BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Using Integration Controls

Controls make it easy to access enterprise resources, such as databases, file systems, Enterprise Java Beans, and so on, from within your application. The control handles the work of connecting to the enterprise resource for you, so that you can focus on your business process' business logic.

Note: In addition to the controls listed in this topic, several extra controls, including a Tuxedo control, are included in the WebLogic Platform installation. For documentation and samples for these controls, go to the *BEA_HOME*\ext_components directory, where *BEA_HOME* stands for the BEA Systems installation directory.

Topics Included in This Section

Using Controls in Business Processes

An introduction to working with integration controls.

Controls and Transactions

Describes how controls relate to business process transactions and which controls are transactional.

Message Broker Controls

Describes the Message Broker resource, which provides a publish and subscribe message−based communication model for WebLogic Integration business processes. This section describes Message Broker Publish and Subscription controls, and File, JMS, Email, and Timer event generators, which facilitate publishing events to Message Broker channels.

File Control

Describes how to create File controls and use them to read, write, or append to files in a file system.

Email Control

Describes how to create Email controls and use them to allow WebLogic Integration business processes to send e−mail to a specific destination.

WLI JMS Control

Describes how to create WLI JMS controls and use them to allow WebLogic Integration business processes to easily interact with messaging systems that provide a JMS implementation.

Application View Control

Describes how to create Application View controls and use them to allow WebLogic Integration business processes to access an enterprise application using an Application View.

ebXML Control

Describes how to create ebXML controls and use them to allow WebLogic Integration business processes to exchange business messages and data with trading partners via ebXML.

RosettaNet Control

Describes how to create RosettaNet controls and use them to allow WebLogic Integration business processes to exchange business messages and data with trading partners via RosettaNet.

TPM Control

Describes how to create TPM controls and use them to provide WebLogic Integration business processes with query (read−only) access to trading partner and service information stored in the TPM repository.

Worklist Controls

Describes how to create Worklist controls (Task and Task Worker controls) and use them to allow WebLogic Integration business processes to interact with a Worklist.

Process Control

Describes how to create Process controls and use them to allow WebLogic Integration business processes to invoke other business processes.

Service Broker Control

Describes how to create Service Broker controls and use them to allow WebLogic Integration business processes to interface with a single control that provides relays, based upon decision criteria, to any number of other services or business processes.

Using Control Factories

Describes how to create controls as control factories.

Using Message Attachments

Describes how message attachments are used in ebXML and RosettaNet business messages.

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

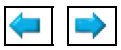
You can use the ControlContext Interface for access to a control's properties at run time and for handling control events. Property values set by a developer who is using the control are stored as annotations on the control's declaration in a JWS, JSP, or JPD file, or as annotations on its interface, callback, or method declarations in a JCX file.

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:

- Message Broker Controls
- File Control
- WLI JMS Control
- Application View Control
- ebXML Control
- RosettaNet Control
- TPM Control
- Worklist Controls
- Email Control
- Process Control
- Service Broker Control
- Database Control
- EJB Control
- Portal Controls
- Web Service Control
- Timer Control
- BEA Tuxedo Control
- Liquid Data 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): Application View, Email, File, Process, Task, and Task Worker controls.



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:

- Application View (if JCA adapter is transactional)
- ebXML
- Message Broker
- Process (see the previously listed qualifications)
- RosettaNet
- WLI JMS
- Worklist

The following integration controls are not transactional:

- File
- Email
- Service Broker
- TPM

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

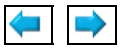
When you call Web Service controls asynchronously from business processes, it is recommended that you buffer the asynchronous call. After creating the Web Service control, specify that the asynchronous calls from the business process to the control are buffered. By doing so, you ensure that the message 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.

To learn how to buffer the methods, see Buffering Methods and Callbacks. 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.

Related Topics

Transaction Boundaries

Building Synchronous and Asynchronous Business Processes



Application View Control

Note: The Application View control uses application views defined using the Application Integration Design Console, provided with WebLogic Integration. The Application View control is available in WebLogic Workshop only if you are licensed to use WebLogic Integration.

The Application View control allows your web service or business process to access an enterprise application using an application view. An application view must be created using the Application Integration Design Console before it can be referenced using an Application View control. To learn more about application views and their relationship to enterprise applications, see Overview: Application Integration.

Like other WebLogic Workshop controls, the Application View control allows WebLogic Workshop web services and business processes to interact with enterprise applications using simple Java APIs. They allow a developer to access an enterprise application even if they don't know any of the details of the application's implementation.

The Application View control provides a means for a developer to invoke application view services both synchronously and asynchronously, and start a new business process when an EIS event occurs. In both the service and event cases, the developer uses XML and mapping tools to interact with the Application View control. The developer need not to understand the particular protocol or client API for the enterprise application (hereafter referred to as an Enterprise Information System or EIS). Events are delivered using the Message Broker Subscription control. Message Broker integration is provided by publishing all application view events to the Message Broker through its API.

Topics Included in this Section

Overview: Application Integration

Describes the relationship between enterprise application adapters, WebLogic Integration application views, and the Application View control.

Creating a New Application View Control

Describes how to create and configure an Application View control.

Updating an Application View Control

Describes how to update an Application View control when the underlying application view changes.

Using an Application View Control

Describes how to use an existing Application View control from within a business process.

Prerequisites for Integrating Applications Using WebLogic Workshop

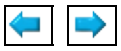
The WebLogic Workshop Application View control is designed to make it easy for you to use an existing, deployed application view from within your business process. WebLogic Workshop is specifically not designed to help you develop and deploy application views. Please consult *Using the Application Integration Design Console* to learn how to use the Application Integration Design Console to create and publish application views.

Any WebLogic Workshop application which uses the application integration capabilities of WebLogic Integration must contain a project explicitly named Schemas. The Schemas project is used to store the wlai.channel file and application view schemas (published as XML Bean classes). If the Schemas project does not exist in the application, you must create it before publishing application views.

Application views with services that are published to a WebLogic Workshop application must not contain underscores in the application view service names. Also, no underscores are allowed in the Application View control name. When building a Control Receive node, WebLogic Workshop only allows a single underscore in a method name, which is automatically generated from the control name and the method name.

Related Topics

Using Controls in Business Processes

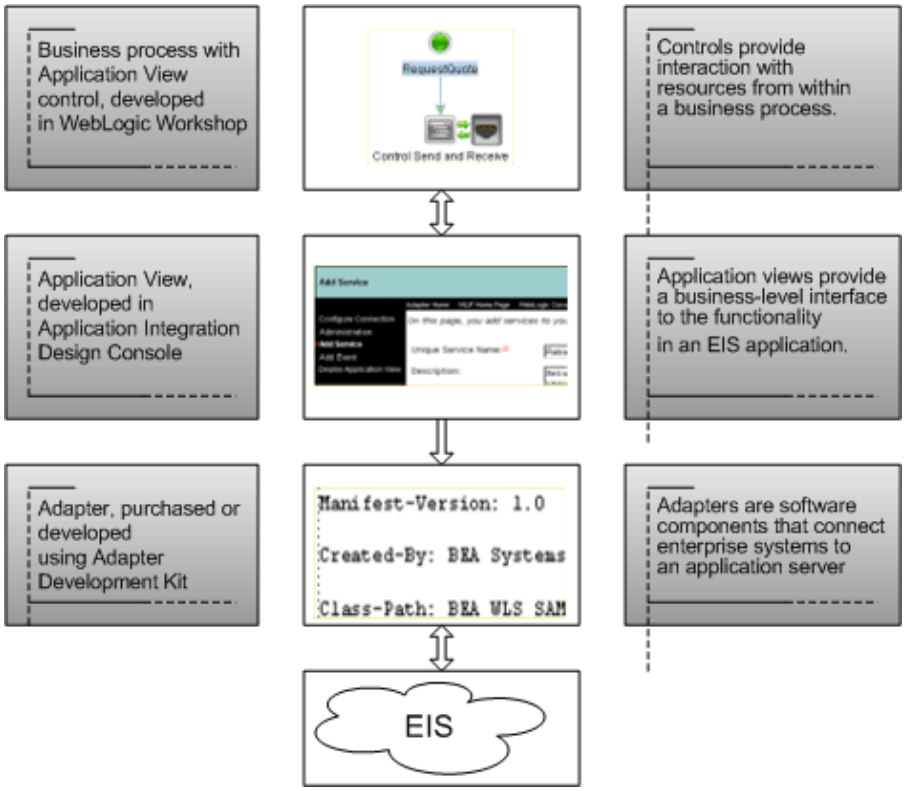


Overview: Application Integration

WebLogic Integration provides a standards−based integration solution for connecting applications both within and between enterprise applications (also called Enterprise Information Systems or EISs). An EIS is typically a large−scale business application such as a Customer Relationship Management (CRM), Enterprise Resource Planning (ERP) or Human Resources (HR) application. Examples of EISs include SAP, PeopleSoft, or Siebel. WebLogic Integration provides the following tools for integrating applications:

- Adapters
- Application Views
- Application View Control

The following figure shows how the various application integration components interact.



By using these tools, you can integrate all your enterprise information systems (EIS). Typical IT organizations use several highly specialized applications. Without a common integration platform, integration of such applications requires extensive, highly specialized development efforts.

Adapters

In order to integrate the operations of an enterprise, the data and functions of the various EISs in an organization must be exposed. In the Java 2 Enterprise Edition (J2EE) model, EIS functionality is exposed to Java clients using an adapter (sometimes called a resource adapter or a connector) according to the J2EE Connector Architecture. WebLogic Integration makes use of adapters to establish a single enterprise−wide framework for integrating current or future applications. Adapters greatly simplify your integration efforts because they allow you to integrate each application with a single application server, and thus avoid the need to integrate every application with every other application. Adapters for popular EISs are available from

applications vendors, from BEA Systems, and from third−party vendors.

As an extension to BEA WebLogic Integration, BEA offers a growing portfolio of BEA WebLogic Adapters. These adapters completely conform to the J2EE Connector Architecture specification, and feature enhancements that enable faster, simpler and more robust integration of your business−critical applications. Each adapter provides bi−directional, request−response integration with a specific application or technology. User information on specific adapters is available at http://e−docs.bea.com. Please contact Customer Support for platform support information.

If your business requires a specialized, custom adapter, WebLogic Integration provides an Adapter Development Kit. The ADK is a set of tools for implementing the event and service protocols supported by WebLogic Integration. These tools are organized in a collection of frameworks that support the development, testing, packaging, and distribution of resource adapters for WebLogic Integration. Specifically, the ADK includes frameworks for design−time operation, run−time operation, logging, and packaging. For more information on the ADK, see *Developing Adapters*.

Application Views

In addition to defining and implementing adapters, the AI component of WebLogic Integration enables a developer to create application views. An application view provides a layer of abstraction on top of an adapter; whereas adapters are closely associated with the specific functions available in the EIS, an application view is associated with business processes that must be accomplished by clients. The application view converts the steps in the business process into operations on the adapter.

An application view exposes services and events that serve the business process. Each WebLogic Workshop Application View control is associated with a particular application view, and makes the services and methods of the application view available to WebLogic Workshop web services as control methods and callbacks.For information on defining application views, see *Using the Application Integration Design Console*.

A *service* represents a message that requests a specific action in the EIS. For example, an adapter might define a service named AddCustomer that accepts a message defining a customer and then invokes the EIS to create the appropriate customer record.

An *event* issues messages when events of interest occur in the EIS. For example, an adapter might define an event that sends messages to interested parties whenever any customer record is updated in the EIS. Events are delivered using the Message Broker Subscription control. Message Broker integration is provided by publishing all application view events to the Message Broker through its API. To learn about the Message Broker Subscription control, see Message Broker Controls.

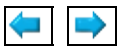
Application View Control

You use application view controls in WebLogic Workshop to interact with an EIS through an application view. Application view controls allow a business process engineer to browse the hierarchy of application views, invoke a service as an action in a business process, and start a new business process when an EIS event occurs.

Events are delivered using the Message Broker Subscription control. Message Broker integration is provided by publishing all Application View events to the Message Broker through its API.

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

Processes.



Creating a New Application View Control

This topic describes how to create a new Application View control.

To learn about Application View controls, see Application View Control.

To learn about WebLogic Workshop controls, see Using Controls in Business Processes.

To create a new Application View control:

1. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **ApplicationView** to display the **Insert ApplicationView** dialog.
4. In the **Variable name for this control** field, type the variable name used to access the new Application View control instance from your business process. The name you enter must be a valid Java identifier.
5. In the Step 2 pane, choose the **Create a new ApplicationView control to use** radio button.
6. In the **New JCX name** field, type the name of your new JCX file. The .jcx filename extension is automatically appended to the name you enter.
7. 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. For more information about control factories, see Control Factories: Managing Collections of Controls.
8. In the Step 3 pane, click **Browse...** The Application Views Browser dialog opens, displaying application views that are published in the current domain. Application views that have not been published using the Application Integration Design Console do not appear in the list.
9. Select the published application view you want this Application View control to represent. Services offered by the selected application view are displayed in the **Services to Invoke Asynchronously** list.
10. Select services that will be invoked asynchronously by selecting the check box next to the name of the service. Selecting a service causes it to be generated in the control's JCX file as an asynchronous service. This means it has a call−in with a void return and a callback with param as the service response type. Click **OK**.
11. In the **Insert ApplicationView** dialog, enter the WebLogic Workshop application name in the **Application Name** field. The application name is usually the same as that of the current WebLogic Workshop application.

