Configuring SSL In MQSeries Control

SSL functionality is available only if you selected TCP Connection mode while configuring an MQSeries control. For more information on configuration options for the MQSeries control, see Creating and Configuring a New Instance of the MQSeries Control.

This topic includes the following sections:

Setting the SSL Cipher Suite

The cipher suite algorithm is used to encrypt and decrypt message communication between the MQSeries server and the MQSeries client. If you selected either of the two SSL options while creating a new MQSeries control, you must set the SSL cipher suite before you put or get messages from the queue. This can be done using the following function:

```
void setSSLCipherSuite(java.lang.String cipherSuite);
```

The parameter to this function is the string representing the selected SSL cipher suite. You can get the different values for the cipher suites from the final static variables of the MQControlConstants class.

Setting Server-side SSL Properties

After enabling either of the SSL options for your MQSeries control, you must set server-side SSL properties before you put or get messages from the queue. You can do this using the following function:

```
void setServerSideSSL(java.lang.String trustStoreLocation,
java.lang.String trustStoreType, java.lang.String trustStorePassword)
throws com.bea.control.ControlException;
```

The parameters to this function are:

- `trustStoreLocation`

  The path representing the location of the trust store. This parameter is optional, and if not provided, takes the default trust store of the Weblogic JDK.

- `trustStoreType`

  The type of trust store, for example, JavaKeyStore (JKS). This parameter is optional, and if not provided, takes the default trust store type.

- `trustStorePassword`

  The password for the trust store. This parameter is mandatory if the `trustStoreLocation` parameter is provided.

Setting Client-side SSL Properties

After enabling two-way SSL for your MQSeries control, you must set server-side and client-side SSL properties before you put or get messages from the queue. To set the client-side SSL properties, use the following function:

```
void setClientSideSSL(java.lang.String keyStoreLocation, java.lang.String
keyStoreType, java.lang.String keyStorePassword, java.lang.String
keyPassword);
```

The parameters to this function are:

- `keyStoreLocation`

  The path representing the location of the key store. This parameter is mandatory.

- `keyStoreType`

  The type of key store, for example, JKS. This parameter is optional, and if not provided takes the default key store type.

- `keyStorePassword`

The password for the key store. This parameter is mandatory.

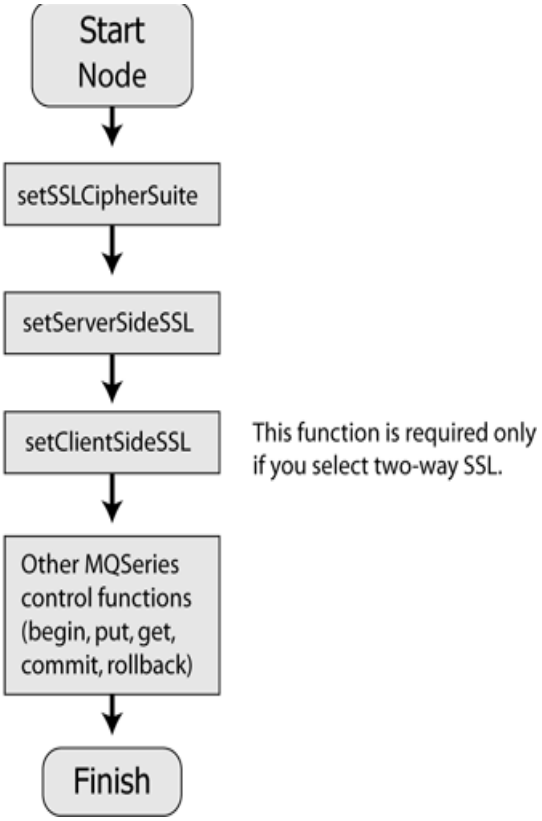
- `keyPassword`

  The password for the key. This parameter is optional, and if not provided, considers the `keyStorePassword` as the `keyPassword`.

Example: Configuring SSL Within a Workflow

After selecting either of the two SSL options while creating a new MQSeries control, your workflow must adhere to the order of MQSeries control function calls represented in Figure 9-8.

Figure 9-8 Example: Configuring SSL Within a Workflow



WARNING: If the sequence represented in Figure 9-8 is not followed in the workflow when SSL authentication is required, the MQSeries Control will throw an exception at runtime.

For information on how to set up the Queue Manager for SSL connections, refer to the MQSeries Product documentation at http://www.IBM.com. SSL support is available only from WebSphere MQ version 5.3 onwards.

Using the MQSeries Event Generator

The MQSeries event generator polls the MQSeries queue for messages and publishes them to WebLogic Message Broker channels. The MQSeries event generator supports three different data types — Bytes, String, and XML.

You can configure event generator channels for different data types using a Message Broker channel name, which instructs that any message coming into the specified MQSeries queue will be published to that message broker channel.

Similar to the MQSeries control, the MQSeries event generator also provides two modes of connections — TCP-IP and Bindings. You can also implement content-filters to filter messages based on the specific content that you want. By doing this, you can ensure that you generate events only for the messages that you require.

The MQSeries event generator can also spawn multiple threads of events. Each thread can separately poll the MQSeries queue. You can configure the number of messages to be picked by the event generator thread in each poll.

To learn more, see Event Generators.

Process Control

Note: The Process control is only available in BEA WorkSpace Studio if you are licensed to use WebLogic Integration.

The Process control is used to send requests to and receive callbacks from another business process. The Process control is typically used to call a subprocess (child process) from a parent process.

For information on how to add control instances to business processes, see Using Controls in Business Processes.

Topics Included in This Section

Overview: Process Control
Describes the Process control

Creating a New Process Control
Describes how to create and configure a new Process control.

Process Control Methods
Introduces Process control methods and details how you set Process Control properties.

Process Control Design Time Considerations
Outlines some issues you should consider when designing your Process control.

Process Control Run-Time Considerations
Details some issues that affect Process control operation at run time.

Maintaining Process Controls
Outlines some Process control maintenance issues.

Using Dynamic Binding
Describes how to customize a Process control.

Notes on Process Controls Annotations
Details some annotations that have specific rules that you should follow when you use them with Process controls.

Overview: Process Control

The Process control allows a Web service, business process, or pageflow to send requests to, and receive callbacks from, a business process. Process control invocations are Java Remote Method Invocation (RMI) calls.

The target business process must be hosted on the same WebLogic Server domain as the caller. The Process control is typically used to call a subprocess (child process) from a parent business process.

The first step in using a Process control is creating a Java file. The control java file can be automatically generated from a target business process using BEA WorkSpace Studio, or can be created using the control wizard. The methods and callbacks on the control java file correspond to the operations and callbacks of the target business process. An instance of this java file is used by a parent process to call the target process. Process control java files can have selector annotations only on start methods or, for stateless target services, on any method.

To learn about creating a Process control, see Creating a New Process Control.

Creating a New Process Control

This topic describes how to create a new Process control.

You can create a Process control in two different ways, which are described in the following sections:

- Creating a New Process Control Using the Control Wizard
- Generating a Process Control from a Process File

Creating a New Process Control Using the Control Wizard

You can use the **Insert Process** dialog to create a new Process control and add it to your business process. If you are not in the Design view, click the **Design** tab.

To create a new Process control:

1. In the **Package Explorer** pane, double-click the business process to which you want to add the Process control. The business process is displayed in the **Design** view.

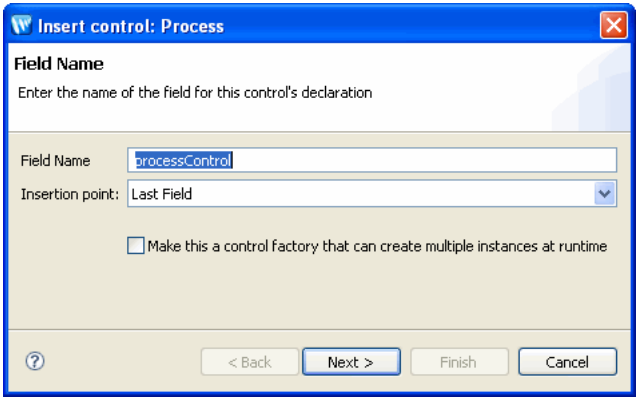
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **Process**.

   The **Insert control: Process** dialog box appears (see Figure 10-1).

Figure 10-1 Insert control: Process



4. In the **Insert control: Process** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new Process control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point:** from the drop-down list select the point where you want the field name to be inserted in the process file.

– Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

– Click **Next**.

The **Create Control** wizard appears.

5. In the **Create Control** wizard enter the following details:

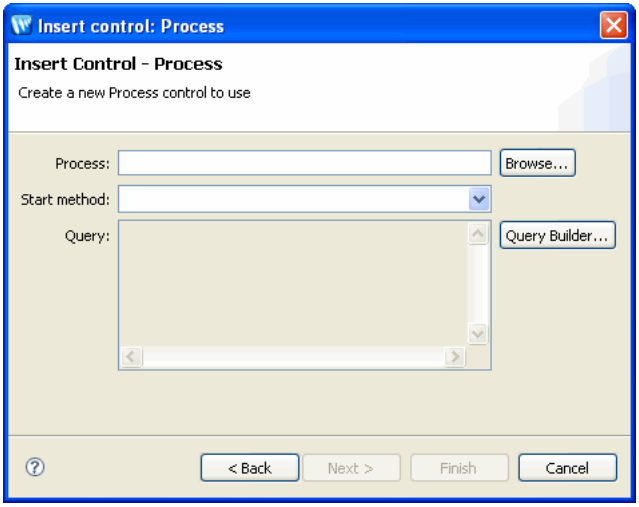
– In the **Name** field, type the name of your new control extension file.

– Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

– Click **Next**.

6. In the **Insert control: Process** dialog box enter the following details (see Figure 10-2).

Figure 10-2 Insert Control - Process



– In the **Process** field, select the business process you want to access by selecting the name of a business process file.Click **Browse**, choose from the available list and click **OK**.

– Select a start method from the **Start Method** menu. Only those start methods contained in the specified business process are displayed.

– To specify a dynamic selector, enter a query in the **Query** field or click the **Query Builder** button to display the **Dynamic Selector** query builder.

If you invoked the **Dynamic Selector** query builder, perform the following steps to build and test a query:

- Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose `TPM` to bind lookup values to properties in the TPM repository. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain-wide `DynamicProperties.xml` file. You should only edit the `DynamicProperties.xml` file to bind lookup values to dynamic properties if the domain is inactive. If the domain is active, it is recommended that you use the WebLogic Integration Administration Console to bind lookup values.

For more information on binding lookup values to dynamic properties using the WebLogic Integration Administration Console, see "Adding or Changing Dynamic Control Selectors" in Process Configuration.

- In the **Start Method Schema** area, select an element from the schema to associate it with the start method of the control. Only XML elements are displayed; non-XML elements are not supported. The resulting query appears in the **XQuery** area.
- Click **Create**.

– Click **Finish**.

The Process control is created and displayed in the Package Explorer pane. An instance of the control is also created and is added to the Data Palette.

For more information, see "Step 6: Invoke a Business Process Using a Process Control" in Tutorial: Building Your First Business Process.

Generating a Process Control from a Process File

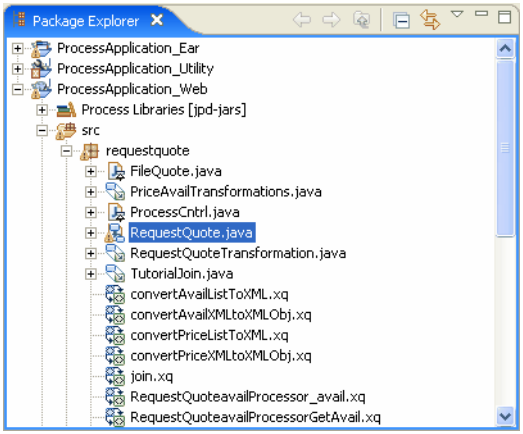
You can create a new Process control from an existing Process.java file.

Note: If the **Package Explorer** pane is not visible in BEA WorkSpace Studio, click **Window > Show View > Other > Java > Package Explorer** from the menu bar.

To generate a Process control from a Process file:

1. Open the application that contains the business process for which you want to create the Process control.
2. The **Package Explorer** pane displays the Process.java file for the process (see Figure 10-3).

Figure 10-3 Process.java file



3. Right-click on the Process file.

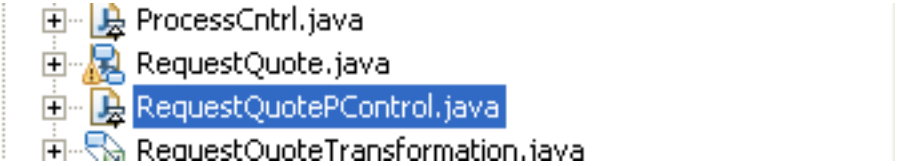
   A menu appears displaying a list of options.

4. Select **Generate > Process Control**.

   The **Save As** dialog box appears.

5. Enter the required details and click **OK**.

6. The Process control is created and displayed below the .java file in the Package Explorer pane. The name is generated by appending PControl to the Process.java name. For example, if you generate a Process control.java file from `RequestQuote.java`, the resulting java file is named `RequestQuotePControl.java`.

   Double-click the Process control java file in the Package Explorer pane to display the control in Design view.

Alternatively, you may create a Process control file manually. For example, you may copy an existing Process control file and modify the copy.

Notes on XQueries

When you are using XQuery expressions and the XQuery Builder, it is important to remember:

- If you create a new Process control and want to use the XQuery Builder to associate an element in the schema with the start method for the new control, remember that XQuery Builder will only declare namespaces for schemas used in argument 2. So, if argument 1 uses "schema1", but argument 2 does not use "schema1", it will not declare namespace "schema1".
- If you use the XQuery Builder to generate XQuery expressions that involve an optional element, it is possible that you will have to manually edit the expression to generate a correct XPath location
- Some XQuery expressions are not supported when using complex Predicates.

Process Control Methods

To learn about the methods available on a Process control, see the Interface ProcessControl.

Example: Process Control Declaration

When you create a new Process control using the control wizard and drag a method from the control onto a business process, its declaration appears in the process.java file. The following code snippet is an example of what the declaration looks like:

```
@org.apache.beehive.controls.api.bean.Control

       private proc2Control proc2Control1;
```

Setting Process Control Properties

The Process control possesses the capability of dynamically binding some properties of the control. Dynamic binding of properties can be achieved:

- Using selectors
- Using the `setProperties()` API
- Using setter methods for individual properties, such as `setTargetURI()`.

To retrieve the current property settings, except for username and password, use the `getProperties()` method.

Starting with the method with the highest precedence, the hierarchy of property settings is:

1. Properties dynamically bound using the `com.bea.wli.common.control.Selector` tag and the `DynamicProperties.xml` file
2. Properties set using the `setProperties()` method or other setter methods inherited from the Process control (`setConversationID`, `setTargetURI`, `setPassword`, and `setUsername`)
3. Properties set using static annotations

The `ProcessControlProperties` type is an XML Beans class that is generated out of the corresponding schema element defined in `DynamicProperties.xsd`. The `DynamicProperties.xsd` file is located in the schemas/system folder or in the utility folder of new Process Applications.

The `setProperties()` method uses this XML Bean class to set properties on a control instance. A selector on a Process control method returns an XML document that conforms to the `ProcessControlProperties` element. The following sample shows how to programmatically set the username property for a control. You add the bold code lines to the code generated when the control is created, overriding properties set using dynamic binding and static annotations:

```
import com.bea.wli.control.dynamicProperties.
ProcessControlPropertiesDocument;

import com.bea.wli.control.dynamicProperties.
ProcessControlPropertiesDocument.ProcessControlProperties;

    ProcessControlPropertiesDocument props= null;
    ProcessControlProperties sprops = null;

    public void sBC8InvokeSetProperties() throws Exception

    {

        props = ProcessControlPropertiesDocument.Factory.newInstance();
        sprops = props.addNewProcessControlProperties();

        sprops.setUsername("smith");
```

You construct the `ProcessControlPropertiesDocument` instance in the above section of code through a Data Transformation method or through XMLBean APIs. For more information on Data Transformations, see Guide to Data Transformation.

The following code provides a sample of a `ProcessControlPropertiesDocument` instance:

```
this.pcp = ProcessControlPropertiesDocument.Factory.newInstance();

this.pcp.addNewProcessControlProperties();
```

```
this.pcp.getProcessControlProperties().setUsername("uname");
this.pcp.getProcessControlProperties().setPassword("pword");

this.pcp.getProcessControlProperties().setTargetURI("http://localhost:7001
/";);
```

Some control properties can be specified dynamically or in annotations (statically) on the control file. For example, the Process control allows you to specify the target process in the `@com.bea.wli.common.control.Location(uri = <URL>)`annotation at the top of the java control file or dynamically using the `TargetURI` element in `DynamicProperties.xml`. In all such cases, a dynamically bound value for the property takes precedence over the static annotation.

If the domain is active, it is recommended that you use the WebLogic Integration Administration Console to perform dynamic binding. For more information on binding lookup values to dynamic properties using the WebLogic Integration Administration Console, see "Adding or Changing Dynamic Control Selectors" in Process Configuration.

Dynamic properties can also be specified by calling `setProperties` on the control, or by calling one of the setter methods, such as `ProcessControl.setUsername()`.

Properties applied using selectors remained bound until one of the following conditions occurs:

- A method marked `finish` on the java control file is invoked
- A start method is invoked again
- The property is programmatically set by calling `setProperties` or a setter method.

`ProcessControl.reset()` resets all configured selector values.

Process Control Design Time Considerations

This section outlines some of the issues you must consider when you are designing a process control for your business process. The issues covered in this section are:

- Using a Process Control in Stateless and Stateful Business Processes
- Using Process Controls in Synchronous and Asynchronous Business Processes
- Using a Process Control from the Parent Process
- Process Control Location

Using a Process Control in Stateless and Stateful Business Processes

The number of transactions contained in a business process determines whether the process is stateless or stateful. For more information, see Building Stateless and Stateful Processes. This section covers some of the issues you should consider when designing a Process control for stateless and stateful business processes.

When designing your Process control, adhere to the following rules:

- Whether you are designing a stateless or stateful business process, the Process control can invoke any request or response node. However, for stateful processes, the first request must be a start request.
- The start method that invokes the call governs which continue methods may be called during the call.
- The order in which the methods on the control are called matters in a stateful process.
- Process controls that refer to stateful processes only refer to one process at a time.

Using Process Controls in Synchronous and Asynchronous Business Processes

Business processes can have both synchronous or asynchronous request methods. For more information, see Building Synchronous and Asynchronous Business Processes. This section covers some of the issues you should consider when designing a Process control for synchronous and asynchronous business processes.

When designing your Process control, remember:

- In a synchronous invocation, the input and response arguments are combined into a single operation.
- In an asynchronous invocation, one-to-one mapping takes place. Any response node becomes a method within the callback interface.

Using a Process Control from the Parent Process

The Process control is typically used to call a subprocess from a parent business process. When the Process control is invoked from the parent process, the control is invoked normally if the subprocess is in the same application.

If the subprocess control is in a different application, you must copy the subprocess control into the parent process control and change the location of the subprocess control so that it is the same as the location of the parent process control, if you want the call to invoke normally.

Process Control Location

When you create a Process control, it is displayed in the Package Explorer pane and an instance of the control is added to the Data Palette. The location of the Process control is displayed in the JPD Configuration pane.

Note: If the **JPD Configuration** pane is not visible in BEA WorkSpace Studio, click **Window > Show View > JPD Configuration** from the menu bar.

To view the location of the Process control:

1. Double-click the Process control file in the Package Explorer pane.

   The Process control is displayed in the Design view and the JPD Configuration pane displays the properties of the Process control.

   The location of the Process control is displayed in the **location** section of the JPD Configuration, in the **uri** field. The location is not an actual HTTP address, though it may appear that it is. The uri actually displays the location of the object within the Java Naming and Directory Interface (JNDI) tree.

Process Control Run-Time Considerations

This section outlines some of the run-time issues you must consider when you are invoking a Process control in your business process. When you are invoking a Process control from a parent business process, you are making a Java Remote Method Invocation (RMI) to the subprocess. The transfer of information is governed by RMI rules and not by serialization rules.

The topics covered in this section are:

- Run-Time Rules for Process Controls in Stateless and Stateful Business Processes
- Run-Time Rules for Process Controls in Synchronous and Asynchronous Business Processes
- Security

Run-Time Rules for Process Controls in Stateless and Stateful Business Processes

The number of transactions contained in a business process determines whether the process is stateless or stateful. For more information, see Building Stateless and Stateful Processes. This section covers some of the run-time issues you should consider when using a Process control in stateless and stateful business processes.

The following rules govern the run-time operation of Process controls in stateless and stateful business processes:

- In a stateless business process, the invocation from the parent business process generically picks any stateless subprocess and can call any method from the Process control on the parent business process.
- In a stateful business process the first instantiation has to be a start method.
- In a stateful business process, the subprocess selected depends on the method specified by the Process control. A new process instance is created based on the method specified by the Process control and a new conversation ID is assigned to the process instance. You can specify the conversation ID. If you don't specify a conversation ID, the instance automatically generates its own conversation ID. For more information on conversations, see Designing Conversational Web Services. For more information on specifying the conversation ID, see "Interface ProcessControl".
- In a stateful business process, once a call is invoked and a conversation ID generated, any further continue methods that are invoked, interact with the original conversation ID. You need to specify a new start method to invoke an interaction with a new conversation ID.

Run-Time Rules for Process Controls in Synchronous and Asynchronous Business Processes

Business processes can have both synchronous or asynchronous request methods. For more information, see Building Synchronous and Asynchronous Business Processes. This section covers some of the run-time issues you should consider when using a Process control in synchronous and asynchronous business processes.

