Working with Beehive Controls

BEA Workshop for WebLogic Platform incorporates Beehive controls that make it easy for you to encapsulate business logic and to access enterprise resources such as databases, web services, EJBs, JMS message queues, and legacy applications.

Current Release Information:

- What's New
- Upgrading to 10.0

Useful Links:

- Tutorials
- Tips and Tricks

Other Resources:

- Online Docs
- Dev2Dev
- Discussion Forums
- Development Blogs

Topics Included in This Section

Getting Started with Beehive Controls

Provides an overview of Beehive controls.

Tutorial: Creating a Web Service with Timer Control

Walks you through creating a web service that uses the timer control.

Tutorial: Testing Controls with JUnit

Shows you how to create a control and test it using JUnit.

Using Controls

Describes how to use existing controls.

Using System Controls

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

Developing Custom Controls

Explains how to develop your own custom controls and share them with others.

Control Dialogs

These topics explain the control related UI dialogs and wizards.

Related Topics

Workshop for WebLogic Platform Documentation

Timer Control

Custom Controls

Apache Beehive Documentation

EJB Control Developer's Guide

Jdbc Control Developer's Guide

The JMS Control Developer's Guide

Controls: Getting Started

Getting Started with Beehive Controls

Controls provide a convenient way for your applications to access resources and encapsulate application logic.

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

What Are Controls?

Control Types: System and Custom Controls

Control Authoring Models

What Are Controls?

Controls are reusable components you can use (almost) anywhere within a enterprise application. You can use the system controls provided with Workshop for WebLogic Platform, or you can create your own.

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

- Provide access to resources such as databases or other resources.
- Encapsulate logic for reuse in other applications.
- Modularize logic you want to keep separate from other application code.

Setting Control Properties with Annotations. Controls take advantage of Java 5 annotations for setting control properties. The system controls have a rich set of properties that are parameterized through annotation settings. As for custom controls, the control author can define annotation parameterization for any set of control properties.

Apache Beehive Control Framework. Controls are built on the Apache Beehive Control framework. For more information, see Controls: Getting Started in the Apache Beehive documentation.

Note: Controls that require assembly are not currently supported within an EJB.

Control Types: System and Custom Controls

Controls are divided into two main types: system controls and custom controls.

System controls provide you a way to connect to common application resources, such as databases, EJBs, JMS queues, web services, and so on. These controls require little or no modification to use in your application.

Workshop for WebLogic Platform provides several system controls, mainly designed to provide access to enterprise 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 system controls, see Using System Controls.

Note that the System controls fall into two groups.

One group represents unmodified Beehive controls. These controls are:

- JDBC
- JMS
- EJB

The other group represents controls that are provided by BEA or some other vendor, based on the Beehive control framework. These controls are:

- Timer
- Service Control

Custom controls provide a way to fully customize access to a resource or encapsulate some application functionality. You can design a custom control to do any task in an application.

You can build your own custom controls that are based on the same framework on which system controls are based. You design a custom control from the ground up, designing its interface and implementation, 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 Controls.

Control Authoring Models

There are three kinds of authoring and usage models for Beehive controls.

1. Ground-up authoring. On this model you author the control interface and implementation from scratch. The author also defines what control properties are accessible through annotations. This authoring model applies only to custom controls.
2. Extension authoring. On this model, the core control classes already exists. You add to the

base functionality of the control by extending the base control interface class. On this model, the control user may also set control properties through annotation property setters defined by the control author. Applies to all system controls except the timer control.

3. No authoring/unmodified usage. On this model, you simply import the control and call its methods directly, without authoring a new extension class. Applies to the timer control.

Related Topics

Tutorial: Accessing a Database from a Web Application

Tutorial: Creating a Web Service with Timer Control

In this mini-tutorial, you will create and test a web service with a timer control. This is an advanced tutorial and we assume that you know the basics of Workshop for WebLogic, including how to create workspaces, projects and packages and how to run a web service and test operations with the Test Client. If you are new to Workshop for WebLogic, we recommend that you complete Tutorial: Getting Started and Tutorial: Web Service before beginning this tutorial.

> **Note:** This tutorial requests that you create a new workspace; if you already have a workspace open, this will restart the IDE. Before beginning, you might want to launch help in standalone mode to avoid an interruption the restart could cause, then locate this topic in the new browser. See Using Help in a Standalone Mode for more information.

The tasks in this step are:

- Create the Web Service
- Set up Web Service and Timer Control Annotations
- Test the Web Service / Timer Control

In this tutorial, we will create a web service that declares/instantiates a timer control (an instance of the `com.bea.control.TimerControl` base class) that calls back the client web service every two seconds. For this example, we will have two web service operations (web methods):

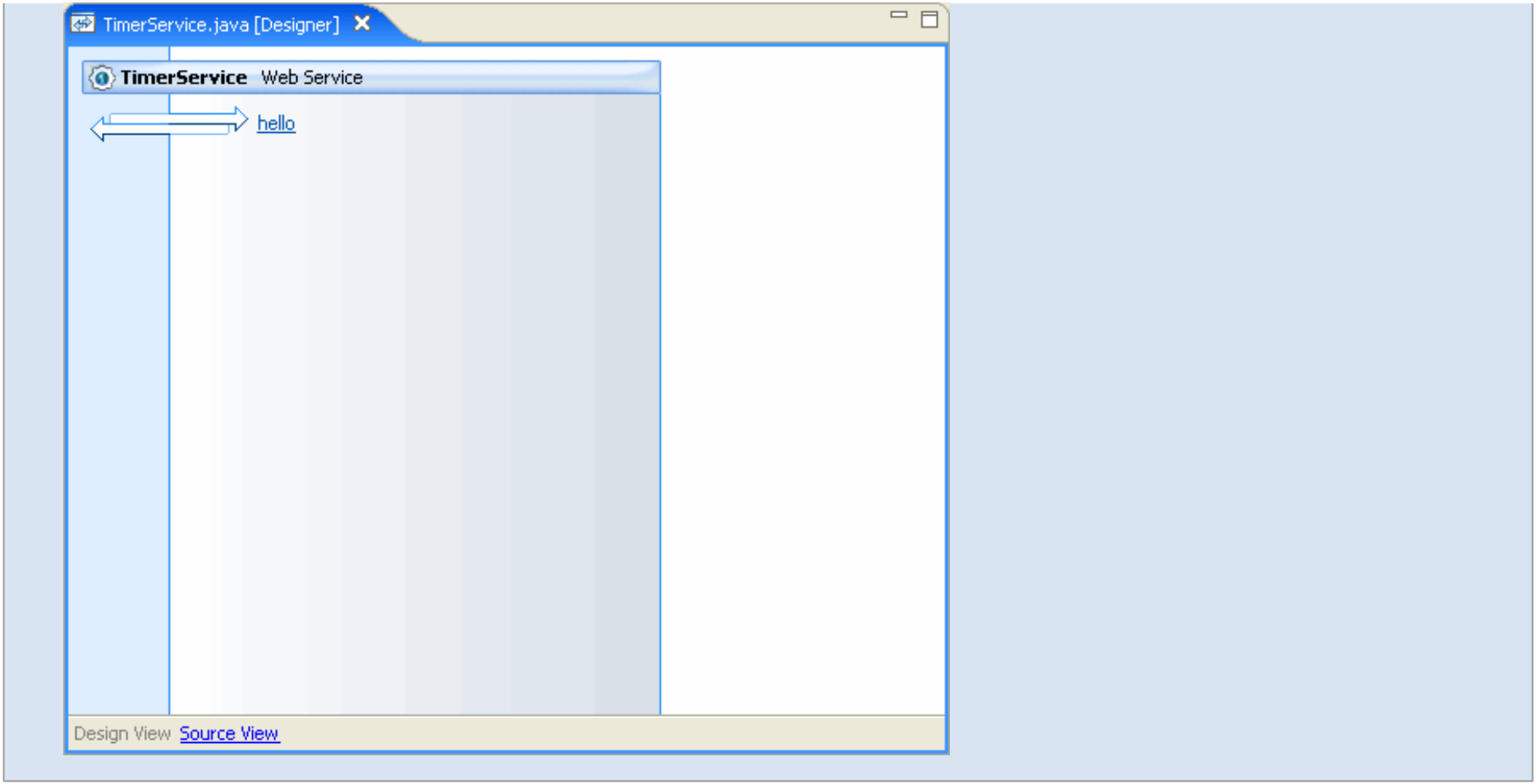
- **start** will start the timer control; once started the timer control will call back the web service every two seconds
- **stop** will stop the timer control

When the callback is received, the sample program will simply display text in the console window. Normally some programming logic would be inserted in the callback routine to perform some appropriate action.

To Create the Web Service

A timer control can only be created within a conversational (not stateless) web service. To create the web service for this tutorial:

1. Launch Workshop for WebLogic and create a new, empty workspace.
2. Create an EAR project with **File > New > Project > J2EE > Enterprise Application Project**.
3. Define a new domain and server.
4. Click **File > New > Project**. Expand **Web Services** and double click on **Web Service Project**. Specify the project name. Click **Add project to an EAR** and choose your EAR project from the pulldown. Click **Finish**.
5. Create a package for your web service by right clicking on the **Java Resources/src** folder of your web service project in the **Project Explorer** view at the left and clicking **New > Package**. Specify the package name and click **Finish.**
6. Create a web service within the package by right clicking on the package in the **Project Explorer** view and clicking **New > WebLogic Web Service**. Specify the web service name as **TimerService** and click **Finish**. The web service designer displays the new, empty web service with a single default method.



To Set up Web Service and Timer Control Annotations

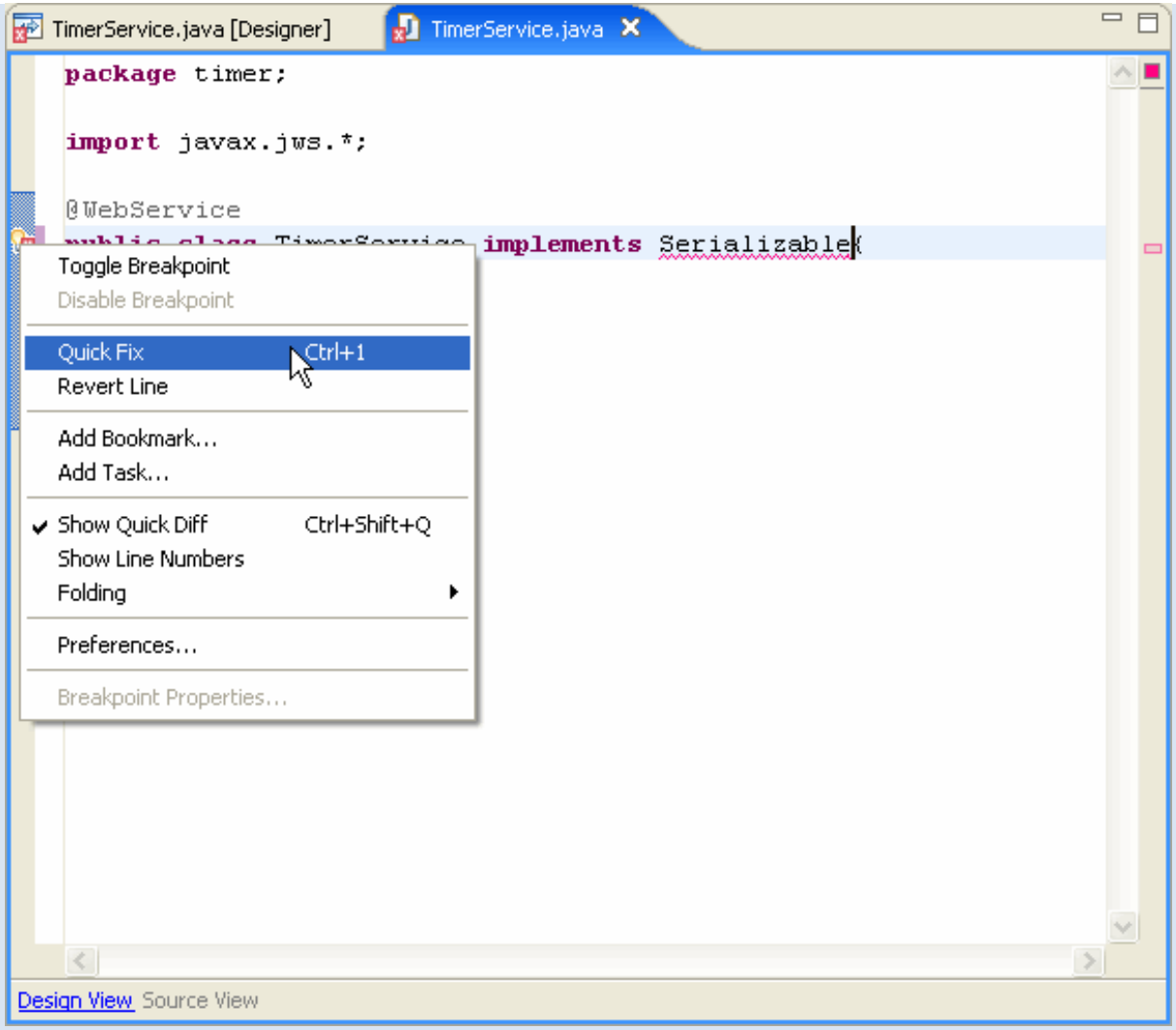
At this point, you have a project containing a package with a web service with the **Web Service Design View (Designer pane)** displayed in the editor pane.

Conversational web services must implement the java.io.Serializable interface. To set this in your web service:

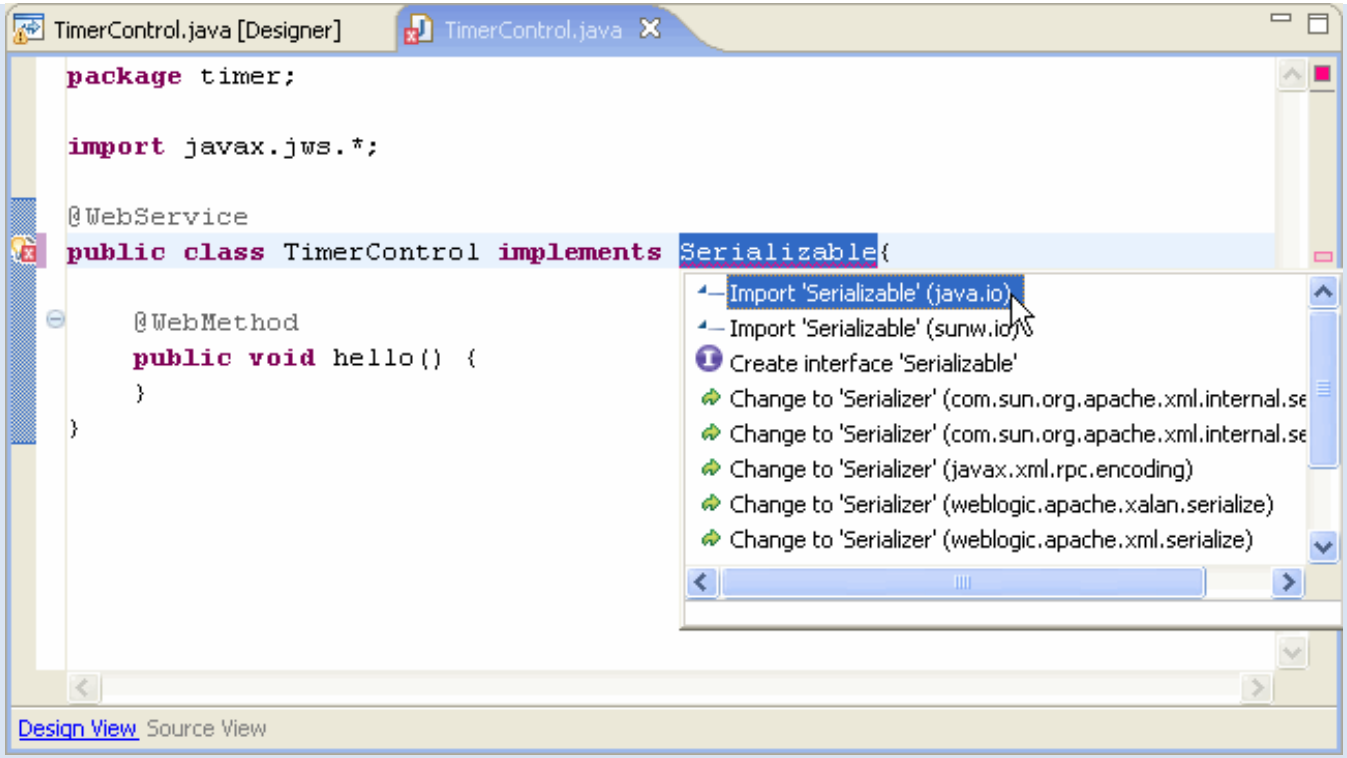
1. Edit the source file for your web service by right clicking on the designer and choosing **Edit Source**. On the class definition line for your web service, insert `implements Serializable` so that the class definition looks something like this:

   ```
   public class TimerService implements Serializable {
   ```

   Note that an error marker has appeared in the marker bar on the class declaration line. Right click on the error marker and choose **Quick Fix**.



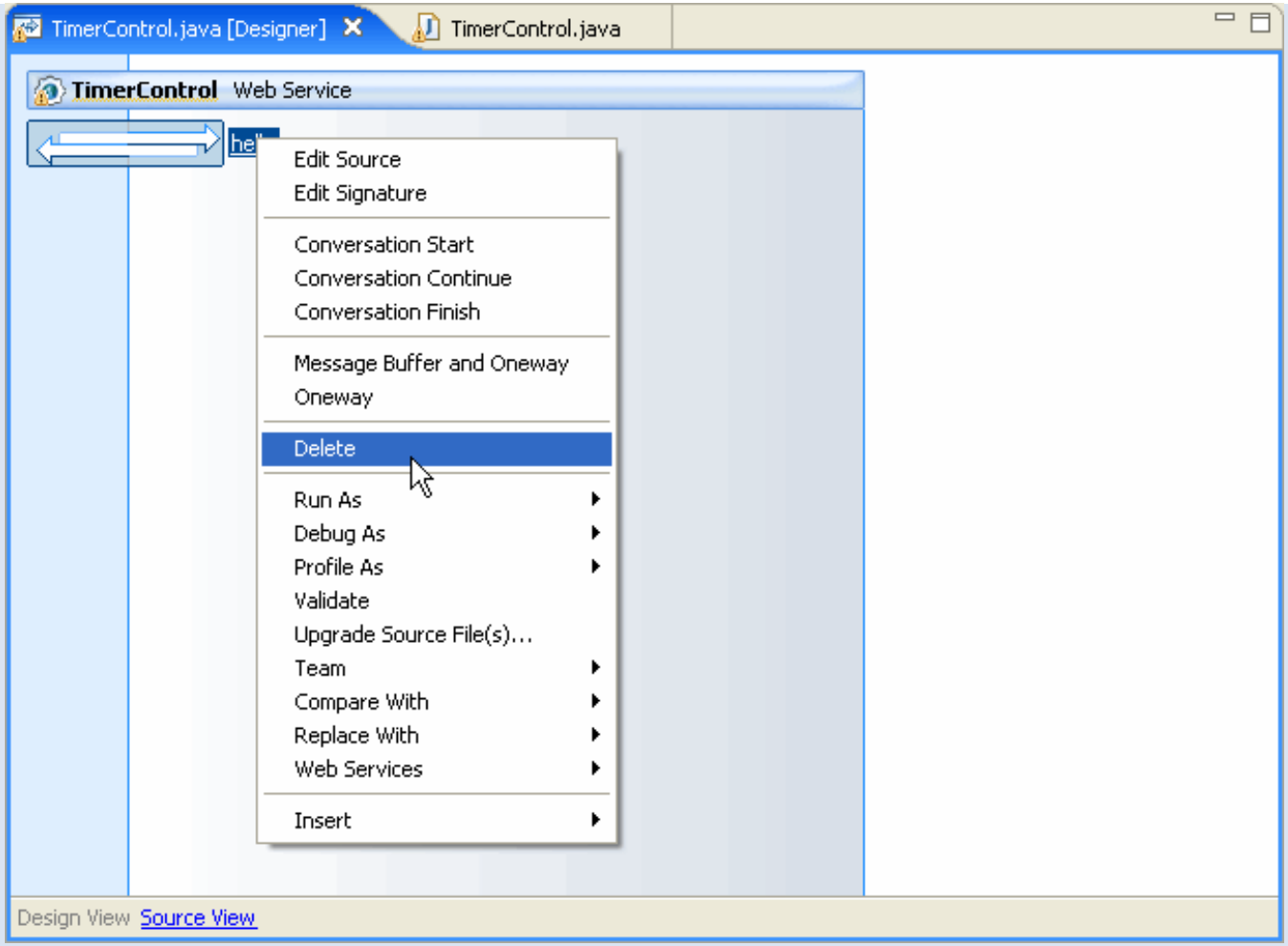
2. The **Quick Fix** pull-down will appear:



Click **Import Serializable** and press Enter and a new import line will be generated to resolve the error. Save the file with **File > Save**.

By default, a hello() method was inserted when you created the web service. This method is not needed.

1. Click on the **Designer** tab of your web service. Right click on the standard `hello()` method and choose **Delete**.



Confirm the deletion by clicking **Yes**.

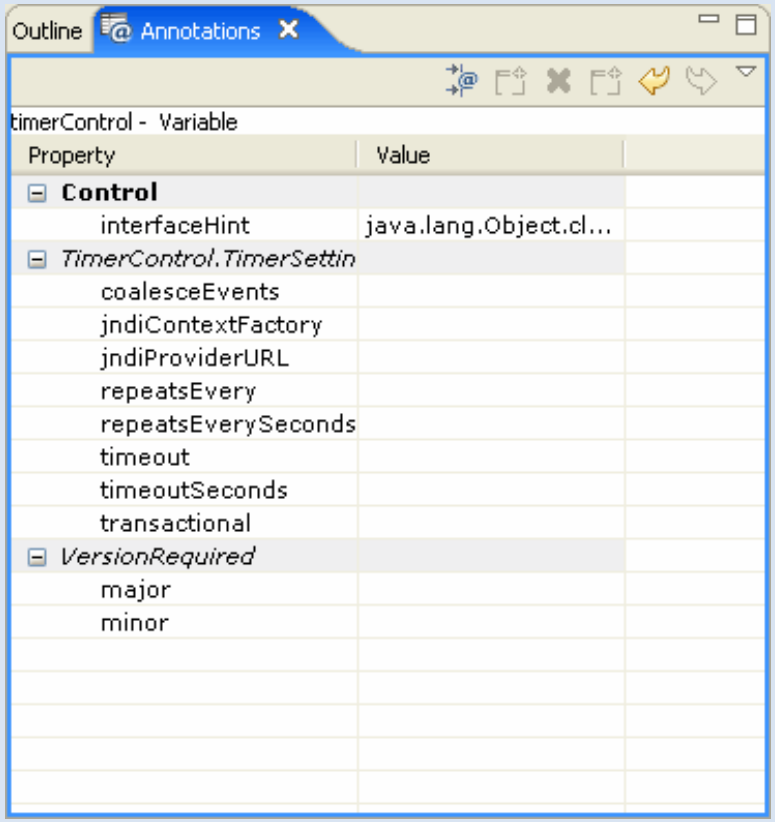
To create the timer control:

1. Right click on the **Designer** pane and click **New Control Reference** . Under **New System Control,** click on **Timer Control** and click **OK**. The control declaration is inserted into the body of the class, with the name (**timerControl**) highlighted. If you click on the **TimerControl.java** tab, you can verify that the following code was inserted:

   ```
   @Control
   private TimerControl timerControl;
   ```

   The properties of the `TimerControl` annotation are displayed in the **Annotations** view to the right. Click on the **Annotations** tab if it is not already open and visible.

2. Click in the cell under the **Value** column beside the **repeatsEverySeconds** property.



Type **2** and push the Enter key. In the source code editor window, the control annotation is updated to:

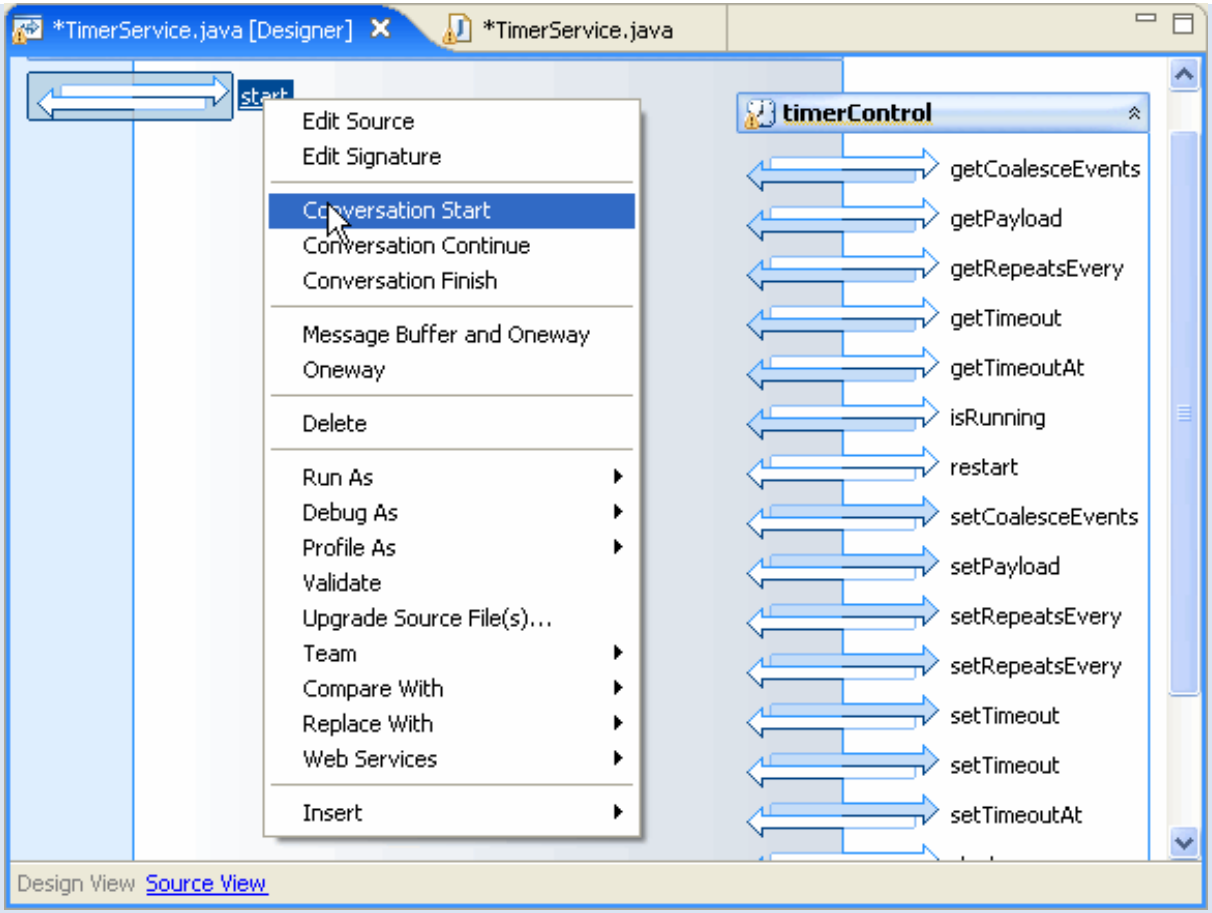
```
@Control
@TimerControl.TimerSettings(repeatsEverySeconds=2)
private TimerControl timerControl;
```

and the new property value is also displayed in the **Annotations** view.

We now have a timer control called **timerControl** which will call back the web service every two seconds. Next we will define two web methods, one to start the timer control and one to stop it.

To define the web method to start the timer:

1. From the **Designer** pane, right click and choose **New Web Method**. A new operation (web method) is created in the editor, with the default name of the method highlighted and the properties of the web method in the **Annotations** view at the right. Enter the new method name: **start** and press the Enter key.
2. Right click on the **start** method name and choose **Conversation Start** from the pulldown.



3. Now edit the source code and replace the return; statement of the method body to call the timer start method with

```
timerControl.start();
System.out.println("**************");
System.out.println("Timer started");
System.out.println("**************");
```

The web method now looks like this:

```
@Conversation(Conversation.Phase.START)
@WebMethod
public void start()
{
   timerControl.start();
   System.out.println("**************");
   System.out.println("Timer started");
   System.out.println("**************");
}
```

To define the web method that ends the timer:

1. Insert a web method named **stop** as above.
2. Right click on the method name and choose **Conversation Finish** .
3. From the source code editor, change the method declaration to return a String value.
4. Replace the return; statement of the method body to call the timer stop method with

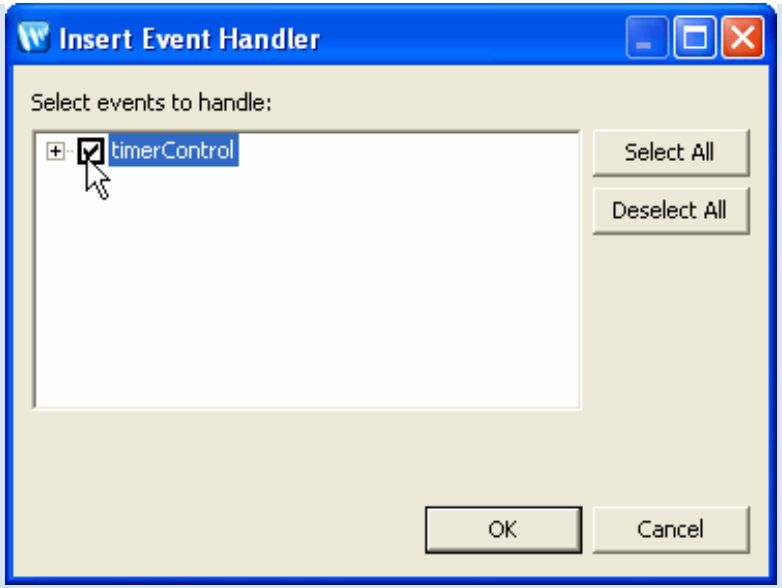
```
timerControl.stop();
System.out.println("**************");
System.out.println("Timer stopped");
System.out.println("**************");
return "ok";
```

The web method now looks like this:

```
@Conversation(Conversation.Phase.FINISH)
@WebMethod
public String stop()
{
   timerControl.stop();
   System.out.println("**************");
   System.out.println("Timer stopped");
   System.out.println("**************");
   return "ok";
}
```

To define the event handler for when the timer control signals that the timer has elapsed:

1. Right click on the **Designer** pane or the source code pane and choose **Insert > Control Event Handler**.



From the **Insert Event Handler** dialog, click on **timerControl** and click **OK**.

2. Insert the following lines into the event handler body in the source code pane:

```
System.out.println("**********************************");
System.out.println("Callback received from timer firing");
System.out.println("**********************************");
```

The event handler should look like this:

```
@EventHandler(field="timerControl", eventSet=TimerControl.Callback.class, eventName="onTimeout")
protected void timerControl_Callback_onTimeout(long p0, Serializable p1) {

{
   System.out.println("**********************************");
   System.out.println("Callback received from timer firing");
   System.out.println("**********************************");
}
```

3. Save all of your changes with the **File > Save** command.

We now have a web service that contains:

- a timer control
- an operation (web method) to start the timer
- an operation (web method) to stop the timer
- an event handler to invoke whenever the timer elapses (every two seconds in our case)

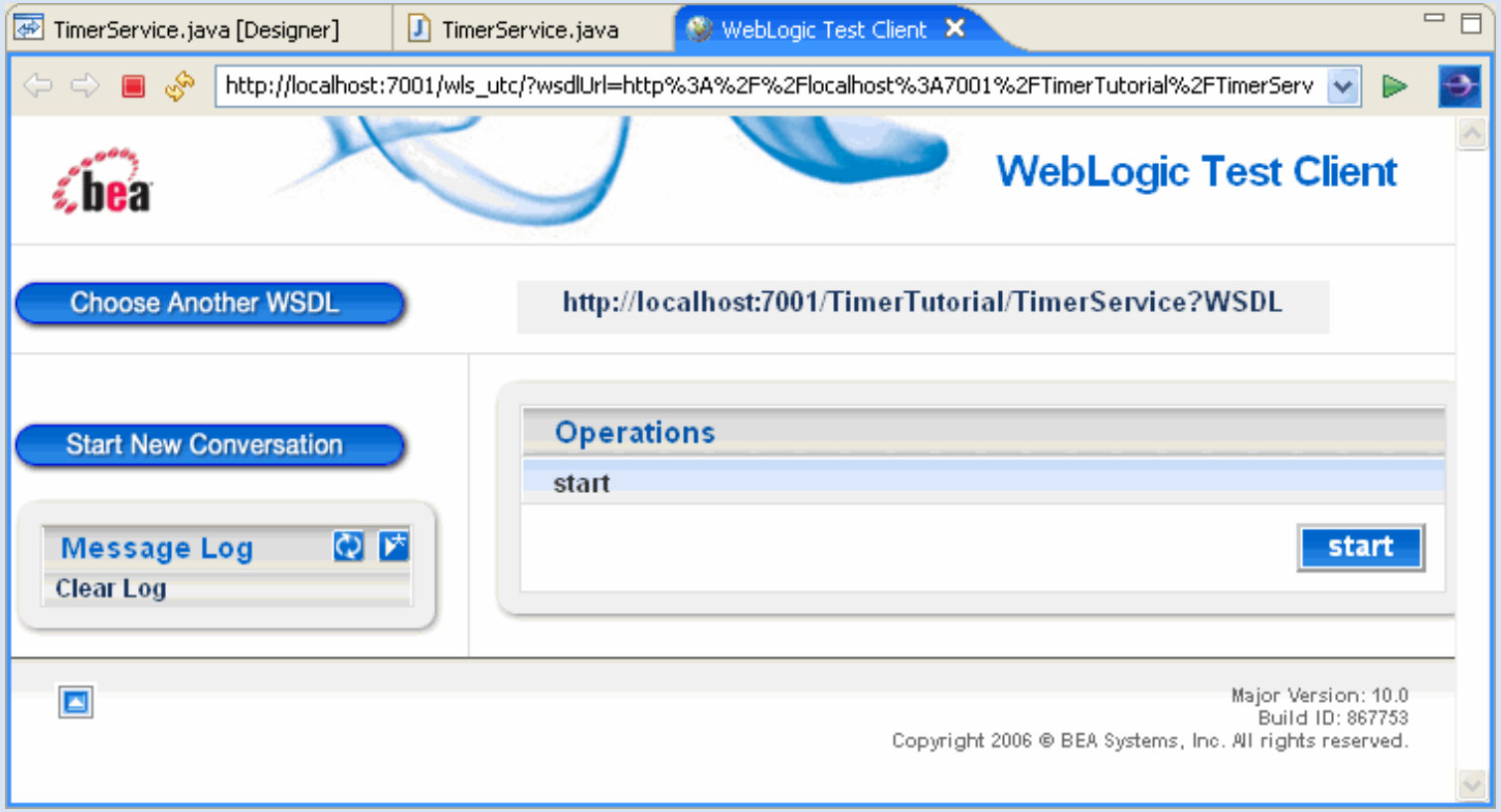
The source for your web service should now look like this:

```
package timer;

import java.io.Serializable;

import javax.jws.*;
import org.apache.beehive.controls.api.bean.Control;
import com.bea.control.TimerControl;
import weblogic.jws.Conversation;
import org.apache.beehive.controls.api.events.EventHandler;

@WebService
public class TimerService implements Serializable{

  @Control
  @TimerControl.TimerSettings(repeatsEverySeconds=2)
  private TimerControl timerControl;
  private static final long serialVersionUID = 1L;


  @WebMethod
  @Conversation(Conversation.Phase.START)
  public void start() {
               timerControl.start();
         System.out.println("***************");
               System.out.println("Timer started");
               System.out.println("***************");
  }

  @WebMethod
  @Conversation(Conversation.Phase.FINISH)
  public String stop() {
               timerControl.stop();
               System.out.println("***************");
     System.out.println("Timer stopped");
               System.out.println("***************");
               return "ok";
  }

  @EventHandler(field = "timerControl", eventSet = TimerControl.Callback.class, eventName = "onTimeout")
  protected void timerControl_Callback_onTimeout(long p0, Serializable p1) {
               System.out.println("***********************************");
               System.out.println("Callback received from timer firing");
               System.out.println("***********************************");
         }
}
```

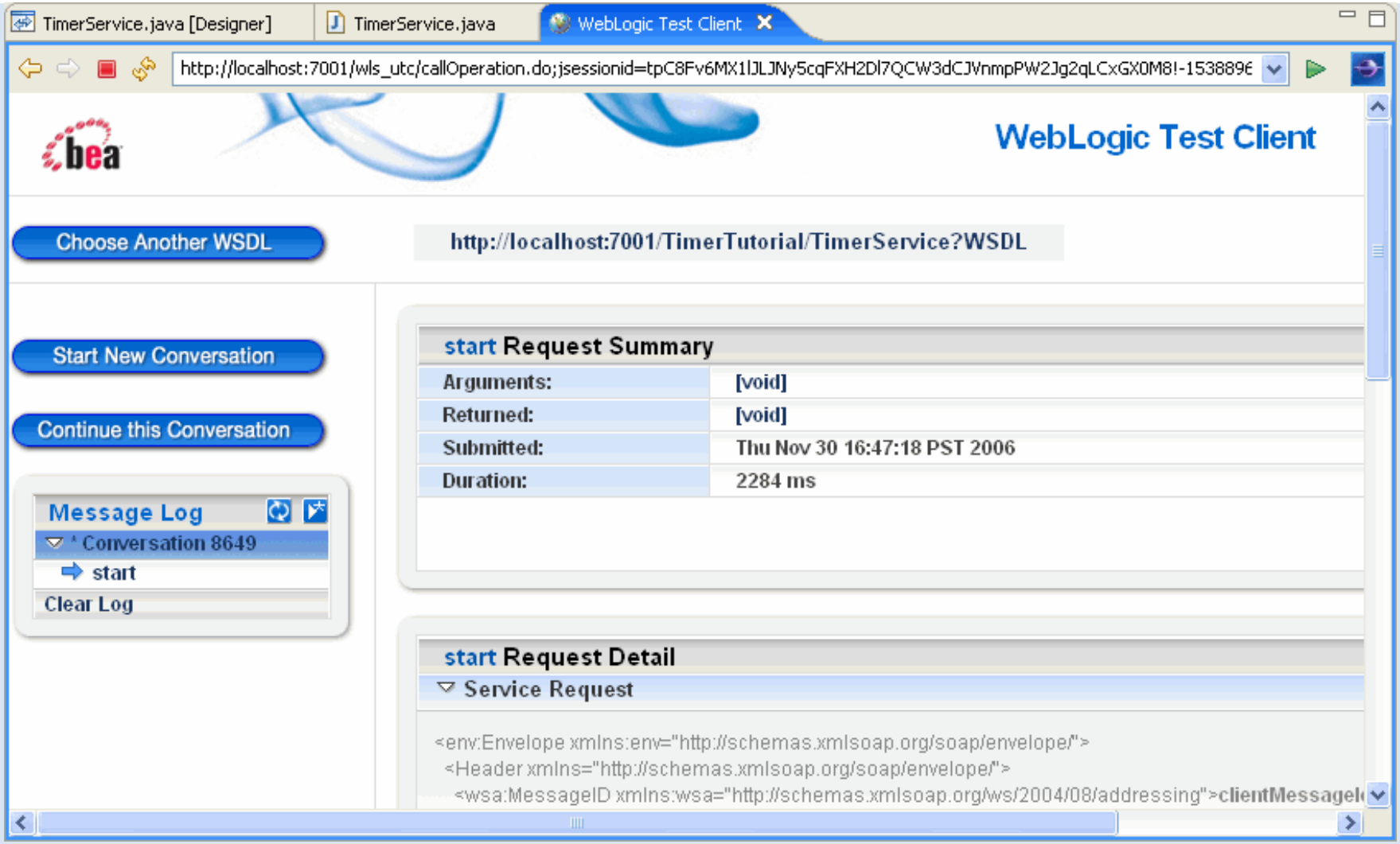
Test the Web Service / Timer Control

To test the web service and the timer control:

1. Right click on the editor pane and choose **Run As > Run on Server**. The WebLogic Test Client will run in a tab in the editor window.



