



BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Copyright © 2003 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software–Restricted Rights Clause at FAR 52.227–19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227–7013, subparagraph (d) of the Commercial Computer Software—Licensing clause at NASA FAR supplement 16–52.227–86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E–Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Table of Contents

Working with Java Controls.....	1
Getting Started with Java Controls.....	3
Invoking a Control Method.....	10
Handling Control Callbacks.....	12
Handling Control Method Exceptions.....	14
Control Factories: Managing Collections of Controls.....	15
Using Built-In Java Controls.....	18
Working with Built-In Controls.....	19
Timer Control.....	22
Overview: Timer Controls.....	23
Creating a New Timer Control.....	24
Using a Timer Control.....	26
Specifying Time on a Timer Control.....	28
Database Control.....	31
Overview: Database Controls.....	34
Creating a New Database Control.....	35
Adding a Method to a Database Control.....	37
Returning a Single Value from a Database Control Method.....	39
Returning a Single Row from a Database Control Method.....	40
Returning Multiple Rows from a Database Control Method.....	43
Modifying Data with a Database Control.....	48
Returning XMLBeans from a Database Control.....	49
Parameter Substitution in @jc:sql Statements.....	55

Table of Contents

Automatically Generating Primary Key Values with Database Controls.....	59
Stored Functions.....	61
Stored Procedures.....	63
Limiting the Size of Returned Data.....	67
Declaring Exceptions in Database Control Methods.....	69
Designing a Database Control.....	70
RowSet Control.....	73
Mapping Database Field Types to Java Types in the Database Control.....	79
Configuring a Data Source.....	81
Web Service Control.....	82
Overview: Web Service Controls.....	84
Creating a New Web Service Control.....	86
Declaring Compliance with a WSDL File.....	88
Specifying the Default Protocol and Message Format.....	89
Specifying Conversation Shape.....	91
Buffering Methods and Callbacks.....	92
Defining Java to XML Translation with XML Maps.....	95
EJB Control.....	97
Overview: Enterprise JavaBeans and EJB Controls.....	98
Creating a New EJB Control.....	100
Using an EJB Control.....	104
Handling EJB Exceptions.....	107
JMS Control.....	110

Table of Contents

Overview: Messaging Systems and JMS.....	112
Creating a New JMS Control.....	115
Configuring a JMS Control.....	117
Specifying the Message Body.....	119
Specifying Message Headers and Properties.....	121
Supported and Unsupported Messaging Scenarios.....	123
JCX Files: Extending Controls.....	128
Building Custom Java Controls.....	130
Working with Custom Controls.....	131
Source Files for Java Controls.....	133
Adding Portal Controls to Java Page Flows.....	136
Using Portal Controls.....	138
Portal Control Properties.....	143
Portal Control Declaration.....	145
Portal Control Security.....	146
Group Provider Control.....	147
Profile Control.....	148
Property Control.....	149
Rules Executor Control.....	150
Rules Manager Control.....	151
User Info Control.....	152
User Login Control.....	153
User Provider Control.....	154

Table of Contents

Click Content Event Control.....	156
Display Content Event Control.....	157
Generic Event Control.....	158
Generic Tracking Control.....	159
Rule Event Control.....	160
Session Login Event Control.....	161
User Registration Event Control.....	162
Using Liquid Data Controls to Develop Workshop Applications.....	163
WebLogic Workshop and Liquid Data.....	164
Liquid Data Control JCX File.....	165
Creating Liquid Data Controls.....	168
Modifying Existing Liquid Data Controls.....	177
Using NetUI to Display Liquid Data Results.....	180
Security Considerations With Liquid Data Controls.....	189
Moving Your Liquid Data Control Applications to Production.....	192

Working with Java Controls

WebLogic Workshop provides Java controls that make it easy for you to encapsulate business logic and to access enterprise resources such as databases, legacy applications, and web services. There are three different types of Java Controls: built-in Java controls, portal controls, and custom Java controls.

Built-in controls provide easy access to enterprise resources. For example, the Database control makes it easy to connect to a database and perform operations on the data using simple SQL statements, whereas the EJB control enables you to easily access an EJB. Built-in controls provide simple properties and methods for customizing their behavior, and in many cases you can add methods and callbacks to further customize the control.

A portal control is a kind of built-in Java control specific to the portal environment. If you are building a portal, you can use portal controls to expose tracking and personalization functions in multi-page portlets.

You can also build your own custom Java control from scratch. Custom Java controls are especially powerful when used to encapsulate business logic in reusable components. It can act as the nerve center of a piece of functionality, implementing the desired overall behavior and delegating subtasks to built-in Java controls (and/or other custom Java controls). This use of a custom Java control ensures modularity and encapsulation. Web services, JSP pages, or other custom Java controls can simply use the custom Java control to obtain the desired functionality, and changes that may become necessary can be implemented in one software component instead of many.

Topics Included in This Section

Getting Started with Java Controls

Provides an overview of Java controls.

Invoking a Control Method

Describes how to invoke a control method from another software component.

Handling Control Callbacks

Describes how to implement a callback handler.

Handling Control Method Exceptions

Discusses how to handle exceptions that may be thrown by your control.

Using Built-In Controls

Describes the Java controls that WebLogic Workshop provides for you to connect to databases, set timers, send messages, and perform other common tasks.

Building Custom Controls

Explains how to create Java controls to use in your projects and applications.

Working with Java Controls

Adding Portal Controls to Java Page Flows

Introduces Portal controls and provides links to other portal topics to help you work with them.

Related Topics

Getting Started: Java Controls

This tutorial provides a quick introduction to developing Java Controls.

Getting Started with Java Controls

When you're building WebLogic platform applications, Java controls provide a convenient way to incorporate access to resources and encapsulate business logic. If you've used WebLogic Workshop, you may be familiar with built-in Java controls such as the Database control, EJB control, Web Service control, and so on. These are included with the IDE, but you can also create your own custom Java control. You can use controls from within the many kinds of components that make up WebLogic platform applications. A good practice is to use the custom Java control to implement your business logic and call built-in controls when the implementation of the business logic requires this.

This topic provides an overview of Java controls in platform applications. It includes the following sections:

What Are Java Controls?

Java Controls in the IDE

Building Custom Java Controls

Ways to Think About Custom Controls

Working with Java Control Sources

What Are Java Controls?

Java controls are reusable components you can use anywhere within a platform application. You can use built-in controls provided with WebLogic Workshop, or you can create your own.

Note: In previous versions, controls were represented as CTRL files. While applications built with these controls are still supported, they are deprecated for future versions. You should build new applications with Java controls based on the new model.

Uses for Java controls. The framework that supports Java controls is flexible, supporting a wide variety of uses for controls. Java controls can:

- Contain business logic you want to keep separate from other application code, or which may be reused.
- Provide access to resources such as databases or other resources.
- Collect logic that coordinates multiple actions, such as those that involve multiple database queries, calls to Enterprise JavaBeans (with the EJB control), and so on. A control participates in the implicit transaction of a conversational container, such as a web service that is conversational.

Built-in and custom Java controls. WebLogic Workshop provides several built-in controls, and you can build your own.

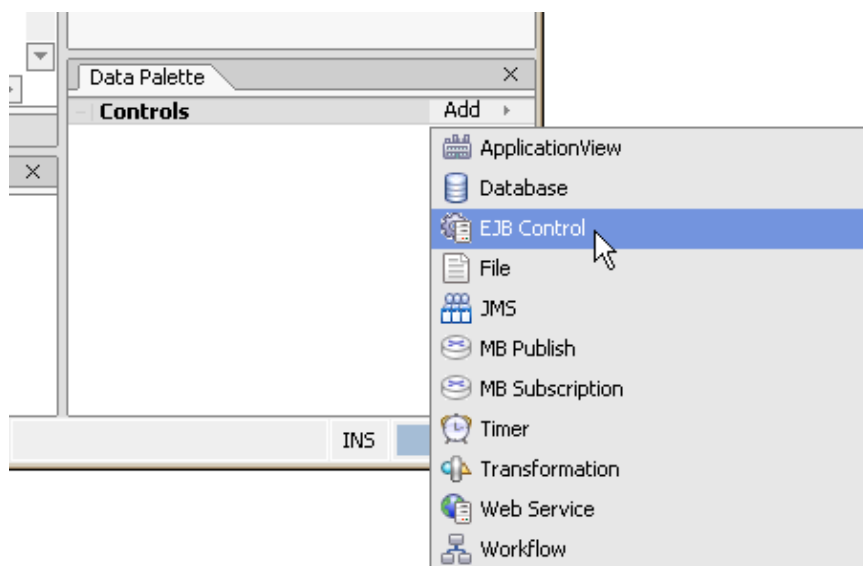
- WebLogic Workshop provides several *built-in controls*, mostly designed to provide access to resources. For example, you can use the EJB control for access to Enterprise JavaBeans, the JMS control for access to the Java Message Service, and so on. For more information about the built-in controls, see Using Built-In Java Controls.
- You can build your own controls that are based on the same framework on which built-in controls are based. You design a custom control from the ground up, designing its interface and business logic,

Working with Java Controls

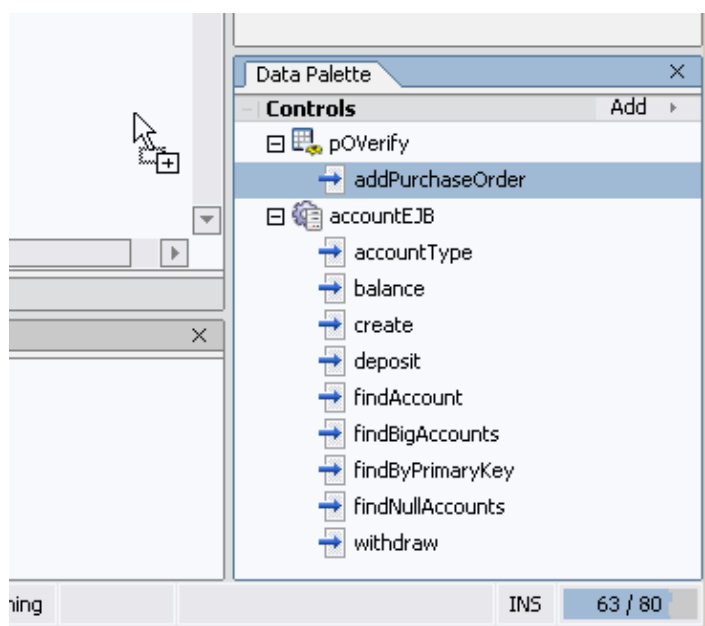
adding other controls as needed. You can design a custom control for use in one project, or you can design a custom control for easy reuse in multiple projects. For more information about the custom controls, see *Building Custom Java Controls*. The tutorial, *Tutorial: Java Control*, provides a hands-on introduction to building custom controls. Much of this topic also includes information on building your own controls.

Java Controls in the IDE

Built-in Java controls, and custom Java controls that have been set up for use in multiple projects, are listed in the WebLogic Workshop Data Palette. By default, the Data Palette is displayed in the lower-right corner of the IDE. You can add new controls to a design by clicking the Data Palette's Add menu, as shown here:



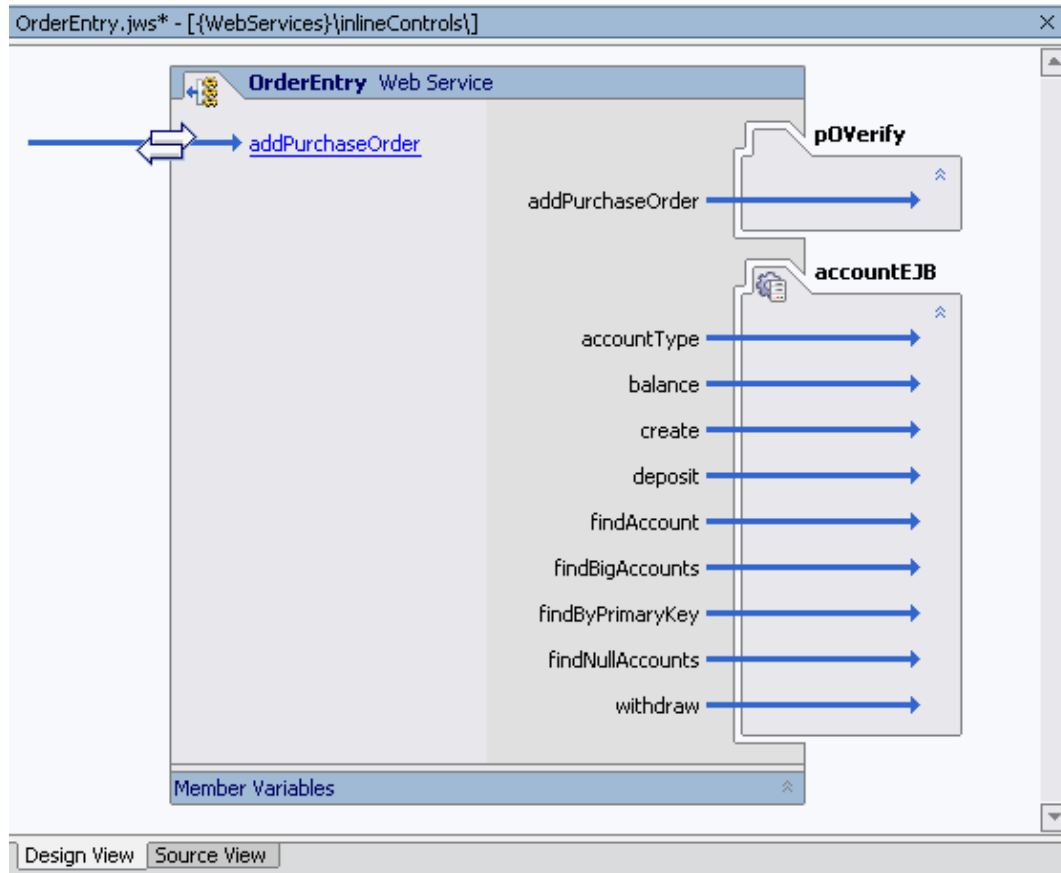
When a control is in your design, its methods and callbacks are also listed in the palette. You can also drag methods and callbacks onto your design to create "pass-through" methods. A pass-through is a shortcut way to call a control's method from your current design.



Working with Java Controls

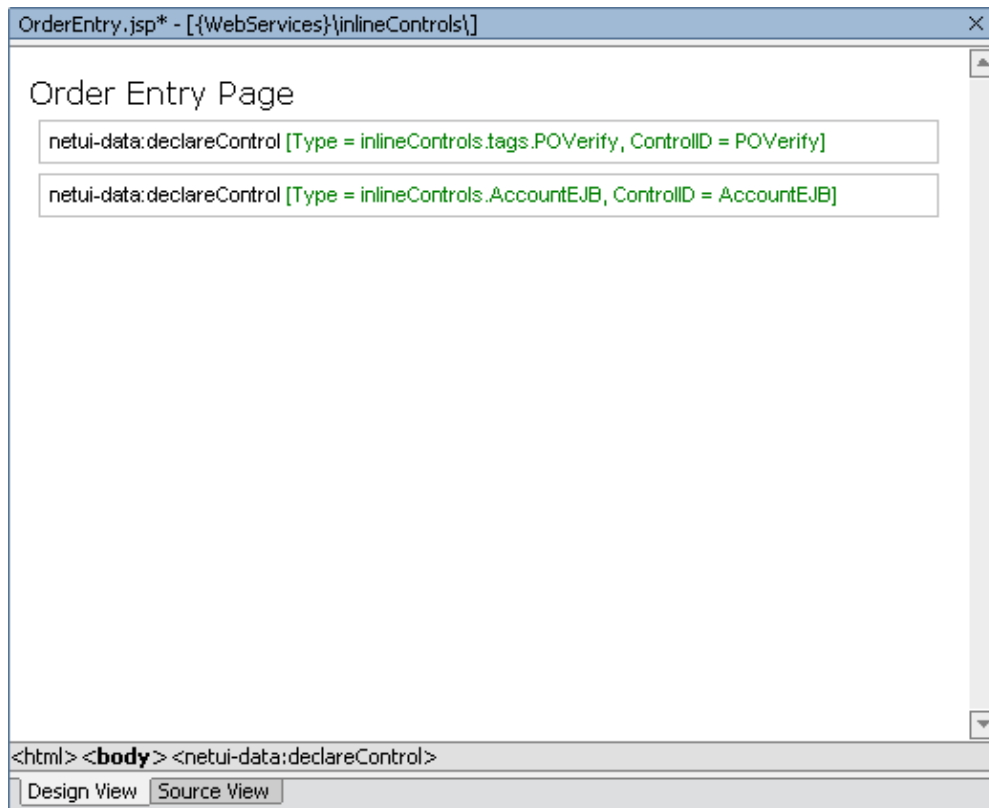
Java controls appear in Design View, but will look slightly different depending on the file type you are designing with. The following illustrations show a custom (POVerify) and built-in (EJB control) Java control in various types of component source files.

Controls in a web service (JWS) design:

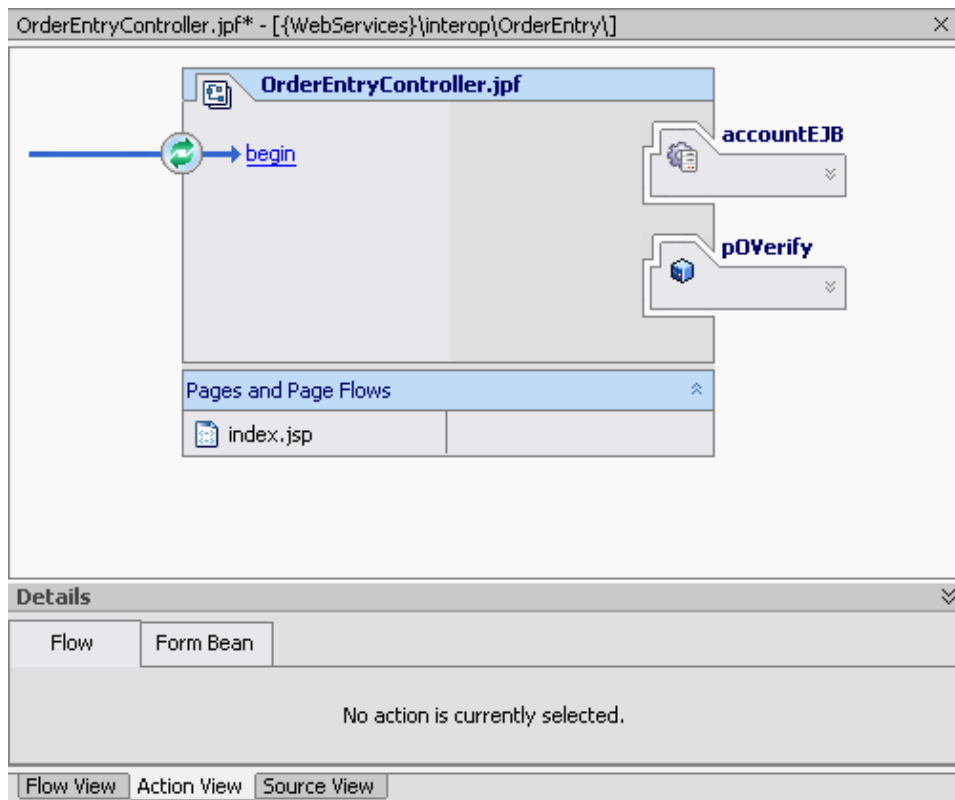


Controls in a JavaServer Pages (JSP) design:

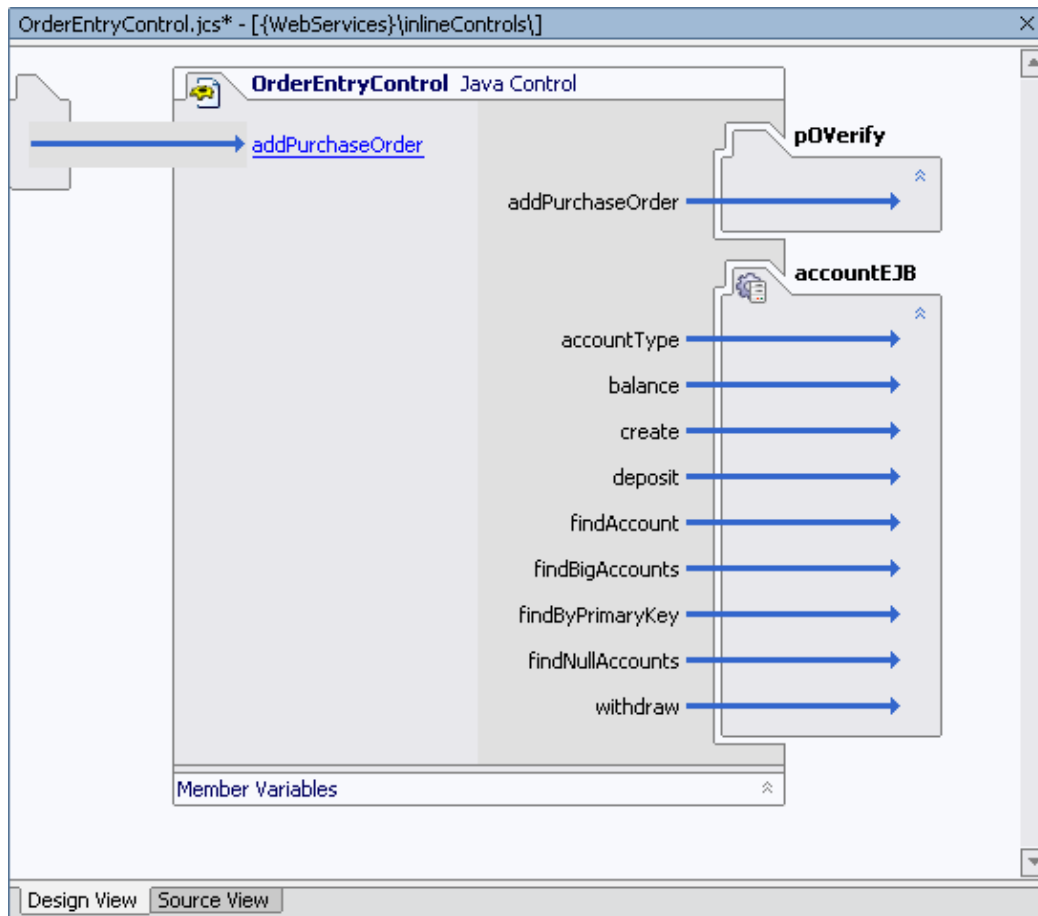
Working with Java Controls



Controls in a Java page flow (JPF) action view:



Controls in a Java control (JCS) design:



Building Custom Java Controls

You can build your own Java controls. Java controls you build are like those provided with WebLogic Workshop in that they provide a way to encapsulate business logic or access a resource, and yet expose a simple interface. However, unlike controls that are built into WebLogic Workshop, where you have access to an interface that extends the source, you can use your own control source files nested within the project that accesses the control. This makes Java controls particularly useful as containers for code that should reside in the same project but which is best kept separate.

Ways to Think About Custom Controls

Local controls and control projects. You can use controls locally as source, or group them into control projects.

- A control is said to be *local* when its source files reside in the same project as the code that uses the control. This is the simplest way to use Java controls. You can create a JCS file, add methods and callbacks to it, then call the methods from code in the same project. This is most likely the way you will use Java controls you create. For step-by-step instructions on creating and using local Java controls, see *How Do I: Create and Use a Custom Java Control Within an Existing Project?* You might also be interested in *Tutorial: Java Control*. For examples of local controls, see the *SamplesApp* application installed with WebLogic Workshop; there, look in the *WebServices* project, in its *localControls* subfolder.

Working with Java Controls

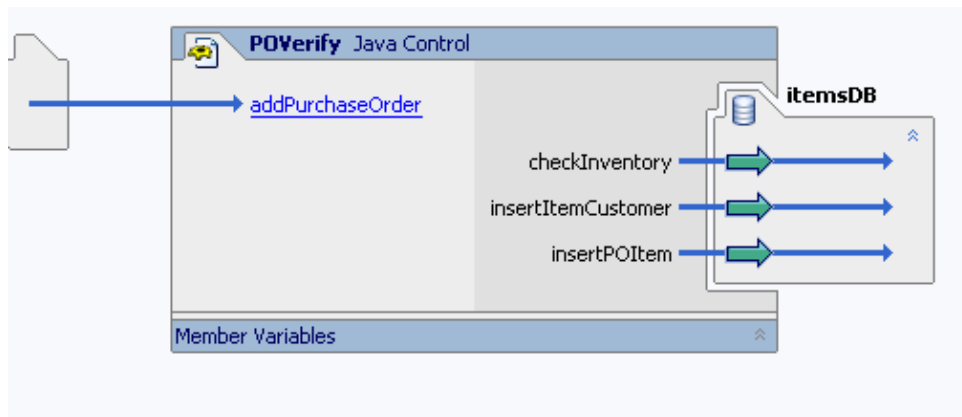
- *Control projects* provide a way to group related controls, and to package them for distribution among multiple projects. You create a control project just as you would other kinds of projects, then add files for your controls. The result of a control project is a JAR file you can distribute for use in any WebLogic Workshop application. By default, building a control project will automatically copy the resulting JAR to the Libraries folder of the application containing the project. Tutorial: Java Control includes information about packaging controls into a control project. The ControlProject project in the SamplesApp application installed with WebLogic Workshop is an example of a control project. For step-by-step instructions on beginning a control project, see How Do I: Create and Use a Java Control Within a Control Project?

Regular and customizable controls. Some controls provide a static interface, some are customizable.

- Most of the custom controls you build will probably be *regular* controls. They don't provide a customizable interface. That is, their interface is already defined when an application developer adds them to an application. In this way, a regular control is like a library of reusable code. The built-in Timer control is an example of a regular control, as are the controls whose source code is included in the SamplesApp application installed with WebLogic Workshop.
- Most of the built-in controls provided with WebLogic Workshop are *customizable* controls. That is, when you add a new one to a project, WebLogic Workshop generates a JCX file that extends the control. In some cases, such as with the Database control or JMS control, you can customize the control by adding or editing methods defined in the JCX file. Others are customized for you, as with the EJB control, which is customized based on the EJB the control will be accessing. Building customizable custom controls is an advanced subject beyond the scope of this documentation. For an example of a customizable custom control, see the JcxCreate sample in the ControlDevKit sample application available with WebLogic Workshop.

Working with Java Control Sources

Design view for custom controls. You begin building a Java control much as you would start building other WebLogic Workshop components. After you create a Java control source (JCS) file, Design View provides a space in which you can create a visual representation of your control's interface as well as the controls it may itself be using. The Java control design space is very much like the web service design space. The left side displays operations that will be visible to the control's clients, while the right side displays nested controls.



Note: You have easy access to a control's source file when the source is in the same project as the design you're editing. When you are building a web service, page flow, or Java control that includes a control whose source is in the same project, you can double-click the control at the right side of the design to open its JCS.

Working with Java Controls

When you add a new Java control source file to a project, WebLogic Workshop also adds a JAVA file that contains the control's public interface. By default, as you work in the JCS file, adding methods, callbacks, and implementation code, WebLogic Workshop keeps the interface in sync. For example, adding an operation to the JCS will also add a corresponding method to the JAVA file. Note that the JAVA file will be kept in sync only with respect to those methods with an `@common:operation` annotation. This means that if you add a method to the JCS, then remove its `@common:operation` annotation, WebLogic Workshop will remove the method from the JAVA file.

For step-by-step information on beginning a Custom Java control, see [How Do I: Begin a New Custom Java Control](#). You might also be interested in [How Do I: Create and Use a Custom Java Control Within an Existing Project?](#)

Properties for Java controls. Controls you create can expose properties. For example, the Database control provides properties that specify its database connection, log category name, and so on.

You define properties by creating an annotation XML file that describes them. You then associate the file with the control source code through the JCS file's control-tags property. When a developer is using the control, setting its properties, the settings are saved as annotations in the developer's code.

For step-by-step information on defining control properties, see [How Do I: Define Properties for a Java Control?](#)

IDE characteristics for a control. You can define certain IDE characteristics for your Java control. These include the icon that represents it in palettes and menus (and whether it is displayed in the palette at all), its description in the Property Editor, and so on. You'll find settings for these characteristics in the JCS file's `jc-jar` property. For more information, see [How Do I: Specify IDE Characteristics for a Java Control?](#)

Testing Java controls. While you can't "run" a JCS file in Test View, as you can with the similar JWS file, WebLogic Workshop does provide a shortcut for easy testing. You can generate a JWS file through which you can exercise the control's code as you're writing it. For more information, see [How Do I: Generate a JWS File to Test a Java Control?](#)

Keep in mind that testing isn't complete until you've tried out the control in all the scenarios you expect it to support. Whether the control's container is likely to be a JWS file, a JSP file, or a JPF file, it's a good idea to build a test application that uses your control for the purpose for which you've designed it.

Related Topics

[Working with Java Controls](#)

[Tutorial: Java Control](#)

Invoking a Control Method

Once you've added a Java control to your application, you can invoke the methods of the Java control from Source view using the standard Java dot notation. For example, assume that you have added the CustomerDb Java control and declared a variable for the control as custDb, and that the control defines a method as follows:

```
String [] getAllCustomerNames()
```

You can invoke this method from your code as follows:

```
String [] custNames;
```

```
custNames = custDb.getAllCustomerNames();
```

Overriding Control Property Settings

Note that when you declare a control (a JCS or JCX file) within a client you can override the control's default properties.

Suppose you have a database control with the connection property defined so that the data-source-jndi-name attribute points at the data source dataSource.

DatabaseControl.jcx

```
/**
 * @jc:connection data-source-jndi-name="dataSource"
 */
public interface DBControl extends com.bea.control.ControlExtension, DatabaseControl
```

The database control is declared with its default properties in the following way.

MyWebService.jws

```
/**
 * @common:control
 */
private DatabaseControl dbControl;
```

To override the default connection property on the database control, use the following declaration.

```
/**
 * @common:control
 * @jc:connection data-source-jndi-name="myOtherDataSource"
 */
private DatabaseControl dbControl;
```

In the above declaration, the database control will use myOtherDataSource instead of its default dataSource. This override value will apply to all method calls from within MyWebService.jws.

There are two important caveats when overriding control properties in this way.

Working with Java Controls

- (1) The property must be at the class level, not the method level. Only those properties that are scoped to the entire control class may be overridden in this way.
- (2) The override occurs at the property level, not at the attribute level. It is not possible to override only a single attribute on a property. You must override all of the property's attribute values. For example, suppose that the connection property had three attributes instead of just one.

DatabaseControl.jcx

```
/**
 * @jc:connection data-source-jndi-name="dataSource" attribute2="value2" attribute3="value3"
 */
public interface DBControl extends com.bea.control.ControlExtension, DatabaseControl
```

In this case, when you override the connection property, you override all of that property's attributes, not merely the data-source-jndi-name attribute. In short, the following override sets the data-source-jndi-name attribute to myOtherDataSource and attribute2 and attribute3 to null.

MyWebService.jws

```
/**
 * @common:control
 * @jc:connection data-source-jndi-name="myOtherDataSource"
 */
private DatabaseControl dbControl;
```

For this reason, you should specify all of the attributes when you override a control property.

```
/**
 * @common:control
 * @jc:connection data-source-jndi-name="myOtherDataSource" attribute2="otherVal2" attribute3="otherVal3"
 */
private DatabaseControl dbControl;
```

Related Topics

None

Handling Control Callbacks

Custom Java controls and several built-in control types allow the specification of callbacks. Callbacks provide a way for a control or a web service to asynchronously notify a client that an event has occurred.

A callback is a method signature that is defined by a resource like a control where the method implementation must be provided by the client. The client enables reception of a callback by implementing a callback handler.

A simple way to think of a callback is as an event that your code can respond to. When a Timer control fires, it is an event, and WebLogic Workshop calls the Timer control's `onTimeout` callback handler. The callback handler is a method like any other, except that your code does not control when it is called. The callback handler may be called by WebLogic Workshop or by code running in a client. In the case of a callback defined by a control, the application that's using the control is the client. So your application implements callback handlers for callbacks defined by controls, and you can write code in the callback handler to run when the event occurs.

For more information on callbacks and asynchrony, see *Designing Asynchronous Interfaces*.

Callback Definition

A callback definition in a Java control may look like the following:

```
void onReportStatus(String status);
```

This declaration appears in the source code for the service or control that defines the callback. There's no code associated with the callback definition — only the method signature, including the return type and any parameters.

It is common for callbacks to have names that begin with *on* because a callback represents an event and the callback handler in the client will be called on occurrence of the event. The name of this callback handler suggests that the handler will be invoked when the report status is provided by a client.

Callback Handler Definition

Your application is responsible for implementing the handler for a callback defined by a control. In Design view, a callback defined by a control is highlighted as a hyperlink; when you click on this link, the callback handler is created for you in your code. You can then add code to it to respond to the callback.

The following shows an example of a callback handler as it might appear in your application:

```
void exampleControl_onReportStatus(String status)
{
    // add your code here to take appropriate action given the status of the report
}
```

In WebLogic Workshop, callback handler names are determined by the name of the control instance and the name of the callback. In the example above, the control instance from which we wish to receive the callback is `exampleControl`. The full name of the callback handler, `exampleControl_onReportStatus`, is the control instance name followed by an underscore and the name of the callback.

Related Topics

Designing Asynchronous Interfaces

Handling Control Method Exceptions

The designer of a Java control may choose whether or not to explicitly declare that exceptions are thrown by the control's methods. If a control method is declared to throw exceptions, you must enclose your invocations of that method in a try–catch block.

Even if the designer of the control chooses not to declare exceptions, the support code that implements the control can still throw exceptions. The type of exception thrown is `com.bea.control.ControlException`.

You should strongly consider handling all control exceptions that may be thrown by the controls you use in a web service. If you do not handle the exception, the web service method will fail and the exception will be passed on to the client of your web service. In most cases, the exception is useless to the client and the client does not have the necessary information to diagnose or remedy the problem.

To learn more about exceptions and try–catch blocks, see [Introduction to Java](#).

Related Topics

None

Control Factories: Managing Collections of Controls

This topic describes *control factories*, which are collections of built-in or custom Java control instances.

What Is a Control Factory?

A control factory allows a single application to manage multiple instances of the same control.

For example, imagine a credit approval application that accepts batches of approval requests (one per applicant) and uses an external web service, via a Web Service control, to evaluate the requests. The application could use a control factory to create multiple instances of the Service control and dispatch requests to the Service control instances in parallel. If the control uses callbacks, a single parameterized callback handler in the calling application handles the callbacks received from all of the control instances.

You can only use control factories within a Java Web Service (JWS) or Java Process Definition (JPD) file. For more information on building Java Web Services with WebLogic Workshop, see *Building Web Services*. For more information on using JPD files, see *Guide to Building Business Processes*.

Automatically Generated Factory Classes

For any control interface called `MyControl`, WebLogic Server generates a control factory interface called `MyControlFactory` that has the following very simple shape:

```
interface MyControlFactory
{
    MyControl create();
}
```

The implicit factory class is located in the same package as the control class; that is, if the full classname of the control interface is `com.myco.mypackage.MyControl`, then the full classname of the factory is `com.myco.mypackage.MyControlFactory`. An automatic factory class is not generated if there is a name conflict (i.e., if there is already an explicit user class called `MyControlFactory`). Therefore, if you want WebLogic Workshop to automatically generate factory classes for a built-in or custom control, make sure that the name of that control does not end with the word "Factory".

A control factory instance can be included in a file just as a control instance can, with the same Javadoc annotation preceding the factory declaration that would precede a single control declaration.

For example, an ordinary Web Service control is declared as follows:

```
/**
 * @common:control
 */
MyServiceControl oneService;
```

Meanwhile, a Web Service control factory is declared as follows:

```
/**
 * @common:control
 */
MyServiceControlFactory manyServices;
```

Working with Java Controls

Note again that the set of annotations allowed and required on a factory are exactly the same as the set of annotations on the corresponding control. The factory behaves as if those annotations were on every instance created by the factory.

Once an application includes a control factory declaration, a new instance of a single control can be created as follows:

```
// creates one control
MyServiceControl c = manyServices.create();

// then you can just use the control, store it, or whatever.
c.someMethod();

// For example, let's associate a name with the service...
serviceMap.put("First Service", c);
```

Factory classes are automatically generated on-demand, as follows. When resolving a class named FooFactory:

1. First the class is resolved normally. For example, if there is a CLASS file or JAVA file or JCX file that contains a definition for FooFactory, then the explicitly defined class is used.
2. If there is no explicit class FooFactory, then, since the classname ends in "Factory", we remove the suffix and look for an explicit class called Foo (in the same package).
3. If Foo is found but does not implement the Control interface (i.e., is not annotated with @common:control), it's considered an error (as if Foo were never found).
4. However, if Foo is found and implements the Control interface, then the interface FooFactory is automatically created; the interface contains only the single create() method that returns the Foo class.

All instances of the control are destroyed when the application instance that created them is destroyed.

Parameterized Callback Handlers

Since there may be multiple controls that were created with a single control factory, and they all have the same instance name, a mechanism is provided to enable you to tell which instance of the control is sending a callback.

For example, for the oneService example above, an event handler still has the following form:

```
void oneService_onSomeCallback(String arg)
{
    System.out.println("arg is " + arg);
}
```

For callback handlers that are receiving callbacks from factory-created control instances, the callback handler must take an extra first parameter that is in addition to the ordinary parameters of the callback. The first parameter is typed as the control interface, and the control instance is passed to the event handler.

The manyServices factory callback handler looks like this:

```
void manyServices_onSomeCallback(MyServiceControl c, String arg)
{
```

Working with Java Controls

```
// let's retrieve the remembered name associated with the control
String serviceName = (String)serviceMap.get(c);

// and print it out
System.out.println("Event received from " + serviceName);
}
```

Related Topics

[Using WebLogic Built-In Controls](#)

[Building Custom Java Controls](#)

Using Built-In Java Controls

WebLogic Workshop's built-in Java controls make it easy to access enterprise resources like databases and Enterprise JavaBeans from within your application. The control handles the work of connecting to the enterprise resource for you, so that you can focus on the business logic to make your application work.

You can also create your own custom Java controls to encapsulate business logic in a reusable component. For information on creating custom Java controls, see [Building Custom Java Controls](#).

For additional controls available for building WebLogic Integration applications, be sure to see [Using Integration Controls](#).

Topics Included in This Section

Working with Built-In Controls

Explains basic concepts for working with built-in controls.

Timer Control

Describes how to use the Timer control to run code at a given interval.

Database Control

Explains how to use the Database control to access a relational database.

Web Service Control

Describes how to use the Web Service control to call a web service from within your application.

EJB Control

Explains how to use the EJB control to access an existing Enterprise JavaBean (EJB).

JMS Control

Describes how to use the JMS control to send and receive messages via a Java Message Service (JMS) queue or topic.

Control Factories: Managing Collections of Controls

Explains how to use control factories with built-in controls.

Related Topics

JCX Files: Implementing Controls

Working with Built-In Controls

This topic describes how to use a built-in Java control. It explains how to:

- Add a new built-in control
- Copy an existing control
- Add an existing control

The built-in control is typically used by a custom Java control to delegate subtasks, but it can also be used directly by a web service in much the same way. Built-in controls, as well as custom Java control, can furthermore be invoked from a web page, although the procedure for invoking these controls from a web page environment is somewhat different. For more information, see *Developing Web Applications and Page Flows and JSPs*.

Note: In previous releases, JCX files were known as CTRL files. CTRL files are still supported.

You can add controls to the following file types:

- Java Control (JCS file)
- Java Page Flow (JPF file)
- Java Server Page (JSP file)
- Java Web Service file (JWS file)
- Process file (JPD file)

To Add a New Built-In Control

1. Display the Data Palette (**View**—>**Windows**—>**Data Palette**) and click **Add**. Select the control from the drop-down list. For some file types, you can also click the **Insert** menu, and select the control to insert.
2. In the **Insert Control** dialog, provide a variable name for the control. This is the name you will use to refer to the control from your code. Also, provide values for the other fields. The exact fields depend on the type of build-in control you are adding.

For many controls, when you add a built-in control to your application, you are actually creating a new control file. In the **Insert Control** dialog, you specify a name for the new control file that WebLogic Workshop creates. By default WebLogic Workshop adds this control file with a JCX extension to the same folder as the file that is currently open in Design View.

Note: In previous releases, JCX files were known as CTRL files. CTRL files are still supported.

Some controls, like the Timer control, do not have an associated control file. You can't reuse a Timer control—you create a new instance of the Timer control each time you need it.