    The Application Name parameter allows you to reuse an application view between WebLogic Workshop applications. You must first define the application view in the context of the primary application (app1) using the Application Integration Design Console. Then, define an Application View control in a process or web service within a second application (app2) and specify app1 in the **Application Name** field in the **Insert ApplicationView** dialog. Because the browse function uses the context of the current application, you cannot use the browse function to locate application views in another WebLogic Workshop application. When reusing an application view from another application, all services are accessed synchronously.

12. Click **Create**.

Application View Control Methods

To learn about the methods available on the Application View control, see the com.bea.wlai.control package in the WebLogic Integration Javadoc at the following URL:

http://edocs.bea.com/wli/docs81/javadoc/com/bea/wlai/control/package−summary.html

Example: Application View Control

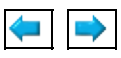
When you create a new Application File control, it appears in your project directory as a JCX file. The following is an example of an Application View control:

```
package FunctionDemo;

import weblogic.jws.*;
import com.bea.wlai.control.ApplicationViewControl;
import com.bea.xml.XmlObject;
/**
 * This ApplicationView provides some simple services to
 * create/get/update customers in the sample CUSTOMER_TABLE table.
 * It also defines an event
 * indicating a customer record has been updated.
 * @jc:av−identity name="FunctionDemo.CustomerMgmt"
 * app="sampleApp"
 */
public interface CustomerMgmtAppView extends
 com.bea.control.ControlExtension, ApplicationViewControl
{

    /**
     * Get a customer record given first and last name.
     * @jc:av−service name="GetCustomer" async="false"
     */
public wlai.functionDemo.
customerMgmtGetCustomerResponse.RowsDocument
GetCustomer(wlai.functionDemo.customerMgmtGetCustomerRequest.
InputDocument request)
        throws Exception;

    /**
     * Update the customer's email address.
     * @jc:av−service name="UpdateCustomer" async="false"
     */

public wlai.functionDemo.customerMgmtUpdateCustomerResponse.
RowsAffectedDocument UpdateCustomer(wlai.functionDemo.
customerMgmtUpdateCustomerRequest.InputDocument request)
        throws Exception;

    /**
     * Select all customers in the customer table.
     * @jc:av−service name="GetAllCustomers" async="true"
     */
    public void GetAllCustomers()
        throws Exception;

    /**
```

```
     * Create a new customer given first and last name,
     * and date of birth.
     * @jc:av-service name="CreateCustomer" async="false"
     */
public wlai.functionDemo.
customerMgmtCreateCustomerResponse.RowsAffectedDocument
CreateCustomer(wlai.functionDemo.
customerMgmtCreateCustomerRequest.InputDocument request)
        throws Exception;


    public interface Callback extends
 ApplicationViewControl.Callback

    {

        /**
         * Async response callback method for
         * GetAllCustomers service.
         */
public void onGetAllCustomersResponse(wlai.functionDemo.
customerMgmtGetAllCustomersResponse.RowsDocument response);

        /**
         * Callback to handle errors with async service requests.
         * Note that only one async request can be in flight
         * for a given control instance at any given time.
         *
         * @param errorMsg
         * The error message text for the failed async request.
         */
        public void onAsyncServiceError(String errorMsg);
    }
}
```

If a business process or Web service does not implement the onAsyncServiceError callback, the Application View control does not write errors to the server log. WebLogic Workshop informs you when you build the application if the onAsyncServiceError callback is not implemented with warning messages similar to the following:

```
WARNING: dummy.jpd:35: The Callback interface InsertAsyncCtrl.Callback
defines a method void
onInsertServiceResponse(wlai.dbms.masterApplicationViewInsertServiceResponse.
RowsAffectedDocument), but you don't define an equivalent event handler.

WARNING: dummy.jpd:35: The Callback interface InsertAsyncCtrl.Callback defines a
method void onAsyncServiceError(java.lang.String), but you don't define an
equivalent event handler.
```



Customizing an Application View Control

You can customize an Application View control in several ways. You may modify the properties of the control itself or the properties of the control's methods. Each of these modifications is described in more detail in the sections that follow.

You can also use the ControlContext interface for access to a control's properties at run time and for handling control events. 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 JPD file, or as annotations on its interface, callback, or method declarations in a JCX file.

Control Properties

The Application View control exposes the av−identity property with the name and app attributes. For a description of the av−identity property and its attributes, see @jc:av−identity Annotation.

Method Properties

Each method of an Application View control exposes the av−service property that binds the Application View control method to an application view service. For a description of the av−service property and its attributes, see @jc:av−service Annotation.

Related Topics

Using an Application View Control

Using Controls in Business Processes

ControlContext Interface

Updating an Application View Control

Once an application view is designed in the Application Integration Design Console and then published, control of the Application view is passed to the WebLogic Workshop application. If changes are required to the design of the application view, you must make them in the Application Integration Design Console and republish the application view. You must then regenerate the Application View control to ensure that the design changes are available to the WebLogic Workshop application.

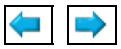
Updating a Control when an Application View Changes

To update an Application View control when the target application view changes, you must regenerate the Application View control.

Rename or delete the old Application View control JCX file before generating a new Application View control with the same name. If you customized control properties, these customizations must be redone on the new control.

Related Topics

Creating a New Application View Control



Using an Application View Control

This topic describes how to use an existing Application View control in your web service.

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

To learn about Application View controls, see Application View Control.

To learn how to create a Application View control, see Creating a New Application View Control.

Using an Existing Application View Control

All controls follow a consistent model. Therefore, most aspects of using an existing Application View control are identical to using any other existing control. To use an existing Application View control:

1. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **ApplicationView** to display the **Insert ApplicationView** dialog.
4. In the **Variable name for this control** field, type the variable name used to access the existing Application View control instance from your business process. The name you enter must be a valid Java identifier.
5. In the Step 2 pane, choose the **Use an ApplicationView control already defined by a JCX file** radio button.
6. Click **Browse** to browse for existing Application View controls. The Select dialog is displayed. When you find the control you want to use, select it and click **Select**.
7. Click **Create**.

Customizing an Application View Control

There are properties that are specific to the Application View control. If you choose to copy and customize an existing Application View control, the properties you may wish to modify are:

- av−identity
  For more information, see @jc:av−identity Annotation.
- av−service
  For more information, see @jc:av−service Annotation.

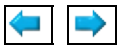
ApplicationViewControl Interface

All Application View controls are subclassed from the ApplicationViewControl interface. The interface defines methods that may be called on Application View control instances from a web service.

To learn more, see Application View Control Interface.

Related Topics

Creating a New Application View Control



ebXML Control

Note: The ebXML control is available in WebLogic Workshop 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* at the following URL:

http://edocs.bea.com/wli/docs81/tpintro/index.html

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

Using Built−In Java Controls

Introducing Trading Partner Integration at http://edocs.bea.com/wli/docs81/tpintro/index.html

Trading Partner Management at http://edocs.bea.com/wli/docs81/manage/tpm.html

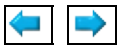
EBXMLControl Interface

Tutorial: Building ebXML Solutions at http://edocs.bea.com/wli/docs81/tptutorial/ebxml.html

Building ebXML Participant Business Processes

@jpd:ebXML Annotation

@jpd:ebXML method Annotation



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 EBXMLControl Interface. 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 the following URL:

http://edocs.bea.com/wli/docs81/tpintro/index.html

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* at the following URL:

http://edocs.bea.com/wli/docs81/tpintro/index.html.

To learn about adding or updating information in the TPM repository, see Trading Partner Management in *Managing WebLogic Integration Solutions* at the following URL:

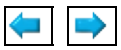
http://edocs.bea.com/wli/docs81/manage/tpm.html

Related Topics

Creating an ebXML Control

Using an ebXML Control

Example: ebXML Control



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 learn about ebXML controls, see ebXML Control.

To create a new ebXML control

1. If you are not in Design View, click the **Design View** tab.
2. On the **Controls** section of the **Data Palette**, click **Add**.

   Note: If the **Controls** tab is not visible in WebLogic Workshop, choose **View** > **Windows** > **Data Palette** from the menu bar. Instances of controls already available in your project are displayed in the Controls tab.

3. In the pop−up menu, click **Integration Controls** to display a drop−down list of controls that represent the resources with which your business process can interact.
4. Click **ebXML** to display the **Insert Control − Insert ebXML** dialog box.
5. In the Step 1 pane, in the **Variable name for this control** field, 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.
6. In the Step 2 pane, select one of the following options:
   - **Use an ebXML control already defined by a JCX file**

     Enter the name of the JCX file, or click the **Browse** button to find and select it.

   - **Create a new ebXML control to use**

     Enter the name of the new JCX file to create.

7. If you are creating a new control, in the Step 3 pane, specify the following information:

Field	Description
ebxml−service−name	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.
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.
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.
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.
ebxml−action−mode	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.

8. Click the **Create** button.
9. If you are prompted, select a subfolder in which to save the JCX file.

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

After you create the JCX file, the name of the JCX file becomes available as a service on the Services tab in the WebLogic Integration Administration Console.

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 (JCX file). You can do this by double−clicking on the file in the Application 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.

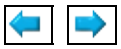
To learn more about the request and response methods, see EBXMLControl Interface.

Related Topics

Overview: ebXML Control

Using an ebXML Control

Example: ebXML Control



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. To learn about WebLogic Workshop controls, see Using Built−In Java 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 ebXML Control Interface.

The ebXML control is a JCX file. To learn about using JCX files, see JCX Files: Extending Controls.

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 JCX 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. Valid data types include:

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 @jc: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:

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 in the @jc:ebxml−method tag. To learn more about the envelope annotation, see @jc:ebxml−method 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 (@jc:ebxml−method Annotation) and the DynamicProperties.xml file
2. properties set using the setProperties() method
3. properties set at the JCX instance level using the @jc:ebxml Annotation annotation in the JPD
4. properties set at JCX class level using @jc:ebxml Annotation annotation in the JCX

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. Display the business process in Design View that contains the ebXML control for which you want to specify a dynamic selector.
2. In Design View, select the ebXML control node in the Data Palette.
3. Locate the **from−selector** or **to−selector** property in the Property Editor 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 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 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. To learn about importing schemas, see How do I: Import Schemas into a Project Schemas Folder.
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.

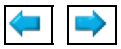
To display the current property settings, use the getProperties() method.

Related Topics

Overview: ebXML Control

Creating an ebXML Control

Example: ebXML Control



Example: ebXML Control

For examples of how to use the ebXML control, see Tutorial: Building ebXML Solutions at the following URL:

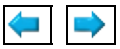
http://edocs.bea.com/wli/docs81/tptutorial/ebxml.html

Related Topics

Overview: ebXML Control

Creating an ebXML Control

Using an ebXML Control



Message Broker Controls

Note: The Message Broker controls are available in WebLogic Workshop 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.

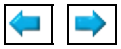
In addition to business processes that can publish messages to Message Broker channels, WebLogic Integration supports event generators, which can publish external events to message broker channels. WebLogic Integration provides native event generators, including JMS, Email, File, and Timer event generators. 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. WebLogic Integration also works with Application View event generators, which work with J2EE−CA connectors. To learn about creating and managing event generators using the WebLogic Integration Administration Console, see Event Generators in *Managing WebLogic Integration Solutions* at the following URL:
http://edocs.bea.com/wli/docs81/manage/evntgen.html

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.

You can use the ControlContext Interface for access to a control's properties at run time and for handling control events. 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 JPD file, or as annotations on its interface, callback, or method declarations in a JCX file.

Topics Included in This Section

- Message Broker Publish Control
- Message Broker Subscription Control
- Using Event Generators to Publish to Message Broker Channels



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:

- To Create an Instance of a Message Broker Publish Control
- Using Methods of the MB Publish Interface
- Example Code for MB Publish Control

To Create an Instance of a Message Broker Publish Control

1. Click Add on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **MB Publish**. The **Insert Control** dialog box is displayed.
4. In the **Insert Control** dialog box (**Step 1**), enter a name for the instance of this control. The name you enter must be a valid Java identifier.
5. In the **Insert Control** dialog box (**Step 2**), select one of the following options:
   - Use a MB Publish control already defined by a JCX file

     Enter a filename for the MB control in the **JCX file** field, or click **Browse** to find the JCX file in your file system.

   - Create a new MB Publish control to use

     Enter a filename in the **New** JCX name field.