2. Click the **start** button to invoke the **start** operation. The Test Client will display the results returned from the **start** operation.



3. Switch to the WebLogic Server console window (command prompt window with the header bar **WebLogic Server - 10.0**) to see the startup and callback results. The console window is iconized on the status bar by default. When you open the console window, you can see the timer starting and the message lines:

   ```
   **************
   Timer started
   **************
   ```

   generated by the **start** operation. This message will quickly scroll away, since the timer will immediately begin firing every two seconds.

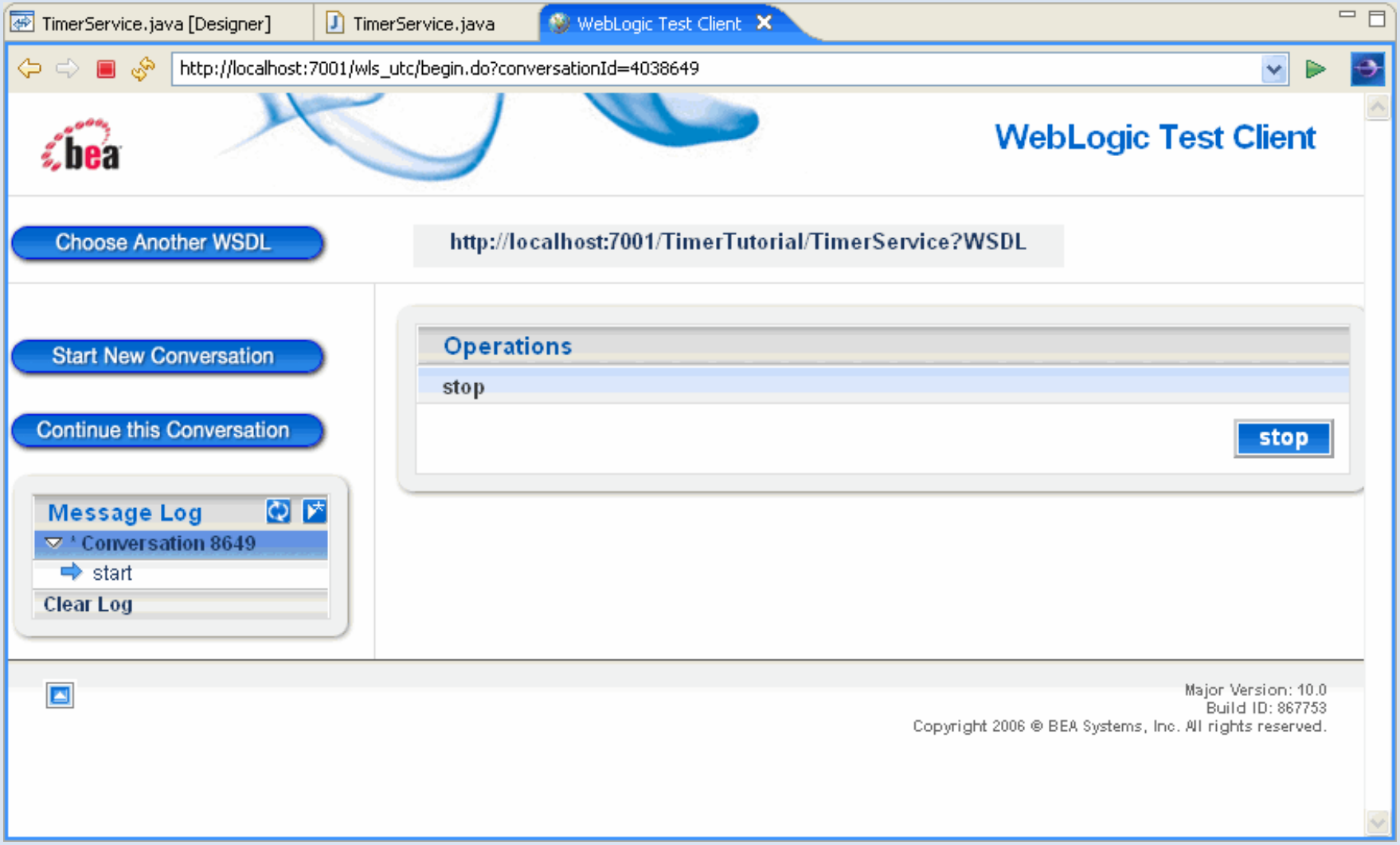
   The console window will then show the timer firing. Each time the timer fires, a block of status information will be displayed, including the lines:

   ```
   **********************************
   Callback received from timer firing
   ```

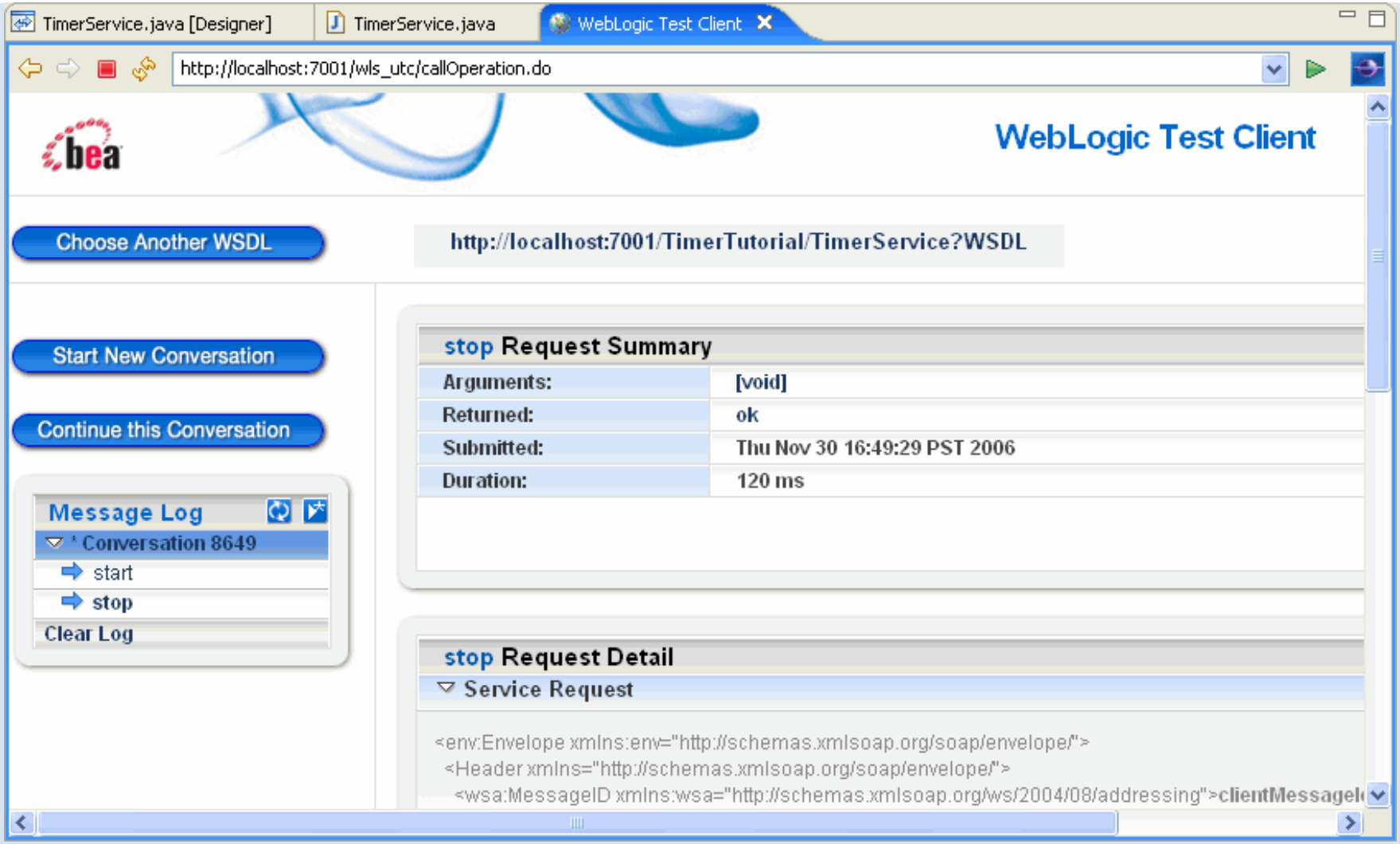
```
************************************
```

that are generated by the event handler that receives callbacks when the timer fires every two seconds.

4. To stop the timer firing, click on the **Continue this conversation** link just above the **start Request Summary** in the body of the test client window. The test client will then display the operation(s) for the next phase of the conversation, in this case, the **stop** operation.



5. Click **stop.** The result of the **stop** operation will be displayed in the test client window:



The console window will no longer show the timer firing every two seconds, and will display the lines:

```
**************
Timer stopped
**************
```

to indicate that the **stop** operation was successful.

Related Topics

Tutorial: Getting Started with BEA Workshop for WebLogic Platform

Using WebLogic System Controls

Timer Control

TimerControl Interface

Timer Control Reference

Tutorial: Creating a Web Service with Timer Control

Tutorial: Testing Controls with JUnit

This tutorial shows you how to build a simple custom control and test it using JUnit with Workshop for WebLogic.

Focus of this Tutorial

As you work through this tutorial, you will:

- Learn about Workshop for WebLogic utility projects.
- Create a simple custom control.
- Test your control using JUnit.
- Learn Workshop for WebLogic shortcuts for developing JUnit tests.

Steps in this Tutorial

Create a Custom Control
: Use Workshop for WebLogic to create a workspace, utility project and a custom control.

Create the Test Class
: Use Workshop for WebLogic to create a JUnit test class.

Run the Test Case
: Use Workshop for WebLogic to run the test case.

Related Topics

Testing Controls

Utility Projects

Click the following arrow to navigate through the tutorial:

Step 1: Create a Custom Control

In this step you will create a Utility project to house your control so they can be used by multiple modules in an application.

In this step, you will:

- Start Workshop for WebLogic
- Create a workspace
- Create an Utility project
- Create an Custom Control

To Start Workshop for WebLogic

If you haven't started Workshop for WebLogic yet, use these steps to do so.

... on Microsoft Windows

If you are using a Windows operating system, follow these instructions.

- From the **Start** menu, click **All Programs > BEA Products > Workshop for WebLogic Platform 10.0**

...on Linux

If you are using a Linux operating system, follow these instructions.

- Run BEA_HOME/workshop100/workshop4WP/workshop4WP.sh

To Create a Workspace

You use a workspace to contain related source code. This one will end up containing both your control source and the source you'll test the control with.

1. If the **Workshop Launcher** dialog is not displayed, select **File > Switch Workspace**. Otherwise, skip to the next step.
2. In the **Workspace Launcher** dialog, click **Browse**, then browse to the directory that you want to contain your new workspace directory.

   This can be any directory. You'll be creating a new directory *inside* this one for your workspace.

3. When you have a directory selected, click **Make New Folder**. Name the new folder `JUnitTutorial` press **Enter** to create the folder, then click **OK**.
4. In the **Workspace Launcher**, click **OK**.
5. Close the **Welcome** view.

Workshop for WebLogic will create a new empty workspace in the folder you created, then refresh to display the workspace. Note that the Navigator view is empty.

To Create an Utility Project

An utility project contains shared code that can be used across multiple different projects.

1. Click **File > New > Project.**
2. In the **New Project** dialog, expand **J2EE**, select **Utility Project**, then click **Next**.
3. In the **New Java Utility Module** dialog, in the **Project name** field, enter `MySharedControls`, then click **Next**.
4. Under **Select Project Facets**, confirm that the "Beehive Contols" facet is selected. (This facet must be selected because it contains ControlTestCase, an extension of junit. framework.TestCase, as well as the control validation and build libraries.)

   Click **Finish**.

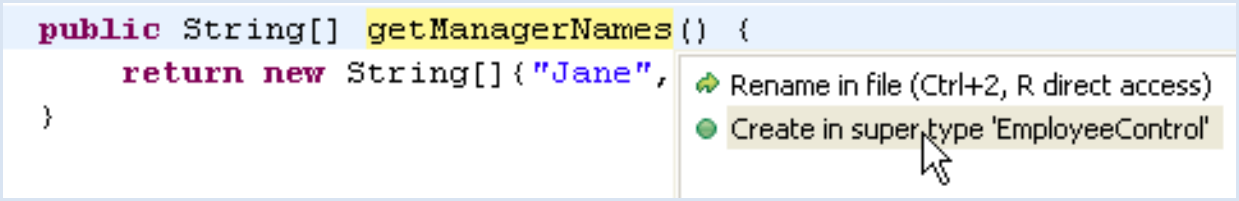
To Create a Custom Control

In this step you will create the control to be tested.

1. On the **Project Explorer** view, expand the node **MySharedControls**, right-click the **src** folder and select **New > Package**.
2. In the **New Java Package** dialog, in the **Name** field, enter `sharedcontrols`, and click **Finish**.
3. On the **Project Explorer** view, right-click the **sharedcontrols** package and select **New > Custom Control**.
4. In the **New Control** dialog, in the **Control name** field, enter `EmployeeControl` and click **Finish**.
5. On the **Project Explorer** view, open the package **sharedcontrols** and double-click **EmployeeControlImpl.java** to open the file's source code. Edit the source code so it appears as follows. Code to add appears in red.

```
package sharedcontrols;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import java.io.Serializable;

@ControlImplementation
public class EmployeeControlImpl implements EmployeeControl, Serializable {
    private static final long serialVersionUID = 1L;

    public String[] getManagerNames() {
        return new String[]{"Jane","Bob","Amy"};
    }
}
```

6. Place the cursor inside the method name **getManagerNames** and press **Ctrl+1**.
   This will bring up an options menu.
   Double-click the option **Create in super type 'EmployeeControl'**.
   This will add the method signature to the control interface file EmployeeControl.java.



7. Press **Ctrl+Shift+S** to save your work.

Related Topics

Testing Controls

Click one of the following arrows to navigate through the tutorial:

Step 2: Create a Test Class

In this step you will create a test class that will run tests on the control you just created. You can put tests in a separate source folder for better organization and so it's easier to exclude them later during packaging for production. Better yet, you can put the tests in a completely separate project, provided that the project dependencies are correctly configured. But, in this simple case, we will leave the control and test class in the same project.

In this section, you will:

- Create a new Source Folder and Package
- Create the Test Class

To Create a New Source Folder and Package

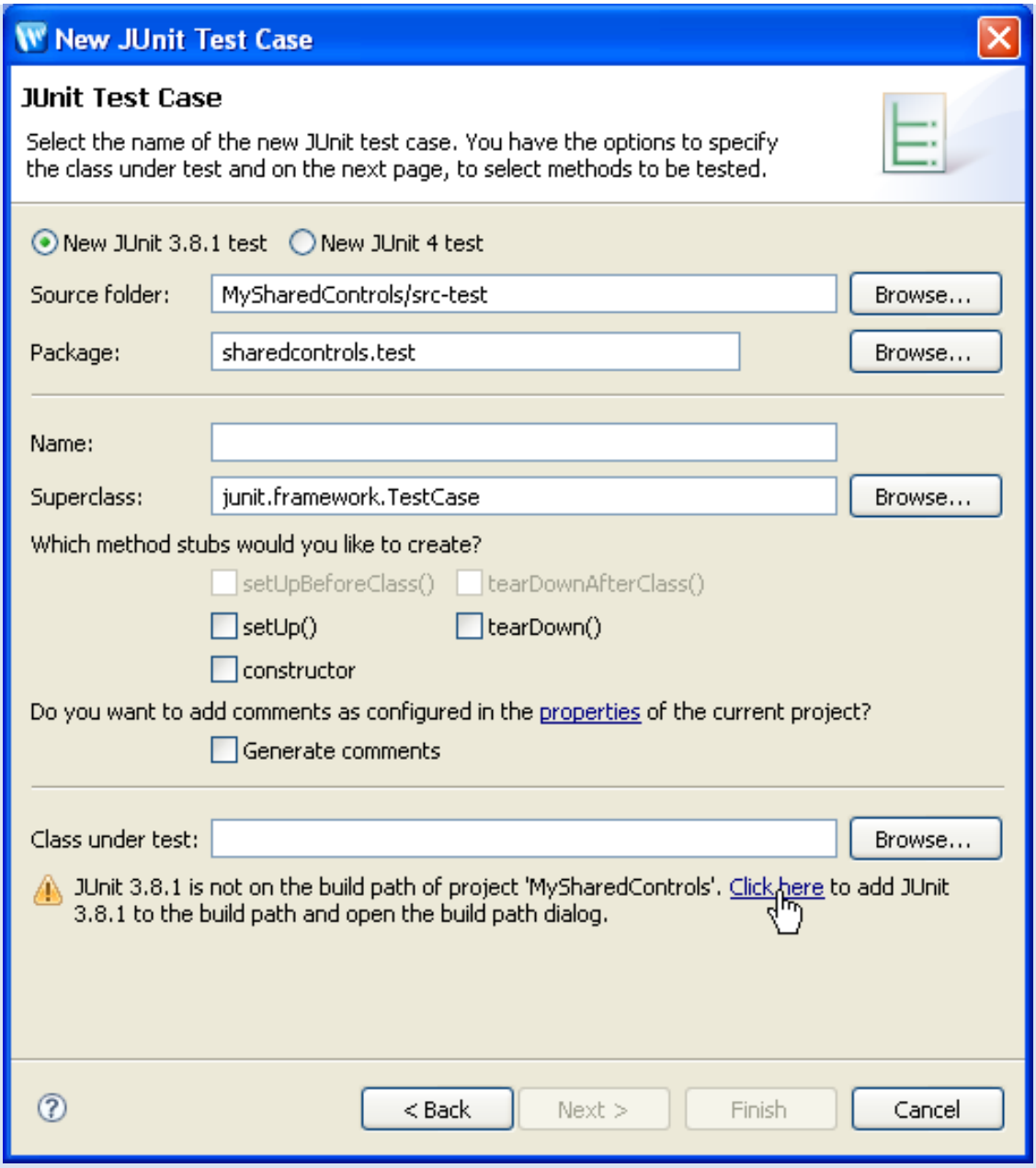
Here, you'll create a new source folder to hold the test class.

1. On the **Project Explorer** view, right-click **MySharedControls** and select **New> Other**.
2. In the **New** dialog, open the **Java** node, select **Source Folder**, and click **Next**.
3. In the **Folder name** field, enter `src-test` and click **Finish**.
4. On the **Project Explorer** view, right-click the **src-test** folder and select **New > Package**.
5. In the **New Java Package** dialog, in the **Name** field, enter `sharedcontrols.test` and click **Finish**.

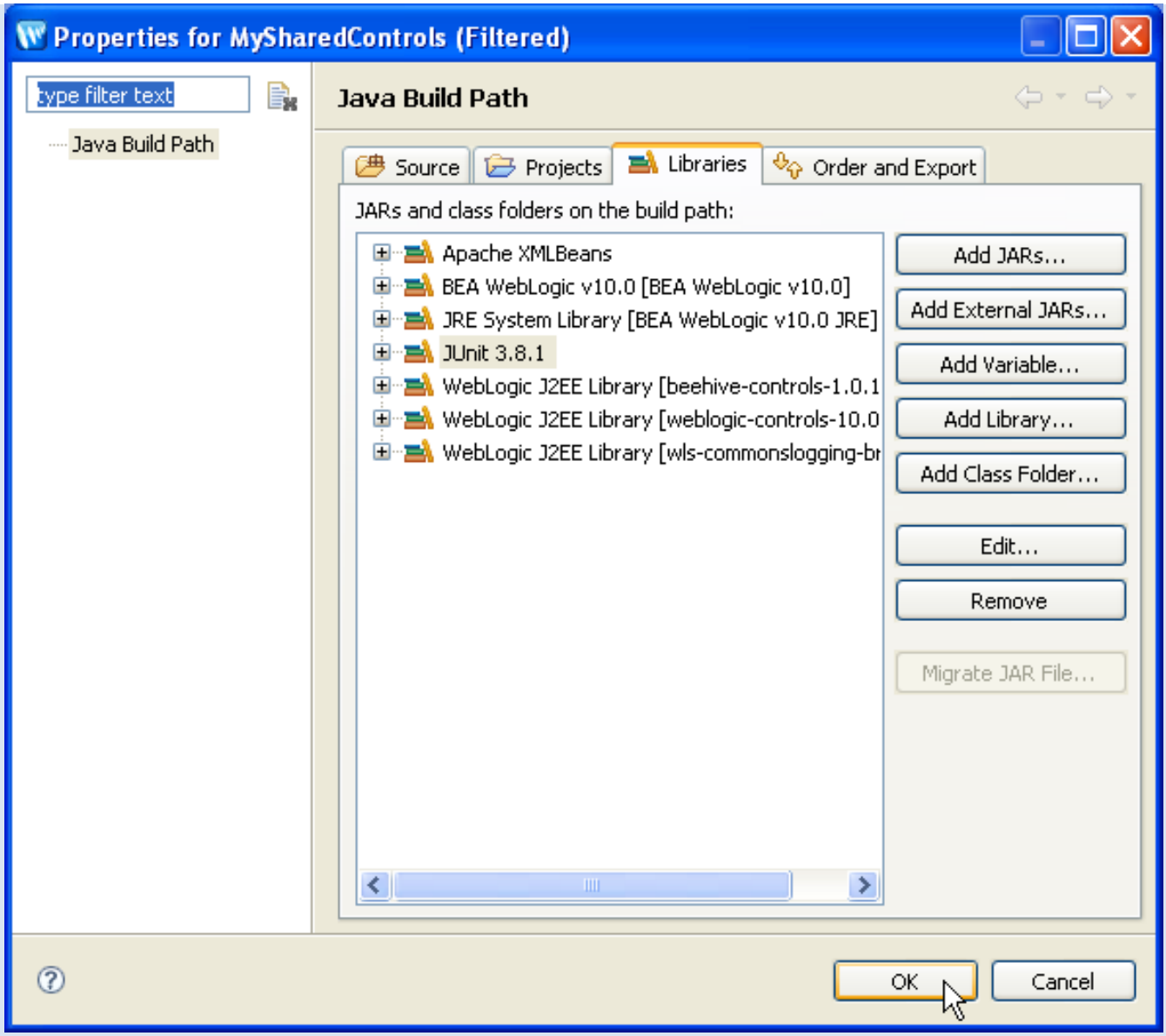
To Create the Test Class

In this step you will create the class that tests your control.

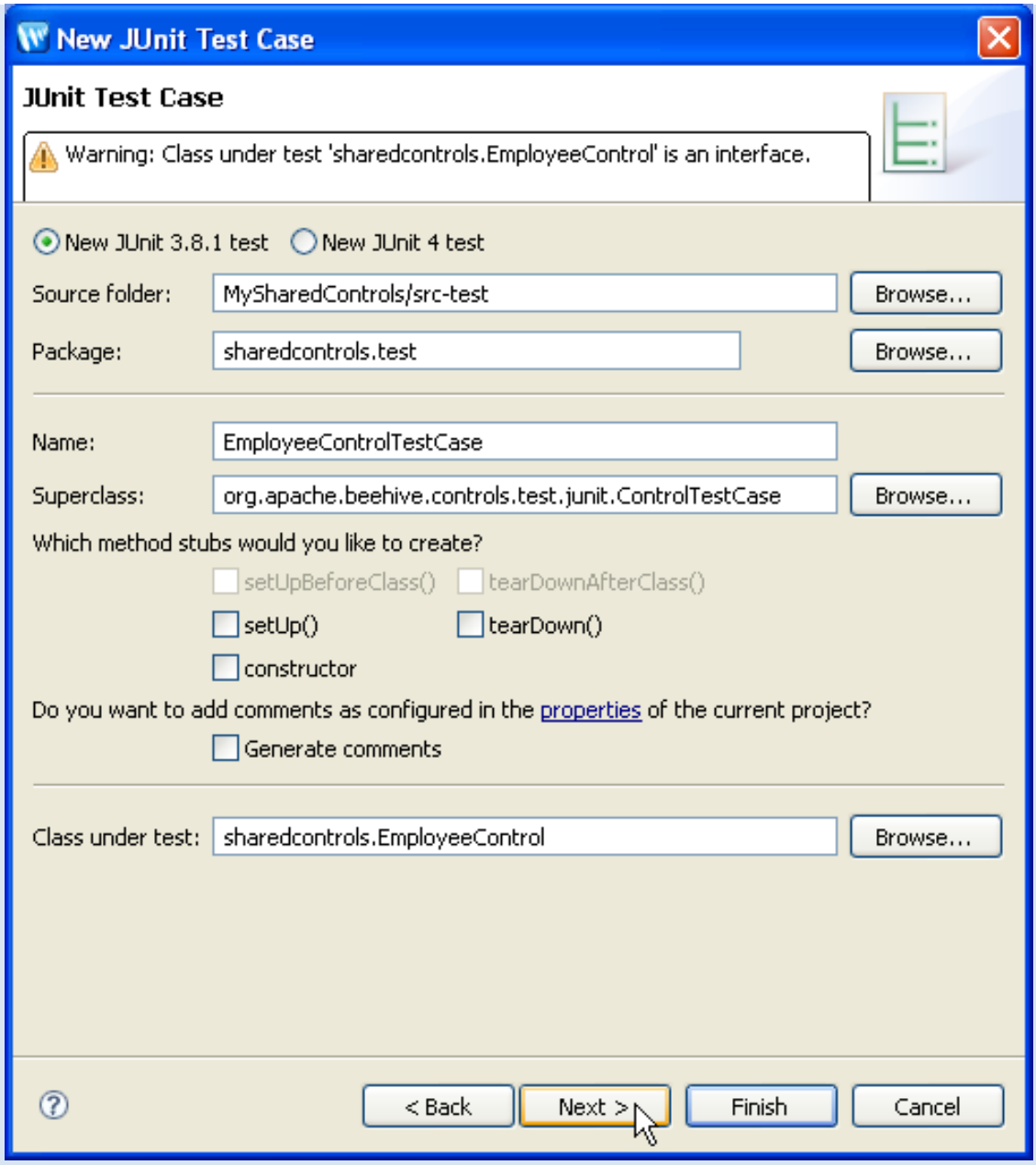
1. On the **Project Explorer** view, right-click the **sharedcontrol.test** package and select **New > Other**.
2. In the **New** dialog open nodes **Java > JUnit**, select **JUnit Test Case**, and click **Next**.
3. In the **New JUnit Test Case** dialog, click the link **Click here**.



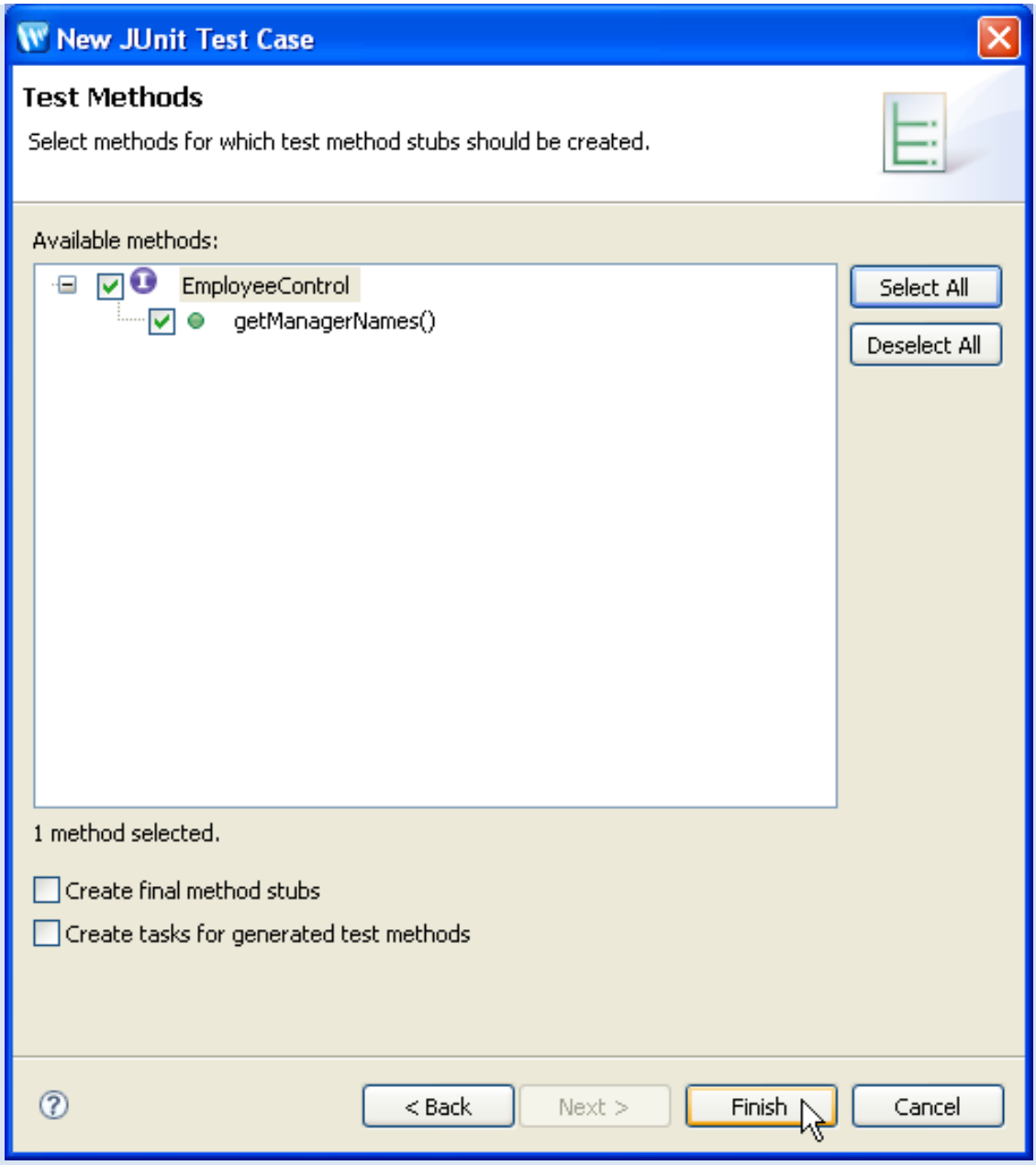
4. In the **Properties for MySharedControls** dialog, click **OK**.



5. In the **New JUnit Test Case** dialog, in the **Name** field, enter `EmployeeControlTestCase`.
   In the **Superclass** field, enter `org.apache.beehive.controls.test.junit.ControlTestCase`.
   (**Hint:** in the **Superclass** field enter `ControlTestCase` and press **Ctrl+Space Bar** to fill in the remaining package names.)
   In the **Class under test** field, enter `sharedcontrols.EmployeeControl`.
   Click **Next**.



6. Click **Select All** and then **Finish**.



7. Edit the source code for **EmployeeTestCase.java** so it appears as follows. Code to add appears in red. Make sure to delete the line of code `fail("Not yet implemented");`.

```
package sharedcontrols.test;

import org.apache.beehive.controls.api.bean.Control;
import org.apache.beehive.controls.test.junit.ControlTestCase;

public class EmployeeControlTestCase extends ControlTestCase {

    @Control
    sharedcontrols.EmployeeControl employeeControl;

    /*
     * Test method for 'sharedcontrols.EmployeeControl.getManagerNames()'
     */
    public void testGetManagerNames() {
        String[] mgrs = employeeControl.getManagerNames();
        assertNotNull("Didn't find managers!", mgrs);
        assertTrue("Found wrong number of managers!",
                   mgrs.length == 3);
```

```
        }
    }
```

(Notice that you were able to use the control simply by using the @Control field notation and didn't have to programmatically instantiate it yourself. That's the magic of ControlTestCase. By extending that class you inherit its setUp() and tearDown() methods that do the declarative wire-up for you.)

8. Select **Ctrl+Shift+S** to save your work.

Related Topics

Testing Controls

Click one of the following arrows to navigate through the tutorial:

Step 3: Run the Test Case

In this step you'll run the test against the control class.