Control References in Source Code

When you add a control to your application, WebLogic Workshop modifies your file's source code to include an annotation and variable declaration for the control. The annotation ensures that the control is recognized by

WebLogic Workshop, and the variable declaration gives you a way to work with the control from your code. For example, if you create a new Database control named CustomerDb in the customers folder in your project, and specify a variable name of custDb, the following code will be added to your file:

```
/**
 * @common:control
 */
private customers.CustomerDb cus
```

Using an Existing Control

If you have access to a built-in Java control that you either implemented yourself or that was implemented by another developer, you can add this control to a web service or custom Java control. You have access to a control if you have access to its JCX file in your project. If the built-in control is not in your project, you can copy it to your project.

Copying the Control JCX File

If the JCX file for the control you wish to use is not in your project, you can copy it to your project. The destination to which you copy the control depends on the expected usage. If the control will be used only by a single web service or custom control in your project, you may choose to copy the JCX file to the same folder as your web service or custom control. If you expect that you will reuse the built-in control in multiple projects, consider creating a Java control project. For more information, see [How Do I: Create and Use a Java Control Within a Control Project?](#)

Be aware that if you copy a JCX file to a new location, you are creating a new control. If you do not change the definition of the copy, you now have two identical but separate controls. If the original copy of the control is subsequently changed, the second copy will remain unchanged. This means that any bug fixes or feature enhancements made to the original control will *not* be reflected in the copy.

If you decide to copy the control's JCX file, it *must* be in your WebLogic Workshop project. Java imposes rules on references between packages that prevent you from using a control if it is not located in the correct place. To learn about WebLogic Workshop project organization, see [Applications and Projects](#).

Modifying the Control's Package

If you drag a JCX file from one location to another within your WebLogic Workshop project, the package statement is automatically modified for you and you don't need to read the remainder of this section. However, if you decide to copy a control's JCX file in some other way than dragging it in the Application pane, you must change the package statement in the JCX file.

In Java the location of a source file in the directory hierarchy determines (and must agree with) the Java package to which the classes in the file belong. If a Java file is located in the <project>\controls\financial directory, the Java package statement in the file must be the following:

```
package controls.financial;
```

After copying a control's JCX file to a new location, you must change the package statement in the file to reflect the new location.

For more information on Java packages, see [Introduction to Java](#).

To Add an Existing Control

If you have an existing JCX file in your project, you can add a reference to that control by dragging the JCX file from the Application pane to the Design View of the web service or custom control from which you would like to use the built-in control. You may also use the Insert menu.

You can make multiple references to the same existing built-in control, for instance by referencing it in several custom controls in the same project. Note that you are making a reference and are not creating a copy. When you change the built-in control, the control is modified for every file in which it is referenced.

Related Topics

[Timer Control](#)

[Database Control](#)

[Web Service Control](#)

[EJB Control](#)

[JMS Control](#)

Timer Control

A Timer control notifies your application when a specified period of time has elapsed or when a specified absolute time has been reached.

All Timer controls are instances of the `com.bea.control.TimerControl` base class. Unlike most controls, a Timer control is declared directly in a JWS file; there is no subclass created for a Timer control.

To learn about WebLogic Workshop controls, see [Using WebLogic Built-In Controls](#).

Topics Included in this Section

Overview: Timer Controls

Provides a brief introduction to creating and configuring Timer controls.

Creating a New Timer Control

Explains how to create a new Timer control and provides a sample of the resulting declaration.

Using a Timer Control

Explains how to configure a Timer control you've already created.

Specifying Time on a Timer Control

Explains how to specify relative and absolute time when setting the attributes of a Timer control.

Timer Control Samples

`SimpleTimer.jws` Sample

`AdvancedTimer.jws` Sample

`HelloWorldAsync.jws` Sample

Related Topics

TimerControl Interface

[Using WebLogic Built-In Controls](#)

Overview: Timer Controls

Some transactions and events require a certain amount of time to complete. Others can run indefinitely if not aborted, and eat up resources. Still others must occur at a specific time. The Timer control provides the developer with a way to respond from code when a specified interval of time has elapsed or when a specified absolute time has been reached.

Configuring Timer Controls

You configure a separate Timer control for each event or operation you want to regulate. You can specify settings for a Timer control in several different ways. One way is to set the Timer control's properties in Design view. Another way is to call the methods of the TimerControl interface. The Timer control notifies the application of the parameters using the onTimeout callback.

To learn more about setting these values and using callbacks, see [Using a Timer Control](#).

Specifying Time Values

You can specify time values for the Timer control in two ways: you can set a relative time limit or an absolute time limit. Relative time values are relative to the present. For example, you can schedule an event to occur three days, three hours, and three minutes from now. Absolute time refers to a specific moment. For example, you can schedule an event for 6:00 PM on October 18, 2002.

To learn more about specifying time, see [Specifying Time on a Timer Control](#).

Related Topics

[Timer Control](#)

[Creating a New Timer Control](#)

Creating a New Timer Control

A Timer control notifies your application when a specified period of time has elapsed or when a specified time has been reached. Each Timer control is customized with a particular behavior. Timer controls are declared locally in an application's JWS file, rather than in a separate file.

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

Creating a New Timer Control

You can create a new Timer control in any of the following types of files:

- Java Control (JCS file)
- Java Page Flow (JPF file)
- Java Server Page (JSP file)
- Java Web Service file (JWS file)
- Process file (JPD file)

To create a new Timer control, follow these steps:

1. Open the file to which you want to add the control in the WebLogic Workshop design environment.
2. Display the Data Palette (**View**→**Windows**→**Data Palette**).
3. On the Data Palette, click the **Add** drop-down, and choose **Timer**. The **Insert Control – Insert Timer** dialog opens.
4. In the **Variable name for this control** field, type the name of the new control. The name you enter must be a valid Java identifier.
5. In the **timeout** field, specify the amount of time you want to elapse before the timer fires the first time.
6. If applicable, in the **repeats–every** field, specify the interval between firings after the Timer fires the first time.
7. 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*.
8. Click **Create**.

When you create a Timer control, WebLogic Workshop imports the Timer control class, `com.bea.control.TimerControl`, into your source file.

Example: Timer Control Declarations

When you create a new Timer control, its declaration appears in the source file. The following code snippet is an example of what the declaration looks like:

```
import com.bea.control.TimerControl;
...
/**
 * @common:control
 * @jc:timer timeout="5 seconds" repeats-every="5 seconds"
 */
TimerControl delayTimer;
```

Working with Java Controls

The actual attributes that are present on the `@jc:timer` annotation depend on the values you entered in the Insert Control – Insert Timer dialog.

The `@common:control` annotation informs WebLogic Workshop that the associated declaration is a control. Without this annotation, the control is not properly connected to supporting code and will not function.

For more information on the `@common:control` annotation, see `@common:control` annotation.

The `@jc:timer` annotation controls the behavior of the Timer control. All of the attributes of the `@jc:timer` annotation are optional and have default values.

For more information on the `@jc:timer` annotation, see `@jc:timer` annotation.

The Timer control, named `myTimer` in the example above, is declared as an instance of `TimerControl`. Timer controls are unusual in that you can instantiate the control class directly. Most other controls requires that you declare a subclass of the base class and then declare the new class.

Related Topics

[Using WebLogic Built-In Controls](#)

[Timer Control](#)

[Using a Timer Control](#)

[Specifying Time on a Timer Control](#)

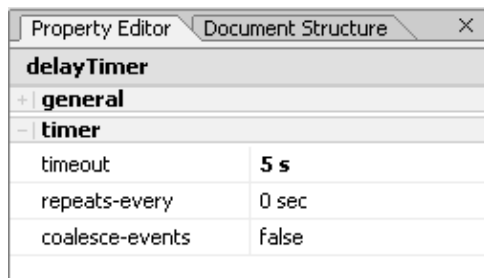
Using a Timer Control

A Timer control notifies your application when a specified period of time has elapsed or when a specified absolute time has been reached. For example, you can use a Timer control to run a process at certain intervals throughout the day, or to cancel an operation that is taking too long.

The following sections describe how to configure the Timer control.

Setting Default Timer Control Behavior

You can specify the behavior of a Timer control in Design View by setting the control's timeout and repeats-every properties in the Property Editor window. For example, if you select a Timer control instance named `delayTimer` in Design View, the Property Editor window displays the following properties:



These properties correspond to attributes of the `@jc:timer` annotation, which identifies the Timer control in your code. The `@jc:timer` annotation has the following attributes:

- `timeout`, which specifies the time until the Timer control fires the first time, once started
- `repeats-every`, which specifies how often the Timer control should fire after the first time
- `coalesce-events`, which specifies how the Timer control should behave if delivery of its events is delayed

To learn more about specifying default Timer control behavior with attributes of the `@jc:timer` annotation, see [@jc:timer Annotation](#).

You can set these attributes to specify relative time. Relative time is an interval of time in relation to the present, such as three hours from now. You can also specify that the Timer control fire at an absolute time, such as 3:00 AM, by calling the Timer control's `setTimeoutAt` method. To learn more about specifying absolute and relative time, see [Specifying Time on a Timer Control](#).

Using Methods of the TimerControl Interface

Once you have declared and configured a Timer control, you can invoke its methods from within your application to start and stop the timer and to change its configuration. For complete information on each method, see [TimerControl Interface](#).

The following list contains the methods of the `TimerControl` interface that you can use to start and stop the timer:

- `start`: starts timer operation. The Timer control will fire after the period specified by the `timeout`

attribute has passed.

- restart: resets the Timer control such that the next firing will occur after the period specified by the timeout attribute has passed.
- stop: stops the Timer control from firing again until start or restart has been called.

The TimerControl interface also defines the following methods that you can use with a Timer control:

- setTimeoutAt
- getTimeoutAt
- setTimeout
- getTimeout
- setRepeatsEvery
- getRepeatsEvery
- setCoalesceEvents
- getCoalesceEvents

Handling Timer Control Callbacks

The Timer control defines one callback: onTimeout. You can add code to the callback handler to run when the timer fires. The callback handler for the onTimeout event is named *timerName_onTimeout*, where *timerName* is the name of the Timer control instance.

The callback handler takes a single parameter, which is the time at which the callback was scheduled. Note that this is not the same as the time at which the callback handler executes. A delay may occur between Timer control expiration and callback handler invocation, depending on the system load.

To create the callback handler for a Timer control's onTimeout callback, click the onTimeout link associated with the Timer control in Design View. If a callback handler does not exist for the selected callback, WebLogic Workshop creates one, switches to Source View, and places the cursor in the callback handler for the selected callback.

Related Topics

[Using WebLogic Built-In Controls](#)

[Timer Control](#)

[Creating a New Timer Control](#)

[Specifying Time on a Timer Control](#)

[TimerControl Interface](#)

Specifying Time on a Timer Control

This topic describes how to specify relative and absolute time values for a Timer control. A relative time value specifies an interval of time that is relative to the present, such as next Thursday or twenty minutes from now. An absolute time value specifies time according to the clock. For example, 5:00 PM is an absolute time.

Specifying Relative Time

You specify relative time when you want an event to fire at a time relative to the present. For example, you might want to wait only 5 minutes for another system to respond. So your application sets a timer with a relative timeout of 5 minutes and starts the timer. If the timer expires (that is, calls its callback) before the system responds, the web service stops waiting and continues with other operations.

To specify that the timer fires after an interval, set the timeout and, optionally, the repeats—every attribute of the Timer control. You can set these attributes in the Properties pane or in source code, or by calling the setTimeout or setRepeatsEvery methods of the TimerControl interface.

When relative time is expressed as a text string, it is formatted as integers followed by case-insensitive time units. These time units can be separated by spaces. For example, the following code sample is a valid duration specification that exercises all the time units, spelled out fully:

```
/**
 * @common:control
 * @jc:timer timeout="99 years 11 months 13 days 23 hours 43 minutes 51 seconds"
 */
Timer almostCentury;
```

This example creates a Timer control whose default initial firing will occur in almost 100 years.

Units may also be truncated. For example, valid truncations of "months" are "month", "mont", "mon", "mo", and "m". If both months and minutes are specified, use long enough abbreviations to be unambiguous.

The string "p" (case insensitive) is allowed at the beginning of a text string. If it is present, then single-letter abbreviations and no spaces *must* be used and parts must appear in the order y m d h m s.

The following Timer control declaration is equivalent to the previous example, but uses the fully truncated form:

```
/*
 * @control
 * @timer timeout="P99Y11Mo13D23H43M51S"
 */
Timer almostCentury;
```

Durations are computed according to Gregorian calendar rules, so if today is the 17th of the month, 3 months from now is also the 17th of the month. If the target month is shorter and doesn't have a corresponding day (for example, no February 31), then the closest day in the same month is used (for example, February 29 on a leap year).

Specifying Absolute Time

Absolute time is useful when you know the exact moment you want operations to begin and end. For example, your application can have your web service send a reminder email to remind you that someone's birthday is coming up.

You can configure a Timer control to fire at an absolute time by calling the `setTimeoutAt` method of the `TimerControl` interface.

The `setTimeoutAt` method configures the timer to fire an event as soon as possible on or after the supplied absolute time. If you supply an absolute time in the past, the timer will fire as soon as possible.

If `setTimeoutAt` is called within a transaction, its effect (any work performed in the callback handler) is rolled back if the transaction is rolled back, and its effect is committed only when the transaction is committed.

If `setTimeoutAt` is called while the timer is already running, it will have no effect until the timer is stopped and restarted.

The `setTimeoutAt` method takes as its argument a `java.util.Date` object. Please see the documentation for the `java.util.Date` class to learn how to manipulate Date objects. Other Java classes that are useful when dealing with Date are `java.util.GregorianCalendar` and `java.text.SimpleDateFormat`.

The `getTimeoutAt` method returns the time at which the timer is next scheduled to fire, if the `repeats` attribute is set to a value greater than zero. If the `repeats` attribute is set to zero, then the `getTimeoutAt` method returns the value set by the `setTimeoutAt` method or the value set in the `timeout` attribute. If you call the `getTimeoutAt` method from within the `onTimeout` callback handler, the first timeout has already fired, so `getTimeoutAt` will return either the time of the next timeout or the time of the first timeout if the timer is not set to repeat.

The following example calls the `setTimeoutAt` method to specify that the first timeout fires at thirty seconds past the current minute, then calls the `setRepeatsEvery` method to specify that the timer subsequently fires every sixty seconds. The `onTimeout` event provides information about the Timer control's firing.

```
/**
 * @common:operation
 * @jws:conversation phase="start"
 */
public void StartTimer()
{
    Calendar cd = new GregorianCalendar();
    cd.set(cd.SECOND, 30);
    tTimer.setTimeoutAt(cd.getTime());
    tTimer.setRepeatsEvery(60);
    tTimer.start();
}
public void tTimer_onTimeout(long time)
{
    callback.FireTimeout("The timer was scheduled to fire at: " + new Date(time)
        + ". The current time is: " + new Date()
        + ". The timer will fire again at: " + tTimer.getTimeoutAt());
}
```

Related Topics

Working with Java Controls

Using WebLogic Built-In Controls

Timer Control

Using a Timer Control

Database Control

A Database control makes it easy to access a relational database from your application. Using the Database control, you can issue SQL commands to the database. The Database control automatically performs the translation from database queries to Java objects, so that you can easily access query results.

A Database control can operate on any database for which an appropriate Java Database Connectivity (JDBC) driver is available and for which a data source is configured in WebLogic Server. When you add a new Database control to your application, you specify a data source for that control. The data source indicates which database the control is bound to.

Topics Included in this Section

Overview: Database Controls

Introduces the basic concepts behind Database controls.

Creating a New Database Control

Explains how to create a new Database control.

Adding a Method to a Database Control

Describes how to write methods on a Database control.

Returning a Single Value from a Database Control Method

Explains how to return a single value from a Database control method and provides examples of the operation.

Returning a Single Row from a Database Control Method

Explains how to return a class of values from a Database control method.

Returning Multiple Rows from a Database Control Method

Describes the options for returning a set of values from a Database control method.

Modifying Data with a Database Control

Describes the options for returning a set of values from a Database control method.

Returning XMLBeans from a Database Control

Describes how to return XMLBean types from a Database Control.

Parameter Substitution in @jc:sql Statements

Lists the criteria for substitution and describes the process of substituting parameters.

Automatically Generating Primary Key Values and Database Controls

Working with Java Controls

Explains how to use Oracle's and SQL Server's auto-generation of primary keys in conjunction with a Database control.

Stored Functions

Explains how to call and create stored functions with a Database control.

Stored Procedures

Explains how to call and create stored procedures with a Database control.

Limiting the Size of Returned Data

Explains how to set the maximum number of records returned by a query.

Declaring Exceptions in Database Control Methods

Provides information to help you decide whether or not to declare exceptions in Database control methods.

Designing a Database Control

Provides some guidelines for designing your database control.

RowSet Control

Describes a special type of Database control used to provide CRUD operations on a database table or view.

RowSet Controls and SQL Join Queries

Describes how to edit a RowSet control to incorporate SQL join queries.

Mapping Database Field Types to Java Types in a Database Control

Provides tables showing how to map field types to Java types.

Configuring a Data Source

Configure a data source with the Data Source Viewer.

Database Control Samples

CustomerDBClient.jws Sample

LuckyNumberDBClient.jws Sample

Related Topics

Using WebLogic Built-In Controls

Database Control

Applications and Projects

JCX File: Implementing Controls

How Do I: Create a RowSet Control to Access a Database?

Overview: Database Controls

A Database control makes it easy to access a relational database from your Java code using SQL commands. The Database control handles the work of connecting to the database, so you don't have to understand JDBC to work with a database.

The methods that you add to a Database control execute SQL commands against the database. You can send any SQL command to the database via the Database control, so that you can retrieve data, perform operations like inserts and updates, and even make structural changes to the database, although a web service interface is generally not the most efficient way of doing so.

All Database controls are subclassed from the DatabaseControl interface. The interface defines methods that Database control instances can call from an application. To learn more about this relationship, see [The DatabaseControl Interface](#).

Database operations may occur within the context of an implicit transaction that wraps each web service method invocation. To learn more about WebLogic Workshop's default transaction semantics, see [Default Transactional Behavior in WebLogic Workshop](#).

Related Topics

None

Creating a New Database Control

A Database control makes it easy to access a relational database via SQL commands. When you create a new Database control, you specify which database it connects to and write methods to access data using SQL commands. This topic describes the mechanics of creating a Database control.

Adding a Database Control

You can add a Web Service control in any of the following types of files:

- Java Control (JCS file)
- Java Page Flow (JPF file)
- Java Server Page (JSP file)
- Java Web Service file (JWS file)
- Process file (JPD file)

To add a new Database control, follow these steps:

1. Open the file to which you want to add the control in the WebLogic Workshop design environment.
2. Display the Data Palette (**View**→**Windows**→**Data Palette**).
3. On the Data Palette, click the **Add** drop-down, and choose **Web Service**. The **Insert Control – Insert Database** dialog opens.
4. In the **Variable name for this control** field, type the name for your database control.
5. Select the **Create a new Database control to use with this service** radio button.
6. In the **New JCX** name field, type the name of the new control file.

Note: In previous releases, JCX files were known as CTRL files. CTRL files are still supported.

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** to select a data source. The **JNDI Entries** dialog appears. Navigate to the data source you want to select and click **Select**.
Note: The cgSampleDataSource data source is available for experimentation. For more information about data sources, see the Choosing a Data Source section, below.
9. Click **Create**.

You can also create a Database control file manually by copying an existing Database control file and modifying the copy.

To learn how to add a method to a Database control, see Adding a Method to a Database Control.

Choosing a Data Source

Before you can perform operations on a database, you must have a connection to the database. The Database control handles all of the details of managing the database connection, but you must supply the name of a *data source* that has been configured with the information necessary to access a database.

A default data source called cgSampleDataSource is configured when WebLogic Workshop is installed. This data source uses the PointBase database.

Working with Java Controls

To learn how to create, configure and register a data source, see [How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle](#).

Once the data source is configured and registered in the JNDI registry, the data source name may be used in the `data-source-jndi-name` attribute of the `@jc:connection` annotation.

For detailed information on the `@jc:connection` annotation, see [@jc:connection Annotation](#).

Related Topics

[Built-In Controls Overview](#)

[Database Control](#)

[How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle](#)

[Adding a Method to a Database Control](#)

[CustomerDBClient.jws Sample](#)

[LuckyNumberDBClient.jws Sample](#)

[DatabaseControl Interface](#)

[@jc:sql Annotation](#)

[@jc:connection Annotation](#)

Adding a Method to a Database Control

The Database control provides access to a relational database. The methods that you add to the Database control execute SQL commands against the database. This topic discusses the mechanics of adding a method to a Database control.

Adding a Method

You can add a method directly to a Database control in Design view by right-clicking on the Database control instance and selecting Add Method, then typing a name for the method.

When you add a method, it initially has no parameters and no associated SQL statement. The following section explains how to edit the parameter list and the associated SQL statement.

Specifying the SQL Statement

A method on a Database control always has an associated SQL statement, which executes against the database when the method is called. The method's @jc:sql annotation describes the method's SQL statement.

You can specify the SQL statement in one of the following ways:

- In design view, right-click the method arrow to the left of the method name and select **Edit SQL**. This brings up the **Edit SQL and Interface** dialog, where you can modify both the method's SQL statement and the method signature.
- In source view, modify the @jc:sql annotation and the method signature.
Note: In WebLogic Workshop 7.0, Database controls used the @jws:sql annotation. The deprecated @jws:sql annotation is still supported. To learn more about the @jc:sql annotation, see @jc:sql Annotation.

The method's SQL statement may include substitution parameters. These parameters are replaced at runtime with the values that were passed to the method. The names of the substitution parameters in the SQL statement must match those in the method signature, so that WebLogic Workshop knows which parameter to replace with which value. Within the SQL statement, substitution parameters are enclosed in curly braces.

The following example Database control method illustrates using parameter substitution in Database control methods:

```
/**
 * @jc:sql statement="UPDATE customer SET address = {customerAddress} WHERE custid={customerID}"
 */
public int changeAddress(int customerID, String customerAddress);
```

In the example above, the SQL statement includes the substitutions {customerAddress} and {customerID}. These map to the customerID and customerAddress parameters of the findCustomer method. When the method is invoked, the values of any referenced parameters are substituted in the SQL statement before it is executed. Note that parameter substitution is case sensitive, so parameters mentioned in substitutions must exactly match the spelling and case of the parameters to the method.

The method signature declares a method that a user of this control may invoke. You should design this method so that its arguments and return value are convenient and useful to developers of applications that will

use this control.

The rules of parameter substitution in Database control method SQL statements are described in [Parameter Substitution in @jc:sql Statements](#).

The return type of the database operation is determined by the return type of the Java method. WebLogic Workshop attempts to format the results in whatever type you have specified for the method to return.

A method of a Database control can return a single value, a single row, or multiple rows. To learn more about the values returned by Database control methods, see the following three topics:

- [Returning a Single Value from a Database Control](#)
- [Returning a Single Row from a Database Control](#)
- [Returning Multiple Rows from a Database Control](#)

Related Topics

[Using WebLogic Built-In Controls](#)

[Database Control](#)

[Creating a New Database Control](#)

[CustomerDBClient.jws Sample](#)

[LuckyNumberDBClient.jws Sample](#)

[DatabaseControl Interface](#)

Returning a Single Value from a Database Control Method

This topic describes how to write methods that return a single value from the database. The example provided represents a `SELECT` statement that requests only a single field of a single row. The return value of the method should be an object or primitive of the appropriate type for that field's data.

Returning a Single Column

The following example assumes a `Customers` table in which the field `custid`, representing the customer ID, is the primary key. Given the customer ID, the method looks up a single customer name.

```
/**
 * @jc:sql statement="SELECT name FROM customer WHERE custid={customerID}"
 */
public String getCustomerName(int customerID);
```

In this example, the `name` field is of type `VARCHAR`, so the return value is declared as `String`. The method's `customerID` parameter is of type `int`. When the SQL statement executes, this parameter is mapped to an appropriate numeric type accepted by the database. To learn more about these relationships, see [Mapping Database Field Types to Java Types in the Database Control](#).

Related Topics

[Database Control](#)

[Creating a New Database Control](#)

[Parameter Substitution in @jc:sql Statements](#)

[Returning a Single Row from a Database Control Method](#)

[Returning Multiple Rows from a Database Control Method](#)

Returning a Single Row from a Database Control Method

This topic describes how to write methods on a Database control that return a single row from the database. When you return a single row with multiple fields, your method must have a return type that can contain multiple values—either an object that is an instance of a class that you have built for that purpose, or a `java.util.HashMap` object.

If you know the names of the fields returned by the query, you will probably want to return a custom object. If the number of columns or the particular field names returned by the query are unknown or may change, you may choose to return a `HashMap`.

Returning an Object

You can specify that the return type of a Database control method is a custom object, an instance of a class whose members correspond to fields in the database table. In most cases, a class whose members hold corresponding database field values is declared as an inner class (a class declared inside another class) in the Database control's JCX file. However, it may be any Java class that meets the following criteria:

- The class must contain members with names that match the names of the columns that will be returned by the query. Because database column names are case-insensitive, the matching names are case-insensitive. The class may also contain other members, but members with matching names are required.
- The members must be of an appropriate type to hold a value from the corresponding column in the database. For information on mapping between database types and Java types, see Mapping Database Field Types to Java Types in the Database Control.
- The class must be declared as `public static` if the class is an inner class.

The following example declares a `Customer` class with members corresponding to fields in the `Customers` table. The `findCustomer` method returns an object of type `Customer`:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}
/**
 * @sql statement="SELECT custid,name FROM customer WHERE custid={customerID}"
 */
Customer findCustomer(int customerID)
```

Note: The `Customer` class above is simplified for the sake of clarity. For data modelling classes, it is generally good design practice to have private fields, with public setter and getter methods.

```
public static class Customer
{
    private int custid;
    private String name;

    public Customer() {};
```

Working with Java Controls

```
public int getCustid()
{
    return this.custid;
}

public void setCustid(int custid)
{
    this.custid = custid;
}

public String getName()
{
    return this.name;
}

public void setName(String name)
{
    this.name = name;
}
}
```

Handling Empty Values When Returning Objects

If a database field being queried contains no value for a given row, the class member is set to null if it is an object and to 0 or false if it is a primitive. This may affect your decisions regarding the types you use in your class. If the database field contained no data, an Integer member would receive the value null, but an int member would receive the value 0. Zero may be a valid value, so using int instead of Integer makes it impossible for subsequent code to determine whether a value was present in the database.

If there is no column in the database corresponding to a member of the class, that member is also set to null or 0, depending on whether the member is an primitive or an object.

If the query returns columns that cannot be matched to the members of the class, an exception is thrown. If you don't know the columns that will be returned or they may change, you should consider returning a HashMap instead of a specific class. For more information, see the Returning a HashMap section, below.

If no rows are returned by the query, the returned value of the Database control method is null.

In the example given above, the method is declared as returning a single object of type Customer. So even if the database operation returns multiple rows, only the first row is returned to the method's caller. To learn how to return multiple rows to the caller, see Returning Multiple Rows from a Database Control Method.

Returning a HashMap

If the number of columns or the particular column names returned by the query are unknown or may change, you may choose to return a HashMap. To return a HashMap, declare the return value of the method as java.util.HashMap, as shown here:

```
/**
 * @jc:sql statement="SELECT * FROM customer WHERE custid={custID}"
 */
public java.util.HashMap findCustomerHash(int custID);
```

Working with Java Controls

The `HashMap` returned contains an entry for each column in the result. The key for each entry is the corresponding column name. The capitalization of the key names returned by `HashMap.keySet()` depends on the database driver in use, but all keys are case-insensitive when accessed via the `HashMap`'s methods. The value is an object of the Java Database Connectivity (JDBC) default type for the database column. To learn more about mapping between database types, Java types and JDBC types, see [Mapping Database Field Types to Java Types in the Database Control](#).

In the example above, the method is declared as returning a single object of type `java.util.HashMap`. So even if the database operation returns multiple rows, only the first row is returned to the method's caller.

To learn how return multiple rows to the caller, see [Returning Multiple Rows from a Database Control Method](#)

The following code allows you to access the name field of the returned record:

```
/**
 * @common:control
 */
private CustomerDBControl custDB;

/**
 * @common:operation
 */
public String getCustomerName(int custID)
{
    java.util.HashMap hash;
    String name;
    hash = custDB.findCustomerHash(custID);
    if( hash != null )
    {
        name = (String)hash.get( "NAME" );
    }
    else
    {
        name = new String("Customer not found");
    }
    return name;
}
```

If the query returns no rows, the returned value of the Database control method is null.

Related Topics

[Parameter Substitution in @jc:sql Statements](#)

[Mapping Database Field Types to Java Types in the Database Control](#)

Returning Multiple Rows from a Database Control Method

This topic describes how to write a method on a Database control that returns multiple rows from the database. It describes the ways in which you can perform this operation, including returning an array, returning an Iterator object, and returning a resultset.

Deciding How to Return Multiple Rows

A SELECT query may return one or more fields from multiple rows. A method on a Database control that returns multiple rows should have a return type that can store these values. The Database control method can return an array of objects, an Iterator, or a resultset.

Returning an array of objects is the easiest way to return multiple rows, so it is a good choice if you think your users will prefer simplicity when using your control. However, when an array is returned only one database operation is performed and the entire resultset must be stored in memory. For large resultsets, this is problematic. You can limit the size of the returned array, but then you cannot provide a way for your user to get the remainder of the resultset. To learn how to return an array of objects, see the Returning an Array of Objects section, below.

While Iterators require more sophistication on the part of users of your control, they are more efficient at handling large resultsets. An Iterator is accessed one element (row) at a time via the Iterator's next() method, and it transparently makes repeated requests from the database until all records have been processed. An Iterator does not present the risk of running out of memory that an array presents. However, note that an Iterator returned from a database control cannot be used within a Page Flow file (JPF), because an Iterator wraps a ResultSet object, which is always closed by the time it is passed to the web-tier (where page flow files reside). For this reason, your database control should return an array of objects (see above) when it is called from a Page Flow file. Also, an Iterator cannot be returned to a stateful process, because stateful processes cannot maintain an open database connection (which Iterators require). To learn about returning a java.util.Iterator, see the Returning an Iterator section, below.

Finally, you can choose to return a java.sql.ResultSet from a Database control method. This grants complete access to the results of the database operation to clients of your control, but it requires knowledge of the java.sql package. Also, note that a ResultSet returned from a database control cannot be used within a page flow file (JPF), because a ResultSet object is always closed by the time it is passed to the web-tier (where page flow files reside). For this reason, your database control should provide an array of objects when it is called from a page flow file. To learn about returning a java.sql.ResultSet, see the Returning a Resultset section, below.

Returning an Array of Objects

To return an array of objects, declare the method's return type to be an array of the object you want to return. That type may be either a type you define, or it may be java.util.HashMap.

Examples of both of these techniques are provided in the following sections.

Returning an Array of User-Defined Objects

The following example demonstrates how to return an array of objects whose type you have declared. In this case, an array of Customer objects is returned:

```
public static class Customer
{
    public int custid;
    public String name;
}
/**
 * @jc:sql statement="SELECT custid,name FROM customer WHERE custage<19"
 *     array-max-length=100
 */
Customer [] findAllMinorCustomers()
```

This example returns all rows in which the custage field contains a value less than 19.

When returning an array of objects, the class declared as the return type of the method must meet the criteria described in the Returning an Object section of the Returning a Single Row from a Database Control topic. If no rows are returned by the query, the returned value of the Database control method is a zero-length array.

If you are returning an array from Database control method, you can limit the size of the array returned by setting the array-max-length attribute of the @jc:sql annotation. This attribute can protect you from very large resultsets that may be returned by very general queries. If array-max-length is present, no more than that many rows are returned by the method.

The default value of array-max-length is 1024.

The array-max-length attribute may have the special value "all" (in quotes), indicating that all rows satisfying the query should be returned. Note that if the query is too general, this can result in use of all available memory. To avoid excessive memory usage, return an Iterator object as described below in the Returning an Iterator section, below.

Returning an Array of HashMaps

Returning an array of HashMaps is analogous to returning an array of user-defined objects, which is described in the preceding section.

The following example demonstrates returning an array of HashMaps:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {}
}
/**
 * @jc:sql statement="SELECT custid,name FROM customer WHERE custage<19"
 *     array-max-length=100
 */
java.util.HashMap [] findAllMinorCustomersHash()
```

Working with Java Controls

The array of HashMaps returned contains an element for each row returned, and each element of the array contains an entry for each column in the result. The key for each entry is the corresponding column name. The capitalization of the key names returned by `HashMap.keySet()` depends on the database driver in use, but keys are case-insensitive when accessed via the `HashMap`'s methods. The value returned is an object of the Java Database Connectivity (JDBC) default type for the database column.

If no rows are returned by the query, the returned value of the Database control method is a zero-length array.

For information on mapping between Java types and JDBC types (database types), see [Mapping Database Field Types to Java Types in the Database Control](#).

The following code shows how to access the name field of the returned records:

```
/**
 * @common:control
 */
private CustomerDBControl custDB;

java.util.HashMap [] hashArr;
String name;

hashArr = custDB.findAllMinorCustomersHash();
for(i=0; i<hashArr.length; i++)
{
    name = (String)hashArr[i].get("NAME");
    // say hello to the all of the minors

    System.out.println("Hello, " + name + "!");
}
```

Returning an Iterator

When you want to return an `Iterator` object, you declare the method's return type to be `java.util.Iterator`. You then add the `iterator-element-type` attribute to the `@jc:sql` annotation to indicate the underlying type that the `Iterator` will contain. The specified type may be either a type you define, or it may be `java.util.HashMap`. Examples of these techniques are given in the following sections.

The `Iterator` that is returned is only guaranteed to be valid for the life of the method call to which it is returned. You should not store an `Iterator` returned from a Database control method as a static member of your web service's class, nor should you attempt to reuse the `Iterator` in subsequent method calls if it is persisted by other means.

Returning an Iterator with a User-Defined Object

To return an `Iterator` that encapsulates a user-defined type, provide the class name as the value of the `iterator-element-type` attribute of the `@jc:sql` annotation, as shown here:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {}
}
/**
```

Working with Java Controls

```
* @sql statement="SELECT custid,name FROM customer"
*     iterator-element-type="Customer"
*/
java.util.Iterator getAllCustomersIterator()
```

The class specified in the `iterator-element-type` attribute must meet the criteria described in [Returning an Object](#).

The following example shows how to access the returned records:

```
CustomerDBControl.Customer cust;
java.util.Iterator iter = null;
iter = custDB.getAllCustomersIterator();
while (iter.hasNext())
{
    cust = (CustomerDBControl.Customer)iter.next();
    // say hello to every customer
    System.out.println("hello, " + cust.name + "!");
}
```

Returning an Iterator with HashMap

To return an Iterator that encapsulates a HashMap, provide `java.util.HashMap` as the value of the `iterator-element-type` attribute of the `@jc:sql` annotation, as shown here:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {}
}
/**
 * @sql statement="SELECT custid,name FROM customer"
 *     iterator-element-type="java.util.HashMap"
 */
java.util.Iterator getAllCustomersIterator()
```

The following code shows how to access the returned records:

```
java.util.HashMap custHash;
java.util.Iterator iter = null;
int customerID;
String customerName;
iter = custDB.getAllCustomersIterator();
while (iter.hasNext())
{
    custHash = (java.util.HashMap)iter.next();
    customerID = (int)custHash.get("CUSTID");
    customerName = (String)custHash.get("NAME");
}
```

The HashMap contains an entry for each database column that is returned by the query. The key for each entry is the corresponding column name, in all uppercase. The value is an object of the JDBC default type for the database column.

For information on mapping between Java types and JDBC types (database types), see [Mapping Database](#)

Field Types to Java Types in the Database Control.

Returning a Resultset

The Database control is designed to allow you to obtain data from a database in a variety of ways without having to understand the classes in the `java.sql` package. If you and your users do understand these classes, however, you can gain complete access to the `java.sql.ResultSet` object returned by a query.

If you want to return a resultset, you declare the method's return type to be `java.sql.ResultSet`. A client of your control then accesses the resultset directly to process the results of the database operation.

The following example demonstrates returning a resultset:

```
/**
 * @jc:sql statement="SELECT * FROM customer"
 */
public java.sql.ResultSet findAllCustomersResultSet();
```

The following code shows how to access the returned resultset:

```
java.sql.ResultSet resultSet;
String thisCustomerName;
resultSet = custDB.findAllCustomersResultSet();
while (resultSet.next())
{
    thisCustomerName = new String(resultSet.getString("name"));
}
```

This example assumes the rows returned from the database operation include a column called `name`.

Related Topics

[Returning a Single Value from a Database Control Method](#)

[Returning a Single Row from a Database Control Method](#)

[Parameter Substitution in @jc:sql Statements](#)

Modifying Data with a Database Control

You can use the Database control to perform INSERT, UPDATE, and DELETE operations, which return the number of rows affected.

In the following example, an UPDATE operation is performed and the number of rows affected is returned:

```
/**
 * @jc:sql statement="UPDATE customer SET address={customerAddr} WHERE custid={customerID}"
 */
public int setCustomerAddress (int customerID, String customerAddr);
```

This example updates the customer table. For each record in the table in which the custid field matches the value of the customerID parameter, the address field is set to the value of the customerAddr parameter.

Related Topics

Overview: Database Controls

Returning XMLBeans from a Database Control

The following topic explains how to return XMLBean types from both RowSet and custom database controls.

An XMLBean is essentially an XML document with a Java API attached to it. The API is used for parsing and manipulating the data in the XML document. A typical XMLBean might represent database data in the following form.

```
<!DOCTYPE XCustomer>
<XCustomer xmlns="java:///database/customer_db" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <XCustomerRow>
    <CUSTID>1</CUSTID>
    <NAME>Fred Williams</NAME>
    <ADDRESS>123 Slugger Circle</ADDRESS>
  </XCustomerRow>
  <XCustomerRow>
    <CUSTID>2</CUSTID>
    <NAME>Marnie Smithers</NAME>
    <ADDRESS>5 Hitchcock Lane</ADDRESS>
  </XCustomerRow>
  <XCustomerRow>
    <CUSTID>3</CUSTID>
    <NAME>Bill Walton</NAME>
    <ADDRESS>655 Tall Timbers Road</ADDRESS>
  </XCustomerRow>
</XCustomer>
```

The data can be accessed and manipulated using the XMLBean's API. For example, assume that custBean represents the XML document above. The following Java code extracts the Fred Williams from the document.

```
String name = custBean.getXCustomer().getXCustomerRowArray(1).getName();
```

Retrofitting database controls to return XMLBeans rather than RowSets, ResultSets, or Iterators, is a powerful technique because there are few restrictions on where XMLBeans can be imported. This is not the case with ResultSets and Iterators, which cannot be passed directly to web-tier classes (web services and page flows). Also, data in XMLBean form is very easy to manipulate because there is a rich API attached to the XMLBean.

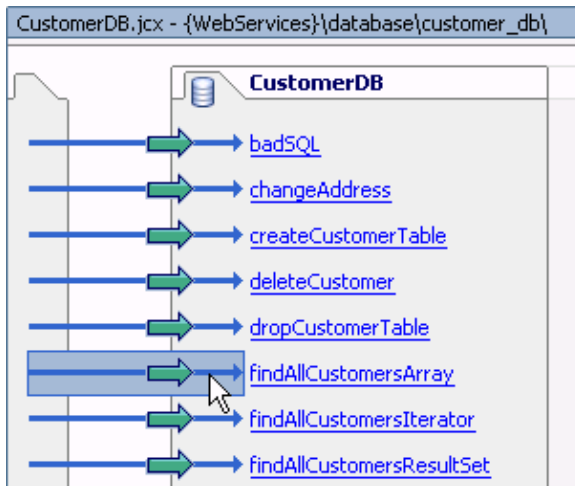
Creating a Schema

The first step in using XMLBean classes is creating a schema from which the XMLBean classes can be generated. The schema you create for a database control must be capable of modeling the sorts of data returned from the database.

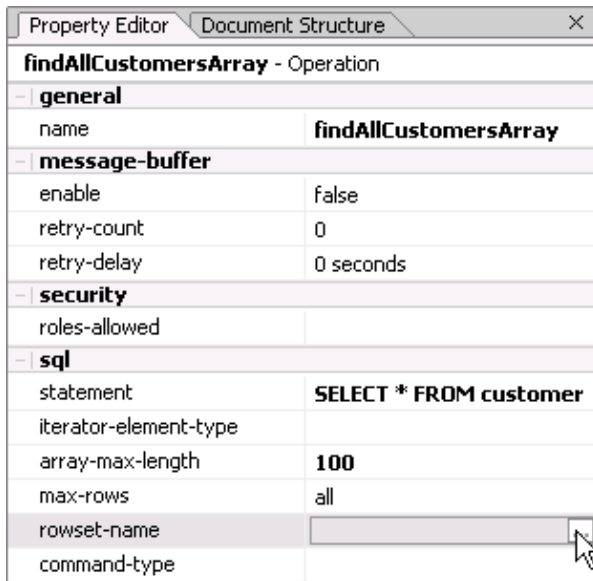
If you write your own schema, at a minimum, the schema's elements should have the same names as the fields in the database, which allows data returned from the database to be automatically mapped into the XMLBean.