The following rules govern the run-time operation of Process controls in synchronous and asynchronous business processes:

- A synchronous subprocess called through a Process control runs in the same transaction as its caller (parent) process. Synchronous subprocesses behave differently than asynchronous subprocesses, particularly when it comes to unhandled exceptions.
- When an error occurs in synchronous business processes, exceptions are raised. An unhandled exception in a subprocess causes the shared transaction to be marked as rollback only, which causes the subprocess and parent processes to roll back. This is the default behavior because it prevents a scenario in which one of the processes is rolled back, leaving the other process in an inconsistent or uncompensated state. You can override this default behavior by changing the value of the **on sync failure** property specified in the JPD Configuration pane of the subprocess. The available options are **rollback** and **rethrow**. The **rethrow** option returns an error to the parent process when an exception occurs. To override the default behavior, change the **on sync failure** property from **rollback** to **rethrow**.
- In asynchronous business processes, the transaction is never propagated to the subprocess. In other words, the subprocess runs in its own transaction. The parent business process uses Java Message Service (JMS) to drop the messages into a JMS queue. Once the parent process drops the message into the queue, it assumes that message delivery is successful. If an error occurs on the subprocess, the parent process is not informed of the error. For information on enhancing your business process so that you can detect failures, see "Asynchronous Subprocesses" in Building Synchronous and Asynchronous Business Processes.
- In asynchronous business processes, the parent business process continues to execute until the process finishes or until the process receives a callback from the subprocess. The callback is always associated with the process instance initially created by the parent business process.
- In a synchronous business process, once the parent process invokes a message request to the subprocess, through the Process control, it finishes.
- In an asynchronous business process, once the parent process invokes a message request to the subprocess, through the Process control, it continues to instantiate and make further requests to the subprocess. If the subprocess does not have time to respond to the request from the parent process before it receives a second request, the second request will be dropped and the subprocess won't receive it at all. For this reason, you should design the parent process so that is blocks and waits for a callback from the subprocess before taking further action. For an example of using an Event Choice node to perform this function, see "Asynchronous Subprocesses" in Building Synchronous and Asynchronous Business Processes.

- The `com.bea.control JwsContext` interface provides access to container services that support Web services (JWS files). For more information, see “com.bea.control JwsContext Interface” in com.bea.control Package. The `setCallbackLocation()` method is not supported for Process control callbacks.

Security

The Process control conforms to all standard security checks associated with Java Remote Method Invocation (RMI). SSL is not supported. For more security information, see Security.

Maintaining Process Controls

In general, Process controls are utilized in situations where the parent process and subprocess are in the same application, or in situations where the parent process and subprocess are in different applications.

In both situations, if the Process control is changed on the subprocess side of the transaction, you must regenerate the control on the parent side to insure that the control will operate normally. If the subprocess and parent process are in different applications, and the Process control is changed in any way (i.e. request/response removed, etc.), backward compatibility may be compromised. If this happens, you must merge the Process control on the subprocess side and repropagate the control back to the parent business process.

JAX-RPC Handlers

JPDs implicitly inherit all behavior from JWSs. This allows you to add JAX-RPC handlers to a JPD to "intercept" messages sent to requests and responses before and/or after the request.

However, these handlers are not executed for JPD requests that come in via a Process control execution. In other words, handlers are not executed for requests to/from Process control invocations.

Using Dynamic Binding

In many cases, control attributes are statically defined using annotations. Some controls provide a Java API to dynamically change certain attributes. Dynamic controls, including the Service Broker and Process controls, provide the means to dynamically set control attributes. Attributes are determined at runtime using a combination of lookup rules and lookup values, a process called *dynamic binding*. Controls that support dynamic binding are called *dynamic controls*. The business process developer specifies lookup rules using WebLogic Workshop while the

administrator specifies look-up values using the WebLogic Integration Administration Console. This powerful feature means that control attributes can be completely decoupled from the application and can be reconfigured for a running application, without redeployment.

To learn more about dynamic binding, see How the Service Broker Control Uses Dynamic Binding.

Notes on Process Controls Annotations

This section outlines some rules that you should adhere to when using the following Process control annotations:

- `@com.bea.control.annotations.MessageBuffer`
- `@com.bea.wli.common.Conversation`

@com.bea.control.annotations.MessageBuffer

The `@com.bea.control.annotations.MessageBuffer` annotation specifies that there should be a queue between the component's implementation code and the message transport wire for the specified method or callback. For more information, see "@MessageBuffer Annotation" in Java Control Annotations.

@com.bea.wli.common.Conversation

The `@com.bea.wli.common.Conversation` annotation specifies the role that a control's methods or callbacks play in a conversation. For more information, see "@com.bea.wli.common.Conversation annotation" in Java Control Annotations.

RosettaNet Control

Note: The RosettaNet control is available in BEA Workshop for WebLogic Platform only if you are licensed to use WebLogic Integration.

RosettaNet is a consortium of major companies working to create and implement industry-wide, open e-business process standards. These standards form a common e-business language, aligning processes between supply chain partners on a global basis. RosettaNet is a subsidiary of the Uniform Code Council, Inc. (UCC). To learn about RosettaNet, see `http://www.rosettanet.org`.

The RosettaNet control enables BEA Workshop for WebLogic Platform business processes to exchange business messages and data with trading partners via RosettaNet. You use RosettaNet controls only in initiator business processes to manage the exchange of RosettaNet business messages with participants. For an introduction to RosettaNet solutions, see Introducing Trading Partner Integration.

Topics Included in This Section

Using a RosettaNet Control
Describes how to use a RosettaNet control in a business process.

Example: RosettaNet Control
Provides links to examples of how to use the RosettaNet control.

Related Topics

Introducing Trading Partner Integration

Trading Partner Management

Interface RosettaNet Control

Building RosettaNet Participant Business Processes

Overview: RosettaNet Control

You use RosettaNet controls in *initiator* business processes to exchange RosettaNet business messages with participants. The RosettaNet control provides methods for sending and receiving business messages, as described in the Interface RosettaNetControl Javadoc. Callbacks handle RosettaNet messages, acknowledgements, rejections, and errors received from the participant.

You should *not* use RosettaNet controls in participant business processes to respond to incoming messages. Instead, you use client request nodes to handle incoming business messages from the initiator and client response nodes to handle outgoing business messages to the initiator. To learn about building participant business processes that use RosettaNet, see Building RosettaNet Participant Business Processes. To learn about designing business processes that use RosettaNet, see Introducing Trading Partner Integration .

At run-time, the RosettaNet control relies on trading partner and service information stored in the TPM repository. To learn about the TPM repository, see Introducing Trading Partner Integration. To learn about adding or updating information in the TPM repository, see Trading Partner Management in *Using The WebLogic Integration Administration Console*.

Creating a RosettaNet Control

This topic describes how to create a new RosettaNet control. You add one RosettaNet control per public initiator business process. To learn more about public vs. private processes see, “Types of Business Processes” in “Trading Partner Business Process Concepts” in Introducing Trading Partner Integration. To learn about RosettaNet controls, see RosettaNet Control.

To create a new RosettaNet control

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the **Rosettanet** control. The business process is displayed in the **Design** view.

2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA Workshop for WebLogic Platform, click **Window > Show View > Data Palette** from the menu bar.

3. Select **Rosettanet**.

   The **Insert control:Rosettanet** dialog box appears.

4. In the **Insert control:Rosettanet** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new Rossatanet control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.

   – Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

   – Click Next.

   The **Create Control** dialog-box appears.

5. In the **Create Control** dialog box enter the following details:

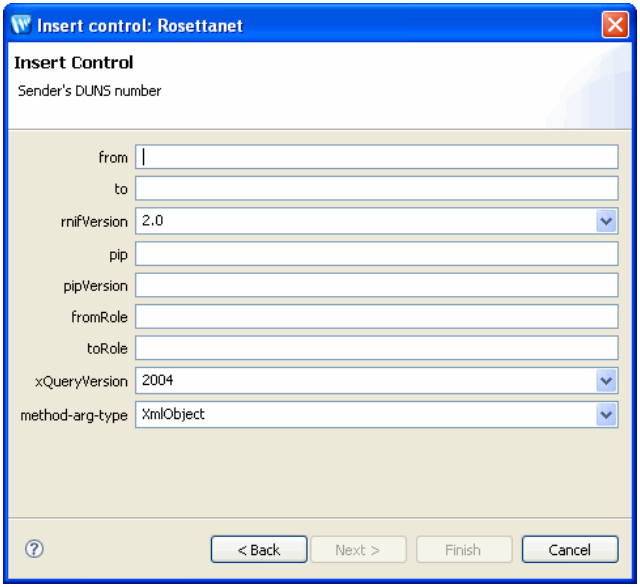
   – In the **Name** field, type the name of your new control extension file.

   – Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

   – Click **Next**.

   The **Insert control: Rosettanet** dialog-box appears (see Figure 11-1).

Figure 11-1 Insert Control: Rosettanet



6. In the **Insert control: Rosettanet** dialog box, specify the following information (see Table 11-1).

   Note: Where applicable, the values entered here must match their corresponding settings in the TPM repository.

Table 11-1 Rosettanet Fields

Field	Description
`from`	Sender's DUNS number. Must be defined in the TPM repository.
`to`	Recipient's DUNS number. Must be defined in the TPM repository.
`rnif-version`	Version of the RNIF (RosettaNet Implementation Framework). One of the following values: • 1.1 • 2.0
`pip`	RosettaNet PIP code, such as `3B2`. Must be a valid PIP code as defined in `http://www.rosettanet.org`.

Table 11-1 Rosettanet Fields

`pip-version`	RosettaNet PIP version. Must be a valid version number associated with the PIP.
`from-role`	RosettaNet role name for the sender as defined in the PIP specification, such as Buyer, Initiator, Shipper, and so on. A PIP request might be rejected if an incorrect value is specified.
`to-role`	RosettaNet role name for the recipient as defined in the PIP specification, such as Seller, Participant, Receiver, and so on. A PIP request might be rejected if an incorrect value is specified.
`method-arg-type`	Required. Type of attachment. Includes the standard RNIF XML parts. One of the following values: • `XmlObject`—Default. Represents data in untyped XML format. The XML data is not specified at design time. • `RawData`—Represents any non-XML structured or unstructured data and for which no MFL file (and therefore no known schema) exists. Not recommended, as the payload includes standard RNIF XML parts. • `MessageAttachment[]`—Array containing one or more parts of a business message. Message parts can be untyped XML data (`XmlObject` data type) or non-XML data (`RawData` data type). Used when sending different kinds of payloads (XML and non-XML) in the same message. The actual number of message parts might not be known until processed. To learn about working with `MessageAttachment` objects, see Using Message Attachments. To learn more about data types, see Working with Data Types.

7. Click **Finish**.

   A RosettaNet control instance is displayed in the **Controls** tab.

Using a RosettaNet Control

All WebLogic Workshop controls follow a consistent model. Many aspects of using RosettaNet controls are identical or similar to using other WebLogic Workshop controls.

After you have added a RosettaNet control to an initiator business process, you can use methods on the control to exchange RosettaNet messages with participant trading partners. In the Design

View, you expand the node for the RosettaNet control in the Data Palette to expose its methods, and then drag and drop the methods you want onto the business process. Common tasks include:

- Sending Messages to Participants
- Handling Messages from Participants
- Retrieving Message Elements
- Dynamically Specifying Business IDs

To learn more about these methods, see Interface RosettaNetControl.

Sending Messages to Participants

The RosettaNet control provides methods for sending the initial request message to a participant and also for responding to the participant's reply. To add the method to a business process, you drag it from the Data Palette onto the business process, which creates a **Control Send** node.

Sending a Request Message

You use the `sendMessage` method to send a RosettaNet request message to participants. After creating the **Control Send** node in the business process, you need to specify the payload parts and their Java data types. Valid data types include:

Table 11-2 Data Type

Type	Description
`XmlObject`	Data in untyped XML format.
`RawData`	Any non-XML structured or unstructured data for which no MFL file (and therefore no known schema) exists.
`MessageAttachment`	Data in both untyped XML and non-XML format. To learn about working with `MessageAttachment` objects, see Using Message Attachments.

Note: Attachments can also be typed XML or typed MFL data as long as you specify the corresponding XML Bean or MFL class name in the parameter.

Responding to Participant Replies

After sending a RosettaNet message, the initiator business process awaits a response from the participant. After receiving the participant's response to the request, a business process can either acknowledge and accept the response, reject the response, or notify the participant that an error has occurred. The RosettaNet control provides the following methods for responding to participant replies:

Table 11-3 Method Type

Method Name	Description
`sendAck`	Sends a RosettaNet acknowledgement of receipt to the participant.
`sendError`	Sends a RosettaNet error to the participant.
`sendReject`	Sends a RosettaNet rejection to the participant.

Handling Messages from Participants

Participants can respond to initiator requests in the following ways:

- acknowledge that the request was received
- reply to the request
- notify that an error has occurred

To handle responses from participants, initiator business processes use the following callback methods:

Table 11-4 Callback Methods

Method Name	Description
`onAck`	Handles the acknowledgement of the message receipt from the participant.
`onError`	Handles an error sent by the participant.
`onMessage`	Handles the message reply sent by the participant.

To receive a RosettaNet message from a participant, you use the appropriate method. To add the method to a business process, you drag it from the Data Palette onto the business process, which creates a **Control Receive** node.

For the `onMessage` method, after creating the **Control Receive** node, you need to specify the payload parts and their Java data types for the incoming message. To learn about valid data types, see Sending Messages to Participants.

The `onError` and `onAck` methods are system-level methods. Both use the `XmlObject` argument, which will contain a RosettaNet payload. These arguments are not seen in the default control but you can drag them onto the business process from the Data Palette. If your application contains a schema project that includes the Exception schema file (for RNIF2.0), and if the schema is already built, you can extract the values you want by creating a query (in the XQuery language) using the mapper functionality of WebLogic Workshop. To learn about creating queries with the mapper functionality, see Transforming Data Using XQuery.

Retrieving Message Elements

You can retrieve specific message elements from your RosettaNet messages by using the `RosettaNetContext` XMLBean. The following message elements can be retrieved and are returned as `java.lang.string`:

Table 11-5 Message Elements

Element Name	Description
`from`	Sender's DUNS number.
`to`	Recipient's DUNS number.
`pip`	RosettaNet PIP code specified for the message.
`pip-version`	PIP version specified for the message.
`from-role`	RosettaNet role name for the sender as defined in the PIP specification. Examples include: Buyer, Initiator, Shipper, and so on.
`to-role`	RosettaNet role name for the recipient as defined in the PIP specification. Examples include: Seller, Participant, Receiver, and so on.

Table 11-5 Message Elements

Element Name	Description
`failure-report-administrator`	Trading partner id of the trading partner which is specified to be the failure administrator. (In WebLogic Integration, this is specified in the sender trading partner's binding).
`global-usage-code`	Indicates whether the message was sent in test or production mode.
`debug-mode`	Returns `true` if the message was sent in debug mode.
`message-tracking-id`	Instance id of the action to which this message is in reply.
`protocol-name`	Name of the protocol used.
`protocol-version`	Version of the protocol used.
`conversation-id`	Id of the conversation.
`process-instance-id`	Instance id of the receiving process.
`process-uri`	URI of the receiving process.
business-action	The business action of the message, such as: Purchase Order Request, Purchase Order Confirmation, etc.
document-datetimestamp	The time and date the document was created.
proprietary-identifier	A unique number which tracks the document.

When you use the `RosettaNetContext` XMLBean, be sure to import the following classes:

```
com.bea.wli.control.rosettanetContext.RosettaNetContextDocument;
com.bea.wli.control.rosettanetContext.RosettaNetContextDocument.RosettaNet
Context;
```

The following are code examples of how to use `RosettaNetContext`:

Note: If you use the code samples provided in this section, remember to also modify the return type of your corresponding methods in your RosettaNet control definition file (control

file). In other words, `public void sendMessage()` needs to be changed to `public RosettaNetContextDocument sendMessage()`.

- *Initiator business process receiving a message*:

```
public void rn_onMessage(RosettaNetContextDocument doc,
                         XmlObject msg)
   {
     System.out.println(">>>>> ContextInitiator.rn_onMessage()");
     RosettaNetContextDocument.RosettaNetContext context =
        doc.getRosettaNetContext();
     System.out.println("    from=" + context.getFrom());
     System.out.println("    to=" + context.getTo());
     System.out.println("    pip=" + context.getPip());
     System.out.println("    failure-report-admin=" +
        context.getFailureReportAdministrator());
   }
```

- *Initiator business process sending a message:*

```
public void rnSendMessage() throws Exception
   {
     String rnInfo = "Service Content";
     XmlObject xObj = XmlObject.Factory.parse(rnInfo);
     RosettaNetContextDocument doc = rn.sendMessage(xObj);
     System.out.println(doc.toString());
   }
```

Where *`Service Content`* is the service content of your RosettaNet message.

- *Participant business process receiving a message:*

```
public void onMessage(RosettaNetContextDocument doc, XmlObject msg)
   {
     System.out.println(">>>>> ContextParticipant.onMessage()");
     RosettaNetContext context = doc.getRosettaNetContext();
     System.out.println("    from=" + context.getFrom());
     System.out.println("    to=" + context.getTo());
     System.out.println("    pip=" + context.getPip());
     System.out.println("    failure-report-admin=" +
        context.getFailureReportAdministrator());
   }
```

- *Participant business process interface for callbacks:*

```
@CallbackInterface()
public interface Callback extends ServiceBrokerControl
{
/**
* com.bea.control.annotations.MessageBuffer enable="false"
*/
@com.bea.wli.common.MessageBuffer(enable = false)
public RosettaNetContextDocument sendReply(XmlObject msg);
/**
* com.bea.control.annotations.MessageBuffer enable="false"
*/
@com.bea.wli.common.MessageBuffer(enable = false)
public void sendReceiptAcknowledgement();
/**
* com.bea.control.annotations.MessageBuffer enable="false"
*/
@com.bea.wli.common.MessageBuffer(enable = false)
public void sendError(String msg);
}
@com.bea.wli.jpd.Callback()
public Callback callback;
```

- *Participant business process sending a reply:*

```
public void reply()
   {
     XmlObject xObj = null;
     try {
       xObj = XmlObject.Factory.parse("Service Content");
     } catch (Exception e) {
```

```
        e.printStackTrace();
      }

      RosettaNetContextDocument doc= callback.sendReply(xObj);
      System.out.println(doc.toString());
    }
```

Where *`Service Content`* is the service content of your RosettaNet message.

Dynamically Specifying Business IDs

The RosettaNet control adds the capability of dynamically binding business IDs for the initiator (`from` property) and the participant (`to` property) of the control. Dynamic binding of properties can be achieved the following ways:

- Using selectors
- Using the `setProperties()` method

Order of Precedence

The hierarchy of property settings is as follows, starting with the approach having the highest precedence:

1. properties dynamically bound using selectors (RosettaNetControl Annotation) and the `DynamicProperties.xml` file
2. properties set using the `setProperties()` method
3. properties set at the control instance level using the RosettaNet Annotation in the Process.java file.
4. properties set at Control file class level using RosettaNet Annotation in the Control file.

Dynamic selectors have a higher precedence than static selectors.

Using Selectors

Using a dynamic selector, RosettaNet controls allow you to decide at run time which one of multiple trading partners to send a business message to. When you specify a dynamic selector, you build and test an XQuery that retrieves the business ID you need.

To use a dynamic selector

1. Display the business process in Design view that contains the RosettaNet control for which you want to specify a dynamic selector.
2. In Design view, select the RosettaNet control node in the Data Palette.
3. Locate the **from-selector** or **to-selector** property in the Properties pane and select the associated **xquery** parameter. Click the button next to the **xquery** field indicated by three dots (**...**). The **Dynamic Selector** query builder is displayed.
4. In the **Start Method Schema** area, select an element from the schema to associate it with the start method of the control. The resulting query appears in the **XQuery** area.
5. Click **OK**.

Using setProperties

The `setProperties` method accepts a `RosettaNetPropertiesDocument` parameter. The `RosettaNetPropertiesDocument` type is an XML Beans class that is generated out of the corresponding schema element defined in `DynamicProperties.xsd`. The `DynamicProperties.xsd` file is located in the system folder of New Process Applications or in the system folder of the Schemas project.

If your application contains a schema project that includes the `DynamicProperties.xsd` file, and if the schema is already built, you can extract the values you want by creating a query (in the XQuery language) using the mapper functionality of WebLogic Workshop. To learn about creating queries with the mapper functionality, see Transforming Data Using XQuery.

To set business IDs dynamically using the setProperties method

1. Verify that your application contains a schema project that includes the `DynamicProperties.xsd` file, and that the schema is already built.
2. Create a **Control Send** node in a business process.
3. From the **Data Palette**, drag the `setProperties` method and drop it onto the **Control Send** node.
4. In the **Send Data** tab, select **Transformation**, specify variables that contain the `to` and `from` values, and then create a transformation to map them to the corresponding elements in `RosettaNetPropertiesDocument`.

To display the current property settings, use the `getProperties()` method.

Example: RosettaNet Control

For examples of how to use the RosettaNet control, see Tutorials: Building RosettaNet Solutions, which is located in the following directory:

`http://edocs.bea.com/wli/docs102/tptutorial/rosettanet.html`

Service Broker Control

Note: The Service Broker control is only available in BEA WorkSpace Studio if you are licensed to use WebLogic Integration.

The Service Broker control allows a business process to send requests to and receive callbacks from another business process, a Web service, or a Web service or business process defined in a WSDL file.

The Service Broker control lets you dynamically set control attributes. This allows you to reconfigure control attributes without having to redeploy the application.

For information on how to add control instances to business processes, see Using Controls in Business Processes.

Topics Included in This Section

Overview: Service Broker Control

Describes the purpose of the Service Broker control.

Creating a New Service Broker Control

Describes how to create a new Service Broker control using the control wizard or by automatically generating the control from a business process or Web service.

Overview: Service Broker Control

The Service Broker control allows a business process to send requests to and receive callbacks from another business process, a Web service, or a remote Web service or business process. The Service Broker control is an extension of the Web Service control.

A remote Web service or business process is accessed using Web services and is described in a Web Services Description Language (WSDL) file. A WSDL file describes the methods and callbacks that a Web service implements, including method names, parameters, and return types. You can generate a WSDL file for any business process by right clicking on a Process.java file in the Package Explorer pane and choosing **Generate WSDL File**.