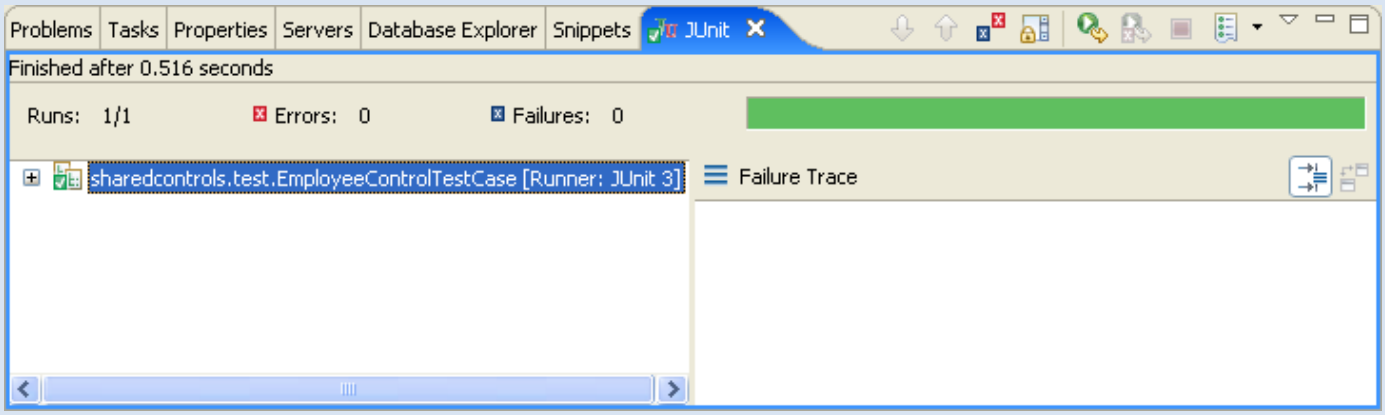
In this step, you will:

- Run a Successful Test
- Run a Failed Test

Run a Successful Test

1. In the **Project Explorer**, right-click **EmployeeControlTestCase** and select **Run As > JUnit Test**.

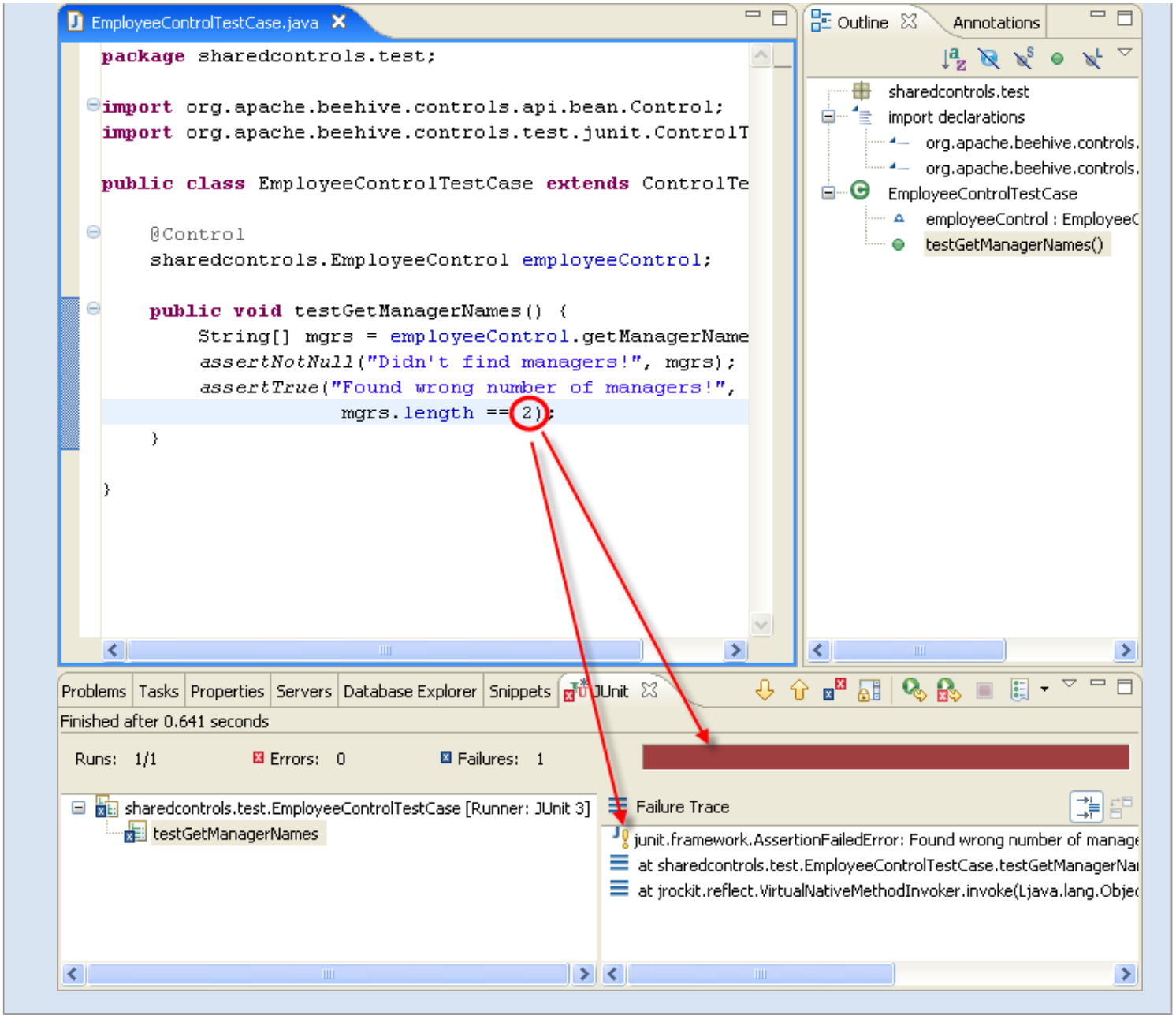
   The **JUnit** view will appear with the green progress bar, meaning that all tests have passed.



Run a Failed Test

1. Edit the **testGetManagerNames** method so that the expected array length is 2 instead of 3 and run the test again.

   The **JUnit** view will look like the following. The red bar indicates that the test has failed and an error message is shown.



Notice that at no time during this test was a server started, the test was running in a pure JUnit environment. The only special class used was a standard sub-class of TestCase. (This is not universally true for all controls. In cases where the control takes a dependency on a server-bound resource, the control cannot be thoroughly tested in a pure JUnit environment.)

Also, notice that the setup of the test case was very straight forward. Most of the setup was in making sure the test class lived in its own source folder. By extending ControlTestCase and letting the JUnit wizard generate the stub methods, all you had to do was write your test code.

Related Topics

Testing Controls

Click one of the following arrows to navigate through the tutorial:

Using Controls

BEA Workshop for WebLogic Platform's controls make it easy to access encapsulated logic.

You can also create your own custom controls to encapsulate business logic in a reusable component. For information on creating custom controls, see Custom Controls.

Topics Included in This Section

Invoking a Control Method
: Describes how to insert a control into your code and invoke control methods.

Overriding Control Properties
: Discusses how to override the properties that are set in the control.

Handling Control Events
: Describes how to work with events generated by the control.

Handling Control Method Exceptions
: Describes how to handle exceptions thrown by a control.

Control Transactions
: Describes how to do transactions with a control.

Related Topics

Working with Beehive Controls

Invoking a Control Method

The following topic explains how to utilize a (system or custom) control resource in an application. This topic is divided into the following sections:

Adding a Control Declaration

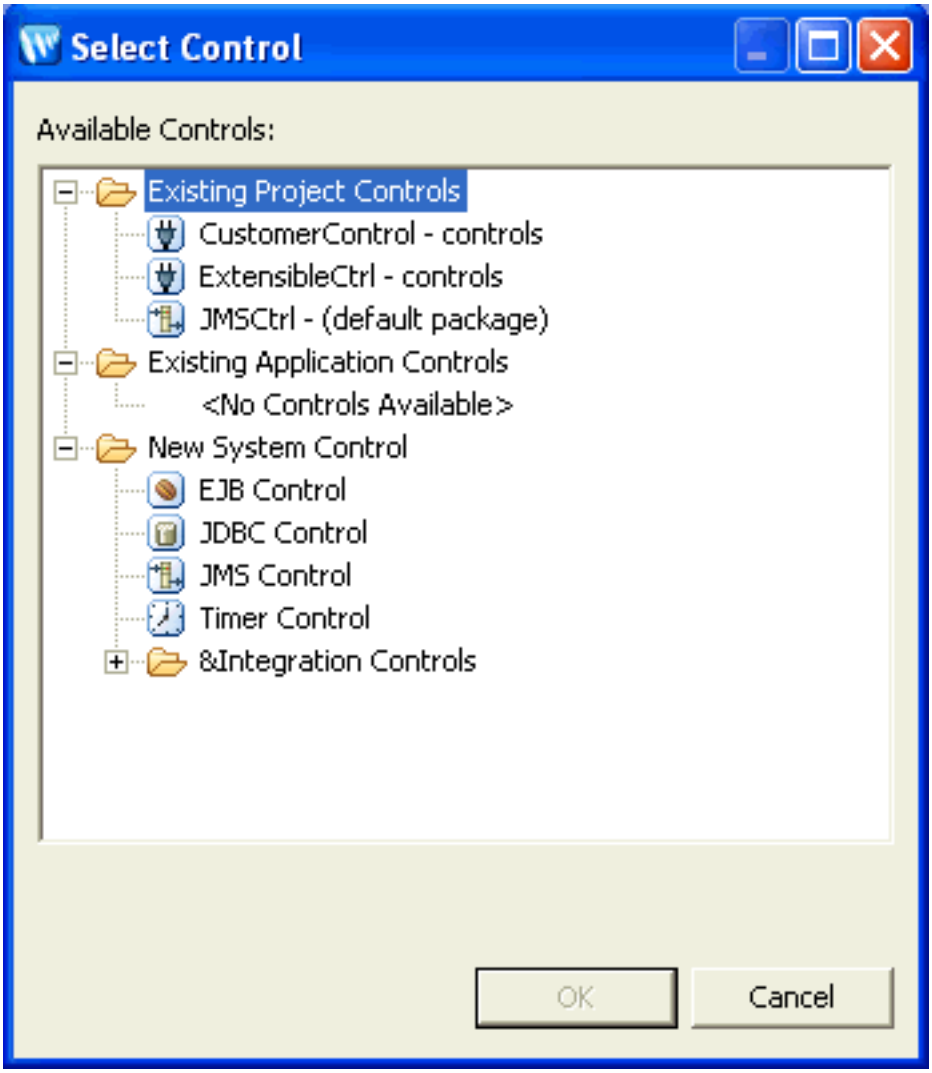
Invoking a Method

Adding a Control Declaration

To invoke a control method, first add a control declaration to the calling client and then invoke a method on that control.

To add a control declaration, open the J2EE perspective (**Window > Open Perspective > J2EE**), right-click anywhere with the Java source of the client class and select **Insert > Control**. Select from the list of controls available to your client.

See the topic Select Control Dialog for a complete description of the dialog below.



Selecting a control from the dialog will add two things to your client class: (1) a control class import statement and (2) a control class declaration:

MyClient.java

```
import controls.CustomerControl;

@Control
private CustomerControl customerControl;
```

Once the control class declaration is in place, you can call the control's methods.

Invoking a Method

Once you've added a control class declaration to your client class, you can invoke its methods using the standard Java dot notation. For example, assume that you have added a declaration for the control class CustomerControl:

```
@Control
```

```
    private CustomerControl customerControl;
```

Also assume that the control defines a method `getCustomers()`:

```
    public Customer[] getCustomers();
```

You can invoke this method from your client code as follows:

```
    Customer[] custResult = customerControl.getCustomers();
```

Overriding Default Control Properties

Sometimes it is desirable to override a control property from within client code without editing the control code. For example, suppose you have a database control where the JNDI data source name is set by an annotation.

```
@JdbcControl.ConnectionDataSource(jndiName = "myDataSource")
public interface CustomerDB extends JdbcControl
```

But also suppose the JNDI name has changed, or you want to reuse the database control in another context where the JNDI name of the data source is different. It might be inconvenient (or impossible) to manually change the annotation value and recompile the control. In this case it is desirable to override the JNDI value directly from within client code. To override the annotation value you call into another class: the control's associated ControlBean class. The ControlBean class implements all of the control's methods but also gives you programmatic access to the control's annotation-based properties. The ControlBean is a generated JavaBean class that is created automatically by Beehive when a control is built.

The following sections explain what this generated ControlBean class is and how to use it to override default control annotation values.

You can also override annotation values that are set in the client class. For details see Overriding Control Annotation Values Through the ControlBean below.

The ControlBean Generated Class

Every control has an associated ControlBean class. The ControlBean is generated automatically and provides a programmatic way to access settings that otherwise would only be available through the controls annotations. The ControlBean class typically does not exist as a JAVA source file; instead only the compiled CLASS file is present within the application.

The ControlBean generated class is derived from the members and methods in the control. The ControlBean generated class is really a superset of the original control class that provides a broader set of access points into the control. In the case of the Timer control, the shape of the ControlBean generated class is fixed (because the members and methods of the Timer control are fixed). But in the case of extensible system controls (EJB, JDBC, and JMS controls) and custom controls the ControlBean generated class is variable (because the members and methods of the source controls are variable).

The name of the generated ControlBean class is the control name appended with 'Bean'. If the control name is `CustomerDB.java`, then the generated ControlBean class will be `CustomerDBBean.class`.

For more infomation about the ControlBean generated from control sources see the Apache Beehive documentation: The Control Authoring Model.

Related Topics

The Control Authoring Model

Overriding Control Properties

Overriding Control Properties

Sometimes it is desirable to override a control definition property from within client code or from an external configuration file. Common examples overriding the endpoint address on a service control or the data source on a database control. Suppose that you have a database control definition where the data source is configured as follows.

DatabaseControl.java (Control Definition Code)

```
@JdbcControl.ConnectionDataSource(jndiName = "myDataSource")
public interface CustomerDB extends JdbcControl
```

What if you want to reuse the database control in another context where the data source is different? It might be inconvenient (or impossible) to manually change the annotation value and recompile the control definition. In this case it is desirable to override the data source value from another source.

To override the annotation value in the control definition, you can do one of the following:

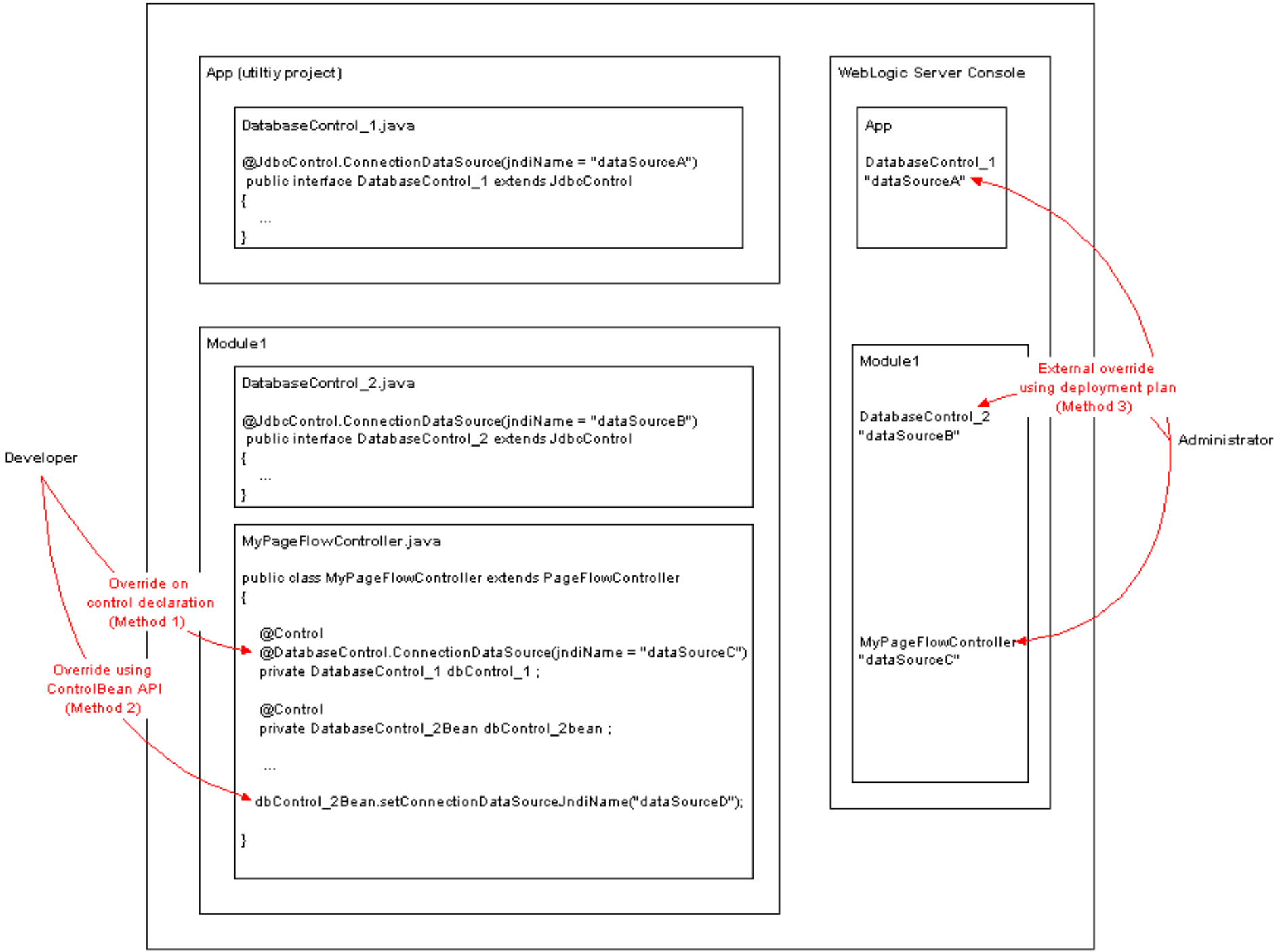
1. configure the value in the client code's control declaration (for use by developers)
2. set the value programmatically using the ControlBean API (for use by developers)
3. configure the value using a deployment plan (for use by administrators)

Developers use option (2) when the properties of the control instance are not known until runtime. In this case you will typically first determine the appropriate control properties and then reset those properties programmatically using the ControlBean API.

Option (3) is for use by server administrators who are managing the deployment of an application into a different environment. For example, if an application is being promoted from a testing environment into a real world production environment. A deployment plan can override annotation values on any of the control contexts in the application, including *either* the control definition *or* the control declaration in client code. Note that deployment plans can only override contexts already established by annotations, they cannot add new contexts or override any value set through the ControlBean.

The hierarchical rules of precedence for these override methods are described below in Order of Precedence.

The diagram below shows the different options available to developers and administrators.



Requirements for Annotation Overrides

Annotation overrides are supported only when the following requirements are met:

- Overrides are supported only for applications, not stand-alone modules.
- The application must reference the weblogic controls EAR library (weblogic-controls-1.0).
- Application scoped controls must be packaged in the APP-INF/lib directory. Controls referenced through the manifest classpath are not supported.

Method 1: Overriding Control Annotation Values Through the Control Declaration Field

You can override a control's default properties on the control's declaration field in your client code.

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

MyWebService.java (Client Code)

```
@Control
private DatabaseControl dbControl;
```

To override the default `jndiName` property, use the following declaration.

```
@Control
@DatabaseControl.ConnectionDataSource(jndiName = "myOtherDataSource")
private DatabaseControl dbControl;
```

In the above declaration, the database control will use `myOtherDataSource` instead of the value used in the control definition. This override value will apply to all method calls from within MyWebService.java.

Method 2: Overriding Control Annotation Values Through the ControlBean Class

The ControlBean class implements all of the control's methods but also gives you programmatic access to the control's annotation-based properties. The ControlBean is a generated JavaBean class that is created automatically by Beehive when a control is built. (Note that the control author may disable overrides of control properties. See the @PropertySet annotation for details.)

The name of the generated ControlBean class is the control name appended with 'Bean'. If the control name is `CustomerDB.java`, then the generated ControlBean class will be `CustomerDBBean.class`.

Use caution when you use this approach because it cannot be overridden by a deployment plan (method 3 below).

For more infomation about the ControlBean generated from control sources see the Apache Beehive documentation: The Control Authoring Model.

Calling ControlBean Methods

Suppose you have a database control where `jndiName` attribute points at the data source `myDataSource`.

DatabaseControl.java

```
@JdbcControl.ConnectionDataSource(jndiName = "myDataSource")
public interface CustomerDB extends JdbcControl
```

To override all other competing values of the `urls` property, add the database control's generated ControlBean class to your client and reset the JNDI (by calling the appropriate method). Note that using the ControlBean class will override all competing values, including the value in the control definition, any value set on a control declaration field, and any external configuration through a deployment plan.

To reference the ControlBean class, use the following declaration.

```
@Control
private controls.CustomerDBBean customerDBBean;
```

To override the JNDI value, call the `setConnectionDataSourceJndiName(String name)` method.

```
customerDBBean.setConnectionDataSourceJndiName("myOtherDataSource");
```

Calling Methods on the Control Definition Class

In some cases the appropriate setter method is not on the ControlBean class, but on the control definition class. For example, to override the target url of a service control, you call a method on a service control definition. The following control definition property:

```
@ServiceControl.Location(urls = {"http://some.domain.com/WebServices/HelloWorld"})
...
public interface HelloWorldServiceControl extends ServiceControl
```

is overridden by the setEndpointAddress(String url) method on the control definition class.

Method 3: External Configuration in a Deployment Plan

Some annotation values can be overridden through an external configuration file as part of a deployment plan. For details on deployment plans see Configuring Applications for Production Deployment in the WebLogic Server documentation.

Deployment plan configuration files can override annotation values both in the control definition class *and* on the control declaration field (in client code). Also deployment plans can only change the values for annotations that are already present in the code. You cannot create a new configuration context through a deployment plan.

Control authors have control over which control properties can be overridden using external configuration in a deployment plan. For details see the @PropertySet annotation.

To create a deployment plan for a control, follow these steps:

1. Ensure that the application is packaged and deployed as an EAR file. The WebLogic Server console does not support deployment plans for exploded applications.

   To package your application as an EAR, select **File > Export > EAR file**.

   To use the WebLogic Server console to deploy the application EAR, see Install an Enterprise Application in the WebLogic Server documentation.

2. Click the **Lock & Edit** button and click the **Deployments** link to go to the **Summary of Deployments** page and click the application name.

3. Select the **Deployment Plan** tab and then the ***Resource Dependencies*** (sub-)tab.

4. Within the **Controls** node, controls are located either under the module's name or **Application** (if the controls are part of a utility project). Controls are organized by classname, then instances, then fields. Select the annotation to be overridden. This will display the **Array Value Overrides** table.

5. Select a cell in the **Override Value** column, enter the override value, and press the **Enter** key.

6. After entering the new value click **Save** button on the Array Value Overrides table.

7. Select a location to store the deployment plan and click **Finish**.

8. Return to the Deployments page by clicking the **Deployments** link.

   If the application is not started (its state is marked as "Prepared"), then select the checkbox next to the application, and click the **Start** button.

   If the application is already running (its state is marked as "Active"), then select the checkbox next to the application and click the **Update** button. Select the first choice: **Update this application in place,** then click the **Finish** button.

9. Click the **Activate Changes** button.

Order of Precedence

The ControlBean always takes the highest order of precedence when overriding annotations values. The following order of precedence is used:

1. First, the values programmatically set through the ControlBean are consulted (see Method 2 above).

2. Second, any deployment plan overrides of values configured on the control declaration field are consulted (see Method 3 above).

3. Third, the values configured on the field declaration are consulted (Method 1 above).

4. Fourth, any deployment plan overrides of values set in the control definition are consulted (see Method 3 above).

5.

Finally, the values in the control definition class are consulted.

Note that programmatically calling ControlBean methods takes precedence over all other values.

For example, suppose you declare a timer control in your client code and set the `timeoutSeconds` value with an annotation:

```
@TimerControl.TimerSettings(repeatsEverySeconds=2, timeoutSeconds = 100)
@Control
private TimerControl timerControl;
```

You can override this timeout value programmatically by calling into the TimerControlBean class.

First declare the TimerControlBean in your client.

```
@Control
private com.bea.control.TimerControlBean timerControlBean;
```

Then call the appropriate method to reset the timeout value.

```
timerControlBean.setTimerSettingTimeoutSeconds(200);
```

Related Topics

The Control Authoring Model

Handling Control Events

Controls allow the specification of event methods. Event methods provide a way for a control to asynchronously notify its client that something has occurred. Event methods are especially useful when you don't want client resources to be bound up with a network request or a long running operation. Instead of forcing the client to waste resources waiting for the control to return a value, the client can disengage from the control and engage in other processes while it listens for an event from the control.

A control event causes something to happen in the client code. When an control event method is triggered it sends an event to the client, causing an event handler (implemented in the client) to execute. The event handler is a method like any other, except that the client code does not determine when it is called; instead the control event method determines when the event handler is executed.

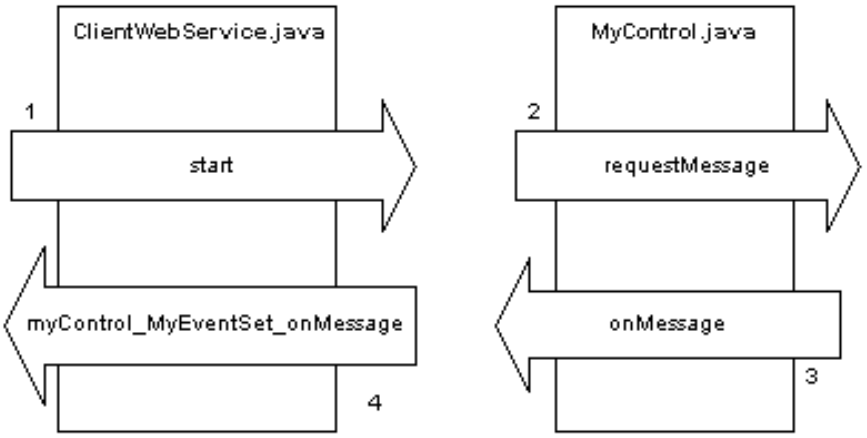
Events and Callbacks

Events and callbacks both have same underlying intent: to provide a way to asynchronously notify client code that something has occurred. The difference between the two technologies is, for the most part, terminological and a question of scope: web services send "callback" messages, controls send "event" messages.

A Control Event Scenario

The diagram below shows a simple control event scenario. The scenario contains two main components: a web service (the client) and a control (used by the client web service). (Notice that only the control interface class is depicted below; the implementation class is not depicted.)

Rightward pointing arrows depict ordinary method calls; leftward pointing arrows depict events and event handlers.



The following sequence explains how the client web service invokes the control and receives an event from the control.

1. The client web service method start is executed, which invokes the control method requestMessage.

ClientWebService.java

```
@Control
private MyControl myControl;

public void start() {
    myControl.requestMessage();
}
```

2. The control method requestMessage is executed

Notice that the requestMessage method signature is defined in the control interface class while the method body is defined in the control implementation class. The message body invokes the event method. (See stage 3 below for details on event method syntax.)

MyControl.java (Control Interface)

```
public void requestMessage();
```

MyControlImpl.java (Control Implementation)

```
public void requestMessage() {

            // Invoke the event method to send the event to the client.
            eventSet.onMessage("This is a message from the custom control.");
}
```

3. The the event method onMessage is invoked an sends an event to the client.

The control interface class exposes the event set interface. Notice that the event set interface must be decorated by the @EventSet annotation. The @EventSet annotation exposes all methods in the interface as event methods: methods capable of triggering the corresponding event handlers in the client.

MyControl.java (Control Interface)

```
@EventSet
public interface MyEventSet {
    void onMessage(String aMessage);
}
```

Notice that is there is no implementation of the onMessage event method in either the control interface or implementation classes. Only the event method signature (`void onMessage(String aMessage)`) exists in the control interface class. This is because the only purpose of an event method is to invoke the event handler in the client and pass data to that handler.

In this scenario, the event method onMessage has one parameter String aMessage, which is transmitted to the client's method handler.

The @Client annotation causes the ControlBean to initialize an implementation of the event interface. This implementation is used to fire events back to the client.

MyControlImpl.java (Control Implementation)

```
@Client
MyEventSet eventSet;

public void requestMessage() {
    eventSet.onMessage("This is a message from the custom control.");
}
```

4. The event handler executes.

ClientWebService.java

```
@EventHandler(field = "myControl", eventSet = MyControl.MyEventSet.class, eventName = "onMessage")
protected void myControl_MyEventSet_onMessage(String aMessage) {
    System.out.println("Got message from myControl: " + aMessage);
}
```

The @EventHandler annotation provides the pathway that allows the control event method onMessage to invoke the client's event handler method. Notice that the @EventHandler contains all of the information necessary to indicate which event method the handler is sensitive to: the target control, the event set, and the particular event in that event set.

The event handler name can be anything (because the @EventHandler annotation does all of the work of sensitizing the handler to the appropriate event method). By convention we have named the event handler according to the following rule:

<control-reference-field-name>_<event-set-name>_<event-name>

Workshop for WebLogic uses this naming rule by default when a event handler is added to a client class using **Right-click > Insert > Control Event Handler**.

Note that the parameter set of the event method must match the parameter set of the event handler for the event handler to be successfully invoked. In the above example, both the event method and its handler have matching parameter sets, namely, one String parameter:

```
@EventSet
public interface MyEventSet {
    void onMessage(String aMessage);
}


@EventHandler(field = "myControl", eventSet = MyControl.MyEventSet.class, eventName = "onMessage")
protected void myControl_MyEventSet_onMessage(String aMessage) {
    ...
}
```

Control Event Set Definition

An event set definition in a control consists of two elements:

(1) An @EventSet declaration on the event set interface:

MyControl.java

```
@ControlInterface
public interface MyControl {

    public void requestMessage();

    @EventSet
    public interface MyEventSet {
        void onMessage(String aMessage);
   }

}
```

Multiple event methods may be defined in the @EventSet declaration:

MyControl.java

```
@ControlInterface
public interface MyControl {

    public void requestMessage();

    @EventSet
    public interface MyEventSet {
```

```
        void onMessage(String aMessage);
        void onReady(boolean boolReady);
    }

}
```

(2) A @Client declaration in the control implementation:

MyControlImpl.java

```
@Client MyEventSet eventSet;
```

For events to be sent, you must invoke the event method somewhere within the control implementation.

```
public void requestMessage() {
    eventSet.onMessage("This is a message from the custom control.");
}
```

Note that the event method definition has no body -- only the method signature, including the return type and any parameters.

It is common for event methods to have names that begin with *on* because the event handler in the client will be called *on* occurrence of the event.

Event Handler Definition

The client application is responsible for implementing the handler for a control's event method.

The following shows an example of a event handler as it might appear in a client application:

```
@EventHandler(field = "myControl", eventSet = MyControl.MyEventSet.class, eventName = "onMessage")
protected void eventHandler(String aMessage) {

    // do something with the message here ...

}
```

For the event handler to successfully listen for an associated event, the following two conditions must be fulfilled:

(1) The @EventHandler annotation must point at the appropriate control, event set, and event.

This means that the field attribute must refer to the control field as it is declared in the client. Suppose the control field is declared as so:

```
@Control
private MyControl myControl;
```

Then the @EventHandler must refer to this field:

```
@EventHandler(field = "myControl",
```

The @EventHandler must also refer to the EventSet and the particular event method as they are defined on the control. Suppose the control defines the following EventSet and event method:

```
@EventSet
public interface MyEventSet {
    void onMessage(String aMessage);
```

```
}
```

Then the @EventHandler must refer to the EventSet and event method as so:

```
@EventHandler(field = "myControl", eventSet = MyControl.MyEventSet.class, eventName = "onMessage")
```

(2) The event method and its handler must have matching parameter sets. That is, the number of parameters, their order, and their types must match. For example, the following event method and handler have matching parameter sets.

```
@EventSet
public interface MyEventSet {
    void onMessage(String aMessage, boolean status);
}

@EventHandler(...)
protected void eventHandler(String aMessage, boolean status) {
```

Limitations for External Events

External events are supported only for web service clients. Other clients cannot cannot handle event notification over a network protocol.

Adding Event Sets and Event Handlers with Workshop for WebLogic

The following commands are available for adding event sets and event handlers.

To Add an Event Set

To add an event set to a control, right-click anywhere within the control interface source view and select **Insert > Event Set**. By default an event set interface named NewEventSet with one event method, named onEvent1(), is added to the control interface:

SomeControl.java

```
@EventSet
public interface NewEventSet {
    void onEvent1();
}
```

And the corresponding @Client declaration is added to the control implementation file:

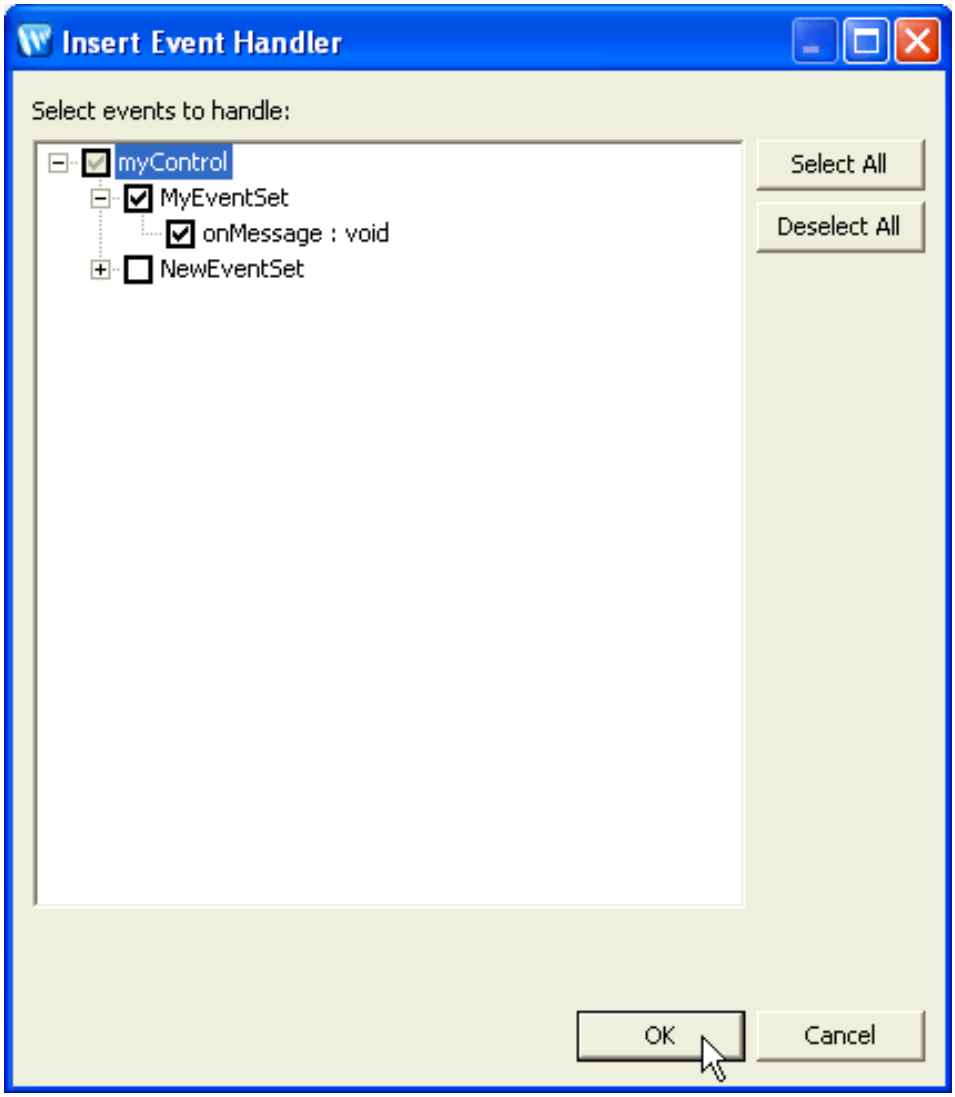
SomeControlImpl.java

```
       @Client
NewEventSet eventSetClient;
```

You must manually rename default the event set and event method names. Add additional event methods as necessary.

To Add an Event Handler

To add an event handler to a control client, right-click anywhere in source view and select **Insert > Control Event Handler**. A dialog will appear presenting you with the controls declared on the client and the events exposed by those controls. Select the control, event set, and particular event method to construct an event handler for that event method.



An event handler will be added to your client's source:

```
@EventHandler(field = "myControl", eventSet = MyControl.MyEventSet.class, eventName = "onMessage")
protected void myControl_MyEventSet_onMessage(String aMessage) {

}
```

Related Topics

Designing Asynchronous Interfaces

Apache Beehive documentation: @EventSet

Apache Beehive documentation: @EventHandler

Apache Beehive documentation: @Client

Handling Control Method Exceptions

The designer of a custom 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. If you do not handle the exception, the exception will be passed on to the client of your control. 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.