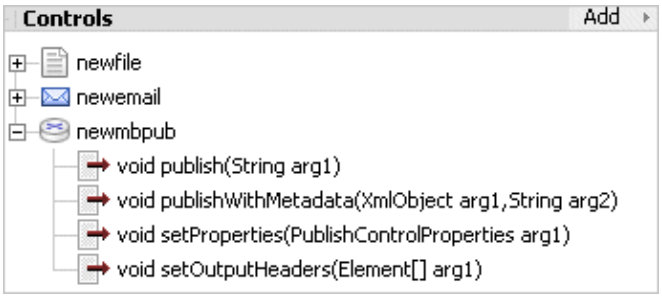
6. In the **Insert Control** dialog box (**Step 3**), specify the channel name as follows:
   - **channel−name** Select a channel to which you want your business process to publish.

     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. To learn how to create this file, see Adding a Channel File to Your Project.

   - **message type** This read−only field displays the type of data published to 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.

7. Click **Create** to close the **Insert Control** dialog box.

   An instance of a MB Publish control is created in your project and displayed in the **Controls** tab. The following figure shows an example instance of a MB Publish control displayed in the **Controls** tab:



JCX File for Your MB Publish Control

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

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

/**
 * Defines a new Publish control.
 *
 * @jc:mb-publish-control channel-name="/controls/channe1"
 */

public interface mbPublish extends PublishControl,
 com.bea.control.ControlExtension
    /**
     * @jc:mb-publish-method message-body="{value}"
     */
    void publish(String value);
    /**
     * @jc:mb-publish-method message-metadata="{metadata}" message-body="{value}"
     */
    void publishWithMetadata(XmlObject metadata, String value);
}
```

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 PublishControl Interface 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 JPD file.

```
/*
 * @jc:mb−publish−control channel−name="/controls/potopic"
 */
```

```
interface MyPublishControl extends PublishControl,com.bea.control.ControlExtension {
    /**
     * @jc:mb-publish-method
     *      message-metadata="<custom-header>ACME Corp</custom-header>"
     *      message-body="{myMsgToSend}"
     */

    void publishPO(XmlObject myMsgToSend);
}


/*
 * @common: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. Click Add on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **MB Subscription**. The **Insert Control** dialog box is displayed.
4. In the **Insert Control** dialog box (**Step 1**), enter a name for the instance of this control. The name you enter must be a valid Java identifier.
5. In the **Insert Control** dialog box (**Step 2**), select one of the following options:
   - ♦ Use a MB Subscription control already defined by a JCX file

     Enter a filename for the MB control in the **JCX file** field, or click **Browse** to find the JCX file in your file system.

   - ♦ Create a new MB Subscription control to use

     Enter a filename in the **New** JCX name field.

6. In the **Insert Control** dialog box (**Step 3**), specify the channel name as follows:

- **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. To learn how to create this file, see Adding a Channel File to Your Project.

- **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.

7. Select the **This subscription will be filtered** checkbox if you want to subscribe using filter values.
8. Click **Create** to close the **Insert Control** dialog box.

An instance of a MB Subscription control is created in your project and displayed in the **Controls** tab. The following figure shows an example instance of a MB Subscription control displayed in the **Controls** tab:

The control declaration is written to your JPD source file.

```
/**
  * @common:control
  */
  private processes.mbSubscribe mbSubscribe;
```

JCX File for Your MB Subscription Control

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

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

/**
 * Defines a new Subscription control.
 *
 * @jc:mb−subscription−control channel−name="/controls/channel1"
 */

public interface mbSubscribe extends SubscriptionControl,
 com.bea.control.ControlExtension
{
    /**
     * @jc:mb−subscription−method filter−value−match="{value}"
     */

    void subscribeWithFilterValue(String value);
```

```
    interface Callback extends SubscriptionControl.Callback {
        /**
         * @jc:mb-subscription-callback message-body="{message}"
         */

       void onMessage(XmlObject message);
    }
}
```

You must select the **This subscription will be filtered** checkbox to ensure that the subscribeWithFilterValue() method in included in the JCX 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 SubscriptionControl Interface 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

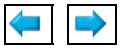
MB Subscription controls must be extended. The following is an example of how to code a MB Subscription control in your JPD file.

```
/*
 * @jc:mb−subscription−control
 *         channel−name="/controls/stocks"
 *         xquery="$message/StockSymbol/text()"
 */
interface MySubscriptionControl extends SubscriptionControl, ControlExtension {
    /**
     * @jc:mb−subscription−method
     *      filter−value−match="BEA"
     */
    void subscribeToBEA();
    /**
     * @jc:mb−subscription−method
     *      filter−value−match="{symbol}"
     */
    void subscribeToSymbol(String symbol);
    interface Callback {
        /**
         * @jc:mb−subscription−callback message−body="{myMsgReceived}"
         */
        onXMLMessage(XmlObject myMsgReceived);
   }
}
.
.
.
/*
 * @common:control
 */
 MySubscriptionControl subCtrl;
// subscribe to a message

void subscribeIt() {
      subCtrl.subscribeToBEA();
}
// receive a message after subscribing
subCtrl_onXMLMessage(XmlObject myMsgReceived)
{
}
```

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.



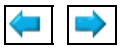
Using Event Generators to Publish to Message Broker Channels

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

- File Event Generators
- JMS Event Generators
- Email Event Generators
- Timer Event Generators

To learn about creating and managing event generators using the WebLogic Integration Administration Console, see Event Generators in *Managing WebLogic Integration Solutions* at the follwong URL:

http://edocs.bea.com/wli/docs81/manage/evntgen.html



Email Control

Note: The Email control is available in WebLogic Workshop 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 *Managing WebLogic Integration Solutions* at the following URL:

http://edocs.bea.com/wli/docs81/manage/evntgen.html

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

Topics Included in This Section

Overview: Email Control

Provides an overview of the Email control.

Configuring an Email Control

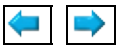
Describes how to configure an existing Email control.

Creating a New Email Control

Describes how to create and configure an Email control.

Sample Email Messages

Provides sample e−mail messages with different formats.



Overview: Email Control

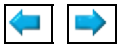
The Email control enables WebLogic Workshop web services and business processes to send e−mail to a specific destination. The body of the e−mail message can be text (plain, HTML, or XML) or can be an XML object. The control is customizable, allowing you to specify e−mail transmission properties in an 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 (.jcx) or create a new one. Depending on the data type of the message body you select, the .jcx 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 .jcx file.

```
/**
 * @jc:send-email 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);


/**
 * @jc:send-email 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.
 * @jc:email
 *          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
{
    /**
     * @jc:send-email 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.

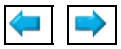
The following is an example of an XML variable used to set dynamic properties:

```
<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>
```

Related Topics

EmailControl Interface

Email Control Annotations



Creating a New Email Control

This topic describes how to create a new Email control.

To learn about Email controls, see Email Control.

To learn about WebLogic Workshop controls, see Using Controls in Business Processes.

To create a new Email control:

1. If you are not in Design View, click the Design View tab.
2. Click **Add** on the **Controls** tab to display a drop−down list of controls that represent the resources with which your business process can interact.

   Note: If the **Controls** tab is not visible in WebLogic Workshop, choose **View** > **Windows** > **Data Palette** from the menu bar. Instances of controls available to your project are displayed in the **Controls** tab.

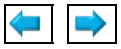
3. Choose **Integration Controls** to display the list of controls used for integrating applications.
4. Choose **Email** from the list to display the **Insert Control − Email** dialog box.
5. In the Step 1 pane, in the **Variable name for this control** field, type the variable name used to access the new Email control instance from your business process or web service. The name you enter must be a valid Java identifier.
6. In the Step 2 pane, choose the **Create a new Email control to use** radio button.
7. In the **New JCX name** field, type the name of your new JCX file. The .jcx filename extension is automatically appended to the name you enter.
8. 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. For more information about control factories, see Control Factories: Managing Collections of Controls.
9. In the Step 3 pane, enter the following name and address parameters:
   - **smtp−address** 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
10. Select the type of data contained in the message body using the **body−type** menu.
11. Click Create.

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 Property Editor:

- **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.

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. The following table lists commonly used 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 WebLogic Workshop 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

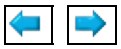
Describes how to create a new File control using the WebLogic Workshop graphical design interface.

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.

Example: File Control

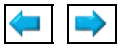
Provides an example of a File control in the context of a business process.



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 operations such as copy, rename, and delete. Typically, you use these operations to manipulate large files, without having to read their entire contents. You can also list 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 JCX file.

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

Creating a New File Control

1. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **File** to display the **Insert Control − File** dialog box.
4. In the **Variable name for this control** field, enter the name of the new control. The name you enter must be a valid Java identifier.
5. Select **Create a new File control to use** and enter a name for the JCX file which will define the control in the **New JCX name** field.

   You can use an existing control by selecting **Use a File control already defined by a JCX file** and entering a filename in the **JCX file** field.

6. Choose whether or not 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** checkbox. For more information about control factories, see Control Factories: Managing Collections of Controls.
7. 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.

8. In the **file−mask** 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−mask** 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−mask** field is optional when inserting a control, but this property must then be set dynamically before performing a file operation.

9. Select the type of data contained in the file using the **file−type** 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.
10. 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.
11. 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.

    If a record size is specified in the **record−size** field, the file is processed one record at a time. If no record size is specified, the file is processed one line at a time, delimited by the NEWLINE character of the operating system. This style of file processing can be used with any size file. If you are processing files one record at a time, and you are not using the NEWLINE character as a delimiter, enter the size of an individual record in the file (in bytes) in the **record−size** field.

12. Click **Create**.

File Control Methods

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

Example: File Control Declaration

When you create a new File control, its declaration appears in the JCX file. The following code snippet is an example of what the declaration looks like when you choose the **The directory contains large files to be processed** option:

```
import weblogic.jws.*;
import com.bea.control.*;
import java.io.*;
import com.bea.data.RawData;
import com.bea.xml.XmlObject;
 ...
 /**
  * @jc:file directory-name="C:\directory"
  * file-mask="tax_file.txt"
  *
```

```
 */
 public interface TaxFileControl extends
 FileControl,com.bea.control.ControlExtension
 {
/**
 * @jc:file-operation io-type="readline"
 *                record-size="80"
 */
 RawData readLine();
 }
```

The actual attributes that are present on the @jc:file and @jc:file−operation annotations depend on the values you entered in the Insert Control dialog.

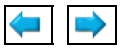
The @jc:file annotation controls the behavior of the File control. All of the attributes of the @jc:file annotation are optional and have default values.

To learn more, see @jc:file Annotation.

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

Related Topics

@jc:file Annotation



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.

You can also use the ControlContext interface for access to a control's properties at run time and for handling control events. 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 JPD file, or as annotations on its interface, callback, or method declarations in a JCX file.

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 Property Editor. These properties correspond to attributes of the @jc:file and @jc:file−operation annotations, which identify the File control in your code. The following attributes specify class− and method−level configuration attributes for the File control.

Annotation	Attribute	Description
@jc:file	*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.
	create−mode	Specifies whether a file is overwritten or renamed when a new file of the same name is created.
	ftp−host−name	Name of the FTP host, for example, ftp://ftp.bea.com.
	ftp−user−name	Name of the FTP user.
	ftp−password	FTP user's password. If you specify this attribute, you cannot specify the *ftp−password−alias* attribute.
	ftp−password−alias	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.
	ftp−local−directory	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.
@jc:file−operation	*io−type*	Type of file operation (read, readline, write, or append).
	file−content	Description of the contents of the file.
	record−size	Size of an individual record (in bytes) within a file to be processed record by record.
	encoding	Character set encoding of the file.

To learn more about specifying default File control behavior with attributes of the @jc:file annotation, see @jc:file 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 FileControl Interface.

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

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.

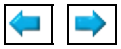
Error Handling When Reading Files

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.

Related Topics

FileControl Interface

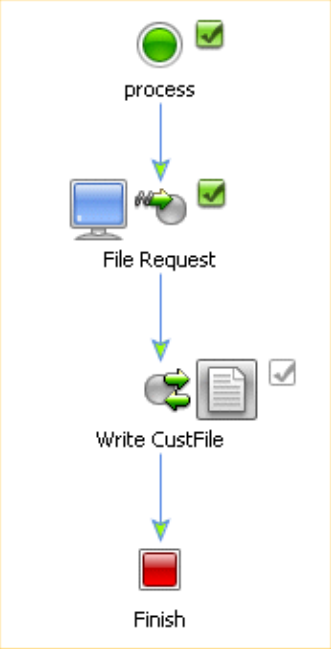
@jc:file Annotation



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:



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 Add on the **Data Palette Controls** tab to display a list of controls that represent the resources with which your business process can interact.
2. Click **Integration Controls**, then choose **File**. The **Insert Control** dialog box is displayed.
3. In the **Insert File Control** dialog box:
    a. In **Step 1**, enter **myFile** as the variable name for this control.
    b. In **Step 2**, ensure that the following option is selected: **Create a new File control to use**. Then, enter **MyFile** in the **New JCX name** field.
    c. In **Step 3**, enter values in the following fields:

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.

file−mask Enter a name for the file. For example, enter CustFile.xml.

file−type Select **XmlObject** from the drop−down list.

d. Click **Create** to close the **Insert Control** dialog box.