The first step in using a Service Broker control is creating a Control file. The Control can be automatically generated from a target service (Web service, business process, or WSDL file) using BEA WorkSpace Studio, or can be created using the **Add** function in the **Controls** section of the BEA WorkSpace Studio **Data** palette. The methods and callbacks on the Control file correspond to operations and callbacks of the target service. An instance of this Control file is used by a parent service to call the target service and can also be used to get callbacks from the target service. Service Broker control files can only have selector annotations on start methods

for stateful target services, whereas they can have selector annotations on any method for stateless target services.

Note: The parent process and the target process must both be configured to use the same protocol. Protocol matching and enabling is not handled automatically.

To learn about creating a Service Broker control, see Creating a New Service Broker Control. For more information on Service Broker Control methods, see Interface ServiceBroker Control.

Creating a New Service Broker Control

This topic describes how to create a new Service Broker control.

You can create a Service Broker control in two different ways, which are described in the following sections:

- Creating a New Service Broker Control Using the Control Wizard
- Creating a Service Broker Control from a Business Process

Creating a New Service Broker Control Using the Control Wizard

You can create a new Service Broker control and add it to your Web service or business process by using the **Insert Control - Service Broker** dialog.

Notes: When creating a Service Broker control that references a business process (Process.java), the business process must be in the current BEA WorkSpace Studio application.

If you are not in Design View, click the **Design** tab.

To create a new Service Broker control:

1. In the **Package Explorer** pane, double-click the business process to which you want to add the Service control. The business process is displayed in the **Design view**.
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Windows > Show View > Data Palette** from the menu bar.

3. Select **Service Broker**.

   The **Insert Control: ServiceBroker** dialog appears.

4. In the **Insert Control: ServiceBroker** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new Service Broker control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

     The **Create Control** wizard appears.
5. In the **Create Control** wizard enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.

     The **Insert control: Service Broker** dialog-box appears.
6. In the **Insert control: Service Broker** dialog box enter the following details:
   - In the **Process or WSDL** field, select the business process you want to access by selecting the name of a business process file.Click **Browse**, choose from the available list and click **OK**.
   - Select a start method from the **Start Method** menu. Only those start methods contained in the specified business process are displayed.
   - To specify a dynamic selector, enter a query in the **Query** field or click the **Query Builder** button to display the **Dynamic Selector** query builder.

     If you invoked the **Dynamic Selector** query builder, perform the following steps to build and test a query:

     - Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose `TPM` to bind lookup values to properties in the TPM repository. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain-wide `DynamicProperties.xml` file. You should only edit the `DynamicProperties.xml` file to bind lookup values to dynamic properties if the

domain is inactive. If the domain is active, it is recommended that you use the WebLogic Integration Administration Console to bind lookup values.

For more information on binding lookup values to dynamic properties using the WebLogic Integration Administration Console, see "Adding or Changing Dynamic Control Selectors" in Process Configuration.

- In the **Start Method Schema** area, select an element from the schema to associate it with the start method of the control. Only XML elements are displayed; non-XML elements are not supported. The resulting query appears in the **XQuery** area.
- Click **Create**.

7. Click **Finish**.

   The Service Broker control is created and displayed in the **Package Explorer** pane. An instance of the control is also created and is added to the **Data Palette**.

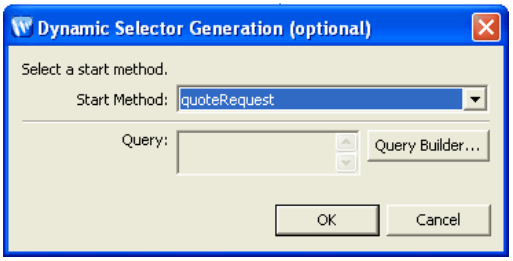
Creating a Service Broker Control from a Business Process

You can create a Service Broker control from an existing business process.

1. Right-click a business process in the **Package Explorer** pane and select **Generate > Service Broker Control**.

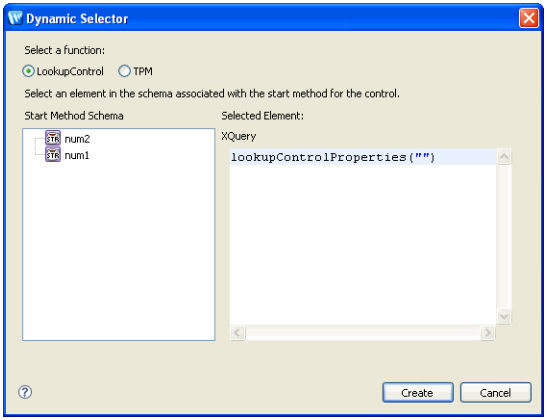
   The **Dynamic Selector Generation (optional)** dialog box is displayed (see Figure 12-1).

Figure 12-1 Dynamic Selector Generation



2. Select a start method from the **Start Method** menu. Only those start methods contained in the specified business process are displayed.
3. To specify a dynamic selector, enter a query in the **Query** field or click the **Query Builder** button to display the **Dynamic Selector** query builder (see Figure 12-2).

Figure 12-2 Dynamic Selector



If you invoked the **Dynamic Selector** query builder, perform the following steps to build and test a query:

a. Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose **TPM** to bind lookup values to properties in the TPM repository. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain-wide `DynamicProperties.xml` file. You should only use **LookupControl** to bind lookup values to dynamic properties if the domain is inactive. If the domain is active, it is recommended that you use the WebLogic Integration Administration Console to bind lookup values.

   For more information on binding lookup values to dynamic properties using the WebLogic Integration Administration Console, see "Adding or Changing Dynamic Control Selectors" in Process Configuration.

b. In the **Start Method Schema** area, select an element from the schema to associate it with the start method of the control. The resulting query appears in the **XQuery** area.

c. Click **OK**.

4. A new control file is displayed, indented beneath the selected business process file. The Service Broker control file is named using a prefix of SBControl to help distinguish it from Service controls. For example, if the associated Business Process file is `MyProcess.java`, the generated Service Broker control file is named `MyProcessSBControl.java`.

Note: Alternatively, you may create a Service Broker control file manually. For example, you may copy an existing Service Broker control file and modify the copy.

Notes on XQueries

When you are using XQuery expression and the XQuery Builder, it is important to remember:

- If you create a new Process control and want to use the XQuery Builder to associate an element in the schema with the start method for the new control, remember that XQuery Builder will only declare namespaces for schemas used in argument 2. So, if argument 1 uses "schema1", but argument 2 does not use "schema1", it will not declare namespace "schema1".
- If you use the XQuery Builder to generate XQuery expressions that involve an optional element, it is possible that you will have to manually edit the expression to generate a correct XPath location.
- Some XQuery expressions are not supported when using complex Predicates.

Setting Service Broker Properties

The Service Broker control adds the capability of dynamically binding some properties of the control. Dynamic binding of properties can be achieved the following ways:

- Using selectors
- Using the `setProperties()` API
- Using setter methods for individual properties, such as `setEndPoint()`.

To retrieve the current properties settings, use the `getProperties()` method. Note that this method does not return security-related settings such as `username`/`password`, `keyAlias`/`keyPassword`, and `keyStoreLocation`/`keyStorePassword`.

Starting with the method with the highest precedence, the hierarchy of property settings is:

1. properties dynamically bound using the `com.bea.wli.common.control.Selector` annotation and the `DynamicProperties.xml` file
2. properties set using the `setProperties()` method or other setter methods inherited from the Service control (`setConversationID`, `setEndPoint`, `setOutputHeaders`, `setPassword`, and `setUsername`)
3. properties set using static annotations

The `ServiceBrokerControlProperties` type is an XML Beans class that is generated out of the corresponding schema element defined in `DynamicProperties.xsd`. The

DynamicProperties.xsd file is located in the system folder of New Process Applications or in the system folder of the Schemas project.

Note: The ServiceBrokerControlProperties document's "password" field and associated setPassword() method should contain the intended password alias, not the actual password.

The setProperties() method uses this XML Beans class to set properties on a control instance. A selector on a Service Broker control method returns an XML document that conforms to the ServiceBrokerControlProperties element. The following sample shows how to programmatically set the endpoint property for the control. You add the bold code lines to the code generated when the control is created, overriding properties set using dynamic binding and static annotations:

```
import com.bea.wli.control.dynamicProperties.
ServiceBrokerControlPropertiesDocument;

import com.bea.wli.control.dynamicProperties.
ServiceBrokerControlPropertiesDocument.ServiceBrokerControlProperties;

    ServiceBrokerControlPropertiesDocument props= null;
    ServiceBrokerControlProperties sprops = null;

    public void sBC8InvokeSetProperties() throws Exception

    {

        props =
ServiceBrokerControlPropertiesDocument.Factory.newInstance();
        sprops = props.addNewServiceBrokerControlProperties();


sprops.setEndpoint("http://localhost:7001/BVTAppWeb/ServiceBrokerControl
        /SBC8DynPropHierarchyChild_2.jpd");
```

You construct the ServiceBrokerControlPropertiesDocument instance in the above section of code through a Data Transformation method or through XML Bean APIs. For more information on Data Transformations, see Guide to Data Transformation.

The following code provides a sample of a ServiceBrokerControlPropertiesDocument instance:

```
this.sbcp = ServiceBrokerControlPropertiesDocument.Factory.newInstance();

this.sbcp.addNewServiceBrokerControlProperties();
```

```
this.sbcp.getServiceBrokerControlProperties().setEndpoint("http://localhos
t:7001/");
```

Some control properties can be specified dynamically or in annotations (statically) on the control file. For example, the Service Broker control allows you to specify the `http-url` of the target service in the `@com.bea.wli.common.control.Location` annotation at the top of the control, or dynamically using the endpoint element in `DynamicProperties.xml`. In all such cases, a dynamically bound value for the property takes precedence over the static annotation.

If the domain is active, it is recommended that you use the WebLogic Integration Administration Console to perform dynamic binding. For more information on binding lookup values to dynamic properties using the WebLogic Integration Administration Console, see "Adding or Changing Dynamic Control Selectors" in Process Configuration.

Dynamic properties can also be specified by calling `setProperties` on the control, or by calling one of the setter methods, such as `ServiceBrokerControl.setEndPoint()`. Properties specified in this way take precedence over properties bound by selectors or annotations.

Properties applied using selectors remained bound until one of the following conditions occurs:

- A method marked `finish` on the control file is invoked
- A start method is invoked again
- The property is programmatically set by calling `setProperties` or a setter method

`ServiceControl.reset()` is overwritten by the Service Broker control to reset all dynamically set properties (in addition to all conversational states). Programmatically specified properties remain bound until `reset()` is invoked.

Property values set by a developer who is using the control are stored as annotations on the control's declaration in a JWS, JSP, or Process.Java file, or as annotations on its interface, callback, or method declarations in a Control file.

Service Broker Control Design Time Considerations

This section outlines some of the issues you must consider when you are designing a Service Broker control for your business process. The issues covered in this section are:

- Using a Service Broker Control in Stateless and Stateful Business Processes
- Using Service Broker Controls in Synchronous and Asynchronous Business Processes
- Using a Service Broker Control from the Parent Process

- Service Broker Control Location

Using a Service Broker Control in Stateless and Stateful Business Processes

The number of transactions contained in a business process determines whether the process is stateless or stateful. For more information, see Building Stateless and Stateful Processes. This section covers some of the issues you should consider when designing a Service Broker control for stateless and stateful business processes.

When designing your Service Broker control, adhere to the following rules:

- Whether you are designing a stateless or stateful business process, the Service Broker control can only be invoked by any method. However, you must always start a stateful process with a start method before using any continue methods.
- The start method that invokes the call governs which continue methods may be called during the call.
- The order in which the methods on the control are called matters in a stateful process.
- Stateful processes deal with only one particular instance of the control at a time.

Using Service Broker Controls in Synchronous and Asynchronous Business Processes

Business processes can have both synchronous or asynchronous request methods. For more information, see Building Synchronous and Asynchronous Business Processes. This section covers some of the issues you should consider when designing a Service Broker control for synchronous and asynchronous business processes.

When designing your Service Broker control, remember:

- In a synchronous invocation, the input and response arguments are combined into a single argument.
- In an asynchronous invocation, one-to-one mapping takes place. Any response node becomes a method within the callback interface.

Using a Service Broker Control from the Parent Process

The Service Broker control is typically used to call a subprocess from a parent business process. When the Service Broker control is invoked from the parent process, the control is invoked normally if the subprocess is in the same domain.

In cases where the parent process and subprocess are in different domains, or in the same domain but in different applications, and you want to use the Service Broker control for the parent process to communicate with the subprocess, you must create the Service Broker control in the subprocess domain or application, then manually copy it to the parent process domain or application, and then reference it as an existing control file.

Additionally, when the parent process and subprocess are in different applications, you will manually have to change the JMS and/or HTTP location of the location attribute.

Service Broker Control Location

When you create a Service Broker control, it is displayed in the **Package Explorer** pane and an instance of the control is added to the **Data Palette**. The location of the Service Broker control is displayed in the **JPD Configuration** pane.

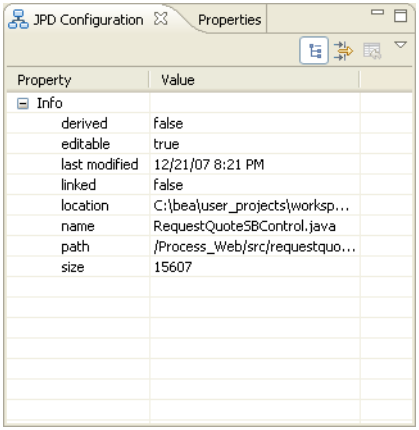
Note: If the **JPD Configuration** pane is not visible in BEA WorkSpace Studio, choose **Windows > Show View > Properties** from the menu bar.

To view the location of the Service Broker control:

1. Double-click the Service Broker control file in the **Package Explorer** pane.

   The Service Broker control is displayed in the **Design** view and the **JPD Configuration** pane displays the properties of the Service Broker control (see Figure 12-3).

Figure 12-3 Service Broker Control Properties



The location of the Service Broker control is displayed in the **location** section of the **JPD Configuration** pane. The location can contain a HTTP and/or JMS attribute. The HTTP location is a true HTTP address and the JMS location is a JMS url.

Service Broker Control Run-Time Considerations

This section outlines some of the run-time issues you must consider when you are invoking a Service Broker control in your business process.

The topics covered in this section are:

- Run-Time Rules for Service Broker Controls in Stateless and Stateful Business Processes
- Run-Time Rules for Service Broker Controls in Synchronous and Asynchronous Business Processes
- Security

Run-Time Rules for Service Broker Controls in Stateless and Stateful Business Processes

The number of transactions contained in a business process determines whether the process is stateless or stateful. For more information, see Building Stateless and Stateful Processes. This section covers some of the run-time issues you should consider when using a Service Broker control in stateless and stateful business processes.

The following rules govern the run-time operation of Service Broker controls in stateless and stateful business processes:

- In a stateless business process, the parent process makes a Simple Object Access Protocol (SOAP) or a non-SOAP invocation on the subprocess. The SOAP invocation is a generic process and the invocation is not simply the method called. In fact, a Web Services Description Language (WSDL) operation is what is actually invoked. For a non-SOAP invocation, the Service Broker control can make `FORM-POST` and `FORM-GET` requests.
- In a stateless business process, a WSDL file is used to communicate interface information between the parent process and the subprocess. For more information on WSDL files, see WSDL Files: Web Service Descriptions.
- In a stateful business process, the subprocess selected depends on the method specified by the Service Broker control. A new process instance is created based on the method specified by the Service Broker control and a new conversation ID is assigned to the process instance. You can specify the conversation ID. If you don't specify a conversation ID, the instance automatically generates its own conversation ID. For more information on conversations, see Designing Conversational Web Services.
- In a stateful business process, once a call is invoked and a conversation ID generated, any further continue methods that are invoked, interact with the original conversation ID. You need to specify a new start method to invoke an interaction with a new conversation ID.

Run-Time Rules for Service Broker Controls in Synchronous and Asynchronous Business Processes

Business processes can have both synchronous or asynchronous request methods. For more information, see Building Synchronous and Asynchronous Business Processes. This section covers some of the run-time issues you should consider when using a Service Broker control in synchronous and asynchronous business processes.

The following rules govern the run-time operation of Service Broker controls in synchronous and asynchronous business processes:

- In a synchronous business process, the Service Broker control allows a business process (or Web service) to invoke and receive callbacks from another service using one of several protocols. The most commonly used protocol is SOAP over HTTP. The target subprocess must expose the particular binding in the WSDL interface that corresponds to the protocol the client is using.

- In a synchronous business process, the transaction contexts are not propagated over the Service Broker control calls because the transport used is HTTP or JMS.
- In a synchronous business process, once a call is invoked, the parent process blocks and waits for a response from the subprocess before continuing to execute.
- In a synchronous business process, if a system level error is thrown in the transaction, a SOAP fault is generated and returned to the parent process or a `com.bea.control.ServiceControlException` is generated.
- In asynchronous business processes, all calls are invoked by a Web service which dumps the call instance into a Java Message Service (JMS) queue.
- In asynchronous business processes, the transaction is never propagated to the subprocess. In other words, the subprocess runs in its own transaction. The parent business process uses Java Message Service (JMS) to drop messages into a JMS queue. Once the parent process drops a message into the queue, it assumes that message delivery is successful. If an error occurs on the subprocess, the parent process is not informed of the error. For information on enhancing your business process so that you can detect failures, see "Asynchronous Subprocesses" in Building Synchronous and Asynchronous Business Processes.
- In asynchronous business processes, the parent business process continues to execute until the process finishes or until the process receives a callback from the subprocess. The callback is always associated with the process instance initially created by the parent business process.
- In an asynchronous business process, once the parent process invokes a message request to the subprocess, through the Service Broker control, it continues to instantiate and make further requests to the subprocess. If the subprocess doesn't have time to respond to the request from the parent process before it receives a second request, the second request will be dropped and the subprocess will not receive it at all. For this reason, you should design the parent process so that it blocks and waits for a callback from the subprocess before taking further action. For an example of using an Event Choice node to perform this function, see "Asynchronous Subprocesses" in Building Synchronous and Asynchronous Business Processes.
- In a asynchronous business process, if a system level error is thrown in the transaction, the error is not propagated back to the parent process.
- The `com.bea.control JwsContext` interface provides access to container services that support Web services (JWS files). For more information, see "com.bea.control JwsContext Interface" in com.bea.control packages. The `setUnderstoodInputHeaders()` method of

the `com.bea.control JwsContext` interface is not supported for asynchronous processes.

- It is recommended that you do not use callback methods that have return values, since Process files do not support return values on Service Broker Controls.

Security

The Service Broker control conforms to all standard security checks associated with JMS and WSDL. SSI is not supported. For more security information, see Security.

If you want to configure security settings for your Service Broker control, it is recommended that you use the WebLogic Integration Administration Console to configure these settings and do not set them on the process. For more information, see "Updating Security Policies" in Process Configuration in *Using The WebLogic Integration Administration Console*.

Maintaining Service Broker Controls

In general, Service Broker controls are utilized in situations where the parent process and subprocess are in the same application, or in situations where the parent process and subprocess are in different applications.

In both situations, if the Service Broker control is changed on the subprocess side of the transaction, you must regenerate the control on the parent side to insure that the control will operate normally. If the subprocess and parent process are in different applications, and the Service Broker control is changed in any way (i.e. request/response removed, etc.), backward compatibility may be compromised. If this happens, you must merge the Service Broker control on the subprocess side and repropagate the control back to the parent business process.

Note: Much like the Service control, the Service Broker control can throw a ServiceControlException class when a SOAP fault occurs. To access this SOAP fault, use the JPD's JpdContext.ExceptionInfo interface to see if the offending cause is a ServiceControlException. For more information on the ServiceControlException class, see "ServiceControlException Class" in com.bea.control packages. For more information on the JpdContext.ExceptionInfo interface, see "JpdContext.ExceptionInfo Interface" in com.bea.jpd Package.

Using Dynamic Binding

In many cases, control attributes are statically defined using annotations. Some controls provide a Java API to dynamically change certain attributes. Dynamic controls, including the Service

Broker and Process controls, provide the means to dynamically set control attributes. Attributes are determined at runtime using a combination of lookup rules and lookup values, a process called *dynamic binding*. Controls that support dynamic binding are called *dynamic controls*. The business process developer specifies lookup rules using BEA WorkSpace Studio while the administrator specifies look-up values using the WebLogic Integration Administration Console. This powerful feature means that control attributes can be completely decoupled from the application and can be reconfigured for a running application, without redeployment.

How the Service Broker Control Uses Dynamic Binding

The following scenario shows how the Service Broker control uses dynamic binding. `POService.java` needs to call an external service to obtain a quote on a specific item. Several vendors offer this service. The administrator needs to be able to access multiple implementations of the outside service without changing or redeploying `POService.jpd`.

Components Used in Dynamic Binding

This topic describes the capabilities that provide dynamic binding to the quote service using the Service Broker control.

com.bea.wli.common.control.Selector Tag

The method-level annotation, com.bea.wli.common.control.Selector, allows dynamic definition of certain properties of the control. The selector has an attribute, `xquery`, which is an XQuery expression, as shown in the following example:

```
/**
 * com.bea.wli.common.Conversation.Phase.Start
 * com.bea.wli.common.control.Selector xquery ::
 *     lookupControlProperties($request/vendorID) ::
 */
public void requestQuote(PurchaseRequest request);
```

The value of the selector's XQuery expression is an XML document with a schema that contains control property values. If you are accessing a TPM repository, the XQuery expression appears as follows:

```
/**
 * @com.bea.wli.common.Conversion(value =
com.bea.wli.common.Conversation.Phase.START)
 * @com.bea.wli.common.control.Selector( xquery=
"lookupControlProperties($request/vendorID)")
 * public void requestQuote(PurchaseRequest request);
```

When invoking a method on the control, the system looks for a selector annotation. If one is present, the XQuery expression is evaluated, possibly binding arguments of the Java call to arguments of the XQuery expression. The result of the XQuery expression is a String value that defines dynamic properties for the control.