Related Topics

None

Control Transactions

Ordinary control methods, event set methods, and web service control callback methods are transaction-enabled. This topic provides an overview of transaction support in controls.

Transaction Behavior

The transaction behavior of a method, event set method, or callback method is specified by the @TransactionAttribute annotation. The following example shows a web service callback decorated with @TransactionAttribute.

```
public interface MyServiceControl extends ServiceControl
{

    @ServiceControl.HttpSoapProtocol
    @ServiceControl.SOAPBinding(style = ServiceControl.SOAPBinding.Style.RPC, use = ServiceControl.
SOAPBinding.Use.ENCODED)
    @EventSet(unicast=false)
    public interface Callback    {
        @TransactionAttribute(TransactionAttributeType.REQUIRES)
        public void onCallback(java.lang.String message);
    }

    public void requestCallback();
}
```

In the above example, TransactionAttributeType.REQUIRES means that if a transaction was already started in the callback thread, that transaction will be used; otherwise a new transaction will be started just before the callback method is called. Depending on the result of the operation, the transaction could either be committed, rolled-back, or marked as rollback (if started elsewhere).

The @TransactionAttribute has 6 legal settings:

- `REQUIRED` - If the client is running within a transaction, the method/callback executes within the client's transaction. If the client is not associated with a transaction, the interceptor starts a new transaction before running the method/callback. Most control runtime-managed transactions use `REQUIRED`. This is the default setting.
- `REQUIRES_NEW` - If the client is running within a transaction, the control runtime suspends the client's transaction, starts a new transaction, delegates the call to the method/callback, and finally resumes the client's transaction after the method/callback completes. If the client is not associated with a transaction, the control runtime starts a new transaction before running the method/callback.
- `MANDATORY` - If the client is running within a transaction, the method/callback executes within the client's transaction. If the client is not associated with a transaction, the control runtime throws a ControlException. Use the `MANDATORY` attribute if the method/callback must use the transaction of the client.
- `NOT_SUPPORTED` - If the client is running within a transaction and invokes the control bean's method/callback, the control runtime suspends the client's transaction before invoking the method/callback. After the method/callback has completed, the control runtime resumes the client's transaction. If the client is not associated with a transaction, the control runtime does not start a new transaction before running the method/callback. Use the `NOT_SUPPORTED` attribute for method/callbacks that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

SUPPORTS - If the client is running within a transaction and invokes the control bean's method/callback, the method/callback executes within the client's transaction. If the client is not associated with a transaction, the control runtime does not start a new transaction before running the method/callback. Because the transactional behavior of the method/callback may vary, you should use the SUPPORTS attribute with caution.

- NEVER - If the client is running within a transaction and invokes the control bean's method/callback, the control runtime throws a ControlException. If the client is not associated with a transaction, the control runtime does not start a new transaction before running the method/callback.

Exception Handling

Checked exceptions (those exceptions that are a sub-class of Exception, not RuntimeException) are handled differently from other exceptions.

You can control the response to checked exceptions with the annotation attribute `rollbackOnCheckedException`:

```
@TransactionAttribute( rollbackOnCheckedException=<boolean> )
```

If `rollbackOnCheckedException` is true, then the transaction will be rolled back (or be marked for rollback) when a control method or callback method results in a checked exception. By default, `rollbackOnCheckedException` is set to false. The default behavior is that a transaction will not be automatically rolled back, even if the method/callback invocation results in a checked exception being thrown. If invoking a control method/callback results in a *system* exception (exceptions that are a sub-class of java.lang.RuntimeException and java.lang.Error), the transaction will be rolled back (or marked for rollback) automatically.

Related Topics

@TransactionAttribute

Using System Controls

BEA Workshop for WebLogic Platform's system controls make it easy to access J2EE 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 controls to encapsulate business logic in a reusable component. For information on creating custom controls, see Custom Controls.

Topics Included in This Section

Timer Control
: Describes how to use the timer control to run code at specific time intervals.

Service Control
: Discusses how to use a web service control to access web service operations through method calls.

EJB Control
: Describes how to use the EJB control to access an Enterprise JavaBean.

JMS Control
: Describes how to access a JMS message queue or topic with a JMS control.

JDBC Control
: Describes how to access use a JDBC control to access a database.

Related Topics

Working with Beehive Controls

Timer Control

A timer monitors elapsed time:

- When a specific relative amount of time has passed (e.g., an hour)
- When a specific absolute moment has passed (e.g., midnight of February 12, 2015)
- At recurring intervals (e.g., every ten seconds)
- After an absolute time or a relative amount of time has passed, in recurring intervals thereafter (e.g., after an hour, then every five minutes thereafter OR after noon on Tuesday, then every hour thereafter)

The BEA Workshop for WebLogic Platform (Workshop for WebLogic) timer control allows you to easily incorporate timer functionality into a web service.

To learn about other Workshop for WebLogic controls, see Using WebLogic System Controls.

Topics Included in this Section

Tutorial: Creating a Web Service with Timer Control

Walks through a step-by-step description of how to implement a simple timer control that calls back every two seconds.

Overview: Timer Control

Discusses how the timer works, how times are specified, general techniques for working with a timer control.

Creating and Configuring a Basic Timer Control

Describes how to declare/instantiate a timer control, configure settings, how to set up single-instance timers and recurring timers, timer control methods and properties.

Setting up Web Service Operations to Access a Timer Control

Explains how to create web methods to start/stop a timer control and an event handler to process the callback(s) when the timer elapses.

Changing Timer Settings Dynamically

Explains how to use methods on `TimerControlBean` as an alternative to using the `TimerControl` interface and the **Annotations** view to change settings dynamically.

Using a Timer Control

Provides a detailed feature summary.

Related Topics

TimerControl Interface

Using WebLogic System Controls

Overview: Timer Control

The timer control can notify a web service in the following ways:

1. A specific (absolute) timeout has passed (e.g., January 23, 2012 midnight)
2. A relative timeout has passed (e.g., an hour and seventeen minutes)
3. A recurring time interval has elapsed (e.g., every 3 minutes)
4. Both an intial timeout and a recurring interval (e.g., after one hour, every 5 minutes OR after January 1, 2010, every hour thereafter)

The timer control can perform only one of these tasks at a time. If you wish to do more than one timer task, you may create additional timer controls. Alternately, you may run a timer and on completion, change the timer settings and start the timer again.

If a timer is created with NO time specified, then it will perform the default timer--a relative timeout of 0 seconds.

All timer controls are instances of the `com.bea.control.TimerControl` base class or the `com.bea.control.TimerControlBean` base class. A timer control is declared directly in a .java file. Timer controls are based on the EJB 2.1 timer service. Timer controls make a best-effort to do a callback to the client when a timer elapses, suitable for application timers. However timer controls are not a true real-time time service. Timer controls are allowed only in conversational (stateful) web services.

Creating and Configuring a Basic Timer Control

To create a basic timer control, the IDE provides commands that will insert the declaration/annotation for a `TimerControl` into the code for your web service and set attributes.

To specify a relative timeout and/or recurring time, set the timer property values in the **Annotations** view.

To specify an absolute timeout, call the `setTimeoutAt` method on the timer control instance before starting the timer. You can also stop the timer, reset the absolute timeout value and start the timer again. If a recurring time value has been specified previously from **Annotations** view, the recurring timer begins after the initial absolute time has elapsed.

To define a payload to be returned on the callback, call the `setPayload` method in your web service before starting the timer.

To set other configuration values, you can use the **Annotations** view to set properties that optionally specify the type of timer intervals (coalesced vs. non-coalesced), whether the timer is transactional, and the JNDI Provider URL and Context Factory.

Setting up Web Service Operations to Access a Timer Control

To run the timer, once the timer has been declared and configured, you must create a conversational web service and then within your web service you create operations (web methods) that start/stop/restart the timer, based on the annotation settings or based on method calls. The timer control requires a stateful conversational web service, so the web methods must be set up appropriately.

To specify what happens when the timer elapses, you must set up an event handler in the web service to process callbacks.

Changing Timer Settings Dynamically

To change settings programatically, (other than the absolute timeout value), you must use `TimerControlBean`. The bean is generated by the Beehive control framework at build time and implements the `TimerControl` interface, providing additional properties and methods that allow you to control all of the properties in the **Annotations** view at run-time.

Related Topics

Using WebLogic System Controls

Timer Control

TimerControl Interface

Timer Control Reference

Tutorial: Creating a Web Service with Timer Control

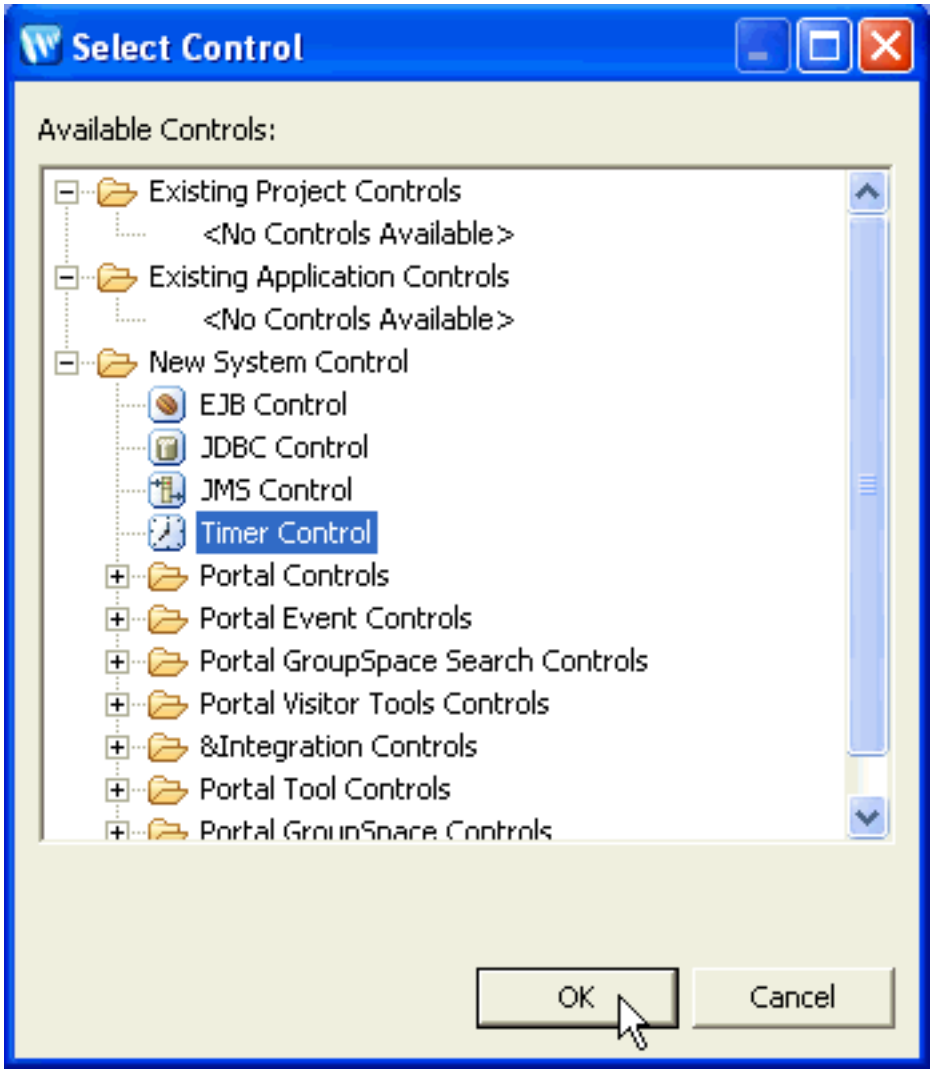
Creating and Configuring a Basic Timer Control

To set up a timer control, you must insert the initial timer control into the web service and then configure the settings for the control.

Using IDE Commands to Insert a Timer Control

To declare a new timer control, follow these steps:

1. Be sure that you are in the **J2EE** perspective. The current perspective is displayed just below the toolbar at the top of the workbench. If you are not in **J2EE** perspective, click **Window > Open Perspective > J2EE** to switch perspectives.
2. Double click the .java file name in the **Project Explorer** view at the left to open the Java source file in the editor.
3. Right click on the editor pane and choose **Insert > Control.** The **Select Control** dialog opens.
4. Expand **New System Control** and choose **Timer Control**. Click **OK**.



5. The timer control annotation is inserted into the code:

   ```
   @Control
   private TimerControl timerControl1;
   ```

   and the default name of the new control (**timerControl1**) is highlighted.

When you create a timer control, Workshop for WebLogic also inserts a line at the top of your .java file to import the timer control class, `com.bea.control.TimerControl`.

Note that you cannot create a new timer control in a Beehive page flow or a Java Server Page (JSP file), because the timer control defines a callback, the `onTimeout` method, but page flow and JSP pages cannot accept callbacks.

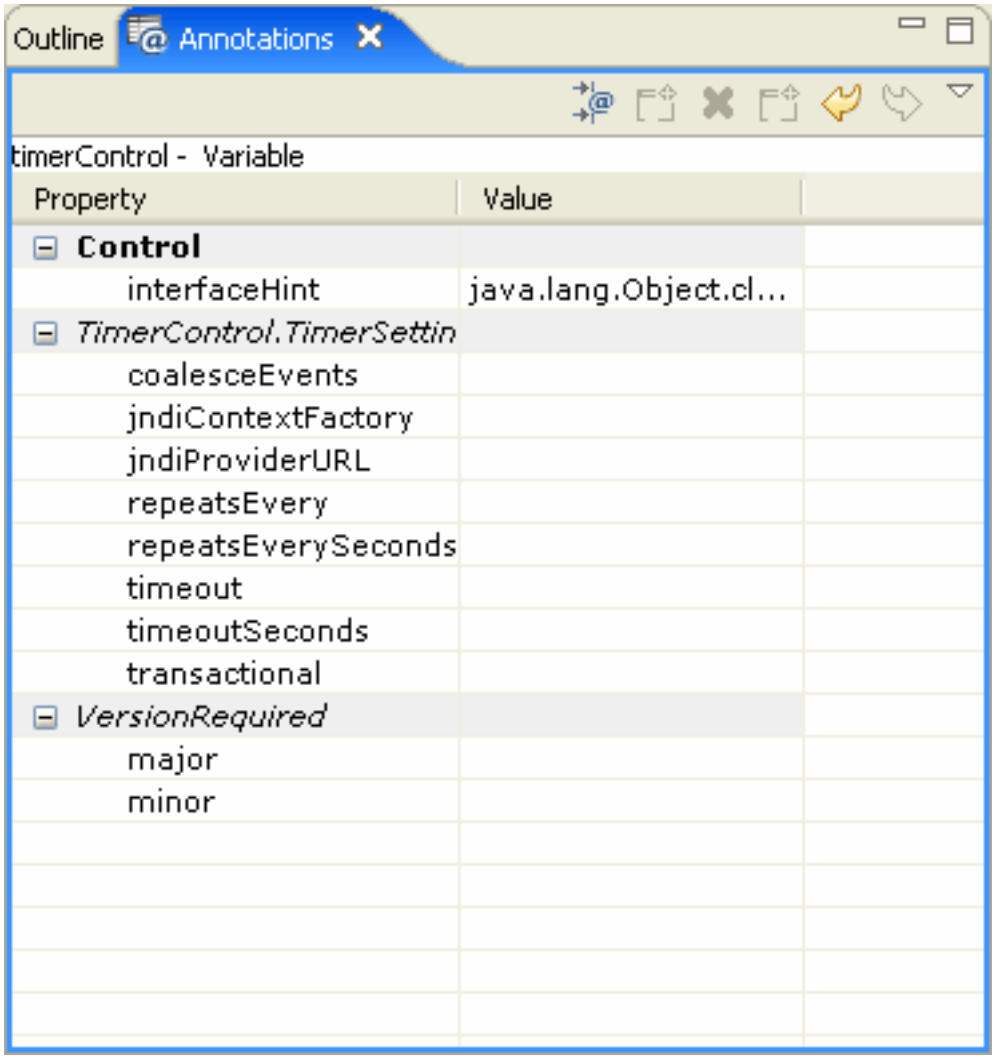
Before using a timer control, you must specify when the timer elapses. If you do not specify when the timer elapses, the timer control will time out immediately (i.e., it will default to zero seconds). Elapsed time is specified in an additional annotation `@TimerControl.TimerSettings()` that is inserted into your code. Typically you will not edit the annotation manually but instead will use the **Annotations** view to set property values.

Specifying Relative Timer Values through the Annotations View

For relative times, you can specify the timeout or the repeated time interval from the **Annotations** view, which is available in **J2EE** perspective. Absolute times must be specified by calling the control's method(s) as described in Setting an Absolute Timer with a Method Call.

To set a relative timeout or recurring time interval:

1. From the editor window, as long as the control name is highlighted, the **Annotations** view at the right displays the properties of the new control. To set a relative timeout, click the **timeout** or **timeoutSeconds** field and enter the amount of time you want to elapse before the timer fires. This timer will elapse only once. The time value is a string, as described below.



2. To set a recurring timer, in the **repeatsEvery** or **repeatsEverySeconds** field, specify the interval between firings after the timer fires the first time. The time value is a string, as described below.

Note that you should set ONLY ONE of timeout, timeoutSeconds and only ONE of repeatsEvery, or repeatsEverySeconds. If you set both values, the xxxSeconds value will be used.

Entering Relative Time Value Strings

When entering timeout or recurring time values, you must specify the relative time as a text string--integers followed by case-insensitive time units. These time units can be separated by spaces.

For example,

```
1 hour 30 min
```

means one hour and 30 minutes and

```
30 s
```

means 30 seconds.

The following time string is a valid duration specification that exercises all the time units, spelled out fully:

```
99 years 11 months 13 days 23 hours 43 minutes 51 seconds
```

This example creates a timer control which will elapse 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.

Text strings may also be in ISO 8601 extended format with the string "p" (case insensitive) 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. (http://www.w3.org/TR/xmlschema-2/#duration)

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

```
P99Y11Mo13D23H43M51S
```

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 28 in a normal year, February 29 in a leap year).

Using the Properties of the TimerControl Interface

The timer control interface has three properties:

- `timeoutAt`: initial timeout interval (date)
- `isRunning`: tests if the timer control is currently running (boolean)
- `payload`: extra data that you can set before starting a timer; this data is passed back to the event handler during each timeout event (serializable)

Property	Type	Getter	Setter	Description
timeoutAt	Date	Date **getTimeoutAt**()	void **setTimeoutAt** (Date arg0)	Initial timeout interval (relative time). Format is java.util.Date.
isRunning	boolean	boolean **isRunning**()		Whether the timer control is currently running (true=running, false=stopped).
payload	Serializable	Serializable **getPayload**()	void **setPayload** (Serializable arg0)	Extra data to be passed back to the event handler in each timer callback.

Setting a Timer Payload with a Method Call

To set a payload that will be supplied to the callback handler during a timeout event, call the

`setPayload(Serializable payload)`

method before starting the timer. Calling this method after the timer is started will have no effect. Since the payload is serializable, you must be careful that the class that the serializable represents still exists, so that the object can be deserialized.

Setting an Absolute Timer with setTimeoutAt()

You can configure a timer control to fire at an absolute time by setting the `TimeoutAt` property of the `TimerControl` interface by calling `setTimeoutAt()`.

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 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-every attribute is set to a value greater than zero. If the repeats-every attribute is set to zero, then the `getTimeoutAt` method returns the value set by the `setTimeoutAt` method or the

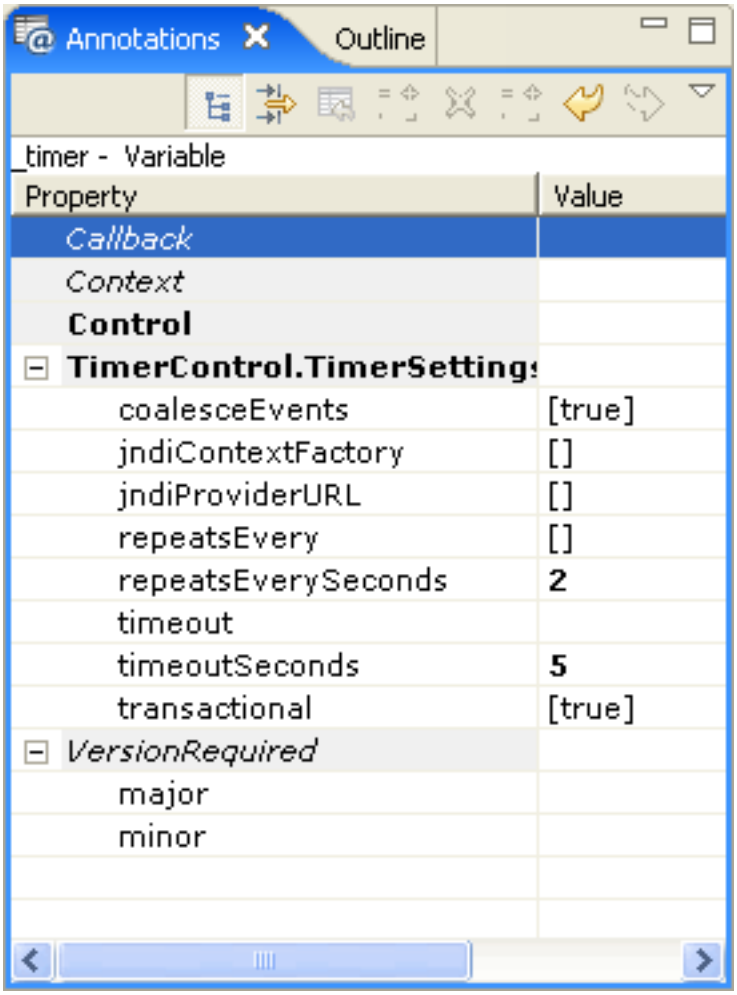
value set in the timeout attribute. If you call the `getTimeoutAt` method from within the `onTimeout` callback handler, the first timeout has already fired, so `getTimeoutAt` will return either the time of the next timeout or the time of the first timeout if the timer is not set to repeat.

The following code snippet 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.

```
@Control
private TimerControl tTimer;
@WebMethod()
@Conversation(value= Conversation.Phase.START)
public void StartTimer()
{
    Calendar cd = new GregorianCalendar();
    cd.set(cd.SECOND, 30);
    tTimer.setTimeoutAt(cd.getTime());
    tTimer.setRepeatsEvery(60);
    tTimer.start();
}
```

Setting Other Timer Control Behavior

You can specify other settings of a timer control in **J2EE** perspective by setting the control's properties in the **Annotations** view. For example, if you highlight the name of the timer control instance named `_timer` in the code editor, the **Annotations** view displays the following properties:



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

Property	Description / Values
`timeout`	Time until the timer control fires the first time, once started (default: 0 seconds).
`timeoutSeconds`	Time in seconds until the timer control fires the first time, once started (default: 0).
`repeatsEvery`	How often the timer control should fire after the first time (default: 0 seconds i.e., non-recurring).
`repeatsEverySeconds`	How often the timer control should fire after the first time (default: 0 i.e., non-recurring).

coalesceEvents	Specifies whether multiple undelivered firing events of a Timer control are delivered as a single onTimeout (true) or as separate callbacks (false). Default: true. At times, a Timer control may be unable to deliver one or more callbacks to its referring service. This may occur because the referring service is busy or because high system load delays delivery. A set of undelivered callbacks may accumulate. If the coalesceEvents attribute is true, these accumulated callbacks are collapsed into a single callback when the service becomes available. If coalesceEvents is false, the accumulated callbacks are delivered individually.
transactional	True if timers participate in transactions and are durable (default: true).
jndiContextFactory	JNDI context factory class (default: none).
jndiProviderURL	JNDI provider URL (default: none).

Understanding the Timer Control Declaration

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

```
import com.bea.control.TimerControl;
...
@TimerControl.TimerSettings(repeatsEverySeconds=5)
@Control
private TimerControl delayTimer;
```

The actual attributes that are present on the `@TimerControl()` annotation depend on the values you specify for the properties in the **Annotations** view.

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

The `@TimerControl()` annotation controls the behavior of the timer control. All of the attributes of the `@TimerControl()` annotation are optional and have default values.

The timer control, named `delayTimer` in the example above, is declared as an instance of `TimerControl`.

Related Topics

Using WebLogic System Controls

Timer Control

TimerControl Interface

Timer Control Reference

Tutorial: Creating a Web Service with Timer Control

Setting up a Web Service to Access a Timer Control

Timer controls are used within conversational, stateful web services. Conversational web services have three phases: start, continue, and finish.

To use a timer in a conversational web service, after declaring the timer control as described in Setting up a Timer Control, you must implement a conversational web service to access the timer control. The minimum requirements to access a timer control are:

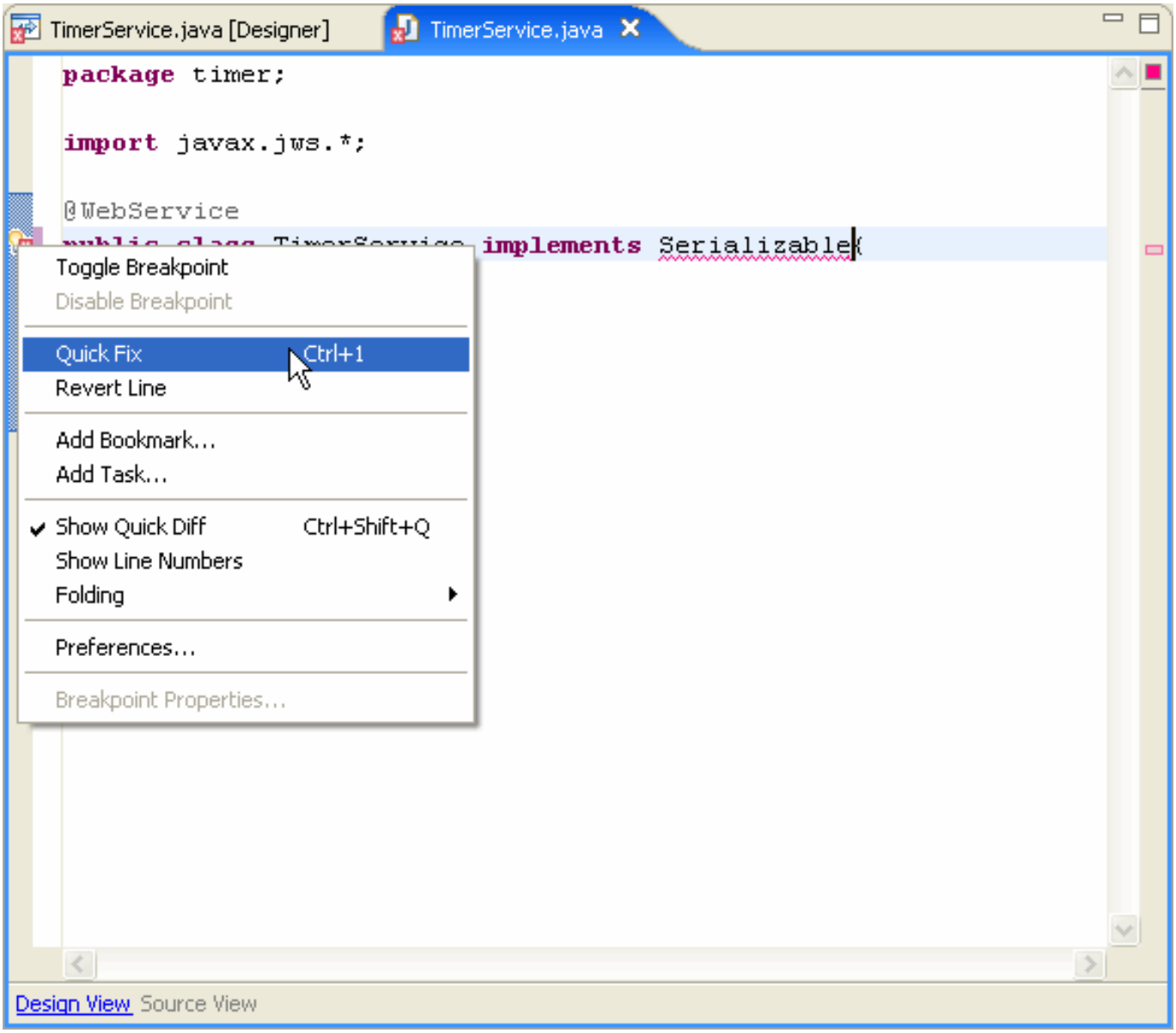
1. Ensure that the web service implements the Serializable interface.
2. Define at least a start and finish operation.
3. Define an event handler to receive the callback(s) from the timer control when the timer elapses.

Setting the Web Service to Implement Serializable

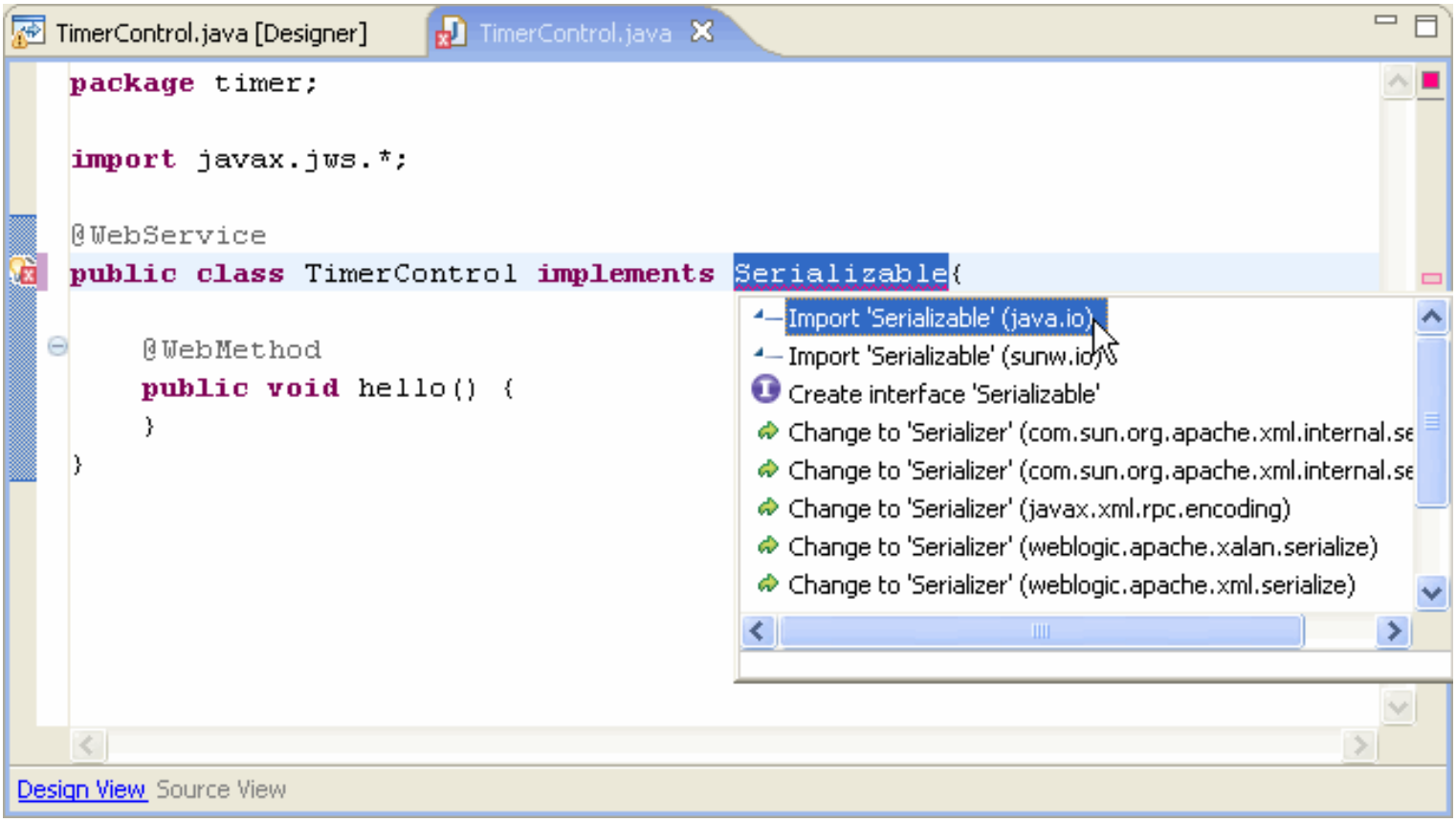
Conversational web services must implement the java.io.Serializable interface. To set this in your web service, on the class definition line for your web service, insert `implements Serializable` so that the class definition looks something like this:

```
public class TimerService implements Serializable {
```

This will cause an error marker to appear in the marker bar on the class declaration line. Right click on the error marker and choose **Quick Fix**.



Right click on the error marker and choose **Quick Fix**. The **Quick Fix** pull-down will appear:



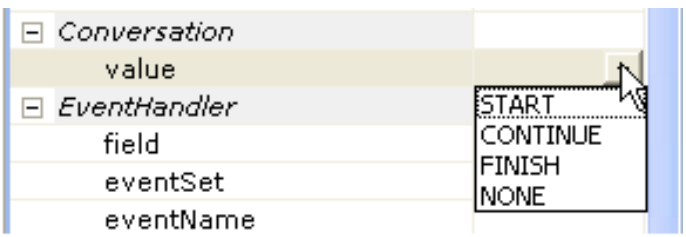
Click **Import Serializable** and press Enter and a new import line will be generated to resolve the error.

Defining Start/Stop Conversational Web Methods

A conversational web service differs from a regular web service in only one way: the **Conversation** property is set to **START, CONTINUE** or **FINISH**. A web service typically has three operations:

1. An operation to start the timer which has **Conversation** set to **START** and contains a call to the timer control's start() method.
2. An operation to stop the timer which has **Conversation** set to **STOP** and contains a call to the timer control's stop() method.
3. An optional operation to restart the timer which has **Conversation** set to **CONTINUE** and contains a call to the timer control's restart() method. This is primarily used for resetting a relative timeout or recurring time intervals.

To define a conversational web method, right click on the code editor window and click **Insert > Web Method.** With the name of the new web method highlighted, click on the **Conversation** setting in the **Annotations** view. Use the pulldown to set the conversational state for the method.



Calling the 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`, `timeoutSeconds`, `repeatsEvery` or `repeatsEverySeconds` attribute has passed. The `start()` method can be called in either the START or CONTINUE phase of the conversation.
- `restart()`: stops the timer control and starts it again. This method is only useful when working with relative timeout or recurring time intervals, since the timer is restarted with the same parameters.
- `stop()`: stops the timer control from firing again. Calling `start()` will start the timer again. The `stop()` method can be called in either the FINISH or CONTINUE phase of the conversation. Be sure to call the `stop()` method when the conversation ends. Note that if you do not call the `stop()` method, the container will automatically terminate the timer for you.

Setting up an Event Handler for Timer Callbacks

In addition to the web methods that start/stop the timer, the web service must provide for callbacks when the timer elapses.

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

The callback event handler takes two parameters: the time in seconds since the timer was started and the payload. 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 event handler for a timer control's `onTimeout` callback, right click on the code window and choose **Insert > Control Event Handler**. Workshop for WebLogic creates a callback event handler and places the cursor in the callback event handler.

Related Topics

Using WebLogic System Controls

Timer Control

TimerControl Interface

Timer Control Reference

Tutorial: Creating a Web Service with Timer Control

Changing Timer Settings Dynamically

If you want to reconfigure timer settings dynamically, you must use `TimerControlBean` instead of the basic `TimerControl` interface.