An instance of a File control, named **myFile**, is created in your project and displayed in the **Controls** tab.

4. 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(com.bea.xml.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 alreay exist, you can choose **Create new variable...** to create it now.
8. Click **Apply** and **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 **Apply** and **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 WebLogic Workshop only if you are licensed to use WebLogic Integration.

Hypertext Transport Protocol (Http), is a request−response protocol which is used for communication between a client and a server that allows the client to access resources on the server. 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 WebLogic Workshop clients. The Http Control complements the other controls provided in WebLogic Integration and can be used with WebLogic Workshop 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. For more information, 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.

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.

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

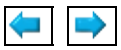
Describes the method used to log debug messages.

Http Control Caveats

Lists out the known limitations and caveats of the WebLogic Integration Http control.

The Http Event Generator

Describes the Http Event Generator briefly, with a link to a more detailed information source.



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. Click Add on the **Data Palette Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.

3. Choose **Http** to display the **Insert Control − Http** dialog.
4. In **Step 1**, in the **Variable name for this contro**l field, enter the name for your Http control.
5. In **Step 2**, select the **Create a new Http control to use** radio button.
6. In the **New JCX name** field, provide a name for the new file that you are about to create.
7. 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 runtim**e check box. For more information about control factories, see Control Factories: Managing Collections of Controls.
8. In **Step 3**, specify the target URL for your Http control. You need to specify the URL in the following format:

Http://www.bea.com

9. Select the Http mode that you want to use. You can select either the GET, or the Post mode. For more information about the two methods, see Understanding Http methods.
10. 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.
11. 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.
12. Click **Create**. Alternatively, you may create a Http control JCX file manually. For example, you may copy an existing Http control JCX file and modify the copy.

The JCX file for the Http control

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

```
package processes;

import com.bea.control.*;

import com.bea.wli.control.HTTPResponse.ResponseDocument;
```

```
import com.bea.wli.control.HTTPParameter.ParametersDocument;

import com.bea.xml.XmlObject;

/*
         * A custom HTTP control.

         */

         /**

         * @jc:HTTPsend−data url−name="HTTP://localhost:7001/console"

         */

public interface GET extends HTTPControl, com.bea.control.ControlExtension

         {

         /*

* A version number for this JCX. This will be incremented in new versions of

this control to ensure that conversations for instances of earlier versions were invalid.

         */

         static final long serialVersionUID = 1L;

ResponseDocument sendDataAsHTTPGet(ParametersDocument parameters,String charset);

byte[] getResponseBodyData();
```

The contents of the Http control's JCX file depend on the selections made in the Insert Http dialog. The given example was generated in response to selection of byte[] as 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.
10. Click **Apply** and **Close**.

This step completes the design of your Http control node.

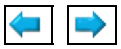
At run time, the dynamic property that you defined will override the static property defined using the Property Editor.

Specifying Http Control Properties

Most aspects of a Http control can be configured from the Properties Editor 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 Property Editor, or, you can change the properties in the Source View of the Http control's JCX file. For more information on the JCX file for the Http control, see The JCX file for the Http control.

When you modify properties for your Http control using the Property Editor, your changes are reflected in the Source View of the control's JCX 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 Property Editor. 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 Table 9−1, 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:

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 Property Editor.	`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)`
		`void setConnectionRetrycount(int retryCount)`

Setting Connection Retry Count.	This method defines the number of times your Http control will try to establish connection with the target.	
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 passwor`
Configuring Server−side SSL	This property allows you to configure server−side Secure Socket Layer authentication process.	`void setServerSideSSL(String trustStoreLocation, boolean hostVerificati`
Configuring Client−side SSL	This property allows you to set client−side authentication.	`void setClientSideSSL(String keyStoreType, String keyStoreLocation, Str`
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(ParametersDocument parameters,String`
Setting Headers for Http Post	This method allows you to set the header	`Void setHeadersForHttpPost(HeaderDocument headers)`

	properties for the Http Post mode.	

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 JCX 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 JCX 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.

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://

   <xs:element name="HttpControlProperties">

      <xs:complexType>

         <xs:sequence>

            <xs:element name="URLName" type="xs:string"/>

         </xs:sequence>

      </xs:complexType>

   </xs:element>

</xs:schema>
```

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>
```

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 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 that 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.

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/Ht

   <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"                    minOccur
                </xs:sequence>
             </xs:complexType>
          </xs:element>
       </xs:sequence>
    </xs:complexType>
 </xs:element>
</xs: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>
```

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.

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/Ht

   <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>
```

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>
```

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.

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/Ht
   <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>
```

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>
```

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.

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"

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

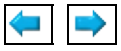
Method: String getResponseBodyAsString / XmlObject getResponseBodyAsXml / byte[ ] getResponseBodyAsBytes

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 WebLogic Workshop log properties file. You can find the workshop log properties file, workshopLogCfg.xml, in the *WL_HOME\weblogic81\common\lib* folder.

To log all the debug statements for HttpControlImpl and HttpResource class files, add the following lines to the workshopLogCfg.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.

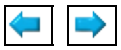
Http Control Caveats

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.



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 WebLogic Workshop log properties file. You can find the workshop log properties file, workshopLogCfg.xml, in the *WL_HOME\weblogic81\common\lib* folder.

To log all the debug statements for HttpControlImpl and HttpResource class files, add the following lines to the workshopLogCfg.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.

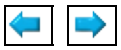
Http Control Caveats

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.



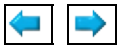
The Http Event Generator

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.

For more information on the Http Event Generator, see Event Generators in *Managing WebLogic Integration Solutions*.



WLI JMS Control

Note: The WLI JMS control is available in WebLogic Workshop 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 WebLogic Workshop 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 is an extension of the JMS control, providing 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

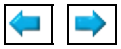
Describes scenarios in which the WLI JMS control may not be used.

Creating a New WLI JMS Control

Describes how to create and configure a WLI JMS control.

Using an Existing WLI JMS Control

Describes how to use an existing WLI JMS control from within a web service or business process.



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:

@jc:jms−headers Annotation

@jc:jms−property Annotation

Related Topics

WLI JMS Control

WLI JMS Control Interface

@jc:jms−headers Annotation

@jc:jms−property 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 WebLogic Workshop 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 @jc:jms property. Also, specify the send−type attribute as queue. To learn how to create a WLI JMS control, see Creating a New 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:

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 @jc: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 @jc:jms 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 @jc:jms annotation. Also, specify the receive−type 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:

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 @jc:jms property. Also, specify the send−type 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 receive−jndi−name attribute of the WLI JMS control's @jc:jms annotation. Also, specify the receive−type 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.

The following is an example of this scenario:

Related Topics

Overview: Messaging Systems and JMS

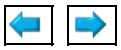
Messaging Scenarios Not Supported by the WLI JMS Control

WLI JMS Control Interface

@jc:jms Annotation

@jc:jms−headers Annotation

@jc:jms−property Annotation



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 WebLogic Workshop 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.

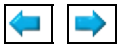
A business process must be performing work on behalf of a specific client and, in asynchronous situations, as part of a specific conversation. When an unsolicited messages is received from a queue, it is not possible for the WLI JMS control to determine the appropriate conversation or client with which 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.

Related Topics

Overview: Messaging Systems and JMS

Messaging Scenarios Supported by the WLI JMS Control



Creating a New 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. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.

3. Choose **WLI JMS** to display the **Insert Control − WLI JMS** dialog
4. In **Step 1**, in the **Variable name for this contro**l field, enter the name for your JMS control.
5. In **Step 2**, select the **Create a new WLI JMS control to use** radio button.
6. In the **New JCX name** field, enter the name of the new file.
7. 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 runtim**e check box. For more information about control factories, see Control Factories: Managing Collections of Controls.
8. In **Step 3**, from the **Message typ**e 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.
9. From 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.
10. In the **send−jndi−name** 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.
11. From the **JMS receive destination typ**e 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.
12. In the **receive−jndi−name** 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.
13. In the **connection−factory** 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.
14. Click **Create**. Alternatively, you may create a WLI JMS control JCX file manually. For example, you may copy an existing WLI JMS control JCX file and modify the copy.

WLI JMS Control Methods

To learn about the methods available on the WLI JMS control, see the WliJMSControl Interface.

The JCX File for a WLI JMS Control

When you create a new WLI JMS control, you create a new JCX file in your project. The following is an example JCX file:

```
package FunctionDemo;

import com.bea.control.*;
import com.bea.xml.*;
import java.io.Serializable;

/**
 *   @jc:jms send-type="queue" send-jndi-name="myqueue.async.request"
 *   receive-type="queue" receive-jndi-name="myqueue.async.response"
 *   connection-factory-jndi-name="weblogic.jws.jms.QueueConnFactory"
 */
public interface SimpleQueueControl extends
 WliJMSControl,com.bea.control.ControlExtension
{
    /**
     * this method will send a javax.jms.TextMessage to send-jndi-name
     */
    public void sendTextMessage(XmlObject payload);
    /**
     * this method will send a javax.jms.TextMessage to send-jndi-name
     */
    public void sendAnotherTextMessage(String payload);