To autogenerate a schema for a database or RowSet control, follow this procedure.

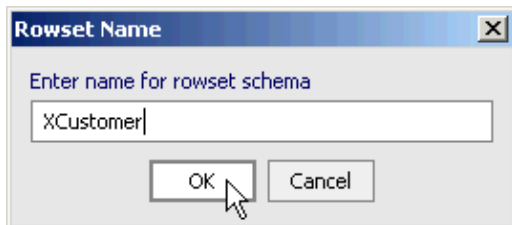
1. Display the database control file in Design View.
2. Select a method that returns data from the database. (A method that does not return data, such as a CREATE TABLE statement will not work.)



3. On the Properties Editor tab, in the section labeled **sql**, select the property named **rowset-name**.



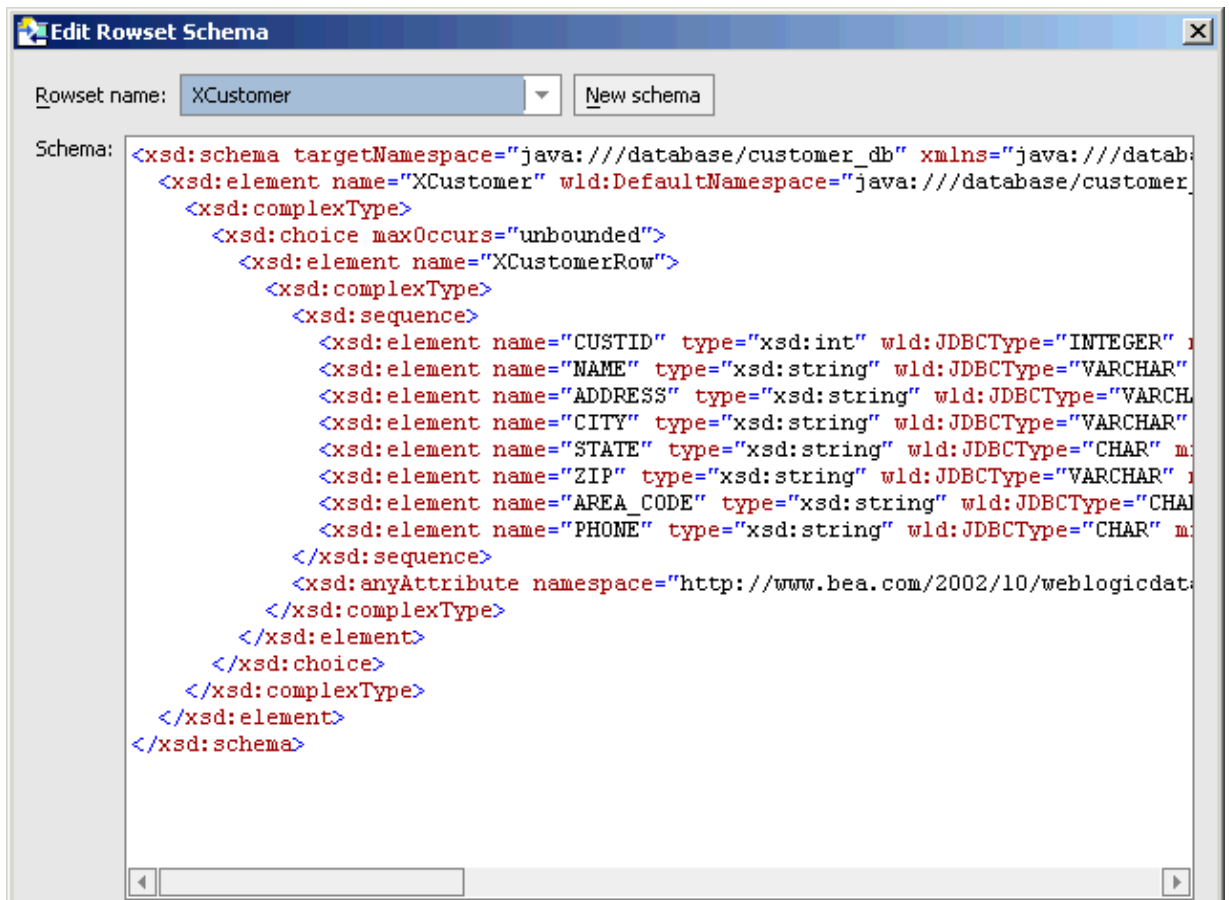
4. Enter a RowSet name. (The name you enter will become part of the schema, and the XMLBean classes derived from the schema.)



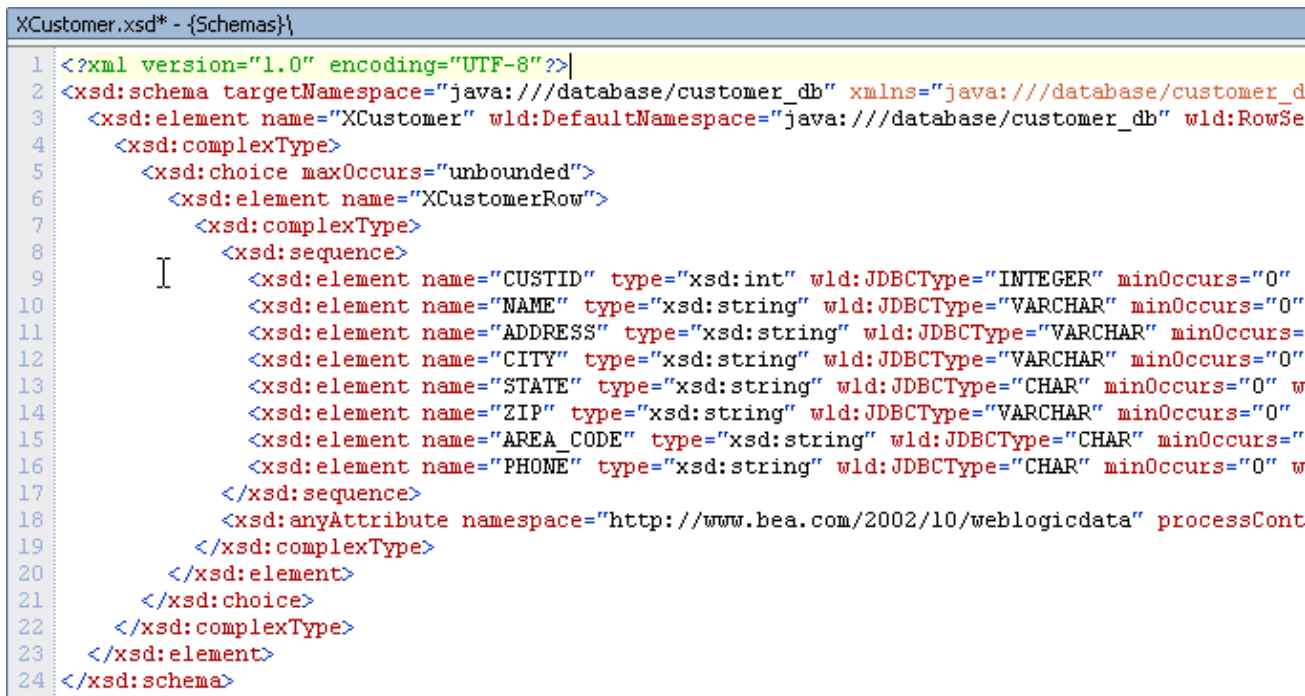
5. Select and copy the schema text from the Edit RowSet Schema dialog. (The dialog does not support right-clicking, you must use **Ctrl+C** to copy the schema text.)
Note that XCustomer appears in the schema in two places:

```
<xsd:element name="XCustomer" ...
```

```
<xsd:element name="XCustomerRow">
```

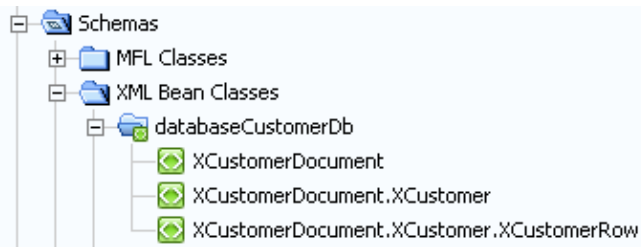



6. Cancel out of the dialog and save the schema text in an XSD file in a Schemas project. Make sure that the XSD file begins with the element `<?xml version="1.0"?>`.



When the XSD file is compiled, XMLBean types are generated that can be returned by the methods in the

database control. The XMLBean classes appear in the Schema project's XML Bean Classes folder.



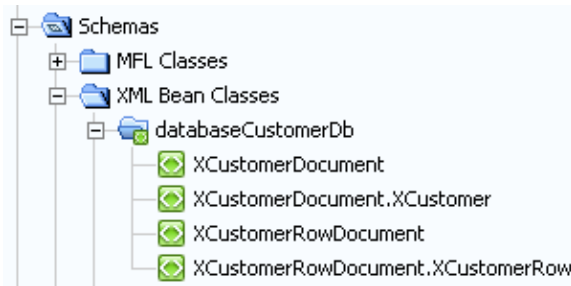
Editing Schemas to Create New "Document" Types

Note that only one of the generated types is a "Document" XMLBean type: `XCustomerDocument`. The other types, `XCustomerDocument.XCustomer` and `XCustomerDocument.XCustomer.XCustomerRow`, can only be used with reference to the "Document" type. This distinction is especially important because only "Document" types are eligible for direct participation in a business process, or to be passed to a web service. For this reason you may want to edit your schema to include "Document" types corresponding to other types in the Schema, especially if you have a very large schema with many nested types defined in terms of a single "Document" type.

To generate a new Document type for some element, move that element so that it becomes a top-level element in the schema. In the following example, the `XCustomerRow` element has been moved to the top-level of the schema: its original position has been replaced with a reference element: `<xsd:element ref="XCustomerRow"/>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="java:///database/customer_db" xmlns="java:///database/customer_db"
  <xsd:element name="XCustomer" wld:DefaultNamespace="java:///database/customer_db" wld:RowSet="true"
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element ref="XCustomerRow"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="XCustomerRow">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CUSTID" type="xsd:int" wld:JDBCType="INTEGER" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="NAME" type="xsd:string" wld:JDBCType="VARCHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="ADDRESS" type="xsd:string" wld:JDBCType="VARCHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="CITY" type="xsd:string" wld:JDBCType="VARCHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="STATE" type="xsd:string" wld:JDBCType="CHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="ZIP" type="xsd:string" wld:JDBCType="VARCHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="AREA_CODE" type="xsd:string" wld:JDBCType="CHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
        <xsd:element name="PHONE" type="xsd:string" wld:JDBCType="CHAR" minOccurs="0" wld:Tablename="CUSTOMER" />
      </xsd:sequence>
      <xsd:anyAttribute namespace="http://www.bea.com/2002/10/weblogicdata" processContents="lax" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

There are now two top-level elements, `XCustomer` and `XCustomerRow`, which compile into two corresponding "Document" types: `XCustomerDocument` and `XCustomerRowDocument`.



Returning a XMLBean Types from Control Methods

Once you have generated XMLBean types that model the database data, you can import these types into your database control.

```

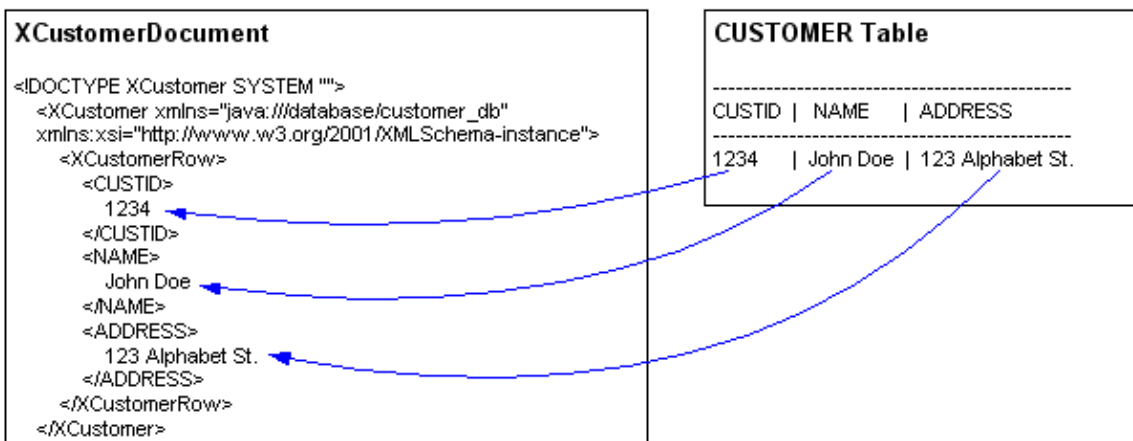
import databaseCustomerDb.XCustomerDocument;
import databaseCustomerDb.XCustomerDocument.XCustomer;
import databaseCustomerDb.XCustomerDocument.Factory;
  
```

XMLBean types can be returned from the control's methods.

```

/**
 * @jc:sql statement="SELECT custid, name, address FROM customer"
 */
public XCustomerDocument findAllCustomersDoc();
  
```

The data returned from the query is automatically mapped into the XMLBean because the names of the database fields match the fields of the XMLBean.



Related Topics

Java Types Generated from User-Derived Schema Types

@jc:sql Annotation

Samples

CustomerDB_XMLBean Sample

ItemsDB_XMLBean Sample

Parameter Substitution in @jc:sql Statements

You can use parameter substitution in the @jc:sql annotation's statement attribute to form a query dynamically. The client calls the method on the Database control, passing in values for the method's parameters, and those parameter values are substituted into the SQL statement.

This topic describes substitution techniques and rules, including how to treat curly braces, how to substitute whole SQL statements, SQL phrases, simple parameters, and indirect parameters.

Substitution Criteria

Substitution is subject to the following criteria:

Substitution matching is case sensitive. For example, the method parameter CustCity will *not* match the substitution pattern {custCity}.

The type of the method parameter must be compatible with the type of the associated database field in the statement. If you attempt to substitute a Java String where the database expects a NUMBER, the statement will fail. For information on mapping between database types and Java types, see Mapping Database Field Types to Java Types in the Database Control.

Substitution will not occur if the substitution pattern contains spaces. The Java Database Connectivity (JDBC) API allows access to built-in database functions via *escapes* of the form {fn user()}. If spaces occur in an item enclosed in curly braces ({ }) item, the Database control treats the item as a JDBC escape and passes it on without substitution. For example, the custCity method parameter will not be substituted if the substitution is specified as {custCity } or { custCity}. For more information on JDBC escapes, please consult the documentation for your JDBC driver.

When substituting date or time values, use the classes in the java.sql package. For example, attempting to substitute java.util.Date in a SQL Date field will not work. Use java.sql.Date instead.

Generic Substitution

To pass a whole SQL statement to the database, use the substitution syntax shown in red.

```
/**
 * @jc:sql statement="{sql: sqlStatement}"
 */
public myRecordType myQuery( String sqlStatement );
```

The SQL statement placed within the bracket syntax {sql: } is escaped and passed directly to the database.

You can use same substitution syntax to pass in any part of a SQL statement, such as a WHERE or LIKE clause, or a column name. In the following example, filtering phrases can be substituted into the base SQL statement.

```
/**
 * @jc:sql statement="SELECT * FROM CUSTOMER {sql: whereClause}"
 */
public myRecordType myQuery( String whereClause );
```

In the following example, a column name is dynamically written to the SQL statement by means of the {sql: } bracket syntax.

```
/**
 * @jc:sql statement="SELECT SUM( {sql: colName} ) FROM MYTABLE"
 */
public int sumColumn(String colName);
```

Referring to Functions in Substitution Statements

If your database supports internal functions, you can refer to the internal function within the substitution syntax {sql: }. The following method refers to the function in(), by placing the function call within the brackets {sql: }.

```
/**
 * @jc:sql statement="SELECT * FROM customer WHERE {sql:fn in(custid,{customerIDs})}"
 */
Customer[] callInternalFunction(Integer[] customerIDs);
```

Not all databases and database drivers support internal functions within substitution brackets, for example, Oracle drivers do not support this scenario.

Substituting Simple Parameters

If you are substituting individual values into a WHERE, LIKE, or AND clause, you may substitute them directly in the @jc:sql annotation's statement parameter without escaping the values with the {sql: } substitution syntax.

The following example illustrates simple parameter substitution:

```
/**
 * @jc:sql statement="SELECT name FROM customer WHERE city={custCity} AND state={custState}"
 */
public String [] getCustomersInCity( String custCity, String custState );
```

The value of the custCity method parameter is substituted in the query in place of the {custCity} item, and the value of the custState method parameter is substituted in the query in place of the {custState} item.

Treatment of Curly Braces Within Literals

Curly braces ({}) within literals (strings within quotes) are ignored. This means statements like the following will not work as you might expect. In the following example the curly braces have lost their substitution functionality, because they appear within single quotes.

```
/**
 * @jc:sql statement::
 *     SELECT name
 *     FROM employees
 *     WHERE name LIKE '%{partialName}%'
 * ::
 */
public String[] partialNameSearch(String partialName);
```

Working with Java Controls

Since the curly braces are ignored inside the literal string, the expected substitution of the `partialName` Java String into the SELECT statement does not occur. To avoid this problem, pre-format the match string before invoking the Database control method, as shown below. Note that single quotes are not included in the pre-formatted string because single quotes are implicitly added to the substitution value when it is passed to the SQL query.

```
String partialNameToMatch = "%" + matchString + "%"
String [] names = myDBControl.partialNameSeach(partialNameToMatch);
```

Then pass the pre-formatted string to the Database control:

```
/**
 * @jc:sql statement::
 *      SELECT name
 *      FROM employees
 *      WHERE name LIKE {partialNameToMatch}
 * ::
 */
public String[] partialNameSearch(String partialNameToMatch);
```

Substituting Indirect Parameters

Assume the following class is declared and is accessible to the Database control:

```
public static class Customer
{
    public String firstName;
    public String lastName;
    public String streetAddress;
    public String city;
    private String state;
    public String zipCode;
    public String getState() {return state;}
}
```

You can then refer to the members of the Customer class in the SQL statement, as shown in the following example:

```
/**
 * @jc:sql statement="SELECT name FROM customer WHERE city={cust.city} AND state={cust.state}"
 */
public String [] getCustomersInCity( Customer cust );
```

Note: Class member variables and accessor (getXxx) methods must be public in order for the Database control to substitute them.

The dot notation is used to access the members of the parameter object.

The following list describes the precedence for resolving dot notations in substitutions given the substitution pattern {myClass.myMember}:

- If class `myClass` exposes public `getMyMember()` *and* `setMyMember()` methods, `getMyMember()` is called and the return value is substituted. For Boolean variables, substitute `isMyMember()` for `getMyMemnber()`.

Working with Java Controls

- Else if class myClass exposes a public field named myMember, myClass.myMember is substituted.
- Lastly, if class myClass implements java.util.Map, myClass.get("myMember") is called and the return value is substituted.
- Any combination of these may exist, as in {A.B.C} where B is a public member of A and B has a public getC() method.

If none of these conditions exist, the Database control method will throw a com.bea.control.ControlException.

Calling Stored Procedures and Functions

For detailed information on calling Stored procedures and functions see the help topics Stored Procedures and Stored Functions.

Related Topics

Mapping Database Field Types to Java Types in the Database Control

Returning a Single Value from a Database Control Method

Returning a Single Row from a Database Control Method

Returning Multiple Rows from a Database Control Method

Stored Procedures

Stored Functions

Automatically Generating Primary Key Values with Database Controls

The following topic how to auto-generate primary key values using an Oracle SEQUENCE and a SQL Server IDENTITY field. The topic also explains how to use these auto-generation techniques in conjunction with a Workshop database control.

The topics in this section are:

- Primary Key Generation Using Oracle's Sequence
- Primary Key Generation Using SQL Server's IDENTITY

Primary Key Generation Using an Oracle Sequence

Oracle provides the 'sequence' utility to automatically generate unique primary keys.

In your Oracle database, you must create a sequence table that will create the primary keys, like is shown in the following example:

```
/**
 * @jc:sql statement::
 * CREATE SEQUENCE MY_ORACLE_SEQUENCE
 * START WITH 1
 * NOMAXVALUE::
 */
public void createSequence() throws SQLException;
```

This creates a sequences of primary key, starting with 1, followed by 2, 3, and so forth. The sequence table in the example uses the default increment 1, but you can change this by specifying the increment keyword, such as increment by 3.

When inserting new records into a table, you can generate a new primary key value by referring to the next value in the sequence.

```
/**
 * @jc:sql statement="INSERT INTO CUSTOMER VALUES (MY_ORACLE_SEQUENCE.NEXTVAL, {firstname},
 */
void insertCustomer(String firstname, String lastname);
```

When database processes the insert statement, it will consult the sequence for the next value, and use that value as the primary key value of the new record.

Primary Key Generation Using a SQL Server IDENTITY Field

In SQL Server (2000) you can use the IDENTITY keyword to indicate that a primary-key needs to be auto-generated. The following example shows a common scenario where the first primary key value is 1, and the increment is 1:

```
/**
```

Working with Java Controls

```
* @jc:sql statement::
* CREATE TABLE jobz
* (
*     job_id smallint
*         IDENTITY(1,1)
*     PRIMARY KEY CLUSTERED,
*     job_desc varchar(50) NOT NULL
* )::
*/
void createTable();
```

When new records are inserted into the table, SQL Server will generate a primary key value: you do not need to pass any primary key value to the database when inserting records.

```
/**
 * @jc:sql statement::
 * INSERT INTO jobz (job_desc)
 * VALUES ({description})::
 */
public int insertJob(String description);
```

Related Topics

[@ejbgen:automatic-key-generation Annotation](#)

Stored Functions

This topic explains how to call and create stored functions using a database control file.

Calling Stored Functions

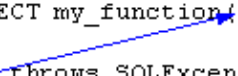
To call a stored function, place the function call in a `@jc:sql` statement annotation. When the Java method `callMyFunction` is called the SQL statement in the `@jc:sql` statement annotation is passed to the database. Any data returned by the SQL statement is passed back to, and returned by, the Java method.

```
/**
 * @jc:sql statement="SELECT my_function FROM DUAL"
 */
int callMyFunction() throws SQLException;
```

In most cases, WebLogic Workshop automatically converts between the appropriate database data types to the Java data types. For example, if the database function `my_function` returns the database type `INTEGER`, the Java method `callMyFunction()` will automatically convert it into the Java type `int`. Standard datatype mappings are described in detail in the help topic [Mapping Database Field Types to Java Types in the Database Control](#).

You can substitute values dynamically into the database function call using curly braces. The following method passes the parameter `int x` to the function call.

```
/**
 * @jc:sql statement="SELECT my_function({x}) FROM DUAL"
 */
int callMyFunction(int x) throws SQLException;
```



Before the function call is passed to the database, WebLogic Workshop constructs a complete function call from the static elements and the substituted value. Assuming that the parameter `x` has the value `3`, the following function call is constructed and passed to the database.

```
SELECT my_function(3) FROM DUAL
```

Creating Stored Functions

You can also send any DDL statement to the database through a database control method.

```
/**
 * A stored function that takes an integer, squares it, and returns the
 * result through the database control method.
 *
 * @jc:sql statement::
 CREATE OR REPLACE FUNCTION fn_squareInt
   (field1 IN INTEGER)
   RETURN INTEGER IS field2 INTEGER;
BEGIN
   field2 := field1 * field1;
   RETURN field2;
END fn_squareInt;
```

Working with Java Controls

```
::  
*/  
void create_fn_squareInt() throws SQLException;
```

Some XA database drivers contain restrictions on code that rollback or commits a transaction independently of the driver's transaction management. Since DDL statements are implicitly transactional (COMMIT is called whether it or not it explicitly appears in the DDL statement), you may have to suspend the transaction with these XA drivers. For example if you send a DDL statement using the Oracle XA thin client without suspending the transaction, the driver throws the following exception.

```
ORA-02089: COMMIT is not allowed in a subordinate session
```

The following code suspends the transaction, executes the DDL statement, and then resumes the transaction.

```
import javax.transaction.Transaction;  
import weblogic.transaction.TransactionManager;  
import weblogic.transaction.TxHelper;  
  
    TransactionManager tm = TxHelper.getTransactionManager();  
Transaction saveTx = null;  
try  
{  
  
    // Suspend the transaction  
    saveTx = tm.forceSuspend();  
  
    // Execute the DDL statement  
    myDBCControlFile.create_fn_squareInt();  
}  
finally  
{  
  
    // Resume the transaction  
    tm.forceResume(saveTx);  
}
```

Related Topics

[Parameter Substitution in @jc:sql Statements](#)

[Stored Procedures](#)

[@jc:sql Annotation](#)

Stored Procedures

The following topic explains how to call and create stored procedures using a database control file.

Calling Stored Procedures with IN Parameters

If the stored procedure contains only IN parameters, you can call the procedure by passing method parameters to the procedure.

Assume the following procedure `sp_updateData` has been created on the database.

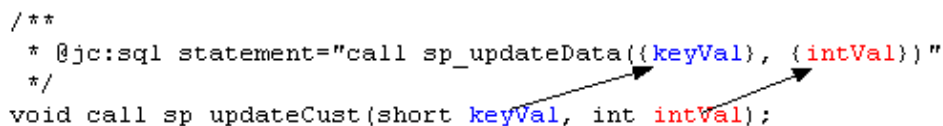
```
CREATE OR REPLACE PROCEDURE sp_updateData
    (pkID    IN SMALLINT,
     intVal  IN INT)
AS
BEGIN
    UPDATE CUSTOMER
    SET NAME = intVal
    WHERE CUSTID = pkID;
END sp_updateData;
```

The following database control method calls the procedure `sp_updateData` and passes two method parameters to the procedure.

```
/**
 * @jc:sql statement="call sp_updateData({keyVal}, {intVal})"
 */
void call_sp_updateCust(short keyVal, int intVal);
```

Note that the method parameters are substituted into the procedure call using the curly brace substitution syntax.

```
/**
 * @jc:sql statement="call sp_updateData({keyVal}, {intVal})"
 */
void call_sp_updateCust(short keyVal, int intVal);
```



Before the procedure call is passed to the database, WebLogic Workshop constructs a complete statement from the static elements and the substituted values. Assuming that `keyVal` and `intVal` have the values 1 and 2 respectively, the complete statement sent to the database is

```
call sp_updateData(1, 2)
```

Calling Stored Procedures with OUT Parameters

To call a procedure that contains OUT parameters: (1) use a `SqlParameter` Array as the parameter of the Java method that calls the procedure and (2) use question marks as placeholders for the parameters within the procedure call.

For example, assume that the following procedure `sp_squareInt` exists on the database.

```
CREATE OR REPLACE PROCEDURE sp_squareInt
```


Working with Java Controls

```
(field1 IN INTEGER, field2 OUT INTEGER) IS
BEGIN
    field2 := field1 * field1;
END sp_squareInt;
```

The following Java method will call the procedure sp_squareInt.

```
/**
 * @jc:sql statement="{call sp_squareInt(?, ?)}"
 */
void call_sp_squareInt(SQLParameter[] params) throws SQLException;
```

Note that the method parameter params is not explicitly substituted into the procedure call {call sp_squareInt(?, ?)}. The substitution syntax {call ...} has special meaning within the jc:sql statement annotation. When WebLogic Workshop encounters the substitution syntax {call myStoredProc(?,?,?...)}, it automatically distributes the elements of params into the procedure call.

```
/**
 * @jc:sql statement="{call sp_squareInt(?, ?)}"
 */
void call_sp_squareInt(SQLParameter[]  params) throws SQLException;
```

The following shows how to construct an SQLParameter[] to call the procedure sp_squareInt.

```
// Construct a SQLParameter[]
// to hold two SQLParameter objects
SQLParameter[] params = new SQLParameter[2];

// Construct two objects corresponding to the initial values of the
// stored procedure's two parameters.
Object obj0 = new Integer(x);
Object obj1 = new Integer(0);

// The stored procedure sp_squareInt has two parameters:
// an IN parameter of data type INTEGER
// and an OUT parameter of data type INTEGER.
// params[0] is build to correspond to the IN parameter,
// params[1] is build to correspond to the OUT parameter.
params[0] = new SQLParameter(obj0, Types.INTEGER, SQLParameter.IN);
params[1] = new SQLParameter(obj1, Types.INTEGER, SQLParameter.OUT);

// Call the stored procedure.
// Note that the procedure does not return any value.
// Instead the result of the procedure is loaded directly into the OUT parameter,
// and, in turn, into params[1].
myDBControlFile.call_sp_squareInt(params);

// Get the result loaded directly into params[1].
return Integer.parseInt(params[1].value.toString());
```

Note that database control method call_sp_squareInt does not return the result of the procedure call. Instead the result of the procedure is loaded directly into the procedure's OUT parameter, and this in turn is loaded directly into the corresponding SQLParameter object. To get the result of the procedure, examine the .value property of the of the SQLParameter object.

```
params[1].value
```

Wrapping Procedures in Functions

An alternative to calling stored procedures directly is to wrap them in stored functions, then call the wrapping function from your database control file.

For example the following database control method will create a function that wraps the procedure `sp_squareInt`.

```
/**
 * Wraps a procedure in a function.
 *
 * @jc:sql statement::
 * CREATE OR REPLACE FUNCTION wrapProc
 *   (p1 INTEGER)
 *   RETURN INTEGER IS p2 INTEGER;
 * BEGIN
 *   -- Call the stored procedure sp_squareInt. p2 corresponds to sp_squareInt's OUT p
 *   sp_squareInt(p1, p2);
 *   RETURN p2;
 * END;
 * ::
 */
public void create_wrapProc();
```

Once the procedure has been wrapped, you can call the function, instead of calling the procedure directly.

```
/**
 * @jc:sql statement::
 * SELECT wrapProc({x}) FROM DUAL
 * ::
 */
public int callWrapProc(int x, int y);
```

Creating Stored Procedures

You can also send any DDL statement to the database through a database control method.

```
/**
 * A stored procedure that takes an integer, squares it, and loads
 * the result into an OUT parameter.
 *
 * @jc:sql statement::
 * CREATE OR REPLACE PROCEDURE sp_squareInt
 *   (field1 IN INTEGER, field2 OUT INTEGER) IS
 * BEGIN
 *   field2 := field1 * field1;
 * END sp_squareInt;
 * ::
 */
void create_sp_squareInt() throws SQLException;
```

Some XA database drivers contain restrictions on code that rollback or commits a transaction independently of the driver's transaction management. Since DDL statements are implicitly transactional (COMMIT is called whether it or not it explicitly appears in the DDL statement), you may have to suspend the transaction with these XA drivers. For example if you send a DDL statement using the Oracle XA thin client without

Working with Java Controls

suspending the transaction, the driver throws the following exception.

```
ORA-02089: COMMIT is not allowed in a subordinate session
```

The following code suspends the transaction, executes the DDL statement, and then resumes the transaction.

```
import javax.transaction.Transaction;
import weblogic.transaction.TransactionManager;
import weblogic.transaction.TxHelper;

    TransactionManager tm = TxHelper.getTransactionManager();
Transaction saveTx = null;
try
{
    // Suspend the transaction
    saveTx = tm.forceSuspend();

    // Execute the DDL statement
    myDBControlFile.create_sp_squareInt();
}
finally
{
    // Resume the transaction
    tm.forceResume(saveTx);
}
```

Related Topics

[Parameter Substitution in @jc:sql Statements](#)

[Stored Functions](#)

[@jc:sql Annotation](#)

Limiting the Size of Returned Data

This topic explains how to limit the size of data sets returned from the database.

You can set the maximum number of records returned by the database at design time or at runtime.

Setting Max Records at Design Time

To set the maximum records at design time, use the @jc:sql max-rows attribute. The follow method returns the first four records that match the query SELECT * FROM ITEMS.

```
/**
 * @jc:sql max-rows="4" statement="SELECT * FROM ITEMS"
 */
public Item[] allItemsArray();
```

Note that the max-rows attribute is a design time device only that you cannot set dynamically at run time. The the following section for information on limiting the size of returned data dynamically.

Setting Max Records at Run Time

To set the maximum number of rows dynamically at run time, use the syntax provided by the database vendor. The following table shows the different syntax used by different database vendors. Note that you can use the substitution syntax {sql: ...} to limit the number of rows dynamically.

<i>Vendor</i>	<i>Method for Limiting Data Sets</i>	<i>Sample Query</i>	<i>Sample Dynamic Database Control Method</i>
DB2	Use a FETCH FIRST N clause.	SELECT * FROM ITEMS FETCH FIRST 4 ROWS ONLY	<pre>/** * @jc:sql statement="SELECT * FROM ITEMS FETCH FIRST {sql: n} */ public Item[] getNItems_DB2Syntax(int n);</pre>
Oracle	Use a nested query	SELECT * FROM (SELECT * FROM ITEMS ORDER BY ITEMNUMBER DESC) WHERE ROWNUM <= 4	<pre>/** * @jc:sql statement:: * SELECT * FROM * (SELECT * FROM ITEMS ORDER BY ITEMNUMBER DESC) * WHERE ROWNUM <= {sql: n} * :: */ public Items[] getNItems_OracleSyntax(int n);</pre>
MS SQL Server	Use a TOP N clause	SELECT TOP 4 * FROM ITEMS	<pre>/** * @jc:sql statement="SELECT TOP {sql: n} * FROM ITEMS" */ public Item[] getNItems_SQLServerSyntax(int n);</pre>

Related Topics

Parameter Substitution in `@jc:sql` Statements

`@jc:sql` Annotation

Declaring Exceptions in Database Control Methods

The database operations associated with methods in a Database control may result in error conditions. You can choose whether or not to declare that your Database control method throws exceptions. For example, you can declare that a method in a Database control throws the `java.sql.SQLException`, as shown here:

```
/**
 * @jc:sql statement="DROP TABLE customer"
 */
public void dropCustomerTable() throws SQLException;
```

However, if you do so, you are forcing the user of your Database control to explicitly handle the exception by surrounding calls to this method in a try-catch block. For example, to call the `dropCustomerTable` method declared above on an control named `custDB`, your users will have to write code as follows:

```
try {
    custDB.dropCustomerTable()
}
catch (java.sql.SQLException sqle)
{
    // handle error here
}
```

If you choose not to declare that your methods throw exceptions, exceptions that are generated are still thrown, but they are wrapped as `com.bea.control.ControlException` objects beforehand. `ControlException` is a subclass of `java.lang.RuntimeException`, which does not require that calls to methods that throw it to be wrapped in try-catch blocks. `ControlException` can still be caught by users of your Database control, but making it optional can increase convenience for users of your control.

If a `ControlException` (containing a `SQLException`) is thrown by your Database control method and is not handled in the method that calls it, then the calling method returns the exception to its client. In most cases, this is not appropriate; the client code will most likely not be able to take action on the exception because the exception is related to the internal workings of the application. It would be better design to handle the exception in the application and return an appropriate error to the client if necessary.

To learn about handling exceptions in controls, see [Handling Control Method Exceptions](#).

Related Topics

[Database Control](#)

[Creating a New Database Control](#)

Designing a Database Control

This topic provides some tips for designing a Database control to efficiently perform the operations required of it, including designing for user convenience, designing for reuse, providing accurate access rights, compensating for large resultsets, and choosing between objects and primitives.

Designing for User Convenience

When designing a Database control, as with designing any control or application, think carefully about the information needs of the users of your control. Choose operations and data structures that are convenient for the application developer.

One way to accomplish this is to provide Java classes that represent the records or partial rows on which your Database control methods operate on. Consistent use of these classes in the Database control's interface will simplify use of the Database control.

Designing for Reuse

Wherever possible, try to avoid building rigidity into your Database controls.

For example, avoid hard-coding values that are likely to change into queries. It is typically desirable to hard-code the table name because operations are typically table specific. It is typically not desirable to hard-code absolute dates or times into queries. If you provide parameters for methods on your Database control instead of hard-coding them, your control will be more flexible and reusable.

Partitioning the Interface for Access Rights

When designing a new Database control, you may wish to consider creating multiple Database controls for the same database. Interactions with a database typically fall into one of the following categories:

- Administration operations: these typically include CREATE and DROP operations on database tables.
- Privileged operations: these typically include INSERT and possibly UPDATE operations. Many applications restrict which users or applications may modify the contents of the database.
- Retrieval operations: these are typically SELECT operations. Operations that do not modify the contents of the database are typically available to a wider class of users.

If users in your application can be partitioned into these groups, you may wish to create separate Database controls for each class of user. In this way, you can expose only the less privileged operations to web service developers whose services do not require administrative access.

Note that database administration activities are usually carried out as part of application deployment. In a production environment, there is not typically a requirement for an application interface to provide administrative activities such as table creation. The Database control samples provided with WebLogic Workshop include table creation methods for convenience as samples, not as examples of good design.

Planning for Potentially Large Resultsets

When you design database operations in your Database control, be aware of the potential size of the resultsets

that might be generated by the operations. With large databases, it is easy to accidentally execute queries that return resultsets that are larger than the available memory on the machine.

Here are some ways to protect against out-of-memory errors due to large resultsets:

Limit the number of columns returned from a query to only those required. Avoid "SELECT *" statements unless they are explicitly necessary.

Perform filtering in the database. If you are only interested in a subset of the records that might be returned from a query, refine the query so that the database system performs the filtering. Database systems are optimized for this type of operation, and are also designed to perform filtering and sorting operations in a memory-efficient manner.

Use the array-max-length attribute to limit array size. If your Database control method returns an array, you should explicitly provide the array-max-length attribute on the @jc:sql annotation for the method. Set the value so that it is higher than the number of rows you expect to be returned, but not so large that you might run out of memory should that many rows be returned. Note that there is no way to subsequently return the additional rows should the array size limit be reached by a query. There is also no way to set array-max-length using an API call.

The default value of the array-max-length attribute is 1024.

Use the max-rows attribute to limit the size of other data types. If your database control method does not return an array such as an Iterator, RowSet, or ResultSet, you can use the max-rows attribute to limit the size of the data set returned. For techniques for limiting the size of returned data at runtime, see Limiting the Size of Returned Data.

Use an Iterator. A Database control can return a java.util.Iterator instead of an array. The Iterator wraps a java.sql.ResultSet object that accesses the database efficiently. The Iterator and ResultSet objects transparently make occasional requests to the database to obtain more data as needed as you iterate through the resultset. Using this technique, you may enable processing of resultsets that are larger than the available memory.

However, note that ResultSets and Iterator types cannot be returned directly to classes in the web-tier (web services and page flows reside in the web-tier). This is because ResultSets and Iterators will always be closed by the time they reach the web tier. As a workaround for this limitation, you should return custom array types to the web tier (such as Customer[]).

Make calls asynchronous. If a method of a web service initiates a long-running or resource-intensive SQL query, consider making that method asynchronous so that the client does not have to wait for it to return.

To learn more about writing Database control methods that return multiple rows, see Returning Multiple Rows from a Database Control Method.

Consciously Choosing Between Objects and Primitives

As is noted in the topics describing how to return data from Database control methods, the value returned when a database field is NULL depends on the Java type to which the database field is converted. Java primitives such as int and float are not capable of reflecting a null value; they must always have a valid value. So if a database field with no value is converted to an int, its value is 0.

Working with Java Controls

If it is important to you or the clients of your Database control to differentiate between zero and null, you should use the Java wrapper classes for basic types. For example, use `java.lang.Integer` instead of `int`, and `java.lang.Float` instead of `float`.

Related Topics

[Creating a New Database Control](#)

RowSet Control

A RowSet control is a special kind of Database control that is configured to perform CRUD (create, read, update, delete) operations on a specific database table or view using `javax.sql.RowSet` objects. All of the methods of a RowSet control operate directly on or return `javax.sql.RowSet`.

The RowSet Control Wizard

You should usually use the RowSet Control Wizard to create a RowSet control. A RowSet control must be configured in a precise way in order to function properly, with proper coordination between the database table metadata encoded in the control's JCX file and the control's methods.

To learn how to create a RowSet control using the RowSet Control Wizard, see [How Do I: Create a RowSet Control to Access a Database?](#).