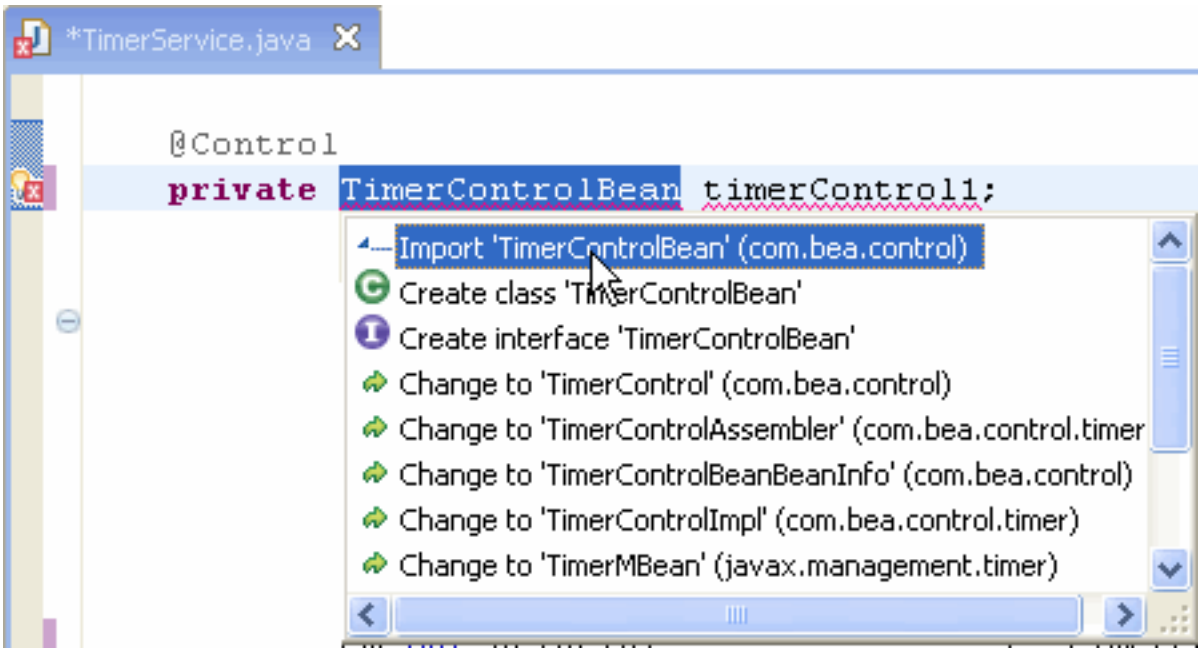
`TimerControlBean` provides additional methods to programmatically get/set all of the parameter values displayed on the **Annotations** view.

Using the TimerControlBean

When you use the **Insert > Control** command to create a timer control, the default annotation that is inserted uses the TimerControl interface.

```
@Control
private TimerControl timerControl;
```

To use `TimerControlBean` instead, simply change `TimerControl` to `TimerControlBean` in the timer control declaration line. This will cause an error marker to appear in the marker bar on the class declaration line because `TimerControlBean` requires an additional import. Right click on the error marker and choose **Quick Fix**. The **Quick Fix** pull-down will appear:



click on **Import 'TimerControlBean'** and press Enter. A new import line

```
import com.bea.control.TimerControlBean;
```

will be inserted at the top of your code and the problem marker will disappear.

Calling Methods on the TimerControlBean

TimerControlBean implements the TimerControl interface and also defines the following additional methods to get/set timer properties to get/set the property values:

Method	Description
String **getTimerSettingJNDIContextFactory**()	Returns the JNDI Context Factory.
String **getTimerSettingJNDIProviderURL**()	Returns the JNDI Provider URL.
String **getTimerSettingRepeatsEvery**()	Returns the relative time string for the recurring timer setting.
Long **getTimerSettingRepeatsEverySeconds**()	Returns the number of seconds for the recurring timer.
String **getTimerSettingTimeout**()	Returns the relative time string for the timeout.
Long **getTimerSettingTimeoutSeconds**()	Returns the number of seconds for the timeout.
void **setTimerSettingJNDIContextFactory**(String arg0)	Sets the JNDI Context Factory.
void **setTimerSettingJNDIProviderURL**(String arg0)	Sets the JNDI Provider URL.
void **setTimerSettingRepeatsEvery**(String arg0)	Sets the recurring timer by specifying a relative time string.
void **setTimerSettingRepeatsEverySeconds**(Long arg0)	Sets the recurring timer to the specified number of seconds.
void **setTimerSettingTimeout**(String arg0)	Sets the relative timeout to the relative time string.
void **setTimerSettingTimeoutSeconds**(Long arg0)	Sets the relative time to the specified number of seconds.

Note that the **setTimerSettingxxx** methods have no effect if the timer is already started. To set values, you must stop the timer, call the method to set the value, and start the timer. This will not create a new timer object, but will simply restart the existing control.

The **getTimerSettingxxx** methods work whether the timer is running or stopped.

Related Topics

Using WebLogic System Controls

Timer Control

TimerControl Interface

Timer Control Reference

Tutorial: Creating a Web Service with Timer Control

Using a Timer Control

IDE Commands to Work with Timer Control

To insert a timer control, right click on the code editor and choose **Insert > Control**.

To insert a timer control using the TimerControlBean:

1. Right click on the code editor and choose **Insert > Control**.
2. Manually edit the annotation to change `TimerControl` to `TimerControlBean`.
3. Do the required import for `TimerControlBean` by right clicking on the problem marker in the marker bar beside the annotation line and and choosing **Quick Fix**. Use the **Quick Fix** menu to do the necessary import for `TimerControlBean`.

To display the properties of the timer control (or timer control bean) in Annotations view,

1. Be sure that **J2EE** perspective is open (as displayed just below the toolbar at the top of the workbench window); if it is not open, click **Window > Open Perspective** to open **J2EE** perspective.
2. Double click on the timer control (or bean) name. The properties of the annotation will be displayed in the **Annotations** view.

The IDE does not validate time strings that are entered in the **Annotations** view. Times are validated only at run-time.

Requirements for Web Services that use a Timer Control

Timer controls can only be used within a web service that is conversational and implements Serializable. See Designing Conversational Web Services for more information.

TimerControl Annotation

See the TimerControl.TimerSettings annotation for a description of the annotations to set timer control parameters

The TimerControl Interface

The following properties and methods are provided on the `TimerControl` interface.

Properties

Property	Type	Getter	Setter	Description
`running`	boolean	`boolean` **`isRunning`**`()`		Tests if the timer is currently running.
`timeoutAt`	Date	`Date` **`getTimeoutAt`**`()`	`void` **`setTimeoutAt`**`(Date arg0)`	Initial timeout (absolute date);date is java.util.Date. Calling `setTimeoutAt()` replaces any existing relative timer with the specified absolute time. Setting an absolute date in the past causes an immediate callback.
`payload`	Serializable	`Serializable` **`getPayload`**`()`	`void` **`setPayload`**`(Serializable arg0)`	Extra data that is passed back to event handler on a callback. If you specify a payload, be sure that when the timer callback occurs, the object still exists, since the payload is serializable.

Methods

There are three methods for TimerControl.

Method	Description
`void` **`start`**`()`	Starts timer operation. The timer control will fire after the period specified by the `timeout`, `timeoutSeconds`, `repeatsEvery` or `repeatsEverySeconds` attribute has passed. The `start()` method can be called in either the START or CONTINUE phase of the conversation. Calling the `start()` method a second time on a timer has no effect. If a timer has been stopped, calling `start()` starts the timer again.
`void` **`stop`**`()`	Stops the timer control from firing again. Calling `start()` will start the timer again. The `stop()` method can be called in either the FINISH or CONTINUE phase of the conversation. Be sure to call the `stop()` method when the conversation ends. Note that if you do not call the `stop()` method, the container will automatically terminate the timer for you. Calling the `stop()` method for a stopped event timer does not generate an exception. Calling the `stop()` method more than once on a timer does not generate an exception.
`void` **`restart`**`()`	Stops the timer control and starts it again. This method is only useful when working with relative timeout or recurring time intervals, since the timer is restarted with the same parameters.

Events

There is one event generated by the timer control

Event		Description
`onTimeout`	`void callback(long timeout, Serializable payload)`	Occurs when the timer expires.

TimerControlBean

`TimerControlBean` implements the `TimerControl` interface and provides the following **additional** methods and properties.

Properties

Property	Type	Getter	Setter	Description
`coalesceEvents`	`boolean`	`boolean` **`isTimerSettingCoalesceEvents`**`()`	`void` **`setTimerSettingCoalesceEvents`** `(boolean arg0)`	Whether the timer control is set to use the coalesce method for recurring events.
`transactional`	`boolean`	`boolean` **`isTimerSettingTransactional`**`()`	`void` **`setTimerSettingTransactional`** `(boolean arg0)`	Whether the timer control is currently running in transactional mode.

Methods

The following additional methods are provided to get/set the property values displayed in the **Annotations** view:

Method	Description
`String` **`getTimerSettingJNDIContextFactory`**`()`	Returns the JNDI Context Factory.
`String` **`getTimerSettingJNDIProviderURL`**`()`	Returns the JNDI Provider URL.
`String` **`getTimerSettingRepeatsEvery`**`()`	Returns the relative time string for the recurring timer setting.
`Long` **`getTimerSettingRepeatsEverySeconds`**`()`	Returns the number of seconds for the recurring timer.
`String` **`getTimerSettingTimeout`**`()`	Returns the relative time string for the timeout.
`Long` **`getTimerSettingTimeoutSeconds`**`()`	Returns the number of seconds for the timeout.
`void` **`setTimerSettingJNDIContextFactory`**`(String arg0)`	Sets the JNDI Context Factory.
`void` **`setTimerSettingJNDIProviderURL`**`(String arg0)`	Sets the JNDI Provider URL.
`void` **`setTimerSettingRepeatsEvery`**`(String arg0)`	Sets the recurring timer by specifying a relative time string.
`void` **`setTimerSettingRepeatsEverySeconds`**`(Long arg0)`	Sets the recurring timer to the specified number of seconds.
`void` **`setTimerSettingTimeout`**`(String arg0)`	Sets the relative timeout to the relative time string.
`void` **`setTimerSettingTimeoutSeconds`**`(Long arg0)`	Sets the relative time to the specified number of seconds.

Note that the **setTimerSettingxxx** methods have no effect if the timer is already started. To set time values, you must stop the timer, call the method to set the value, and start the timer. This will not create a new timer object, but will simply restart the existing control.

If both **repeatEverySecond** and **repeatEvery** are set, the timer does not generate an error, it uses the **repeatEverySecond** value.

If both **timeout** and **timeoutSeconds** are set, the timer does not generate an error, it uses the **timeoutSeconds** value.

Setting an absolute date in the past or a negative time value in either the `setTimeout` or `setTimeoutSeconds` methods causes an immediate callback.

The **getTimerSettingxxx** methods work whether the timer is running or stopped.

The following methods are for internal use only and should NOT be used:

Methods for Internal Use Only
`getCallbackListener`
`addCallbackListener`
`removeCallback`
`removeCallbackListener`
`getControlImplementation`
`setControlImplementation`

Events

There are no additional events generated by `TimerControlBean`.

Relative and Absolute Time Strings

Relative Time Strings

Relative time values are used for relative timeouts and for recurring times. Relative time values are set either through the **Annotations** view or by calling a method on `TimerControlBean`.

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:

```
@TimerControl.TimerSettings(timeout="99 years 11 months 13 days 23 hours 43 minutes 51 seconds")
@Control()
```

```
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.

Text strings may also be in ISO 8601 extended format with the string "p" (case insensitive) 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. (http://www.w3.org/TR/xmlschema-2/#duration)

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

```
@TimerControl.TimerSettings(timeout="P99Y11Mo13D23H43M51S")
@Control()
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).

Absolute Time Strings

To specify that the timer fires at a specific (absolute time), you must use the `setTimeoutAt` method.

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. Specific times are set as for with a **java.util.Date** object (Java 1.5). Since the **TimerSetting** annotation does not allow you to specify a timeout in **java.util.Date** format, you must use the `setTimeoutAt` method. For more information, see Using Methods of the TimerControlBean Interface to Set Parameters.

Methods on TimerControl for Backward Compatibility

An additional group of methods are provided on the TimerControl interface to provide backward compatibility with WebLogic Workshop 8.1 applications. These methods are no longer recommended. The backward compatibility methods are:

Method	Description
`long` **`getTimeout`**`()`	Get the current relative timeout value.
`String` **`getRepeatsEvery`**`()`	Returns the recurring time interval in ISO 8601 format (Pxxx).
`boolean` **`getCoalesceEvents`**`()`	Returns whether the timer is running in coalesced or non-coalesced mode. **Deprecated**.
`void` **`setCoalesceEvents`**`(boolean coalesce)`	Turns on coalesced mode.
`void` **`setRepeatsEvery`**`(long seconds)`	Sets the recurring timer by specifying a recurring time in seconds.
`void` **`setRepeatsEvery`**`(String interval)`	Sets the recurring timer to the specified interval string.
`void` **`setTimeout`**`(String arg0)`	Sets the relative timeout to the relative time string.
`void` **`setTimeout`**`(long seconds)`	Sets the relative time to the specified number of seconds.

In WebLogic Workshop 8.1, a call to the start() method on a timer that was already started could potentially generate new timer events. In Workshop for WebLogic 9.2 and 10.0, calling start() on a timer control that has already been started has no effect.

Caveats and Implementation Notes

The file **weblogic_timer_control.jar** contains the implementation of the timer control. This file defines the control interface and implementation and a stateless session bean. This JAR is deployed automatically when a web services project is created.

The timer control implementation is provided by **com.bea.wlw.control.timer.EjbTimerControlImpl** which uses the EJB 2.1 Timer Service. A control bean delegates the creation and canceling of timers through the stateless session bean, using an EJB Local interface. When a timer is expired, the `ejbTimeout` method on the session bean will be invoked by the EJB container. The

`ejbTimeout` method in turn retrieves the ControlHandle that was stored with the timer when the timer was created, and uses the ControlHandle to dispatch the callback event to the target TimerControl. This implementation is agnostic of what container the TimerControl is running in.

Since timer controls are based on the EJB 2.1 timer service, they are a best-effort to do a callback to the client as requested, not a true real-time time service.

Related Topics

Using WebLogic System Controls

Timer Control

TimerControl Interface

Timer Control Reference

Tutorial: Creating a Web Service with Timer Control

Service Control

A service control is a proxy for a web service. The web service client uses this proxy to access the web service. Using a service control allows the web service client to access the operations of a web service through simple method calls to the service control. A service control makes it easy to access an external web service from a Workshop for WebLogic client application. You can create a service control for any web service that publishes a WSDL file.

> **Note**: You should not use a service control to invoke a web service that resides in the **same** application. Invoking a web service via a 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.

Topics Included in this Section

Overview: Service Controls and Web Service Clients

Introduces public contracts, service clients, and other aspects of service controls.

Creating and Using a Service Control

Explains how to create a new service control from a WSDL file, how to add an existing Web Service control, and how to invoke the control from a web service client.

Handling Web Service Callback Messages

Describes how to handle callback messages sent from web services to a web service control.

Related Topics

Web Service Callbacks

Overview: Service Controls and Web Service Clients

A **service control** provides web service clients with easy access to a web service. The service control is provided as one of the system controls. When using a service control, you can invoke a web service operation by simply calling a method of the service control. The service control manages the SOAP message exchange with the web service and returns the results of the web service operation.

All service controls are interfaces that extend the `com.bea.control.ServiceControl` base class.

A 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 service control, you can connect to any web service for which a WSDL file is available, whether or not it was built using Workshop for WebLogic.

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 Workshop for WebLogic is the collection of all methods marked with the `@WebMethod` annotation plus all members of the `Callback` interface. Each public contract is completely defined in the Java source file for the web service. When you generate a WSDL file from a web service, 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 Workshop for WebLogic, 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 service control for Y. The 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 service control also converts returned messages from web service Y back into Java method invocations on service X.

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

Related Topics

Service Control

Creating and Using a Service Control

A service control makes it easy to access a web service from your application. You create a new service control to access an existing web service (the target web service).

You can create a service control for a target web service if that web service publishes a WSDL file.

If the target web service was developed with Workshop for WebLogic, you can generate a WSDL file by right clicking on the web service .java file and choosing **Web Services > Generate WSDL.**

Creating a Web Service Control from a WSDL File

This procedure describes how to create a service control from the WSDL file for the target web service.

1. Import (or drag and drop) the WSDL file for the web service into a package under the project's **src** folder. The project should be a web service project.
2. Browse to the WSDL file in the **Project Explorer** view.
3. Right-click on the WSDL file name and select **Web Services > Generate Service Control**.

Working with Complex Data Types

When you generate a service control, you will be prompted to create a JAR file that contains the complex data types, if the web service returns data that is not standard Java data types (integer, string, etc.).

If you want to work with the complex types without generating a service control, you can right click on the WSDL and click **Web Services > Generate Types JAR File** to generate a JAR file containing the classes needed to support the WSDL's complex types.

Using a Service Control

You use a service control in a client just as you would use any other control: first you declare the control in the client and then you invoke methods on the control.

You declare the control as follows:

```
@Control
private HelloWorldServiceControl helloWorldCtrl;
```

You invoke methods as follows:

```
helloWorldCtrl.helloWorld();
```

For more information on invoking control methods in a client, see Invoking a Control Method and Handling Control Events.

Related Topics

Overview: Web Service Controls

Handling Web Service Callback Messages

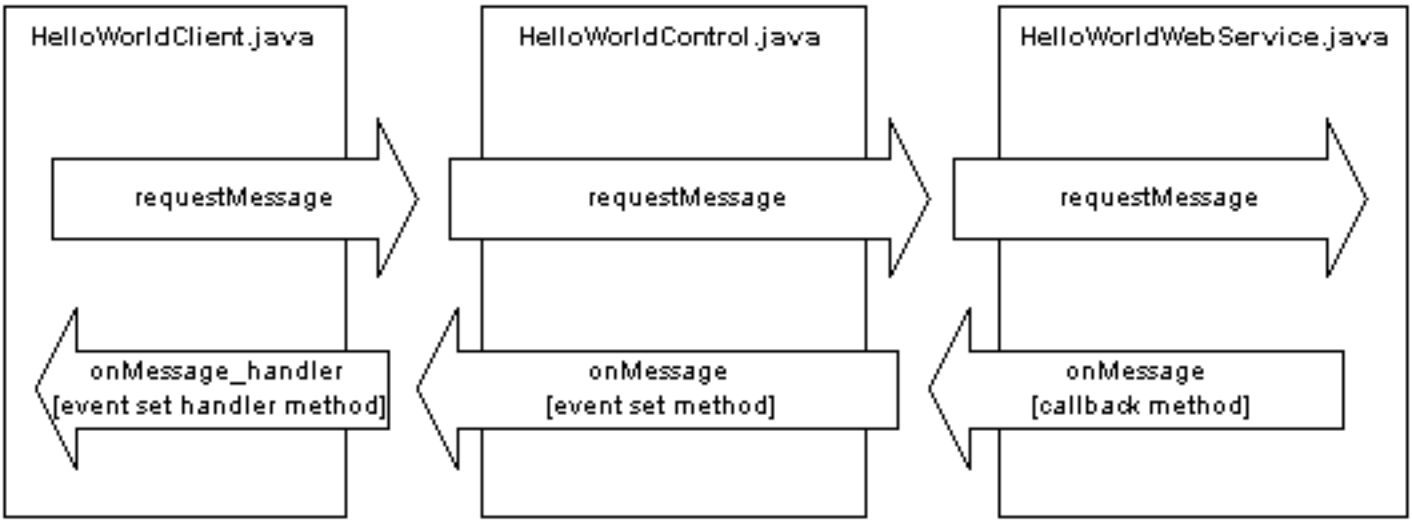
This topic explains how a web service control handles callback messages sent from a web service and passes them on to a control client.

A Web Service Callback Scenario

Suppose you have a web service that sends callback messages back to the client through a service control.

The following diagram shows how the client, service control, and web service are related. The client is a web service named HelloWorldClient.java, the service control is HelloWorldControl.java, and the target web service is HelloWorldService.java.

Arrows pointing the right are ordinary methods. Arrows pointing to the left depict an event handler method (onMessage_handler on the client), an event set method (onMessage on the service control), and a callback method (onMessage on the target web service).



Setting up a Callback Message Handler

Assume that the source code for the callback method onMessage on HelloWorldWebService.java is as follows:

HelloWorldWebService.java

```
@WebService
public class HelloWorldWebService {

    ...

    @CallbackService
    public interface CallbackSvc {
```

```
        @WebMethod
        public void onMessage(String aMessage);
    }
}
```

To handle this callback message in a service control, add an event set interface that includes a method with the same name as the callback method, decorated with the annotation @ServiceControl.ExternalCallbackEvent.

HelloWorldControl.java

```
    @EventSet(unicast=true)
    public interface Callback
    {
        @ServiceControl.ExternalCallbackEvent
        public void onMessage(java.lang.String aMessage_arg);
    }
```

For information on using a service control in a client, see Creating and Using a Service Control.

For information on setting up a callback interface in a web service see Web Service Callbacks.

Automatically Generating a Service Control

You can automatically generate a service control with the appropriate event set methods by right-clicking on the target web service's WSDL and selecting **Web Service > Generate Service Control**. A service control will be generated with event set methods corresponding to all of the callback methods in the target web service.

Related Topics

Creating and Using a Service Control

Web Service Callbacks

EJB Control

The EJB control makes it easy to use an existing, deployed Enterprise JavaBean (EJB) from your application.

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. Workshop for WebLogic also provides tools for developing and deploying EJBs.

The EJB control in Workshop for WebLogic is the standard Beehive EJB control.

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.

Beehive Documentation for EJB Control
: Describes how to work with session and entity beans and how to handle exceptions that might be thrown by an EJB.

Related Topics

Using System 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, custom control 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 support 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, 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 stateless 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 Workshop for WebLogic application, as well as interactions between EJBs and web services, custom controls or page flows in the same Workshop for WebLogic 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 Workshop for WebLogic 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 System 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.

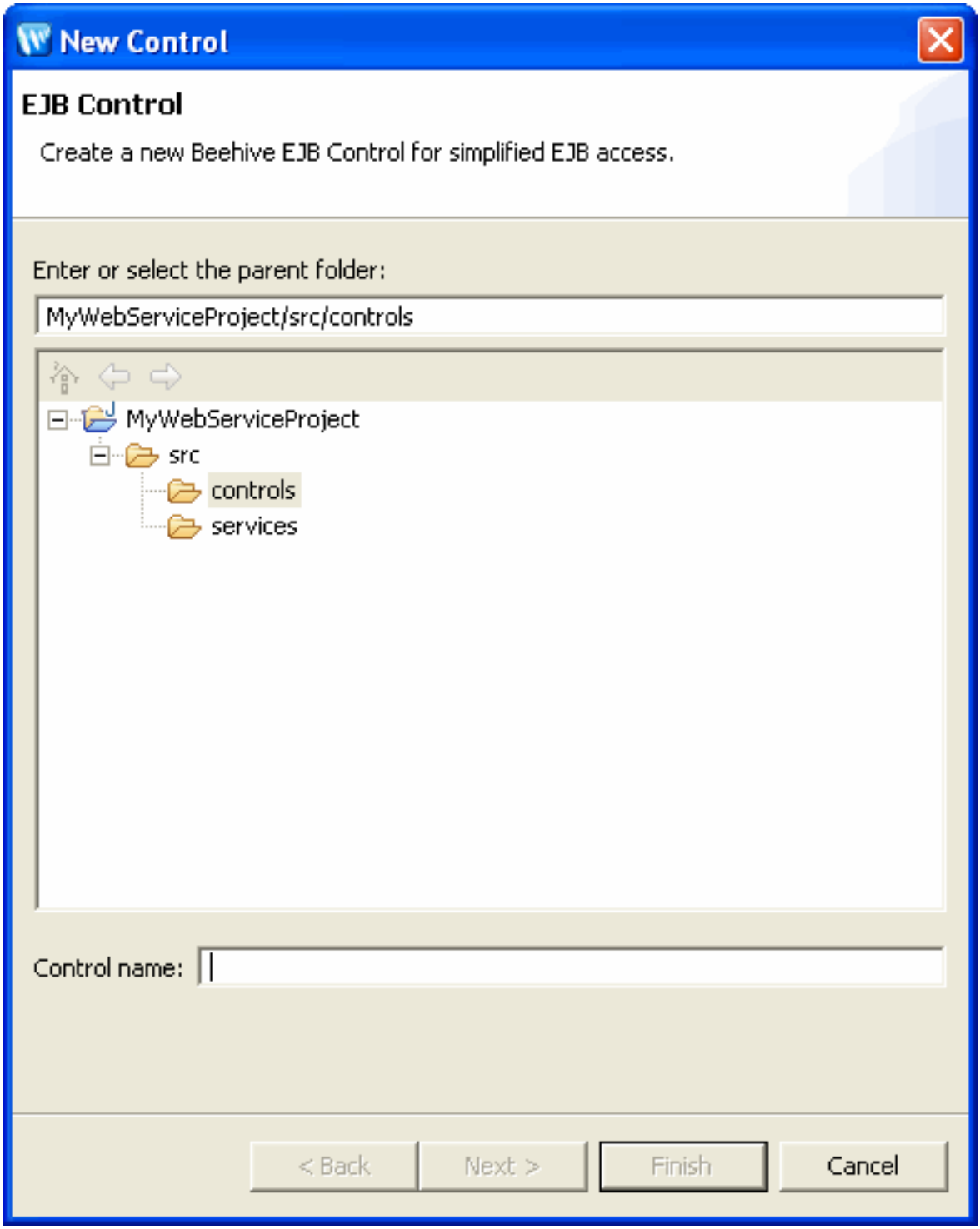
Making EJB Interfaces Available to Your Application

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 Workshop for WebLogic application. While the complete EJB JAR file allows Workshop for WebLogic 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.

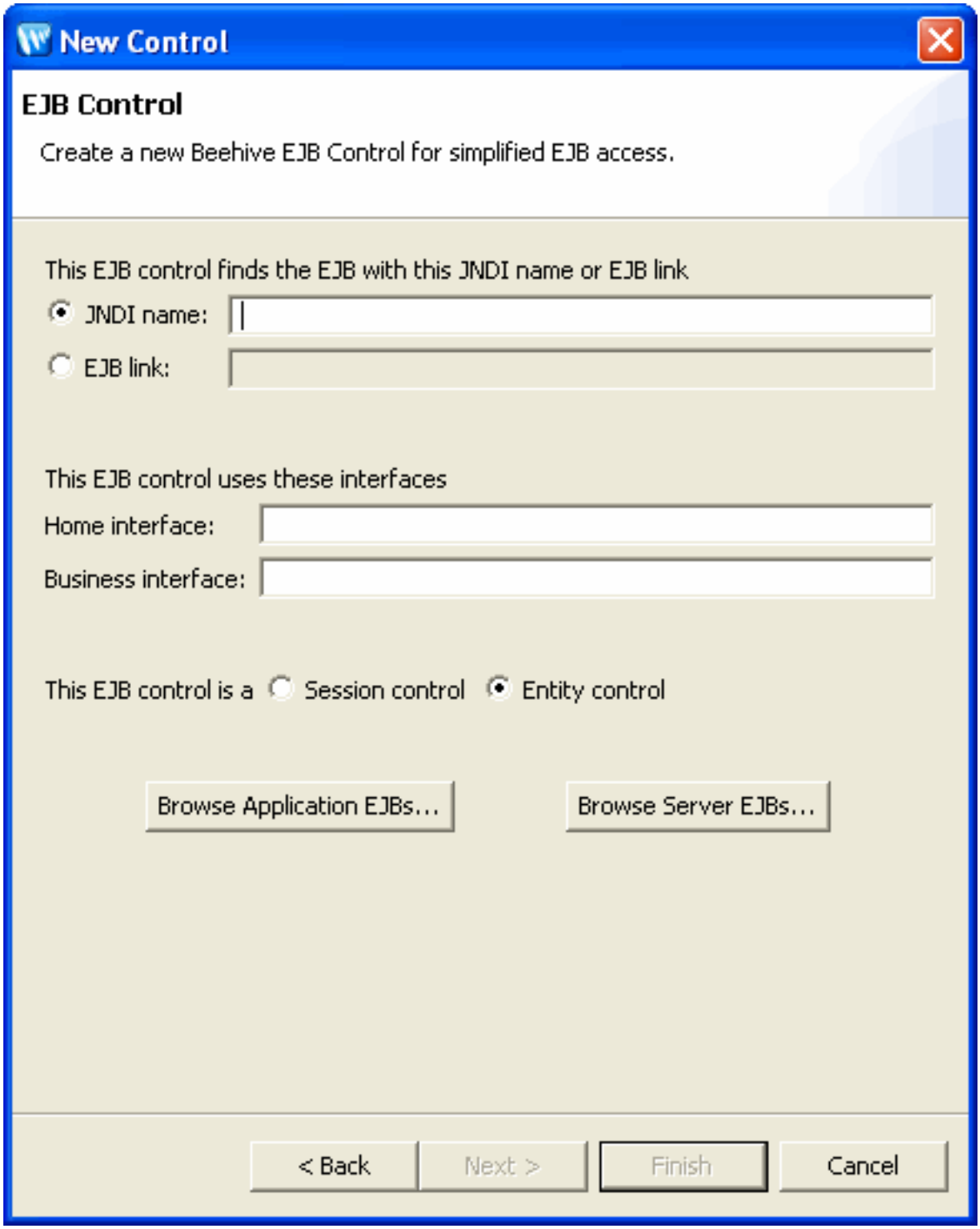
Creating a New EJB Control

To create a new EJB control:

1. Locate or create the package (folder) where you want to create the EJB control. This can be a package in a utility project, dynamic web project, or web service project.
2. Right-click the package and choose **New** > **EJB Control**. The **New Control** dialog opens.
3. In the **Control name** field, enter the name of the new EJB control. Click **Next**.



4. Click the **Browse application EJBs** or **Browse server EJBs** button. The **Browse Resources** dialog appears and you can select the EJB for which this control is being created.



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 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. If an EJB's local interface and local JNDI name are defined, the EJB will also appear with a `local jndi` reference. The same analogy applies to remote interfaces.

If the server is running, 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.

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

6. Click **Finish**.

To insert the EJB control into a web service, web application or another control, see Using an EJB Control.

Modifying EJB Controls

If you open an EJB control in the editor window, 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 see the EJB methods exposed by the EJB control, insert an EJB control in a web service or page flow.

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.

For more information, consult the Beehive documentation for EJB control.

Related Topics

Using System Controls

Using an EJB Control

Tutorial: Enterprise JavaBeans

Using an EJB Control

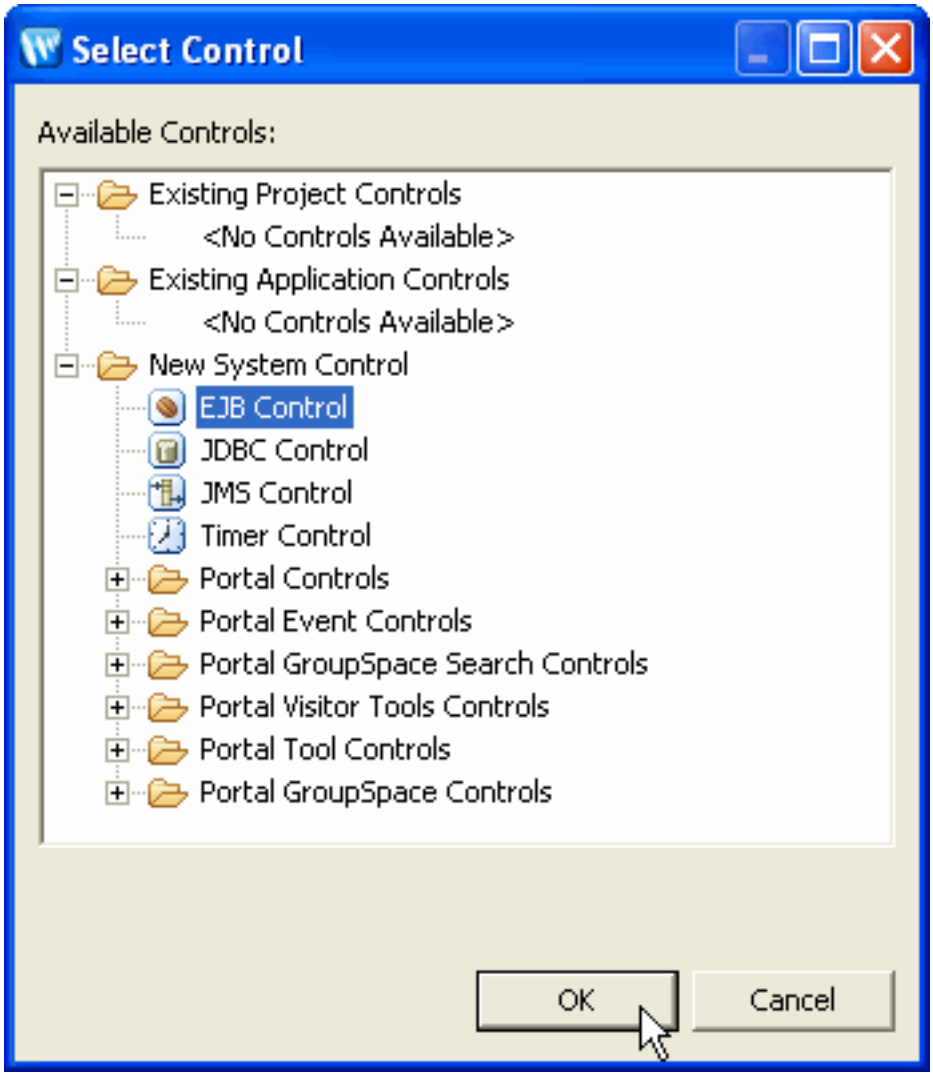
You can add an EJB control to any of the following:

- another control
- a page flow
- a web service

To Insert a New EJB Control

To create a new EJB control and insert it into your source code in a single step:

1. Make sure you have opened the target web service, page flow controller or control in the editor window.
2. Right click on the editor window and choose **Insert > Control**.

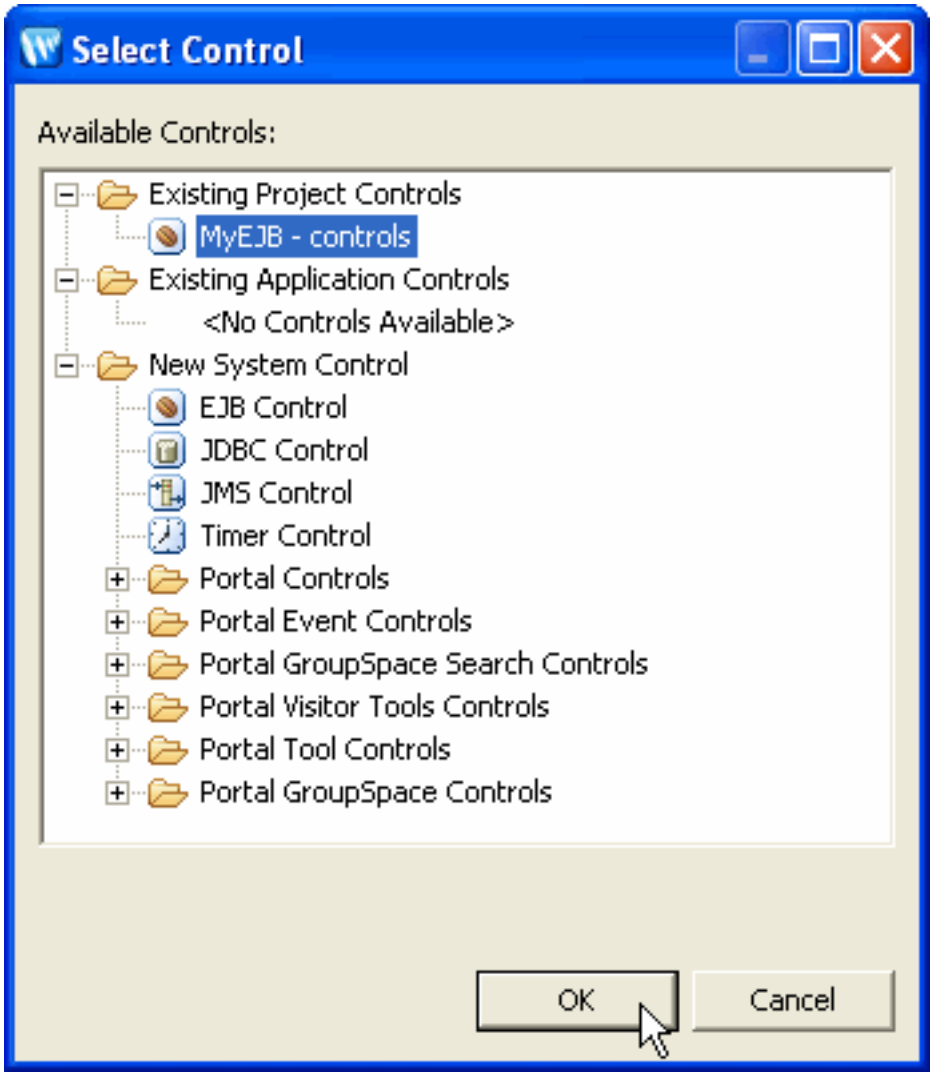


Under **New System Control**, click on **EJB Control** and click **OK**.