    /**
     * If your control specifies receive-jndi-name,
     * that is your process expects to receive messages
     * from this control, you will need to implement callback handlers.
     */
    interface Callback extends WliJMSControl.Callback
    {
        /**
         * Define only 1 callback method here.
         *
         * This method defines a callback that can handle
         * text messages from receive-jndi-name
         */
        public void receiveTextMessage(XmlObject payload);
    }
}
```

The JCX 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 JCX 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 JCX 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 Properties Editor in Design View. These properties are encoded in the JMS control's JCX file as attributes of the @jc:jms annotation. To retrieve current parameter settings, use the getControlProperties() method. (Note that 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 detailed information on the @jc:jms annotation and its attributes, see @jc:jms Annotation.

You can also use the ControlContext interface for access to a control's properties at run time and for handling control events. 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 JPD file, or as annotations on its interface, callback, or method declarations in a JCX file.

To learn how to create, configure and register JMS queues, topics and connection factories, see Programming WebLogic JMS at the following URL:

http://edocs.bea.com/wls/docs81/jms/index.html

Two queues are configured when WebLogic Workshop 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. See the WLI JMS Control Caveats section below for more information.

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 Property Editor 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 @jc: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 message (unsuccessfully) forever. For information on setting the redelivery limit, see the"Programming WebLogic JMS at http://edocs.bea.com/wls/docs81/jms/index.html.

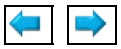
Related Topics

Overview: Messaging Systems and JMS

WLI JMS Control Interface

@jc:jms−headers Annotation

@jc:jms−property Annotation



Using an Existing WLI JMS Control

This topic describes how to use an existing WLI JMS control in your web service.

To learn about WLI JMS controls, see WLI JMS Control.

To learn how to create a WLI JMS control, see Creating a New WLI JMS Control.

Using an Existing WLI JMS Control

All controls follow a consistent model. Therefore, most aspects of using an existing WLI JMS control are identical to using any other existing control. To use an existing WLI JMS control:

1. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

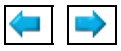
   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **WLI JMS** to display the **Insert Control − WLI JMS** dialog.
4. In the **Variable name for this control** field, type the variable name used to access the existing WLI JMS control instance from your business process. The name you enter must be a valid Java identifier.
5. In the Step 2 pane, choose the **Use a WLI JMS control already defined by a JCX file** radio button.
6. Click **Browse** to browse for existing WLI JMS controls. The Select dialog is displayed. When you find the control you want to use, select it and click **Select**.
7. Click **Create**.

Related Topics

Overview: Messaging Systems and JMS

WliJMSControl Interface



MQSeries Control

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

MQSeries is a middleware product from IBM that runs on multiple platforms and enables applications to send messages to other 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 do not have to be executing at the same time. MQSeries takes care of all the storage, logging and communications details required to guarantee delivery of the message to the destination queue.

The MQSeries control enables WebLogic Integration business processes to work with MQSeries for sending and receiving messages, to and from 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. These properties can also be dynamically set 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.

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

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

Topics Included in This Section

Before You Add an MQSeries Control

Describes the tasks to be carried out before a new MQSeries control can be created.

Creating and Configuring a New Instance of MQSeries Control

Describes the tasks to be carried out to create and configure a new MQSeries control.

Using Exit Implementation

Describes how to implement the Exit functionality in the MQSeries control

Understanding Transaction Management

Describes the modes of transaction management that are supported within MQSeries control.

Using Message Descriptors

Describes how the message descriptor attributes of the message can be set and retrieved.

Sending and Receiving Messages

Describes the methods used to send and receive messages.

Working with MQSeries Message Descriptor Format

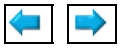
Describes the method used to send messages of built−in MQSeries formats.

Setting Dynamic Properties

Describes how to modify the MQSeries control properties at run time.

Using the MQSeries Event Generator

Describes the MQSeries Event Generator in brief, with a reference to more information.



Before You Add an MQSeries Control

Before you add an MQSeries Control to WebLogic Platform, you need to 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. Enable MQSeries control logging, by adding the following lines to the workshopLogCfg.xml file:

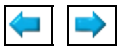
   <category "name=com.bea.control.MQControl">
   <!−− NOTE: DO NOT CHANGE THIS PRIORITY LEVEL W/O WLI SUPPORT APPROVAL −−>
   <!−− Debug−level log information is frequently the only tool available to diagnose failures! −−>
   <priority value="debug"/>
   <appender−ref ref="SYSLOGFILE"/>
   <appender−ref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

   The MQSeries Control uses the workshop debugger for logging messages.

   Note: You need to enable logging before you start the WebLogic server.

4. Import the com.ibm.mq.jar file from the MQSeries client installation, into the Libraries folder of the WebLogic Workshop application where the MQSeries control is used.

Go ahead and add a new MQSeries Control to send and receive messages.



Creating and Configuring a New Instance of MQSeries Control

This topic describes how to create and configure a new instance of MQSeries control.

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

1. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.

3. Choose **MQSeries Control** to display the **Insert Control − MQSeries** dialog
4. In **Step 1**, in the **Variable name for this contro**l field, enter the name for your MQSeries control.
5. In **Step 2**, select the **Create a new MQSeries control to use** radio button.

Note: If you want to use an existing MQSeries Control, click the **Browse** button to select the JCX file from your system. When you use an existing MQSeries Control, the properties that were originally selected for the existing control will be populated in the JCX file. You cannot modify the properties using the **Insert Control − MQSeries Control** dialog. However, you can modify the dynamic properties during run time. For more information, see Setting Dynamic Properties.

6. In the **New JCX name** field, enter the name of the new file.
7. 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 runtim**e check box. For more information about control factories, see Control Factories: Managing Collections of Controls.
8. In **Step 3**, in the General tab, from the **Connection Type** drop−down list, select the type of connection that you want to establish. You can select either Bindings or TCP as your connection type. Using the Bindings mode, you can obtain connection to queue managers present in the local system only. Using the TCP connection mode, you can obtain connections to remote queue managers also.
9. In the **MQ Pool Size** text box, specify the number of MQSeries connections to be maintained in the MQSeries connection pool.
10. In the **Connection Timeout (Seconds)** field, type the number of seconds after which the connection should time out.
11. From the **Require MQ Authorization** drop−down list, select either Yes or No. MQ authorization is applicable only in the TCP mode. If you require MQ authorization, the MQSeries user name and password must be provided in the Authorization tab.
12. By default, the **Implicit Transaction Required** option is selected. When selected, MQSeries control handles transactions implicitly for each Put and Get, individually, without the need for an explicit transaction boundary. When this option is not selected, you need to explicitly set the transaction boundaries. For more information, see Understanding Transaction Management.
13. In the **Default Queue Name** field, type in the default queue name which is to be used by the MQSeries control for sending and receiving messages.
14. In the **Connection** tab, in the **Queue Manager Name** field, specify the name of the Queue Manager

to which the connection is to be obtained.

15. If you have selected TCP as your connection mode, you need to specify TCP Settings as follows:
    a. In the **Host** field, type in the host name of The host name of the machine containing the queue manager to connect to.
    b. In the **Port** field, enter the port number on which the queue manager is available for connection.
    c. In the **Channel** field, type the MQSeries server connection channel configured in the queue manager.
    d. In the **CCSID** field, type the Coded Character Set to be used while when connection is established. The CCSID is used mainly for i18n (Internationalization) support.
16. Click the **Test Connection** button to ensure that the values entered are correct and that you are able to connect to the queue manager.
17. If you have requested MQ authorization, you need to specify the MQ user name and password in the Authorization tab.

Note: The Authorization tab is enabled only if you have selected the TCP connection mode.

18. In the Exits tab, in the **Send Exit Class** field, type in the fully qualified name of the class implementing the MQSeries MQSendExit interface.
19. In the **Receive Exit Class** field, type in the fully qualified name of the class implementing the MQSeries MQReceiveExit interface.
20. In the **Security Exit Class** field, type in the fully qualified name of the class implementing the MQSeries MQSecurityExit interface.

    For more information on the Exit functionality, see Using Exit Implementation.

Note: The Exits tab is enabled only if you have selected the TCP connection mode. However, the fields in this tab are not mandatory.

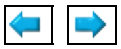
21. Click **Create**.

The JCX File for an MQSeries Control

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

```
package processes;
import com.bea.control.*;
import com.bea.xml.XmlCursor;
import com.bea.control.MQControl;
import com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument;
import com.bea.wli.control.mqDynamicProperties.MQDynamicPropertiesDocument;
import javax.resource.ResourceException;
import com.bea.xml.XmlObject;
/*
* A custom MQ control.
*/
/**
* @jc:MQConnectionType connectionType="Bindings"
* @jc:MQConnectionPoolProps mqPoolSize="20"
* @jc:ConnectionPoolTimeout conTimeout="3600"
* @jc:MQQueueManager queueManager="QM_bea_aruna"
* @jc:MQAuthorization requireAuthorization="No"
* @jc:TCPSettings host=""
```

```
        port="1414"
        channel=""
        ccsid="819"
        user=""
        password=""
        sendExit=""
        receiveExit=""
        securityExit=""

* @jc:DefaultQueue defaultQueueName="default"
* @jc:ImplicitTransaction implicitTransactionRequired="true" */
public interface newjcx extends MQControl, com.bea.control.ControlExtension {
/*
* A version number for this JCX. This will be incremented in new versions of
* this control to ensure that conversations for instances of earlier
* versions were invalid.
*/
static final long serialVersionUID = 1L;
}
The contents of the MQSeries control's JCX file depend on the selections made in the Insert MQS
```



Using Exit Implementation

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

To implement an exit, you 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 possibly alter the data sent to the queue manager by the WebSphere MQ Client for Java.

- MQReceiveExit

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

- MQSecurityExit

  The MQSeriesSecurityExit interface allows you to customize the security flows that occur when an attempt is made to connect 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

In order 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.MQSecurityExit 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,by
{
        return agentBuffer;
}
public byte[] receiveExit(MQChannelExit channelExit,MQChannelDefinition channelDefnition
{
        return agentBuffer;
}
public byte[] securityExit(MQChannelExit channelExit,MQChannelDefinition channelDefnitio
{
        return agentBuffer;
```

```
}
}
```

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

For a Send exit, the agentBuffer parameter contains the data that is about to be sent. For a Receive exit or a Security exit, the agentBuffer parameter contains the data that has just been received.

For the Send and Security exits, 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 in the WebLogic classpath, by editing the setDomainEnv.cmd file, which is present in the WebLogic domain directory. To do this, locate the following line in the setDomainEnv.cmd file:

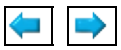
   set CLASSPATH=%ARDIR%\ant\ant.jar;%JAVA_HOME%\jre\lib\rt.jar

   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



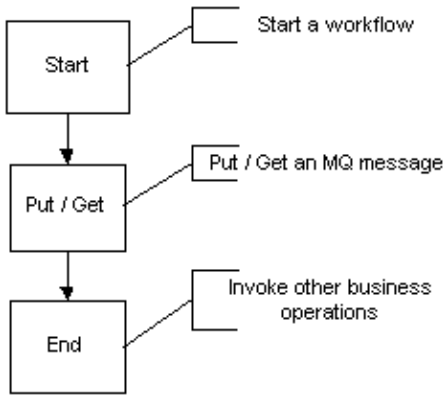
Understanding Transaction Management

There are two modes of transaction management supported by the MQSeries control, as follows:

- Implicit Transaction Management
- Explicit Transaction Management

Implicit Transaction Management

The implicit transaction mode 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 the how an implicit transaction is handled by the MQSeries control.

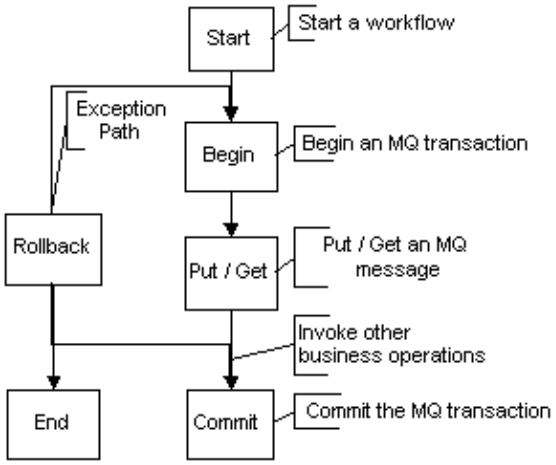


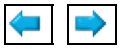
When you use implicit transaction, you cannot group several Get and Put functions together as a part of a transactional unit. Implicit transaction handles each Get or Put individually within a transaction boundary.

Explicit Transaction Management

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

The following flow diagram describes the process of creating a workflow using the explicit transaction mode.





Using Message Descriptors

A Message Descriptor is an attribute representing a property of the message, that is either being sent or received. For example, the Message Type of the message, the Message ID of the message, and the Priority of the message. For a detailed listing of all the message descriptors that are supported by the MQSeries control, see Table 11−1, "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 message that are retrieved from the queue. This facility is supported with the help of the MQMDHeaders document which is provided as an input to the putMessage and getMessage functions. The MQMDHeaders document is represented using an XMLbean, which conforms to the MQMDHeaders schema which is present in the MQSchemas.jar.

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

Table 11−1 Elements of the MQMDHeaders XML document

Element Name	Description	Permissible Values	Relevance
MessageType	Message 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	Message 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 while sending and receiving 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	Priority of the message	0−9	Put Request, Put Response, Get Response
Format	Format of the message	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 Characterset	Put Request, Put Response, Get Response
Persistence	Persistence property of the message	0−for a non−persistent message. 1−for persistent message	Put Request, Put Response, Get Response
Segmentation	Segmentation property of the message	0−for segmentation not allowed. 1−for segmentation allowed.	Put Request
Expiry	Expiry of the message	Any positive integer or −1 (for unlimited expiry)	g Request, Put Response, Get Response
UserId	User Id of the message	Any string	Put Request, Put Response, Get Response
MessageSequenceNumber	Message Sequence Number of the message	Any positive integer other than 0	Put Request, Put Response, Get Request, Get Response
GroupOptions	In the Put Request this element should be provided only if the message being Put is a group		Put Request, Get Response

	message. In the Get Response, this element appears only if the message retrieved is a group message.		
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
COA	Confirmation on Arrival. COA Report options COA−only the COA report without any data of the original message. COAWithData−the COA report with the first 100 bytes of the original message. COAWithFullData−the COA report with all the data of the original message	COA, COAWithData, COAWithFullData, None	Put Request
COD	Confirmation of Delivery COD Report options COD−only 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 data of the original message.	COD, CODWithData, CODWithFullData, None	Put request
Exception	Exception Report options Exception−only the Exception report without any data of the original message. ExceptionWithData−the Exception report with the first 100 bytes of the original message. ExceptionWithFullData−the Exception report with all the data of the original message.	Exception, ExceptionWithData, ExceptionWithFullData, None	Put request

Expiration	Expiration Report options Expiration−only the Expiration report without any data of the original message. ExpirationWithData−the Expiration report with the first 100 bytes of the original message. ExpirationWithFullData − The expiration report with all the data of the original message.	Expiration, ExpirationWithData, ExpirationWithFullData, None	Put request
Feedback	Feedback of the message	Postive integer value	Put Request, Put Response, Get Response
ReplyToQueueName	The queue to which the reports or the reply (only in the case of 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
WaitInterval	The interval to lapse (in milliseconds) before getting a message.	Any positive integer, or −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.
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	Specify whether data conversion is required for the message during a Get operation. This element needs to be set to True for retrieving messages of the EBCDIC characterset.	True or False	Get request

Table 11−2

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

Attributes of the MQMDHeaders document.

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"
<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" maxOcc
                </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="Reply
<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

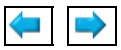
The following is a sample MQMDHeaders document that contains most of the message descriptors that you can set using MQSeries control:

```
<?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:Expiry>5000</even:Expiry>
<even:UserId>WebLogic</even:UserId> <even:MessageSequenceNumber>1</even:MessageSequenceNumber>
        <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: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, by using XML beans. An example for setting the MQMDHeader element values prior to the putMessage function call is as follows:

```
headers =
com.bea.wli.control.mqmdHeaders.MQMDHeadersDocument.Factory.newInstance();com.bea.wli.control.mqmdHeade
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 using the MQSeries control, by using the Put and Get functions. You can send and receive messages in the form of Bytes, String or XML data.

Sending Messages

Depending on the data type of the message that you want to send, you can use any of the following putMessage functions:

- 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 needs to be Put. If no value is provided during 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.

The return value of the function is the MQMDHeadersDocument representing the MQMD attributes of the message Put into the queue.