Built-In XQuery Functions

Two types of XQuery functions are supplied to help you write selector expressions: `lookupControlProperties` and `lookupTPMProperties`. The `lookupControlProperties` function looks up values for dynamic properties specified in a domain-wide `DynamicProperties.xml` file. The `lookupTPMProperties` function looks up values from properties in the TPM (Trading Partner Management) repository.

To learn about the TPM repository, see Introducing Trading Partner Integration. To learn about adding or updating information in the TPM repository, see Trading Partner Management. The TPM control provides WebLogic Workshop business processes and Web services with query (read-only) access to trading partner and service information stored in the TPM repository. To learn about the TPM control, see TPM Control.

If the selector expression uses the `lookupControlProperties` function, the fully-qualified class name of the control together with the result of evaluating the selector are used as a lookup key into the `DynamicProperties.xml` file. If a match is found, the dynamic properties are applied before making the call to the target service.

DynamicProperties.xml File

`DynamicProperties.xml` is an XML file managed through the WebLogic Integration Administration Console. It contains mappings between values from the message payload (the lookup key) and corresponding control properties. It is a domain-wide file shared by all WebLogic Integration applications in the domain. This file allows you to administer dynamic properties without redeploying the application. The file is located in a subdirectory of the domain root named `wliconfig`. To learn about managing dynamic selectors, see Processes Configuration.

`DynamicProperties.xml` contains a sequence of `<control>` elements, one for each dynamic control control file. Each `<control>` element has a `name` attribute whose value is the fully-qualified Java class name of a control file. Nested inside the `<control>` element is a sequence of `<key>` elements which map string values that match the value of the parameter passed into the `lookupControlProperties` within the caller's selector to dynamic properties, as shown in the following example:

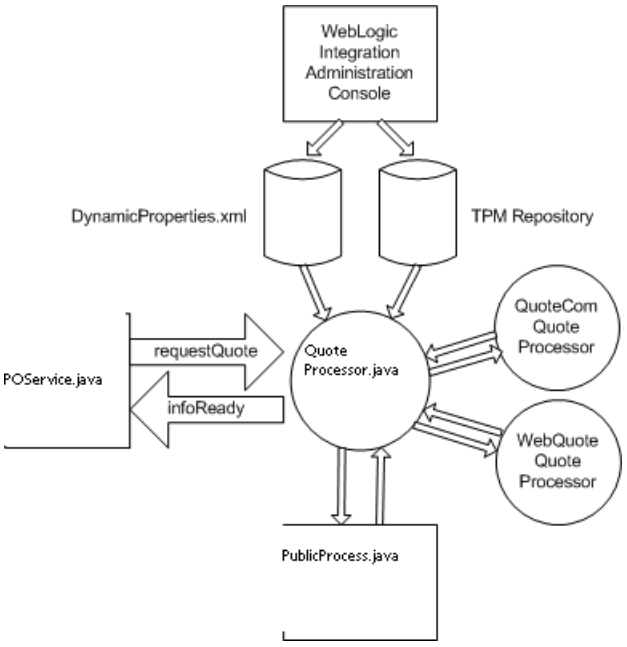
```
<DynamicProperties
      xmlns="http://www.bea.com/wli/control/dynamicProperties">

  <control name context-path="/sbcAppWeb"
           controlType="ServiceBrokerControl">
    <key value="QuoteCom">
      <ServiceBrokerControlProperties>
 <endpoint>http://www.quotecom.com/quotes/QuoteService</endpoint>
      </ServiceBrokerControlProperties>
    </key>

    <key value="WebQuote">
      <ServiceBrokerControlProperties>
<endpoint>http://www.webquote.com/quoteEngine/getQuote</endpoint>
      </ServiceBrokerControlProperties>
    </key>
  </control context-path>

  <control name context-path="quote.InternalQuote"
controlType="ProcessControl">
    <key value="OurQuote">
      <ProcessControlProperties>
        <targetURI>http://acme/myApp/PublicProcess.jpd</targetURI>
      </ProcessControlProperties>
    </key>
  </control context-path>
</DynamicProperties>
```

The WebLogic Integration Administration Console allows an administrator to view and edit entries in the `DynamicProperties.xml` file.

Quote Processing Example

This section shows how dynamic controls and selectors can help to implement the quote processing scenario. Figure 12-4 shows the components that participate in the dynamic binding:

Figure 12-4 Dynamic Binding



To achieve the required dynamic binding to the target service, the business process defined in `POService.java` uses a Service Broker control, `QuoteProcessor.java`, to call the quote service. Since the target is dynamically specified, the `com.bea.wli.common.control.Location` tag is not used. The Service Broker control is defined by the following control file:

```
import com.bea.control.ServiceBrokerControl;
import com.bea.control.ControlExtension;
import org.applications.PurchaseRequest;
import org.applications.PurchaseReply;


public interface QuoteProcessor
    extends ServiceBrokerControl, ControlExtension
{
    public interface Callback
    {
```

```
        public void infoReady (PurchaseReply reply);
    }
    /**
     * @com.bea.wli.common.Conversation(value =
com.bea.wli.common.Conversation.Phase.START)"
     * @com.bea.wli.common.control.Selector( xquery=
"lookupControlProperties($request/vendorID)")
public void requestQuote(PurchaseRequest request
}
```

At runtime, the control container needs to bind the proxy represented by the control to the proper implementation. This is driven by a selector XQuery expression tagged on the start method of the Service Broker control interface (`com.bea.wli.common.control.Selector`).

Note: For controls representing stateless components, each method can have a selector. For methods without selectors, the default location defined in the annotation is used. If the target location is not resolved after applying the selector, a runtime exception is raised.

The selector returns an XML fragment that contains the dynamic properties of the control. For example:

```
<ServiceBrokerControlProperties>
    <endpoint>
       http://www.quotecom.com/quotes/QuoteService/endpointURI>
    </endpoint>
    <username>fred</username>
    <password>@$%&*</password>
</ServiceBrokerControlProperties>
```

In this example, the selector uses a standard XQuery function called `lookupControlProperties()`. This function looks up the control properties from the `DynamicProperties.xml` file based on the key passed to it. In the example, the key is the vendor ID that is extracted from the payload. The result passed back by `lookupControlProperties()` is a `<ServiceBrokerControlProperties>` element.

The key-attribute mapping information used by `lookupControlProperties()` is stored in the `DynamicProperties.xml` file. The schema for the dynamic properties file can handle all the attributes that are valid for dynamic controls. You can define selectors when you create the control or by directly editing the control source code.

An administrator can define the mapping between the selector value and the implementation using the WebLogic Integration Administration Console. The WebLogic Integration Administration Console allows an administrator to specify the following properties:

- Endpoint URI
- Protocol to use when making the call: http-soap, http-xml, jms-soap, jms-xml, form-get and form-post. The default is http-soap.

  Note: The parent process and the target process must both be configured to use the same protocol. Protocol matching and enabling is not handled automatically.

- Any credentials needed to make the call:
  - User name and password to invoke the remote service (base authentication)
  - Certificate alias and password, if the remote service requires SSL with two-way authentication
  - Certificate alias and password, if digital signature is required
  - Keystore location, password and type, in case a client certificate is required

Notes on Service Broker Control Annotations

This section provides information on the Service Broker control specific behavior of the following annotations:

- `@com.bea.wli.common.ParameterXml`
- `@com.bea.wli.common.Conversation`
- @jc:parameter xml
- @com.bea.wli.common.Protocol

@com.bea.wli.common.ParameterXml

The `@com.bea.wli.common.ParameterXml` annotation specifies that there should be a queue between the component's implementation code and the message transport wire for the specified method or callback. For more information, see "@com.bea.wli.common.ParameterXml Annotation" in Java Control Annotations.

When you generate a Service Broker control request method, do not edit any of the attributes of the `@common:message-buffer` annotation. The `@common:message-buffer` annotation is visible in the **message-buffer** section of the **JPD Configuration** pane.

@com.bea.wli.common.Conversation

The `com.bea.wli.common.Conversation` annotation specifies the role that a control's methods or callbacks play in a conversation. For more information, see "`com.bea.wli.common.Conversation` annotation" in Java Control Annotations.

When you generate a Service Broker control, do not edit the phase attribute of the `com.bea.wli.common.Conversation` annotation. The `com.bea.wli.common.Conversation` annotation is visible in the **conversation** section of the **JPD Configuration** pane.

@jc:parameter xml

The `@jc:parameter-xml` annotation specifies characteristics for marshaling data between XML messages and the data provided to the parameters of a Web service operation. For more information, see "@jc:parameter-xml Annotation" in Java Control Annotations.

It is recommended that you do not use this annotation when designing a Service Broker control.

@com.bea.wli.common.Protocol

The `@com.bea.wli.common.Protocol`annotation specifies which protocols and message formats can be accepted by the Web service represented by a Web Service control, and by the operations on that Web service. For more information, see "@com.bea.wli.common.Protocol Annotation" in Java Control Annotations.

If you use the Service Broker control to call properties, do not use the `http-xml` attribute of `com.bea.control.ServiceControl.JmsSoapProtocol`. This attribute is not supported for use with the Service Broker control. This attribute is visible in the protocol section of the **JPD Configuration** pane.

ALSB Control

Note: The ALSB control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

The ALSB control allows your web service or business process to invoke an AquaLogic Service Bus (ALSB) proxy service, and provides support for security and transaction context propagation.

Topics Included in this Section

Related Topics

AquaLogic Service Bus User Guide

Service Broker Control

Overview: ALSB Control

The ALSB control allows a business process to invoke the ALSB proxy services, which have Service Bus Transport configured through RMI, and send requests to and receive callbacks from a Web service, or a remote Web service or business process.

ALSB control extends Service Broker control, and the functionality supported by the service control is also supported by the ALSB control. For more information on Service Broker Control, see Service Broker Control.

ALSB control does the following:

- It supports functionality of service control such as conversations, callbacks, message and transport level security, SOAP (1.1 and 1.2), WS-Policy, and WS-Addressing.
- It supports security and transaction context propagation.
- It supports synchronous, one-way invocation of ALSB proxy services, which are configured with SB transport.
- It transports the SOAP, abstract SOAP, and abstract XML messages over RMI.

The first step is creating a ALSB Control file. The Control can be automatically generated from a target service (WSDL file) using BEA WorkSpace Studio, or can be created using the **Add** function in the **Controls** section of the BEA WorkSpace Studio **Data** palette. The methods and callbacks on the Control file correspond to operations and callbacks of the target service. An instance of this Control file is used by a parent service to call the target service and can also be used to get callbacks from the target service. ALSB control files can only have selector annotations on start methods for stateful target services, whereas they can have selector annotations on any method for stateless target services.

Note: The parent process and the target process must both be configured to use the same protocol. Protocol matching and enabling is not handled automatically.

Creating an ALSB Control

You can create an ALSB control:

– From an ALSB proxy service using the proxy service's WSDL file, see Creating an ALSB Control from a WSDL.

– From a proxy service in an existing ALSB Project to a WLI project in the same wokspace, see Creating an ALSB Control Using the Service Consumption Framework.

Creating an ALSB Control from a WSDL

You can create a new ALSB control and add it to your Web service or business process by using the **Insert Control - ALSB** dialog.

Note: If you are not in Design view, click the **Design** tab.

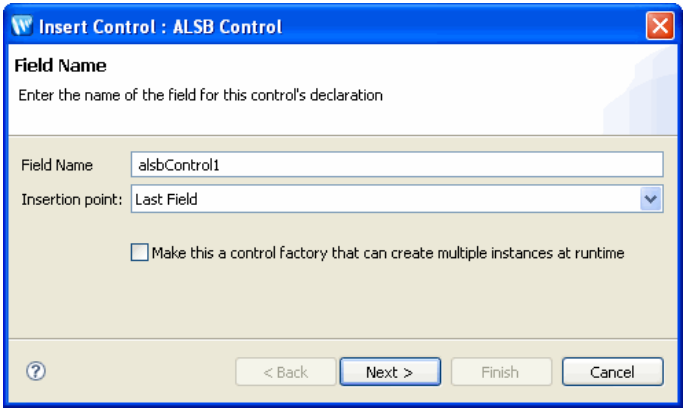
To create a new ALSB control:

1. In the **Package Explorer** pane, double-click the business process to which you want to add the ALSB control. The business process is displayed in the **Design** view.
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   If the **Data Palette** view is not visible in BEA WorkSpace Studio, click **Window** > **Show View** > **Data Palette** from the menu bar.
3. Select **ALSB**.

   The **Insert Control: ALSB Control** dialog appears (see Figure 13-1).

Figure 13-1 Insert Control:ALSB Control



4. In the **Insert Control:ALSB** dialog box enter the following details:

– In the **Field Name**, type the variable name used to access the new ALSB control instance from your business process. The name you enter must be a valid Java identifier.

– In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.

– Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

– Click **Next**.

The **Create Control** wizard appears.

5. In the **Create Control** wizard enter the following details:

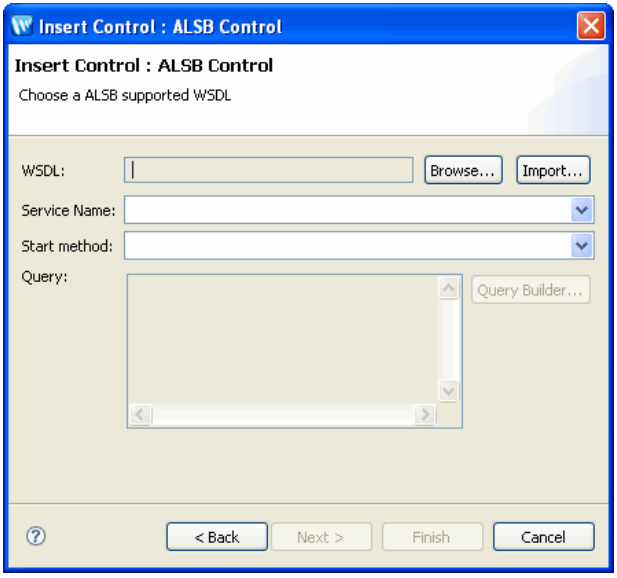
– In the **Name** field, type the name of your new control extension file.

– Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

– Click **Next**.

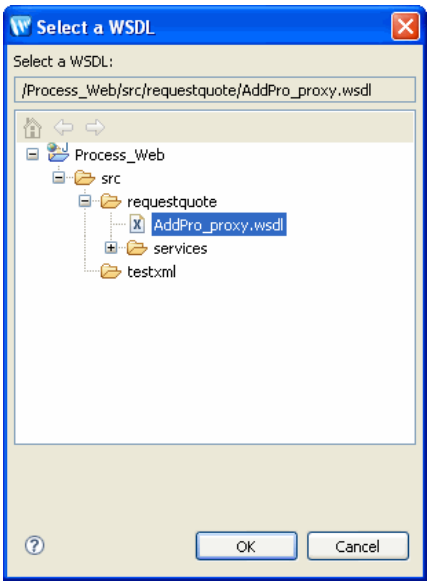
The **Insert control: ALSB Control** dialog-box appears (see Figure 13-2).

Figure 13-2 Insert Control: ALSB Control



6. In the **Insert control: ALSB Control** dialog box enter the following details:
   – In the **WSDL** field do one of the following:
     • Click **Browse** and select the WSDL file you want to access from the available list and, click **OK** (see Figure 13-3).

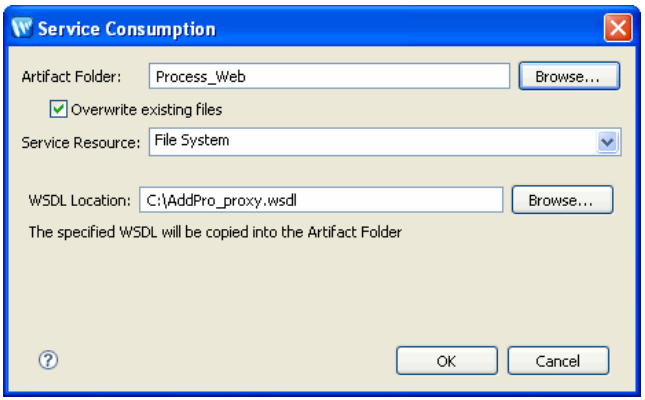
Figure 13-3 Select a WSDL



     • Click **Import**, and in the **Service Consumption** dialog-box select a **Artifact Folder** and the **Service Resource** from the drop-down list next to it, and click **OK** (see Figure 13-4).

Note: An ALSB Control cannot be created from a WSDL file located outside the current Web project.

Figure 13-4 Service Consumption



– Select a service name from the **Service Name** menu. Only those service name contained in the specified business process are displayed.

– Select a start method from the **Start Method** menu. Only those start methods contained in the specified business process are displayed.

– To specify a dynamic selector, enter a query in the **Query** field, by clicking the **Query Builder** button to display the **Dynamic Selector** query builder.

  If you invoked the **Dynamic Selector** query builder, perform the following steps to build and test a query:

  • Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose `TPM` to bind lookup values to properties in the TPM repository. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain-wide `DynamicProperties.xml` file. You should only edit the `DynamicProperties.xml` file to bind lookup values to dynamic properties if the domain is inactive. If the domain is active, it is recommended that you use the WebLogic Integration Administration Console to bind lookup values.

  For more information on binding lookup values to dynamic properties using the WebLogic Integration Administration Console, see “Adding or Changing Dynamic Control Selectors” in Process Configuration.

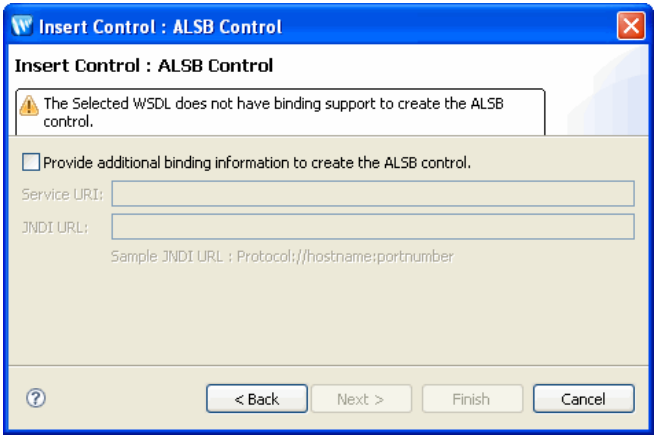
  • In the **Start Method Schema** area, select an element from the schema to associate it with the start method of the control. Only XML elements are displayed; non-XML elements are not supported. The resulting query appears in the **XQuery** area.

  • Click **Create**.

– Click **Next**.

If the ALSB bindings are not configured in your WSDL file, the Insert Control: ALSB Control dialog box appears (see Figure 13-5), do the following:

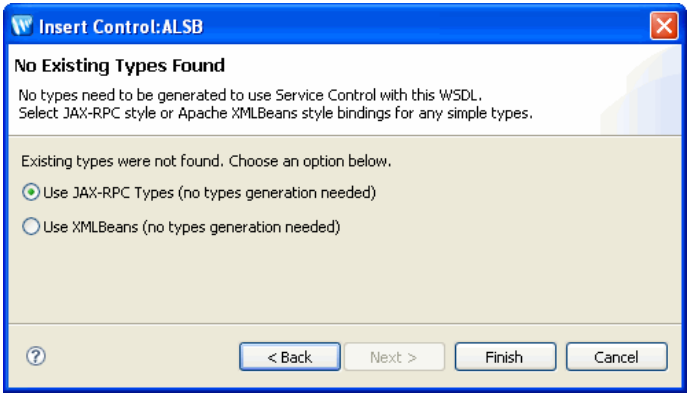
Figure 13-5 Insert Control: ALSB Control



- Check the **Provide additional binding information to create ALSB transport control** check-box.
- Enter the required information in the **Service URI** field**.**
- Enter the required information in the **JNDI URL** field.
- Click **Next**.

The **No Existing Types Found** dialog-box appears (see Figure 13-6).

Figure 13-6 No Existing Types Found



– Select one of the option and click **Finish**.

The ALSB control is created and displayed in the **Package Explorer** pane. An instance of the control is also created and is added to the **Data Palette**.

Creating an ALSB Control Using the Service Consumption Framework

To create an ALSB Control using the proxy service, you need to have an existing ALSB project.

1. In the **Package Explorer** pane, double-click the business process to which you want to add the ALSB control. The business process is displayed in the **Design view**.
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.
3. Select **ALSB**.

   The **Insert Control: ALSB** dialog appears
4. In the **Insert Control:ALSB** dialog box enter the following details:
   – In the **Field Name**, type the variable name used to access the new ALSB control instance from your business process. The name you enter must be a valid Java identifier.
   – In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.

– Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

– Click **Next**.

The **Create Control** wizard appears.

5. In the **Create Control** wizard enter the following details:

– In the **Name** field, type the name of your new control extension file.

– Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

– Click **Next**.

The **Insert control: ALSB** dialog-box appears.

6. In the **Insert control: ALSB** dialog-box, do the following:

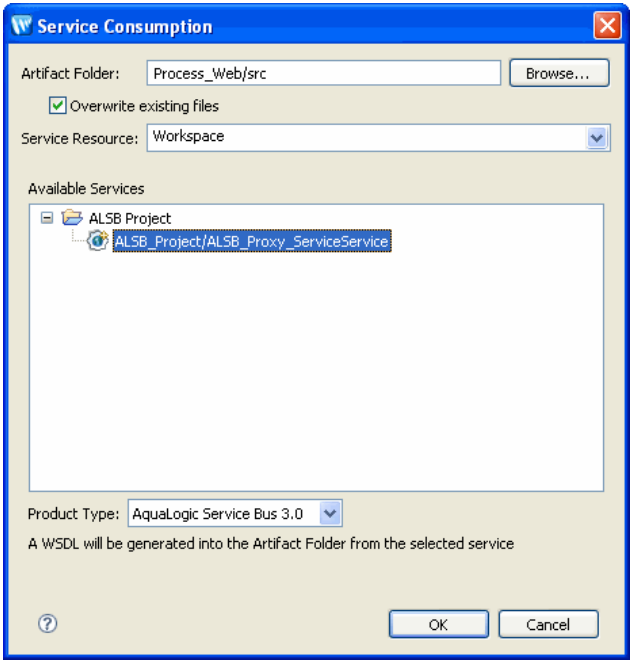
– In the **WSDL** field, click **Import.**

The **Service Consumption** dialog-box appears.