3. Follow the instructions in Creating a New EJB Control.

To Insert an Existing EJB Control

1. Make sure you have opened the target web service, page flow controller, control or JSP in the editor window.
2. Right click on the editor window and choose **Insert > Control**.



Click on the name of the control and click **OK**.

Accessing the Methods of an EJB

After you have created an EJB Control, you can invoke a 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. Consult the Beehive documentation for EJB control for more information.

Related Topics

Overview: Enterprise JavaBeans and EJB Controls

Creating a New EJB Control

Getting Started with Session Beans

Getting Started with Entity Beans

JMS Control

The JMS control enables applications built in Workshop for WebLogic to easily interact with messaging systems that provide a JMS implementation, such as WebLogic Server.

A specific JMS control is associated with particular facilities of the messaging system. Once a JMS control is defined, clients may use it like any other Workshop for WebLogic control. To learn how to create, configure and register JMS queues, topics and connection factories, consult the WebLogic Server documentation on Programming WebLogic JMS.

The following changes have been made in the JMS control that was provided in WebLogic Workshop 8.1:

- In WebLogic Workshop 8.1, there was {parm} support in method level annotations. This is no longer supported. You have to annotate parms to specify header or property values.
- New transacted annotation needs to be false when used in a container-managed transaction.

The JMS control in Workshop for WebLogic is the standard Beehive JMS control.

Topics Included in This Section

Creating a New JMS Control
: Describes how to create a JMS control.

Using a JMS Control
: Describes how to use a JMS control in a client.

Beehive Documentation for JMS Control
: Describes how to set message body, header and properties.

Related Topics

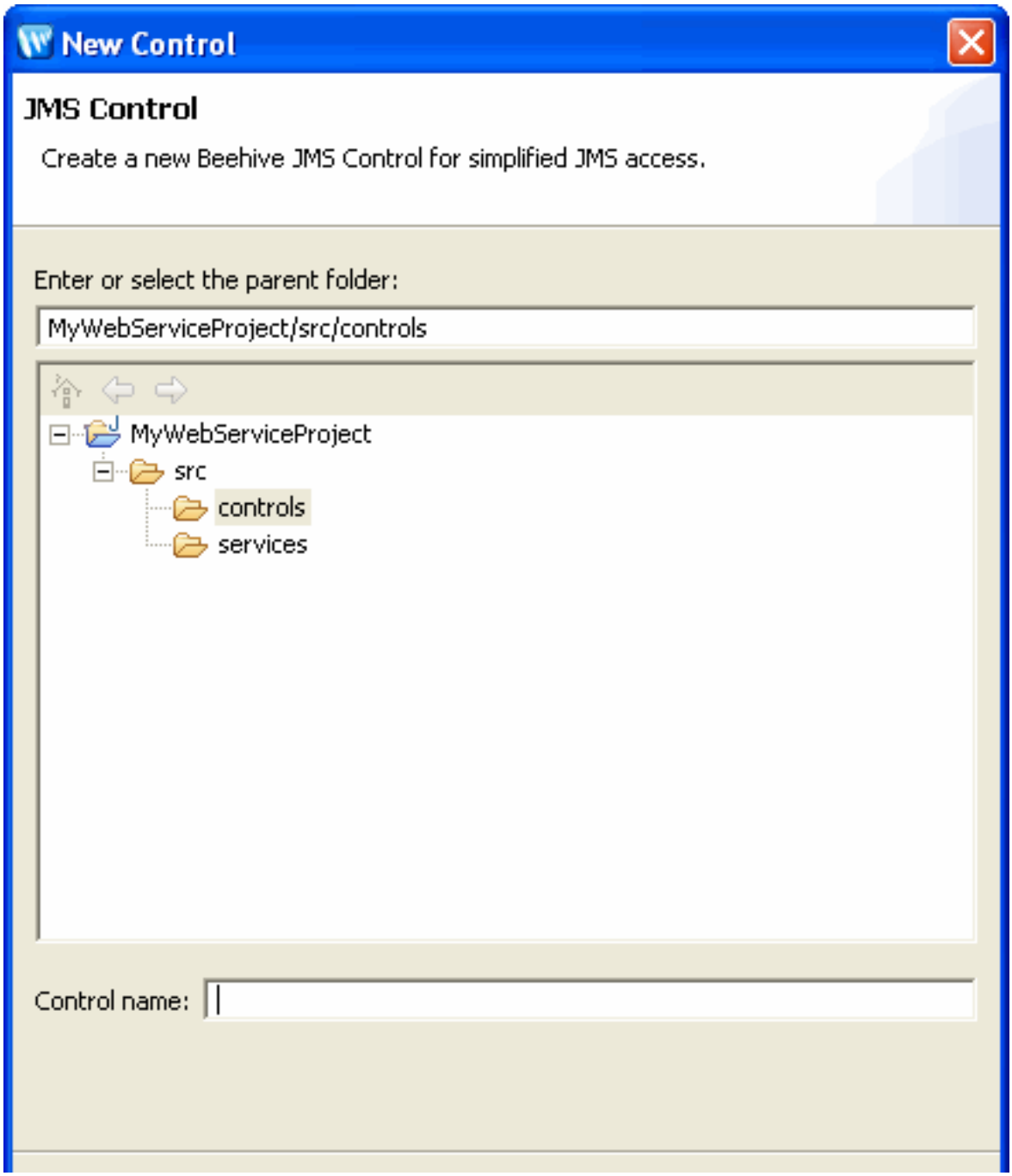
Using System Controls

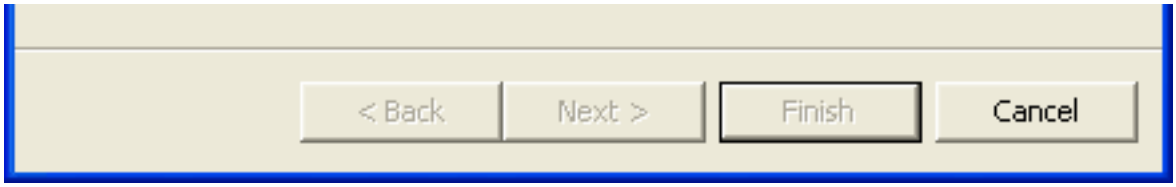
Creating a New JMS Control

The JMS control enables applications built in Workshop for WebLogic 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.

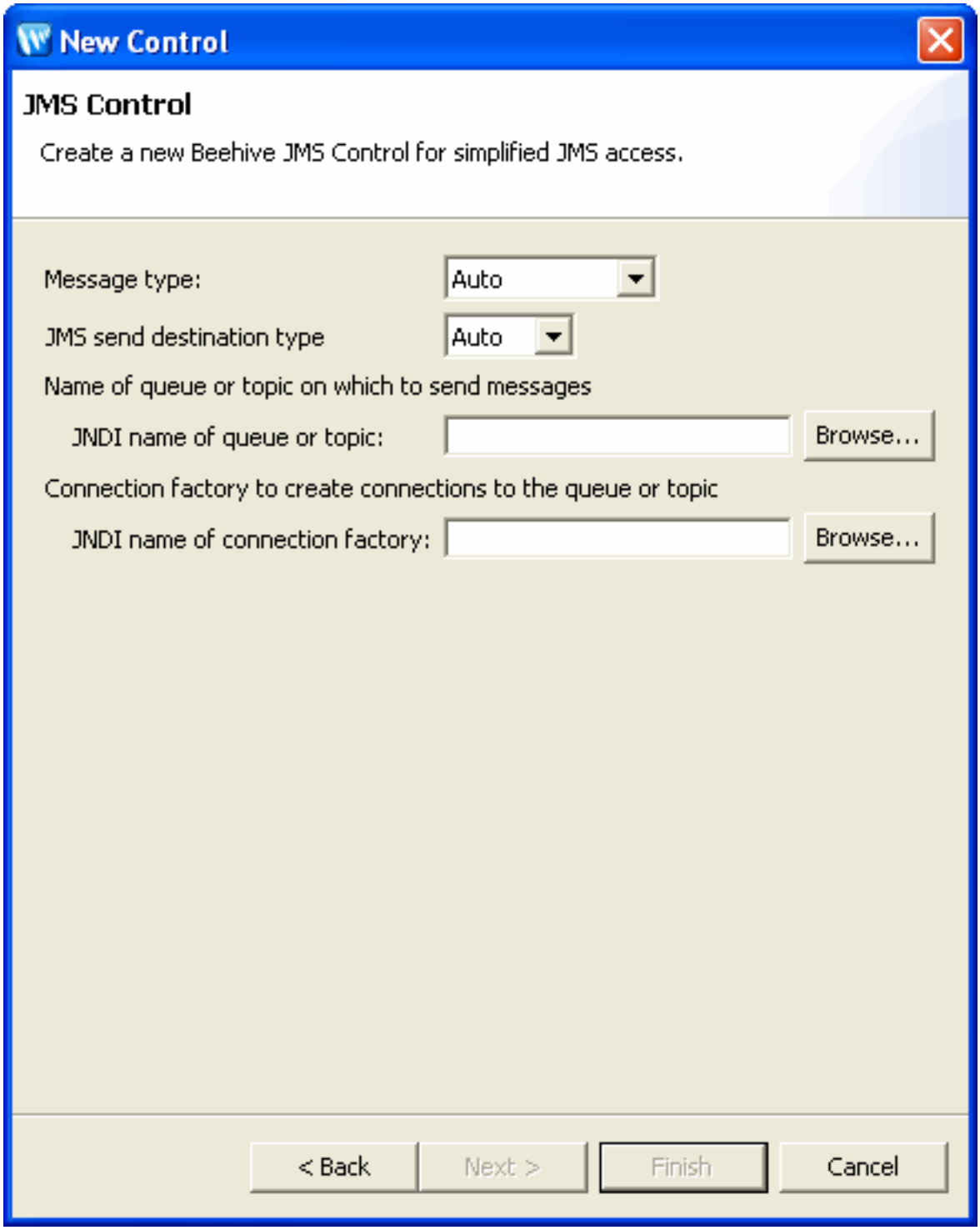
To create a new JMS control:

1. Locate or create the package (folder) where you want to create the JMS control. This can be a package in a utility project, dynamic web project, or web service project.
2. Right-click the package and choose **New** > **JMS Control**. The **New Control** dialog opens.
3. In the **Control name** field, enter the name of the new JMS control. Click **Next**.





4. From the next dialog, choose the JMS queue settings:



In the **JNDI name of queue or topic** field, type the name of the queue or topic that will receive messages. If you do not know the name, click **Browse**, choose from the available list and click **Finish**.

In the **JNDI name of 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**, choose from the available list and click **Finish**.

5. Click **Finish**.

To insert the JMS control into a web service, web application or another control, see Using a JMS Control.

Related Topics

JMS Control

Using System Controls

Using a JMS Control

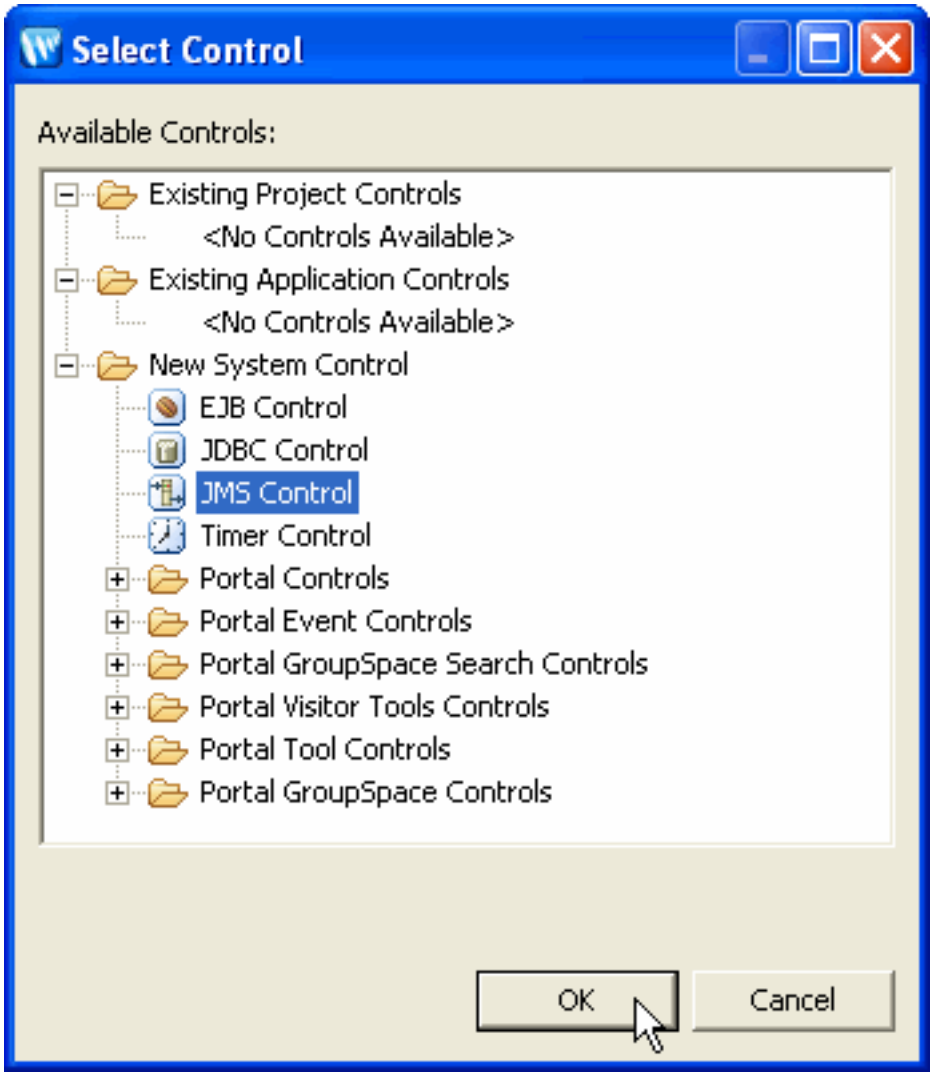
You can add a JMS control to any of the following:

- another control
- a page flow
- a web service

To Insert a New JMS Control

To create a new JMS control and insert it into your source code in a single step:

1. Make sure you have opened the target web service, page flow controller or control in the editor window.
2. Right click on the editor window and choose **Insert > Control**.

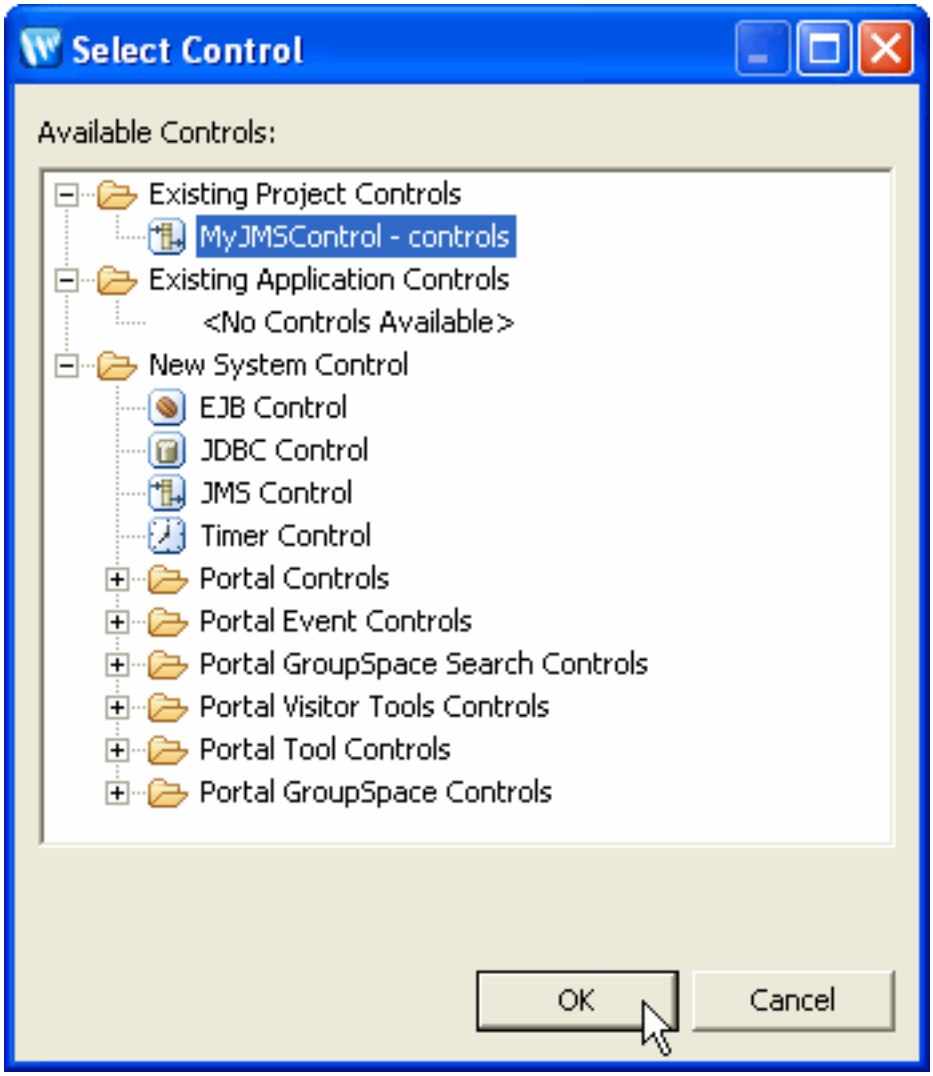


Under **New System Control**, click on **JMS Control** and click **OK**.

3. Follow the instructions in Creating a New JMS Control.

To Insert an Existing JMS Control

1. Make sure you have opened the target web service, page flow controller or control in the editor window.
2. Right click on the editor window and choose **Insert > Control**.



Click on the name of the control and click **OK**.

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

JMS Control Properties

The properties of the JMS control can be set using **Annotations** view from J2EE perspective when the client .java file is open in the editor and the cursor is located in the JMS control name. You specify the messaging style (queue, topic or automatic) and provide the JNDI identification for the

queue or topic. You also specify a connection factory for the control. For detailed information on the `@JMSControl` annotation and its attributes, see the Beehive documentation for the JMS control.

JMS Control Methods

The JMS control has the following default methods:

- getConnection() - returns the javax.jms.Connection

  getDestination() - returns the javax.jms.Destination

  getSession() - gives you programmatic access to the JMS session.
- setHeader(HeaderType, Object) - Sets a JMS header to be assigned to the next JMS message sent.

  setHeaders(Map) - Sets multiple JMS headers to be assigned to the next JMS message sent.

  setProperties(Map) - Sets JMS properties to be assigned to the next JMS message sent.

  setProperty(String, Object) - Sets a JMS property to be assigned to the next JMS message sent.
- 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.

All of the methods you define on the JMS control send or publish to the queue or topic named by the `sendJndiName` attribute of the `@JMSControl` annotation.

Consult the Beehive documentation for JMS control for more information.

Related Topics

JMS Control

Using System Controls

JDBC Control

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

A JDBC control can operate on any database for which an appropriate Java Database Connectivity (JDBC) driver is available or which has a data source configured in WebLogic Server. When you add a new JDBC 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

Tutorial: Accessing a Database from a Web Application
: Provides step-by-step instructions for using a JDBC control in a web application (page flow).

Overview: JDBC Controls
: Introduces the basic concepts behind database controls.

Creating a New JDBC Control
: Explains how to create a new database control.

Using a JDBC Control
: Explains how to use an existing database control.

Adding a Method to a Database Control
: Describes how to write methods on a database control.

Using the Backward-Compatible RowSet feature (WebLogic Workshop 8.1)
: Describes how to work with a RowSet (using an XSD to define metadata) from an application upgraded from WebLogic Workshop 8.1. This feature is deprecated, and should not be used for developing new applications.

Beehive documentation for JDBC control
: Describes how to work with a JDBC control including controlling the data returned, handling exceptions, parameter substitution, stored functions and procedures, and JDBC control return types (including mapping ResultSets to RowSets).

Related Topics

Using System Controls

Overview: JDBC Controls

A JDBC control makes it easy to access a relational database from your Java code using SQL commands.

The methods that you add to a JDBC control execute SQL commands against the database. You can send any SQL command to the database via the JDBC control, so that you can retrieve data, perform operations like inserts and updates, and even make structural changes to the database.

All JDBC controls are subclassed from the `JdbcControl` interface. The interface defines methods that JDBC control instances can call from an application.

Related Topics

None

Creating a New JDBC control

A JDBC control makes it easy to access a relational database via SQL commands. When you create a new JDBC 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 JDBC control.

Choosing a Data Source

Before you can perform operations on a database, you must have a connection to the database. The JDBC 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.

To learn how to create, configure and register a data source, see the documentation provided for WebLogic Server.

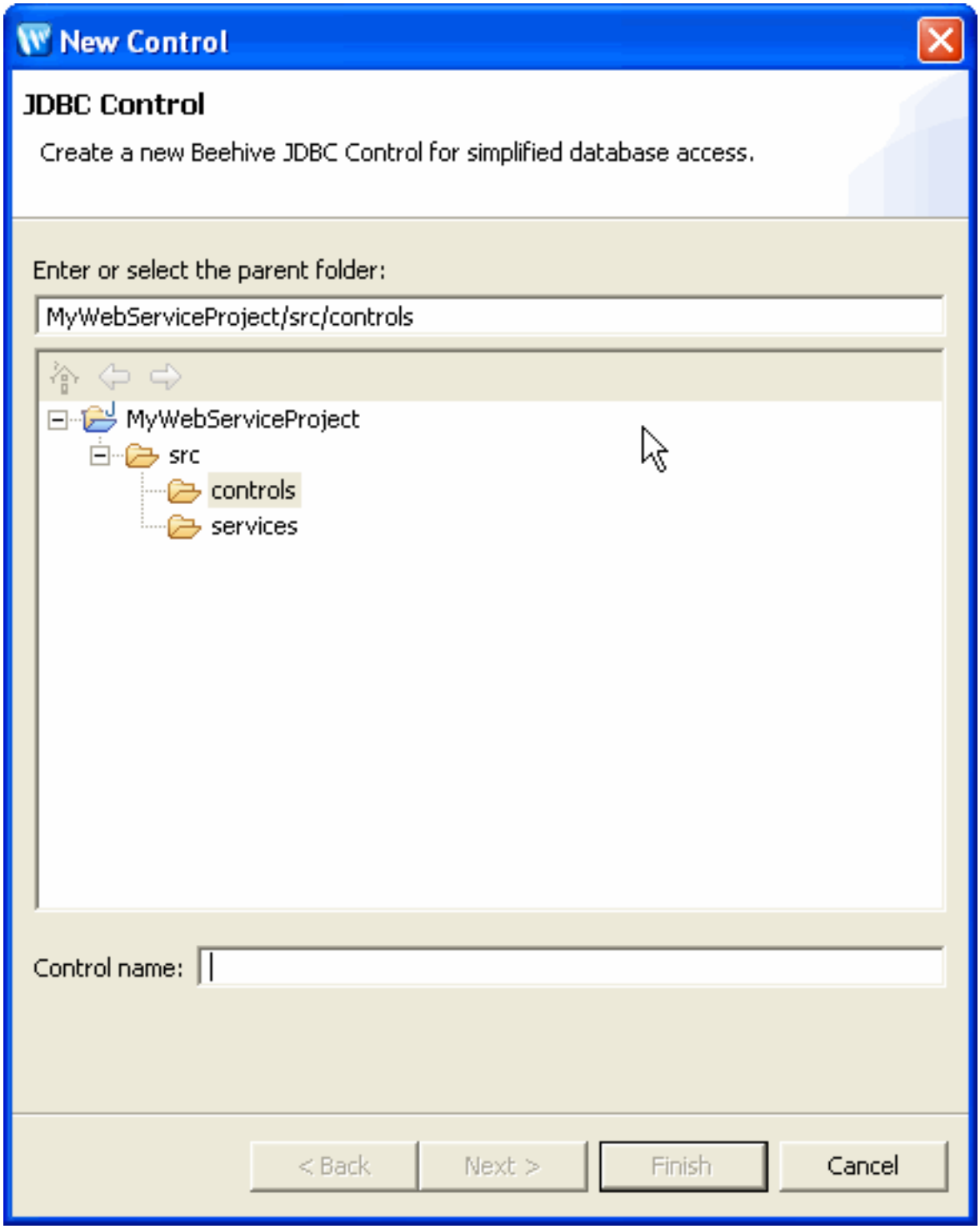
Adding a JDBC control

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

- another control
- a page flow
- a web service

To create a new JDBC control:

1. Locate or create the package (folder) where you want to create the JDBC control. This can be a package in a utility project, dynamic web project, or web service project.
2. Right-click the package and choose **New** > **JDBC Control**. The **New Control** dialog opens.
3. In the **Control name** field, enter the name of the new JDBC control. Click **Next**.



4. Click the **Browse** button to select the data source.

   The **JNDI Entries** dialog appears. Navigate to the data source you want to select and click **Select**.

5. Click **Finish**.

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

Related Topics

System Controls Overview

JDBC control

Using a JDBC Control

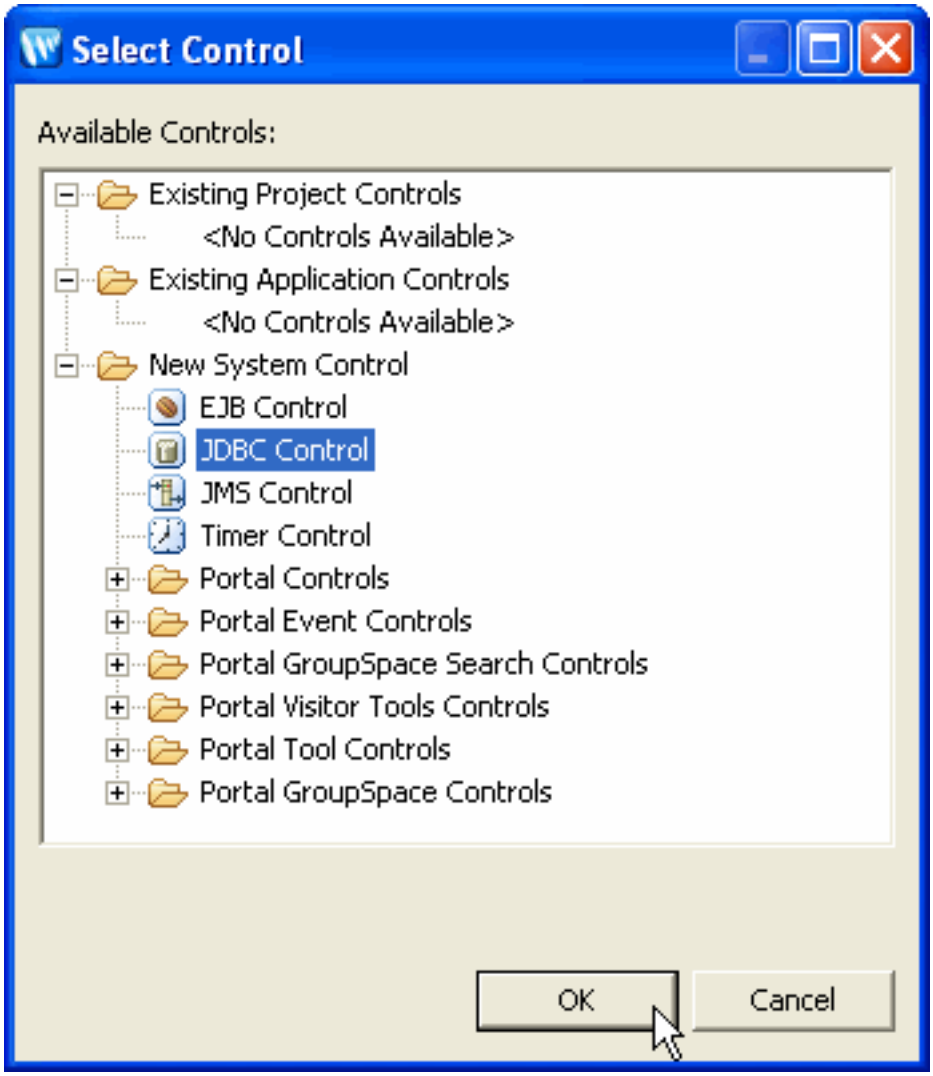
You can add a JDBC control to any of the following:

- another control
- a page flow
- a web service

To Insert a New JDBC Control

To create a new JDBC control and insert it into your source code in a single step:

1. Make sure you have opened the target web service, page flow controller or control in the editor window.
2. Right click on the editor window and choose **Insert > Control**.

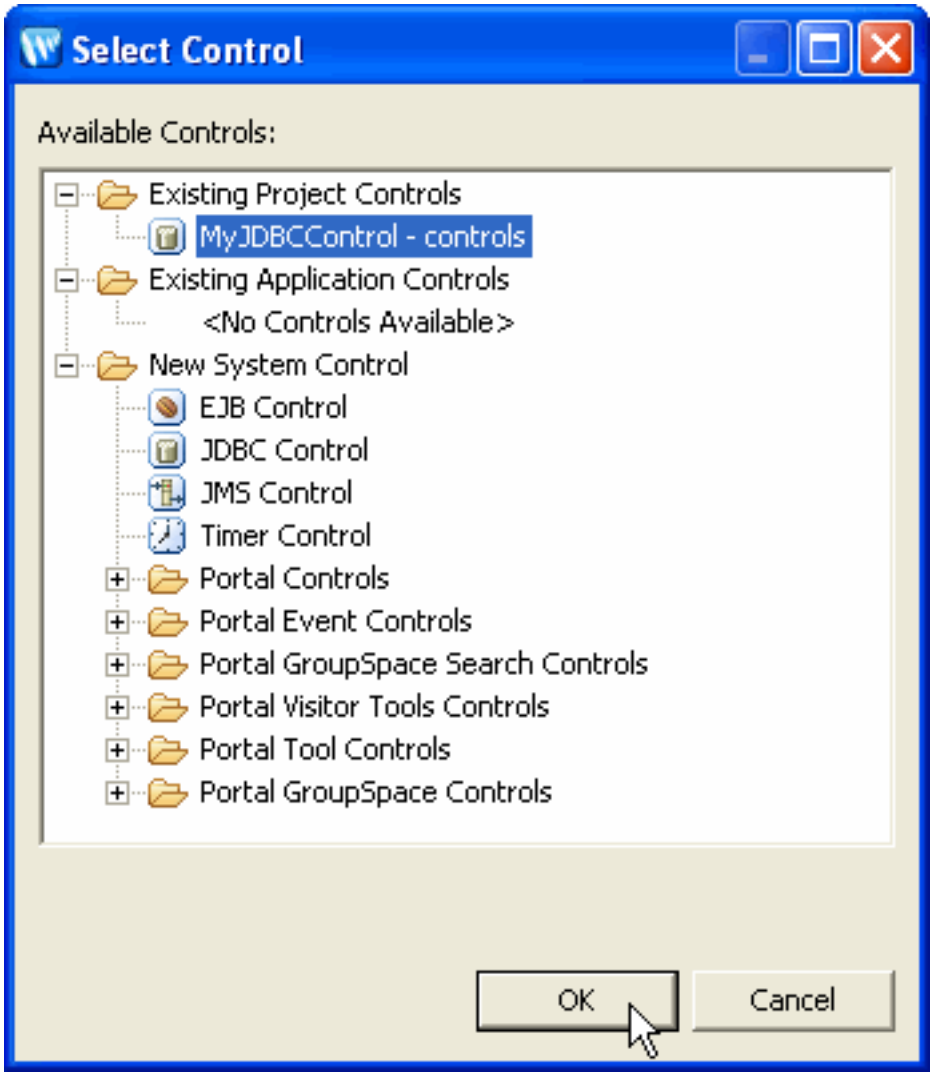


Under **New System Control**, click on **JDBC Control** and click **OK**.

3. Follow the instructions in Creating a New JDBC Control.

To Insert an Existing JDBC Control

1. Make sure you have opened the target web service, page flow controller or control in the editor window.
2. Right click on the editor window and choose **Insert > Control**.



Click on the name of the control and click **OK**.

Accessing the Methods of a JDBC Control

After you have created a JDBC Control, you can invoke the methods of the control in the same way as accessing the methods of a regular object. Consult the Beehive documentation for JDBC control for more information.

Related Topics

Using System Controls

Creating a JDBC Control

Adding a Method to a JDBC control

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

Specifying the SQL Statement

A method on a JDBC control always has an associated SQL statement, which executes against the database when the method is called. The method's `@JdbcControl.SQL` annotation describes the method's SQL statement.

A sample method is provided in the comment header when you create a JDBC control:

```
@JdbcControl.SQL(statement="SELECT ID, NAME FROM CUSTOMERS WHERE ID = {id}")

Customer findCustomer(int id) throws SQLException;
```

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 Workshop for WebLogic knows which parameter to replace with which value. Within the SQL statement, substitution parameters are enclosed in curly braces.

In the example above, the SQL statement includes the substitution`{id}`. This maps to the `id` parameter 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 JDBC control method SQL statements are described in the Beehive documentation for JDBC control.

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

A method of a JDBC control can return a single value, a single row, or multiple rows. To learn more about the values returned by JDBC control methods, see the Beehive documentation for JDBC control.

Related Topics

Using System Controls

JDBC control

Creating a New JDBC control

Using the Backward-Compatible RowSet feature (WebLogic Workshop 8.1)

This deprecated feature should not be used for new development.

When you upgrade a WebLogic Workshop 8.1 application, the upgrade wizard converts the RowSet control to use a backward-compatible JDBC control that includes support for RowSets. The RowSet control is documented in WebLogic Workshop 8.1 documentation at:

- RowSet Control
- RowSet Controls and SQL Join Queries

The backward-compatible JDBC control is documented at:

- Backward-Compatible JDBC control that supports RowSet feature
- RowSet Annotation

Developing Custom Controls

BEA Workshop for WebLogic Platform allows you to create custom controls tailored to your project or application. Custom controls can be used to create re-usable controls that might be found in a company for sharing, or those provided by ISVs for their products. This section explains how to create these controls and how to share them.

For a complete overview of controls in Workshop for WebLogic, including how to create them, see Getting Started with Beehive Controls.

Topics Included in This Section

Creating Custom Controls
: Describes the basics of creating and using custom controls.

Source Files for Custom Controls
: Describes the files that are necessary in any custom control.

Testing Controls
: Discusses how to test custom controls.

Exporting Controls into JARs
: Describes how to export controls into a JAR file that can be shared.

Distributing Controls as Plug-Ins
: Shows you how to customize controls more extensively and how to package/distribute controls for a wider audience.

Related Topics

Using System Controls

Creating Custom Controls

This topic describes how to use a custom custom control. It explains how to:

- Create a custom control
- Use a custom control in your application

Custom control files can be located:

- In your web project.
- In a utility project. To access such controls in a web application, both the web project and the utility project must be linked to the same EAR project.

To Create a Custom Control

The following instruction assume you are in the J2EE perspective (**Window > Open Perspective > J2EE**).

1. You cannot create a control in the default package. So the first step is to create a package for the control. For example:

   <ProjectRoot>/src/controls/myControl/

2. Right-click the package and select **New** > **Custom Control**.
3. In the **Control name** field, enter the class name for the control.

   The Java interface and implementation classes will be based on the name entered here. For example, if you enter Hello, two classes will be created:

   Hello.java (=the interface class)

   and

   HelloImpl.java (=the implementation class)

4. Click **Finish**.

Default control interface and implementation classes are produced. Assuming that your control is named Hello, the following class files are produced:

Hello.java Interface Class File

```
package controls.myControl;
```

```
import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
public interface Hello {

}
```

HelloImpl.java Implementation Class File

```
package controls.myControl;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import java.io.Serializable;

@ControlImplementation
public class HelloImpl implements Hello, Serializable {
        private static final long serialVersionUID = 1L;

}
```

Continue the composition of the custom control by adding methods to these class files.

To Use a Custom Control in an Application

If you have an existing custom control in your project or in a utility project in the current workspace, you can add a reference to that control to a control client by right-clicking anywhere within the client's Java source file and selecting **Insert > Control**.

A list of available controls appears. The heading **Existing Project Controls** lists the controls in the same project as the client. The heading **Existing Application Controls** lists the controls in the utility projects in the same workspace.