Structure of a RowSet Control

When a RowSet control is created via the RowSet Control Wizard, metadata describing the table or view on which the control operates is written to the control's JCX file as an XML Schema. Special annotations on the control's methods reference the schema and constrain the changes that are legal within the RowSet argument for each method.

The code below shows the body of a RowSet control generated for the simple CUSTOMER table used in the Database control sample found in `SamplesApp/WebServices/database/CustomerDBClient.jws`:

```
package customerrowset;

import java.sql.SQLException;
import javax.sql.RowSet;
import com.bea.control.DatabaseControl;
import com.bea.control.DatabaseFilter;
import com.bea.control.ControlExtension;

/**
 * @jc:connection data-source-jndi-name="cgSampleDataSource"
 * @common:schema file="#rowset-schemas" inline="true"
 * @common:define name="rowset-schemas" value::
 *     <xsd:schema targetNamespace="java:///customerrowset" xmlns="java:///customerrowset" xmlns:
 *     <xsd:element name="CUSTOMERRowSet" wld:DefaultNamespace="java:///customerrowset" wld:Row
 *     <xsd:complexType>
 *     <xsd:choice maxOccurs="unbounded">
 *     <xsd:element name="CUSTOMERRow" wld:DatabaseProductName="PointBase">
 *     <xsd:complexType>
 *     <xsd:sequence>
 *     <xsd:element name="CUSTID" type="xsd:int" wld:JDBCType="INTEGER" minOccurs="0" wld
 *     </xsd:element>
 *     <xsd:element name="NAME" type="xsd:string" wld:JDBCType="VARCHAR" minOccurs="0" wld
 *     </xsd:element>
 *     <xsd:element name="ADDRESS" type="xsd:string" wld:JDBCType="VARCHAR" minOccurs="0"
 *     </xsd:element>
 *     ... remaining elements omitted ...
 *     </xsd:sequence>
 *     <xsd:anyAttribute namespace="http://www.bea.com/2002/10/weblogicdata" processConten
 *     </xsd:anyAttribute>
```

Working with Java Controls

```
*      </xsd:complexType>
*      </xsd:element>
*      </xsd:choice>
*      </xsd:complexType>
*      </xsd:element>
*      </xsd:schema>::
*
*/
public interface CustomerRowSet extends ControlExtension, DatabaseControl
{
    /** Disable default Java serialization */
    public final static long serialVersionUID = 1L;

    /**
     * @jc:sql command-type="grid"
     *   rowset-name="CUSTOMERRowSet"
     *   max-rows="1000"
     *   statement::
     *   SELECT CUSTID,NAME,ADDRESS,CITY,STATE,ZIP,AREA_CODE,PHONE FROM WEBLOGIC.CUSTOMER {sql:
     *   ::
     */
    public RowSet getAllCustomer( DatabaseFilter filter )
        throws SQLException;

    /**
     * @jc:sql command-type="detail"
     *   rowset-name="CUSTOMERRowSet"
     *   statement::
     *   SELECT CUSTID,NAME,ADDRESS,CITY,STATE,ZIP,AREA_CODE,PHONE FROM WEBLOGIC.CUSTOMER WHERE
     *   ::
     */
    public RowSet detailsCustomer( Integer x )
        throws SQLException;

    /**
     * @jc:sql command-type="update"
     *   rowset-name="CUSTOMERRowSet"
     */
    public RowSet updateCustomer( RowSet changedRs )
        throws SQLException;

    /**
     * @jc:sql command-type="delete"
     *   rowset-name="CUSTOMERRowSet"
     */
    public void deleteCustomer( RowSet oldRs )
        throws SQLException;

    /**
     * @jc:sql command-type="templateRow"
     *   rowset-name="CUSTOMERRowSet"
     */
    public RowSet detailsCustomerTemplate()
        throws SQLException;

    /**
     * @jc:sql command-type="insert"
     *   rowset-name="CUSTOMERRowSet"
     */
    public RowSet insertCustomer( RowSet changedRs )
        throws SQLException;
}
```



```
/**
 * @jc:sql command-type="insertedRow"
 *   rowset-name="CUSTOMERRowSet"
 *   statement::
 *     SELECT CUSTID,NAME,ADDRESS,CITY,STATE,ZIP,AREA_CODE,PHONE FROM WEBLOGIC.CUSTOMER WHERE
 *     ::
 */
public RowSet getInserted()
    throws SQLException;
}
```

RowSet Metadata

When a RowSet control is created via the RowSet Control Wizard, the metadata for the database table or view on which the control will operate is recorded in the control's JCX file as an XML Schema. The `@common:schema` annotation declares a schema and references the schema definition. The `@common:define` annotation specifies the schema contents for a schema that is expressed inline in a JCX file. The schema defines the structure of the RowSet objects used in the control's methods.

RowSet Control Methods

The set of methods created by the RowSet Control Wizard depends on the choices you make in the wizard. In the preceding example, all operation on the database table were enabled while using the wizard.

Each RowSet control method is annotated with a `@jc:sql` annotation that specifies the `rowset-name` and `command-type` attributes. The `rowset-name` attribute references an element of the RowSet schema defined elsewhere in the file. The `command-type` attribute identifies the type of database operation that the method performs and also constrains the types of changes to a RowSet that each method will accept.

When creating a page flow using a RowSet control, the `command-type` attribute on the RowSet control's method is used by the page flow wizard to select the appropriate control method to invoke for each page flow action.

When each method of the RowSet control is invoked, the underlying Database control code uses the `command-type` attribute type on the method to constrain the types of changes that are allowed in the RowSet. For example, if a method with `command-type="update"` is invoked with a RowSet that contains row insertions, the control will throw an exception.

Related Topics

Database Control

`@common:define` Annotation

`@common:schema` Annotation

`@jc:sql` Annotation

RowSet Controls and SQL Join Queries

Suppose you have two database tables, CUSTOMERS and ORDERS. CUSTOMERS is a unique list of customers. ORDERS records customers' orders. ORDERS contains a foreign key field which associates each order with a record on the CUSTOMERS table. The foreign key field identifies the customer by their customer id number, the primary key of the CUSTOMERS table.

The schemas for these tables are shown below. (The syntax shown is Oracle syntax.)

```
CREATE TABLE CUSTOMERZ
(
    customerz_id NUMBER(10) primary key,
    firstname VARCHAR2(100),
    lastname VARCHAR2(100)
)

CREATE TABLE PARTORDER
(
    partorder_id NUMBER(12) NOT NULL
        CONSTRAINT partorder_pk PRIMARY KEY,
    customerz_id NUMBER(12) NOT NULL,
    sku NUMBER(8),
    CONSTRAINT partorder_customerz_fk
        FOREIGN KEY (customerz_id)
        REFERENCES customerz(customerz_id)
)
```

If you generate a RowSet control from the ORDERS table, the queries that are generated will refer to the customer id number, but not the customer name, or any other data from the CUSTOMERS table.

```
/**
 * @jc:sql command-type="grid"
 *   rowset-name="PARTORDERRowSet"
 *   max-rows="1000"
 *   statement::
 *   SELECT PARTORDER_ID,CUSTOMERZ_ID,SKU FROM DEMO.PARTORDER {sql: filter.getWhereClause (
 *   ::
 */
public RowSet getAllPartorder(DatabaseFilter filter)
    throws SQLException;
```

But how can you edit the RowSet control to return the customer name and other customer data? Below are general rules for editing an auto generated RowSet control to accommodate join queries.

Step 1: Add additional methods for the join queries.

For example, suppose you want the following join query included in your RowSet control.

```
SELECT O.PARTORDER_ID,O.CUSTOMERZ_ID,C.FIRSTNAME,C.LASTNAME,O.SKU FROM customerz C, partorder O
```

You could add the query by adding the following method to the RowSet control.

```
/**
 * @jc:sql command-type="grid"
 *   rowset-name="PARTORDERRowSet"
 *   max-rows="1000"
 *   statement::
 *     SELECT O.PARTORDER_ID,O.CUSTOMERZ_ID,C.FIRSTNAME,C.LASTNAME,O.SKU FROM customerz C, pa
 *     ::
 */
public RowSet getAllCustomerorders(DatabaseFilter filter)
    throws SQLException;
```

Note: If you plan to generate a page flow from your RowSet control, you should keep the column list in the @jc:sql annotation on one line. This will ensure that the page flow wizard can read your control methods correctly.

Step 2: Edit the RowSet–schema annotation to add columns mentioned in the join queries

Each RowSet control contains an RowSet–schema annotation that controls the data returned from the database. For example, the following RowSet–schema is generated when you autogenerate a RowSet control for the ORDERS table.

```
/**
 * @jc:connection data-source-jndi-name="oradev_source"
 * @common:schema file="#rowset-schemas" inline="true"
 * @common:define name="rowset-schemas" value::
 *   <xsd:schema targetNamespace="java:///database.oracle.relationalDB.rowset" xmlns="java:///
 *   <xsd:element name="PARTORDERRowSet" wld:DefaultNamespace="java:///database.oracle.relati
 *   <xsd:complexType>
 *   <xsd:choice maxOccurs="unbounded">
 *   <xsd:element name="PARTORDERRow" wld:DatabaseProductName="Oracle">
 *   <xsd:complexType>
 *   <xsd:sequence>
 *   <xsd:element name="PARTORDER_ID" type="xsd:integer" wld:JDBCType="NUMERIC" minOccurs
 *   </xsd:element>
 *   <xsd:element name="CUSTOMERZ_ID" type="xsd:integer" wld:JDBCType="NUMERIC" minOccurs
 *   </xsd:element>
 *   <xsd:element name="SKU" type="xsd:integer" wld:JDBCType="NUMERIC" minOccurs="0" wld
 *   </xsd:element>
 *   </xsd:sequence>
 *   <xsd:anyAttribute namespace="http://www.bea.com/2002/10/weblogicdata" processContent
 *   </xsd:anyAttribute>
 *   </xsd:complexType>
 *   </xsd:element>
 *   </xsd:choice>
 *   </xsd:complexType>
 *   </xsd:element>
 *   </xsd:schema>::
 *
 */
public interface OrdersRowSet extends ControlExtension, DatabaseControl
```

When you add columns to a RowSet–schema, keep the following points in mind:

- Make sure the columns are in the right position in the column list. The rowset schema can contain a superset of the columns in the queries' select list, but they must be in the correct relative order.

Working with Java Controls

- For the joined in field(s) make sure the following properties are set:

```
type=[schema type]
minOccurs="0"
wld:JDBCType=[JDBC type of the source database column]
wld:ReadOnly="true"
wld:TableName="[name of joined in table]"
```

You must set the attribute `wld:ReadOnly` to `true`, because only one table is updatable through a given `RowSet` control.

For the sample join query above, add the columns shown in red below.

```
/**
 * @jc:connection data-source-jndi-name="oradev_source2"
 * @common:schema file="#rowset-schemas" inline="true"
 * @common:define name="rowset-schemas" value::
 *   <xsd:schema targetNamespace="java:///database.oracle.relationalDB.rowset" xmlns="java:///
 *   <xsd:element name="PARTORDERRowSet" wld:DefaultNamespace="java:///database.oracle.relati
 *   <xsd:complexType>
 *     <xsd:choice maxOccurs="unbounded">
 *       <xsd:element name="PARTORDERRow" wld:DatabaseProductName="Oracle">
 *         <xsd:complexType>
 *           <xsd:sequence>
 *             <xsd:element name="PARTORDER_ID" type="xsd:integer" wld:JDBCType="NUMERIC" minOccu
 *             </xsd:element>
 *             <xsd:element name="CUSTOMERZ_ID" type="xsd:integer" wld:JDBCType="NUMERIC" minOccu
 *             </xsd:element>
 *             <xsd:element name="FIRSTNAME" type="xsd:string" wld:readOnly="true" wld:JDBCType="
 *             </xsd:element>
 *             <xsd:element name="LASTNAME" type="xsd:string" wld:readOnly="true" wld:JDBCType="V
 *             </xsd:element>
 *             <xsd:element name="SKU" type="xsd:integer" wld:JDBCType="NUMERIC" minOccurs="0" wl
 *             </xsd:element>
 *           </xsd:sequence>
 *           <xsd:anyAttribute namespace="http://www.bea.com/2002/10/weblogicdata" processConten
 *           </xsd:anyAttribute>
 *         </xsd:complexType>
 *       </xsd:element>
 *     </xsd:choice>
 *   </xsd:complexType>
 * </xsd:element>
 * </xsd:schema>::
 */
public interface OrdersRowSet extends ControlExtension, DatabaseControl
```

Related Topics

Parameter Substitution in `@jc:sql` Statements

Stored Functions

Stored Procedures

`@jc:sql` Annotation

RowSet Control

Mapping Database Field Types to Java Types in the Database Control

When you are submitting or requesting data from a database using the Database control, you may need to understand how Java data types map to SQL data types, and vice versa, in order to handle the data in your code. The table below shows the most common data type mappings, although others are possible.

In some cases, there are multiple mapping possibilities for a single type. For example, if you return data from a column of type `smallint` in the Pointbase sample database, you can store that data in a Java variable of type `byte` or type `short`, depending on your needs.

The type mappings shown in the tables below are the most direct mappings and are shown as examples. Please consult the documentation for your database system to determine all possible database to Java mappings in the database system you are using.

Type Mappings for PointBase

The following table lists the relationships between database types and Java types for the PointBase Version 4.4 database, which is installed with WebLogic Server. If you are using a different database, please consult the documentation for the JDBC driver of your database software.

Java Data Types	JDBC Data Types	PointBase SQL Data Types (Version 4.4)
<code>boolean</code>	BIT	<code>boolean</code>
<code>byte</code>	TINYINT	<code>smallint</code>
<code>short</code>	SMALLINT	<code>smallint</code>
<code>int</code>	INTEGER	<code>integer</code>
<code>long</code>	BIGINT	<code>numeric/decimal</code>
<code>double</code>	FLOAT	<code>real</code>
<code>float</code>	REAL	<code>float</code>
<code>double</code>	DOUBLE	<code>double</code>
<code>java.math.BigDecimal</code>	NUMERIC	<code>numeric</code>
<code>java.math.BigDecimal</code>	DECIMAL	<code>decimal</code>
<code>String</code>	CHAR	<code>char</code>
<code>String</code>	VARCHAR	<code>varchar</code>
<code>String</code>	LONGVARCHAR	<code>clob</code>
<code>java.sql.Date</code>	DATE	<code>date</code>
<code>java.sql.Time</code>	TIME	<code>time</code>
<code>java.sql.Timestamp</code>	TIMESTAMP	<code>timestamp</code>
<code>byte[]</code>	BINARY	<code>blob</code>
<code>byte[]</code>	VARBINARY	<code>blob</code>
<code>byte[]</code>	LONGVARBINARY	<code>blob</code>
<code>java.sql.Blob</code>	BLOB	<code>blob</code>
<code>java.sql.Clob</code>	CLOB	<code>clob</code>

Type Mappings for Microsoft SQL Server

For table showing the relationships between JDBC types and Microsoft SQL Server types, see Data Types in the WebLogic Server 8.1 documentation.

Type Mappings for Oracle 8i

The following table lists the relationships between database types and Java types for the Oracle 8i database.

Java Data Types	JDBC Data Types	Oracle SQL Data Types (Version 8i)
boolean	BIT	NUMBER
byte	TINYINT	NUMBER
short	SMALLINT	NUMBER
int	INTEGER	NUMBER
long	BIGINT	NUMBER
double	FLOAT	NUMBER
float	REAL	NUMBER
double	DOUBLE	NUMBER
java.math.BigDecimal	NUMERIC	NUMBER
java.math.BigDecimal	DECIMAL	NUMBER
String	CHAR	CHAR
String	VARCHAR	VARCHAR2
String	LONGVARCHAR	LONG
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP	DATE
byte[]	BINARY	NUMBER
byte[]	VARBINARY	RAW
byte[]	LONGVARBINARY	LONGRAW
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB

Related Topics

[Returning a Single Value from a Database Control Method](#)

[Returning a Single Row from a Database Control Method](#)

[Returning Multiple Rows from a Database Control Method](#)

[Parameter Substitution in @jc:sql Statements](#)

Configuring a Data Source

When you create a Database control, you associate the control with a database by specifying a data source for the control's `data-source-jndi-name` property. The data source is defined by WebLogic Server, along with the connection pool that manages the connection to the database. To set up a new data source in WebLogic Server, you can use the WebLogic Server console, or you can use the Data Source Viewer that is part of WebLogic Workshop.

This topic describes how to configure a data source using the Data Source Viewer. For information on using the WebLogic Server console, see [How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle?](#)

To manage connection pools and data sources with the Data Source Viewer:

1. To display the Data Source Viewer, start WebLogic Server, and choose **Tools—>WebLogic Server—>DataSource Viewer**.
2. Select a data source from the navigation tree on the left. You can examine its properties or delete it by clicking the **Drop Data Source** button.
3. Select a connection pool from the navigation tree on the left. You can examine its properties or delete it by clicking the **Drop Pool** button.

To create a new data source:

1. In the Data Source Viewer, click the **New Data Source** button.
2. Enter a unique name for the new data source.
3. Select the connection pool to use from the **Pool** drop-down list.

To create a new connection pool:

1. In the Data Source Viewer, click the **New Data Source** button.
2. From the **Pool** drop-down list, select **<Create New Pool>**.
3. Specify properties for the new connection pool, as follows:
 - ◆ **Pool Name:** A unique name that will be the JNDI name for this connection pool.
 - ◆ **Driver:** The driver to use for the database you are connecting to.
 - ◆ **URL:** The URL that identifies the database to connect to.
 - ◆ **User:** The user name.
 - ◆ **Password:** The user password.
 - ◆ **Properties:** Any custom properties to specify for the connection.

Related Topics

[How Do I: Connect a Database Control to a Database Such as SQL Server or Oracle?](#)

Web Service Control

A Web Service control makes it easy to access an external web service from a WebLogic Workshop application. You can create a Web Service control for any web service that publishes a WSDL (Web Service Definition Language) file. A WSDL file describes the methods and callbacks that a web service implements, including method names, parameters, and return types. It also describes the protocols that a web service supports. To learn more about WSDL files, see [WSDL Files: Web Service Descriptions](#).

Note: You should not use a Web Service control to invoke a web service that resides in the same application. Invoking a web service via a Web Service control means marshalling the method parameters into a SOAP message on the calling end and unmarshalling the SOAP message on the receiving end, then again for the method return value. This is very inefficient when the invocation is local. You would usually be tempted to invoke one web service from another if the called web service included business logic you want to access from the calling web service.

In general, you should place business logic in custom Java controls instead of in web services. This allows you to access the business logic from various contexts in the application (web services, other controls, page flows) without incurring the cost of data marshalling and unmarshalling. Web Service controls should *only* be used to invoke web services that are truly external to your application.

Topics Included in this Section

Overview: Service Controls

Introduces public contracts and other aspects of Service controls.

Creating a New Web Service Control

Explains how to create a new Web Service control from a JWS or WSDL file, and how to add an existing Web Service control.

Declaring Compliance with a WSDL File

Describes how to use the `@jc:wsl` annotation to declare compliance with the public contract of a WSDL file.

Specifying the Default Protocol and Message Format

Explains how to use the WSDL file of a target web service to determine the protocol and message formats it supports.

Specifying Conversation Shape

Explains why it is recommended that you do *not* modify the conversation shape of a Web Service control.

Buffering Methods and Callbacks

Describes how to add buffers to your Web Service control so that calls made to the control are placed in a queue.

Defining Java-to-XML Translation with XML Maps

Web Service Control

Explains how to use an XML map to modify the relationship between a Java interface and the XML messages sent to the target web service.

Samples that Use Web Service controls

[AdvancedTimer.jws Sample](#)

[Conversation.jws Sample](#)

[CreditReport.jws Sample](#)

Samples that Customize Web Service controls

[QuoteClient.jws Sample](#)

[Related Topics](#)

[Applications and Projects](#)

[JCX Files:Implementing Controls](#)

[The ServiceControl Interface](#)

Overview: Web Service Controls

A Web Service control provides an interface between your application and a web service, which allows your application to invoke the methods and handle the callbacks of that web service. Using a Web Service control, you can connect to any web service for which a WSDL file is available, whether or not it was built using WebLogic Workshop. To learn more about WSDL files, see [WSDL File: Web Service Descriptions](#).

In order to use Web Service controls, it may help you to understand several concepts. This topic provides an overview of some of these concepts.

Understanding Public Contracts

Web services define and expose a public contract, which is typically expressed in a WSDL file. A public contract describes two things: the operations that the web service can perform and the format of the messages sent to the service to access its operations and receive operation results. The contract is completely under the control of the author of the web service; it cannot be altered by a client of the web service.

The public contract for a web service developed with WebLogic Workshop is the collection of all methods marked with the `@common:operation` annotation plus all members of the Callback interface. Each public contract is completely defined in the JWS file for the web service. When you generate a WSDL file from a JWS file, the public contract is expressed according to the WSDL standard.

The Web Service control cannot violate or modify the public contract of the web service it represents. This restricts the type of changes you can make to a Web Service control. For example, you can't modify the Service control to use a communication protocol that the target web service doesn't understand.

Understanding Web Service Controls: Proxies for Web Services

In WebLogic Workshop, a Web Service control serves as an intermediary, or proxy, for a web service. When web service X wants to invoke an operation of web service Y, web service X calls a Java method of the Web Service control for Y. The Web Service control converts the Java method invocation into an appropriate message to send to the web service Y, and it communicates with web service Y using a protocol that service Y can understand. The Web Service control also converts returned messages from service Y back into Java method invocations on service.

In these ways, the Web Service control allows service X to use service Y merely by implementing application-level Java code. As the author of service X, you do not need to know the details of message formats or protocols.

Managing Shared Web Service Controls

When modifying Web Service controls, keep in mind that a Web Service control may be shared by multiple web services. If all of the web services using a Service control are referring to a single JCX file, changing the behavior of that JCX file affects the behavior of the Web Service control for all of the referring web services.

Related Topics

[Web Service Control](#)

Declaring Compliance with a WSDL File

Creating a New Web Service Control

A Web Service control makes it easy to access a web service from your application. You create a new Web Service control to access an existing web service. The existing web service is referred to here as the target web service.

You can create a Web Service control for a target web service if that web service publishes a WSDL file. If the target web service was developed with WebLogic Workshop, you can create a Web Service control directly from the web service's JWS file.

Creating a Web Service Control from a JWS File

This procedure describes how to create a Web Service control if you have a WebLogic Workshop web service (a JWS file) and want other WebLogic Workshop web services to be able use your web service.

1. Ensure that the JWS file for the web service is in your project.
2. Browse to the JWS file in the *Application pane*.
3. Right-click on the JWS file in the *Application pane* and select *Generate JCX File*.

The resulting JCX file is a Web Service control. Note that WebLogic Workshop adds the word "Control" to the end of the name of generated controls. If you generate a JCX file from the web service HelloWorld.jws, the JCX file is named HelloWorldControl.jcx.

When you generate a JCX file from a JWS file in this manner, the JCX file is automatically linked to its parent JWS file. When you modify the web service, the JCX file is automatically regenerated to reflect the change. For example, if you add a method to the web service, an associated JCX file is automatically regenerated, and the new method is available to every web service that employs that JCX file as a Web Service control.

If you modify the source code for an autogenerated JCX file, when you attempt to save the file WebLogic Workshop warns you that continuing the save operation will turn off autogeneration for this file. If you proceed, the JCX file will no longer be linked to its parent web service. In general, you probably want to avoid turning off autogeneration for a JCX file.

Creating a Web Service Control from a Local WSDL File

This procedure describes how to create a Web Service control if you have a WSDL file for the target web service.

1. Ensure that the WSDL file for the web service is in your project.
2. Browse to the WSDL file in the Application pane.
3. Right-click on the WSDL file in the Application pane and select *Generate JCX*.

Adding an Existing Web Service Control

You can add a Web Service control in any of the following types of files:

- Java Control (JCS file)
- Java Page Flow (JPF file)

Working with Java Controls

- Java Server Page (JSP file)
- Java Web Service file (JWS file)
- Process file (JPD file)

If a Web Service control already exists for another web service that you would like to use from your application, you only need to add an instance of the Web Service control to your application. To add a Web Service control, follow these steps:

1. Open the file to which you want to add the control in the WebLogic Workshop design environment.
2. Display the Data Palette (**View**→**Windows**→**Data Palette**).
3. On the Data Palette, click the **Add** drop-down, and choose **Web Service**. The **Insert Control – Insert Web Service** dialog opens.
4. In the **Variable name for this control** field, type the variable name used to access the new Web Service control instance from your application.
5. Choose the **Use a Web Service control already defined by a JCX file** radio button.
6. If you know the name and filepath of the Web Service control you want to add, enter them in the **JCX file** field. If you do not know the name and path, click **Browse**. The **Select** dialog appears.
7. Navigate to the Web Service control you want to add and click **Add**.
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. Click **Create**.

You can also create a new Web Service control from the **Insert Control – Insert Web Service** dialog by indicating that you want to create a new Web Service control and then specifying the location of the WSDL file on which the Web Service control should be based. The WSDL location may be local (within your WebLogic Workshop application) or remote.

Related Topics

Overview: Web Service Controls

QuoteClient.jws Sample

Declaring Compliance with a WSDL File

A WSDL (Web Services Definition Language) file is a public contract that a web service publishes so that a client knows how to call it. A Web Service control typically declares explicitly that it complies with the public contract expressed in a WSDL file. This compliance is declared with the `@jc:wsdl` annotation. The `@jc:wsdl` annotation's `file` attribute names the WSDL file with which the Web Service control complies.

The `@jc:wsdl` annotation may refer to a file that is inline within the Web Service control. The WSDL contents are specified as the value of a `@common:define` annotation. The `@jc:wsdl` annotation's `file` attribute then refers to `#Name` instead of an actual file name, where `Name` is the value of the `@common:define` annotation's `name` attribute. The following code sample illustrates this arrangement:

```
import com.bea.control.ServiceControl;
/**
 * @jc:location http-url="creditreport/IRS.jws" jms-url="creditreport/IRS.jws"
 * @jc:wsdl file="#IRSWsdl"
 */
public interface IRSControl extends ServiceControl
{
    ...
}
/**
 @common:define name="IRSWsdl" value::
    <?xml version=1.0 encoding=utf-8?>
    <definitions ...>
        ...remainder of WSDL here...
    </definitions>
 ::
 */
```

If a Web Service control has an associated `@jc:wsdl` annotation, the methods in the Web Service control's interface must be capable of producing messages that comply with the operations defined in the WSDL file. Note, however, that this does not guarantee that the signature of a method in a Web Service control is identical to the signature of a method in the JWS file from which it was generated, in the case of an autogenerated Web Service control. The Web Service control must only adhere to the public contract defined by the WSDL, meaning that it can produce WSDL-compliant messages; because the WSDL makes no requirements about implementation, the Web Service control may construct those messages differently than the web service does.

To learn more about the `@jc:wsdl` annotation, see `@jws:wsdl` annotation, which is used within the web service on which an autogenerated Web Service control is based.

Related Topics

[@jws:wsdl Annotation](#)

[JCX Files: Implementing Controls](#)

[WSDL Files: Web Service Descriptions](#)

[@common:define Annotation](#)

Specifying the Default Protocol and Message Format

When you modify a Web Service control to use a different protocol or message format, you must choose a protocol or message format that the target web service supports. You can determine these protocols and message formats by examining the target service's WSDL file.

Setting a Web Service Control's Default Protocol

A protocol is a low-level network scheme for passing messages between systems. To better understand protocols, imagine that a message is a conventional letter and the message format defines the possible content and interpretation of the letter. The protocol would describe the nature of an envelope and how to print an address on an envelope.

The standards on which web services are based allow a web service to support a variety of protocols and message formats. For example, a web service may support communication via HTTP (Hyper Text Transport Protocol, the basic protocol of web browsers and servers) and/or JMS (Java Message Service, a J2EE protocol used to pass messages). It may support other protocols as well.

Although a web service may support multiple protocols or message formats, when WebLogic Workshop creates a Web Service control for an application, one protocol and one message format are chosen as the default.

You can modify the protocol that a Web Service control uses to communicate with its target web service using the `@jc:protocol` annotation. This annotation specifies which protocols and message formats a web service can accept or a Web Service control will send to the service it represents.

To learn more about specifying protocols, see `@jc:protocol` annotation.

Choosing a Message Format

The standards that make web services possible allow for multiple formats of the messages that pass between applications. The most frequently used standard for web services messages is SOAP. SOAP defines two formats for messages that represent method invocations: Document Literal and SOAP Remote Procedure Call (RPC).

WebLogic Workshop uses Document Literal as its default message format, but is capable of communicating using SOAP RPC, which other development tools may use as their default message format. When you wish to implement communications between two applications developed using different tools, you need to ensure that both are using the same message format. You can do this in two ways. You can specify what format your application expects and dictate that choice to the developer of the other web service, or you can select an alternative format that allows your application to accept the other web service's default format.

To specify the message format that a web service uses to communicate with the Web Service control, you can set the `soap-style` attribute of the `@jc:protocol` annotation to `document` for the Document Literal message format or `rpc` for the SOAP RPC format.

Specifying Per–Service and Per–Method Settings

You can specify protocols and message formats on a Web Service control or on a method of a Web Service control. The settings on the Web Service control apply to all methods and callbacks by default. If a method or callback specifies different protocol or message format settings, those settings override the default settings from the Web Service control.

Related Topics

[@jc:protocol Annotation](#)

Specifying Conversation Shape

The conversation shape of a Web Service control is the specific configuration of conversation phases for the service's various methods or callbacks. If you change the conversation phase of any method or callback, you have changed the conversation shape of the service.

Modifying the conversational shape of a Web Service control is not recommended. It is easy to inadvertently introduce incompatibilities with the target application's public contract.

In particular, WebLogic Workshop web services that use conversations expect special SOAP headers containing conversation parameters to be included in every conversational message. If the target web service is not capable of understanding WebLogic Workshop conversation SOAP headers, including them in messages by adding `@jc:conversation` annotations to the Web Service control's methods will violate the web service's contract and prevent successful communication with the web service.

However, if you do choose to change the conversational shape of a Service control, you can do so by adding `@jc:conversation` annotations to individual operations in the Web Service control.

To learn more about conversations, see [Designing Conversational Web Services](#).

To learn more about the `@jc:conversation` annotation, see `@jws:conversation` annotation, which exposes the same functionality.

Related Topics

[Designing Conversational Web Services](#)

[Implementing Conversations](#)

Buffering Methods and Callbacks

You can add a buffer to a method of your Web Service control to ensure that your application need not wait for the Web Service control before processing other requests. Calls to a buffered method are queued so that the server is not overwhelmed with requests. This topic describes how buffers aid communication between applications and services, and explains how to add a buffer to a method or callback. It also provides an example of code containing buffers.

Using Buffers

If you add a message buffer to a Web Service control's method, outgoing messages (invocations of the method) are buffered on the local machine and the method invocation returns void immediately. This prevents your application from having to wait for a response. In other words, your application doesn't have to wait for the network roundtrip to occur.

If you add a message buffer to a Web Service control's callback, incoming messages (invocations of the callback) are buffered on the local machine. The callback invocation returns to the target web service immediately. This prevents the target service from waiting until your application processes the request. Note that since the buffering occurs on your end of the wire, the target service must wait for the network roundtrip even though it will return a void result. But the target service does not have to wait for your application to process the message.

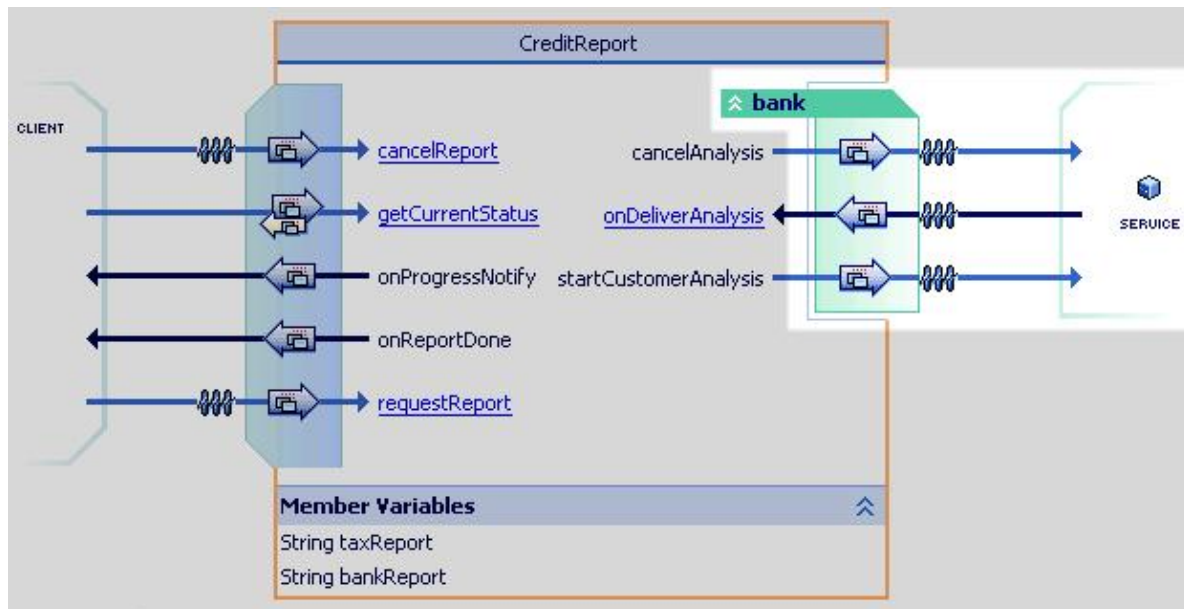
Adding a Buffer to a Method or Callback

You can add message buffers to the methods and callback handlers of any Service control provided it has a void return type. Remember, though, that a Service control is a *proxy* for a target web service. The target service is often located on a remote server, so even if you add message buffers to a Web Service control where this is the case, the buffering always occurs on the local server.

1. To add a buffer to a method or callback handler:
2. If you are not in Design View, click the **Design View** tab.
3. Select the method or callback handler you want to buffer.
4. In the **Property Editor** pane, expand the message-buffer property.
5. Set the enabled attribute to **true**.

Design View displays the message buffer with a "spring" icon on the method or callback, as shown here:

Working with Java Controls



Note that WebLogic Workshop draws message buffers on the "near" end of the wire. In general, you have no control over the execution environment or configuration of remote web services. In other words, you can't change what happens on the "far" end of the wire.

Adding a message buffer to a method makes the method asynchronous, meaning that callers to that method do not wait for a response.

In the case of a Web Service control, your application is the client when sending outgoing messages (invocations of the method) and the target service is the client when sending incoming messages (control callbacks) to your service.

How Message Buffers are Specified in Code

Message buffers are specified in code using the `@common:message-buffer` annotation. In a Web Service control, you can place the `@common:message-buffer` annotation on a method or a callback.

To learn about the `@common:message-buffer` annotation, see `@common:message-buffer` annotation.

The following example contains a `@common:message-buffer` annotation on a Web Service control method:

```
public interface QuoteServiceControl extends ServiceControl
{
    ...
    /**
     * @common:message-buffer enable="true"
     */
    public void getQuote (int customerID, java.lang.String tickerSymbol);
    ...
}
```

The following example contains a `@common:message-buffer` annotation on a Web Service control callback:

```
public interface QuoteServiceControl extends ServiceControl
{
```

Working with Java Controls

```
...
public interface Callback
{
    /**
     * @common:message-buffer enable="true"
     */
    public void onQuoteReady (java.lang.String tickerSymbol, double dQuote);
}
...
}
```

Related Topics

[Designing Asynchronous Interfaces](#)

[Using Buffering to Create Asynchronous Methods and Callbacks](#)

Defining Java to XML Translation with XML Maps

You can use an XML map to modify the relationship between the Java interface exposed to clients of a Web Service control and the XML messages sent to the target web service. You may want to do this if, for example, you want to simplify the methods that the Web Service control exposes, or perform some additional processing before the request is sent to the web service represented by the Web Service control. In all cases, the request that you send to the target web service must always adhere to the public contract outlined in the service's WSDL file.

The following example, which demonstrates that kind of modification, is taken from the QuoteServiceControl.jcx sample Web Service control used by the QuoteClient.jws sample. QuoteServiceControl.jcx was originally generated from QuoteService.jws and then modified. The first parameter of getQuote was removed from the Java signature, meaning callers (that is, web services that use this Web Service control) are no longer expected to pass it. An XML map was then added to getQuote, with the value of the <customerID> element hard-coded. This leaves the message shape as QuoteService expects it, with two parameters. The calling service passes one parameter that the Web Service control combines with the hard-coded parameter to produce the two-parameter message the target service expects.

The following is the code as it was originally generated from QuoteService.jws. Notice that getQuote takes two parameters.

```
import com.bea.control.ServiceControl;
/**
 * @jc:location http-url="QuoteService.jws" jms-url="QuoteService.jws"
 * @jc:wSDL file="#QuoteServiceWSDL"
 */
public interface QuoteServiceControl extends ServiceControl
{
    public interface Callback
    {
        /**
         * @jc:conversation phase="finish"
         */
        public void onQuoteReady (java.lang.String tickerSymbol, double dQuote);
    }
    /**
     * @jc:conversation phase="start"
     */
    public void getQuote (int customerID String tickerSymbol);
}
```

Below is the code after the JCX file has been modified manually. The customerID parameter has been removed from the method signature and hard-coded into the XML map, which was also manually added. Note that the XML map must comply with the contract of the target web service.

```
import com.bea.control.ServiceControl;
/**
 * @jc:location http-url="QuoteService.jws" jms-url="QuoteService.jws"
 * @jc:wSDL file="#QuoteServiceWSDL"
 */
public interface QuoteServiceControl extends ServiceControl
{
    public interface Callback
    {
        /**
```

Working with Java Controls

```
        * @jc:conversation phase="finish"
        */
    public void onQuoteReady (java.lang.String tickerSymbol, double dQuote);
}
/**
 * @jc:conversation phase="start"
 * @jc:parameter-xml xml-map::
 * <getQuote xmlns="http://www.openuri.org/">
 *     <customerID>1234567890</customerID>
 *     <tickerSymbol>{tickerSymbol}</tickerSymbol>
 * </getQuote>::
 */
    public void getQuote (String tickerSymbol);
}
```

Note that you can also make the modifications to the method signature and XML map in the Edit Maps and Interface dialog. Open the dialog by double-clicking on the map icon (the fat arrow) for the `getQuote` method when you are editing `QuoteClient.jws` in Design View.

Related Topics

Handling and Shaping XML Messages with XML Maps

QuoteClient.jws Sample

EJB Control

Enterprise JavaBeans (EJBs) are Java software components of enterprise applications. The Java 2 Enterprise Edition (J2EE) specification defines the types and capabilities of EJBs as well as the environment (or container) in which EJBs are deployed and executed. From a software developer's point of view, there are two aspects to EJBs: first, the development and deployment of EJBs; and second, the use of existing EJBs from client software. The EJB control makes it easy to use an existing, deployed EJB from your application.

Topics Included in this Section

Overview: Enterprise JavaBeans and EJB Controls

Describes Enterprise JavaBeans and their relationship to EJB controls.

Creating a New EJB Control

Describes how to create and configure an EJB control.

Using an EJB Control

Describes how to use an existing EJB control from within a web service.

Handling EJB Exceptions

Describes how to handle exceptions that might be thrown by an EJB.

Related Topics

Using WebLogic Built-In Controls

Developing Enterprise JavaBeans

Overview: Enterprise JavaBeans and EJB Controls

To access the capabilities of an Enterprise JavaBean (EJB) without an EJB control, you must perform several preparatory operations. You must look up the EJB in the JNDI registry, obtain the EJB's home interface, obtain an EJB instance, and then finally invoke methods on the EJB's remote interface to perform tasks.

The EJB control relieves you of all of this preparatory work. Once you have created the EJB control, a web service or page flow can use the control to access the EJB's business methods directly. The EJB control manages communication with the EJB for you, including all JNDI lookup, interface discovery and EJB instance creation and management.

In short, EJB controls provide an alternative approach that makes it easy for you to use an existing, deployed EJB from within an application. EJB controls supports interaction with two of the three types of EJBs, that is, session beans and entity beans. The EJB control does not support direct communication with message-driven EJBs.

Note. You can send requests for messages indirectly to message-driven EJBs using the JMS control instead. However, unlike the EJB control, the JMS control is not used to locate and reference an existing message-driven EJB. For more information, see JMS Control.

A short description of session and entity beans is provided below. To learn more about message-driven beans, J2EE, and EJBs, see *Getting Started with EJB Project*, or consult the J2EE programming book of your choice.