In the **Service Consumption** dialog-box do the following:

- Select an **Artifact Folder**.
- Decide whether you want to overwrite existing files, and select or clear the **Overwrite existing files** check box.
- Select **Workspace** from the **Service Resource** drop-down list.
- Select **AquaLogic Service Bus 3.0**, as the **Product Type**.
- Select the ALSB proxy service, from the **Available Services** (see Figure 13-7).

Figure 13-7 Available Services



- Click **Ok**.

– Select a service name from the **Service Name** menu. Only those service name contained in the specified business process are displayed.

– Select a start method from the **Start Method** menu. Only those start methods contained in the specified business process are displayed.

– To specify a dynamic selector, enter a query in the **Query** field, by clicking the **Query Builder** button to display the **Dynamic Selector** query builder.

– Click **Next**.

The **No Existing Types Found** dialog-box appears.

– Select one of the option and click **Finish**.

The ALSB control is created using a proxy service and displayed in your business process, in the **Package Explorer** pane. An instance of the control is also created and is added to the **Data Palette**.

ALSB Control Annotations

ALSB control extends Service Broker control. In addition to the annotations of the Service Broker Control, other than those annotations it provides the following additional annotations depending on the requirement:

- RmiSoapProtocol
- RmiSoap12Protocol
- SBTransport
- CustomHeader
- CustomHeaders
- RmiXmlProtocol
- OneWay
- Transaction Context Propagation
- Security Context Propagation

See Notes on Service Broker Control Annotations, for more information on Service Broker annotations.

RmiSoapProtocol

This annotation is a marker annotation, specifies the ALSB control supports SOAP 1.1 over RMI. This annotation can only be specified at the class level. This annotation is used for supporting SOAP over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface RmiSoapProtocol
{
}
```

Note: This annotation is auto generated based on the WSDL and, you should not modify this annotation. When this annotation is specified, you should not specify service control supported annotations like `HttpSoapProtocol`, `HttpSoap12Protocol`,

`JmsSoapProtocol`, and `JmsSoapProtocol`. `Location` which is a service control annotation is optional. The service control annotation `WSDL` is required.

RmiSoap12Protocol

This annotation is a marker annotation, specifies the ALSB control supports SOAP 1.2 over RMI. This annotation can only be specified at the class level. This annotation is used for supporting SOAP over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface RmiSoap12Protocol
{
}
```

Note: This annotation is auto generated based on the WSDL and, you should not modify this annotation. When this annotation is specified, you should not specify service control supported annotations like `HttpSoapProtocol`, `HttpSoap12Protocol`, `JmsSoapProtocol`, and `JmsSoapProtocol`. `Location`, which is a service control annotation is optional. The service control annotation `WSDL` is required.

SBTransport

This annotation is used to specify the RMI related parameters required for the SB Thin Client API. This annotation can only be specified at the class level. This annotation is used for supporting SOAP, abstract SOAP, and abstract XML over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
public @interface SBTransport
{
String serviceURI();
boolean execute() default true;
       String jndiURL() default “”;
```

Table 13-1 describes the fields within the SBTransport annotation.

Table 13-1 Fields in the SBTransport Annotation

Fields	Description
`serviceURI`	This specifies the URL of the ALSB proxy service, which is configured with SB transport and is required.
`execute`	This specifies whether the invocation of the external service should happen with in the same security context as the business process or not. If the invocation is false, the invocation happens with in the security context of the JNDI lookup. If true, the invocation happens with in the same security context of the business process and JNDI lookup happens in a different security context depending on whether the principal/credential is set or not. **Note:** The default value is `true`.
`jndiURL`	This specifies the URL of the JNDI provider on the ALSB server. If the URL is not specified then the local JNDI provider would be considered.

Note: This annotation should only be specified along with the annotations `RmiSoapProtocol` or `RmiSoap12Protocol.Location`, which is a service control annotation is optional. The service control annotation `WSDL` is required.

CustomHeader

This annotation is used to specify the custom header to the SB Transport. This annotation can be specified at the class level. This annotation is used for supporting SOAP over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface CustomHeader
{
String name;
String value;
}
```

Table 13-2 describes the fields within the CustomHeader annotation.

Table 13-2 Fields in the CustomHeader Annotation

Fields	Description
Name	This specifies the header name.
Value	Describes the value for the header specified through “name” attribute.

Note: This annotation should only be specified along with the annotations `RmiSoapProtocol` or `RmiSoap12Protocol`.

CustomHeaders

This annotation is used to specify the custom headers to the SB Transport. This annotation can be specified at the class level. This annotation is used for supporting SOAP, abstract SOAP, and abstract XML over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface CustomHeaders
{
CustomHeader[] headers;
}
```

The fields with in the annotation are headers, which represents an array of name value pairs.

Note: This annotation should only be specified along with the annotations `RmiSoapProtocol` or `RmiSoap12Protocol`.

RmiXmlProtocol

This annotation is a marker annotation and used for specifying that the ALSB control supports abstract XML message over RMI. This annotation can only be specified at the class level. This annotation is used for supporting abstract XML and abstract SOAP over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface RmiXmlProtocol
```

```
{
}
```

Note: When you specify this annotation, do not specify `RmiSoapProtocol`, `RmiSoap12Protocol`, and any service control supported annotations. You need to specify the `SBTransport` annotation. In this case there should not be any callback interface (with `@EventSet annotation`) with in the ALSB control.

OneWay

This annotation is a marker annotation and is used to specify that the operation is one way. This annotation can only be specified at the method level. This annotation is used for supporting abstract XML, and abstract SOAP over RMI. This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface OneWay
{
}
```

Note: This annotation should be specified only when `RmiXmlProtocol` is specified.

When this annotation is specified the method signature for:

– Sending the abstract XML over RMI looks like the following:

```
public void sendAbstractXml(XmlObject xmlMessage);
```

– Transporting abstract SOAP messages over RMI looks like the following:

```
public void sendAbstractSoap(SOAPMessage soapMessage);
```

Transaction Context Propagation

ALSB control supports transaction context propagation at method level within a business process, where you can decide whether to propagate the transaction context for that method or not using the `Transactional` annotation. This annotation also specifies the RMI invocation timeout, if you are not propagating the transaction context.

If you specify the `transaction` annotation at the control level, it will be applicable for all the methods with in the control and if specified at the method level, it will be applicable to the method level.

This annotation looks like the following:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Transactional
{
boolean value() default true;
       int timeout() default 60;
}
```

Table 13-3 describes the fields within the Transactional annotation.

Table 13-3 Fields in the Transactional Annotation

Fields	Description
Value	This determines whether to propagate the transaction context.
Time	This specifies the timeout value for RMI invocation, when transaction context is not propagated.

Note: This annotation should only be specified along with the ALSB control defined annotations only. If this annotation is not specified, by default the transaction context would be propagated to ALSB.

Security Context Propagation

If you specify the principal and credentials as part of `SBTransport` annotation, those credentials would be used to authenticate with remote JNDI provider. Depending on the value of the `execute` attribute in `SBTransport` annotation, security context propagation happens accordingly as explained below while transporting the message to ALSB.

- If you specify `true` for the execute attribute, then the JNDI lookup happens with the supplied principal/credential and the transporting the message/remote invocation happens with in the context of the current authenticated subject.
- If you specify `false` for the execute attribute, then the JNDI lookup & transporting the message/remote invocation happens with the supplied principal/credentials.

Example ALSB Control

This section shows an example of ALSB control:

```
package requestquote;
import com.bea.control.SbTransportControl;
import org.apache.beehive.controls.api.events.EventSet;
import org.apache.beehive.controls.api.bean.ControlExtension;
@SbTransportControl.SBTransport(serviceURI = "/MyAddProxy", jndiURL =
"t3://localhost:7001")
@SbTransportControl.RmiSoapProtocol
@SbTransportControl.SOAPBinding(style =
SbTransportControl.SOAPBinding.Style.DOCUMENT, use =
SbTransportControl.SOAPBinding.Use.LITERAL, parameterStyle =
SbTransportControl.SOAPBinding.ParameterStyle.WRAPPED)
@SbTransportControl.WSDL(path = "../requestquote/AddPro_proxy.wsdl",
service = "AddPro")
@ControlExtension
public interface ALSBControl extends SbTransportControl
{
    static final long serialVersionUID = 1L;
    public int addNumbers(int num1_arg,int num2_arg);

   /** This event set interface provides support for the onAsyncFailure
event.
    */
   @EventSet(unicast=true)
   public interface Callback extends SbTransportControl.Callback {};
    public ALSBControl create();         }
```

TIBCO Rendezvous Control

TIBCO® Rendezvous™ (a product from TIBCO: www.tibco.com) enables exchange of data across applications running on distributed platforms. TIBCO Rendezvous (TIBCO RV) Control in WebLogic Integration™ enables seamless connection to, and transfer of data using the Rendezvous daemon. It enables communication via many of the features provided by the TIBCO Rendezvous product, including Certified Message Delivery, Distributed Queue and so on. The sending and receiving applications can be on multiple platforms, as long as the Rendezvous daemon is running on the host machine, or is remotely accessible to the host.

Note: The TIBCO RV control is available in BEA WorkSpace Studio only for licensed users of WebLogic Integration.

Disclaimer

Use of the TIBCO RV control and event generator with BEA WebLogic Integration in no manner confers or grants the right to use TIBCO Rendezvous including "dynamic libraries". In order to use such TIBCO products, the user of the TIBCO RV control and event generator must obtain a valid license from TIBCO. See `http://www.tibco.com` for information on how to obtain a licensed copy of Rendezvous.

Topics Included in This Section

Overview: Rendezvous Control

Describes the function of the TIBCO RV control within WebLogic Integration.

Creating and Configuring a New Instance of the TIBCO RV Control
Describes how to create and configure a new TIBCO RV control.

Using Subject in a Message
Describes how to set and retrieve the subject descriptor attributes of the message.

Sending and Receiving Messages
Describes the methods used to send and receive messages.

Setting Dynamic Properties
Describes how to modify the TIBCO RV control properties at run time.

Overview: Rendezvous Control

The TIBCO RV control enables WebLogic Integration business processes to send and receive messages in the Rendezvous environment. In this environment, the messages are conveyed using Rendezvous daemon (`rvd`) and Rendezvous agent (`rva`) transports.

Using the TIBCO RV control, you can send and receive messages in XML, String and TIBCO proprietary Rendezvous Message (`TibrvMsg`) formats. You can specify TIBCO RV control properties while configuring Rendezvous control or dynamically at run time. Following are some of the other features of TIBCO RV control:

- Sending a request message and waiting for a reply
- Sending a reply for a message
- Asynchronous callback facility to confirm delivery or failure of certified messages
- Registration of anticipated listeners

The TIBCO RV control complements the other controls provided in WebLogic Integration, and can be used with other WebLogic Integration business processes. To learn more, see Using Controls in Business Processes in *Using Integration Controls*.

The TIBCO RV event generator listens on a subject, and publishes the received messages to the WebLogic Integration message broker channels. For more information, see the TIBCO Rendezvous Event Generator.

Creating and Configuring a New Instance of the TIBCO RV Control

You can create and configure a new instance of the TIBCO RV control and add it to your business process. This topic includes the following sections:

Describes how to add a new TIBCO RV control.

To Specify TIBCO RV Control General Settings

Describes how to configure the general settings for the TIBCO RV control such as port id, host name and so on.

To Specify TIBCO RV Control Advanced Settings

Describes how to configure Certified Message settings for the TIBCO RV control.

Prerequisites before creating Tibco RV Control

Before creating the TIBCO RV Control, perform the following:

1. Install and configure TibcoRV on your machine. Ensure that the installation directory should be in server PATH.
2. Enter the tibjrv.jar in the CLASSPATH:
   `EXT_PRE_CLASSPATH=D:\Installs\TIBCO\TIBRV\lib\tibrvj.jar;%EXT_PRE_CLASSPATH%.`
3. TIBCO installation bin and lib folder should be there in PATH and LD_LIBRARY_PATH respectively.

   ```
   export LD_LIBRARY_PATH=/opt/tibco/tibrv/lib:$LD_LIBRARY_PATH
   export PATH=/opt/tibco/tibrv/bin:$PATH
   ```

To Add a New TIBCO RV Control

To add a new TIBCO RV control to WebLogic Integration, perform the following steps:

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the Tibco control. The business process is displayed in the **Design** view.
2. Click  on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **Tibco**.

   The **Insert control: Tibco** dialog box appears.

4. In the **Insert control: Tibco** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new Tibco control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point**: from the drop-down list select the insertion node, you want to add the control.

   – To make this a control factory select the **Make this a control factory that can create multiple instances at runtime** check box, otherwise clear the check box.

   – Click **Next**.

   The **Create Control** dialog-box appears.

5. In the **Create Control** dialog box enter the following details:

   – In the **Name** field, type the name of your new control extension file.

   – Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

   – Click **Next**.

   The **Insert control- Tibco RV** dialog-box appears.

6. In the **Insert control- Tibco RV** dialog box enter the following details:

To Specify TIBCO RV Control General Settings

   – In the **Service** field, enter the service name which the TIBCO RV daemon will use to convey the message.

   – In the **Network** field, enter the name of the network with which the TIBCO RV daemon will communicate. If no network is specified, the default network interface will be used.

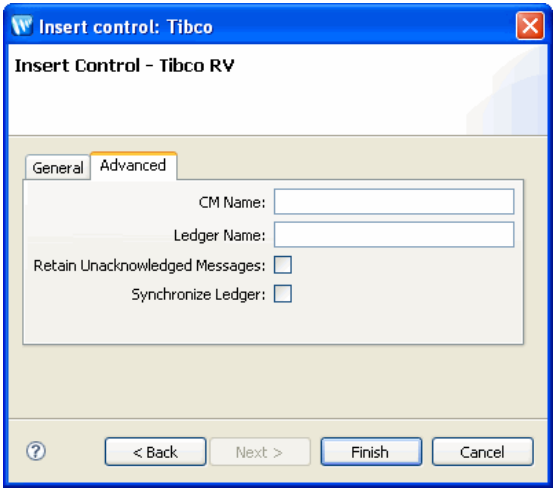
   – In the **Daemon** field, enter the location where the TIBCO RV daemon is running to establish communication. If the TIBCO RV daemon is running on a different network, specify the **remote_host:port_id** details in the **Daemon** field. For example, **`beaserv1:1589`** where **`beaserv1`** is the remote host name and **`1589`** is the port id.

   – Click **Use CM** to select the Certified Messaging option.

To Specify TIBCO RV Control Advanced Settings

To specify certified messaging settings for the TIBCO RV control, perform the following tasks in Step 3 of the **Insert Control - TIBCO RV** dialog; the **Advanced Settings** tab, (see Figure 14-1) below:

Figure 14-1 Insert TIBCO RV Control - Advanced Settings



- Click the **Advanced** tab, to display the advanced options for Certified Messaging. This tab is applicable only if you have selected the **CM** radio button in the **General** tab.
- In the **CM Name** field, provide the CM transport name. The name identifies the CM transport to other CM transports, and is par of the CM label that identifies outbound messages from the CM transport.
- In the **Ledger Name** field, provide the ledger name with its location. Each CM transport keeps a ledger, in which it records information about every unresolved outbound certified message, every subject for which this CM transport receives (inbound) certified messages, and other cooperating CM transports.
- Click **Retain Unacknowledged Messages** radio button to store any unacknowledged messages as part of its decentralized architecture.
- Click **Synchronize Ledger** radio button to perform a synchronized update of the ledger file. Each time the ledger is updated, the call does not return until data is safely written to the storage medium.

The Java Files for a TIBCO RV Control

When you create a new instance of the TIBCO RV control, you create a new Java file in your project. The contents of the TIBCO RV control's file depends on the selections made in the **Insert Control - TIBCO RV** dialog.

The two examples in this section depict control files created for a certified message and a non-certified message.

Sample control File for a TIBCO RV Control Using Certified Messaging

```
package requestquote;

/*

 * An extended TibcoRV control.

 */

@org.apache.beehive.controls.api.bean.ControlExtension

@com.bea.control.TibcoRV.Transport(service="7500",network="beaserv1",daemo
n="beasever1:7500")

@com.bea.control.TibcoRV.CMTransport(cmname="new",ledgername="Ledger",requ
estold=true,syncledger=true)

@com.bea.control.TibcoRV.UseCM(true)

public interface TibcoRV extends com.bea.control.TibcoRV {

       public void addListenerForCM(String cmName, String subject);

       static final long serialVersionUID = 1L;

       public TibcoRV create();

}
```

Sample control File for a TIBCO RV Control Without Certified Messaging

```
package requestquote;

/*

 * An extended TibcoRV control.

 */

@org.apache.beehive.controls.api.bean.ControlExtension
```

```
@com.bea.control.TibcoRV.Transport(service = "7500", network = "beaserv1",
daemon = "beasever1:7500")

@com.bea.control.TibcoRV.CMTransport(cmname = "", ledgername = "",
requestold = false, syncledger = false)

@com.bea.control.TibcoRV.UseCM(false)

public interface TibcoRV extends com.bea.control.TibcoRV {

       static final long serialVersionUID = 1L;

       public TibcoRV create();

}
```

Using Subject in a Message

This section provides details on construction of a subject name. Each message in the TIBCO Rendezvous environment contains a **subject** name. An application creates a message and sends it with a **subject** through the Rendezvous environment. Applications at the other end accept the message by listening on the **subject**.

Subject Name Syntax

Subject name definitions have basic restrictions, for example, its length, structure and usage of special characters. System designers and developers can set the conventions for subject names keeping in mind the following:

- **Structure** — A subject is a string of characters that is divided into elements by the dot (.) character.
- **Length** — The maximum allocated length of a subject (including dot separators) is 255 characters, some of which is reserved for internal use by Rendezvous.
- **Special Characters —**
  - Avoid underscore (_) character at the beginning of the subject name, except if the first element name is `_INBOX` or `_LOCAL`.
  - Avoid the dot (.) character as part of an element as it is the reserved delimiter.
  - Greater-than (>) and Asterisk (*) characters are reservoir wildcard characters.

Caution: The restrictions and conventions are implemented by TIBCO Rendezvous and information in this section is indicative only. Refer TIBCO Rendezvous product

documentation for more up-to-date information on restrictions, guidelines and examples.

http://www.tibco.com

Sending and Receiving Messages

You can send and receive messages with TIBCO RV control using any one of `sendMessage`, `sendReply` or `sendRequest` functions, and the TIBCO Event Generator, respectively. Messages can be in the form of Rendezvous proprietary data format, string and XML.

Sending Messages

This section provides information on the various functions available for sending messages. To send a message, select a function based on the data type of the message that you want to send. All these functions can send reliable and certified messages, as defined while creating the control. Certified message functions will return sequence numbers while reliable message functions will return zero.

The **`sendRequest`** function creates a listener that keeps listening for messages to the reply subject and hence, it does not require explicit creation of listeners. The function returns an instance of the `TibrvMsg`, which can be used for sending replies.

The **`sendRequest`** and **`sendReply`** functions are often used together as pairs. An example of such an implementation is:

```
replymsg = sendRequest(msg, "send.Subject","reply.Subject",5.0);
sendReply( replyMsg, newMsg);
```

sendMessage ()

Used to send a message via RVDTransport, or a labelled message via CMTransport.

```
public long sendMessage(TibrvMsg msg , String subject, double timeout)
```

`msg`: the message that needs to be sent
`subject`: subject of the message
`timeout`: time limit for delivery of the message

sendMessageAsString ()

Used to send a string message via RVDTransport, or a labelled string message via CMTransport.

```
public long sendMessageAsString(String  msg , String fieldName ,String
subject, double timeout)
```

msg: the string message that needs to be sent
fieldName: name of the TibRV field used to send the payload
subject: subject of the message
timeout: time limit for delivery of the message

sendMessageAsXML ()

Used to send an XML message via RVDTransport, or a labelled XML message via CMTransport.

```
public long sendMessageAsXML(XmlObject  msg ,String fieldName ,String
subject, double timeout)
```

msg: the XML message that needs to be sent
fieldName: name of the TibRV field used to send the payload
subject: subject of the message
timeout: time limit for delivery of the message

sendRequest ()

Used to send a request message via RVDTransport, or a labelled request message via CMTransport and wait for a reply.

```
public TibrvMsg sendRequest(TibrvMsg msg, String sendSubject, double
timeout)
```

msg: the request message that needs to be sent
sendSubject: the send subject of the message
timeout: amount of time to wait for the reply

sendRequestAsString ()

Used to send a request string message via RVDTransport, or a labelled request string message via CMTransport and wait for a reply.

```
public TibrvMsg sendRequestAsString(String msg, String fieldName, String
sendSubject, double timeout)
```

msg: the request string message that needs to be sent
fieldName: name of the TibRV field used to send the payload
sendSubject: subject of the message
timeout: amount of time to wait for the reply

sendRequestAsXML ()

Used to send a request XML message via RVDTransport, or a labelled request message via CMTransport and wait for a reply.

```
public TibrvMsg sendRequestAsXML(XmlObject msg, String fieldName,String
sendSubject, double timeout)
```

msg: the request XML message that needs to be sent
fieldName: name of the TibRV field used to send the payload
sendSubject: subject of the message
timeout: amount of time to wait for the reply

sendReply ()

Used to send a reply via RVDTransport, or a labelled reply via CMTransport.

```
public long sendReply(TibrvMsg replyMsg, TibrvMsg sendMsg, double timeout)
```

replyMsg: the reply message
sendMsg: the request message
timeout: time limit for delivery of the message

sendReplyAsString ()

Used to send a string type reply via RVDTransport, or a labelled string type reply via CMTransport.

```
public long sendReplyAsString(TibrvMsg replyMsg, String sendMsg, String
fieldName, double timeout)
```

replyMsg: the reply string message
sendMsg: the request string message
fieldName: the name of the TibRV field used to send the payload
timeout: time limit for delivery of the message

sendReplyAsXML ()

Used to send an XML type reply via RVDTransport, or a labelled XML type reply via CMTransport.

```
public long sendReplyAsXML(TibrvMsg replyMsg, XmlObject sendMsg, String
fieldName, double timeout)
```

replyMsg: the reply XML message
sendMsg: the request XML message
fieldName: the name of the TibRV field used to send the payload
timeout: time limit for delivery of the message

setStringEncoding ()

Used to set the character encoding for converting between Java Unicode strings and wire format strings.

```
void setStringEncoding(java.lang.String encoding) throws
java.io.UnsupportedEncodingException;
```

encoding: determines encoding

Additional Functions for Certified Messaging

You can include the following two functions when using the CMTransport.

onCMMessageReceipt ()

Used to define a callback method to receive confirmation for message sent. It can only be used with **sendMessage** or **sendReply** functions. TIBCO RV control subscribes to two confirmation advisories: `_RV.INFO.RVCM.DELIVERY.COMPLETE.>` and `_RV.ERROR.RVCM.DELIVERY.FAILED.>`.