When you add a control reference to a client, Workshop for WebLogic Platform modifies your client's source code to include an annotation and variable declaration for the control. The annotation ensures that the control is recognized by Workshop for WebLogic Platform, and the variable declaration gives you a way to work with the control from your client code. For example, if you add a new custom control named `Hello`, the following code will be added to your client:

```
    import org.apache.beehive.controls.api.bean.Control;
    import controls.myControl.Hello;

        @Control
        private Hello hello;
```

Once you have a reference to a control, your client can call methods on that control. For more detail on calling a control method, see Invoking a Control Method.

Related Topics

Invoking a Control Method

Source Files for Custom Controls

Source Files for Custom Controls

Custom controls consist of two Java source files: an **interface** class file and an **implementation** class file.

The interface class contains the control's publicly accessible methods. Clients of the control call the methods in the implementation class.

The implementation class contains the control's behind the scenes implementation code.

There is also a third class associated with each custom control: the **generated JavaBean class**. This is a build artifact created from the interface and implementation source files. The generated JavaBean class provides supplemental programmatic access to the control, especially the ability to override default annotation values in the control. For more information about this class see Overriding Control Annotation Values Through the Control JavaBean

Custom Control Interface Classes

A custom control interface class must be decorated with the `@ControlInterface` annotation.

```
package controls.hello;

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

@ControlInterface
public interface Hello {

    ...

}
```

The `@ControlInterface` annotation informs the compiler to treat this class as a part of the Beehive Control framework.

The interface class also lists the control's publicly available methods. The following example shows a control with one publicly available method.

```
package controls.hello;

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

@ControlInterface
public interface Hello {

        public String hello();

}
```

Custom Control Implementation Classes

A custom control implementation class 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.

The minimum requirements for a custom control implementation class are listed below.

1. The class must be decorated with the `@ControlImplementation` annotation.

```
import org.apache.beehive.controls.api.bean.ControlImplementation;

@ControlImplementation
public class HelloImpl
```

2. The class must implement the corresponding custom control interface file.

```
import org.apache.beehive.controls.api.bean.ControlImplementation;

@ControlImplementation
public class HelloImpl implements Hello
```

3. The classes must either:

   (a) implement java.io.Serializable

```
import java.io.Serializable;

@ControlImplementation
public class HelloImpl implements Hello, Serializable
```

   (b) or set @ControlImplementation(isTransient=true)

```
@ControlImplementation(isTransient=true)
public class HelloImpl implements Hello {

}
```

Related Topics

Controls: Getting Started

Testing Controls

Beehive controls can be tested either inside of an application container or outside in a standalone Java environment. Testing in a standalone Java environment is especially useful when running unit tests.

Beehive controls can be integrated into the JUnit test framework using the ControlTestCase base class. This base class provides a control container and provides help in instantiating a control declaratively via the
@Control annotation.

Note that not all controls can be tested within the test container because some controls have requirements beyond what ControlTestCase provides. For example, a control that uses JNDI lookups will not be testable with ControlTestCase. Likewise controls (such as the Service Control) that take a dependency on a J2EE container (such as WebLogic Server) may not be testable out of that J2EE container.

For details on testing controls with ControlTestCase see Control Tutorial: Testing Controls with JUnit.

Related Topics

Testing Controls with JUnit

Exporting Controls into JARs

Workshop for Weblogic Platform lets you package your control classes as JAR files that can be reused in other Java projects. This is the simplest way to distribute controls.

This approach is somewhat limited, providing no custom labels, no custom icons, no insertion wizards. If you are creating controls that will have very wide distribution (e.g., an ISV developing controls for customers), you may want to package your custom control as a plug-in.

To package a Beehive control as a JAR file, select **File > Export > Beehive Control JAR File**.

Only control files in utility projects are available for JAR file packaging; controls in other project types are not available for export.

All Java class files in the utility project are included in the JAR file, including control interface, control implementation classes, and all other Java classes. Note that by default, **only** class files are included in the JAR file. To include the Java source files, place a checkmark next to **Include Java source files**.

To use a control in another web application:

1. Copy the JAR file to the WEB-INF/lib folder.
2. Add a reference to that control to a control client by right-clicking anywhere within the client's Java source file and selecting **Insert > Control**.
3. A list of available controls appears. The heading **Existing Project Controls** lists the available controls, including controls in JAR files.

Alternately, you can:

1. Copy the JAR file to the APP-INF/lib folder of the associated EAR project.
2. Add a reference to that control to a control client by right-clicking anywhere within the client's Java source file and selecting **Insert > Control**.
3. A list of available controls appears. The heading **Existing Application Controls** lists the available controls, including controls in JAR files.

As long as the JAR is inserted into the user's classpath as described above, the control will be discovered automatically by Workshop for WebLogic and property set/event handler features will be provided.

Related Topics

Apache Beehive Documentation

Building Controls

Distributing Controls as Plug-ins

If you want to distribute your custom control to a wide audience (e.g., if you are an ISV developing controls for your customers) or if you want to customize your control more extensively, you can package a control within a plug-in. This method allows:

- Customized label and icon
- Customized insertion wizard

This topic describes how to package a control into a plug-in. This method creates an Eclipse plug-in and basic knowledge of Eclipse plug-ins and their creation would be useful before attempting this process.

Note that this method wraps a control JAR in a plug-in. For distribution within your own company, you may simply want to share the control JAR file directly, without the additional work of creating a plug-in.

This method consists of the following steps:

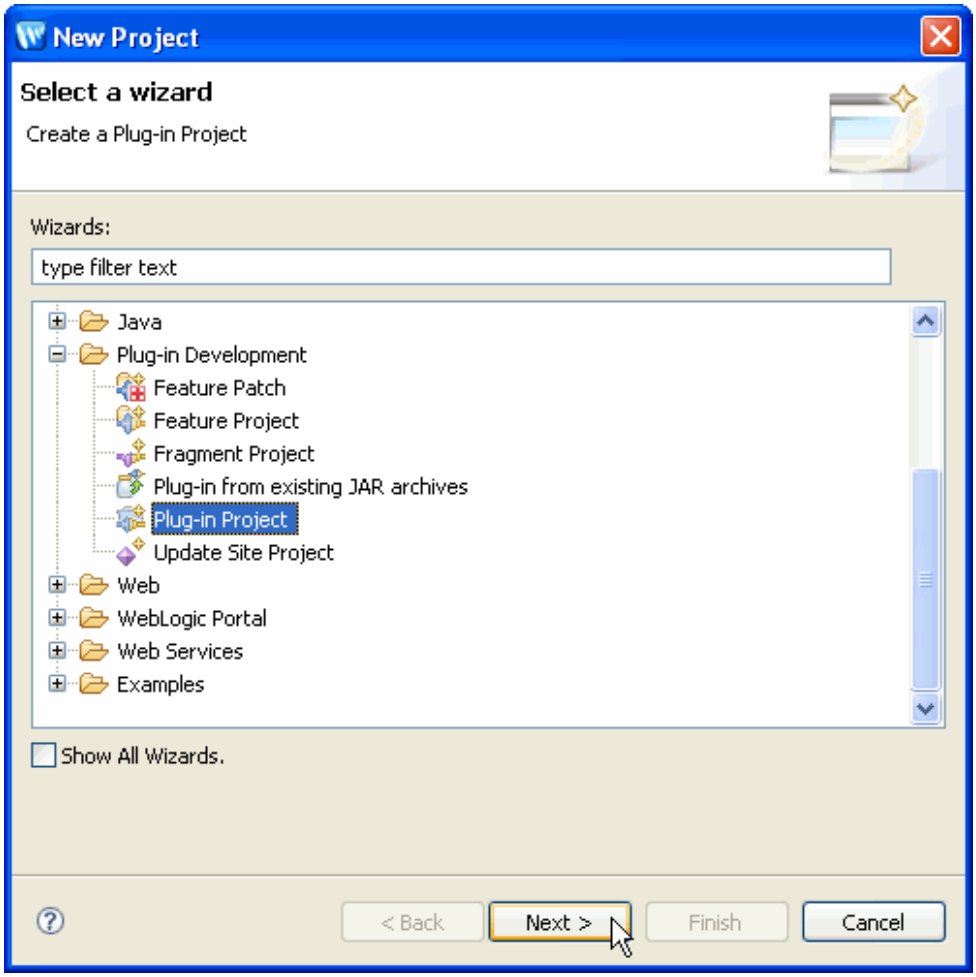
1. Export the control into a control JAR
2. Create the control plug-in project
3. Copy the control JAR into the plug-in project
4. Set plug-in project dependencies
5. Add extension and customize settings
6. Create the insertion delegate code
7. Build and test your plug-in
8. Export the plug-In

Step 1: Export the Control into a Control JAR

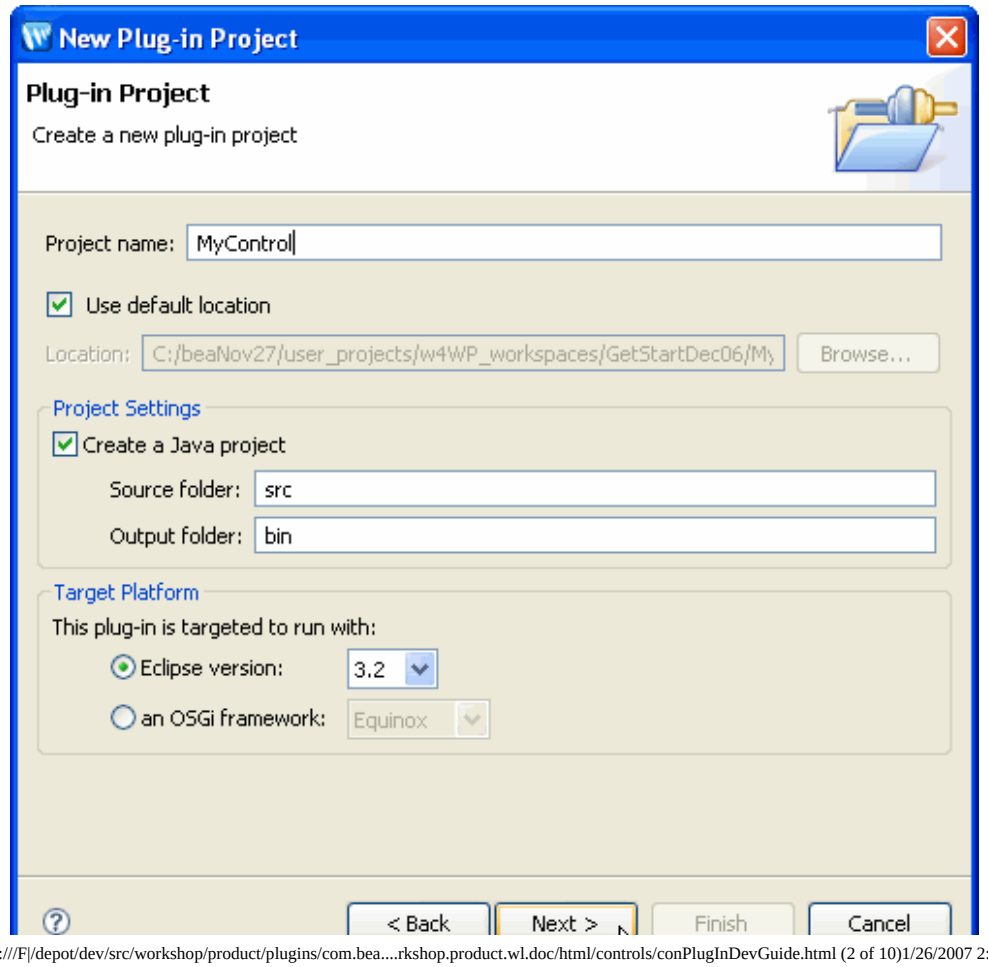
Follow the steps outlined in Exporting Controls into JARs.

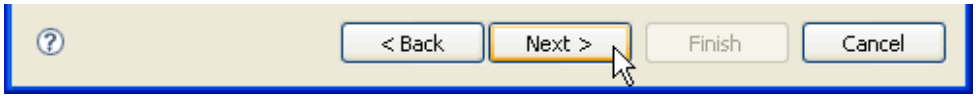
Step 2: Create the Control Plug-in Project

1. Create a plug-in project with **File > New > Project**. Expand **Plug-in Development** and choose **Plug-in Project**. If you do not see the correct project type, you may need to click **Show All Wizards** to display it.

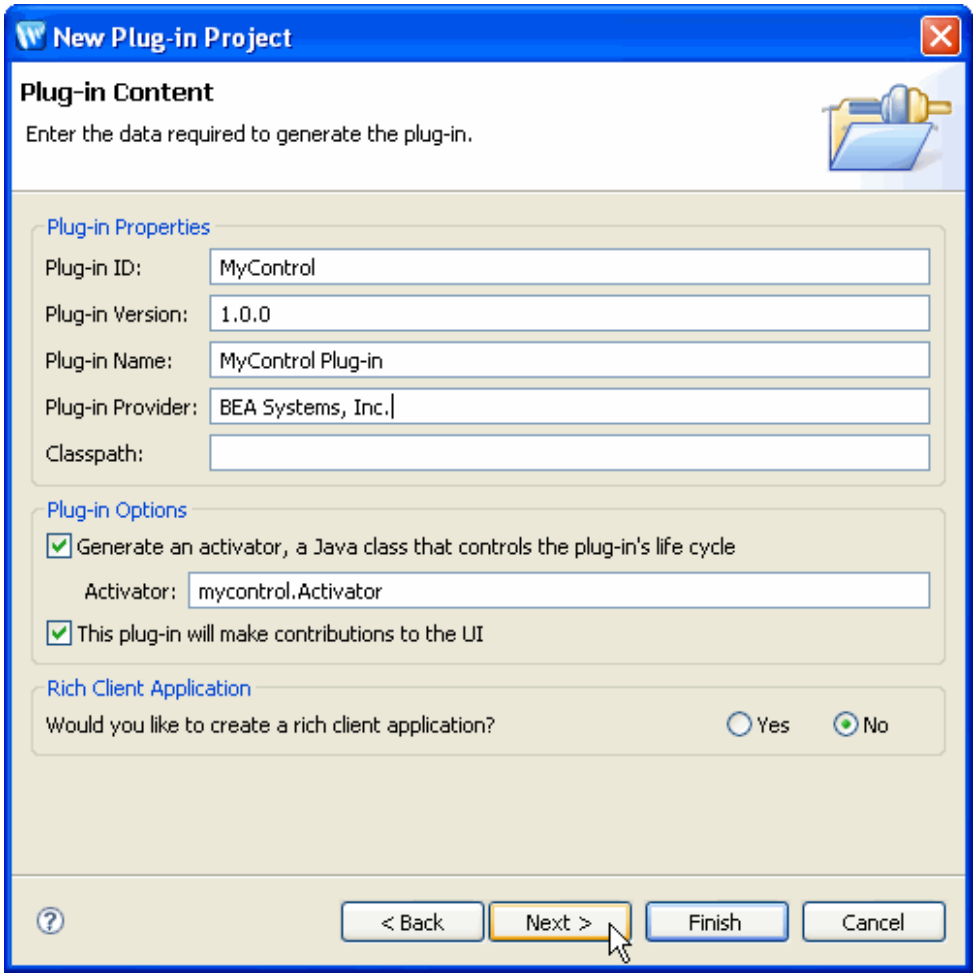


Click **Next** to proceed.





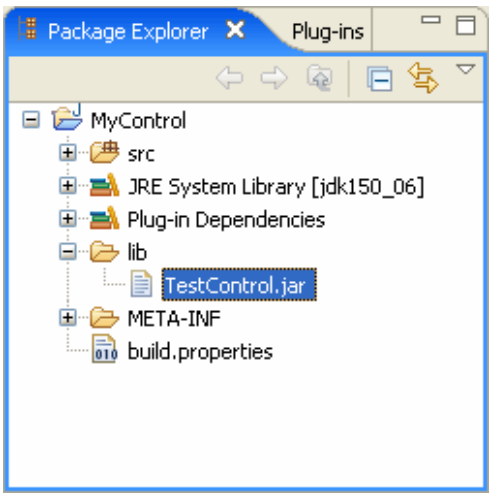
From the next screen, fill in the **Plug-in Provider** field and click **Finish** to create the project.



Click **Yes** to change to Plug-in Development perspective.

Step 3: Copy the Control JAR into the Plug-In Project

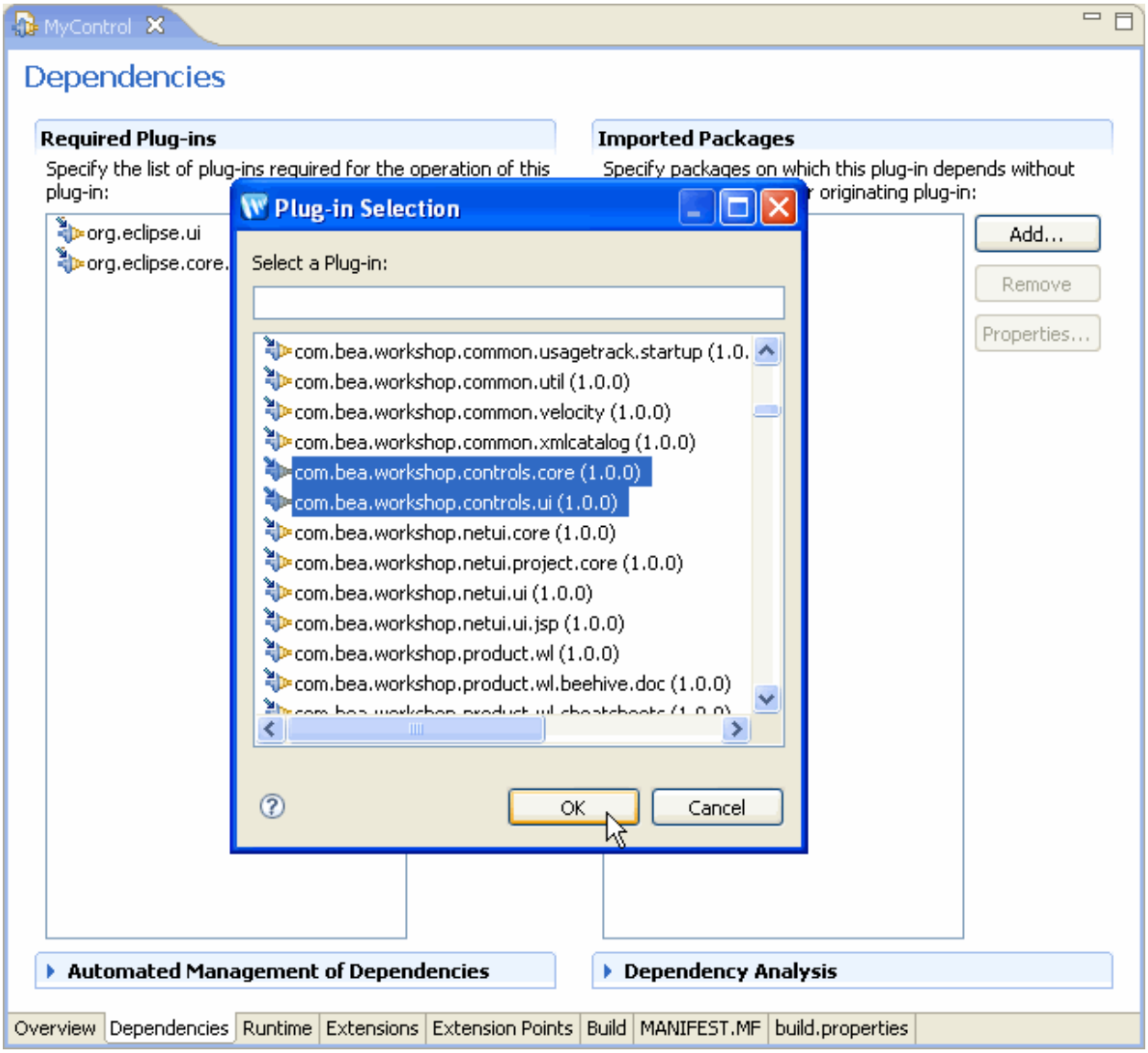
Create a folder named **lib** in the root of your plug-in project (**not** under the **src** folder). Copy the control JAR (created in the previous step) into the **lib** directory.



Step 4: Set Plug-in Project Dependencies

If the manifest editor window is not visible, double click on the MANIFEST.MF file to open it. From the editor, click on the **Dependencies** tab or click on the **Dependencies** link in the **Plug-in Content** section. From the **Required Plug-ins** section, click on the **Add** button and select the following plug-ins:

- com.bea.workshop.controls.core
- com.bea.workshop.controls.ui
- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.eclipse.jdt.core
- org.eclipse.ui

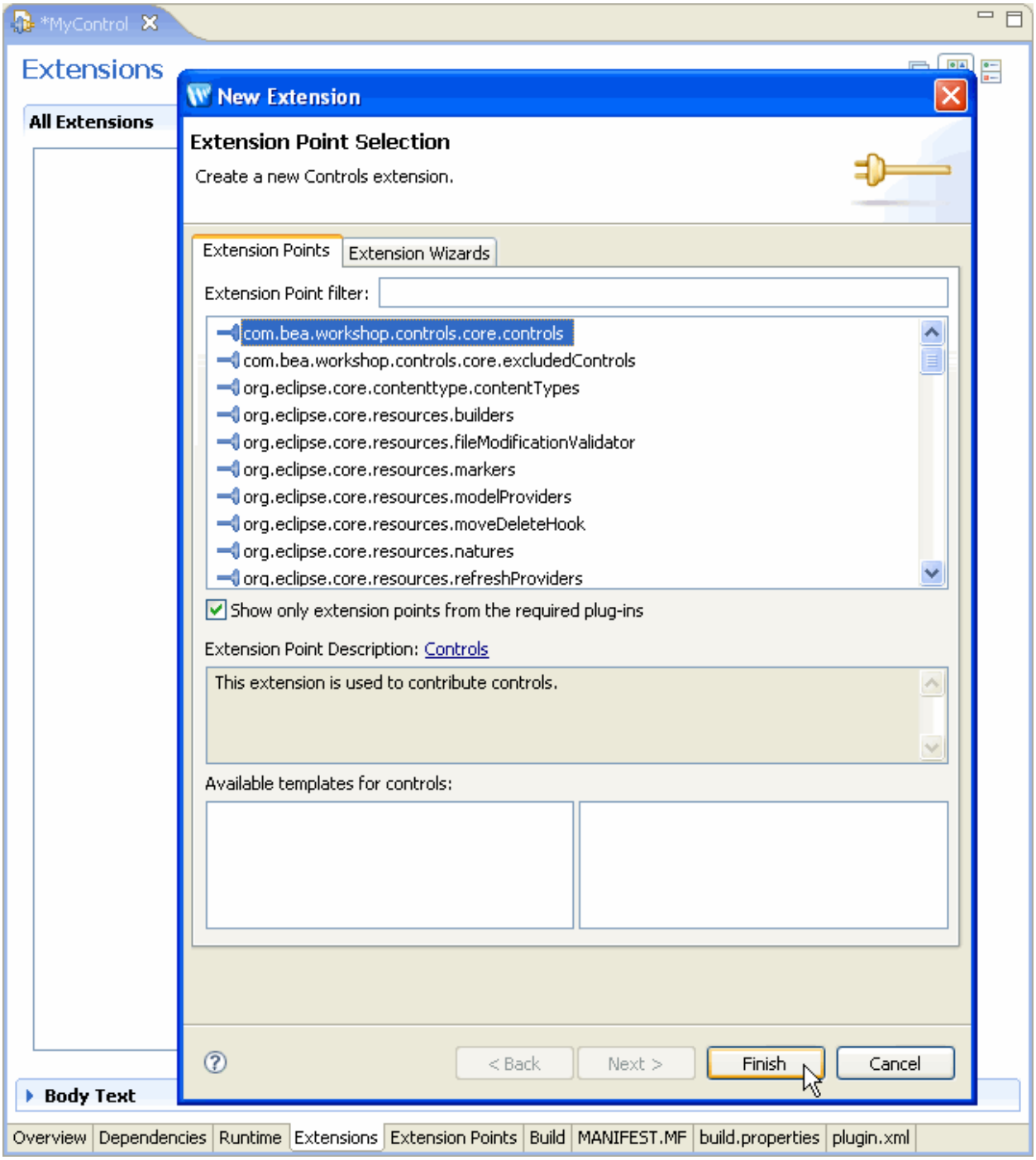


Click **OK** to add the dependencies. Click **File > Save All** to save the dependencies in the MANIFEST.MF file.

Step 5: Add Extension and Customize Settings

Click on the **Extensions** tab. Click **Add** and choose

- com.bea.workshop.controls.core.controls



Click **Finish** to add the extension. Click **File > Save All** to save the change.

Note that a new file: **plugin.xml** has been added to the project. That file now contains the extension information:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="com.bea.workshop.controls.core.controls">
   </extension>
</plugin>
```

The com.bea.workshop.controls.core.controls extension point requires a nested <control> tag with at least the **id, class, isControlExtension,** and **isExtensible** attributes specified. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
  <extension point="com.bea.workshop.controls.Controls">
```

```
    <control
        id="com.mycompany.example.MyExampleControlId"
        class="com.mycompany.example.control.MyExampleControl"
        isControlExtension="false"
        isExtensible="false"  />
  </extension>
</plugin>
```

The <control> tag has the following attributes:

Attribute	Description	Required	Default
id	A unique id string. Cannot be duplcated within the contributed controls in this plug-in.	Yes	[none]
class	Fully qualified classname of the control *interface* class.	Yes	[none]
isControlExtension		Yes	Indicates whether this is an extension of a Beehive extensible control. (See the Beehive control documentation for more information on extensible controls.)
isExtensible		Yes	Indicates whether this is an extensible control. Indicating true will allow the default insertion to better handle requiring the user to create a control extension rather than a regular control. (See the Beehive control documentation for more information on extensible controls.)
label	The text to be displayed on the **Insert > Control** dialog.	No	Simple, unqualified classname from the class attribute.
icon	The icon displayed to the left of the control label on the **Insert > Control** dialog.	No	generic icon
priority	Position relative to others in the same group of controls, ascending order. This is a path relative to the plugin root	No	10
groupName	Group heading for the control(s). Note that if there are less than 3 controls, no group will be created. If there are 3 or more controls, a group will be created if groupName is specified.	No	Value of the label attribute. Note that if controls are not in a group, or if there are not 3 controls in a group, they will all be listed at the top level and the label attribute will be ignored.
groupPriority	Ordering of the group relative to other groups, ascending order.	No	100
insertionDelegateClass	Class triggered when the control is inserted into an application. In addition to any desired actions, the insertion delegate must to copy the control JAR from the plug-in JAR to the user's project.	No	com.bea.workshop.controls.core.DefaultControlInsertionDelegate
description	Description of control.	No	[none]

The following is an example of a <control> tag using more attributes:

```
<extension point="com.bea.workshop.controls.core.controls">
      <control

            class="com.mycompany.controls.MyControl"
            id="MyControl12"
            groupName="My Company"
            groupPriority="10"
            includeInPalette="true"
            insertionDelegateClass="com.mycompany.workshop.MyInsertionDelegate"
            isControlExtension="false"
            isExtensible="false"
            label="Sample Control"
            palettePriority="10"
            priority="10"
      />
</extension>
```

Step 6: Create the Insertion Delegate Code

The **insertionDelegateClass** attribute of the <control> tag indicates the insertion delegate and triggers the delegate when the control is inserted into a file. You can use this for many purposes, but if it's not already in the project (e.g., as a facet or a library module), you would typically use this to copy the control JAR to the user's project, as described below.

When you ship the control in a plug-in, the JAR file is located in the plug-in, NOT in the control user's project. To copy the JAR from the plug-in to the project that is using the control, you must insert code similar to the following into your insertion delegate. This will copy the control JAR to the user's project when your insertion delegate is called.

To create an insertion delegate, create a package in the **src** folder and create a file for the class of the insertion delegate.

Sample insertion delegate code is listed below. You will need to update the package and class name, of course.

```
package org.example.controls.workshop;

import java.io.File;


import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.jdt.core.IJavaElement;
import org.eclipse.jface.dialogs.ErrorDialog;
import org.eclipse.swt.widgets.Display;

import com.bea.workshop.controls.core.model.IControlInsertionDelegateContext;
import com.bea.workshop.controls.ui.actions.DefaultControlInsertionDelegate;

public class SampleInsertionDelegate extends DefaultControlInsertionDelegate {

        @Override
        public IJavaElement insertControl(IControlInsertionDelegateContext ctxt) {
                try {
                        File file = getFileFromPlugin(Activator.getDefault(), "/lib/TestControl.jar");
                        copyJarIfNecessary(ctxt, file, file.getName());
                        return super.insertControl(ctxt);
                } catch (Exception e) {
                        String message = "Error inserting control";
                        error(message,e);
                }
                return null;
        }

        private void error(String message, Exception e) {
                ErrorDialog.openError(Display.getCurrent().getActiveShell(),
                                                                "Control Insert Error",
                                                                message + " - " + e.getMessage(),
                                new Status(IStatus.ERROR,Activator.PLUGIN_ID,1,"Control Insert Error",e));
        }

}
```

The sample above covers the "insert your control into the project" case. To also do a custom wizard that collects parameters and inserts additional annotations, you can update the code to look like this:

```
public IJavaElement insertControl(IControlInsertionDelegateContext ctxt) {
  try {
        //Launch and complete your wizard here, collecting parameters as necessary
        // then proceed to the next steps to copy the file and use the parameters entered
        // as annotation values

        File file = getFileFromPlugin(Activator.getDefault(), "/lib/TestControl.jar");
        copyJarIfNecessary(ctxt, file, file.getName());

        HashMap<String, String> attrs = new HashMap<String, String>();
        attrs.put("attr", "aValue");

        return super.insertControl(ctxt,"org.example.controls.SampleControl.SamplePropertySet",attrs);
  } catch (Exception e) {
        String message = "Error inserting control";
        error(message,e);
```

```
  }
  return null;
}
```

The previous example is a convenience API if you have a single additional annotation to add. If you have multiple annotations to add, you could do something like this with a list of AnnotationInfo objects:

```
public IJavaElement insertControl(IControlInsertionDelegateContext ctxt) {
  try {
        //Launch and complete your wizard here, collecting parameters as necessary
        // then proceed to the next steps to copy the file and use the parameters entered
        // as annotation values

        File file = getFileFromPlugin(Activator.getDefault(), "/lib/TestControl.jar");
        copyJarIfNecessary(ctxt, file, file.getName());

        List infos = new ArrayList();
        HashMap<String, String> attrs1 = new HashMap<String, String>();
        attrs1.put("attr", "aValue");
        AnnotationInfo info1 = new AnnotationInfo(
                        "org.example.controls.SampleControl.SamplePropertySet",attrs1);
        infos.add(info1);

        HashMap<String, String> attrs2 = new HashMap<String, String>();
        attrs2.put("anotherAttribute", "aValue");
        AnnotationInfo info2 = new AnnotationInfo(
                        "org.example.controls.SampleControl.SamplePropertySet",attrs2);
        infos.add(info2);

        return super.insertControl(ctxt,infos);
  } catch (Exception e) {
        String message = "Error inserting control";
        Activator.getDefault().logError(message, e);
        error(message,e);
  }
  return null;
}
```

For more information on the APIs provided by the DefaultControlInsertionDelegate, see the Javadoc.

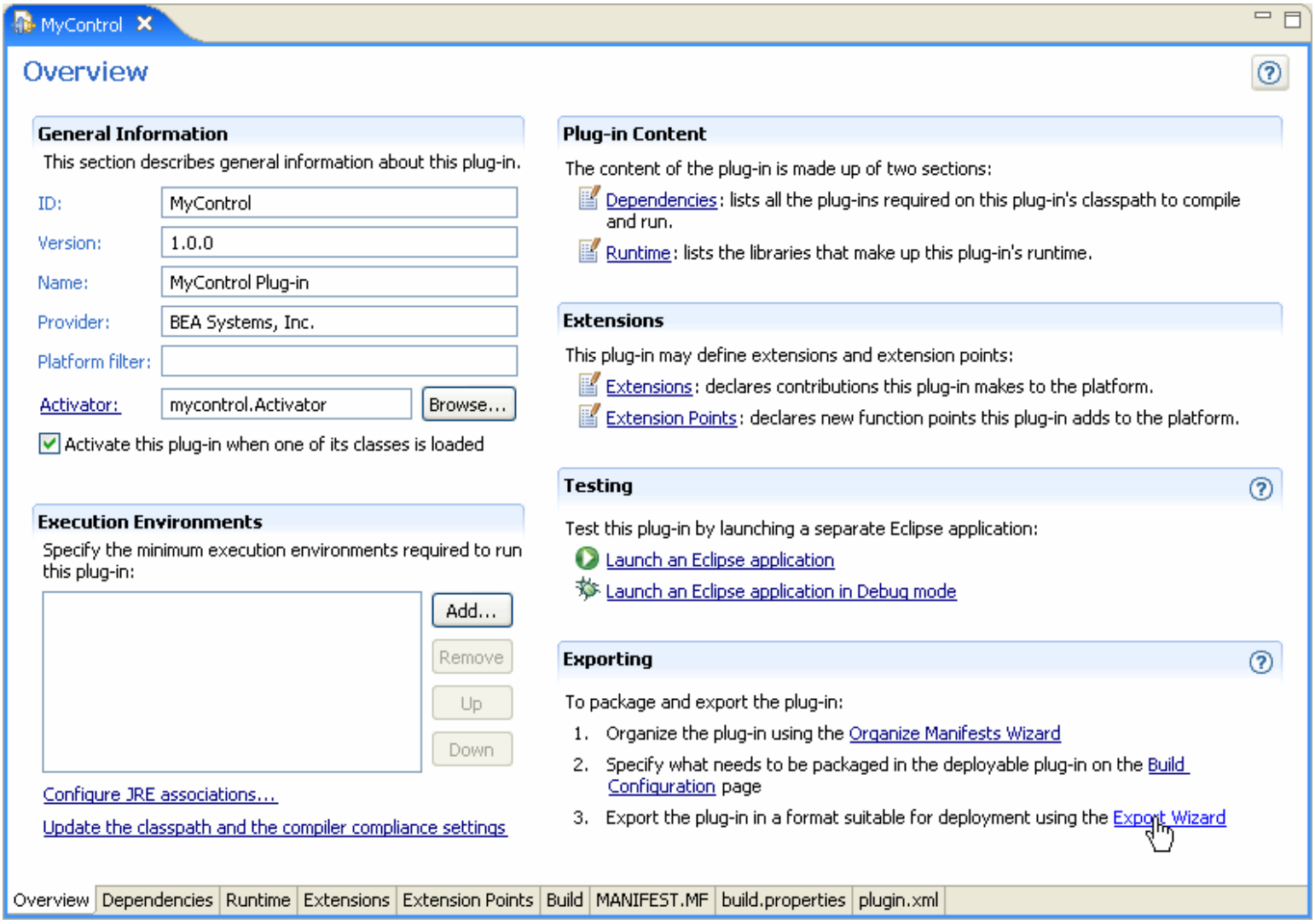
Step 7: Build and Test your Plug-in

Be sure that the plug-in includes the **lib** directly by clicking on the **Build** tab. Click on the **lib** folder under the **Binary Build** section to make sure that it is building correctly.