Session EJBs

A session EJB is used to execute business tasks for a client on the application server. Stateful session beans maintain conversational state when engaged by a client. That is, conversational state is used to keep track of data between method invocations and to ensure that the bean responds to the correct client. Stateless session beans do not use conversational state and the contract with a client only lasts for the duration of the method invocation. A session EJB is not persistent, so when the client terminates, its session EJB disconnects and is no longer associated with the client.

Entity EJBs

An entity EJB represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. The persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. Unlike session beans, entity beans are persistent, allow shared access, have primary keys, and may participate in relationships with other entity beans.

EJB Interfaces

EJB 2.0 exposes four types of interfaces, called the local home interface, the local business interface (or simply, the local interface), the remote home interface, and the remote business interface (or simply, the remote interface). Client applications can obtain an instance of the EJB with which to communicate by using the remote home interface. The methods in the remote home interface are limited to those that create or find EJB instances. Once a client has an EJB instance, it can invoke methods of the EJB's remote business interface to do real work. The business interface directly accesses the business logic encapsulated in the EJB. Interactions between EJBs defined in the same WebLogic Workshop application, as well as interactions

Working with Java Controls

between EJBs and web services or page flows in the same WebLogic Workshop application, can use the local interfaces instead, which provides a performance advantage over remote interfaces. In other words, the local home and business interfaces define the methods that can be accessed by other beans, EJB controls, web services, and page flows in the same WebLogic Workshop application, while the remote home and business interfaces define the methods that can be accessed by other applications.

To create an EJB control to represent an EJB, you must know the names of the home and business interfaces. The name for the home interface is typically of the form `com.mycompany.MyBeanNameHome` or `com.mycompany.MyBeanNameLocalHome`, and the business interface is typically of the form `com.mycompany.MyBeanName` or `com.mycompany.MyBeanNameLocal`. The EJB control uses either the EJB's local interfaces or the remote interfaces. For more information about making an EJB control, see [Creating a New EJB Control](#). To learn more about how EJB controls interact with session and entity EJBs, see [Using an EJB Control](#).

Related Topics

[Getting Started with EJB Project](#)

[EJB Control](#)

[Using WebLogic Built-In Controls](#)

[Designing Conversational Web Services](#)

Creating a New EJB Control

Enterprise JavaBean (EJB) controls make it easy for you to use an existing, deployed session or entity EJB from within an application. To create an EJB control, you must first make sure that the EJB's (local or remote) home and business interfaces are available to your application.

Making EJB Interfaces Available to Your Application

Before you can create an EJB Control, the EJB's local or remote interfaces must be known in your application. If the EJB is not part of the application, you make it available by adding the EJB's JAR file to your WebLogic Workshop application. While the complete EJB JAR file allows WebLogic Workshop to access the EJB's interfaces, the only classes actually required are the EJB's home and remote interface classes and any other classes used externally by the EJB (for example, as method parameters or method return types). The EJB compiler ejbc is capable of producing a client JAR that will serve this purpose. To learn more about EJB client JAR files, see @ejbgen:jar-settings Annotation. To learn more about adding EJB Jar files, see How Do I: Add an Existing Enterprise JavaBean to an Application?

Creating a New EJB Control

You can create a new EJB control and insert it into your page flow or web service application at the same time, or you can create a new EJB control without insertion, for instance when you are developing in a Control project.

You can add a Web Service control in any of the following types of files:

- Java Control (JCS file)
- Java Page Flow (JPF file)
- Java Server Page (JSP file)
- Java Web Service file (JWS file)
- Process file (JPD file)

To insert a new EJB control

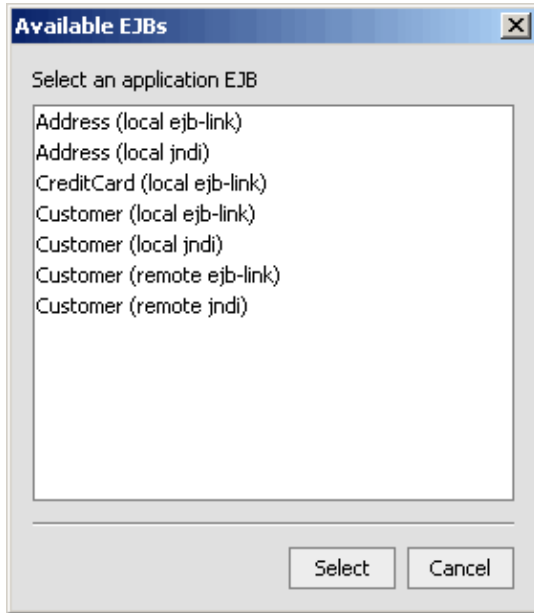
1. Make sure you have opened the target web service or page flow controller file. For web services, if you are not in Design View, click the Design View tab. For page flows, click the Action View tab.
2. Display the Data Palette (**View**→**Windows**→**Data Palette**).
3. On the Data Palette, click the **Add** drop-down, and choose **Web Service**. The **Insert Control – Insert EJB Control** dialog opens.
4. In the **Variable name for this control** field, type the name for your EJB control.
5. Select the **Create a new EJB control to use** radio button.
6. In the **New JCX name** field, type 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 runtime** check box. For more information about control factories, see Control Factories: Managing Collections of Controls.
8. In the Step 3 pane, click the **Browse application EJBs** or **Browse server EJBs** button. The **Available EJBs** dialog appears.

The **Browse application EJBs** button will return a list of all the EJBs known within the current application. These are EJBs developed in the application as well as EJBs defined in EJB JARs added

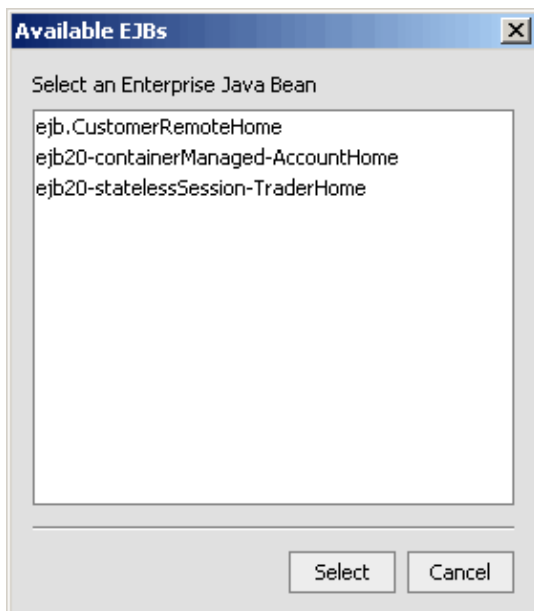
Working with Java Controls

as modules to the application. If only the EJB's local interface is defined, the EJB will appear in the list with a local ejb-link reference, as is shown in the below screenshot for the CreditCard bean. If an EJB's local interface and local JNDI name are defined, the EJB will also appear with a local jndi reference, as is shown below for the Address bean. The same analogy applies to remote interfaces. The Customer bean below, whose remote and local interfaces and JNDI names are defined, appears four times.

An ejb-link locates a bean relative to its EJB JAR within the application. In source code, this might look like `@jc:ejb ejb-link="MyEJBProject.jar#Music"`. For more information, see the `@jc:ejb` reference documentation.



The **Browse server EJBs** button will return a list of all the EJBs known on the server used by the current application. Only included in this list are EJBs whose remote (home and business) interfaces are defined.



Working with Java Controls

9. Select the appropriate EJB from the list and click **Select**. The name appears in the *jndi-name* field, and the interfaces used by this EJB appear in the *home interface* and *bean interface* fields.
10. Click **Create**.

To create a new EJB control

1. Locate or create the folder where you want to create the EJB control. This can be a folder in a Control project, Web project, or Web Service project.
2. Right-click the folder and choose **New**—>**Java Control**. The *New Java Control* dialog opens.
3. In the top pane, select EJB Control. In the *File name* field, enter the name of the new EJB control. Click **Next**.
4. Click the **Browse application EJBs** or **Browse server EJBs** button. The *Select EJB* dialog appears. Go to step 7 above and follow the instructions.

When you open the EJB control in Design View, you might notice that no methods are shown. For more information, see the next section.

Modifying EJB Controls

If you open an EJB control in Source View, you will notice that the control simply extends the EJB's interfaces. Unlike some other controls, the EJB control's class definition does not contain method declarations that invoke the EJB's methods. In other words, the EJB control only serves to reference the EJB and to expose its methods, and cannot be used to limit access to, or modify, these methods. To modify access to the EJB's methods, you must redefine the EJB's interfaces.

To see the EJB methods exposed by the EJB control, insert an EJB control in a web service or page flow, or, for control projects, build the EJB control's control project.

When you modify the EJB's methods or add additional methods to the interfaces that the EJB control references, you do not need to modify the EJB control (but you must rebuild the EJB). When you modify the name of the used interfaces, the JNDI name (if the EJB control uses the jndi name), or the bean name or EJB JAR name (if the EJB control uses a ejb-link), you must modify the EJB control to reflect these changes.

Accessing EJBs on a Different Server

You can access EJBs on a different server with an EJB control, provided the server hosting the EJB control and the server to which the target EJB is deployed are in the same domain. You access EJBs on a different server by using special JNDI syntax in the *jc:ejb* tag's *home-jndi-name* attribute. For example:

```
@jc:ejb home-jndi-name="jndi://username:password@host:7001/my.resource.jndi.object"
```

You can also use environment properties to specify configuration information, such as:

```
@jc:ejb  
home-jndi-name="jndi://host:7001/MyEJBHome?SECURITY_PRINCIPAL=me&SECURITY_CREDENTIALS=me"
```

For more information on environment properties, see the JNDI Tutorial.

You can also override the value of the *@jc:ejb home-jndi-name* attribute using the EJB control method *setJndiName()*. For more information see *Setting the Target EJB Information at Runtime* below and

setJndiName().

To make an EJB control for an EJB on a different server, use the New Control wizard as described above to create an EJB control for an EJB on the local server (or duplicate an existing EJB control), and then go to Source View to adjust the `jc:ejb` tag and the names of the interfaces.

Note: Accessing a remote EJB in a different domain via the EJB control requires advanced transaction configuration. Please consult the "Configuring Domains for Inter-Domain Transactions" section of the Administration Console Online Help topic in the WebLogic Server documentation on e-docs.bea.com.

Setting the Target EJB Information at Runtime

You can set the target EJB information at runtime using the EJB control method `setJndiName()`. The method can set the host, port, and JNDI name of the target EJB. The values set by `setJndiName()` override those set by `@jc:ejb home-jndi-name`. If no EJB is found at the location set by `setJndiName()`, then the value set by `@jc:ejb home-jndi-name` is used as a fallback.

The `setJndiName()` method takes the same syntax as the `@jc:ejb home-jndi-name` annotation. To set a new JNDI name, without setting a new server or port, use the following syntax.

```
myEJBControl.setJndiName("my.resource.jndi.object");
```

To set a new JNDI name, server and port, use the following syntax.

```
myEJBControl.setJndiName( "jndi://username:password@host:7001/my.resource.jndi.object" );
```

Related Topics

Using WebLogic Built-In Controls

Using an EJB Control

How Do I: Create and Use a Java Control Within a Control Project?

How Do I: Add an Existing Enterprise JavaBean to an Application?

How Do I: Test an Enterprise JavaBean?

Tutorial: Enterprise JavaBeans

Using an EJB Control

After you have created an EJB Control, you can invoke an target EJB method via the EJB control. Specifically, the EJB control exposes all and only the EJB methods defined in the EJB interfaces that the control extends. You can invoke these methods simply by invoking the method with the same signature on your EJB control.

The EJB control automatically manages locating and referencing the EJB instance, and directs method invocations to the correct instance of the target EJB. Whether or not you must first create an instance of the target EJB using the EJB's create method depends on whether the EJB control references a session or an entity bean. This is discussed next.

Selecting Instances for Session EJBs

A session EJB is used to execute business tasks for a client on the application server. The session EJB might execute only a single method for a client, in the case of stateless session beans, or it might execute several methods for that same client, in the case of stateful session beans. A session bean never serves multiple clients at the same time. The lifetime of a stateful session bean is tied to the duration of the conversation with the client. In contrast, a small number of pooled stateless session bean instances is used to serve large number of client requests. For more information on the lifetime of stateless and stateful session beans, see *The Life Cycle of a Session Bean*.

Creating a Session EJB

If the target EJB is a stateless session bean, you do not need to invoke the create method of the EJB via the EJB control. Instead, the EJB control automatically creates a reference to an appropriate instance of the EJB whenever one of the EJB's business methods is invoked, as is shown in this code fragment:

```
/**
 * @common:control
 */
private EJBControls.MusicBeanControl library;
...
// create method is not invoked first
allBands = library.getBands();
```

If the target EJB is a stateful session bean you must first invoke (one of) its create method(s) to obtain a reference. For more information about the differences between stateful and stateless session beans, see *Getting Started with Session Beans*.

Caching a Session EJB Reference

After a reference is obtained, it is cached in the EJB control and is used for any subsequent calls to the EJB within the method invocation in which the initial call was made. If for a stateless session bean you explicitly call the create method, or if for a stateful session bean you again call a create method, the EJB control replaces a previously cached reference with the newly created reference.

The lifetime of the cached EJB reference within the EJB control depends on the invoking application and the type of session bean. If a stateful session EJB is invoked by a conversational web service (that is, a web service method that takes part in a conversation), the EJB reference's lifetime is the lifetime of the

conversation. If a stateful or stateless session EJB is invoked by a non-conversational web service, the lifetime of the EJB reference in the EJB control is the lifetime of the web service method invocation. For page flows and both stateful and stateless session EJBs, if the session EJB is defined in the controller class, the lifetime of the reference is the lifetime of the page flow.

Removing a Session EJB

When you call the remove method on an EJB control that represents a session EJB, the currently cached instance of the bean is released. The server might destroy the bean at that time, but the actual behavior is up to the server. Either way, the bean no longer communicates with the EJB control.

Selecting Instances for Entity Beans

Instances of entity EJBs are associated with a particular collection of data. Typically this collection of data is a row in a database table.

Creating an Entity EJB

When you invoke the EJB's create method through the EJB control, you create a new persistent entity, that is, a new record in the underlying database table. In other words, creating a new entity bean with the create method amounts to inserting a new record in a table.

Referencing an Entity EJB

You can reference an entity EJB instance by calling the `findByPrimaryKey` method, or another `findXxx` method provided by the EJB's designer that returns a reference to one entity bean. In other words, the entity bean instance represents an existing record in a database table.

Caching an Entity EJB Reference

The EJB control caches a reference to the EJB instance being used, that is, the instance returned by the most recent call to the create, `findByPrimaryKey` or `findXxx` method, which returns one data record. When you invoke subsequent methods on the EJB control, it invokes that method on the EJB instance to which the cached reference refers. If there is no EJB reference currently cached, the EJB control attempts to invoke the `findByPrimaryKey` method with the last successful key used in a create or `findByPrimaryKey` call. If there is no previous key, the EJB control throws an exception.

The lifetime of the cached entity EJB reference within the EJB control depends on the invoking application. If the entity EJB is invoked by a conversational web service (that is, a web service method that takes part in a conversation), the EJB reference's lifetime is the lifetime of the conversation. For non-conversational web services, the lifetime of the EJB reference in the EJB control is the lifetime of the web service method invocation. For page flows, if the entity EJB is defined in the controller class, the lifetime of the reference is the lifetime of the page flow.

Removing an Entity EJB

When you call the remove method on an EJB control that represents an entity EJB, the record represented by the cached EJB reference is removed from the underlying persistent storage. That is, the row is deleted from the database table.

Returning Multiple Records

A findXxx method may return a Collection object, holding a set of references to entity beans. The EJB control does not cache this object. If you wish to cache the return value of a findXxx method, you should store the object in a member variable of the application invoking the EJB control.

Related Topics

[Overview: Enterprise JavaBeans and EJB Controls](#)

[Creating a New EJB Control](#)

[Getting Started with Session Beans](#)

[Getting Started with Entity Beans](#)

Handling EJB Exceptions

The EJB control makes it easy to use an existing, deployed EJB from within an application. This topic describes how to handle exceptions that might be thrown by the target EJB or the EJB control itself.

Checked Exceptions

If the target EJB method invoked via an EJB control throws a checked exception (that is, an exception that does not extend a `RuntimeException`), a try–catch block must catch the exception. It is generally considered a best practice to catch exceptions that occur within the EJB method (thrown by other methods the EJB method uses), and either overcome the exception or, when this is impossible, rethrow these exceptions either as an *application exception* or an `EJBException`, depending on whether the failure is due to a system–level or business logic error, and whether the transaction should be automatically rolled back.

An application exception is a checked exception that is either defined by the bean developer and does not extend a `RemoteException`, or is predefined in the `javax.ejb` package (that is, `CreateException`, `DuplicateKeyException`, `FinderException`, `ObjectNotFoundException`, or `RemoveException`). The EJB's method explicitly defines any application exception in the throws statement, and a try–catch block must catch the exception.

If the EJB control uses the EJB control's remote interfaces, a `RuntimeException` or an `EJBException` thrown by the EJB method is nested by the EJB container inside a `RemoteException`, and the `RemoteException` is propagated to the client. Because a `RemoteException` is a checked exception, the client must catch the exception. For more information on `RemoteException`, see your favorite J2EE reference documentation or the J2SE API documentation at <http://java.sun.com>. An example of catching a `RemoteException` is given below.

Runtime Exceptions

A `java.lang.RuntimeException` and its subtypes, including `EJBException`, can be thrown by an EJB method via its corresponding EJB control. Although these exceptions do not have to be explicitly caught in your code, it is generally a good idea to catch these exceptions in the client application invoking an EJB control, which uses the EJB's local interfaces. (Remember that for remote interfaces, the `EJBException` is rethrown by the EJB container as a `RemoteException`.)

As mentioned above, a checked exceptions caught in a bean method is often rethrown as an `EJBException`. You can checked for such a nested exception by invoking the `getCause()` or `getCausedByException()` methods on the caught `EJBException`. For more information on `EJBException`, see your favorite J2EE reference documentation or the J2EE API documentation at <http://java.sun.com>. An example of nesting and catching an exception through `EJBException` is given below.

The EJB control will throw a `com.bea.ControlException` when it has a problem locating/referencing the EJB. Although this is a subtype of `RuntimeException` and therefore does not have to be caught explicitly, it again might be a good idea to catch it in the client application. For more information, see the `ControlException` API.

A Nested Exception Example

The following example demonstrates the rethrowing of a checked exception inside an `EJBException` by an EJB method, and the catching of this exception on the client side. Rethrowing the exception inside an `EJBException` in the example is done solely to illustrate the mechanics of exception handling, and should not

be considered recommended design. As mentioned above, checked exceptions should be rethrown either as an *application exception* or an `EJBException`, depending on whether the failure is due to a system-level or business logic error, and whether the transaction should be automatically rolled back.

The first code snippet shows the definition of the `MusicBean` method `addRecording`. Notice that `FinderException`, which can be thrown by `findByPrimaryKey`, is rethrown inside an `EJBException`, and that `CreateException`, which can be thrown by the `BandBean`'s business method `addThisRecording`, is rethrown inside an `EJBException`:

```
/**
 * @ejbgen:local-method
 * @ejbgen:remote-method
 */
public void addRecording(String band, String recording)
{
    try {
        Band bandBean = bandHome.findByPrimaryKey(new BandPK(band));
        if(bandBean != null) {
            bandBean.addThisRecording(recording);
        }
    }
    catch(CreateException ce) {
        throw (EJBException) new EJBException(ce).initCause(ce);
    }
    catch(FinderException fe) {
        throw (EJBException) new EJBException(fe).initCause(fe);
    }
}
```

On the client side, the `MusicBean`'s method is invoked via an EJB control, and an `EJBException` is caught and checked. If the client uses an EJB control that locates the EJB via its local interfaces, you can catch the `EJBException` directly and retrieve the nested exception. The following code snippet shows how this is done in a page flow's action method, using an EJB control that locates the EJB via its local interfaces:

```
/**
 * @common:control
 */
private EJBControls.MusicBeanControl library;

...

/**
 * @jpf:action
 * @jpf:forward name="success" path="addRecording.jsp"
 */
protected Forward addARecording(AddARecordingForm form)
{
    String recording = (form.getRecordingName()).trim();
    String bandChoice = form.getSelectedBand();
    if(recording.length() != 0) {
        try {
            library.addRecording(bandChoice, recording);
            allRecordings = library.getRecordings(bandChoice);
        }
        catch(EJBException ee) {
            Exception ne = (Exception) ee.getCause();
            if(ne.getClass().getName().equals("FinderException"))
                ...
        }
    }
}
```

Working with Java Controls

```
        else if(...)
            ...
    }
}
...
return new Forward("success");
}
```

If the client uses an EJB control that references the EJB via its remote interfaces, you must catch the `RemoteException` instead and retrieve its nested exception. The following code snippet shows how this is done:

```
/**
 * @common:control
 */
private EJBControls.RemoteMusicBeanControl remoteLibrary;

...

/**
 * @jpf:action
 * @jpf:forward name="success" path="addRecording.jsp"
 */
protected Forward addARecording(AddARecordingForm form)
{
    String recording = (form.getRecordingName()).trim();
    String bandChoice = form.getSelectedBand();
    if(recording.length() != 0) {
        try {
            remoteLibrary.addRecording(bandChoice, recording);
            allRecordings = library.getRecordings(bandChoice);
        }
        catch(RemoteException re) {
            EJBException ee = (EJBException) re.getCause();
            Exception ne = (Exception) ee.getCause();
            ...
        }
    }
    ...
    return new Forward("success");
}
```

Related Topics

[Introduction to Java](#)

JMS Control

Java Message Service (JMS) is a Java API for communicating with messaging systems. Messaging systems are often used in enterprise applications to communicate with legacy systems or to provide communication lanes between business components running in different environments or on different hosts. The JMS control enables applications built in WebLogic Workshop to easily interact with messaging systems that provide a JMS implementation, such as WebLogic Server or Message–Oriented Middleware systems (MOMs).

The JMS control enables WebLogic Workshop web services to easily interact with messaging systems that provide a JMS implementation. A specific JMS control is associated with particular facilities of the messaging system. Once a JMS control is defined, web services may use it like any other WebLogic Workshop control.

Note: The JMS control in WebLogic Workshop 8.1 is significantly different than it was in WebLogic Workshop 7.0. The changes that you need to make to update your JMS controls to work in WebLogic Workshop 8.1 are outlined in the following sections. The main differences are as follows:

- XML maps are no longer supported on a JMS control. You can now use XMLBeans to construct the body of a JMS message. For more information, see *Specifying the Message Body*.
- Only one parameter can be used to specify the JMS message body in a method that sends a JMS message, and only a single parameter can be used to receive the message body in a callback that receives a JMS message. You can use an XMLBeans type for this parameter. The `@jws:jms-message` annotation is no longer valid. For more information, see *Specifying the Message Body*.
- The `@jws:jms-header` annotation is now the `@jc:jms-headers` annotation. The `@jc:jms-headers` and `@jc:jms-property` annotations have new syntax for specifying JMS headers and properties. See *Specifying Message Headers and Properties* for more information.
- You must now specify the message style (topic or queue) separately for outgoing and incoming JMS messages. The `@jc:jms` annotation has a `send-type` attribute and a `receive-type` attribute; you must specify one or both for the JMS control.

For more information on upgrading to WebLogic Workshop 8.1, see *Upgrading Workshop Applications*.

Topics Included in This Section

Overview: Messaging Systems and JMS

Describes messaging services in general and the Java Message Service in particular

Creating a New JMS Control

Describes how to create a JMS control.

Configuring a JMS Control

Explains how to configure a JMS control, including adding and defining methods and callbacks, specifying message format, and accessing remote resources.

Specifying the Body of a Message

Describes how to use XML maps to translate between the JMS control's method and callback parameters and the body of messages sent and received by the control.

Specifying Message Headers and Properties

Describes how to apply XML maps to the headers and properties of messages sent and received by the JMS control.

Supported and Unsupported Messaging Scenarios

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

JMSControl Interface

Provides the interface information for the JMS control.

JMS Control Samples

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

Related Topics

Using WebLogic Built-In Controls

Overview: Messaging Systems and JMS

A JMS control makes it easy for your application to communicate with messaging systems. To better understand how to use a JMS control, it helps to understand messaging systems and how JMS control interact with them. This topic describes the characteristics of messaging systems that are accessible using Java Message Service (JMS), and explains how JMS interacts with them.

Understanding 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. This means that a component sends a message to a destination and a message recipient can retrieve messages from a destination, but 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 manages the actual message delivery.

Messaging systems also provide reliability for message delivery. 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*. Each is described in greater detail below.

Using JMS Queues for Point-to-Point Messaging

Point-to-point messaging is accomplished with JMS queues, which are specific named resources configured in a JMS server. A JMS client, of which the JMS control is an example, sends messages to a queue or receives messages from a queue.

Point-to-point messages have a single consumer. Multiple receivers can 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.

The messaging system continues to resend a particular message until a predetermined number of retries have been attempted. Once the message is received, a message consumer acknowledges receipt.

To learn more about sending messages to queues, see [Supported and Unsupported Messaging Scenarios](#).

Using JMS Topics for Publish-and-Subscribe Messaging

Publish-and-subscribe messaging is accomplished with JMS topics. A topic is a specific named resource configured in a JMS server.

A JMS client, of which the JMS control is an example, publishes messages to a topic, or subscribes 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.

To learn more about publishing and subscribing to topics, see Supported and Unsupported Messaging Scenarios.

Connection Factories and Transactions

Before a JMS client can send or receive messages to a queue or topic, it must obtain a connection to the messaging system, 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, where clients wishing to make a connection can look them up.

There is a default connection factory pre-configured in WebLogic Workshop named `cgConnectionFactory`. You can use this connection factory for all JMS controls that do not explicitly override it.

To override the default connection factory and specify another connection factory for the control to use, you must (1) add the connection factory to the deployment descriptor (see the example `weblogic-ejb-jar.xml` file below) and (2) add the connection factory definition to the server's configuration file (see the example `config.xml` file below). If the JMS traffic is to be transacted then (3) you must also make sure that the connection factory is XA-enabled (see the example `config.xml` file below).

(1) To change the connection factory in the deployment descriptor, in the `weblogic-ejb-jar.xml` file, edit the `connection-factory-jndi-name` element to point to the new connection factory class.

weblogic-ejb-jar.xml

```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>myMessageDrivenBean</ejb-name>
    <message-driven-descriptor>
      .
      .
      .
      <connection-factory-jndi-name>some.other.QueueConnectionFactory</connection-factory-jndi-name>
      .
      .
      .
    </message-driven-descriptor>
    <jndi-name>myMessageDrivenBean</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

For detailed information on the `weblogic-ejb-jar.xml` file, see `weblogic-ejb-jar.xml` Deployment Descriptor Reference in the WebLogic Server 8.1 documentation.

(2) To add the connection factory to the server's configuration file, add a `<JMSTConnectionFactory>` element to the server's `config.xml` file.

(3) To ensure that the JMS traffic participates in transactions, set the `XAConnectionFactoryEnabled` attribute to true.

config.xml

```
<JMSConnectionFactory  
  JNDIName="myMessageDrivenBean"  
  Name="JMSConnectionFactory"  
  Targets="myWebLogicServer"  
  XAConnectionFactoryEnabled="true"/>
```

For detailed information on the <JMSConnectionFactory> element, see JMSConnectionFactory in the WebLogic Server 8.1 documentation.

Configuring Java Messaging Service

To learn how to create, configure and register JMS queues, topics and connection factories, please consult the WebLogic Server documentation on Programming WebLogic JMS.

Related Topics

JMS Control

Using WebLogic Built-In Controls

Supported and Unsupported Messaging Scenarios

Creating a New JMS Control

The JMS control enables applications built in WebLogic Workshop to interact with messaging systems that provide a JMS implementation. This topic describes how to create a new JMS control and provides an example of a valid JMS control file.

Adding a New JMS Control to a Web Service or Control

You can add a JMS control in any of the following types of files:

- Java Control (JCS file)
- Java Page Flow (JPF file)
- Java Server Page (JSP file)
- Java Web Service file (JWS file)
- Process file (JPD file)

To add a JMS control, follow these steps:

1. Open the file to which you want to add the control in the WebLogic Workshop design environment.
2. Display the Data Palette (**View**→**Windows**→**Data Palette**).
3. On the Data Palette, click the **Add** drop-down, and choose **Web Service**. The **Insert Control – Insert JMS** dialog opens.
4. In **Step 1**, in the **Variable name for this control** field, type the name for your JMS control.
5. In **Step 2**, select the **Create a new JMS control to use** radio button.
6. In the **New JCX name** field, type 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 runtime** check box. For more information about control factories, see Control Factories: Managing Collections of Controls.
8. In **Step 3**, from the **Message type** drop-down list, select the type of message you want to process. For more information about the types of messages, see the Configuring a JMS Control.
9. From the **JMS 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.
11. 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.
12. 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.
13. Click **Create**.

You can also create a new JMS control file without adding it to a web service or another control file. From the File menu, choose **New**→**Java Control**. Select the JMS control from the list, and follow the instructions to create a new control file.

The values that you provide when you create the control are specified in the control's @jc:jms annotation, as shown in the sample file in the next section.

The JCX File for JMS Control

If you create a new JMS control JCX file using the settings described above, the new JCX file looks like the following example. The contents of the JMS control's JCX file depend on the selections made in the ***Insert Control – Insert JMS*** dialog. The following example was generated with ***Text*** selected from the ***Message type*** drop-down list:

```
import com.bea.control.*;
import java.io.Serializable;
/**
 * @jc:jms send-type="queue" send-jndi-name="jms.SimpleJmsQ"
 *         receive-type="queue" receive-jndi-name="jms.SimpleJmsQ"
 *         connection-factory-jndi-name="weblogic.jws.jms.QueueConnectionFactory"
 */
public interface SimpleQueueControl extends JMSControl
{
    /**
     * this method will send a javax.jms.TextMessage to send-jndi-name
     */
    public void sendTextMessage(String payload);
    /**
     * If your control specifies receive-jndi-name, that is your JWS expects to receive message
     * from this control, you will need to implement callback handlers.
     */
    interface Callback extends JMSControl.Callback
    {
        /**
         * Define only 1 callback method here.
         *
         * This method defines a callback that can handle text messages from receive-jndi-name
         */
        public void receiveTextMessage(String payload);
    }
}
```

The JCX file contains the declaration of a Java interface with the name specified in the dialog. The interface extends the JMSControl base interface. To learn more about JCX files generated for messages of other types, see [Specifying the Body of a Message](#) and [Specifying Message Headers and Properties](#).

Related Topics

[JMS Control](#)

[Using WebLogic Built-In Controls](#)

[Overview: Messaging Systems and JMS](#)

[@jc:jms-headers Annotation](#)

[@jc:jms-property Annotation](#)

[JMSControl Interface](#)

Configuring a JMS Control

The JMS control enables WebLogic Workshop applications to interact with JMS messaging systems. When you create a JMS control, you specify the messaging style (queue or topic) and provide the JNDI identification for the sending and receiving queue or topic. You also specify a connection factory for the control. You can modify these settings later in the Property Editor pane in Design View, or you can directly modify the attributes of the `@jc:jms` annotation in Source View. For detailed information on the `@jc:jms` annotation and its attributes, see `@jc:jms` annotation.

Once you've set up the JMS control, you can add code to send and receive messages via JMS.

JMS Control Methods

The JMS control has the following default methods upon creation:

- The `getSession()` method, which gives you programmatic access to the JMS session. You don't need to call this method, but it's available to you if you want more control.
- One of the following methods for sending a message to the service, depending on what value you selected for the Message Type when you created the control: `sendTextMessage()`, `sendBytesMessage()`, `sendObjectMessage()`, or `sendJMSMessage()`. You can use one or more of these methods, or you can define your own methods for sending a message to the service.
- The `subscribe()` method, which you can call from your code to subscribe to a JMS topic.
- The `unsubscribe()` method, which you can call from your code to unsubscribe from a JMS topic.

All of the methods you define on the JMS control send or publish to the queue or topic named by the `send-jndi-name` attribute of the `@jc:jms` annotation.

JMS Control Callback

The JMS control has a single callback defined by default. You can replace this callback with your own, but you may have only one callback on the JMS control. The reason for this restriction is that the messaging service returns a message to the control via this callback, and it can call only one. Depending on the value you selected for the Message Type when you created the control, the default callback will be one of the following: `receiveTextMessage()`, `receiveBytesMessage()`, `receiveObjectMessage()`, or `receiveJMSMessage()`. The callback you define on the JMS control receives from the topic or queue named by the `receive-jndi-name` attribute of the control's `@jc:jms` annotation.

If the JMS control cannot deliver a message to a control instance via the callback (for example, if there is no conversation ID for a control that listens to a queue), it throws an exception from the `JMSControl.onMessage` method. This causes the current transaction to be rolled back. What happens next depends on how the administrator set up the JMS destination. The destination should be set up to have a small retry count and an error destination, which allows the processing to stop when an error is thrown. However, if the destination is configured with a large retry count or no retry count and with no error destination, the JMS control infrastructure continues attempting to process the message (unsuccessfully) forever.

Note that failure to deliver a message can be caused by an uncaught exception in code running in the callback handler, or in code running in a method called by the callback handler. Make sure that you catch all exceptions so that your code can adapt to this situation if it happens.

Working With Multiple Types of Web Services

If you have multiple types of web services (which is different from having multiple instances of the same type) that reference the same receive-jndi-name for a queue, you must use the receive-selector attribute in a way that causes the web services to partition all received messages into disjoint sets. If this is not handled properly, messages for a particular conversation might 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 JMS control and will violate this caveat.

Experimenting with JMS Control Samples

In order to support JMS control samples, two queues and one topic are configured on the sample Workshop domain when WebLogic Workshop is installed. The queues are named SimpleJmsQ and CustomJmsCtlQ, and the topic is named jms.AccountUpdate. The connection factory that provides connections to these queues has the JNDI name weblogic.jws.jms.QueueConnectionFactory. You can use these resources to experiment with JMS controls.

You can see samples of the JMS control in the SamplesApp application, in the jms directory of the WebServices project.

Note: Each JMS control should use a unique queue. Multiple JMS controls on the same server may not simultaneously use the same queue.

Related Topics

JMS Control

Creating a New JMS Control

Specifying the Message Body

This topic describes some of the ways in which you can specify the body of a message sent via the JMS control.

Note: In WebLogic Workshop 7.0, you specified the message body using an XML map. XML maps are no longer supported for specifying the message body on a JMS control. If you have existing JMS controls with the CTRL extension, these controls will continue to work. However, if you choose to upgrade your controls to JCX control files, you'll need to remove the `@jws:jms-message` tag and any XML maps that it specifies. You can modify these controls to use XMLBeans types instead of XML maps, as described below.

Selecting the Message Type

A JMS control can send and receive text messages (including XML messages), byte array messages, object messages, and `javax.jms.Message` (JMS Message) objects. These are the types defined by the JMS messaging service specification.

When you create a JMS control, you can specify which type of message it sends and receives in the Message Type drop-down of the Insert Control dialog. The value you choose determines what the default send and receive method signatures will look like; for example, if you specify Text/XMLBean as the message type, the control is created with a `sendTextMessage()` method and a `receiveTextMessage()` callback.

You have complete control over the send methods and receiving callback, as long as you are sending a message of a supported type; you can modify the default method signatures as you need to, including adding additional parameters to handle message headers and properties. However, you can only specify one parameter in the method or callback for the message body.

Although you have flexibility over what type of message your code sends and receives, note that the outgoing message and the incoming message must be of the same type; no implicit type conversion is performed on messages.

Sending and Receiving a Simple Text Message

The simplest message body is a text message. The following example shows a simple text message sent to the messaging service via a JMS control:

```
/**
 * @common:operation
 */
public void sendString(String msg) throws Exception
{
    myJMSControl.sendTextMessage(msg);
}
```

The receiving callback takes a parameter of type `String` that the messaging service sends to the control. The following is a simple example of a callback that receives a text message:

```
public void myJMSControl_onTextMessage(String msg)
{
    //this code calls another callback on the client, to return the message to the client
}
```

```
callback.onMessageReceived("[ " + name + " ] Your message is: " + msg );  
}
```

Sending and Receiving an XML Message using XMLBeans

If you need to send a set of values in the message body, you can construct the message body using an XMLBeans object type. WebLogic Workshop's XMLBeans technology generates a set of Java classes from an XML schema (.xsd) file. You can then use these classes to work with XML documents in your code. For more information on XMLBeans, see *Getting Started with XMLBeans*.

If you don't already have a schema file, you can construct one by hand, or you can generate one from an XML document or fragment using XMLSpy. WebLogic Platform includes a specially licensed version of XMLSpy to provide you with additional tools for working with XML. Once you've created your schema file, add it to the Schemas project in your WebLogic Workshop application. When you add a schema to the Schemas project, WebLogic Workshop automatically generates XMLBeans classes for that schema. The XMLBeans classes for all of the schemas in the Schemas project are packaged together in the library Schemas.jar, which appears in the Libraries folder at the root of your application.

In this case, you want to structure your schema so that you can pass a single object as the message body for the JMS message. The object can have any number of properties, but the top-level element of your schema should correspond to the object that you want to send as a message.

Once you've generated the XMLBeans classes from the schema file, you can import those classes into your JMS control class. You can then modify the send method or receiving callback on the JMS control to send or receive a message of the appropriate type.

Note that XMLBeans messages are transmitted as JMS text messages. When you create a JMS control that will use an XMLBeans type for the message body, specify the type in the **Message Type** drop-down as Text/XMLBean.

For an example that shows how to use an XMLBeans type as the body of a JMS message, see the AccountPublish.jws/AccountSubscribe.jws samples in the jms directory of the WebServices project in the SamplesApp application. These samples use an XMLBean type generated from the JmsSchemas.xsd file, which you can find in the Schemas project in the SamplesApp application.

Related Topics

JMS Control

Overview: Messaging Systems and JMS

Specifying Message Headers and Properties

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

@jc:jms Annotation

Specifying Message Headers and Properties

The JMS control includes properties for setting and retrieving headers and properties on a JMS message.

Note: In the previous version of WebLogic Workshop, you could use XML maps to translate between the JMS control's method and callback parameters and the JMS message headers and message properties. XML maps are no longer supported as a means of specifying JMS headers and properties. Instead, WebLogic Workshop provides new annotations for working with message headers and properties. If you upgrade an existing JMS control from a CTRL file to a JCX file, you'll need to update the control to use the new annotations.

Accessing Message Headers

A JMS message includes a number of header fields that contain information used to identify and route messages. You can retrieve the message headers for a incoming message received using the JMS control by using the `@jc:jms-headers` annotation. The `@jc:jms-headers` annotation has attributes that correspond to standard JMS headers. The supported headers for an incoming message are:

- `JMSCorrelationID`
- `JMSDeliveryMode`
- `JMSExpiration`
- `JMSMessageID`
- `JMSPriority`
- `JMSRedelivered`
- `JMSTimestamp`
- `JMSType`

For more information on these headers, see the Sun JMS specification.

You can specify whichever headers you are interested in as attributes on the `@jc:jms-headers` annotation, but each attribute value that you specify must map to a parameter of the same name in the callback signature for the JMS control. This way, WebLogic Workshop knows to retrieve the header value and pass it in to your code via the callback.

Only two of these headers can be set on an outgoing message: the `JMSCorrelationID` header and the `JMSType` header. The `JMSCorrelationID` header must be set to the conversation ID of the service that is listening. The `JMSType` header can be set to an arbitrary value to distinguish the type of message the sender is sending.

To see an example that retrieves JMS headers, take a look at the source code for the sample JMS control file `AccountSubscribeJMSControl.jcx`. This sample is available in the `SamplesApp` application, in the `jms` folder of the `WebServices` project.

Accessing Message Properties

A JMS message can also include properties that you or the message sender can add to send additional information about the message. You can think of them as optional, custom headers. Properties can be of type boolean, byte, short, int, long, float, double, or string. They can be set when a message is sent, or they can be read by the consumer of a message upon receipt. You can add as many properties to the message as you need to.

Working with Java Controls

You can set and retrieve the properties of messages sent and received using the JMS control by using the `@jc:jms-property` annotation. This annotation lets you specify properties as name/value pairs: the key attribute is the property identifier, and the value attribute is the property value.

You can also set properties from your code using the `setProperties()` method on the JMS control interface. You can return property values using the `getProperties()` method.

If the JMS control is receiving an incoming message, the JMS control extracts the incoming property values specified by the `@jc:jms-property` annotation and passes them to the control's callback. The property values must therefore map to the callback's parameters.

If the JMS control is sending a message, the JMS control adds the properties specified by the `@jc:jms-property` annotation to the outgoing message. You can optionally specify that a parameter passed to the method that sends the message should be substituted as a property value on the message.

Note that the properties specified on the outgoing message must match those on the incoming message in name and type. There is no implicit type conversion for properties on a JMS message.

Related Topics

JMS Control

Overview: Messaging Systems and JMS

Specifying the Message Body

SimpleJMS.jws Sample

CustomJMSClient.jws Sample

`@jc:jms` Annotation

`@jc:jms-headers` Annotation

`@jc:jms-property` Annotation

Supported and Unsupported Messaging Scenarios

The JMS control enables WebLogic Workshop applications to interact with messaging systems that provide a JMS implementation. This topic describes specific messaging scenarios that are supported by the JMS control, including sending messages to a queue, enabling two-way messaging with queues, publishing to a topic, and subscribing to a topic. It also describes the scenario of receiving unsolicited messages from a queue, which is not supported by the 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.

The messaging scenarios in the following sections are supported by the JMS control. For descriptions of messaging scenarios that are *not* supported by the JMS control, see A Messaging Scenario Not Supported by the JMS Control below.

Sending Messages to a Queue

An application can send messages to a JMS queue using a JMS control. The administrator who configures the target JMS queue determines the delivery guarantee policies, but in all cases the application does not receive a reply and the queue must exist and be registered in the JNDI registry in order to convey messages.

To send messages to a queue:

1. On the JMS control, specify the name of the target JMS queue by setting the value of the `send-jndi-name` attribute of the `@jc:jms` annotation. To learn how to create a JMS control, see [Creating a New JMS Control](#).
2. From your application, call either the JMS control's default method (`sendTextMessage`, `sendBytesMessage`, `sendObjectMessage` or `sendJMSMessage`, depending on the message type selected when the control was created), or a custom method you have defined for the JMS control.

The following is an example of sending messages to a queue:

```
public class MyService {

    /**
     * @common:control
     */
    private MyQueueControl myQueue;

    /**
     * @common:operation
     */
    public void sendID(String personID)
    {
        myQueue.sendTextMessage( personID );
    }
}
```

Enabling Two-Way Messaging with Queues

Using a JMS control, an application can send messages to one queue and receive reply messages on another queue. A single JMS control might have both send and receive queues configured, and applications can then send and receive via the same control.

Two-way messaging requires that every received messages be correlated with the instance of the application that sent the original outgoing message. This correlation is typically managed for you by the JMS control, so no action is necessary on your part. To learn more about message correlation, see the explanation of the `send-correlation-property` and `receive-correlation-property` attributes for the `@jc:jms` annotation.

To enable two-way messaging with queues:

1. On the JMS control, specify the name of the JMS queue to which you want to send messages by setting the value of the `send-jndi-name` attribute on the `@jc:jms` annotation.
2. Specify the name of the JMS queue from which you want to receive messages by setting the value of the `receive-jndi-name` attribute on the `@jc:jms` annotation.
3. To send a message from your web service, call the JMS control's default method (`sendTextMessage`, `sendBytesMessage`, `sendObjectMessage` or `sendJMSMessage` depending on the message type selected when the control was created), or a custom method you have defined for the JMS control.
4. To be notified when messages are received on the receive queue, implement a callback handler for the JMS control's callback (`receiveTextMessage`, `receiveBytesMessage`, `receiveObjectMessage` or `receiveJMSMessage` depending on the message type selected when the control was created), or a custom callback you have defined for the JMS control.

The following is an example enabling two-way messaging:

```
public class MyService {

    /**
     * @common:control
     */
    private MyQueueControl myQueue;

    /**
     * @common:operation
     */
    public void sendID(String personID) throws Exception
    {
        myQueue.sendTextMessage( personID );
    }

    myQueue_receiveTextMessage(String message)
    {
        // message has arrived from queue, perform desired operations
    }
}
```

Publishing to a Topic

A web service, using a JMS control, can publish messages to a JMS topic. The web service will not receive a reply. The topic must exist and be registered in the JNDI registry.

To publish to a topic:

Working with Java Controls

1. On the JMS control, specify the name of the target JMS topic by setting the value of the `send-jndi-name` attribute of the `@jc:jms` annotation.
2. From your web service, call the JMS control's default method (`sendTextMessage`, `sendMessage`, `sendObjectMessage` or `sendJMSMessage` depending on the message type selected when the control was created); or a custom method you have defined for the JMS control.

The following is an example of publishing to a topic:

```
public class MyService {  
  
    /**  
     * @common:control  
     */  
    private MyTopicControl myTopic;  
  
    /**  
     * @common:operation  
     */  
    public void sendID(String personID) throws Exception  
    {  
        myTopic.sendTextMessage( personID );  
    }  
}
```

Subscribing to a Topic

A web service, using a JMS control, can subscribe to messages on a JMS topic as long as the topic exists and is registered in the JNDI registry. Only messages sent after the web service has subscribed to the topic will be received.

To subscribe to a topic:

1. On the JMS control, specify the name of the target JMS topic by setting the value of the `receive-jndi-name` attribute on the `@jc:jms` annotation.
2. From your application, call the JMS control's `subscribe` method.
3. To be notified when messages are received on the receive topic, implement a callback handler for the JMS control's callback (`receiveTextMessage`, `receiveMessage`, `receiveObjectMessage` or `receiveJMSMessage` depending on the message type selected when the control was created); or a custom callback you have defined for the JMS control.
4. To automatically subscribe to a topic when the JMS control is first initialized, set the `auto-topic-subscribe` attribute of the `@jc:jms` annotation to **true**.
5. To stop being notified when messages are received on the receive topic, call the JMS control's `unsubscribe` method.

The following is an example of subscribing to a topic:

```
public class TopicForwardingService {  
  
    /**  
     * @common:control  
     */  
  
    /**  
     */  
    private MyTopicControl myTopic;
```

```
/**
 * @common:operation
 * @jws:conversation phase=start
 */
public void registerListener()
{
    myTopic.subscribe();
}

/**
 * @common:operation
 * @jws:conversation phase="finish"
 */
public void unregisterListener()
{
    // This isn't strictly necessary, the JWS will always
    // be unsubscribed on conversation finish.
    myTopic.unsubscribe();
}

myTopic_receiveTextMessage(String message)
{
    // message has arrived from queue, perform desired operations
}
}
```

A Messaging Scenario Not Supported by the JMS Control

Some scenarios that are possible in standalone applications are not possible in the WebLogic Workshop environment. The following messaging scenario is not supported by the JMS control.

Receiving Unsolicited Messages from a Queue

An application, via a JMS control, specifies a receive queue and subsequently receives unsolicited messages from that queue. An application must be performing work on behalf of a specific client and, in asynchronous situations, as part of a specific conversation. When an unsolicited message is received from a queue, it is not possible for the JMS control to determine the appropriate conversation or client with which to correlate unsolicited incoming messages.

Note: You can receive unsolicited messages in a web service, but the web service must be a direct JMS client (that is, it must not be using a JMS control for the queue in question). To learn how to use JMS directly, please consult the WebLogic Server documentation topic "Programming WebLogic JMS".

Related Topics

JMS Control

Using WebLogic Built-In Controls

Overview: Messaging Systems and JMS

@jc:jms Annotation

@jc:jms-headers Annotation

@jc:jms-property Annotation

JCX Files: Extending Controls

Files with the extension JCX are WebLogic Workshop *Java control extensions*. They typically include a collection of method definitions that allow you to easily access a resource such as a database or another enterprise resource.

Note: In previous releases, JCX files were known as CTRL files. CTRL files are still supported.

Types of JCX Files

The contents of a JCX file depend on the type of control the file extends. JCX files can represent the following types of controls:

- **Database control:** used to access a database from your web service. To learn more about Database Controls, see Database Control.
- **EJB control:** used to access an existing Enterprise JavaBean (EJB) from your web service. To learn more about EJB Controls, see EJB Control.
- **JMS control:** used to access an existing Java Message Service (JMS) queue or topic from your web service. To learn more about JMS Controls, see JMS Control.
- **Web Service control:** used to communicate with another web service from your service. To learn more about Web Service controls, see Service Control.
- **Extensible custom control:** Advanced users of WebLogic Workshop can create custom controls that are extensible in the same way built-in controls are extensible. To learn more, consult the documentation for the Control Developer's Kit.

For more information on WebLogic Workshop's built-in controls, see Using WebLogic Built-In Controls.

Using Existing JCX Files

In some cases, you may use an existing JCX file that was produced by another member of your team or another organization. For example, if many web services will use the same database, a single author might create a Database control extension (JCX file) that describes the interface to the database. Then multiple web service authors might use that JCX file to create a Database Control in their service and use it to access the common database. The same situation can occur for all of the control types.

Generating a JCX File

Whenever you create a control while editing a web service or other container, WebLogic Workshop generates a JCX file to contain a local representation of the control. The following are examples of situations in which a JCX file will be generated:

- When you create a new Database control: WebLogic Workshop generates a new JCX file to hold the Database control extensions definition. When you add methods to the Database control via the IDE, you are adding methods to the JCX file.
- When you add a Web Service control to access a web service based on the service's WSDL file: you can generate a Web Service control JCX file from the WSDL file, then use the new Web Service control from any control container.
For more information on WSDL files, see WSDL Files.
For more information on Web Service controls, see Web Service Control.

Working with Java Controls

Note: You should not use a Web Service control to invoke a web service that resides in the same application. Invoking a web service via a Web Service control means marshalling the method parameters into a SOAP message on the calling end and unmarshalling the SOAP message on the receiving end, then again for the method return value. This is very inefficient when the invocation is local. You would usually be tempted to invoke one web service from another if the called web service included business logic you want to access from the calling web service.

In general, you should place business logic in custom Java controls instead of in web services. This allows you to access the business logic from various contexts in the application (web services, other controls, page flows) without incurring the cost of data marshalling and unmarshalling. Web Service controls should *only* be used to invoke web services that are truly external to your application.

Related Topics

None.

Building Custom Java Controls

WebLogic Workshop allows you to create custom Java controls tailored to your project or application. This section explains how to create these controls.

For a complete overview of Java controls in WebLogic Workshop, including how to create them, see *Getting Started with Java Controls*. Also, be sure to see the *Tutorial: Java Control* for a step-by-step introduction to building controls.

Topics Included in This Section

Working with Custom Controls

Describes the basics of creating and using custom Java controls.

Source Files for Java Controls

Describes the files that are necessary in any Java control.

Related Topics

Using WebLogic Built-In Controls

Java Control Samples

Working with Custom Controls

This topic describes how to use a custom Java control. It explains how to:

- Create a custom control
- Copy an existing custom control
- Add an existing custom control

A custom Java control can be invoked by other custom controls or by a web service using the procedures described here. A custom Java control can also be invoked from a web page, although the procedure for invoking the control from a web page environment is somewhat different. For more information, see *Developing Web Applications and Page Flows and JSPs*.

To Create a Custom Control

To implement business logic for your web service, you can use a custom Java control. It may be a good idea create a folder in your project specifically for your new control. You cannot create the control in the root of a project:

- Right-click the folder for the new custom control, then choose *New-->Custom Java Control*.
- In the *New File* dialog, in the left-hand pane, select *All*.
- In the right-hand pane, select *Custom Java Control*.
- In the *File name* field, enter a name for your new control source (JCS) file. Be sure to append "Impl" to the file name, preceding the JCS extension.

The JCS file will contain your control's implementation class, which is why it requires an "Impl" ending. Note that WebLogic Workshop will preserve the extension as you type.

- Click *Create*.

WebLogic Workshop creates the JCS file and displays it in Design View. It also creates a JAVA file without the "Impl" ending for your control's public interface. As you build your control, you work in the JCS file, adding code for the control's logic. As you do, WebLogic Workshop updates the JAVA file code to reflect changes to the control's public interface. In other words, you never have to edit the JAVA file manually.

You can now add methods and callbacks to your control. For more information, see *How Do I: Begin a New Custom Java Control?*

Using an Existing Custom Control

If you have access to a custom Java control that you either implemented yourself or that was implemented by another developer, you can add this control to a web service or another custom Java control. You have access to a control if you have access to its JCS file in your project. If the control is not in your project, you can copy it to your project.

Copying the Custom Control

If the JCS file and the associated Java file for the custom control you wish to use is not in your project, you can copy it to your project. The destination to which you copy the control depends on the expected usage. If

the control will be used only by a single web service in your project, you may choose to copy the custom control to the same folder as your web service, or to a different folder in the same project as your web service. If you expect that you will reuse this control in multiple projects, consider creating a Java control project. For more information, see [How Do I: Create and Use a Java Control Within a Control Project?](#)

Be aware that if you copy a custom control, you need to copy both the JCS and the associated Java file. If your custom control in turn uses other controls, all the associated files must be copied. An easy alternative is creating a copy of the folder that contains the files of your custom control and the invoked controls. You can then move this duplicate folder to the other project. To copy a folder:

- Right-click the folder that contains your custom control.
- Select **Duplicate**. The duplicate is created in the same project. You can now drag the duplicate to the target project.

If you copy a custom control, you are in effect creating a new control. If you do not change the definition of the copy, you now have two identical but separate controls. If the original copy of the control is subsequently changed, the second copy will remain unchanged. This means that any bug fixes or feature enhancements made to the original control will *not* be reflected in the copy.

To Add an Existing Custom Control

If you have an existing custom control file in your project, you can add a reference to that control by dragging the JCS file from the Application pane to the Design View of the web service or custom control from which you would like to use it.

You can make multiple references to the same existing built-in control, for instance by referencing it in several web services in the same project. Note that you are making a reference and are not creating a copy. When you change the built-in control, the control is modified for every file in which it is referenced.

Control References in Source Code

When you add a control to your application, WebLogic Workshop modifies your file's source code to include an annotation and variable declaration for the control. The annotation ensures that the control is recognized by WebLogic Workshop, and the variable declaration gives you a way to work with the control from your code. For example, if you create a new custom control named Subscriptions in the CustomerControls folder in your project, and specify the variable name subscription, the following code will be added to your file:

```
/**
 * @common:control
 */
private CustomerControls.Subscriptions subscriptions;
```

Related Topics

[Working with Built-In Controls](#)

[Building Web Services](#)

Source Files for Java Controls

At their most basic, Java controls you develop include a Java control source (JCS) file. You can also add properties to the control by including an annotation XML file.

For a step-by-step introduction to building Java controls, see [Tutorial: Java Control](#).

Java Control Source Files

A Java control source (JCS) file contains the control's logic — the code that defines what the control does. In this file you define what each of the control's methods do. You can also define how control property values set by a developer influence the control's behavior, as in the following example.

When you add a new Java control source file to a project, WebLogic Workshop also adds a JAVA file that contains the control's public interface. Under most circumstances, you should not edit this file. By default, as you work in the JCS file, adding methods, callbacks, and implementation code, WebLogic Workshop keeps the interface in sync. For example, adding an operation to the JCS will also add a corresponding method to the JAVA file. Note that the JAVA file will be kept in sync only with respect to those methods with an `@common:operation` annotation. This means that if you add a method to the JCS, then remove its `@common:operation` annotation, WebLogic Workshop will remove the method from the JAVA file.

For information on creating a new Java control, see [How Do I: Begin a New Custom Java Control?](#)

```
package hello;

import com.bea.control.ControlContext;
import com.bea.control.ControlException;

/**
 * A control implementation class contains the logic for a control.
 *
 * The code-gen annotation tells WebLogic Workshop to create and maintain this
 * control's interface. This removes the necessity for you to do so.
 *
 * The control-tags annotation associates this control source file
 * with the annotation XML file that describes the properties it
 * exposes.
 *
 * @jcs:control-tags file="Hello-tags.xml"
 */
public class Hello
{
    /**
     * A constant to hold the name of the demeanor annotation.
     */
    public static final String TAG_DEMEANOR = "demeanor";

    /**
     * A constant to hold the name of the greetingStyle attribute.
     */
    public static final String ATTR_GREETING_STYLE = "greetingStyle";

    /**
     * The ControlContext interface provides access to aspects of a control's
     * container, including the properties stored for the control.
     */
}
```

```

*
* @common:context
*/
ControlContext context;

/**
 * Control methods are operations, just as with the methods of a
 * web service.
 *
 * @common:operation
 */
public String sayHello()
{
    String response = null;

    /*
     * Use the ControlContext interface to get the value of this control's
     * greetingStyle property attribute.
     */
    String greetingStyle = context.getControlAttribute(TAG_DEMEANOR, ATTR_GREETING_STYLE);

    /*
     * Return a greeting depending on the value the developer set in
     * the greetingStyle property attribute.
     */
    try {
        if (greetingStyle.equals("Cordial")) {
            response = "Hi! Nice weather we're having, eh?";
        }
        else if (greetingStyle.equals("Familiar")) {
            response = "Hi! How's the family?";
        }
        else if (greetingStyle.equals("Professional")) {
            response = "Hi! Great job on that presentation!";
        }
    }

    /*
     * Throw a ControlException if something unexpected happened.
     */
    } catch (Exception e) {
        throw new ControlException("There was an error greeting you!", e);
    }
    return response;
}
}

```

Control Property Definition File

The property definition file is an annotation XML file that defines the properties a control exposes, including their data types.

You create a property definition file based on a particular schema. For more information about the schema, see [Control Property Schema Reference](#). For step-by-step instructions on creating an annotations XML file and connecting it with your Java control, see [How Do I: Define Properties for a Java Control?](#)

The following example illustrates how you might define the properties for the preceding Hello sample. The property characteristics specified in this example include:

Working with Java Controls

- One property for the control—`demeanor`—and one attribute for that property—`greetingStyle`.
- The `greetingStyle` attribute takes one of three enumerated values.

```
<control-tags xmlns="http://www.bea.com/2003/03/controls/">
  <control-tag name="demeanor">
    <description>Defines the style of greeting this control returns.</description>
    <attribute name="greetingStyle" required="false">
      <description>Defines the style of greeting this control returns.</description>
      <type>
        <enumeration>
          <value>Cordial</value>
          <value>Familiar</value>
          <value>Professional</value>
        </enumeration>
      </type>
      <default-value>Cordial</default-value>
    </attribute>
  </control-tag>
</control-tags>
```

Related Topics

Tutorial: Java Control

Adding Portal Controls to Java Page Flows

Page Flows let you control a user's path through an application. The actions (such as a button click) and conditions (such as whether the user's login succeeded or failed or whether or not the user is a manager) determine which JSP the user is taken to next in the application. Portal Controls provide a variety of actions and ways to incorporate conditions that give you precise control over both a user's path through your applications and what occurs on that path.

This section provides descriptions of and guidance on using Portal Controls in your Page Flows.

Topics Included in This Section

Using Portal Controls

This section provides general guidance on adding and configuring Portal Controls in your Page Flows.

Group Provider Control

This control lets you use group management actions in your Page Flows.

Profile Control

This control lets you manage user profiles in your Page Flows.

Property Control

This control lets you manage and retrieve

Rules Executor Control

This control lets you evaluate objects in working memory against a set of rules.

Rules Manager Control

This control lets you look up information about the rule sets that can be used by the Rules Executor Control.

User Info Control

This control lets you retrieve information about users in your Page Flows.

User Login Control

This control lets you add login and logout to your Page Flows.

User Provider Control

This control provides user management actions in your Page Flows.

Click Content Event Control

Working with Java Controls

This control dispatches a `ClickContentEvent` to the event service for use in campaigns and behavior tracking.

Display Content Event Control

This control dispatches a `DisplayContentEvent` to the event service for use in campaigns and behavior tracking.

Generic Event Control

This control dispatches an event to the event service for use in campaigns.

Generic Tracking Control

This control dispatches a tracking event to the event service for use in campaigns and behavior tracking.

Rule Event Control

This control dispatches a `RuleEvent` to the event service for use in campaigns and behavior tracking.

Session Login Event

This control dispatches a `SessionLoginEvent` to the event service for use in campaigns and behavior tracking.

User Registration Event

This control dispatches a `UserRegistrationEvent` to the event service for use in campaigns and behavior tracking.

Note: WebLogic Portal also supports the legacy use of deprecated controls from the initial product release. If you have deprecated controls in your Page Flows, and you want to upgrade to the replacement controls to take advantage of new features, go to your Page Flow's Source View, click in the name of the control, and press F1. The Javadoc that appears tells you which control to use instead of the deprecated control.

Related Topics

[Using Portal Controls](#)

[Working with Java Controls](#)

[Guide to Building Page Flows](#)

Using Portal Controls

Portal Controls are collections of actions (Java methods) you can drag and drop into your Page Flows, making Java development easier and more automated. You can add actions in a graphical interface and configure the actions with the Property Editor, insulating you from working directly with Java code (though you can still work directly with code in Source View). Even if you want to work directly with code, working initially with the graphical interface (Flow View and Action View) automates code entry and makes it more syntax error free.

For example, Portal Controls provide built-in forms on some methods. If you want an action that creates a user, you can use the `createUser` method in the User Provider control. In the Page Flow's Action View, if you drag the `createUser` method from the Data Palette into the control's action area, the control provides a `CreateUserForm` bean that can be added to a JSP and linked to the action automatically. (That process is described in [To use controls that provide forms.](#))

To add a control to a Page Flow:

1. Open an existing Page Flow (.jpf file) or create a new "basic" Page Flow.
2. Select the Action View tab.
3. In the Data Palette, on the Controls bar, click **Add** --> [**Portal Controls or Portal Event Controls**] --> [**control**].
4. In the Insert Control dialog box, enter a name for the new control and click **Create**. The control and all its available methods appear in the Action View, as well as in the Data Palette under the Controls bar.

All the methods in the control are now available to your Page Flow.

5. Add a method (action) to your page flow by dragging a method from the Data Palette into the action area of the Page Flow (the left side of the window in Action View).
6. Switch to Flow View, where you can connect the action to the appropriate location in the Page Flow.

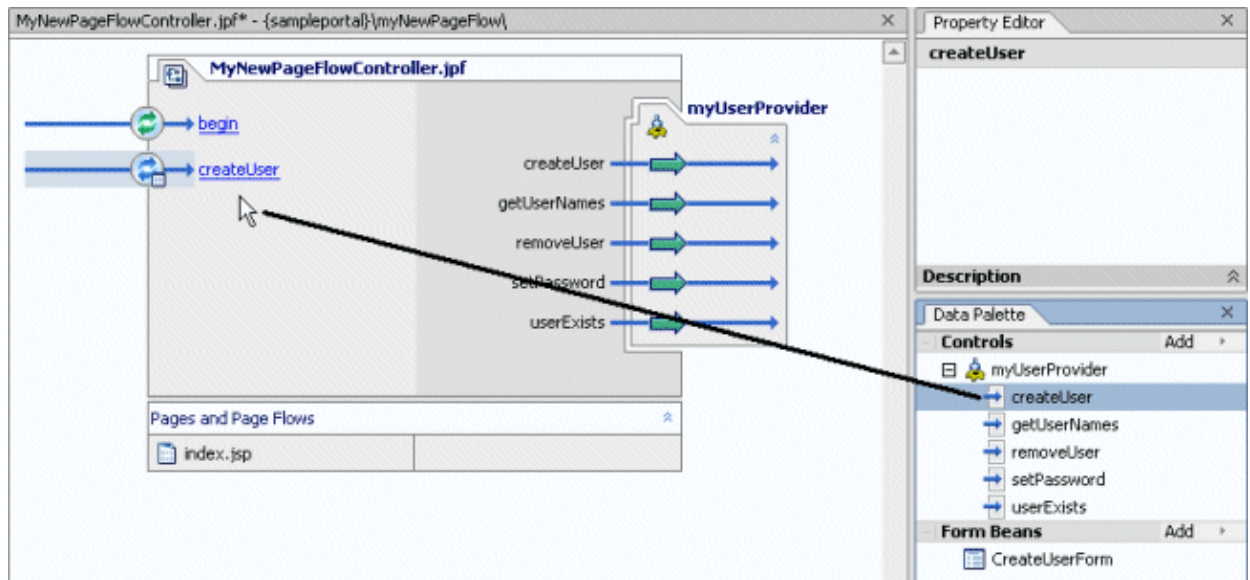
To use controls that provide forms:

Some methods (actions) on controls lend themselves to form entry, such as the `createUser` method on the User Provider control, where you can, for example, have a new user enter a username and password for self-registration. This procedure walks you through the `createUser` scenario to highlight the basics of adding such a form to a Page Flow.

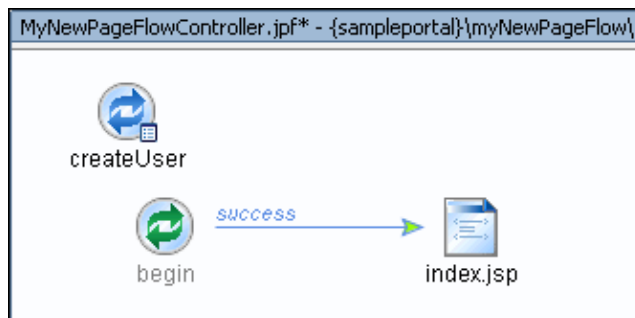
The basic scenario is providing a user self-registration form. When the user enters his username and password, the new user is created and the user is taken to the next JSP.

1. To use a `createUser` action, you must add the User Provider control to the Page Flow, as described in [To add a control to a Page Flow](#), above.
2. With the control in the Page Flow, select Action View in the Page Flow editor, and drag the `createUser` method from the Data Palette to the action area of the Page Flow, as shown in the following illustration. Notice in the action area of the Page Flow editor that the `createUser` icon is different than the begin icon, because the action provides a form. The form is visible in the Data Palette under the Form Beans bar.

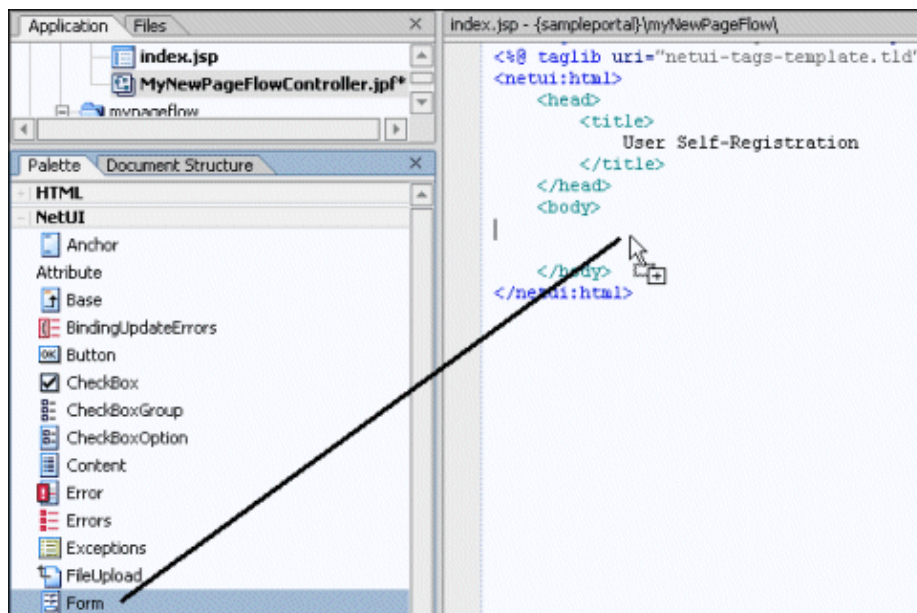
Working with Java Controls



3. The following illustration shows the Page Flow in Flow View, where the **createUser** action now appears. Again, the **createUser** icon shows there is a form associated with the action.

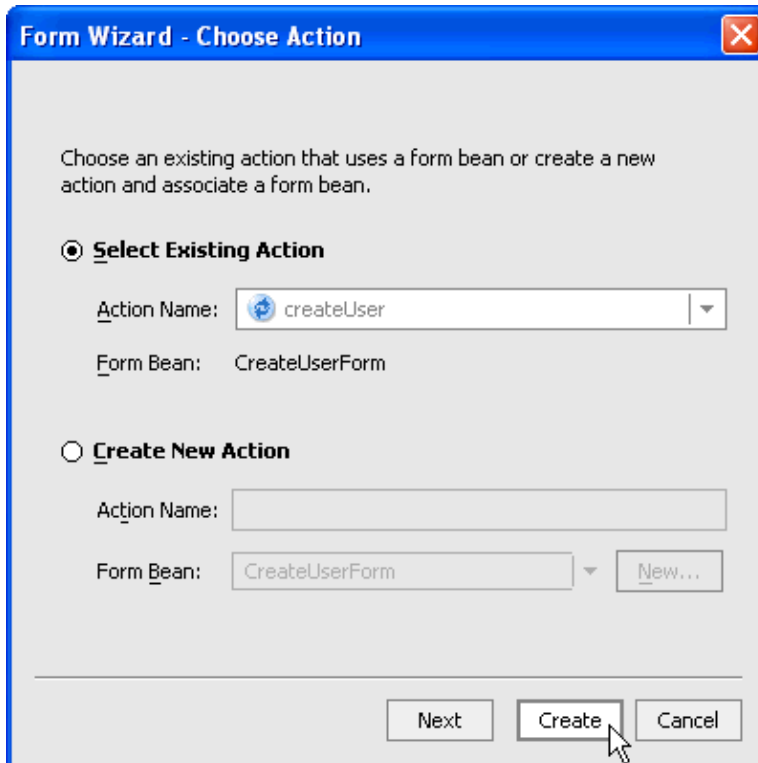


4. Now you can add the **createUser** form to the JSP where users will self-register. Open the JSP and select the Source View tab.
5. With the JSP open, drag the **Form** tag from the Palette into the JSP, as shown in the following illustration.



Working with Java Controls

6. In the Form Wizard window that appears, as shown in the following illustration, select the createUser action and click Create.



The image shows a dialog box titled "Form Wizard - Choose Action". It has a blue title bar with a close button. The main area is light gray. At the top, it says "Choose an existing action that uses a form bean or create a new action and associate a form bean." Below this, there are two radio buttons. The first is selected and labeled "Select Existing Action". Below it, there is a text field for "Action Name" containing "createUser" and a dropdown arrow. Below that is a text field for "Form Bean" containing "CreateUserForm". The second radio button is labeled "Create New Action". Below it, there is a text field for "Action Name" which is empty. Below that is a text field for "Form Bean" containing "CreateUserForm" and a "New..." button. At the bottom, there are three buttons: "Next", "Create", and "Cancel". A mouse cursor is pointing at the "Create" button.

7. The following illustration shows the form that is added to the JSP. Notice the form is tied to the createUser action. When the user fills in the fields and clicks the "createUser" button on the form, the form data is sent to the createUser action, and the new user is added to the authentication provider you specify in the Property Editor (with the control selected in Action View or Source View).

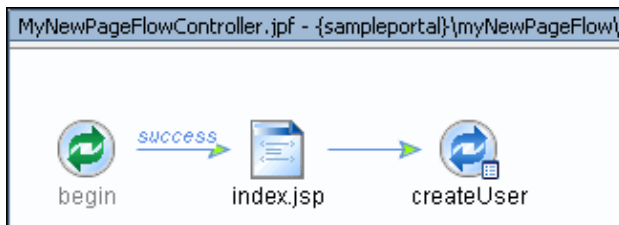
Note: In this example, the authentication provider must support write access for the user to be created. See the WebLogic Administration Portal online help for more information on using multiple authentication providers.

```

<netui:form action="createUser">
  <table>
    <tr valign="top">
      <td>Password:</td>
      <td>
        <netui:textBox dataSource="{actionForm.password}" />
      </td>
    </tr>
    <tr valign="top">
      <td>Request:</td>
      <td>
        <netui:textBox dataSource="{actionForm.request}" />
      </td>
    </tr>
    <tr valign="top">
      <td>Username:</td>
      <td>
        <netui:textBox dataSource="{actionForm.username}" />
      </td>
    </tr>
  </table>
  <br/>
  <netui:button value="createUser" type="submit" />

```

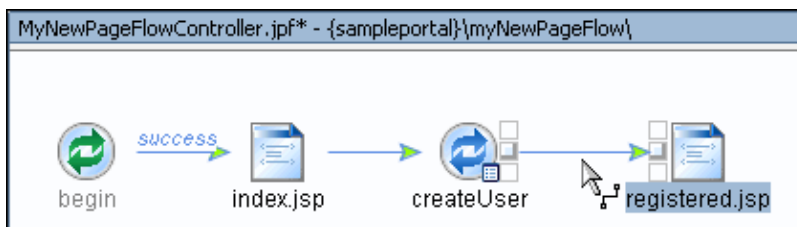
8. Save and close the JSP.
9. Select the Flow View tab on the Page Flow. Now that the form has been used for the createUser action, the Page Flow is automatically updated, as shown in the following illustration. Notice the arrow from the login.jsp to the createUser action. (The developer creating this Page Flow has rearranged the icons in Flow View.)



10. Now you have a form in a JSP that users can fill out to self-register, and you have an action that creates a new user with the user's form data. Now all you need is a JSP that will be displayed after the createUser action succeeds.

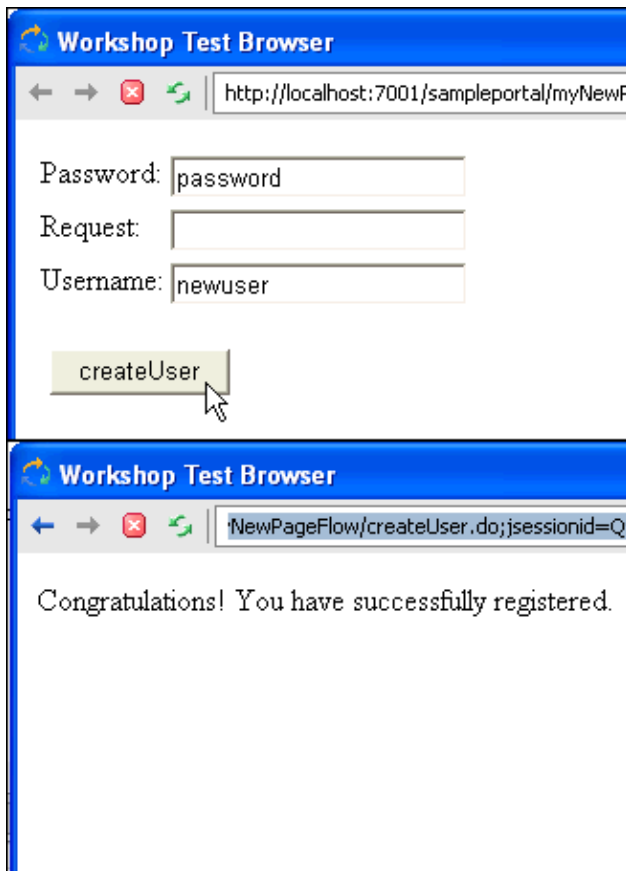
In the Page Flow directory, create a new JSP and give it a text message such as, "Congratulations! You have successfully registered."

11. In Flow View, connect the createUser action to the new JSP, as shown in the following illustration.



12. Save the Page Flow. You can view the results by clicking the Start button in the toolbar or pressing Ctrl+F5.

The user experience is shown in the following illustration.



13. As the previous illustration shows, you may need to modify the default form by rearranging/removing input fields and renaming buttons.

In a real-world Page Flow, you would create JSPs to handle action failures and exceptions, such as if a user entered a username that was already in use.

For detailed information on forms, see [Using Data Binding in Page Flows](#).

Related Topics

[Getting Started with Page Flows](#)

[Adding Portal Controls to Java Page Flows](#)

[Building Java Page Flow Portlets](#)

[Tutorial: Creating a Login Portlet Using Portal Controls](#)

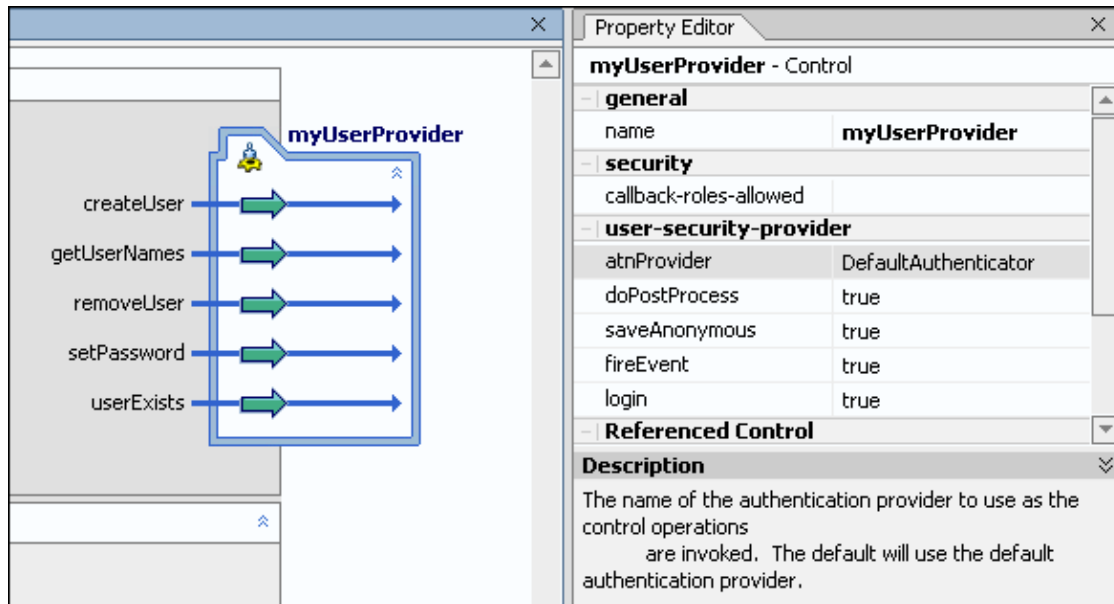
[Portal Control Properties](#)

[Portal Control Security](#)

[Portal Control Declaration](#)

Portal Control Properties

Portal Controls have properties you can edit in the Property Editor. Control properties—also called annotations—provide a convenient way to pass parameters to the underlying API at run time without having to manually pass the properties in your own Java code. For example, the User Provider Control has an `atnProvider` property that lets you enter the authentication provider that should be used for any of the control's actions, as shown in the following illustration.



The following code in Source View shows how the Page Flow uses the `atnProvider` property.