Note: A TIBCO RV control with certified messaging enabled must have an **onCMReceipt()** method implemented in the process definition. Without this, a runtime exception will be thrown.

```
public void onCMMessageReceipt(byte[] data);
```

data: message data

addListenerForCM ()

Used to pre-register an anticipated listener. When a sending application pre-registers listeners, Rendezvous will store all outbound messages in the sender's ledger. So, when the listener requests certified delivery, it receives the backlogged messages. This function is the same as the `addListener` method in Rendezvous. Refer TIBCO Rendezvous product documentation for more details.

```
void addListenerForCM(String cmName, String subject);
```

cmName: the certified message name
subject: subject of the message

Using the sendMessage Function In a Business Process

The following procedure in an example that describes how to add any TIBCO RV control `sendMessage` function to a business process.

1. Open the **Client Request** node.
2. In the **General Settings** tab, enter a name for the new method.
3. Click **Add**, and select the Java check box in the pop-up dialog.
4. Select **String** from the **Java Types** list and enter a name for the variable in the **Name** field.
5. Click **OK** to add your selection to the **Client Request** node. This represents the message for the `sendMessage` function.
6. Repeat steps 3 to 5 above to add two more variables to the list. The new variables represent the field name and the subject name of the `sendMessage` function.
7. In the **Receive Data** tab, create a new variable for each parameter that you created in the **General Settings** tab of the **Client Request** node. You must provide variable names for all the parameters. The variable type is pre-defined, based on the parameters to which you are assigning the variable.
8. Close the **Client Request** node.
9. Drag and drop the **Perform** node from the **Process Nodes Palette** and convert the message data from String to **`TibrvMsg`** format. See sample code below:

```
// Generating a Tibrv message from the string data format
public void perform() throws Exception
{
   com.tibco.tibrv.TibrvMsg tibrvMsg = new com.tibco.tibrv.TibrvMsg();
   tibrvMsg.update(TibcoField,TibcoMessage);
}
```

10. Drag and drop the `sendMessage` function from the **Controls** tab in the **Data Palette** into your business process, just below the **Client Request** node.
11. Open the **Send Data** tab of the `sendMessage` function node. From the **Select variables to assign** drop-down list, assign the variables that you created in the **Receive Data** tab of the **Client Request** node, to the corresponding parameter of the `sendMessage` function listed in the **Control Expects** column.
12. Open the **Receive Data** tab of the `sendMessage` function note. From the Select variables to assign drop-down list, create a new variable in which to store the sequence number provided by the `sendMessage` function.

You can use similar steps to send messages using the `sendMessageAsString` or the `sendMessageAsXML` functions. Ignore step 9 above as these functions do not require conversion to TibrvMsg format.

Receiving Messages

To receive messages, use the TIBCO RV Event Generator utility. For details, refer to TIBCO Rendezvous Event Generator.

Setting Dynamic Properties

You can change the TIBCO RV control properties dynamically at runtime. The TIBCO RV control properties that you can modify are specified in the `TibRVDynamicPropertiesDocument` type document. This document conforms to the TIBCO RV Control Dynamic Properties schema, which is available in the `jpdpublic.jar` file located in `BEA_Home\wli_10.2\lib`.

The following is an example on how to change properties dynamically.

1. Open the **Client Request** node.
2. In the **General Settings** tab, add a variable of type **`TibRVDynamicPropertiesDocument`**.
3. In the **Receive Data** tab, create a new variable for the parameter that you previously created in the **General Settings** tab by entering a name for the variable. The variable type is already pre-defined based on the parameter to which you are assigning the variable.
4. Drag and drop the **setXMLProperties** function from the **Controls** tab of the **Data Palette**, into your business process.
5. Open the **Send Data** tab of the **setXMLProperties** function node. From the **Select variables to assign** drop-down list, assign the variable that you created in the **Receive Data** tab of the **Client Request** node to the corresponding parameter of the **setXMLProperties** function listed in the **Control Expects** column. All TIBCO RV Control send message operations (following the `setXMLProperties` function in the business process) using the properties you specified in the `TibRVControlDynamicPropertiesDocument`.
6. While executing your business process at runtime, provide the `TibRVControlDynamicPropertiesDocument` as input.

Schema of TIBCO RV Control Dynamic Properties

```
<?xml version="1.0"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.bea.com/wli/control/TibRVControlDynamicProperties.xsd
"
```

```
targetNamespace="http://www.bea.com/wli/control/TibRVControlDynamicPropert
ies.xsd"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">
   <xs:element name="TibRVDynamicProperties">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="service" type="xs:string" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="network" type="xs:string" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="daemon" type="xs:string" minOccurs="0"
maxOccurs="1"/>
      <xs:element name="useCM" type="xs:boolean" minOccurs="0"
maxOccurs="1"/>
      <xs:element name="cmName" type="xs:string" minOccurs="0"
maxOccurs="1"/>
      <xs:element name="ledgerName" type="xs:string" minOccurs="0"
maxOccurs="1"/>
      <xs:element name="requestOld" type="xs:boolean" minOccurs="0"
maxOccurs="1"/>
      <xs:element name="syncLedger" type="xs:boolean" minOccurs="0"
maxOccurs="1"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Sample TIBCO RV Control Dynamic Properties Document

The following is a sample TIBCO RV Control document. You must provide this document at runtime when you execute your business process:

```
<tib:TibRVDynamicProperties>
   <!--Optional:-->
   <tib:service>7500</tib:service>
   <!--Optional:-->
   <tib:network>beaserv1</tib:network>
   <!--Optional:-->
```

```
<tib:daemon>beaserv1:7500</tib:daemon>
<!--Optional:-->
<tib:useCM>true</tib:useCM>
<!--Optional:-->
<tib:cmName>cmname.runtime</tib:cmName>
<!--Optional:-->
<tib:ledgerName>c:/file.txt</tib:ledgerName>
<!--Optional:-->
<tib:requestOld>false</tib:requestOld>
<!--Optional:-->
<tib:syncLedger>false</tib:syncLedger>
</tib:TibRVDynamicProperties>
```

TPM Control

Note: The TPM control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

The TPM (trading partner management) control provides BEA WorkSpace Studio business processes and web services with query (read-only) access to trading partner and service information stored in the TPM repository.

All BEA WorkSpace Studio controls follow a consistent model. Many aspects of using TPM controls are identical or similar to using other BEA WorkSpace Studio controls.

Topics Included in This Section

Example: TPM Control
Provides an example of how to use the TPM control.

Related Topics

Introducing Trading Partner Integration

Trading Partner Management

Interface TPMControl

Overview: TPM Control

The TPM control allows BEA WorkSpace Studio business processes and web services to obtain the following trading partner and service information stored in the TPM repository:

- trading partner by name or business ID
- default trading partner
- basic and extended trading partner properties
- default bindings (ebXML or RosettaNet)
- services, service profiles, and service profile bindings (ebXML, RosettaNet, or web service bindings)

Note: Access to the TPM repository is restricted to active trading partners and active profile services only. To learn about activating trading partners and services, see the WebLogic Integration Administration Console Online Help.

You use methods on the TPM control to retrieve information stored in the TPM repository. These methods return XML documents that conform to the TPM schema associated with importing and exporting trading partner data in the WebLogic Integration Administration Console and the Bulk Loader command line utility.

The TPM control provides read-only access to the TPM repository. Therefore, you cannot use TPM controls to modify trading partner and service information. Instead, you must use the WebLogic Integration Administration Console to modify trading partner and service information. To learn more about modifying the TPM repository, see Trading Partner Management in *Using the WebLogic Integration Administration Console*.

TPM controls cannot initiate transactions. To learn more about transactions in business processes, see Transaction Boundaries.

For initiator business processes that use RosettaNet or ebXML to exchange business messages, you can retrieve certain information from the TPM repository—settings for process time-out, retry count, and retry interval—using methods on the RosettaNet or ebXML control instead of the TPM control. To learn about these methods, see RosettaNet Control and ebXML Control.

Creating a TPM Control

This topic describes how to create a new TPM control.

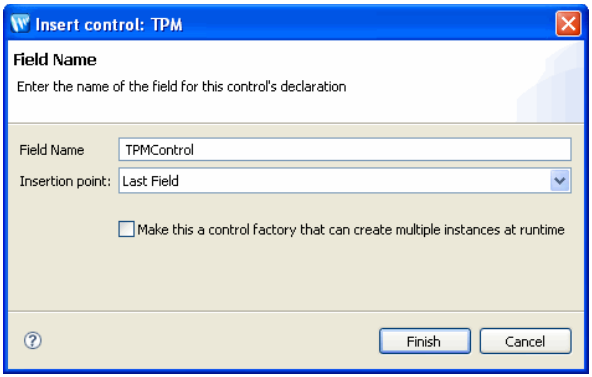
To create a new TPM control

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the TPM control. The business process is displayed in the **Design** view.
2. Click on the **Data Palette** and from the drop-down list select **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Windows > Show View > Data Palette** from the menu bar.
3. Select **TPM**.

   The **Insert control: TPM** dialog box appears (see Figure 15-1).

Figure 15-1 Insert control:TPM



4. In the **Insert control: TPM** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new TPM control instance from your business process. The name you enter must be a valid Java identifier.

– In the **Insertion point**: from the drop-down list select the insertion node, you want to add the control.

– Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

– Click **Finish**.

A TPM control instance is displayed in the **Controls** tab.

Using a TPM Control

After you have added a TPM control to a business process or web service, you can use methods on the control to retrieve information in the TPM repository. For a description of the methods available in the TPM control interface, see the Interface TPM Control.

To use methods in a TPM control

1. Verify that your application contains a schema project that includes the `TPM.xsd` file, and that the schema is already built.
2. In the **Design** view, expand the node for the TPM control in the Data Palette to expose its methods.
3. Drag and drop any methods you want onto the business process.

   Each method you add becomes a **Control Send with Return** node, which will perform a synchronous query request on the TPM repository.
4. Extract the values you want by creating a query (in the XQuery language) using the mapper functionality of WebLogic Workshop. To learn about creating queries with the mapper functionality, see Transforming Data Using XQuery Mapper.

Example: TPM Control

For an example of how to use the TPM Control, see “Step 7: Using the TPM Control and Callbacks” in Tutorial: Building ebXML Solutions, which is located at the following URL:

`http://edocs.bea.com/wli/docs102/tptutorial/ebxml.html`

CHAPTER 16

WLI JMS Control

The WLI JMS control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

JMS (Java Message Service) is a Java API for communicating with messaging systems. Messaging systems are often packaged as products known as Message-Oriented Middleware (MOMs). WebLogic Server includes built in messaging capabilities via WebLogic JMS, but can also work with third-party MOMs. Messaging systems are often used in enterprise applications to communicate with legacy systems, or for communication between business components running in different environments or on different hosts.

The WLI JMS control enables BEA WorkSpace Studio business processes to easily interact with messaging systems that provide a JMS implementation. A specific WLI JMS control is associated with particular facilities of the messaging system. Once a WLI JMS control is defined, business processes may use it like any other WebLogic Workshop control.

The WLI JMS control provides additional features such as RawData message type support, dynamic property configuration, and the ability to control whether to start a new transaction or remain within the calling transaction. You can use the JMS Event Generator to poll for and consume messages produced by the WLI JMS control.

For information on how to add control instances to business processes, see Using Controls in Business Processes.

Topics Included in This Section

Overview: Messaging Systems and JMS
Describes messaging services in general and the Java Message Service in particular

Messaging Scenarios Supported by the WLI JMS Control
Describes appropriate scenarios in which the WLI JMS control may be used.

Messaging Scenarios Not Supported by the WLI JMS Control
Describes scenarios in which the WLI JMS control may not be used.

Creating a WLI JMS Control
Describes how to create and configure a WLI JMS control.

Overview: Messaging Systems and JMS

This topic describes the characteristics of messaging systems that are accessible via JMS (Java Message Service), and therefore via the WLI JMS control.

To learn about the WLI JMS control, see WLI JMS Control.

To learn about specific messaging scenarios that are supported by the WLI JMS control, see Messaging Scenarios Supported by the WLI JMS Control.

Messaging Systems

Messaging systems provide communication between software components. A client of a messaging system can send messages to, and receive messages from, any other client. Each client connects to a messaging server that provides facilities for sending and receiving messages. WebLogic JMS, which is a component of WebLogic Server is an example of a messaging server. WebLogic Server also supports third party messaging systems.

Messaging systems provide distributed communication that is asynchronous. A component sends a message to a destination. A message recipient can retrieve messages from a destination. The sender and receiver do not communicate directly. The sender only knows that a destination exists to which it can send messages, and the receiver also knows there is a destination from which it can receive messages. As long as they agree what message format and what destination to use, the messaging system will manage the actual message delivery.

Messaging systems also may provide reliability. The specific level of reliability is typically configurable on a per-destination or per-client basis, but messaging systems are capable of

guaranteeing that a message will be delivered, and that it will be delivered to each intended recipient exactly once.

JMS supports two basic styles of message-based communications: *point-to-point* and *publish and subscribe*.

JMS Queues for Point-to-Point Messaging

Point-to-point messaging is accomplished with JMS queues. A queue is a specific named resource that is configured in a JMS server.

A JMS client, of which the WLI JMS control is an example, may send messages to a queue or receive messages from a queue. Point-to-point messages have a single consumer. Multiple receivers may listen for messages on the same queue, but once any receiver retrieves a particular message from the queue that message is *consumed* and is no longer available to other potential consumers.

A message consumer acknowledges receipt of every message it receives.

The messaging system will continue attempting to resend a particular message until a predetermined number of retries have been attempted.

JMS Topics for Publish and Subscribe Messaging

Publish and subscribe messaging is accomplished with JMS topics. A topic is a specific named resource that is configured in a JMS server.

A JMS client, of which the WLI JMS control is an example, may publish messages to a topic or subscribe to a topic. Published messages have multiple potential subscribers. All current subscribers to a topic receive all messages published to that topic after the subscription becomes active.

Connection Factories

Before a JMS client can send or receive messages to a queue or topic, it must obtain a connection to the messaging system. This is accomplished via a connection factory. A connection factory is a resource that is configured by the message server administrator. The names of connection factories are stored in a JNDI directory for lookup by clients wishing to make a connection.

There is a default connection factory pre-configured in WebLogic Workshop, named `cgConnectionFactory`. This connection factory is used for all WLI JMS controls that do not

explicitly override it. If you use a connection factory other than the default connection factory, the factory must have the following setting:

```
userTransactionsEnabled="true"
```

Message Components

The components of a JMS message are as follows: a set of standard header fields, a set of application-specific properties, and a message body. Every JMS message contains a standard set of header fields that is included by default and available to message consumers. Some fields can be set by the message producers. The property fields of a message contain header fields added by the sending application. The properties are standard Java name/value pairs. A message body contains the content being delivered from producer to consumer. You can manipulate the content of these components using the following annotations:

JMSHeader Annotation

JMSProperty Annotation

Messaging Scenarios Supported by the WLI JMS Control

This topic describes specific messaging scenarios that are supported by the WLI JMS control.

To learn more about JMS, the Java Message Service, see Overview: Messaging Systems and JMS.

To learn more about the WLI JMS control, see WLI JMS Control.

Supported Messaging Scenarios

The JMS specification supports a wide variety of messaging scenarios. Some scenarios that are possible in standalone applications are not possible in the BEA WorkSpace Studio environment due to the nature of web services.

The messaging scenarios in the following sections are supported by the WLI JMS control. For descriptions of messaging scenarios that are *not* supported by the WLI JMS control, see Messaging Scenarios Not Supported by the WLI JMS Control.

Send Messages to a Queue

A business process, via a WLI JMS control, may send messages to a JMS queue. The business process will not receive a reply. The queue must exist and be registered in the JNDI registry. The administrator who configures the target JMS queue determines the delivery guarantee policies.

To implement this example scenario:

1. On the WLI JMS control, specify the name of the target JMS queue as the value of the `send-jndi-name` attribute of the WLI JMS control's `@com.bea.control.JMSControl.JMS` annotation. Also, specify the `sendtype` attribute as `queue`. To learn how to create a WLI JMS control, see Creating a WLI JMS Control.
2. From your web service, call the WLI JMS control's default method depending on the message type selected when the control was created, or call a custom method you have defined for the WLI JMS control. The default method by message type is as follows:

Table 16-1 Message Type

Message Type	Default Method
Text/XMLBean	`sendTextMessage`
Object	`sendObjectMessage`
Raw Data	`sendBytesMessage`
JMS Message	`sendRawMessage`

Two-Way Messaging with Queues

A business process, via a WLI JMS control, may send messages to one queue and receive reply messages on another queue. A single WLI JMS control may have both send and receive queues configured, and business processes may then send and receive via the same control.

Note: Two-way messaging requires correlation of every received messages with the instance of the business process that sent the original outgoing message. The WLI JMS control ensures that the conversation ID of the sender is sent on the `send_correlation_property` of the outgoing message. To learn more about message correlation, see the explanation of the `send-correlation-property` and `receive-correlation-property` attributes in JMSControl.JMS Annotation.

To implement this example scenario:

1. On the WLI JMS control, specify the name of the JMS queue to which you want to send messages as the value of the `send-jndi-name` attribute of the JMS control's `@` annotation. Also, specify the `send-type` attribute as `queue`.
2. Specify the name of the JMS queue from which you want to receive messages as the value of the `receive-jndi-name` attribute of the WLI JMS control's `@com.bea.control.JMSControl.JMS` annotations. Also, specify the `receivetype` attribute as `queue`.
3. From your web service, call the WLI JMS control's default method depending on the message type selected when the control was created, or call a custom method you have defined for the WLI JMS control. The default method by message type is as follows:

Table 16-2 Message Type (Two-Way Messaging)

Message Type	Default Method
Text/XMLBean	`sendTextMessage`
Object	`sendObjectMessage`
Raw Data	`sendBytesMessage`
JMS Message	`sendRawMessage`

4. To be notified when messages are received on the receive queue, implement a callback handler for the WLI JMS control's callback (`receiveTextMessage`, `receiveBytesMessage`, `receiveObjectMessage` or `receiveRawMessage` depending on the message type selected when the control was created); or a custom callback you have defined for the WLI JMS control.

Publish to a Topic

A business process, via a WLI JMS control, may publish messages to a JMS topic. The business process will not receive a reply. The topic must exist and be registered in the JNDI registry.

To implement this example scenario:

1. On the WLI JMS control, specify the name of the target JMS topic as the value of the `send-jndi-name` attribute of the WLI JMS control's `@com.bea.control.JMSControl.JMS` annotation. Also, specify the `sendtype` attribute as `topic`.

2. From your business process, call the WLI JMS control's default method (`sendTextMessage`, `sendBytesMessage`, `sendObjectMessage` or `sendRawMessage` depending on the message type selected when the control was created); or a custom method you have defined for the WLI JMS control.

Subscribe to a Topic

A business process, via a WLI JMS control, may subscribe to messages on a JMS topic. The topic must exist and be registered in the JNDI registry. Only messages sent after the business process has subscribed to the topic will be received.

To implement this example scenario:

1. On the WLI JMS control, specify the name of the target JMS topic as the value of the `receivejndiname` attribute of the WLI JMS control's `@com.bea.control.JMSControl.JMS` annotation. Also, specify the `receivetype` attribute as `topic`.
2. From your business process, call the WLI JMS control's subscribe method.
3. To be notified when messages are received on the receive topic, implement a callback handler for the WLI JMS control's callback (`receiveTextMessage`, `receiveBytesMessage`, `receiveObjectMessage` or `receiveRawMessage` depending on the message type selected when the control was created); or a custom callback you have defined for the WLI JMS control.
4. To stop being notified when messages are received on the receive topic, call the WLI JMS control's unsubscribe method.

Messaging Scenarios Not Supported by the WLI JMS Control

This topic describes specific messaging scenarios that are not supported by the WLI JMS control.

To learn more about the WLI JMS control, see WLI JMS Control.

Unsupported Scenarios

The JMS specification supports a wide variety of messaging scenarios. Some scenarios that are possible in standalone applications are not possible in the BEA WorkSpace Studio environment due to the nature of web services.

The messaging scenarios in the following section are *not* supported by the WLI JMS control. For descriptions of messaging scenarios that are supported by the WLI JMS control, see Messaging Scenarios Supported by the WLI JMS Control.

Receive Unsolicited Messages from a Queue

A business process may not, via a WLI JMS control, specify a receive queue and subsequently receive unsolicited messages from that queue.

A business process must be performing work on behalf of a specific client and, in asynchronous situations, as part of a specific conversation. When an unsolicited messages is received from a queue, it is not possible for the WLI JMS control to determine the appropriate conversation or client with which to correlate unsolicited incoming messages.

Note: You may receive unsolicited messages in a business process via the JMS Event Generator and the Message Broker capabilities. To learn how to use the Message Broker controls and the JMS Event Generator, see Message Broker Controls.

Creating a WLI JMS Control

This topic describes how to create a new WLI JMS control.

To learn about WLI JMS controls, see WLI JMS Control.

Creating a New WLI JMS Control

You can create a new WLI JMS control and add it to your business process. To define a new WLI JMS control:

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the WLI JMS control. The business process is displayed in the **Design** view.

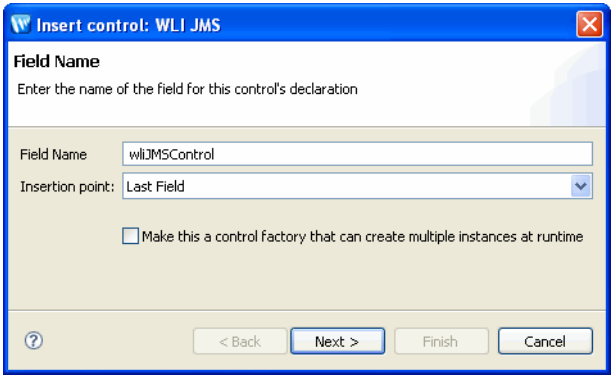
2. Click on the Data Palette and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **WLI JMS**.

   The **Insert control: WLI JMS** dialog box appears (see Figure 16-1).

Figure 16-1 Insert Control: WLI JMS

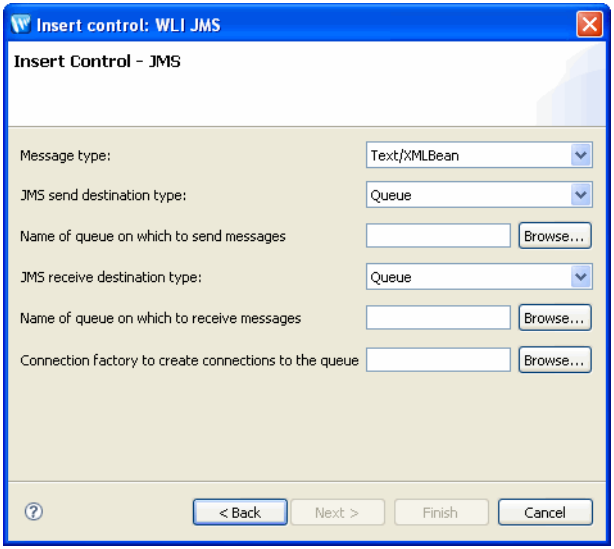


4. In the **Insert control: WLI JMS** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new WLI JMS control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

     The **Create Control** dialog-box appears.
5. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.
   - Click **Next**.

     The **Insert control- JMS** dialog-box appears (see Figure 16-2).

Figure 16-2 Insert Control - JMS



6. In the **Insert control- JMS** dialog box enter the following details:
   - In the **Message type** drop-down list, select the type of message you want to process. For more information about the types of messages, see Specifying the Format of The Message Body.
   - In the **JMS send destination type** drop-down list, select either **Queue** or **Topic**, depending on the kind of messaging service you will be connecting to. For more information about messaging services, see Overview: Messaging Systems and JMS.
   - In the **Name of Queue on which to send message** field, type the name of the queue or topic that will send messages. If you do not know the name, click **Browse** and choose from the available list. You must specify the name of the send queue if the control is to be used to send messages.
   - In the **JMS receive destination type** drop-down list, select either **Queue** or **Topic**, depending on the kind of messaging service you will be connecting to. For more information about messaging services, see Overview: Messaging Systems and JMS.
   - In the **Name of queue on which to receive messages** field, type the name of the queue or topic that will receive messages. If you do not know the name, click **Browse** and choose from the available list. You must specify the name of the receive queue if the control is to be used to receive messages.

– In the **connection-factory to create connections to the queue** field, type the name of the connection factory to create connections to the queue or topic. If you do not know the name, click **Browse** and choose from the available list.

– Click **Finish**.

WLI JMS Control Methods

To learn about the methods available on the WLI JMS control, see the Interface WliJMSControl.

The Java File for a WLI JMS Control

When you create a new WLI JMS control, you create a new Java file in your project. The following is an example Java file:

```
package requestquote;

@org.apache.beehive.controls.api.bean.ControlExtension

@com.bea.control.JMSControl.JMS(sendtype =
com.bea.control.JMSControl.Type.queue, sendjndiname = "l", receivetype =
com.bea.control.JMSControl.Type.queue, receivejndiname = "l",
connectionfactoryjndiname = "o")

public interface WLIJMSCntrl extends com.bea.control.WliJMSControl {


       /**
        * this method will send a javax.jms.TextMessage to sendJndiName
        */
       public void sendTextMessage(String payload);


       /**
        * If your control specifies receiveJndiName, that is your JWS
expects to receive
        * messages from this control, you will need to implement callback
handlers.
        */
       @org.apache.beehive.controls.api.events.EventSet(unicast = true)
```

```
        interface Callback extends com.bea.control.WliJMSControl.Callback {
               /**
                * Define only 1 callback method here.
                *

                * This method defines a callback that can handle text
messages from receiveJndiName
                */
               public void receiveTextMessage(String payload);

        }
        static final long serialVersionUID = 1L;

        public WLIJMSCntrl create();
}
```

The Java file contains the declaration of a Java interface with the name specified in the dialog. The interface extends the control base interface. Invoking any method in the Java interface, other than the callback, results in a JMS message being sent to the specified queue or topic.

The contents of the WLI JMS control's Java file depend on the selections made in the Insert WLI JMS dialog. The example above was generated in response to selection of **Text/XML Bean** as the **Message type** drop-down list.

Configuring the Properties of a JMS Control

Most aspects of a WLI JMS control can be configured from the JPD Configuration pane in Design view. These properties are encoded in the JMS control's Java file as attributes of the `@com.bea.control.JMSControl.JMS` annotations. To retrieve current parameter settings, use the `getControlProperties()` method (this is a different method from the `getProperties()` method on the base JMS control which is used to get the JMS properties of the last message received.)

For more information about the `JMSControl:JMS` annotation and its attributes, see JMSControl:JMS Annotation.

Two queues are configured when BEA WorkSpace Studio is installed, in order to support WLI JMS control samples. These are named `SimpleJmsQ` and `CustomJmsCtlQ`. The connection factory that provides connections to these queues has the JNDI name `weblogic.jws.jms.QueueConnectionFactory`. These resources may be used for experimentation.

Note: Every WLI JMS control deployed on a server should listen on a unique queue. If multiple WLI JMS controls on the same server are simultaneously listening on the same queue, the results may be unpredictable, for more information see "WLI JMS Control Caveats".

Specifying the Format of The Message Body

Within a WLI JMS control, you may define multiple methods and one callback. All methods will send or publish to the queue or topic named by `send-jndi-name`, if present.

JMS defines several message types that may be sent and or published. The WLI JMS control can send the JMS message types `TextMessage`, `ObjectMessage`, `BytesMessage`, and `JMSMessage`. The WLI JMS control dynamically determines which type of message to send based on the configuration of the WLI JMS control method that was called. XML Object and XML typed variables use the `text/XMLBean` message type.

Note: You can send or receive any message type through send and receive methods that take a `javax.jms.Message` argument. (All message types extend `javax.jms.Message`.) To send an `ObjectMessage`, for example, call `myControl.getSession()` to get the JMS session, then call `session.createObjectMessage()`, and then send the message.

If the WLI JMS control method takes a single String or XMLObject argument, a `javax.jms.TextMessage` is sent.

If the WLI JMS control method takes a single argument of type `java.lang.Object`, a `javax.jms.ObjectMessage` is sent.

If the WLI JMS control method takes a single argument of type `javax.jms.BytesMessage`, a `javax.jms.BytesMessage` is sent.

If the WLI JMS control method takes a single argument of type `javax.jms.Message`, a JMS Message object is sent directly.

Specifying Message Headers and Properties

To edit the parameter list controlling the message headers and message properties, display the control in the Design view, select a method, and edit the parameters using the JPD Configuration pane. You can set parameters programatically using the `setProperties()` method. To display current parameter settings, use the `getControlProperties()` method.

You can send additional properties using key-values pairs, using the annotation `@com.bea.control.JMSControl.JMS -property` for each pair. You can also edit the parameters directly in the Source view.

Accessing Remote JMS Resources

The JNDI names specified for `send-jndi-name`, `receive-jndi-name` and `connection-factory` may refer to remote JMS resources. The fully specified form of a JMS resource names is:

```
jms:{provider-host}/{factory-resource}/
{dest-resource}?{provider-parameters}
```

For example:

```
jms://host:7001/cg.jms.QueueConnectionFactory/
jws.MyQueue?URI=/drt/Bank.jws
```

or:

```
jms://host:7001/MyProviderConnFactory/
MyQueue?SECURITY_PRINCIPAL=foo&SECURITY_CREDENTIALS=bar
```

WLI JMS Control Caveats

Bear in mind the following caveats when you work with WLI JMS controls:

- If you have multiple web services (multiple types, not instances) that reference the same `receive-jndi-name` for a queue, you must use the `receive-selector` attribute such that the web services partition all received messages into disjoint sets. If this is not handled properly, messages for a particular conversation may be sent to a control instance that does not participate in that conversation. Note that if you rename a web service that uses a JMS control without undeploying the initial version, the initial version and the new version will be using an identically configured WLI JMS control and will violate this caveat.

- You may have only one callback defined for any WLI JMS control instance (`receiveTextMessage`, `receiveBytesMessage`, `receiveObjectMessage` or `receiveJMSMesage`, or a developer-defined callback).
- Note the difference between the `getControlProperties()` method used to get WLI JMS control properties and the `getProperties()` method on the base JMS control which is used to get the JMS properties of the last message received.
- If the underlying WLI JMS control infrastructure receives a message that it cannot deliver to a control instance (e.g. no conversation ID for a control that listens to a queue), it will throw an exception from the `JMSControl.onMessage` method. This will cause the current transaction to be rolled back. The behavior after that depends on how the administrator set up the JMS destination. Ideally, it should be set up to have a small retry count and an error destination.

Note: If the destination is configured with a large (or no) retry count and no error destination, the WLI JMS control infrastructure will continue attempting to process the the message (unsuccessfully) forever. For information on setting the redelivery limit, see “ Programming WebLogic JMS”.

Worklist Controls

WebLogic Integration Worklist provides the capability to direct the flow of work and manage the routing of tasks to the people in an enterprise. Integral to the flow of work are actions such as receiving, approving, modifying, and routing documents. The documents that accompany work activities provide the information necessary for people to perform and complete tasks. The Worklist enables people to collaborate in business processes including assigning tasks, tracking the status of tasks, handling approvals, and other activities required to manage workflow.

To support the Worklist functionality, WebLogic Integration provides two controls in BEA WorkSpace Studio, the Task control and the Task Batch control. These controls expose Java interfaces that can be invoked directly from your business processes. The Task control enables a business process to create a single Task instance, manage its state and data, and provide callback methods that report status. The Task Worker control allows specified users to acquire ownership of Tasks, work on them, and complete them. It also provides administrative privileges, such as starting, stopping, deleting, and assigning. Access to the Task Worker control can be done with a business process or through a user interface (UI).

Topics Included in This Section

Overview: Worklist Controls

Describes what Tasks are and provides an overview of the Worklist controls.

Creating a New Task Control
Describes how to create a new Task control using the BEA WorkSpace Studio graphical design interface.

Creating a New Task Batch Control
Describes how to create a new Task Worker control using the BEA WorkSpace Studio graphical design interface.

Using Task and Task Batch Controls in Business Processes
Provides information about using the Worklist controls in business processes.

Example: Task Control
Provides a link to the *Tutorial: Building a Worklist Application*, which shows an example of using a Task Control.

Related Topics

Interface TaskControl

Interface TaskWorkerControl

Control Annotations

Using the Worklist

Tutorial: Building a Worklist Application

Overview: Worklist Controls

Worklist controls enable the automated manipulation, creation, and management of Tasks. A Task instance represents a unit of work that requires completion within a certain time period. After the work is completed, you can use a Task instance to represent a detailed record of that unit of work.

A Task instance is a particular object in the run-time Worklist system that represents a work assignment in the real world. Task instances are part of the WebLogic Integration server and exist independently of any controls or business processes. Multiple business processes can interact with a Task throughout its lifecycle concurrently. Tasks remain in the run time indefinitely, either until they are explicitly deleted or purged by the WebLogic Integration purging process. You can create, delete, and manage Tasks through the following mechanisms:

- The Task and Task Batch controls in BEA WorkSpace Studio
- The Worklist area of the WebLogic Integration Administration Console

- The public Worklist API, using Enterprise Java Beans, and Message Beans

Task instances, or simply *Tasks*, offer a variety of properties that describe the work to be done and the *state* of the work. Task instance properties can describe the following:

Table 17-1 Task Instance Properties

Property	Description
Assignees List	The list of users and groups that have permission to claim the task and work on it.
Completion Due Date	The date the work is due.
Task Owner	The user who manages the process of getting the work done.
Claimant	The user who has claimed the Task and completes the work.
Request and response documents	The records that describe the work to be done and the results.

Tasks have the following characteristics, qualities and behaviors that can be defined, configured or used:

Table 17-2 Task Characteristics

Characteristics	Description
Task Due Dates	Due dates can be set to track how long it should take for a Task to get claimed by a user or for the claimant to actually complete the task. Due dates can be set with actual dates, or using business time with a business calendar.
Task States	States can describe such things as whether a Task is complete, started, or aborted.
Task Operations	Tasks depend on users to invoke *operations* that make changes to properties and states. For example, an operation could indicate that a Task is complete or to assign a Task to a new user.

The following Worklist controls are provided for building a Worklist system with WebLogic Integration:

- **Task Control**—creates a single Task instance, manages its state and data, and provides callback methods to report status of the Task. Each Task control operates on a single active Task instance.
- **Task Batch Control**—assumes ownership of Tasks, works on them, completes them, and provides administrative privileges—starting, stopping, deleting, and assigning, among other functions. Task Worker controls allow operations upon several Task instances at the same time.

Worklist controls are extensible. Common extensions include implementing callback functions and performing system queries. Extensibility is provided by Java annotations.

Creating a New Task Control

An instance of a Task control can create a single task instance. If multiple tasks need to be created, use a factory type of Task control. See "Using Task Control Factories" in Advanced Topics in *Using the Worklist Tutorial*, which is located at the following URL:

`http://edocs.bea.com/wli/docs102/worklisttutorial/customuipf.html`

A Task control instance can also interact with a task instance that already exists by setting its *active task ID*. After creating or setting the active task ID, your control instance can get information about that task or update that task in various ways.

You can customize Task controls for different business purposes, by adding new operations or callbacks, or by altering the signatures of existing operations or callbacks.

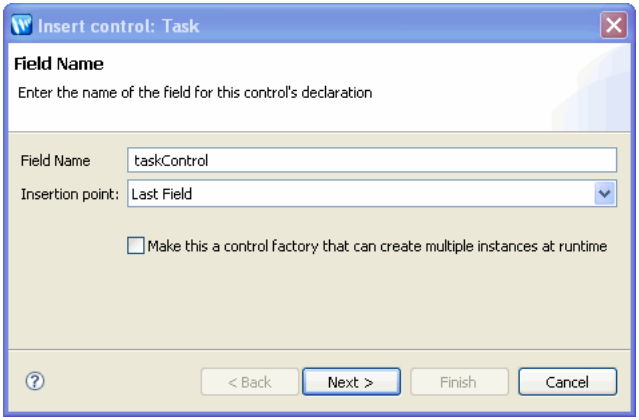
To create a new Task control

1. Open your WebLogic Integration application in BEA WorkSpace Studio.
2. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the logic to integrate business users using the Worklist system. The business process is displayed in the **Design** view.
3. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.
4. Select **Task**.

   The **Insert control: Task** dialog box appears (see Figure 17-1).

Figure 17-1 Insert Control:Task



5. In the **Insert control: Task** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new Task control instance from your business process. The name you enter must be a valid Java identifier.
   - In the Insertion point: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Choose whether you want to make this a control factory by selecting or clearing the **Make this a control factory that can create multiple instances at runtime** check box.
   - Click **Next**.

     The **Create Control** dialog-box appears.
6. In the **Create Control** dialog box enter the following details:
   - In the **Name** field, type the name of your new control extension file.
   - Click **Next**.

     The **Insert control- Task** dialog-box appears.
7. In the **Insert control- Task** dialog box enter the following details:
   - Enter a filename for the Task control in the **Task Plan** field, by clicking **Browse** to find the Java file in your file system and click **OK**.
   - Click **Finish**.

A new Task control and an instance of it are created and the **Insert Control** dialog box is closed.

A new Java file is created and displayed in the **Package Explorer** pane in BEA WorkSpace Studio. (You can double-click any Java file to view or edit it in the **Design** or **Source** view.) The instance of the control is displayed on the Controls tab of the **Data Palette**.

8. To display the base methods provided on a Task control, expand the control instance by clicking the + beside its name on the Data Palette.
9. After you create an instance of the Task control in your business process, you can design the interaction of the business process with the Task control by simply dragging and dropping the Task control methods from the Data Palette onto the Design view at the point in your business process at which you want to design the interaction.

   For examples of designing interactions between a business process and an instance of a Task control, see Using Task and Task Batch Controls in Business Processes.
10. After you create a Task control in your business process, you can view and edit the properties of the control type or the instance of that control type in the **JPD Configuration** pane. The control type is represented as a java file in the **Package Explorer** pane and the instance is represented in the **Data Palette**.

    Task Instances have data values associated with them, many of which are set when the task is created. You can use the **JPD Configuration** pane on a Task control to set the default values for some of these data values. These values are used whenever that control instance creates a new task. Note that the properties set on a factory type Task control propagate to any Task control instances created from that factory.

    Note: To learn how to use the JPD Configuration and Properties pane for specifying properties for control types versus control instances, see Interacting With Resources Using Controls.

Creating a New Task Batch Control

The Task Worker control allows specified users to acquire ownership of Tasks, work on them, and complete them. It also provides administrative privileges, such as starting, stopping, deleting, and assigning. Access to the Task Worker control can be done with a business process or through a user interface (UI). You can customize each Task worker control for different business purposes.

This topic describes how to create a new Task Worker control. Task Worker controls do not have any properties to configure.

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the logic to integrate business users using the Worklist system. The business process is displayed in the **Design** view.

2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window > Show View > Data Palette** from the menu bar.

3. Select **Task Batch**.

   The **Insert control: Task Batch** dialog box appears.

4. In the **Insert control: Task Batch** dialog box enter the following details:

   – In the **Field Name**, type the variable name used to access the new Task Batch control instance from your business process. The name you enter must be a valid Java identifier.

   – In the **Insertion point**: from the drop-down list select the insertion node, you want to add the control.

   – Decide whether you want to make this a control factory and select or clear the **Make this a control factory that can create multiple instances at runtime** check box.

   – Click **Next**.

   The **Create Control** dialog-box appears.

5. In the **Create Control** dialog box enter the following details:

   – In the **Name** field, type the name of your new control extension file.

   – Decide whether you want to add comments as configured in the properties of the current project and select or clear the **Generate comments** check box.

   – Click **Next**.

   The **Insert control: Task Batch** dialog-box appears.

6. In the **Insert control: Task Batch** dialog box, select the Task Plan.

7. Click **Finish.**

   When you click finish, the control java file is displayed in the **Package Explorer** pane. In both **Design** and **Source** view, you can double-click any control java file to view or edit it. The instance of the control is displayed on the Controls tab of the Data Palette.

8. To display the base methods provided for the control instance, click the + beside its name on the **Data Palette**. The following figure shows an example of a Task Worker control instance displayed on the **Controls** tab in the **Data Palette**.
9. After you create an instance of the Task control in your business process, you can design the interaction of the business process with the Task control by simply dragging and dropping the Task control methods from the Data Palette onto the Design view at the point in your business process at which you want to design the interaction.

   For examples of designing interactions between a business process and an instance of a Task control, see Using Task and Task Batch Controls in Business Processes.

Using Task and Task Batch Controls in Business Processes

Before you begin working with the Task and Task Batch controls, you should be familiar with the features and components of the Worklist. To learn more about the Worklist, see *Using the Worklist*.

To design the interaction of a Task or Task Worker control with a business process, you must decide which methods on the control you want to call from the business process to support the business logic.

In the same way that you design the interactions between business processes and other controls in the BEA WorkSpace Studio, you can bind the Worklist control method to the appropriate control node in your business process (**Control Send**, **Control Receive**, and **Control Send with Return**). You do this in the **Design** view by simply dragging a control method from the **Data Palette** onto the business process at the point in your business process at which you want to design the logic.

When you create task control, out of user properties like `Integer` and `Float` in the task plan, the user properties argument become `long` and `double instead of Integer` and `Float`, this change in input will not lead to data loss.

Example: Task Control

To see an example of using a Task control in a business process, see *Tutorial: Building a Worklist Application*, which is located at the following URL:

`http://edocs.bea.com/wli/docs102/worklisttutorial/index.html`

WLI Timer Control

Note: The WLI Timer Control l is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

A Timer Control notifies your application, when a specified period of time has elapsed or when a specific absolute time has been reached.

Topics Included in this Section

Overview: WLI Timer Control

Some transactions and events require a certain amount of time to complete. Others can run indefinitely if not aborted, and eat up resources. Still others must occur at a specific time. The Timer control provides the developer with a way to respond from code when a specified interval of time has elapsed or when a specified absolute time has been reached.

Creating a WLI Timer Control

This topic describes how to create a WLI Timer control.

To create a WLI Timer control:

1. In the **Package Explorer** pane, double-click the business process to which you want to add the WLI Timer control. The business process is displayed in the **Design** view.
2. Click on the **Data Palette** and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA Workshop for WebLogic Platform, click **Window** > **Show View** > **Data Palette** from the menu bar.
3. Select **WLI Timer**.

   The WLI Timer control is created and appears in the **Data Palette**.

Using a WLI Timer Control

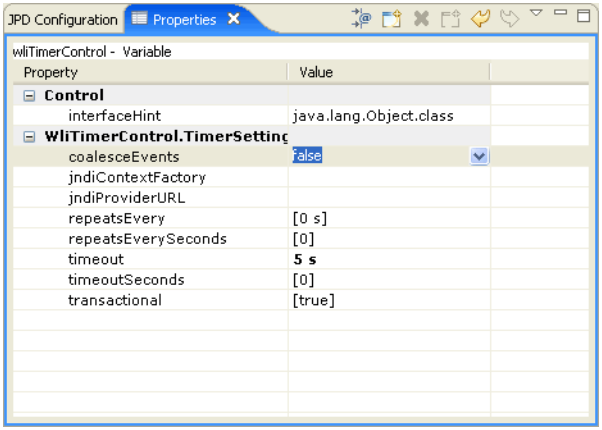
A Timer control notifies your application when a specified period of time has elapsed or when a specified absolute time has been reached. For example, you can use a Timer control to run a process at certain intervals throughout the day, or to cancel an operation that is taking too long.

The following sections describe how to configure the Timer control.

Setting Default Timer Control Behavior

You can specify the behavior of a Timer control in **Design** view by setting the control's timeout and repeatsEvery properties in the **Properties** pane. A Timer control instance displays the following annotations as shown in Figure 18-1.

Figure 18-1 WLI JMS Properties Pane



These annotations correspond to attributes of the `com.bea.control.WliTimerControl.TimerSettings`, which identifies the Timer control in your code. The `com.bea.control.WliTimerControl.TimerSettings` has the following attributes:

- `coalesceEvents`- It specifies how the Timer control should behave if delivery of its events is delayed.
- `jndiContextFactory`- It specifies the JNDI context factory class.
- `jndiProviderURL`- It specifies the JNDI provider URL.
- `repeatsEvery`- It specifies how often the Timer control should fire after the first time
- `repeatsEverySeconds`- It specifies the time in seconds until the timer control fires the first time, once started (default: 0).
- `timeout`- It specifies the time until the Timer control fires the first time, once started and the default value is `0 seconds`.
- `timeoutSeconds`- It specifies the time in seconds until the timer control fires the first time, once started. The default value is `0`.
- `transaction`- It specifies the timer's participation in a transaction, the default value is `true`.

.

You can set these attributes to specify relative time. Relative time is an interval of time in relation to the present, such as three hours from now. You can also specify that the Timer control fire at an absolute time, such as 3:00 AM, by calling the Timer control's setTimeoutAt method. For more information see Interface WLiTimerControl.

Specifying Time on a WLI Timer Control

This topic describes how to specify relative and absolute time values for a Timer control. A relative time value specifies an interval of time that is relative to the present, such as next Thursday or twenty minutes from now. An absolute time value specifies time according to the clock. For example, 5:00 PM is an absolute time.

Specifying Relative Time

You specify relative time when you want an event to fire at a time relative to the present. For example, you might want to wait only 5 minutes for another system to respond. So your application sets a timer with a relative timeout of 5 minutes and starts the timer. If the timer expires (that is, calls its callback) before the system responds, the web service stops waiting and continues with other operations.

To specify that the timer fires after an interval, set the timeout and, optionally, the repeats-every attribute of the Timer control. You can set these attributes in the Properties pane or in source code, or by calling the setTimeout or setRepeatsEvery methods of the TimerControl interface.

When relative time is expressed as a text string, it is formatted as integers followed by case-insensitive time units. These time units can be separated by spaces. For example, the following code sample is a valid duration specification that exercises all the time units, spelled out fully:

```
/**
 * org.apache.beehive.controls.api.bean.Control
 * com.bea.control.WliTimerControl.TimerSettings ="99 years 11 months 13
days 23 hours 43 minutes 51 seconds"
 */
Timer almostCentury;
```

This example creates a Timer control whose default initial firing will occur in almost 100 years.

Units may also be truncated. For example, valid truncations of "months" are "month", "mont", "mon", "mo", and "m". If both months and minutes are specified, use long enough abbreviations to be unambiguous.

The string "p" (case insensitive) is allowed at the beginning of a text string. If it is present, then single-letter abbreviations and no spaces must be used and parts must appear in the order y m d h m s.

The following Timer control declaration is equivalent to the previous example, but uses the fully truncated form:

```
/*
 * @control
 * @timer timeout="P99Y11Mo13D23H43M51S"
 */
Timer almostCentury;
```

Durations are computed according to Gregorian calendar rules, so if today is the 17th of the month, 3 months from now is also the 17th of the month. If the target month is shorter and doesn't have a corresponding day (for example, no February 31), then the closest day in the same month is used (for example, February 29 on a leap year).

Specify Absolute Time

Absolute time is useful when you know the exact moment you want operations to begin and end. For example, your application can have your web service send a reminder email to remind you that someone's birthday is coming up.

You can configure a Timer control to fire at an absolute time by calling the setTimeoutAt method of the `TimerControl interface`.

The `setTimeoutAt` method configures the timer to fire an event as soon as possible on or after the supplied absolute time. If you supply an absolute time in the past, the timer will fire as soon as possible.

If `setTimeoutAt` is called within a transaction, its effect (any work performed in the callback handler) is rolled back if the transaction is rolled back, and its effect is committed only when the transaction is committed.

If `setTimeoutAt` is called while the timer is already running, it will have no effect until the timer is stopped and restarted.

The `setTimeoutAt` method takes as its argument a `java.util.Date` object. Other Java classes that are useful when dealing with Date are `java.util.GregorianCalendar` and `java.text.SimpleDateFormat`.

The `getTimeoutAt` method returns the time at which the timer is next scheduled to fire, if the repeats-every attribute is set to a value greater than zero. If the repeats-every attribute is set to zero, then the `getTimeoutAt method` returns the value set by the `setTimeoutAt` method or the value set in the timeout attribute. If you call the `getTimeoutAt` method from within the onTimeout callback handler, the first timeout has already fired, so `getTimeoutAt` will return either the time of the next timeout or the time of the first timeout if the timer is not set to repeat.

The following example calls the setTimeoutAt method to specify that the first timeout fires at thirty seconds past the current minute, then calls the setRepeatsEvery method to specify that the timer subsequently fires every sixty seconds. The onTimeout event provides information about the Timer control's firing.

```
/**
 * @common:operation
 * @jws:conversation phase="start"
 */
public void StartTimer()
{
    Calendar cd = new GregorianCalendar();
    cd.set(cd.SECOND, 30);
    tTimer.setTimeoutAt(cd.getTime());
    tTimer.setRepeatsEvery(60);
    tTimer.start();
}
public void tTimer_onTimeout(long time)
{
    callback.FireTimeout("The timer was scheduled to fire at: " + new
Date(time)
```

```
    + ". The current time is: " + new Date()
    + ". The timer will fire again at: " + tTimer.getTimeoutAt());
}
```

XML MetaData Cache Control

Note: The XML MetaData Cache Control is available in BEA WorkSpace Studio only if you are licensed to use WebLogic Integration.

The XML MetaData Cache is managed using the WebLogic Integration Administration Console or the MBean API, which allows users to create their own NetUI based consoles. This control can be used to retrieve XML metadata that is present in the XML MetaData Cache.

Topics Included in This Section

Using the XML MetaData Cache Control in a Business Process
Describes how to use the WebLogic Integration in a business process.

Example: XML MetaData Cache Control
Describes a scenario in which the WebLogic Integration is used.

Overview: XML MetaData Cache Control

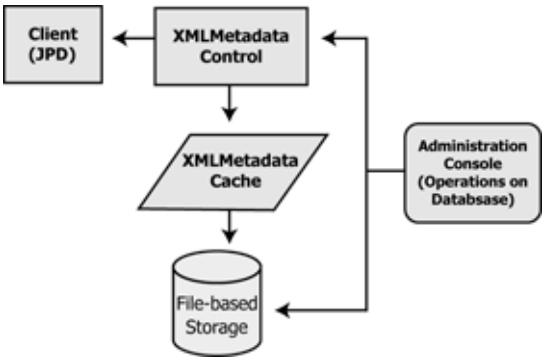
The XML MetaData Cache Control is used for fast access to a managed set of key-value pairs. The keys are of type string and the value contains XML data. For more information on source that uses the WebLogic Integration, see “Code Samples: Weblogic Integration” in the WebLogic Integration Code Library, available on dev2dev, which is available at the following URL: http://dev2dev.bea.com/code/wli.jsp.

Using the XML MetaData Cache Control in your business process flow you can retrieve XML metadata from the XML MetaData Cache. The WebLogic Integration Administration Console manages the entries in the cache (add, delete, update). Alternatively, a custom application (such as a NetUI application) can be written to manage the cache. For more information, see Sharing Cache Data Within a Cluster.

The XML MetaData Cache Control is intended to be used in a read-mostly environment. The XML MetaData Cache should only be used for configuration metadata. It is used to cache runtime xml data. Updating cache entries is expensive, as all cache entries in a cluster must be updated, where as read operations are always in-memory operations. The size of an XML value in the cache should typically be less than 100k bytes. Larger sizes will work, but with an increasing cost of updates. The XML MetaData Cache is a global, domain-wide cache. Data from the cache is made available on a permanent basis through file-based storage.

Figure 19-1 describes the XML MetaData Cache and WebLogic Integration implementation.

Figure 19-1 WebLogic Integration Implementation Overview



Sharing Cache Data Within a Cluster

You can add XML metadata to the cache only through the node on which the Administration Server is running. The Administration Server receives the input, and stores the key and the associated XML metadata in a file, in the following format:

`XMLMetatadaCache_<keyname>_.xml`

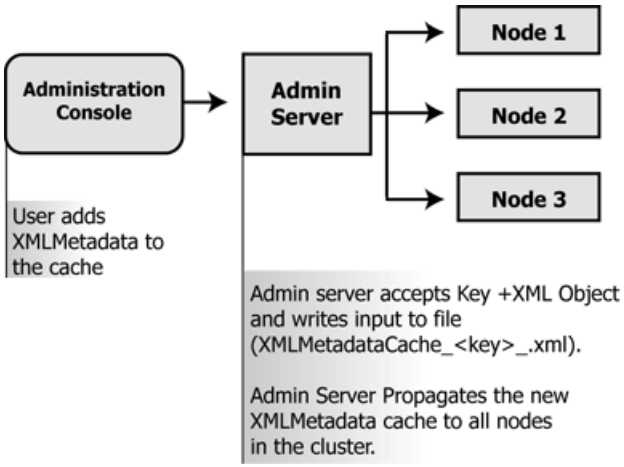
For each XML document that is added to the cache, a new XML MetaData Cache file is created. Once the file is created, the newly added XML document is propagated to all the nodes within the cluster. This ensures that the data is immediately available to any requesting node.

When using the XML MetaData Cache Control in a cluster, if a server on the cluster is restarted and a value changes while the server is offline, the server will receive the change notification when it comes online again.

Note: You cannot modify the XML MetaData Cache when the Administration Server is down.

Figure 19-2 describes how data is shared within a cluster.

Figure 19-2 Sharing Data Within a Cluster



Creating an XML MetaData Cache Control

You can create a new XML MetaData Cache Control and add it to your business process. To create a new XML MetaData Cache Control:

1. In the **Package Explorer** pane, double-click the business process (Process.java file) to which you want to add the XML MetaData Cache control. The business process is displayed in the **Design** view.

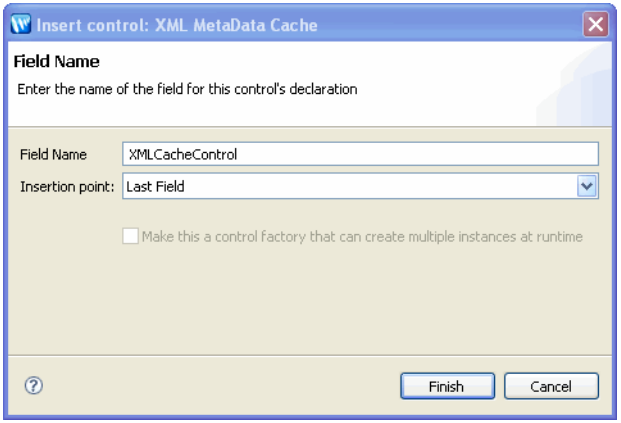
2. Click on the Data Palette and from the drop-down list choose **Integration Controls** to display the list of controls used for integrating applications.

   Note: If the Data Palette view is not visible in BEA WorkSpace Studio, click **Window** > **Show View** > **Data Palette** from the menu bar.

3. Select **XML MetaData Cache**.

   The **Insert Control: XML MetaData Cache** dialog box appears (see Figure 19-1).

Figure 19-3 Insert control: XML MetaData Cache



4. In the **Insert Control: XML MetaData Cache** dialog box enter the following details:
   - In the **Field Name**, type the variable name used to access the new XML MetaData Cache control instance from your business process. The name you enter must be a valid Java identifier.
   - In the **Insertion point**: from the drop-down list select the point where you want the field name to be inserted in the process file.
   - Click **Finish**.

   Note: **Make this a control factory that can create multiple instances at runtime** option is not available for XML MetaData Cache Control, and the option is disabled.

Understanding the XML MetaData Cache Control Get Method

The XML MetaData Cache Control includes the following method:

```
XmlObject get(String key)
```

Using this method, the `get` command uses the key of type `String` to access the XML Metadata in the cache. For more information on how to use this method, see Using the XML MetaData Cache Control in a Business Process.

You can get a document from the cache only if you previously added it to the cache using the Administration Console. For more information on adding an XML document to the cache, see XML Cache.

Using the XML MetaData Cache Control in a Business Process

Once created, you can use the new XML MetaData Cache Control in a business process.

The business process usually starts with a Client Request node. This node represents a request made by a client to the process. In this case, the client invokes the `get(String key)` method on the process to get an XML metadata key from the cache.

To set up your business process to get XML metadata from the cache, do the following:

- Create an instance of the WebLogic Integration. Name it **My**WebLogic Integration. Use the steps provided in Creating an XML MetaData Cache Control.
- Your new XML MetaData Cache Control will be visible under the **Controls** tab in the **Data Palette**. Expand the XML MetaData Cache Control you have created, to view the XMLMetadata `get` method that you can use in your business process.
- Design a Control Send Node in your business process, to prepare the control to get cache data from the cache. For more information on how to do this, see To Prepare the Client Node.

To Prepare the Client Node

1. Open the **Client Request** node. In the General Settings tab, enter **NewXMLMetadata** in the **Method Name** field.
2. Click **Add**, then select **String** from the Java datatype list. Enter *Key1* as the variable in the **Name** field, then click **OK** to add your selection to the Client Request node. This indicates to the client that the control expects the client to send and receive a value of type string. The String represents the datatype used to define the key with which the XML metadata is associated.
3. In the **Receive Data tab**, select a string as the variable that you want to assign to the tab of the Client Request node. The variable type is pre-defined, based on the parameters to which you are assigning the variable. The string represents the datatype used to define the value of the key, which is the core aspect of the object.
4. Close the Client Request node. You can now add a `get (String key)` node to your business process.

To Get an XML metadata Document

You can use the following property to get an XML metadata document using the WebLogic Integration:

```
xmlObject get(String key)
```

The control uses this property to get a specific key from the XML MetaData Cache.

To get a key using the WebLogic Integration in a business process, perform the following steps:

1. Drag the `XmlObject get(String)` method from the **Data Palette** and drop it below the Client Request node in your business process.
2. Double-click the get node to display the **General Settings** tab of the **Node Builder**.
3. Confirm that **My**WebLogic Integration is displayed in the **Control** field and that the following method is selected in the **Method** field:

   ```
   get(String key)
   ```

4. Open the **Send Data** tab. In the **Select variables to assign** drop-down list, select *Key1*(String).
5. Open the **Receive Data** tab. In the **Select variables to assign** list, create a new variable called *XML_1*. The XMLObject datatype is selected by default. This step completes the get request for the XML metadata.
6. Close the window.

The following code sample reflects the configuration of the `add (String key)` node.

```
public void get_metadataGet() throws Exception
{
//#START: CODE GENERATED - PROTECTED SECTION - you can safely add code above
this comment in this method. #//
// input transform
// return method call
this.XML_1 = get_metadata.get(this.Key_1);
// output transform
// output assignments
//#END: CODE GENERATED - PROTECTED SECTION - you can safely add code below
this comment in this method. #//
}
```

Example: XML MetaData Cache Control

The following scenario describes how the XML MetaData Cache is deployed by business processes in a workflow.

The following elements are involved:

- The Router process: A business process that routes the flow of information between the Seller and the client, and the Buyer and the client.
- The Seller process: The Seller process communicates with the Router process, which in turn communicates with other elements in the workflow.
- The Buyer process: The Buyer process communicates with the Router process, which in turn communicates with other elements in the workflow.
- Auction client: The auction Website, with which both the Seller and the Buyer processes interact via the Router.

The Seller puts up goods for sale on the auction client and the buyer bids for the goods. When the bid is approved, the Seller requires the Buyer's details to proceed with the sale. Communication between the various elements of the workflow is handled by the Router process.

When the Buyer's bid is approved, the following scenario takes place:

1. The Seller sends a request to the auction client for the Buyer's metadata. This metadata can include information such as the buyer's name, address, telephone number and so on.
2. The Router examines the request and, based on the content of the request, pings the Buyer process for the required metadata.
3. The buyer's information is contained in an XML document, present in the global XML MetaData Cache. The Router sends a request to the Buyer for the relevant key.
4. The Router retrieves the key from the Buyer and uses the XML MetaData Cache Control to perform a get request for the required metadata. The retrieved XML metadata is returned to the client. In this event, the Router process adds an XML MetaData Cache Control, and uses the `get (String key)` method to retrieve the Buyer's XML metadata. The XML MetaData Cache Control, is in effect, added to the Router process. This allows for smooth data retrieval.

   Note: To enable successful transfer of information, the Buyer's metadata has to be present in the cache, before the Router business process is deployed.

Using MBean APIs to Manage an XML MetaData Cache

You can use the Configuration MBean APIs to create your own cache, and to add, get, and delete data from the cache.

Retrieving the Singleton XMLCacheMBean

Use the following code to retrieve the singleton XMLCacheMBean:

```
Context ctx = new InitialContext();
MBeanHome home = (MBeanHome) ctx.lookup(MBeanHome.LOCAL_JNDI_NAME);
(XMLCacheMBean) xmlCacheMBean =
home.getMBean(XMLCacheMBean.SINGLETON_MBEAN_NAME,
XMLCacheMBean.MBEAN_TYPE);
String key = "key1";
```

Using the Configuration MBean API

The Configuration MBean API provides the following methods to add an entry to, or to get or delete an entry from the XML MetaData Cache:

```
public XmlObject xmlObj = XmlObject.Factory.parse(new File(YourXmlFile))
//create an xmlObject
```

Use this method to create an XMLobject.

```
public xmlCacheMBean.add(key,  xmlObj);
```

Use this method to add a key to the XML MetaData Cache.

```
public XmlObject xmlObject_get = xmlCacheMBean.get(key);
```

Use this method to get a key from the XML MetaData Cache.

```
public XmlObject newxmlObj = ..... //create another xmlObject
```

Use this method to create a new XMLObject.

```
public xmlCacheMBean.update(key,newxmlObj);
```

Use this method to update an XML MetaData Cache.

```
public xmlCacheMBean.delete(key);
```

Use this method to delete a key from the XML MetaData Cache.

```
public boolean keyExists = xmlCacheMBean.keyExists(key);
```

Use this method to find out if a particular key exists within the XML MetaData Cache.

```
public String[] allKeys = xmlCacheMBean.getAllKeys();
```

Use this method to get a list of all keys within the XML MetaData Cache.

Using Control Factories

When creating some controls, you specify whether you want to make the control instance a control factory. A control factory allows a single application to manage multiple instances of the same control. File, Email, WLI JMS, TPM, and Worklist controls can be implemented as control factories.

To make a control a control factory, select the **Make this a control factory that can create multiple instances at runtime** check box when creating the control. When you add a control to a business process, if the control is a factory, the first argument of the control receive method is the controltype. This is displayed in the node builder assignment and mapping panel and you can assign and map to it.

Using Message Attachments

Business processes can exchange business messages with trading partners via ebXML or RosettaNet. These business messages include one or more *attachments* containing XML or non-XML data.

Note: For ebXML messages, each attachment represents a single *payload* in the ebXML message.

Attachments can be any of the following Java types (see Table 21-1):

Table 21-1 Java Type

Type	Description
`XmlObject`	Represents untyped XML format data.
`XmlObject[]`	Used for ebXML only—an array containing one or more `XmlObject` elements.
`RawData`	Represents any non-XML structured or unstructured data for which no MFL file (and therefore no known schema) exists.
`RawData[]`	Used for ebXML only—an array containing one or more `RawData` elements
`MessageAttachment[]`	Represents either untyped XML or non-XML data in a message attachment. Used for payloads in business messages that contain both untyped XML and non-XML data.

Attachments can also be typed XML or typed MFL data as long as you specify the corresponding XML Bean or MFL class name in the parameter.

If you use arrays as attachment type, certain restrictions apply to the order of your arguments. For more informations, see "Specifying XmlObject and RawData Array Payloads" on page 4-7.

For business messages containing both untyped XML and non-XML data, the message payload is represented as an array of `MessageAttachment` objects: `MessageAttachment[]`

The following APIs in the `com.bea.data` package provide access to individual `MessageAttachment` objects within the array (see Table 21-2):

Table 21-2 API Type

Object	Description
MessageAttachment Interface	Represents part of a message attachment in an ebXML or RosettaNet business message. Provides methods for retrieving untyped XML or non-XML data from an attachment.
MessageAttachment.Factory Class	Factory for creating `MessageAttachment` instances. Provides methods for creating `MessageAttachment` instances from untyped XML or non-XML data.

For more information about using the message attachment APIs, see the interfaces listed in the `com.bea.data` package in the Javadoc.

Related Topics

Guide to Building Business Processes