Using the putMessage Function In a Business Process

The following sample 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, provide a name for the new method.
3. Click Add, and select **MQMDHeadersDocument** from the XML Types list. Provide 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, and select String from the Java datatype list. Provide 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, and select **String** from the Java datatype list. Provide 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 need to provide variable names for all the 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 to store the output of the putMessageAsString function which is the MQMDHeaders document, which represents the attributes of the message that was sent.

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

Receiving Messages

Depending on the data type of the message that you want to receive, you can use any of the following getMessage functions:

- 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 is the queue from which the message is to be received. If no value is provided during runtime, that is, if the value is null, the default queue name mentioned in the control property is used.

The second parameter to this 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 retrieving the message corresponding to the MQMD attributes specified in the document. The MQMD attributes of the message obtained from the queue are updated in this XML bean object itself.

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 sample procedure describes how to add any MQSeries Control getMessage function to a business process.

1. Open the Client Request node.
2. In the General Settings tab, provide a name for the new method.

3. Click Add, and select MQMDHeadersDocument from the XML Types list. Provide 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, and select String from the Java datatype list. Provide 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 need to provide variable names for the two variables. The variable type is pre−defined, based on the parameters to which you are assigning the variable.
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 to store the output of the getMessageAsString function which is a String that represents the message that was retrieved from the queue.

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

You can use similar steps to retrieve messages using getMessageAsBytes or getMessageAsXml functions.

Sending Group messages

You can send Group Messages by configuring your business process containing the putMessage function of the MQSeries control, within a loop configured using one of the While do, Do While, and For Each process nodes.

To send group messages, provide the GroupOptions element in the MQMDHeadersDocument. You need to provide this element in the input MQMDHeaders XML document only 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 which the IsLastMessage element has to be set to True.

If you specify a GroupId for the first message, 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

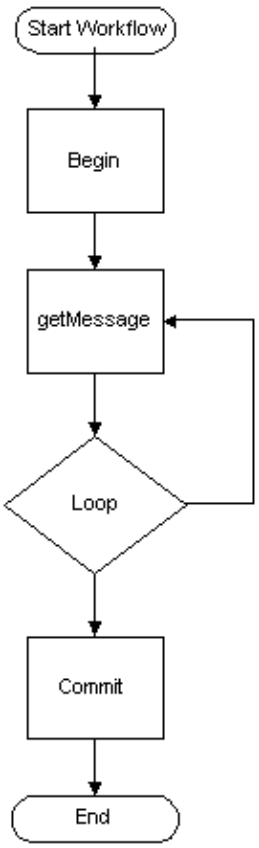
You can retrieve group Messages by configuring your business process containing the getMessage function of the MQSeries control, within a loop configured using one of the While do, Do While, and For Each process

nodes.

Setting the logicalorder Attribute

The MQSeries Control supports retrieving group messages in a logical order. To configure MQControl 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 if you are using the explicit transaction mode. The following figure depicts a sample workflow for retrieving group messages in logical order:



The loop can be continued 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 will 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 get all the messages of the group in their logical order (in the order of their message sequence number starting from 1 for the first message).

Setting the logicalOrder to False in the midst of getting group messages, when its value was True in the previous Get service call would disturb the logical ordering.

Setting the logicalOrder attribute to False or not providing this attribute in the Get request document would mean that the control would get the first message of the group as it appears on the queue irrespective of its

message sequence number.

The Get Request MQMDHeaders document for retrieving group messages in logical order and also wait for all messages of the group, looks as follows:

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

Setting the waitForAllMsgs Attribute

You can configure 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 MQControl 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.

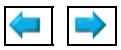
Setting the waitForAllMsgs attribute to False or not providing this attribute in the Get request document would mean that the control can still get group messages from the queue even when not all of the messages of the group are present in the queue.

Setting the GroupId element

The GroupId is an optional element that you can set under the GroupMessage element and its value may not be provided if the hexadecimal group id of the group message is not known. In case there are multiple group messages in the queue, the first group message appearing 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.

Retrieving Group Messages Using MessageSequenceNumber Element

Group Messages can also be retrieved by specifying the MessageSequenceNumber element and the GroupId. But this can be used 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. So the group messages can still be retrieved in a loop by providing the GroupId and incrementing the MessageSequenceNumber by 1 in each Get function call in the loop, the MessageSequenceNumber of the first message being 1.



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 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 MQSeries control you can send messages that correspond to both built−in MQSeries formats as well as user−defined formats. This is possible only by using the putMessageAsBytes function.

To send a message that conforms to an MQSeries Format, you need to write Java code in the process JPD file, as 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 the JPD 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 Palette, into the business process, just below the Client Request node.
3. Open the Perform Node in the Source View and add the following lines of code to it.

   ```
   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 "); /

   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;
}
```

These lines of code set 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 structure to a byte array output stream and returns an array of bytes. The values of the fields that have been given in this example can be modified as required. The actual message can be appended to this byte array at the end and can be Put into the MQSeries queue. This byte array can be provided as the first parameter to the putMessageAsBytes function which is added to the process JPD 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 JPD 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 Palette, into the business process, just below the Client Request node.
3. Open the Perform Node in the Source View and add the following lines of code to it.

   ```
   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;
}
```

These 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 that have been given in this example can be modified as required. The actual message can be appended to this byte array at the end and can be Put into the MQSeries queue. This byte array can be provided as the first parameter to the putMessageAsBytes function which is added to the process JPD 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 JPD 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.

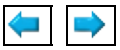
2. Drag and drop the Perform node from the Palette, into the business process, just below the Client Request node.
3. Open the Perform Node in the Source View and add the following lines of code to it.

   ```
   public void perform() throws Exception
   {
   putin.getMQMDHeaders().setFormat(MQC.MQFMT_RF_HEADER_2);
   bytmsg = getMQRFH2Header();
   }
   public byte[] getMQRFH2Header() throws Exception    { ByteArrayOutputStream bstream = new
   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);//
ostream.flush();
byte[] bArr = bstream.toByteArray();
return bArr;
}
```

These lines of code set 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 that have been given in this example can be modified as required. The actual message can be appended to this byte array at the end and can be Put into the MQSeries queue. This byte array can be provided as the first parameter to the putMessageAsBytes function which is added to the process JPD 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 MQSchemas.jar file.

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. You need only to provide a variable name for the variable. The variable type is 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 will 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 the Explicit Transaction mode, the setDynamicProperties function should always be called before the Begin function or after Commit or Rollback functions. If this sequence is not followed, the business process will throw an exception during runtime.

Schema of MQDynamicProperties

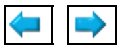
```
<?xml version="1.0"?>

<xs:schema xmlns="http://www.bea.com/wli/control/MQDynamicProperties" xmlns:xs="http://www.w3.o
<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 need to 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>
```



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, that is, Bytes, String and XML.

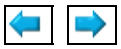
You can configure Event Generator channels for different data types, using a Message Broker channel name, which instructs that any message coming into the specified MQSeries queue will be published to that message broker channel.

Similar to the MQSeries control, the MQSeries Event Generator also provides two modes of connections, that is, 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.

For more information on the MQSeries Event Generator, see *Managing Integration Solutions* at the following location:

http://edocs.bea.com/wli/docs81/manage/index.html



Process Control

Note: The Process control is available in WebLogic Workshop only 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 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.

Editing and Testing a Dynamic Selector

Describes how to edit and test a dynamic selector for a Process control.

Using Dynamic Binding

Describes how to customize a Process control.

Overview: Process Control

The Process control is used to send requests to and receive callbacks from another business process. It's capabilities are similar to those of the Service Broker control. Unlike the Service Broker control, the Process control is only used to target other business processes in the same domain using Java/RMI (Remote Method Invocation). The target of the call can be dynamically specified. The Process control is typically used to call a subprocess from a parent process.

The first step in using a Process control is to create a JCX file. The JCX can be automatically generated from a target business process using WebLogic Workshop, or can be created using the control wizard. The methods and callbacks on the JCX correspond to operations and callbacks of the target business process. An instance of this JCX is used by a parent process to call the target process. Process control JCX 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.

Setting Process Control Properties

The Process 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 property settings, except for username and password, use the getProperties() method.

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

1. Properties dynamically bound using the jc: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 system folder of New Process Applications or in the system folder of the Schemas project.

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 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");
```

Some control properties can be specified both in annotations (statically) on the JCX file or dynamically. For example, the Process control allows you to specify the target process in the jc:location annotation at the top of the JCX 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.

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 JCX is invoked
- A start method is invoked again
- The property is programmatically set by calling setProperties or a setter method.

ProcessControl.reset() resets all dynamically set properties (in addition to all conversational state). Programmatically specified properties remain bound until reset() is invoked.

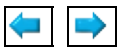
You can also use the ControlContext interface for access to a control's properties at run time and for handling control events. Property values set by a developer who is using the control are stored as annotations on the control's declaration in a JWS, JSP, or JPD file, or as annotations on its interface, callback, or method declarations in a JCX file.

Related Topics

Service Broker Control

Using Dynamic Binding

ProcessControl Interface



Creating a New Process Control

This topic describes how to create a new Process control.

To learn about Process controls, see Process Control.

Creating a New Process Control Using the Control Wizard

You can create a new Process control and add it to your business process by using the Insert Process dialog. If you are not in Design View, click the Design View tab.

To define a new Process control

1. Click Add on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **Process** to display the **Insert Control − Process** dialog.
4. In **Step 1**, in the **Variable name for this control** field, type the name for your Process control.
5. In **Step 2**, select the **Create a new Process control to use** radio button.
6. In the **New JCX name** field, type the name of the new file.
7. In **Step 3a**, select the business process you want to access by selecting the name of a business process (.jpd) file.
8. In **Step 3b**, select a start method from the **Start Method** menu. Only those start methods contained in the specified business process are displayed.
9. **Step 3c** is optional. Process controls allow you to decide at run time which one of multiple subprocesses to call using a dynamic selector. For simple cases, where you know at design time which subprocess you want to call, no selector is necessary.

   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:

   a. Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain−wide DynamicProperties.xml file. Choose TPM to bind lookup values to properties in the TPM repository.
   b. 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.
   c. Click **OK**.
10. Click **Create**. Alternatively, you may create a Process control JCX file manually. For example, you may copy an existing Process control JCX file and modify the copy.

Process Control Methods

To learn about the methods available on a Process control, see the ProcessControl Interface.

Example: Process Control Declaration

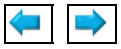
When you create a new Process control using the control wizard and drag a method from the control onto a business process, its declaration appears in the JPD file. The following code snippet is an example of what the declaration looks like:

```
/**
 * @common:control
 */
private FunctionDemo.callprocess callProcess;
```

Creating a Process Control from a Business Process

You can also create a Process control from an existing business process.

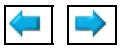
1. Right−click a JPD filename in the Application Pane and choose Generate Process Control.
2. A new JCX file is displayed, indented beneath the selected JPD file. The name is generated by appending PControl to the JPD name, For example, if you generate a Process control JCX file from CustomerMaster.jpd, the resulting JCX file is named CustomerMasterPControl.jcx.
3. Double click the Process control JCX file in the Application Pane to display the control in Design View.
4. Use the Property Editor to edit the dynamic selector as described in Editing and Testing a Dynamic Selector.



Editing and Testing a Dynamic Selector

Process controls allow you to decide at run time which one of multiple subprocesses to call using a dynamic selector. To edit and test a dynamic selector

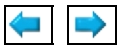
1. Display the business process in Design View that contains the Process control with the dynamic selector you want to edit or test.
2. Select the desired Control node in the business process.
3. Locate the selector property in the Property Editor and select the associated xquery parameter. Click the button next to the xquery field indicated by three dots (...). The Dynamic Selector query builder is displayed
4. Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain−wide DynamicProperties.xml file. Choose **TPM** to bind lookup values to properties in the TPM repository.
5. 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.
6. Click the **Test** tab to display the Source XML and Result XML areas, then click the **Test** button to test the execution of the query. Execution status messages are displayed at the bottom of the Query Builder.
7. Click **OK**.



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 about dynamic binding, see How the Service Broker Uses Dynamic Binding.



RosettaNet Control

Note: The RosettaNet control is available in WebLogic Workshop 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 WebLogic Workshop 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* at the following URL:

http://edocs.bea.com/wli/docs81/tpintro/index.html

Topics Included in This Section

Overview: RosettaNet Control

Describes the RosettaNet control.

Creating a RosettaNet Control

Describes how to create and configure a RosettaNet control.

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

Using Built−In Java Controls

Introducing Trading Partner Integration at http://edocs.bea.com/wli/docs81/tpintro/index.html

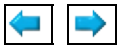
Trading Partner Management at http://edocs.bea.com/wli/docs81/manage/tpm.html

RosettaNetControl Interface

Tutorial: Building RosettaNet Solutions at http://edocs.bea.com/wli/docs81/tptutorial/rosettanet.html

Building RosettaNet Participant Business Processes

@jpd:rosettanet Annotation



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 RosettaNetControl Interface 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 http://edocs.bea.com/wli/docs81/tpintro/index.html.

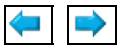
At run−time, the RosettaNet control relies on trading partner and service information stored in the TPM repository. To learn about the TPM repository, see *Introducing Trading Partner Integration* at http://edocs.bea.com/wli/docs81/tpintro/index.html. To learn about adding or updating information in the TPM repository, see Trading Partner Management in *Managing WebLogic Integration Solutions* at http://edocs.bea.com/wli/docs81/manage/tpm.html.

Related Topics

Creating a RosettaNet Control

Using a RosettaNet Control

Example: RosettaNet Control



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* at http://edocs.bea.com/wli/docs81/tpintro/index.html. To learn about RosettaNet controls, see RosettaNet Control.

To create a new RosettaNet control

1. If you are not in Design View, click the **Design View** tab.
2. On the **Controls** section of the **Data Palette**, click **Add**.

   Note: If the **Controls** tab is not visible in WebLogic Workshop, choose **View** > **Windows** > **Data Palette** from the menu bar. Instances of controls already available in your project are displayed in the Controls tab.

3. In the pop−up menu, click **Integration Controls** to display a drop−down list of controls that represent the resources with which your business process can interact.
4. Click **RosettaNet** to display the **Insert Control − Insert RosettaNet** dialog box.
5. In the Step 1 pane, in the **Variable name for this control** field, type the variable name used to access the new RosettaNet control instance from your business process. The name you enter must be a valid Java identifier.
6. In the Step 2 pane, select one of the following options:
   - **Use a RosettaNet control already defined by a JCX file**

     Enter the name of the JCX file, or click the **Browse** button to find and select it.

   - **Create a new RosettaNet control to use**

     Enter the name of the new JCX file to create.

7. If you are creating a new control, in the Step 3 pane, specify the following information:

   Note: Where applicable, the values entered here must match their corresponding settings in the TPM repository.

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/pipdirectory.
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.

8. Click the **Create** button.
9. If you are prompted, select a subfolder in which to save the JCX file.

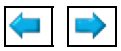
   A RosettaNet control instance is displayed in the **Controls** tab.

Related Topics

Overview: RosettaNet Control

Using a RosettaNet Control

Example: RosettaNet Control



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. To learn about WebLogic Workshop controls, see Using Built−In Java 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 RosettaNetControl Interface.

The RosettaNet control is a JCX file. To learn about using JCX files, see JCX Files: Extending Controls.

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:

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:

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:

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:

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.
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.RosettaNetContext;
```