```
/**
 * @common:control
 * @jc:user-security-provider atnProvider="DefaultAuthenticator"
 */
private com.bea.p13n.controls.securityProvider.UserProviderControl myUserProvider;
```

To see which properties are available on a Portal Control, click in the control name in Source View and press F1 to see the control's Javadoc (the `com.bea.p13n.controls.*` packages).

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Control Security](#)

[Portal Control Declaration](#)

[Working with Java Controls](#)

[Building Custom Java Controls](#)

Portal Control Declaration

When you use WebLogic Workshop to drag and drop a Portal Control onto the Pageflow design view, the following code is automatically placed in the Pageflow controller source:

```
/**  
 * @common:control  
 */  
private com.bea.p13n.controls.login.UserLoginControl myControl;
```

If you are creating a control in the Page Flow's Source View, or outside of WebLogic Workshop, be sure to include this control declaration in this form.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Control Security](#)

[Portal Control Properties](#)

[Working with Java Controls](#)

Portal Control Security

Many Portal Controls have secured methods, meaning that any control attempting to execute such a method would need to be in an authorized security role. You can specify security roles in a Page Flow on each action. A user must be a member of the designated role(s) for the action to be fired. For example, the User Provider Control has a `removeUser()` method that requires the caller to be in the role of "PortalSystemAdministrator" or "Admin." See Portal Control Properties for more information.

For user and group management actions, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine whether or not the user can perform the action.

You can add security roles to a domain using the WebLogic Server Administration Console.

Related Topics

[Security Roles \(WebLogic Server e-docs topic\)](#)

[Adding Portal Controls to Java Page Flows](#)

[Portal Control Properties](#)

[Portal Control Declaration](#)

[Working with Java Controls](#)

Group Provider Control

The Group Provider control provides a convenient way to incorporate group management actions into your Page Flows, such as creating groups and getting a list of users in a group.

Use the `atnProvider` attribute to specify which authentication provider contains the groups you want to manage with the control.

Security: For user and group management actions, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine whether or not the user can perform the action.

If you use the `createGroup` action, also use the Profile Control to create a group profile for the new group.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[User Provider Control](#)

[Profile Control](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[User/Group Management JSP Tags](#)

Profile Control

The Profile control provides a convenient way to incorporate user and group profile management actions into your Page Flows, such as creating profiles for users and groups and getting profile information. For example, use this control to retrieve a user's profile, use the Property Control to put properties in working memory, then use the Rules Executor Control to evaluate and filter the user's profile properties in order to trigger actions based on that evaluation.

You can use this tag to create a standalone profile for a user in an external authentication provider that does not provide read access to its users and groups. Then, when that user logs in, the profile is automatically associated with the user.

Note: It is possible (but not recommended) to store an identical username or group name in more than one authentication provider. For example, user "foo" can reside in the default WebLogic Server LDAP provider and in an external RDBMS provider. In that case, WebLogic Portal uses only one user profile for user "foo."

Security: For profile management actions, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine whether or not the user can perform the action for users and groups. For example, to let users modify their own user profile, make sure the "Self" role is in "Roles That Can Update Users" (which is already a default setting).

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Property Control](#)

[User Provider Control](#)

[Group Provider Control](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[User/Group Management JSP Tags](#)

Property Control

The Property control provides a convenient way to incorporate user profile property management actions into your Page Flows, such as creating, setting, and getting profile properties on users and groups. For example, use the Profile control to retrieve a user's profile, use this control to put properties in working memory, then use the Rules Executor Control to evaluate and filter the user's profile properties in order to trigger actions based on that evaluation.

Security: For profile management actions, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine whether or not the user can perform the action for users and groups. For example, to let users modify their own user profile, make sure the "Self" role is in "Roles That Can Update Users" (which is already a default setting).

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Profile Control](#)

[User Provider Control](#)

[Group Provider Control](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[User/Group Management JSP Tags](#)

Rules Executor Control

The Rules Executor control gives you fine-grained control of your Page Flows using factors such as user profile values and other objects in working memory.

The control provides a convenient way to evaluate objects in working memory against the set of rules you designate. The control, which executes rule sets using the underlying rules engine, also lets you filter the results of the rule evaluation.

It is assumed you know which rules exist in the rules repository.

The `rulesetUri` property is required. This property is the path to the rule set you want to use in evaluating the objects in in working memory. The path is relative to your portal application's `META-INF/data` directory. For example, if the rule set you want to use is located in `META-INF/data/rulesets/myRuleSet.rls`, the `rulesetUri` would be `/rulesets/myRuleSet.rls`.

For information on rule sets, the rules engine, and putting objects into working memory, see [Using Rules in Portal Applications on e-docs](#).

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Rules Manager Control](#)

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

Rules Manager Control

The Rules Manager control lets you access and managing rules and rule sets for the Portal rules manager. It is intended to be used only by Portal system administrators. You can use this control as a development tool to look up rule sets you want to use with the Rules Executor Control.

Rule sets, which must be stored in your portal application's META-INF/data directory, are loaded automatically when the server starts.

Because this control requires the caller be in an authorized role, it cannot be used from a Java Web Service.

Security: The caller must be in PortalSystemAdministrator role to invoke all of the control's methods.

For information on rule sets and the rules engine, see Using Rules in Portal Applications on e-docs.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Rules Executor Control](#)

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

User Info Control

The User Info control queries information about a particular user, such as which groups the user belongs to, which roles the user is in, which roles are available to the user. To perform user maintenance such as creating and removing users and setting passwords, use the User Provider Control.

Security: For retrieving user information, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine whether or not the user can perform the action. To invoke the `getAllGroupNames` method, the caller must be in the `PortalSystemAdministrator` role.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[User Provider Control](#)

[Profile Control](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[User/Group Management JSP Tags](#)

User Login Control

The User Login Control provides a convenient way to add login and logout functionality to a Page Flow.

This control provides a login form that you can add to a JSP. An example of adding a form to a JSP is described in Using Portal Controls.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[User Provider Control](#)

[Profile Control](#)

[Implementing Authentication](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[User/Group Management JSP Tags](#)

User Provider Control

The User Provider control provides a convenient way to incorporate user management actions into your Page Flows, such as creating users, getting a list of users, and setting passwords.

Use the `atnProvider` attribute to specify which authentication provider contains the users you want to manage with the control.

Security: For user and group management actions, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine whether or not the user can perform the action.

If you use the `createUser` action, a user profile is automatically created for the user if you perform `post-user-creation-processing` with the attributes below. Otherwise, you can use the Profile Control to create a user profile for the new user; or have a profile created for the user automatically when the user next logs in.

Following are descriptions of properties on the control:

`doPostProcess`

Optional – Works in conjunction with the `fireEvent`, `login`, and `saveAnonymous` properties. If this property and the `fireEvent`, `login`, and `saveAnonymous` properties are set to true, all three events occur after user creation. If the `doPostProcess` property is set to false, the individual settings on the other three properties are used.

`fireEvent`

Optional – If set to true, a `UserRegistrationEvent` is dispatched to the event service during the `post-user-creation` process. Defaults to true. If this property is set to false, `doPostProcess` is ignored.

`login`

Optional (Boolean) – If set to true, the user is logged in during the `post-user-creation` process. Defaults to true. If true, a user profile is created automatically for the user at login. If this attribute is set to false, `doPostProcess` is ignored, and you must either create a profile for the user or have a profile created for the user automatically when the user next logs in.

`saveAnonymous`

Optional – If set to true, any properties the user may have set during the session before registering are added to the new user's properties during the `post-user-creation` process. Defaults to true. If this property is set to false, `doPostProcess` is ignored.

See *Using Portal Controls* for an example use of the User Provider control.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

Working with Java Controls

Adding Portal Controls to Java Page Flows

Using Portal Controls

Group Provider Control

Profile Control

Tutorial: Creating a Login Control Page Flow Using the Wizard

User/Group Management JSP Tags

Click Content Event Control

The Click Content Event control provides a convenient way to dispatch a `ClickContentEvent` to the event service. Use this control in a Page Flow for one of two primary purposes:

- To trigger a campaign action when the event occurs.
- To persist information about the event for use in behavior tracking and analytics.

This control may not be used in a Java Web Service, because the request and session objects it requires are unavailable from a Java Web Service.

You can obtain the Session and Request objects from a Page Flow with the following code:

```
HttpServletRequest request = this.getRequest();
```

The control's dispatch action lets you pass (`HttpServletRequest request`, `String documentType`, `String documentId`) to the event service for the clicked content.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Using Session, Request, and Event Properties in Campaigns](#)

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[Interaction Management JSP Tags](#)

Display Content Event Control

The Display Content Event control provides a convenient way to dispatch a `DisplayContentEvent` event to the event service. Use this control in a Page Flow for one of two primary purposes:

- To trigger a campaign action when the event occurs.
- To persist information about the event for use in behavior tracking and analytics.

This control may not be used in a Java Web Service, because the request and session objects it requires are unavailable from a Java Web Service.

You can obtain the Session and Request objects from a Page Flow with the following code:

```
HttpServletRequest request = this.getRequest();
```

The control's dispatch action lets you pass (`HttpServletRequest request`, `String documentType`, `String documentId`) to the event service for the displayed content.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Using Session, Request, and Event Properties in Campaigns](#)

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

[Interaction Management JSP Tags](#)

Generic Event Control

The Generic Event control provides a convenient way to dispatch a non-tracked event (whose name you specify) to the event service. A non-tracked event is not designed to be persisted (unless you have implemented your own event listener and persistence classes). Use this control in a Page Flow to trigger a campaign action when the event occurs.

For the `eventType` property, enter the name of the event you want to dispatch.

This control may not be used with a Java Web Service, because the request object it requires is unavailable from a Java Web Service.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Using Session, Request, and Event Properties in Campaigns](#)

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

Generic Tracking Control

The Generic Tracking control provides a convenient way to configure and dispatch a tracked event to the event service. Tracked events are designed to be persisted, such as in a database. Use this control in a Page Flow for one of two primary purposes:

- To trigger a campaign action when the event occurs.
- To persist information about the event for use in behavior tracking and analytics.

For the `eventType` property, enter the name of the event you want to dispatch.

This control may not be used with a Java Web Service, because the request object it requires is unavailable from a Java Web Service.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Using Session, Request, and Event Properties in Campaigns](#)

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

Rule Event Control

The Rule Event Control control provides a convenient way to dispatch a `RuleEvent` to the event service. Use this control in a Page Flow for one of two primary purposes:

- To trigger a campaign action when the event occurs.
- To persist information about the event for use in behavior tracking and analytics.

This control may not be used in a Java Web Service, because the request and session objects it requires are unavailable from a Java Web Service.

You can obtain the Session and Request objects from a Page Flow with the following code:

```
HttpServletRequest request = this.getRequest();
```

The control's dispatch action lets you pass (`HttpServletRequest request`, `String rulesetName`, `String ruleName`) to the event service.

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

[Adding Portal Controls to Java Page Flows](#)

[Using Portal Controls](#)

[Tutorial: Creating a Login Control Page Flow Using the Wizard](#)

Session Login Event Control

The Session Login Event control provides a convenient way to dispatch a SessionLoginEvent event to the event service. Use this control in a Page Flow for one of two primary purposes:

- To trigger a campaign action when the event occurs.
- To persist information about the event for use in behavior tracking and analytics.

This control may not be used in a Java Web Service, because the request and session objects it requires are unavailable from a Java Web Service.

You can obtain the Session and Request objects from a Page Flow with the following code:

```
HttpServletRequest request = this.getRequest();
```

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

Using Session, Request, and Event Properties in Campaigns

Adding Portal Controls to Java Page Flows

Using Portal Controls

Tutorial: Creating a Login Control Page Flow Using the Wizard

User Registration Event Control

The User Registration Event control provides a convenient way to dispatch a `UserRegistrationEvent` event to the event service. Use this control in a Page Flow for one of two primary purposes:

- To trigger a campaign action when the event occurs.
- To persist information about the event for use in behavior tracking and analytics.

This control may not be used in a Java Web Service, because the request and session objects it requires are unavailable from a Java Web Service.

You can obtain the Session and Request objects from a Page Flow with the following code:

```
HttpServletRequest request = this.getRequest();
```

Javadoc

For property, method, and other details on this control, see the control's Javadoc.

Related Topics

Using Session, Request, and Event Properties in Campaigns

Adding Portal Controls to Java Page Flows

Using Portal Controls

Tutorial: Creating a Login Control Page Flow Using the Wizard

Using Liquid Data Controls to Develop Workshop Applications

This chapter describes how to use the Liquid Data control in WebLogic Workshop to develop applications that use data from Liquid Data queries. Applications can use the data to display results in a web application, to use in a Web Service, to use as an input to a WebLogic Integration workflow, or in many other ways. The following topics are included:

- WebLogic Workshop and Liquid Data
- Liquid Data Control JCX File
- Creating Liquid Data Controls
- Modifying Existing Liquid Data Controls
- Using NetUI to Display Liquid Data Results
- Security Considerations With Liquid Data Controls
- Moving Your Liquid Data Control Applications to Production

WebLogic Workshop and Liquid Data

When you install Liquid Data, the installation installs the Liquid Data Control into WebLogic Workshop, which allows you to create Liquid Data Controls directly in Workshop. The Liquid Data Control allows you to very rapidly generate robust applications that use results from Liquid Data queries (for example, to display in a web application or to use in a WebLogic Integration workflow). This section describes the Liquid Data control and the applications you can create with it.

Liquid Data Control

The Liquid Data Control is installed into WebLogic Workshop when you install Liquid Data. The Liquid Data Control is a Java Control Extension that accesses the Liquid Data server to execute queries from applications developed in WebLogic Workshop. The Liquid Data Control is available with all of the other Java Controls in WebLogic Workshop (for example, the database control).

XMLBean Generation

When you create a Liquid Data control in workshop, the Liquid Data Control wizard generates XMLBean classes for each query in the control. The Liquid Data Control wizard uses the schema associated with the stored query in the Liquid Data repository to generate the structure for the XMLBean classes. The XMLBean classes provide Java methods to traverse the XML result set returned from Liquid Data.

Use With Page Flow, Web Services, Workflows

You can use the Liquid Data Control like other controls in WebLogic Workshop, and you can take advantage of Workshop features to use Liquid Data Controls in Web Services, Page Flows and Workflows. For example, you can generate a page flow from your Liquid Data control and then use the XMLBeans to bind the data returned from Liquid Data to the JSPs in your application.

Liquid Data Control JCX File

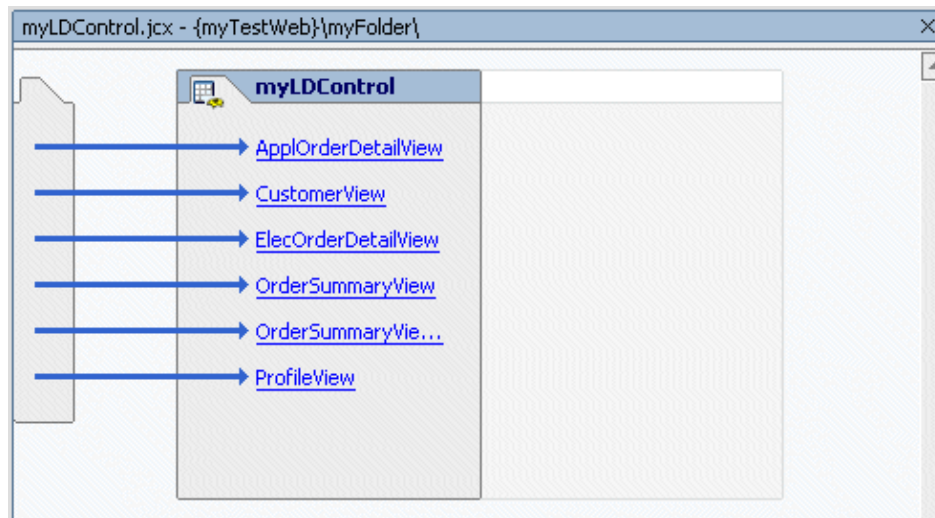
When you create a Liquid Data Control, WebLogic Workshop generates a Java Control Extension (.jcx) file. The file contains methods corresponding to the queries in which the control accesses, shows the schema files of each query as a comment, and contains a commented method which, when uncommented, allows you to pass any XQuery statement to execute an ad-hoc query. This section describes the Liquid Data Control (.jcx) file and includes the following sections:

- Design View
- Source View
- Running Ad-Hoc Queries through the Liquid Data Control

Design View

The design view of the Liquid Data Control (.jcx) file shows the available methods in a graphical view.

Figure 1–1 Design View of a Control File



With the right-click menu, you can add, modify (for example, change the query accessed by a method), rename, and delete methods. The right-click menu is context-sensitive; it displays different items if the mouse cursor is over a method, or in the control portion of the design pane.

Source View

The source view shows the source code of the Java Control Extension (.jcx) file. It includes as comments the schema used to generate the XMLBean classes for each query. The signature for each method shows the return type of the method. The return type is the XMLBean class which was generated for the schemas.

This file is a generated file and the only time you should need to edit the source code is if you want to add a method to run an ad-hoc query, as described in Running Ad-Hoc Queries through the Liquid Data Control.

The following shows the beginning part of the source code for a generated Liquid Data Control (.jcx) file. It shows the package declaration, import statements, connection properties, the schema for the

ApplOrderDetailView query, and the method that executes the ApplOrderDetailView query.

```
package myFolder;

import weblogic.jws.control.*;
import com.bea.ld.control.LDControl;

/**
 * @jc:LiquidData urlKey="myTest.myTestWeb.myFolder.anotherLDControl"
 */
public interface myLDControl extends LDControl, com.bea.control.ControlExtension
{

    /* Generated methods corresponding to stored queries.
    */

    /**
     * @return <?xml version="1.0" encoding="UTF-8"?>
     <xsd:schema attributeFormDefault="unqualified" elementFormDefault="unqualified" targetNamespace="urn:retailerType">
         <xsd:import namespace="urn:retailerType"/>
         <xsd:complexType name="ORDER_DETAIL_VIEW">
             <xsd:sequence>
                 <xsd:element maxOccurs="unbounded" minOccurs="0" ref="retailerType:ORDER_DETAIL_VIEW"/>
             </xsd:sequence>
         </xsd:complexType>
         <xsd:element name="OrderDetailView">
             <xsd:complexType>
                 <xsd:sequence>
                     <xsd:element maxOccurs="unbounded" minOccurs="0" name="ORDER_DETAIL_VIEW"/>
                 </xsd:sequence>
             </xsd:complexType>
         </xsd:element>
     </xsd:schema>
     */
    *
    * @param orderid java.lang.String
    *
    * @param custid java.lang.String
    *
    * @jc:Stored-Query Name="rtl.ApplOrderDetailView"
    */
    retailer.OrderDetailViewDocument ApplOrderDetailView(java.lang.String
        orderid, java.lang.String custid);
}
```

Running Ad-Hoc Queries through the Liquid Data Control

At the bottom of the generated Liquid Data Control (.jcx) file is a comment showing methods you can add which allow you to run an ad-hoc query through the control. To add one of these methods, uncomment the appropriate method and add a return type to the signature.

```
/**
 * Default method to execute an ad hoc query. This method can be customized
 * to have a differnt method name (e.g. runMyQuery), return a XML Bean class
 * (e.g. Customer), * or to have one or both of the following two extra
 * parameters: com.bea.ldi.server.common.QueryParameters and
 * com.bea.ldi.server.common.QueryAttributes
 * e.g. exec(String query, QueryParameters params);
 * e.g. exec(String query, QueryAttributes attrs);
 * e.g. exec(String query, QueryParameters params, QueryAttributes attrs);
 */
```

Working with Java Controls

```
*      com.bea.xml.XmlObject executeXQuery(String query); */
```

Creating Liquid Data Controls

You can create Liquid Data controls in a variety of WebLogic Workshop projects. This section includes the following procedures to create Liquid Data controls:

- General Steps to Create a Liquid Data Control
- To Create a Liquid Data Control in a Web Project
- To Create a Liquid Data Control in a Web Service Project
- To Add a Liquid Data Control to an Existing Web Service File
- To Create a Test Web Service From a Liquid Data Control

The steps are similar for creating Liquid Data controls in other types of WebLogic Workshop projects.

General Steps to Create a Liquid Data Control

This section describes the general steps for creating a Liquid Data control. For detailed steps for creating a Liquid Data control in a Web Project or in a Web Service project, see [To Create a Liquid Data Control in a Web Project](#) or [To Create a Liquid Data Control in a Web Service Project](#).

Step 1: Create a Project in an Application

Before you can create a Liquid Data control in WebLogic Workshop, you must create an application and create a project in the application. You can create a Liquid Data control in most types of Workshop projects, but the most common projects in which you create Liquid Data controls are Web Projects, Web Service Projects, Portal Web Projects, or a Process Web Projects.

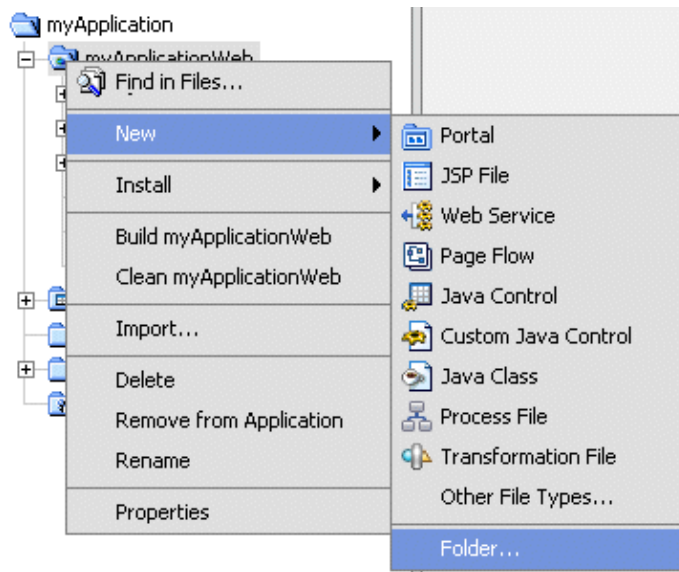
Step 2: Start Liquid Data, If It is Not Already Running

Make sure Liquid Data is running. Liquid Data can be running locally (on the same domain as WebLogic Workshop) or remote (on a different domain from workshop). If Liquid Data is not running, start up the domain in which it runs.

Step 3: Create a Folder in a Project

Create a folder in the project to hold the Liquid Data control(s). You can also create other controls (database controls, for example) in the same folder, if needed. Workshop controls cannot be created at the top-level of a project directory structure; they must be created in a folder. When you create the folder, enter a name that makes sense for your application.

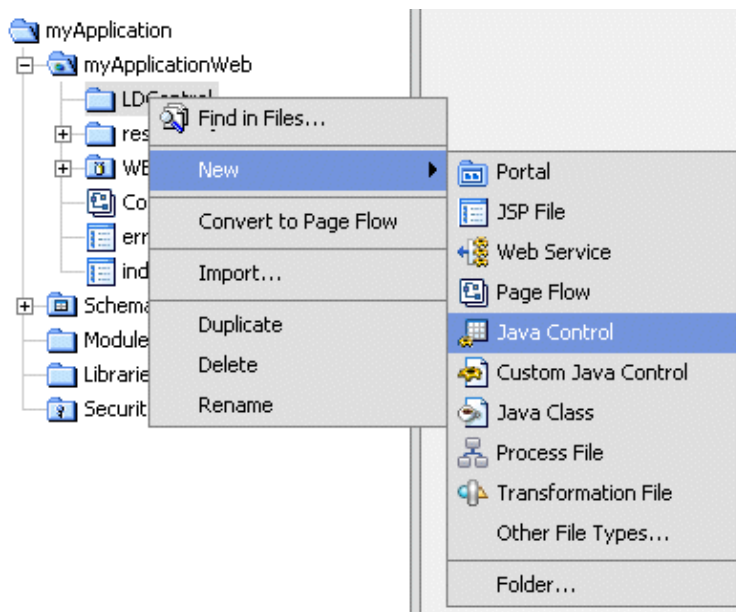
Figure 1–2 Create a New Folder in WorkshopLiquid Data



Step 4: Create the Liquid Data Control

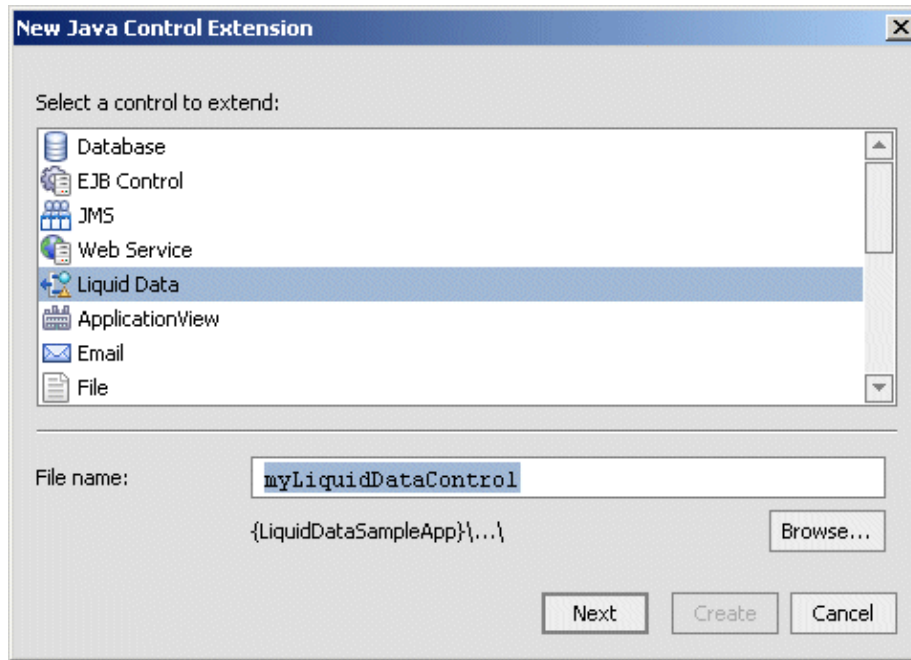
The Liquid Data Control is a Java Control Extension. To create a Liquid Data Control, start the Java Control wizard by selecting your folder within a project, right-clicking, and selecting **New > Java Control**, as shown in Figure 1–3. You can also create a control using the **File > New > Java Control** menu item.

Figure 1–3 Create a New Liquid Data Control



Then select **Liquid Data** from the **New Java Control Extension** dialog, as shown in Figure 1–4. Enter a filename for the control (.jcx) file and click **Next**.

Figure 1–4 Liquid Data Control in WebLogic Workshop



Step 5: Enter Connection Information to the Liquid Data Server

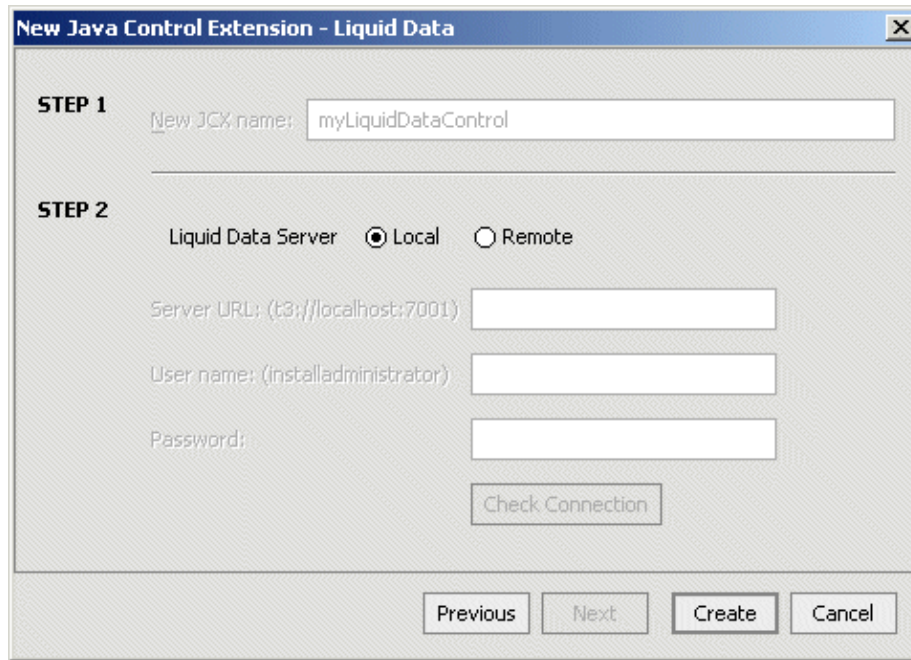
A screen similar to the one in Figure 1–5 allows you to enter connection information to your Liquid Data server. If the server is local, the Liquid Data control uses the connection information stored in the application properties (to view these settings, access the Tools > Application Properties menu item in the IDE).

If the Liquid Data server is remote, click the Remote button and fill in the appropriate server URL, user name, and password.

Note: You can specify a different username and password with which you connect to a local machine on the Liquid Data Control Wizard Connection Information dialog, too. To do this, click the Remote button and enter the connection information (with a different username and password) for your local machine. The security credentials specified through the Application Properties or through the Liquid Data Control Wizard are only used for creating the Liquid Data Control (.jcx) file, not for testing queries through the control. For more details, see Security Considerations With Liquid Data Controls.

When the information is correct, click Create to go to the next step.

Figure 1–5 Liquid Data Control Wizard Connection Information



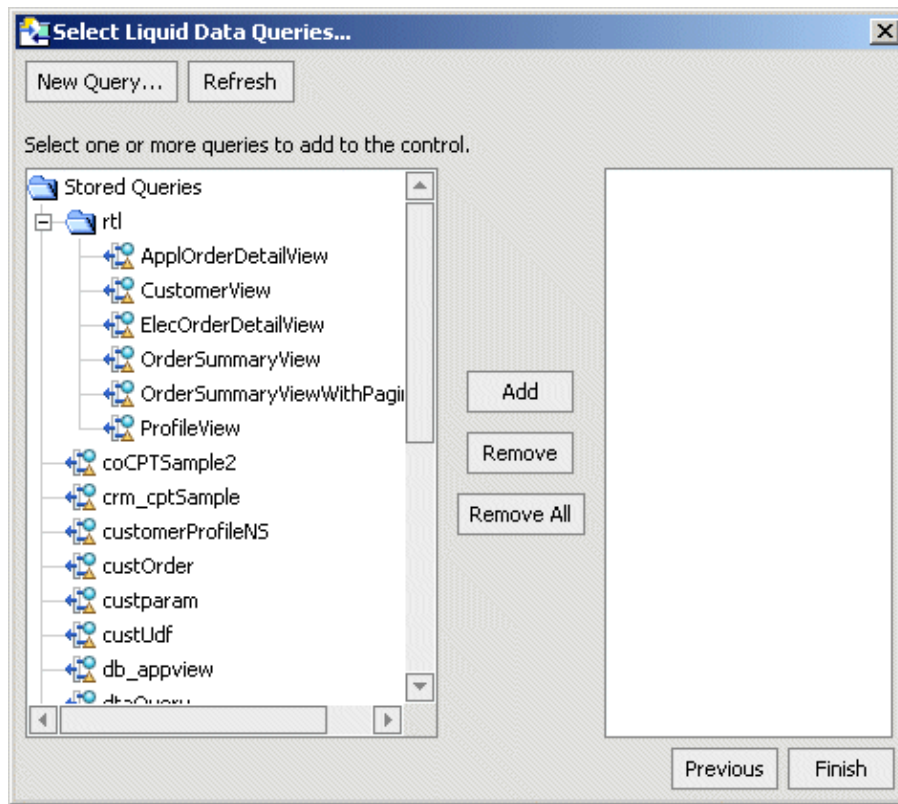
Step 6: Select Queries to Add to the Control

In the Select Liquid Data Queries screen, select queries from the left pane and click Add to add those queries to the control. If you mouse over a query, the signature of the control method for the query appears in a tooltip popup. A "fetching metadata" message appears if the signature has not yet been retrieved from the Liquid Data server.

Note: Only stored queries with a schema configured appear in the Stored Queries list. For details on configuring stored queries, see Configuring Stored Queries in the *Administration Guide*.

Select one or more queries, add them to the right pane, and click Finish. When you click Finish, the Liquid Data Control (.jcx) file is generated and XMLBean classes corresponding to the schema for each stored query in the control are generated. The XMLBeans are stored in the Libraries directory of the Workshop Application. In the Libraries directory, there is one JAR file for each Liquid Data control, with the XMLBeans included in the JAR file. The JAR files are named according to the project and directory hierarchy for the control (.jcx) file.

Figure 1–6 Liquid Data Control Wizard Select Queries



Note: The stored queries should be named according to the Naming Conventions for Stored Queries described in *Building Queries and Data Views*. If a stored query contains illegal characters (for example, a hyphen), the method generated in the Liquid Data Control (.jcx) file might be an invalid Java name, causing compilation errors. If a method name is invalid, you can change the name to make it valid.

New Query

Clicking the New Query button launches the Data View Builder. You can then use the Data View Builder to create, modify, test, and deploy new queries.

Refresh

The Refresh button updates the stored query list from the Liquid Data server. If you create and deploy a new query with the Data View Builder, click the Refresh button to display the new query in the wizard.

To Create a Liquid Data Control in a Web Project

This section describes the basic steps for creating a Liquid Data control in a new Web Project. If you are adding the control to an existing project, you might not need to perform each step (for example, creating a new project). Perform the following steps to create a Liquid Data control in a new WebLogic Workshop Web Project.

1. Make sure your Liquid Data domain is running.
2. Start WebLogic Workshop.
3. Either open an existing Workshop application or create a new application (File > New > Application).
4. Select the top-level folder of your Workshop application, right-click, and select New > Project.

5. Select Web Project as the type of project, enter a name, and click Create.
6. Create a folder in your Web Project. To create the new folder, select the project folder you just created, right-click, and select New > Folder (see Figure 1–2). Enter a name for the folder and click OK.
7. Select the new folder, right-click, and select New > Java Control (see Figure 1–3).
8. In the New Java Control Extension wizard, select Liquid Data as the control type, enter a name, and click Next (see Figure 1–4).
9. If your Liquid Data server is local to your machine, accept the default and click Create (see Figure 1–5). If Liquid Data is running on a remote server, click the Remote button, enter your connection information, test your connection, and click Create.
10. In the Edit Liquid Data Control – Select Queries screen, select any queries you want accessible to your control from the left pane and click Add to add them to the right pane (see Figure 1–6).
11. If you want to create any new queries, click the Create New Query button to launch the Data View Builder, where you can create, test, and deploy queries.
12. If you have added any Liquid Data queries, click Refresh to display the new queries.
13. After you have added all the queries you need in the wizard, click Finish.

Workshop generates the .jcx Java Control Extension file for your Liquid Data control. Each method in the .jcx file returns an XMLBean type corresponding to the stored query schema. The XMLBean classes for each query are automatically generated when you create the Liquid Data control. The XMLBean classes are stored in the Libraries directory of the Workshop Application.

To Create a Liquid Data Control in a Web Service Project

This section describes the basic steps for creating a Liquid Data control in a new Web Service. If you are adding the control to an existing Web Service, you might not need to perform each step (for example, creating a new project). Perform the following steps to create a Liquid Data control in a new WebLogic Workshop Web Service Project.

1. Make sure your Liquid Data domain is running.
2. Start WebLogic Workshop.
3. Either open an existing Workshop application or create a new application (File > New > Application).
4. Select the top-level folder of your Workshop application, right-click, and select New > Project.
5. Select Web Service Project as the type of project, enter a name, and click Create.
6. Create a folder in your Web Service Project. To create the new folder, select the project folder you just created, right-click, and select New > Folder (see Figure 1–2). Enter a name for the folder and click OK.
7. Select the new folder, right-click, and select New > Java Control (see Figure 1–3).
8. In the New Java Control Extension wizard, select Liquid Data as the control type, enter a name, and click Next (see Figure 1–4).
9. If your Liquid Data server is local to your machine, accept the default and click Create (see Figure 1–5). If Liquid Data is running on a remote server, click the Remote button, enter your connection information, test your connection, and click Create.
10. In the Edit Liquid Data Control – Select Queries screen, select any queries you want accessible to your control from the left pane and click Add to add them to the right pane (see Figure 1–6).
11. If you want to create any new queries, click the Create New Query button to launch the Data View Builder, where you can create, test, and deploy queries.
12. If you have added any Liquid Data queries, click Refresh to display the new queries.
13. After you have added all the queries you need in the wizard, click Finish.

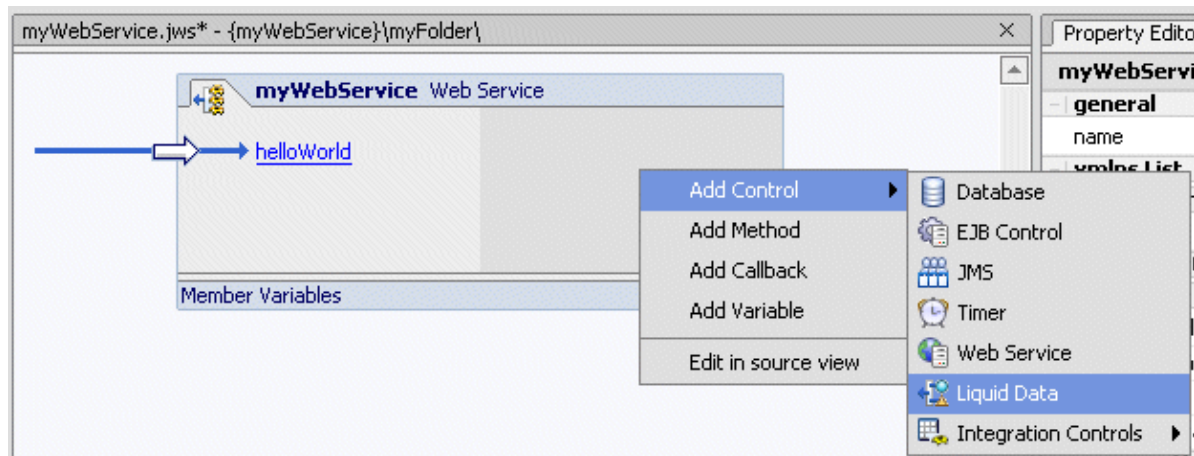
Workshop generates the .jcx Java Control Extension file for your Liquid Data control. Each method in the .jcx file returns an XMLBean of the type corresponding to the schema from the stored query. The XMLBean for each query is automatically generated when you create the Liquid Data control. The XMLBeans are stored in the Libraries directory of the Workshop Application.

To Add a Liquid Data Control to an Existing Web Service File

Perform the following steps to add a Liquid Data Control to an existing Web Service .jws file.

1. Make sure your Liquid Data domain is running.
2. In WebLogic Workshop, open an existing Web Service .jws file.
3. Click the Design View tab on the Web Service.
4. In the graphical representation of the Web Service, right-click and select Add Control > Liquid Data.

Figure 1–7 Add a Liquid Data Control to Web Service



5. In the Insert Control Wizard, enter a variable name for the control (STEP 1 in the dialog in Figure 1–8). The variable name can be any valid variable name that is unique in the Web Service.
6. In the Insert Control Wizard, either browse to an existing Liquid Data Control (it must be in the same project as the Web Service) or click the Create a New Liquid Data Control button.
7. If you want the control to be a factory, check the Make This a Control Factory button. If the control is a factory, it will create multiple instances at runtime if a query is called multiple times. Otherwise, requests to the control are serialized and each request for a given query must complete before another can begin.

Figure 1–8 Insert Control Wizard

8. If your Liquid Data server is running on a separate domain from Workshop, click remote (in STEP 3 of the Insert Control Wizard dialog). For details about specifying local or remote Liquid Data server, see Step 5: Enter Connection Information to the Liquid Data Server.
9. Click the Create button on the Insert Control Wizard.
10. If you created a new control, choose the queries for your control, as described in Step 6: Select Queries to Add to the Control.

To Create a Test Web Service From a Liquid Data Control

Perform the following steps to generate and test a web service from a Liquid Data Control.

1. Select a Liquid Data Control (.jcx) file, right-click, and select Generate Test JWS File.

Workshop generates the .jws Java Web Service file for your Liquid Data control.

2. Select your Web Service project, right-click, and select Build Project.

Workshop builds an asynchronous Web Service from the .jws file.

3. When the build is complete, double-click the .jws file to open it.

Working with Java Controls

4. On the Design View of the Web Service, notice the startTestDrive and finishTestDrive methods, as well as a method for each of the queries you specified in the Liquid Data Control wizard.
5. Click the test button (or select Debug > Start from the Workshop menu) to test the web service.
6. Click the startTestDrive button to start the conversation for the Web Service.
7. Click the Continue this Conversation link (in the left corner of the test page).
8. Enter values for any query parameters (if the query has parameters) and click the button with the name corresponding to the query you want to execute.

The Web Service executes the query and the results are returned to the test browser.

9. If you want to run the query again or run other queries in the Web Service, click Continue this Conversation, enter any needed parameters and click the button with the name corresponding to the query you want to execute.
10. To end the Web Service conversation, click the Continue this Conversation link and then click the finishTestDrive button.

Modifying Existing Liquid Data Controls

This section describes the ways you can modify an existing Liquid Data control. It contains the following procedures:

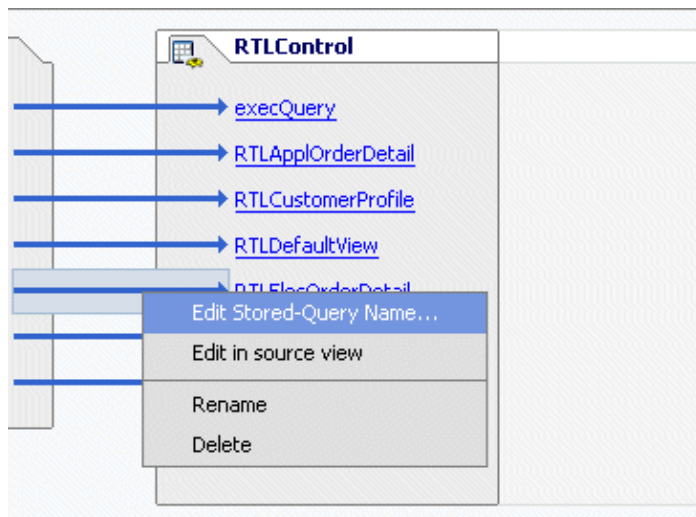
- To Change the Query Associated With a Single Control Method
- To Add a New Method to a Control
- To Invoke the Query Wizard to Modify an Existing Control
- Updating an Existing Control if Schemas Change

To Change the Query Associated With a Single Control Method

Perform the following steps to change the query that a method in a Liquid Data Control accesses.

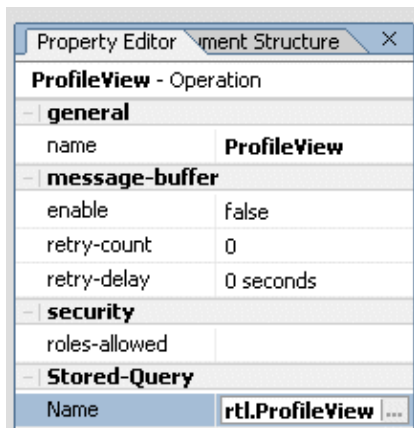
1. In WebLogic Workshop, open the Design View for a Liquid Data Control (.jcx) file.
2. Select the method you want to change, right-click, and select Edit Stored-Query Name to bring up the Liquid Data Control Wizard.

Figure 1–9 Changing the Query a Method Accesses



Note: You can also access the Liquid Data Control Wizard from the property editor

Figure 1–10 Opening the Property Editor from the Stored-Query Name Property



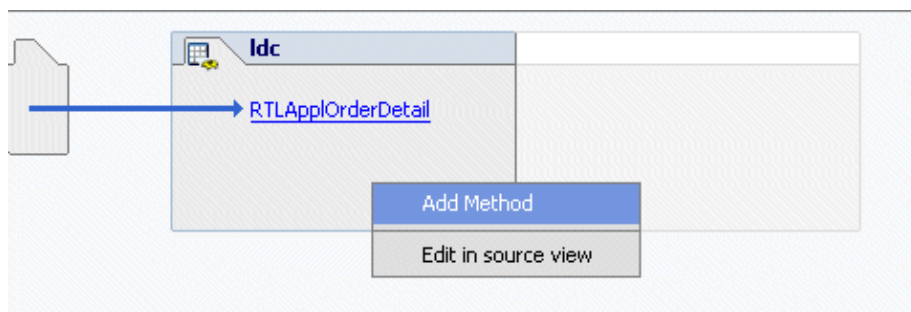
3. If you are accessing a remote Liquid Data server, enter a password on the connection information screen. If you are using a local Liquid Data server, the connection information screen does not appear.
4. In the Property editor, navigate to the query you want the method to access, select it, and click OK.

To Add a New Method to a Control

Perform the following steps to add a new method to an existing Liquid Data control.

1. In Workshop, open an existing control in Design View.
2. In the control Design View, move your mouse inside the box showing the control methods, right-click, and select Add Method, as shown in Figure 1–11.

Figure 1–11 Add a Method to a Control



3. Enter a name for the new method.
4. Move your mouse over the new method, right-click, and select Edit Stored-Query Name to launch the query wizard (see Figure 1–9).

Alternately, you can launch the query wizard from the Stored-Query name property, as shown in Figure 1–10.

5. If you are accessing a remote Liquid Data server, enter a password on the connection information screen. If you are using a local Liquid Data server, the connection information screen does not appear.
6. In the Property editor, navigate to the query you want the method to access, select it, and click OK.

To Invoke the Query Wizard to Modify an Existing Control

You can use the query wizard to modify one or more queries accessed in an existing Liquid Data Control. A query corresponds to a method in the Liquid Data Control (.jcx) file. Perform the following to invoke the Liquid Data query wizard and modify the query selection for an existing Liquid Data Control.

1. In WebLogic Workshop, open the Design View for a Liquid Data Control (.jcx) file.
2. If you are not already viewing the Property Editor, select View > Property Editor from the Workshop menu.
3. If you want to change any of the connection information, enter a value for the url or username attributes in the property editor.
4. In the Liquid Data section of the property editor, click the three dots (see Figure 1–12) to launch the query wizard.

Figure 1–12 Invoking the Query Wizard from the Workshop Property Editor



5. If you entered new values for the url or username attributes, or if the Liquid Data Server is in a remote domain, the Liquid Data Connection Information screen appears. Enter a password and click OK. If you did not change the value of these attributes, then the wizard opens to the Property Editor where you select queries.
6. Add or remove queries as you need in the Property Editor screen and click OK. For details, see Step 6: Select Queries to Add to the Control.

Updating an Existing Control if Schemas Change

If any of the schemas corresponding to any methods in a Liquid Data Control change, then you must update the Liquid Data Control to regenerate the XMLBeans for the changed schemas. Perform the following steps to update a Liquid Data Control

1. In WebLogic Workshop, open the Liquid Data Control (.jcx) file.
2. In the Liquid Data section of the property editor, click the three dots (see Figure 1–12) to invoke the Liquid Data query wizard.
3. If you are accessing a remote Liquid Data server, enter a password on the connection information screen. If you are using a local Liquid Data server, the connection information screen does not appear.
4. In the query wizard Property Editor, click OK. Workshop regenerates the Liquid Data Control and the XMLBeans for all the schemas used by queries in the control.

Using NetUI to Display Liquid Data Results

WebLogic Workshop includes NetUI, which allows you to rapidly assemble applications that display data returned from Liquid Data queries.

When you create a Liquid Data control, an XMLBean is generated for the target schema of each stored query included as a method in the control. The following sections represent the basic steps for using NetUI to display results from a Liquid Data Control:

- Generating a Page Flow From a Control
- Adding a Liquid Data Control to an Existing Page Flow
- Adding XMLBean Variables to the Page Flow
- Dragging XMLBean Variables to a JSP File

Generating a Page Flow From a Control

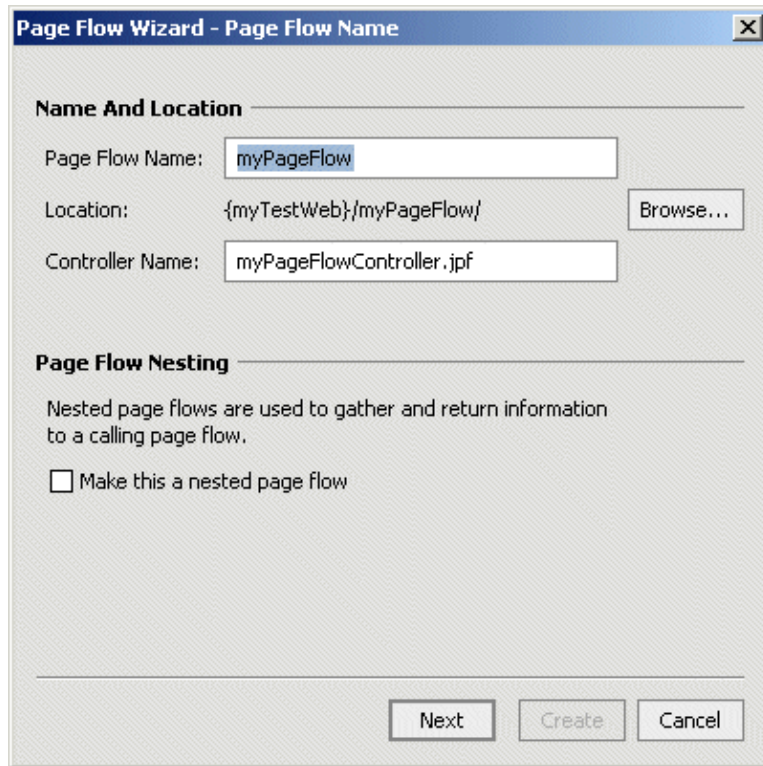
You can generate a page flow from a Liquid Data Control (.jcx) file. When you generate the page flow, Workshop creates the page flow, a start page (index.jsp), and a JSP file for each method you specify in the Page Flow wizard.

To Generate a Page Flow From a Control

Perform the following steps to generate a page flow from a Liquid Data control.

1. Select a Liquid Data Control (.jcx) file from the application file browser, right-click, and select Generate Page Flow.
2. In the Page Flow Wizard, enter a name for your Page Flow and click Next.

Figure 1–13 Enter a Name for the Page Flow



Page Flow Wizard - Page Flow Name

Name And Location

Page Flow Name:

Location:

Controller Name:

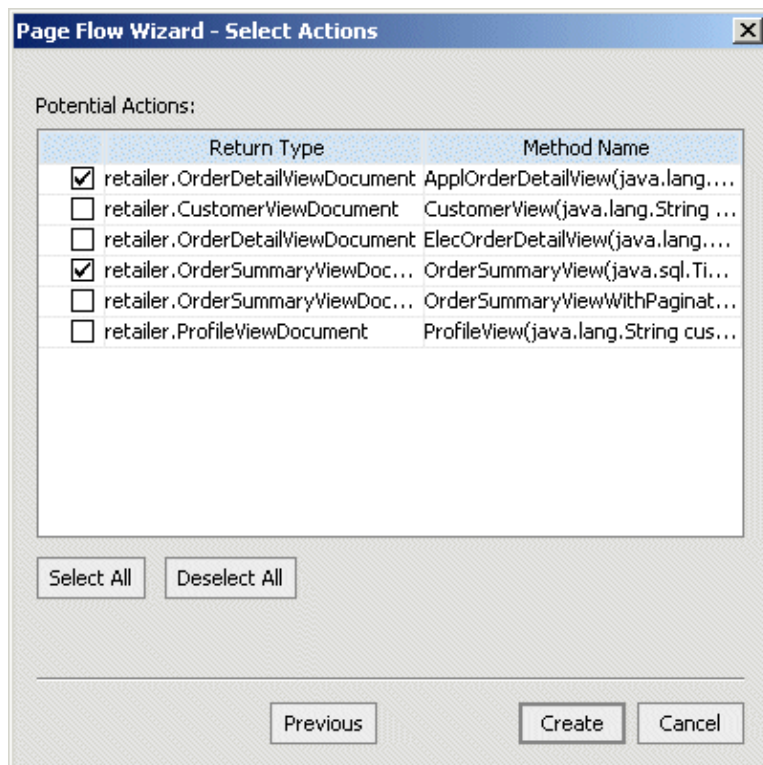
Page Flow Nesting

Nested page flows are used to gather and return information to a calling page flow.

☐ Make this a nested page flow

3. On the Page Flow Wizard – Select Actions screen, check the methods for which you want a new page created. The wizard has a check box for each method in the control.

Figure 1–14 Choose Liquid Data Methods for the Page Flow



Page Flow Wizard - Select Actions

Potential Actions:

	Return Type	Method Name
<input checked="" type="checkbox"/>	retailer.OrderDetailViewDocument	ApplOrderDetailView(java.lang....
<input type="checkbox"/>	retailer.CustomerViewDocument	CustomerView(java.lang.String ...
<input type="checkbox"/>	retailer.OrderDetailViewDocument	ElecOrderDetailView(java.lang....
<input checked="" type="checkbox"/>	retailer.OrderSummaryViewDoc...	OrderSummaryView(java.sql.Ti...
<input type="checkbox"/>	retailer.OrderSummaryViewDoc...	OrderSummaryViewWithPaginat...
<input type="checkbox"/>	retailer.ProfileViewDocument	ProfileView(java.lang.String cus...

4. Click Create.

Workshop generates the .jpf Java Page Flow file, a start page (index.jsp), and a JSP file for each method you specify in the Page Flow wizard.

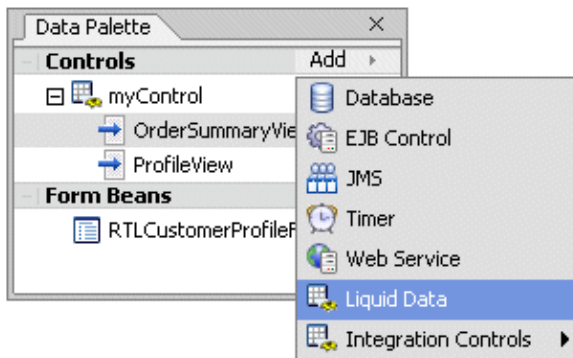
5. Add and initialize the variables to the .jpf file for the XMLBeans. For details, see Adding XMLBean Variables to the Page Flow.
6. Drag and drop the XMLBean variables to your JSPs to bind the data from Liquid Data to your page layout. For details, see Dragging XMLBean Variables to a JSP File.
7. Build and test the application in WebLogic Workshop.

Adding a Liquid Data Control to an Existing Page Flow

You can add a Liquid Data Control to an existing Page Flow .jpf file. The procedure is the same as adding a Liquid Data Control to a Web Service, described in To Add a Liquid Data Control to an Existing Web Service File, except instead of opening the Web Service in Design View, you open the Page Flow .jpf file in Action View.

You can also add a control to an existing page flow from the Page Flow Data Palette (available in Flow View and Action View of a Page Flow), as shown in Figure 1–15.

Figure 1–15 Adding a Control to a Page Flow from the Data Palette



Adding XMLBean Variables to the Page Flow

THIS SHOULD CHANGE FOR SP2

In order to use the NetUI features to drag and drop data from an XMLBean into a JSP, you must first create variables in the page flow .jpf file. NetUI can traverse the XMLBean data structure until it encounters an array. When you create the Liquid Data control and the XMLBeans are generated, the XMLBean generation defines an array for each element in the schema that is repeatable. When there is an array (corresponding to a repeatable element), you need to define a new variable to access the data in the array. You therefore need one variable from each repeatable parent node of the XML data represented in the XMLBean. Define each variable with a type corresponding to the XMLBean object of the parent node.

Define the variables in the class that extend the PageFlowController class. For example, consider the case where you are trying to display XML data of the following form:

```
<CUSTOMER>data</CUSTOMER>
```

Working with Java Controls

```
.....<PROMOTION>promotion data</PROMOTION>
.....
```

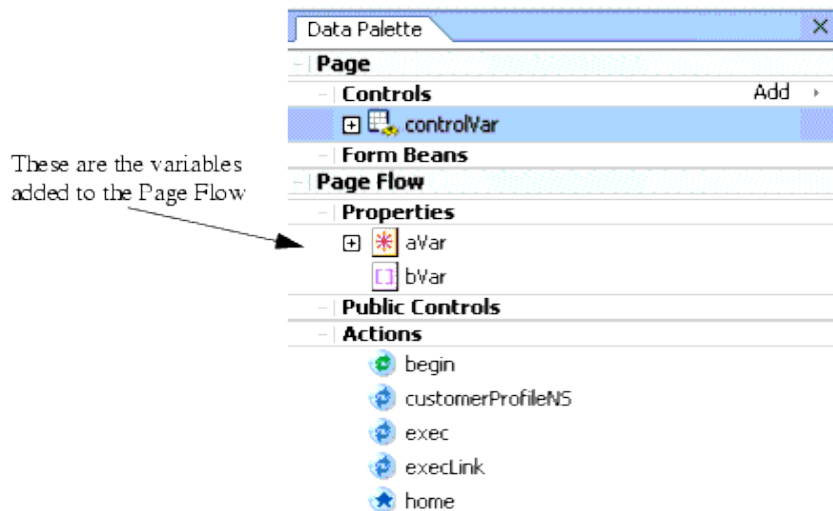
You can add the following code snippet, which shows two variables (shown in bold type) added to the page flow:

```
public class myPageFlowController extends PageFlowController
{
/**
 * This is the control used to generate this pageflow
 * @common:control
 */
private aLDControl myControl;

// add Variables with XMLBean types from the generated XMLBeans
public transient com.mycorp.crm.CUSTDocument aVar;
public transient com.mycorp.crm.CUSTDocument.PROFILE.CUST.PROMOTION[ ] bVar;
```

This code snippet declares two variables in the page flow, aVar and bVar, and these variables will display in the IDE to allow for drag-and-drop operations onto JSP files.

Figure 1–16 Page Flow Variables for XMLBean Objects



When you drag-and-drop an array onto a JSP file, the NetUI Repeater Wizard appears and guides you through selecting the data you want to display.

To Add Variables to a Page Flow

Perform the following steps to add variables of XMLBean types for your query.

1. Open your Page Flow (.jpf) file in Workshop.
2. In the variable declarations of your Page Flow class, enter variables with XMLBean types corresponding to the schema elements you want to display. Depending on your schema, what you want to display, and how many queries you are using, you might need to add several variables.
3. To determine the XMLBean type for the variables, perform the following:

Working with Java Controls

- a. In your Liquid Data control, examine the method signature for each method that corresponds to a query. The return type is the root level of the XMLBean. Create a variable of that type. For example, if the signature for a control method is as follows:

```
mySchema.CUSTOMERPROFILEDocument myQuery(java.lang.String custid);
```

create a variable as follows:

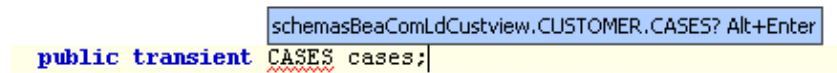
```
public transient mySchema.CUSTOMERPROFILEDocument myCustomerVar;
```

- b. If the schema has complex types and you want to display data in the complex types, examine the schema to determine the name of the complex type element and create a variable with a type corresponding to the schema element. For example, if the schema contains a complex type named CASE, create a variable as follows:

```
public transient CASES customerCases;
```

- c. Workshop might display a message asking if the type is from a particular XMLBean, as follows:

Figure 1–17 Workshop Prompting for Type Clarification



- a. Type Alt+Enter to verify the type.
 - b. Continue this process until you have declared the variables you need.
4. After you create your variables, initialize them as described in To Initialize the Variable in the Page Flow.

To Initialize the Variable in the Page Flow

You must initialize your XMLBean variables in the Page Flow. Initializing the variables ensures that the data bindings to the variables work correctly and that there are no tag exceptions when the JSP displays the results the first time.

Perform the following steps to initialize the XMLBean variables in the Page Flow:

1. Open your Page Flow (.jpf) file in Workshop.
2. In the action for the page in which you are going to display the data, add some code to initialize the variables used in the query displayed in that action.

The following sample code shows an example of initializing a variable on the Page Flow. The code (and comments) in bold is what was added. The rest of the code was generated when the Page Flow was generated from the Liquid Data control (see Generating a Page Flow From a Control).

```
/**  
 * Action encapsulating the control method :RTLCustomerProfile  
 * @jpf:action  
 * @jpf:forward name="success" path="index.jsp"
```

Working with Java Controls

```
* @jpf:catch method="exceptionHandler" type="Exception"
*/
public Forward RTLCustomerProfile( RTLCustomerProfileForm aForm )
    throws Exception
{
    schemasBeaComLdCustview.PROFILEVIEWDocument var =
        myControl.RTLCustomerProfile( aForm.custid );
    getRequest().setAttribute( "results", var );

    //initialize the profile variable to var from the above statement

    profile=var;

    // test to see if there is no data returned to handle cases where
    // the query returns successfully but has no results (when custid
    // does not exist, for example)

    if (var !=null && var.getPROFILEVIEW().getCUSTOMERPROFILEArray().length > 0)
        customerProfile=var.getPROFILEVIEW().getCUSTOMERPROFILEArray(0);

    return new Forward( "success" );
}
```

Dragging XMLBean Variables to a JSP File

Once you create and initialize your variables in the Page Flow, you can drag and drop the variables onto a JSP file. When you drag and drop an XMLBean variable onto a JSP File, Workshop displays a wizard to guide you through the process of selecting the data you want to display.

To Add a Repeater to a JSP File

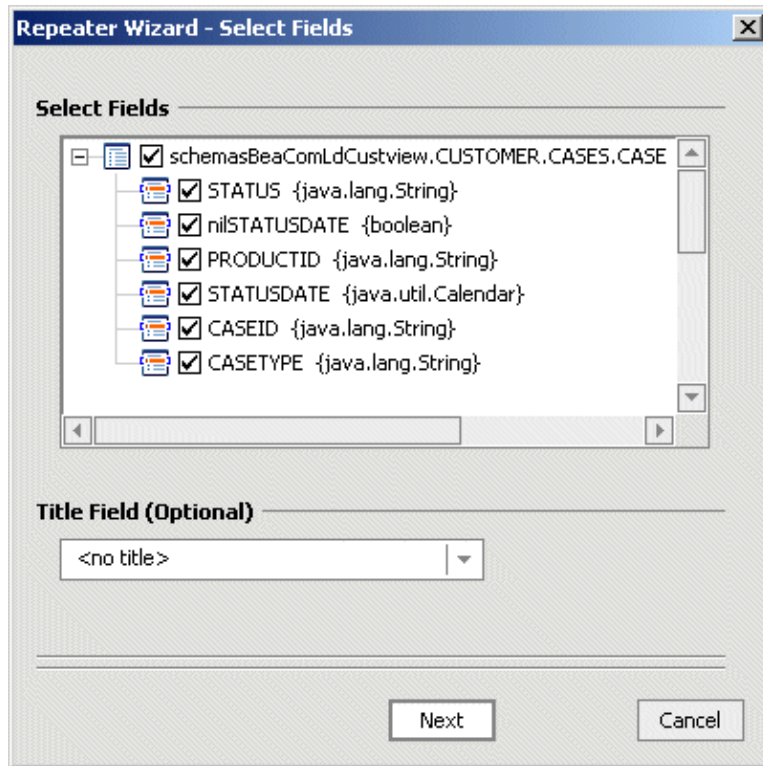
Perform the following to add a NetUI repeater tag (used to display the data from a Liquid Data query) to a JSP file.

1. Open a JSP file in your Page Flow project where you want to display data.
2. In the Data Palette > Page Flow Properties, locate the variable containing the data you want to display.
3. Expand the nodes of the variable to expose the node that contains the data you want to display. If the variable does not traverse deep enough into your schema, you will have to create another variable to expose the part of your schema you require. For details, see To Initialize the Variable in the Page Flow.
4. Select the node you want and drag and drop it onto the location of your JSP file in which you want to display the data. You can do this either in Design View or Source View.

Note: You can only drag and drop leaf nodes from the Page Flow Properties.

5. Workshop displays the repeater wizard.

Figure 1–18 Repeater Wizard

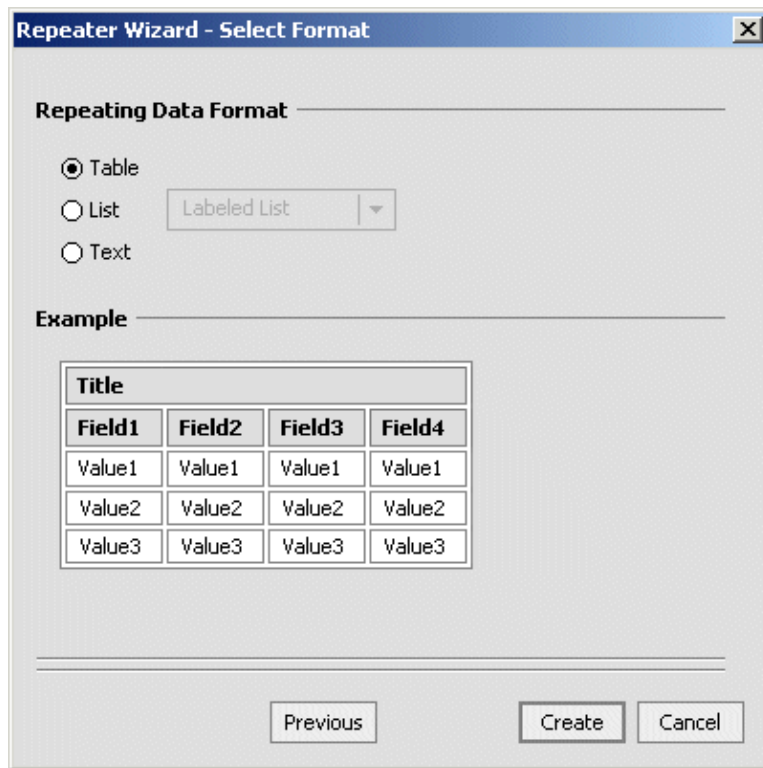


6. In the repeater wizard, navigate to the data you want to display and uncheck any fields that you do not want to display. There might be multiple levels in the repeater tag, depending on your schema.

Note: The repeater wizard displays everything contained in the XMLBean, which includes more than the data fields from your schema. Be sure to uncheck the fields you do not need (for example, nilSTATUSDATE).

7. Click Next. The Select Format screen appears.

Figure 1–19 Repeater Wizard Select Format Screen



8. Choose the format to display your data in and click Create.

Workshop generates the layout for your data.

9. Click the test button to see your data display.

To Add a Nested Level to an Existing Repeater

You can create repeater tags inside of other repeater tags. You can display nested repeaters on the same page (in nested tables, for example) or you can set up Page Flow actions to display the nested level on another page (with a link, for example).

Perform the following steps to create a nested repeater tag.

1. Add a repeater tag as described in To Add a Repeater to a JSP File.
2. Add a column to the table where you want to add the nested level.
3. Drag and drop the array from your variable corresponding to your nested level into the data cell you created in the table.
4. In the repeater wizard, select the items you want to display.
5. Click the test button to test the application.

To Add Code to Handle Null Values

Perform the following steps to add code in your JSP file to handle null values for your data. It is a common JSP design pattern to add conditional code to handle null values. If you do not handle null values, your page will display errors for parameterized queries until values are entered for the parameters.

1. Add a repeater tag as described in To Add a Repeater to a JSP File.

2. Open the JSP file in source view.
3. Find the `netui-data:repeater` tag in the JSP file.
4. Add code before and after the tag to test for null values. The following is sample code. The code in bold is added, the rest is generated by the repeater wizard. This code uses the profile variable initialized in To Initialize the Variable in the Page Flow.

```

<%
PageFlowController pageFlow = PageFlowUtils.getCurrentPageFlow(request);
if ( ((pF2Controller)pageFlow).profile == null
    || ((pF2Controller)pageFlow).profile.getPROFILEVIEW().getCUSTOMERPROFILEArray() == null
    || ((pF2Controller)pageFlow).profile.getPROFILEVIEW().getCUSTOMERPROFILEArray().length
    == 0 ) {
    <p>No data</p>
} else {
<netui-data:repeater dataSource=
    "{pageFlow.profile.PROFILEVIEW.CUSTOMERPROFILEArray}">
    <netui-data:repeaterHeader>
        <table cellpadding="2" border="1" class="tablebody" >
        <tr>
<!-- the rest of the table and NetUI code goes here -->
<td><netui:label value
   ="{container.item.PROFILE.DEFAULTSHIPMETHOD}"></netui:label></td>
        </tr>
    </netui-data:repeaterItem>
    <netui-data:repeaterFooter></table></netui-data:repeaterFooter>
</netui-data:repeater>
    }
}
%>

```

5. Test the application.

Security Considerations With Liquid Data Controls

This section describes security considerations to be aware of when developing applications using Liquid Data controls. The following sections are included:

- Security Credentials Used to Create Liquid Data Control
- Testing Controls With the Run-As Property in the JWS File
- Trusted Domains

Security Credentials Used to Create Liquid Data Control

The WebLogic Workshop Application Properties (Tools > Application Properties) allow you to set the connection information to connect to the domain in which you are running. You can either use the connection information specified in the domain boot.properties file or override that information with a specified username and password.

When you create a Liquid Data Control (.jcx) file and are connecting to a local Liquid Data server (Liquid Data on the same domain as Workshop), the user specified in the Application Properties is used to connect to the Liquid Data server. When you create a Liquid Data Control and are connecting to a remote Liquid Data server (Liquid Data on a different domain from Workshop), you specify the connection information in the Liquid Data Control Wizard Connection information dialog (see Figure 1–5).

When you create a Liquid Data Control, the Control Wizard displays all queries to which the specified user has access privileges. The access privileges are defined by any security policies set on the queries, either directly or indirectly.

Note: The security credentials specified through the Application Properties or through the Liquid Data Control Wizard are only used for creating the Liquid Data Control (.jcx) file, not for testing queries through the control. To test a query through the control, you must get the user credentials either through the application (from a login page, for example) or by using the run-as property in the Web Service file.

Testing Controls With the Run-As Property in the JWS File

For testing, you can use the run-as property to test a control running as a specified user. To set the run-as property in a Web Service, open the Web Service and enter a user for the run-as property in the WebLogic Workshop property editor. Queries run through a Liquid Data Control used by the Web Service

When a query is run from an application, the application must have a mechanism for getting the security credential. The credential can come from a login screen, it can be hard-coded in the application, or it can be imbedded in a J2EE component (for example, using the run-as property in a .jws Web Service file).

Note: The Liquid Data Control property editor shows a run-as property, but the run-as property in the Liquid Data Control does not cause the Liquid Data Control to run as the specified user. If you want to use this feature, you must specify the run-as property in the .jws file, not in the .jcx file.

Trusted Domains

If the Liquid Data server is on different domain from WebLogic Workshop, then both domains must be set up as trusted domains. This is true even if security is not enabled on Liquid Data.

Domains are considered trusted domains if they share the same security credentials. With trusted domains, a user that exists on one domain need not be authenticated on the other domain (as long as the user exists on both domains).

Note: After configuring domains as trusted, you must restart the domains before the trusted configuration takes effect.

To Configure Trusted Domains

Perform the following steps to configure domains as a trusted:

1. Log into the WebLogic Administration Console as an administrator.
2. Click the node corresponding to your domain.
3. At the bottom of the General tab for the domain configuration, click the link labeled "View Domain-wide Security Links."
4. Click the Advanced tab.

Figure 1–20 Setting up Trusted Domains

liquiddata> Domain Wide Security Settings

Connected to : localhost :7001 | You are logged in as : ldsystem | [Logout](#)

Configuration | **Compatibility**

General | **Advanced** | **Filter** | **Embedded LDAP**

This page allows you to define the advanced security settings for this WebLogic Server domain.

☐ **Enable Generated Credential**

Specifies whether a credential (usually a password) should be generated for this WebLogic Server domain. (This credential is used to enable a trust relationship between two domains. For the two domains to establish trust, they must have the same credential.)

Credential:

Confirm Credential:

The credential for this WebLogic Server domain.

5. Uncheck the Enable Generated Credential box, enter and confirm a credential (usually a password), and click Apply.
6. Repeat this procedure for all of the domains you want to set up as trusted. The credential must be the same on each domain.

Working with Java Controls

For more details on WebLogic security, see [Configuring Security for a WebLogic Domain](#) in the [WebLogic Server](#) documentation.

Moving Your Liquid Data Control Applications to Production

When you move any Liquid Data deployment from development to production, you must move Liquid Data and WebLogic Server resources (JDBC Connection Pools, Liquid Data Data Sources, the Liquid Data repository, and so on) from the development environment to the production environment. For details about deploying Liquid Data, see the Liquid Data Deployment Guide.

For applications that use Liquid Data controls, you must also deploy and update the `ldcontrol.properties` file, which contains connection information for Liquid Data controls. This section describes the development to production lifecycle and provides the basic steps for moving an application containing Liquid Data controls from development to production. The following sections are included:

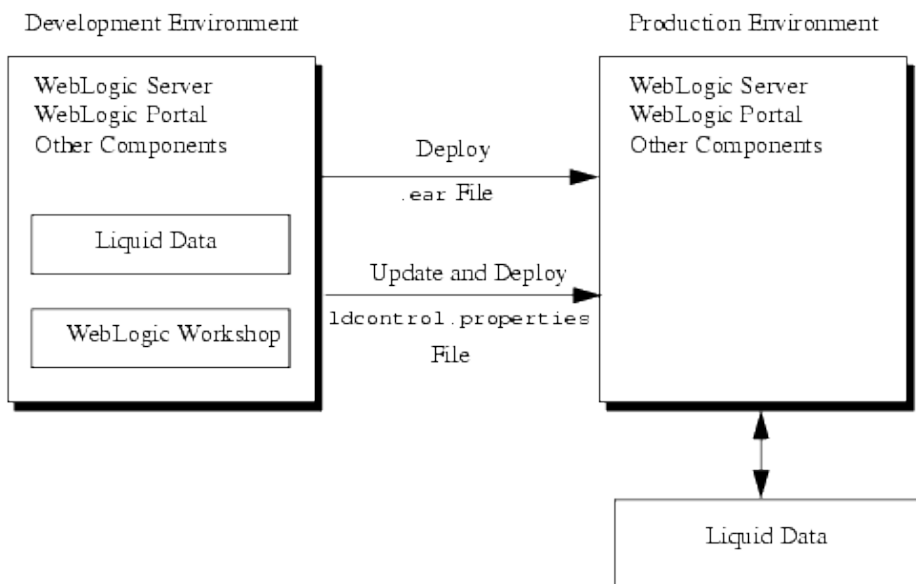
- Development to Production Lifecycle Architecture
- Remove `ldcontrol.jar` and `LDS-client.jar` From Production CLASSPATH
- Steps For Deploying to Production

Development to Production Lifecycle Architecture

In a typical development scenario, you will develop your applications in one environment and then deploy them in another. There are two main artifacts that you need to deploy on the production environment:

- Packaging Liquid Data JAR Files in Application `.ear` Files
- Liquid Data `ldcontrol.properties` File

Figure 1–21 Development to Production Lifecycle



Packaging Liquid Data JAR Files in Application .ear Files

After you have developed and tested your application using WebLogic Workshop in your development environment, you must create a .ear file for deployment to your production server(s). If you are deploying the application to a domain that does not include Liquid Data, you must add the `ldcontrol.jar` and `LDS-client.jar` files to your application library before creating the .ear file.

This section describes the following procedures:

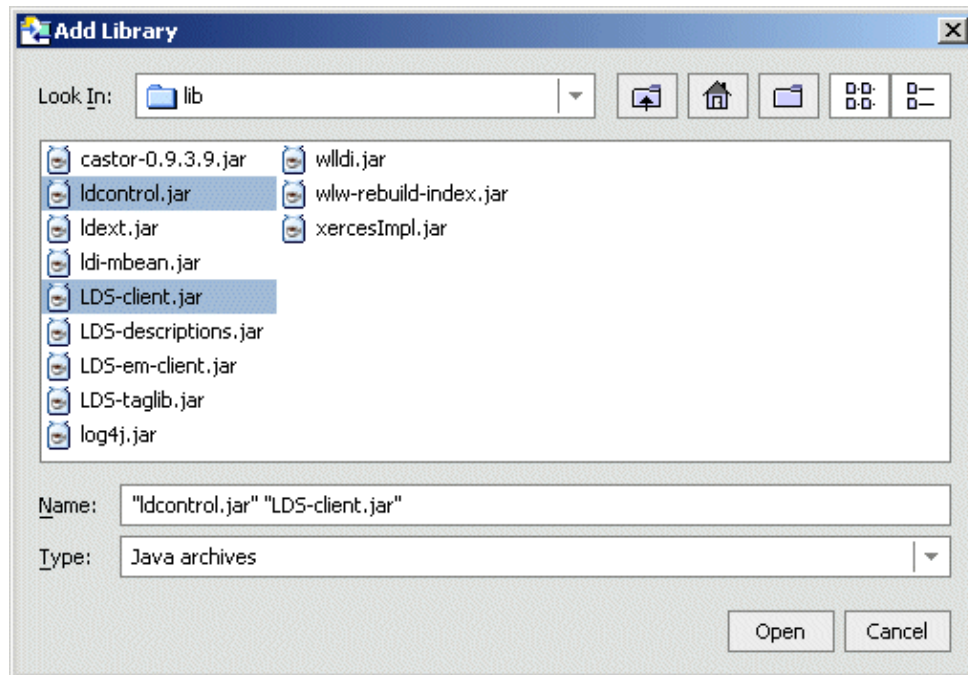
- To Add the `ldcontrol.jar` and `LDS-client.jar` Files to the Application Library
- To Generate the .ear File in Workshop

To Add the `ldcontrol.jar` and `LDS-client.jar` Files to the Application Library

Perform the following steps to add the Liquid Data .jar files to your application:

1. Open your application in WebLogic Workshop, if it is not already opened.
2. Select the Libraries directory, right-click, and select Add Library (or select File > Import Library from the Workshop menu).
3. In the Add Library dialog, navigate to the `BEA_HOME/weblogic81/liquiddata/server/lib` directory and select the `ldcontrol.jar` and `LDS-client.jar` files.

Figure 1–22 Add `ldcontrol.jar` and `LDS-client.jar` to Application Library



4. Click Open to add the `ldcontrol.jar` and `LDS-client.jar` files to your application library.

To Generate the .ear File in Workshop

Perform the following steps to generate an enterprise archive file (.ear) in WebLogic Workshop:

Working with Java Controls

1. Open your application in WebLogic Workshop, if it is not already opened.
2. Add the Liquid Data client jar files to your application library as described in To Add the ldcontrol.jar and LDS-client.jar Files to the Application Library.
3. Select Build >Build EAR from the Workshop menu.

When the build is complete, WebLogic Workshop lists the .ear file location in the Build window.

Liquid Data ldcontrol.properties File

Each domain that runs Liquid Data Control applications has a *single* ldcontrol.properties file, which stores the connection information for *all* Liquid Data Control applications running in the domain. The ldcontrol.properties file is located at the root directory of your domain where the Liquid Data Control application .ear file is deployed. There is an entry in the ldcontrol.properties file for each control you have created in each application.

The entries in the ldcontrol.properties file are of the following form:

```
AppName.ProjectName.FolderName.jcxName=t3\://hostname\:port
```

where:

Name	Description
AppName	The name of the WebLogic Workshop application.
ProjectName	The name of the WebLogic Workshop Project which contains the Liquid Data Control.
FolderName	The name of the folder which contains the Liquid Data Control.
jcxCName	The name of the Liquid Data Control file (without the .jcx extension). For example, if the control file is named myLDControl.jcx, the entry in this file is myLDControl.
hostname	The hostname or IP address of the Liquid Data Server for this control.
port	The port number for the Liquid Data Server for this control.

Note: The colons (:) in the URL must be escaped with a backslash (\) character.

If the URL value is missing, the Liquid Data Control uses the connection information from the domain config.xml file.

The following is a sample ldcontrol.properties file.

```
#Fri Jul 11 15:30:36 PDT 2003
myTest.myTestWeb.myFolder.Untitled=t3\:myLDServer\:7001
myTest.myTestWeb.myFolder.myControl=
SampleApp.LiquidDataSampleApp.Controls.RTLControl=t3\:myLDServer\:7001
SampleApp.Untitled.NewFolder.Untitled=t3\:yourLDServer\:7001
testnew.Untitled.NewFolder.ldc=
test.testWeb.NewFolder.Untitled=
```

Remove ldcontrol.jar and LDS-client.jar From Production CLASSPATH

When you use the Domain Configuration Wizard to create a Liquid Data domain, the domain includes the `ldcontrol.jar` and `LDS-client.jar` files in the server CLASSPATH. These are required for a development environment. For a production environment, however, these two .jar files should not be in the server CLASSPATH. Instead, package the `ldcontrol.jar` and `LDS-client.jar` files in the .ear file, as described in [To Add the ldcontrol.jar and LDS-client.jar Files to the Application Library](#).

If the `ldcontrol.jar` and `LDS-client.jar` files are in the server CLASSPATH, you will need to restart the server in order for the read changes to the `ldcontrol.properties` file. If the `ldcontrol.jar` and `LDS-client.jar` files are not in the server CLASSPATH and you include the files in your application library, then changes to the `ldcontrol.properties` file are reloaded when you redeploy the application. You should therefore remove these files from the server CLASSPATH in the startup file for your production domain.

Steps For Deploying to Production

This section describes the following basic steps for moving an application from development to production:

- Step 1: Add `ldcontrol.jar` and `LDS-client.jar` files to Application Library
- Step 2: Generate Enterprise Application Archive (.ear) in Workshop
- Step 3: Merge `ldcontrol.properties` File entries to Production Server
- Step 4: Deploy Enterprise Application Archive (.ear) on Production Server

Step 1: Add `ldcontrol.jar` and `LDS-client.jar` files to Application Library

Add the `ldcontrol.jar` and `LDS-client.jar` files (from the `BEA_HOME/weblogic81/liquiddata/server/lib` directory) to the Libraries directory in your application. For details of how to do this, see [To Add the ldcontrol.jar and LDS-client.jar Files to the Application Library](#).

Note: If you used the Configuration Wizard to create a Liquid Data-enabled domain, you do not have to do this, but you lose the flexibility of the `ldcontrol.properties` file being automatically reloaded when you deploy/undeploy an application. In this case, you should remove these files from the server CLASSPATH (see [Remove ldcontrol.jar and LDS-client.jar From Production CLASSPATH](#) for more details).

Step 2: Generate Enterprise Application Archive (.ear) in Workshop

Use WebLogic Workshop to generate the .ear file for your application as described in [To Generate the .ear File in Workshop](#).

Step 3: Merge `ldcontrol.properties` File entries to Production Server

Merge the `ldcontrol.properties` file from the root level of your development domain with the `ldcontrol.properties` file in the root level of the production domain. If the `ldcontrol.properties` file does not exist in the production domain, copy it from your development domain.

You must also update the URLs in each entry of the file to reference the production Liquid Data servers. For details on and for the syntax of the `ldcontrol.properties` file, see [Liquid Data ldcontrol.properties File](#).

Step 4: Deploy Enterprise Application Archive (.ear) on Production Server

Deploy your enterprise archive (.ear) file on the production WebLogic Server. You deploy the .ear file from

Working with Java Controls

the *domain* > Deployments > Applications node of the WebLogic Server Administration Console. The .ear file must be accessible from the filesystem in which the WebLogic administration server is running. For details on deploying .ear files, see Deploying WebLogic Server Applications from the WebLogic Server documentation.