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 the return type of your corresponding methods in your RosettaNet control definition file (JCX 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:*

```
public interface Callback
  {
    /**
    * @common:message-buffer enable="false"
    */
    public RosettaNetContextDocument sendReply(XmlObject msg);
    /**
    * @common:message-buffer enable="false"
    */
    public void sendReceiptAcknowledgement();
    /**
    * @common:message-buffer enable="false"
    */
    public void sendError(String msg);
  }

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 (@jc:rosettanet Annotation) and the DynamicProperties.xml file
2. properties set using the setProperties() method
3. properties set at the JCX instance level using the @jc:rosettanet Annotation annotation in the JPD
4. properties set at JCX class level using @jc:rosettanet Annotation annotation in the JCX

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 Property Editor 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. To learn about importing schemas, see How do I: Import Schemas into a Project Schemas Folder.
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.

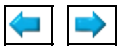
Related Topics

RosettaNet Control

Overview: RosettaNet Control

Creating a RosettaNet Control

Example: RosettaNet Control



Example: RosettaNet Control

For examples of how to use the RosettaNet control, see Tutorials: Building RosettaNet Solutions, which is located in the following directory:

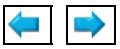
http://edocs.bea.com/wli/docs81/tptutorial/rosettanet.html

Related Topics

Overview: RosettaNet Control

Creating a RosettaNet Control

Using a RosettaNet Control



Service Broker Control

Note: The Service Broker control is available in WebLogic Workshop only 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.

Using Dynamic Binding

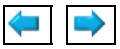
Describes how to dynamically set control attributes.

Creating a New Service Broker Control

Describes how to create a new Service Broker control by using the control wizard or by automatically generating the control from a business process or web service.

Editing and Testing a Dynamic Selector

Describes how to edit and test a dynamic selector for a Service Broker control.



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 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 JPD file in the Application pane and choosing **Generate WSDL File**. To learn more about WSDL files, see WSDL Files: Web Service Descriptions.

The first step in using a Service Broker control is to create a JCX file. The JCX can be automatically generated from a target service (web service, business process, or WSDL file) using WebLogic Workshop, or can be created using the Insert Service Broker dialog box. The methods and callbacks on the JCX correspond to operations and callbacks of the target service. An instance of this JCX is used by a parent service to call the target service. Service Broker control JCX files can have selector annotations only on start methods or for stateless target services on any method.

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.

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(). These setter methods are inherited from the Web Service control interface.

```
package com.bea.control;

public interface ServiceBrokerControl extends ServiceControl {

    void setProperties(ServiceBrokerControlProperties props)
        throws Exception;
}
```

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.

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

1. properties dynamically bound using the jc:selector tag 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.

The setProperties() method uses this XML Bean 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 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");
```

Some control properties can be specified both in annotations (statically) on the JCX file or dynamically. For example, the Service Broker control allows you to specify the http−url of the target service in the jc:location annotation at the top of the JCX 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.

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 JCX 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 state). Programmatically specified properties remain bound until reset() is invoked.

You can also use the ControlContext interface for access to a control's properties at run time and for handling control events. 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 JPD file, or as annotations on its interface, callback, or method

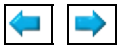
declarations in a JCX file.

Related Topics

Using Dynamic Binding

Creating a New Service Broker Control

ServiceBrokerControl Interface



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.

How the Service Broker Uses Dynamic Binding

The following scenario shows how the Service Broker uses dynamic binding. POService.jpd 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.

@jc:selector Tag

The method−level annotation, @jc: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:

```
/**
 * @jc:conversation phase="start"
 * @jc: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:

```
/**
 * @jc:conversation phase="start"
 * @jc:selector xquery ::
 *     lookupTPMProperties($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* at http://edocs.bea.com/wli/docs81/tpintro/index.html. To learn about adding or updating information in the TPM repository, see Trading Partner Management in *Managing WebLogic Integration Solutions* at http://edocs.bea.com/wli/docs81/manage/tpm.html. 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 JCX 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 in *Managing WebLogic Integration Solutions* at http://edocs.bea.com/wli/docs81/manage/processconfig.html.

DynamicProperties.xml contains a sequence of <control> elements, one for each dynamic control JCX file. Each <control> element has a name attribute whose value is the fully−qualified Java class name of a JCX file. Nested inside the <control> element is a sequence of <key> elements which map arbitrary string values to dynamic properties, as shown in the following example:

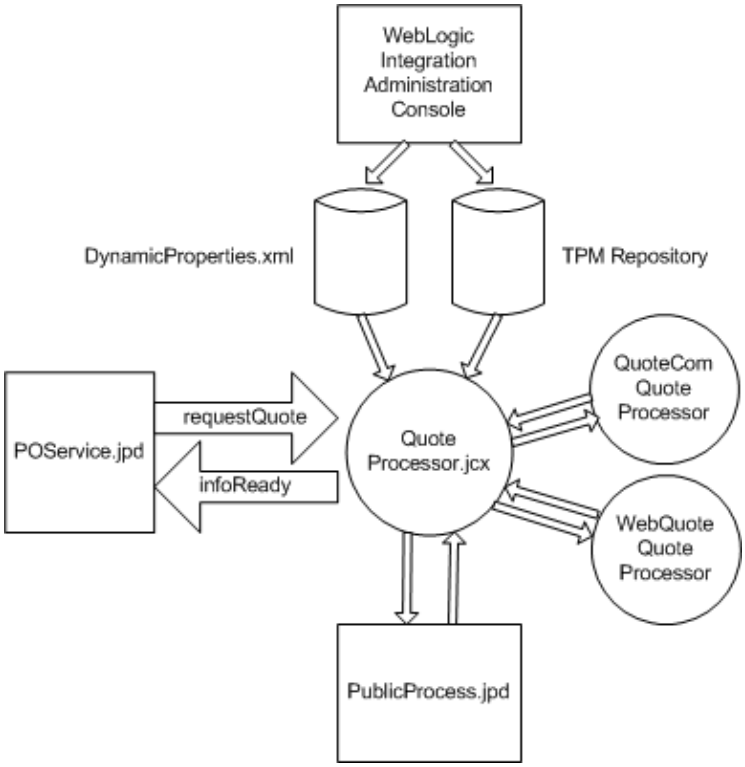
```
<DynamicProperties
      xmlns="http://www.bea.com/wli/control/dynamicProperties">

  <control name="quote.QuoteProcessor"
           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>

  <control name="quote.InternalQuote"
controlType="ProcessControl">
```

```
    <key value="OurQuote">
      <ProcessControlProperties>
        <targetURI>http://acme/myApp/PublicProcess.jpd</targetURI>
      </ProcessControlProperties>
    </key>
  </control>
</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. The following figure shows the components that participate in the dynamic binding:



To achieve the required dynamic binding to the target service, the business process defined in POService.jpd uses a Service Broker control, QuoteProcessor.jcx, to call the quote service. Since the target is dynamically specified, the @jc:location tag is not used. The Service Broker control is defined by the following JCX 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);
    }

    /**
     * @jc:conversation phase="start"
     * @jc: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 selector XQuery expression tagged on the start method of the Service Broker control interface (@jc: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 JCX 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

Creating a New Service Broker Control

This topic describes how to create a new Service Broker control.

To learn about Service Broker controls, see Overview: Service Broker Control.

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 (JPD), the business process must be in the current WebLogic Workshop application.

If you are not in Design View, click the Design View tab.

To define a new Service Broker control:

1. Click **Add** on the **Controls** tab to display a list of controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible in WebLogic Workshop, click **View** > **Windows** > **Data Palette** from the menu bar.

2. Choose **Integration Controls** to display the list of controls used for integrating applications.
3. Choose **ServiceBroker** to display the **Insert Control − ServiceBroker** dialog.
4. In **Step 1**, in the **Variable name for this control** field, type the name for your Service Broker control.
5. In **Step 2**, select the **Create a new Service Broker control to use** radio button.
6. In the **New JCX name** field, type the name of the new file.
7. In **Step 3a**, browse for the file (.jpd, .jws, or .wsdl) representing the specific service you want to access.
8. In **Step 3b**, select a start method from the **Start Method** menu. Only those start methods contained in the specified service are displayed.
9. In **Step 3c**, enter a query in the **Query** field or click the **Query Builder** button to display the **Dynamic Selector** query builder. This step is optional. If you only plan to use the setProperties() method to define properties, you do not need to define a 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 **LookupControl** to bind lookup values to dynamic properties specified in a domain−wide DynamicProperties.xml file. Choose **TPM** to bind lookup values to properties in the TPM repository.
   b. 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.
   c. Click **OK**. The **Insert Service Broker** dialog is displayed with the query shown in the Query field.

10. Click **Create**. Alternatively, you may create a Service Broker control JCX file manually. For example, you may copy an existing Service Broker control JCX file and modify the copy.

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 JPD filename in the Application Pane and choose **Generate Service Broker Control**. The **Dynamic Selector Generation** dialog is displayed.
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.

   If you invoked the **Dynamic Selector** query builder, perform the following steps to build and test a query:

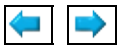
   a. Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain−wide DynamicProperties.xml file. Choose **TPM** to bind lookup values to properties in the TPM repository.
   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 JCX file is displayed, indented beneath the selected JPD file. The Service Broker control JCX file is named using a prefix of SB to help distinguish it from Service controls. For example, if the associated JPD file is MyProcess.jpd, the generated Service Broker control JCX file is named MyProcessSBControl.jcx.

Related Topics

Overview: Service Broker Control

Using Dynamic Binding

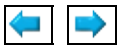
ServiceBrokerControl Interface



Editing and Testing a Dynamic Selector

Service Broker controls allow you to decide at run time which one of multiple subprocesses to call using a dynamic selector. To edit and test a dynamic selector

1. Display the business process in Design View that contains the Service Broker control with the dynamic selector you want to edit or test.
2. Select the desired Control node in the business process.
3. Locate the selector property in the Property Editor 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. Select the type of lookup function for the query by choosing the **LookupControl** or **TPM** radio button. Choose **LookupControl** to bind lookup values to dynamic properties specified in a domain−wide DynamicProperties.xml file. Choose **TPM** to bind lookup values to properties in the TPM repository.
5. 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.
6. Click the **Test** tab to display the Source XML and Result XML areas, then click the **Test** button to test the execution of the query. In addition to the XML elements displayed, you can also select Java class types as a source or result. Execution status messages are displayed at the bottom of the Query Builder.
7. Click **OK**.



TPM Control

Note: The TPM control is available in WebLogic Workshop only if you are licensed to use WebLogic Integration.

The TPM (trading partner management) 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.

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

Topics Included in This Section

Overview: TPM Control

Describes the TPM control.

Creating a TPM Control

Describes how to create a TPM control.

Using a TPM Control

Describes how to use an existing TPM control from within a business process or web service.

Example: TPM Control

Provides an example of how to use the TPM control.

Related Topics

Using Built−In Java Controls

Introducing Trading Partner Integration at http://edocs.bea.com/wli/docs81/tpintro/index.html

Trading Partner Management at http://edocs.bea.com/wli/docs81/manage/tpm.html

TPMControl Interface

Overview: TPM Control

The TPM control allows WebLogic Workshop 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 bulkloader command line utility. To learn about the TPM schema, see TPM Schema in *Managing WebLogic Integration Solutions* at http://edocs.bea.com/wli/docs81/manage/tpmschema.html.

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 *Managing WebLogic Integration Solutions* at http://edocs.bea.com/wli/docs81/manage/tpm.html.

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.

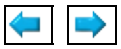
Related Topics

TPM Control

Creating a TPM Control

Using a TPM Control

Example: TPM Control



Creating a TPM Control

This topic describes how to create a new TPM control. To learn about TPM controls, see TPM Control.

To create a new TPM control

1. If you are not in Design View, click the **Design View** tab.
2. On the **Controls** section of the **Data Palette**, click **Add**.

   Note: If the **Controls** tab is not visible in WebLogic Workshop, choose **View** > **Windows** > **Data Palette** from the menu bar. Instances of controls already available in your project are displayed in the Controls tab.

3. In the pop−up menu, click **Integration Controls** to display a drop−down list of controls that represent the resources with which your business process can interact.
4. Click **TPM** to display the **Insert Control − Insert TPM** dialog box.
5. In the **Variable name for this control** field, 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.
6. Click the **Create** button.

   A TPM control instance is displayed in the **Controls** tab.

Related Topics

TPM Control

Overview: TPM Control

Using a TPM Control

Example: TPM Control

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 TPM Control Interface.

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. To learn about importing schemas, see Importing Schemas.
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.

Related Topics

TPM Control

Overview: TPM Control

Creating a TPM Control

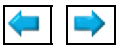
Example: TPM Control

TPMControl Interface

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/docs81/tptutorial/ebxml.html



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 WebLogic Workshop, the Task control and the Task Manager 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 WebLogic Workshop graphical design interface.

Creating a New Task Worker Control

Describes how to create a new Task Worker control using the WebLogic Workshop graphical design interface.

Using Task and Task Worker 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

TaskControl Interface

TaskWorkerControl Interface

Worklist Control Annotations

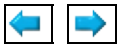
Using the Worklist at http://edocs.bea.com/wli/docs81/worklist/index.html

Tutorial: Building a Worklist Application at http://edocs.bea.com/wli/docs81/wltutorial/index.html

Worklist Administration in *Managing WebLogic Integration Solutions* at http://edocs.bea.com/wli/docs81/manage/worklist.html

Using Built−In Java Controls

BEA WebLogic Integration Javadoc at http://edocs.bea.com/wli/docs81/javadoc/index.html



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 Worker controls in WebLogic Workshop
- 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:

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:

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 Worker 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.

Related Topics

Creating a New Task Control

Creating a New Task Worker Control

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. To learn about factories, see "Using Task Control Factories" in Advanced Topics in *Using the Worklist*, which is located at the following URL:

http://edocs.bea.com/wli/docs81/worklist/advanced.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.

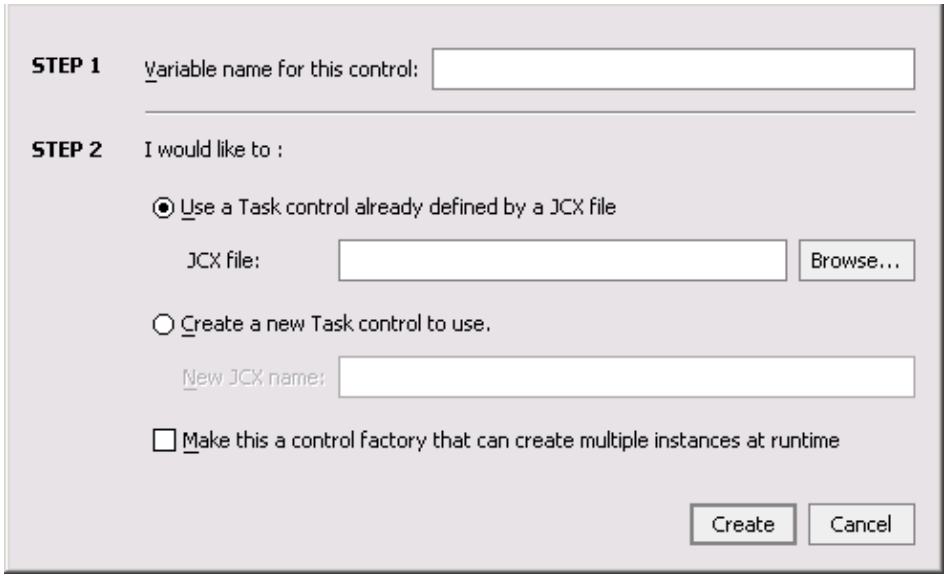
You can customize Task controls for different business purposes, by adding new operations or callbacks, or by altering the signatures of existing operations or callbacks.

To create a new Task control:

1. Open your WebLogic Integration application in WebLogic Workshop.
2. In the **Application** pane, double−click the business process (JPD 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. On the **Data Palette**, in the **Controls** tab, click **Add** > **Integration Controls** to display a list of integration controls that represent the resources with which your business process can interact.

   Note: If the Controls tab is not visible, from the menu bar, click **View** > **Windows** > **Data Palette**.

4. Choose **Task**. The **Insert Control** dialog box is displayed.



5. In the **Insert Control** dialog box (**Step 1**), enter a name for the instance of this control. The name you enter must be a valid Java identifier.
6. In the **Insert Control** dialog box (**Step 2**), select one of the following options:
   - Use a Task control already defined by a JCX file.

Enter a filename for the Task control in the **JCX file** field, or click **Browse** to find the JCX file in your file system.

- Create a new Task control to use.

  Enter a filename in the **New** JCX name field.

7. 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.

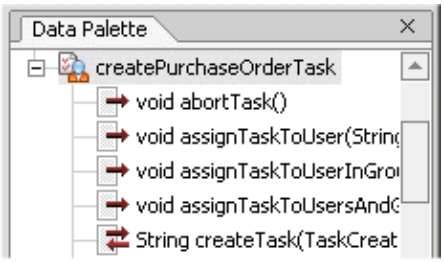
   To learn about factories, see "Using Task Control Factories" in Advanced Topics in *Using the Worklist*, which is located at the following URL:

   http://edocs.bea.com/wli/docs81/worklist/advanced.html

8. Click **Create**. A new Task control and an instance of it are created and the **Insert Control** dialog box is closed.

   A new JCX file is created and displayed in the **Application** tab in WebLogic Workshop. (You can double−click any JCX 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**.

9. To display the base methods provided on a Task control, expand the control instance by clicking the + beside its name on the **Data Palette**.



10. 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 Worker Controls in Business Processes.

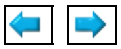
11. After you create a Task control in your business process, you can view and edit the properties of the control type or the instance of that control type in the **Property Editor**. The control type is represented as a JCX file in the **Application** pane and the instance is represented in the **Data Palette**.

    Task Instances have data values associated with them, many of which are set when the task is created. You can use the **Property Editor** 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.

To learn about factories, see "Using Task Control Factories" in Advanced Topics in *Using the Worklist*, which is located at the following URL:

http://edocs.bea.com/wli/docs81/worklist/advanced.html

Note: To learn how to use the **Property Editor** for specifying properties for control types versus control instances, see Setting Control Properties.



Creating a New Task Worker Control

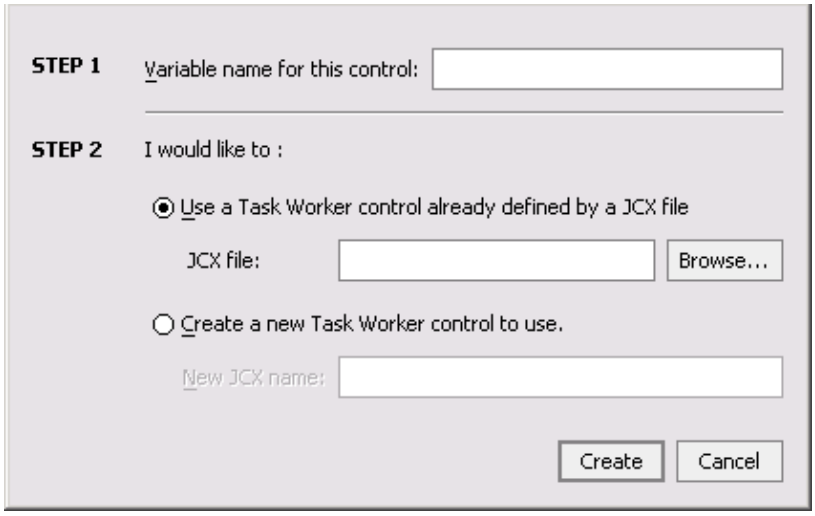
The Task Worker control allows specified users to acquire ownership of Tasks, work on them, and complete them. It also provides administrative privileges, such as starting, stopping, deleting, and assigning. Access to the Task Worker control can be done with a business process or through a user interface (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. Open your WebLogic Integration application in WebLogic Workshop
2. If you are not in **Design View**, click the Design View tab.
3. On the **Data Palette**, in the **Controls** tab, click **Add** > **Integration Controls**. A list of controls representing the resources with which your business process can interact is displayed.

   Note: If the **Controls** tab is not visible, from the menu bar, click **View** > **Windows** > **Data Palette**.

4. Choose **Task Worker**. The **Insert Task Worker** dialog box is displayed.

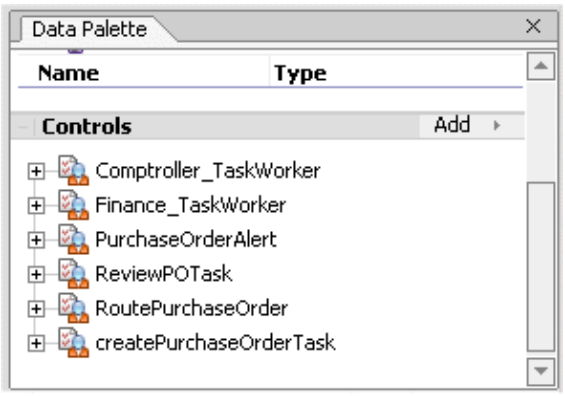


5. In the **Insert Control** dialog box (**Step 1**), enter a name for the instance of this control. The name you enter must be a valid Java identifier.
6. In the **Insert Control** dialog box (**Step 2**), select one of the following options:
   - To use a Task Worker control already defined by a JCX file, in the **JCX file** field, enter a filename for the Task Worker control, or click **Browse** to find the JCX file in your file system.
   - To Create a new Task Worker control to use, in the **New** JCX name field, enter a filename.
7. Click **Create** to close the **Insert Control** dialog box.

   When you click create, the control JCX file is displayed in the **Application** tab. In both **Design** and **Source** View, you can double−click any JCX 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 Worker Controls in Business Processes.

Using Task and Task Worker Controls in Business Processes

Before you begin working with the Task and Task Worker controls, you should be familiar with the features and components of the Worklist. To learn more about the Worklist, see Using the Worklist, which is located at the following URL:

http://edocs.bea.com/wli/docs81/worklist/index.html

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 WebLogic Workshop, 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.

Related Topics

Tutorial: Building a Worklist Application at http://edocs.bea.com/wli/docs81/wltutorial/index.html

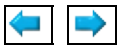
Introduction in *Using the Worklist* at http://edocs.bea.com/wli/docs81/worklist/intro.html

Using Worklist Controls in *Using the Worklist* at http://edocs.bea.com/wli/docs81/worklist/controls.html

Creating and Managing Worklist Tasks in *Using the Worklist* at http://edocs.bea.com/wli/docs81/worklist/tasks.html

Advanced Topics in *Using the Worklist* at http://edocs.bea.com/wli/docs81/worklist/advanced.html

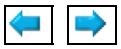
"Using the Task Control Property Editor" in "Using Task and Task Worker Controls in Business Processes" in Using Worklist Controls in *Using the Worklist* at http://edocs.bea.com/wli/docs81/worklist/controls.html



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/docs81/wltutorial/index.html

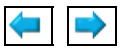


Using Control Factories

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, Application View, 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.

For more information about control factories, see Control Factories: Managing Collections of Controls.



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:

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.

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:

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 bea.com.data package in the Java Class Reference.

Related Topics

Guide to Building Business Processes

ebXML Control

RosettaNet Control

Introducing Trading Partner Integration at http://edocs.bea.com/wli/docs81/tpintro/index.html

WebLogic Workshop Reference

Java Class Reference