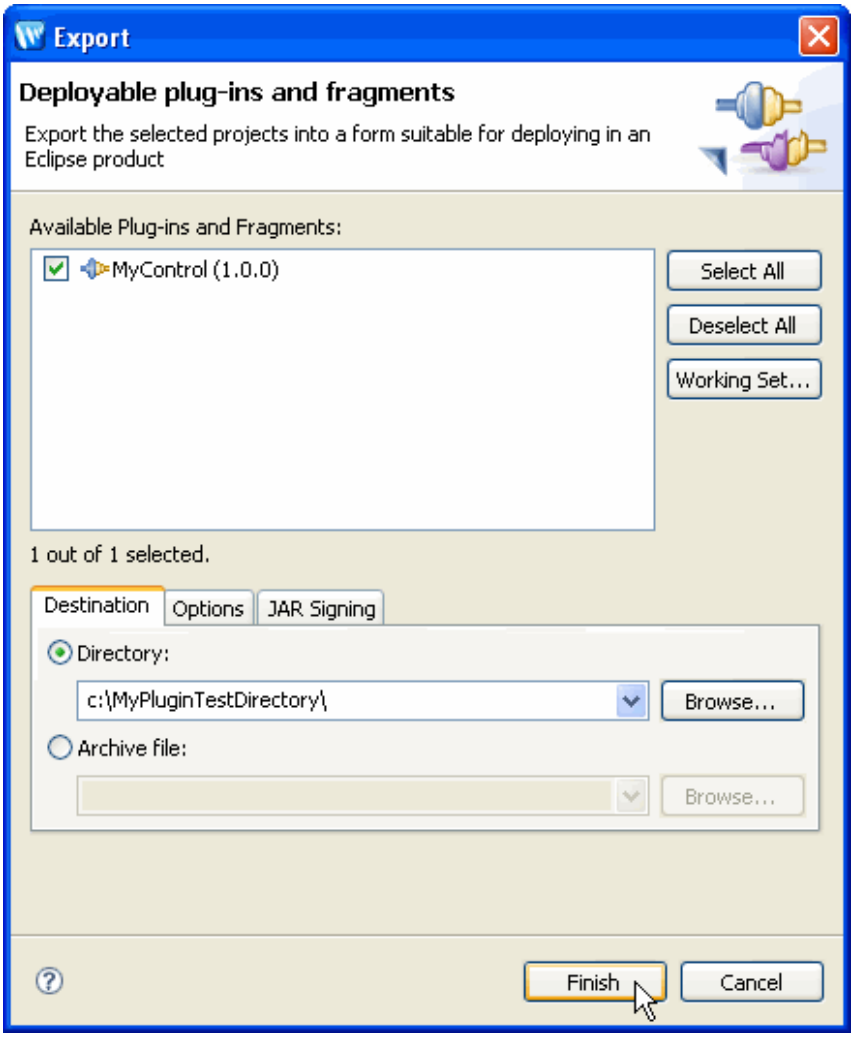
To run your plug-in, use the **Run As > Eclipse Application** command to test that your plug-in works correctly.

Step 8: Export the Plug-in

Click on the **Overview** tab. Click **Export Wizard** to create the plug-in JAR which will include the control JAR, the insertion delegate and any other required files. If this view is not available, you can open it by right clicking on META-INF/MANIFEST.MF and choosing **Open**.



From the export dialog, be sure to set the directory where the plug-in file will be created. Note that by default, the **Include source code** option is disabled so that the control's source code will not be available to plug-in users. Specify a destination directory for the plug-in and click **Finish** to create the control plug-in file.



You can then distribute the resulting plug-in file to other developers, like any other Eclipse plug-in.

Related Topics

Developing Custom Controls

Exporting Controls into JARs

Control Dialogs

These topics describe dialogs and wizards available for creating custom and system controls.

Topics Included in This Section

New Custom Control Dialog
Create a new Beehive-based custom control.

New Extensible Control Dialog
Create a new Beehive-based extensible control.

Insert Control Event Handler Dialog
Create a event handler based on a event method.

New EJB Wizard
Create a new EJB Control.

New JDBC Wizard
Create a new JDBC Control.

New JMS Wizard
Create a new JMS Control.

Service Control Generation Wizard
Generate a Web Service Control from a WSDL file.

Select Control Dialog
Add an Existing Control.

Related Topics

none.

New Control Dialog

Use this dialog to create a new custom control.

How To Open this Dialog

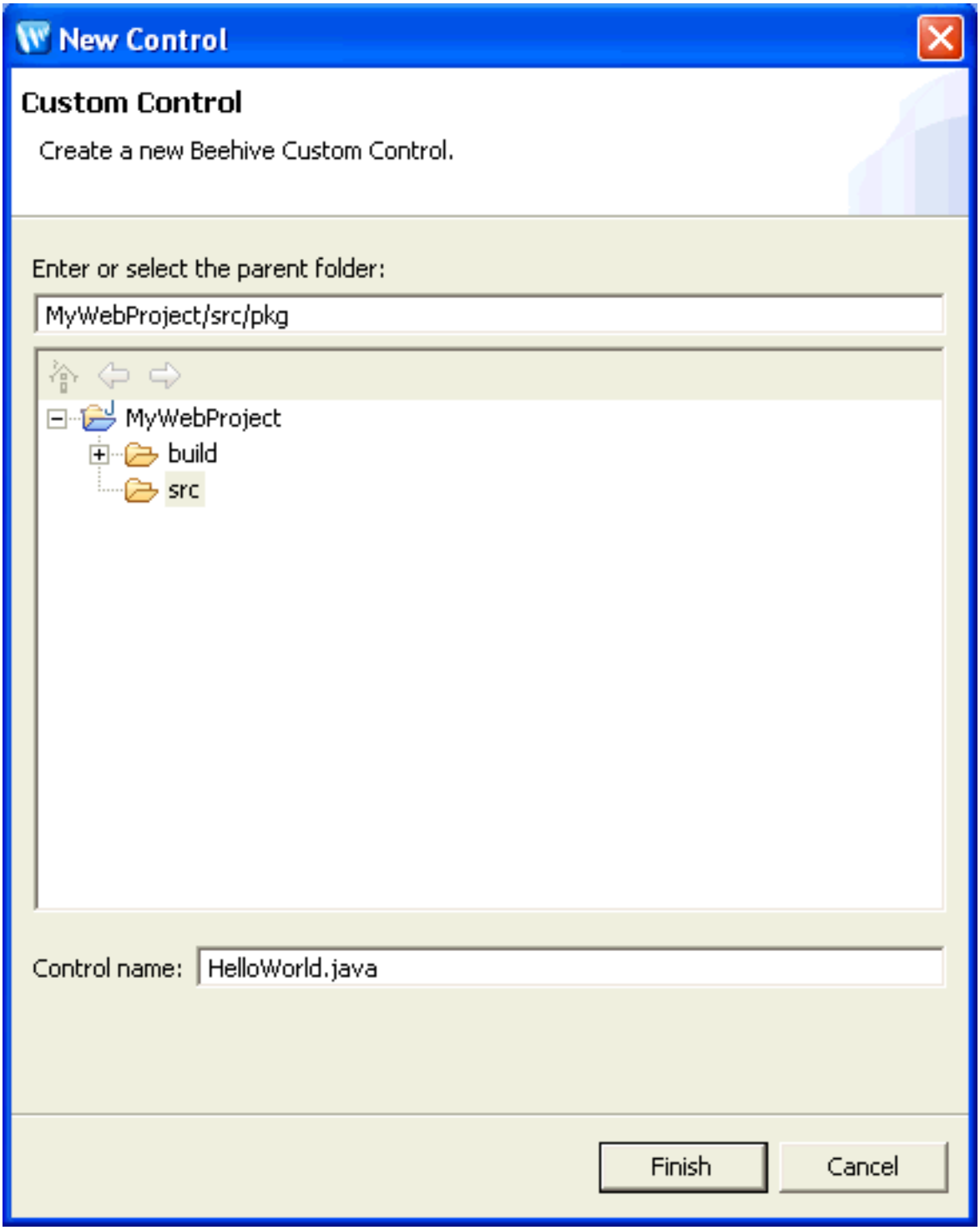
To open the **New Control** dialog, select **File > New > Other > Controls > Custom Control**.

How To Use the Dialog

In **Enter or select the parent folder**, select to the directory location, where the new control is be created. If the desired directory does not already exit, you may enter the directory path to create the desired directory.

In **Control name** enter a valid Java class name. Two files will be created based on this name:

1. <ControlName>.java
2. <ControlName>Impl.java



Related Topics

Custom Controls

New Extensible Control Dialog

Use this dialog to create a new extensible custom control. This is a custom control that can be customized by extension of the interface class.

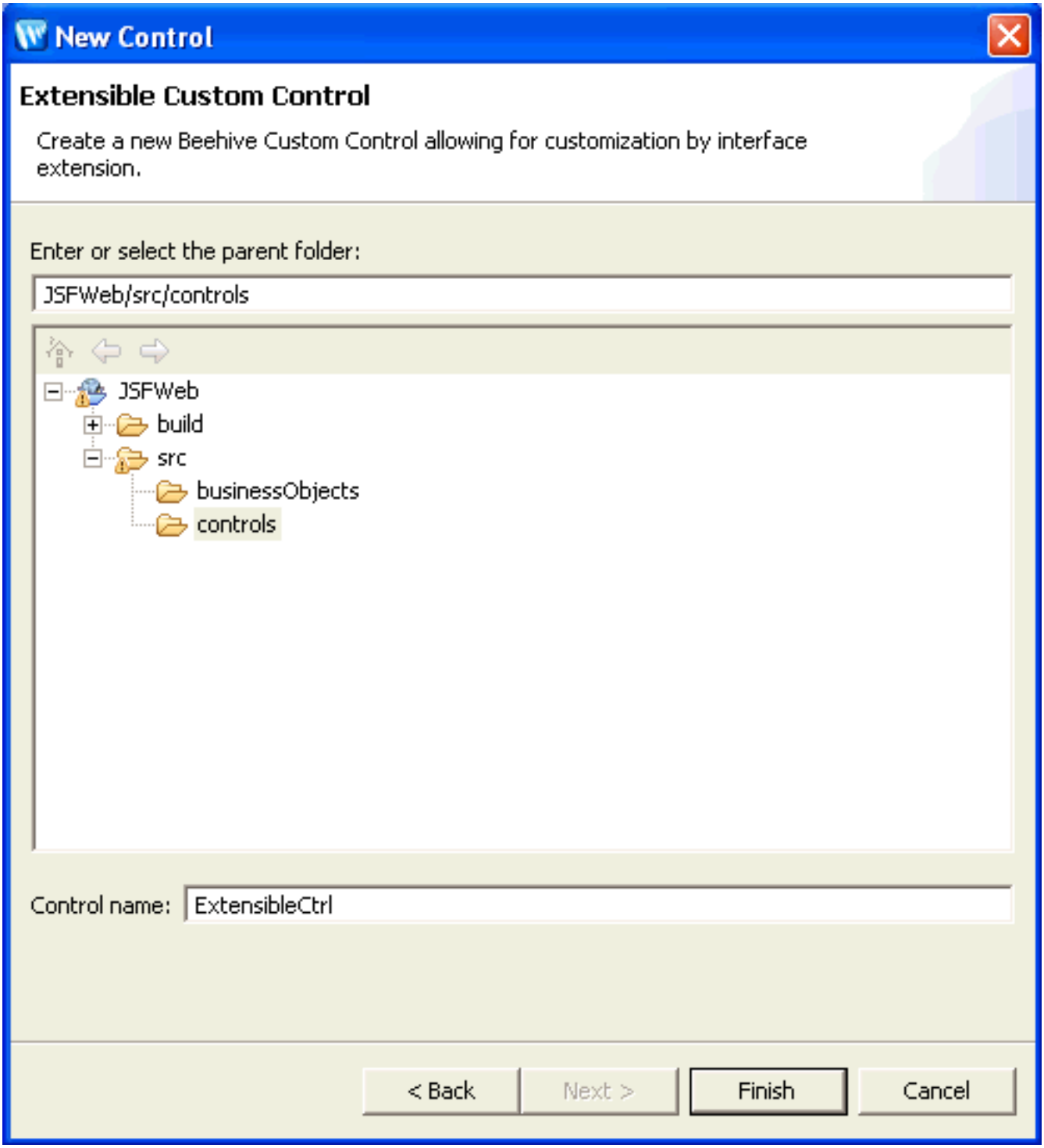
How To Open This Dialog

To open this dialog, select **File > New > Other > Controls > Extensible Custom Control**.

How To Use This Dialog

Upon completion of the dialog, two files will be created based on name entered in the **Control name** field:

1. <ControlName>.java
2. <ControlName>Impl.java



Related Topics

Custom Controls

Insert Control Event Handler Dialog

Use this dialog to add a control event handler based on an event method.

How To Open this Dialog

To open the this dialog, open a client class which contains a control declaration, for example:

```
        @Control
    private HWCallbackServiceControl wsControl;
```

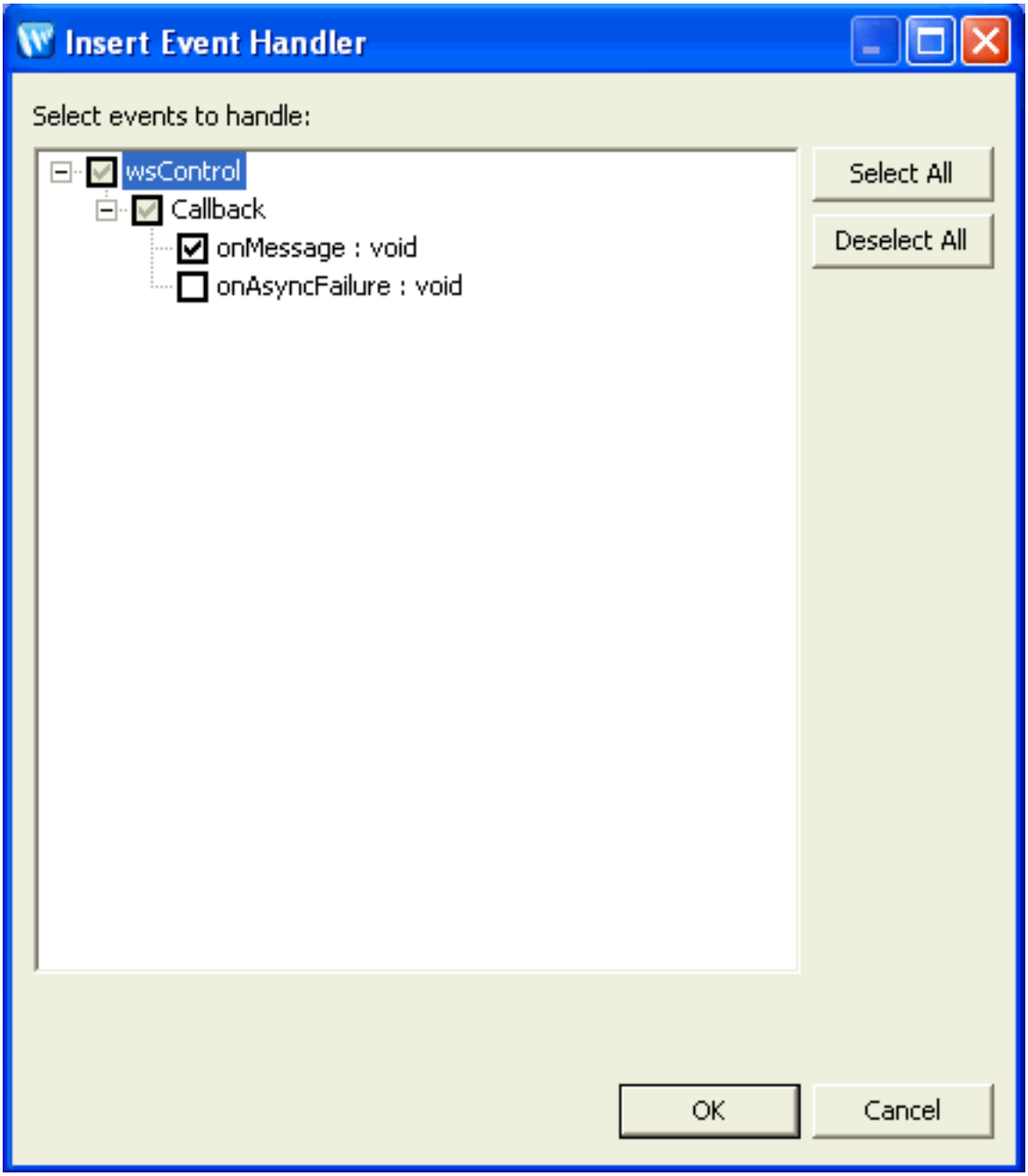
The declared control must contain an event set for the dialog to appear.

Right click anywhere within the client class source view and select **Insert > Control Event Handler**.

The dialog will appear displaying a list of event handlers corresponding to the event sets in the declared control.

How To Use the Dialog

Select the control (first level nodes), event set (second level nodes) and method (third level nodes) for which an event handler should be constructed.



Related Topics

Custom Controls

Handling Control Events

New EJB Control Dialog

Use this dialog to create a new EJB control.

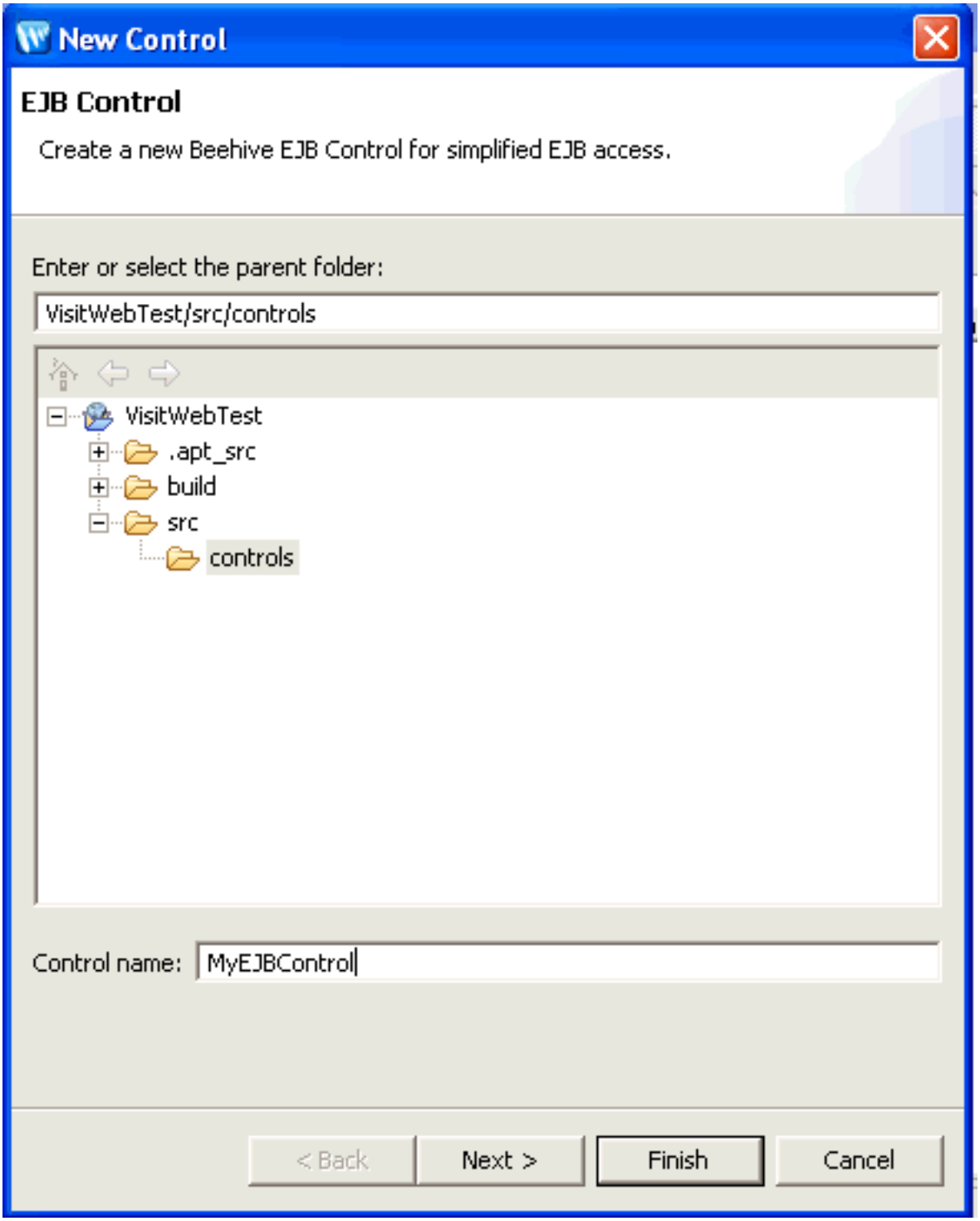
How To Open This Dialog

You can open the dialog in one these ways:

- From any perspective, select **File > New > Other > Controls > EJB Control**.
- In the **J2EE** perspective, select **File > New > EJB Control**.
- In the **Page Flow** perspective, right-click the **Referenced Controls** node on the **Page Flow Explorer** tab, and select **Add Control > New System Control > EJB Control > Ok**.
- From the source editor window, right click and choose **Insert > Control** then expand **New System Control** and choose **EJB Control**.

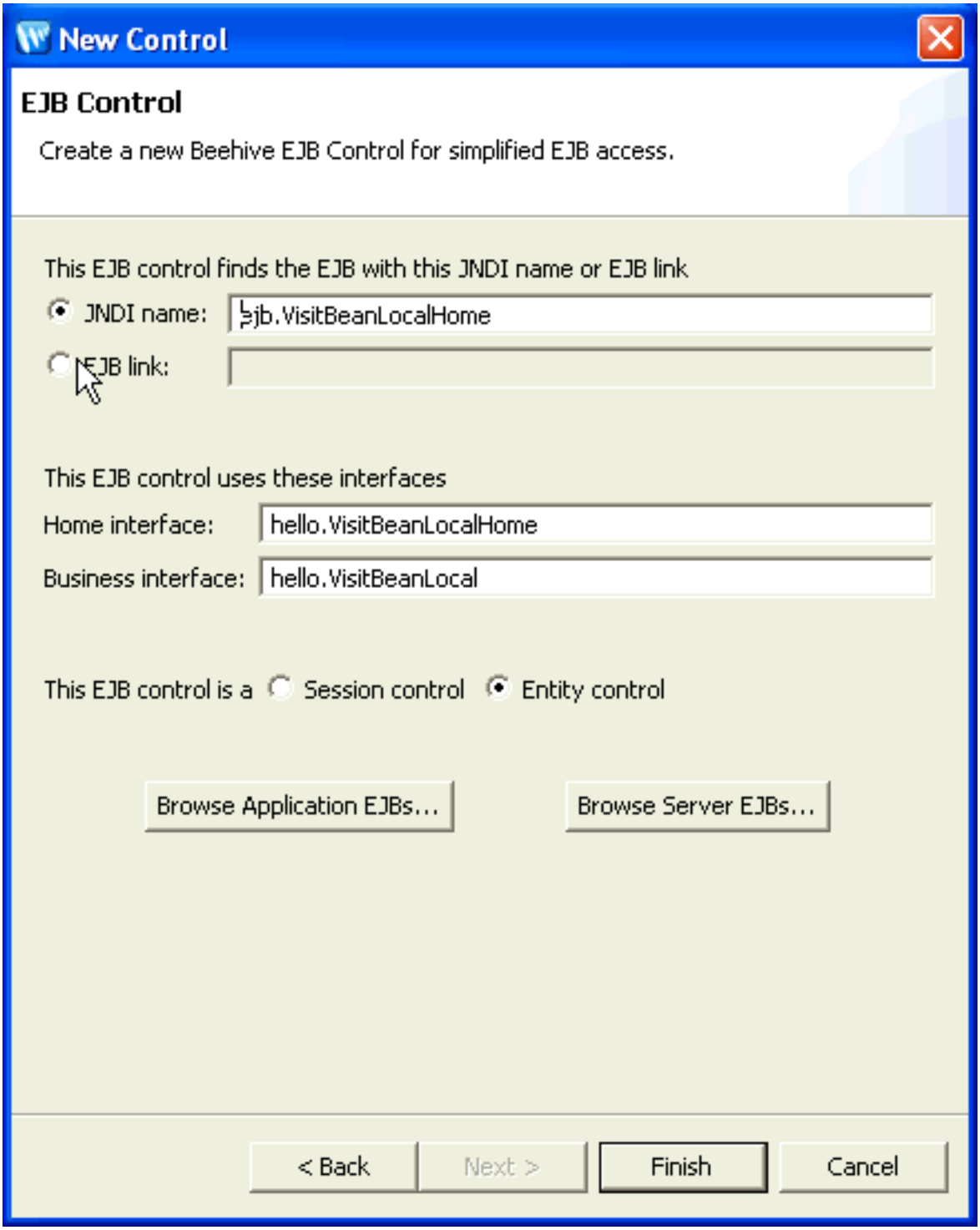
How To Use This Dialog

In **Enter or select the parent folder**, select to the directory location, where the new control is be created. If the desired directory does not already exit, you may enter the directory path to create the desired directory.



The **Browser Application EJBs** button will display a list of EJBs in the current project.

Provided that WebLogic Server is running, you can browse for deployed EJBs using the **Browse Server EJBs** button. If the server is not running and you know the name, you can just type it in.



Related Topics

none.

New JDBC Control Dialog

Use this dialog to create a new JDBC control.

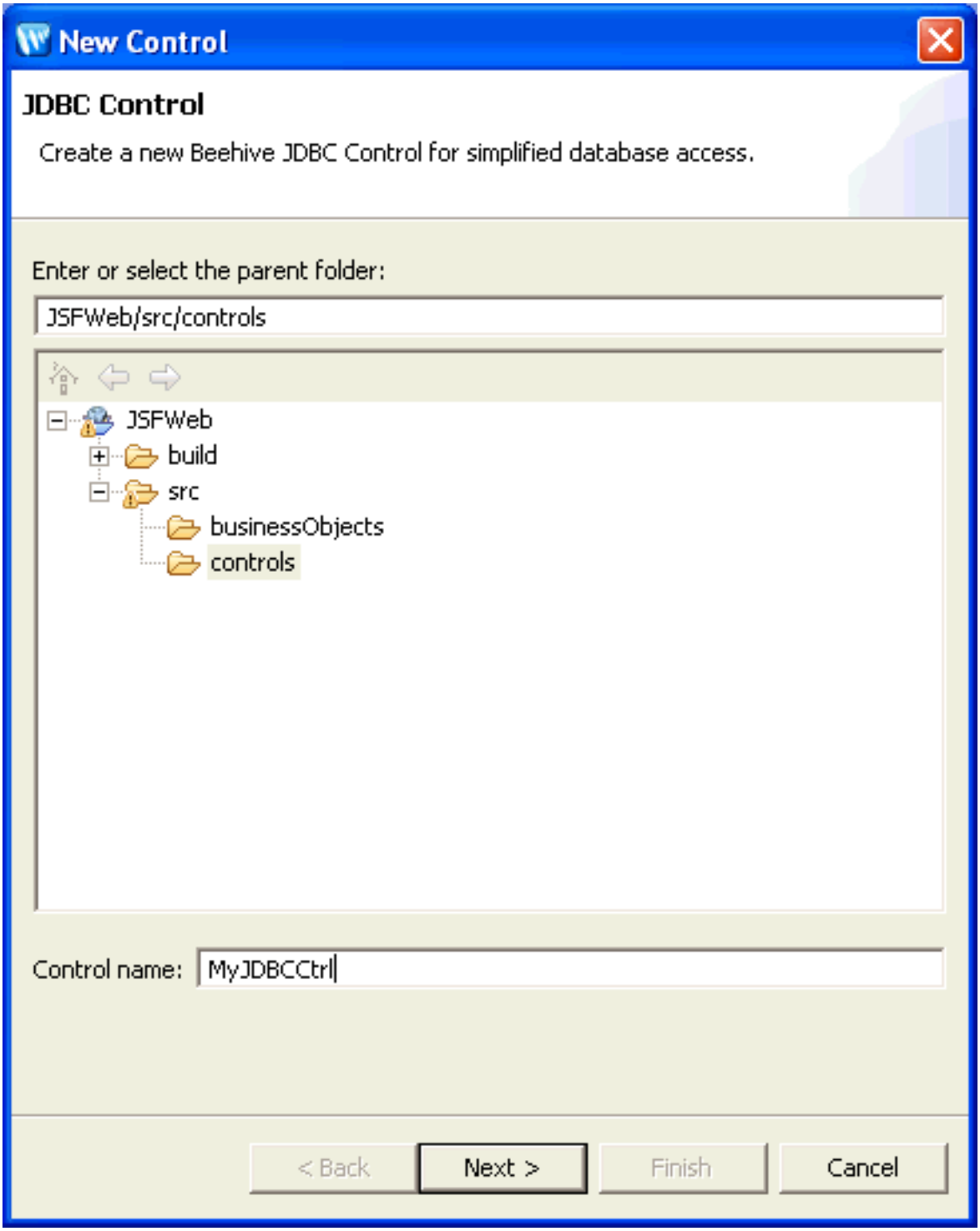
How To Open This Dialog

You can open the dialog in one these ways:

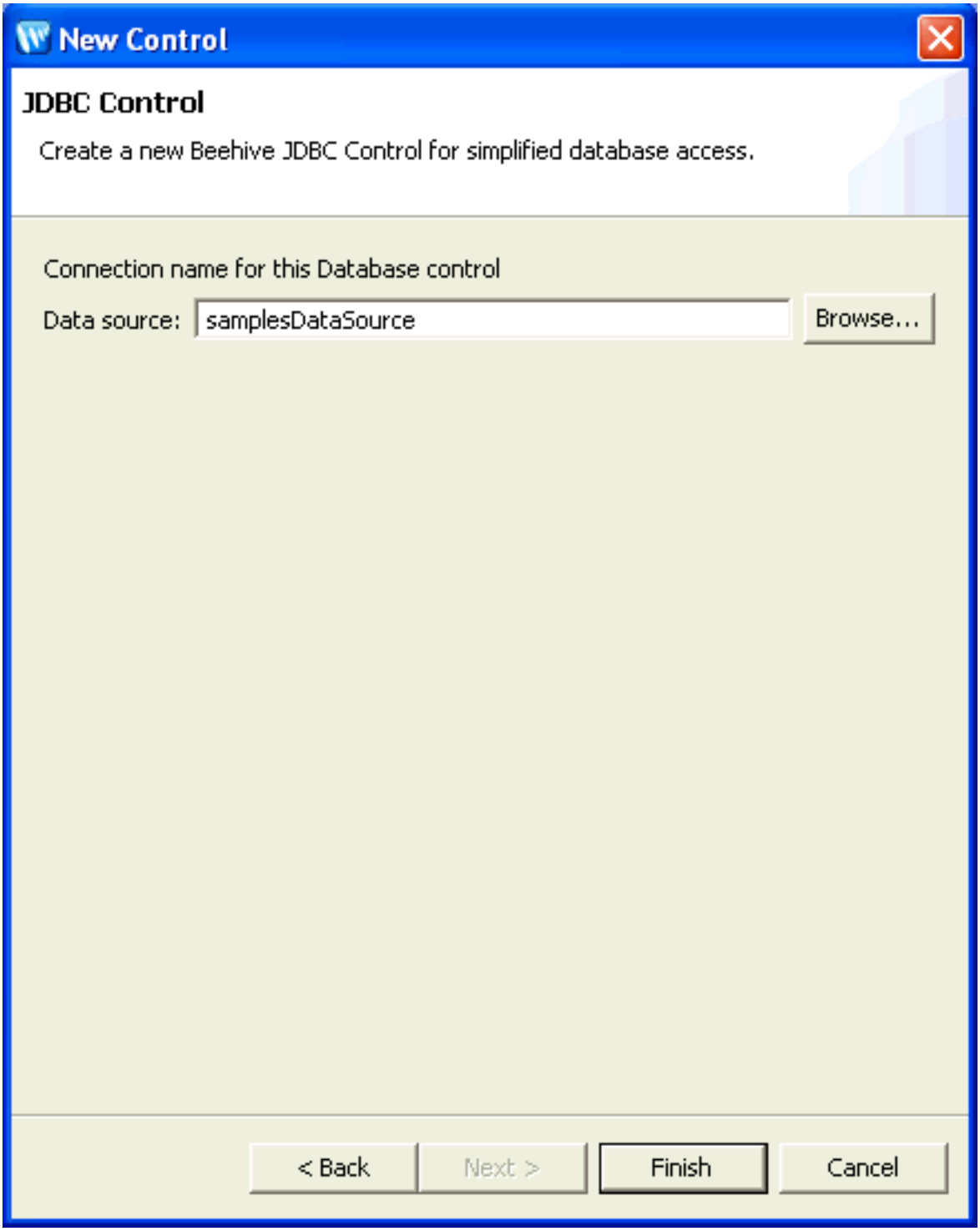
- From any perspective, select **File > New > Other > Controls > JDBC Control**.
- In the **J2EE** perspective, select **File > New > JDBC Control**.
- In the **Page Flow** perspective, right-click the **Referenced Controls** node on the **Page Flow Explorer** tab, and select **Add Control > New System Control > JDBC Control > Ok**.
- From the source editor window, right click and choose **Insert > Control** then expand **New System Control** and choose **JDBC Control**.

How To Use This Dialog

In **Enter or select the parent folder**, select to the directory location, where the new control is be created. If the desired directory does not already exit, you may enter the directory path to create the desired directory.



Provided that WebLogic Server is running, you can browse for available data sources. If the server is not running and you know the name, you can just type it in.



Related Topics

none

New JMS Control Dialog

Use this dialog to create a new JMS control.

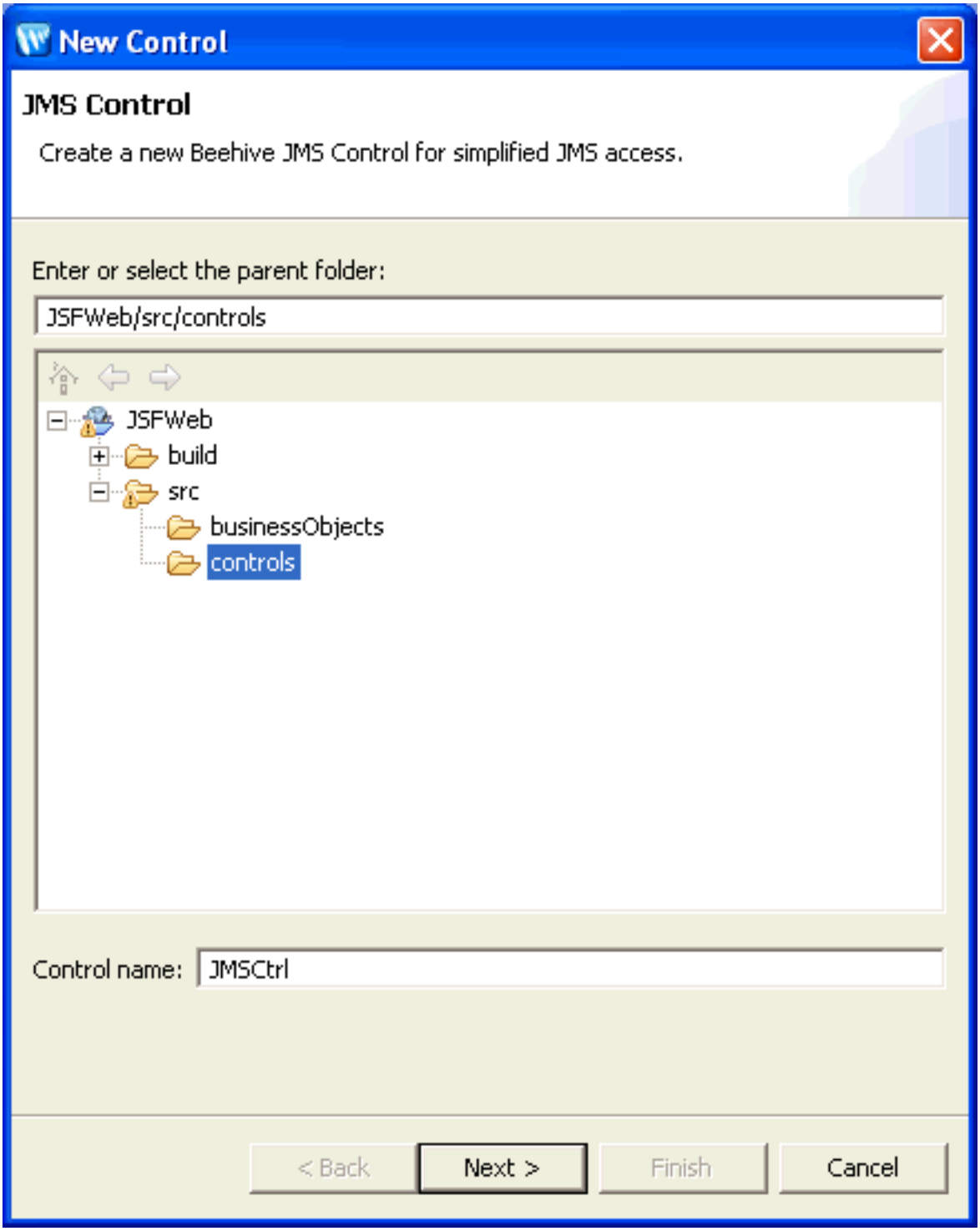
How To Open This Dialog

You can open the dialog in one these ways:

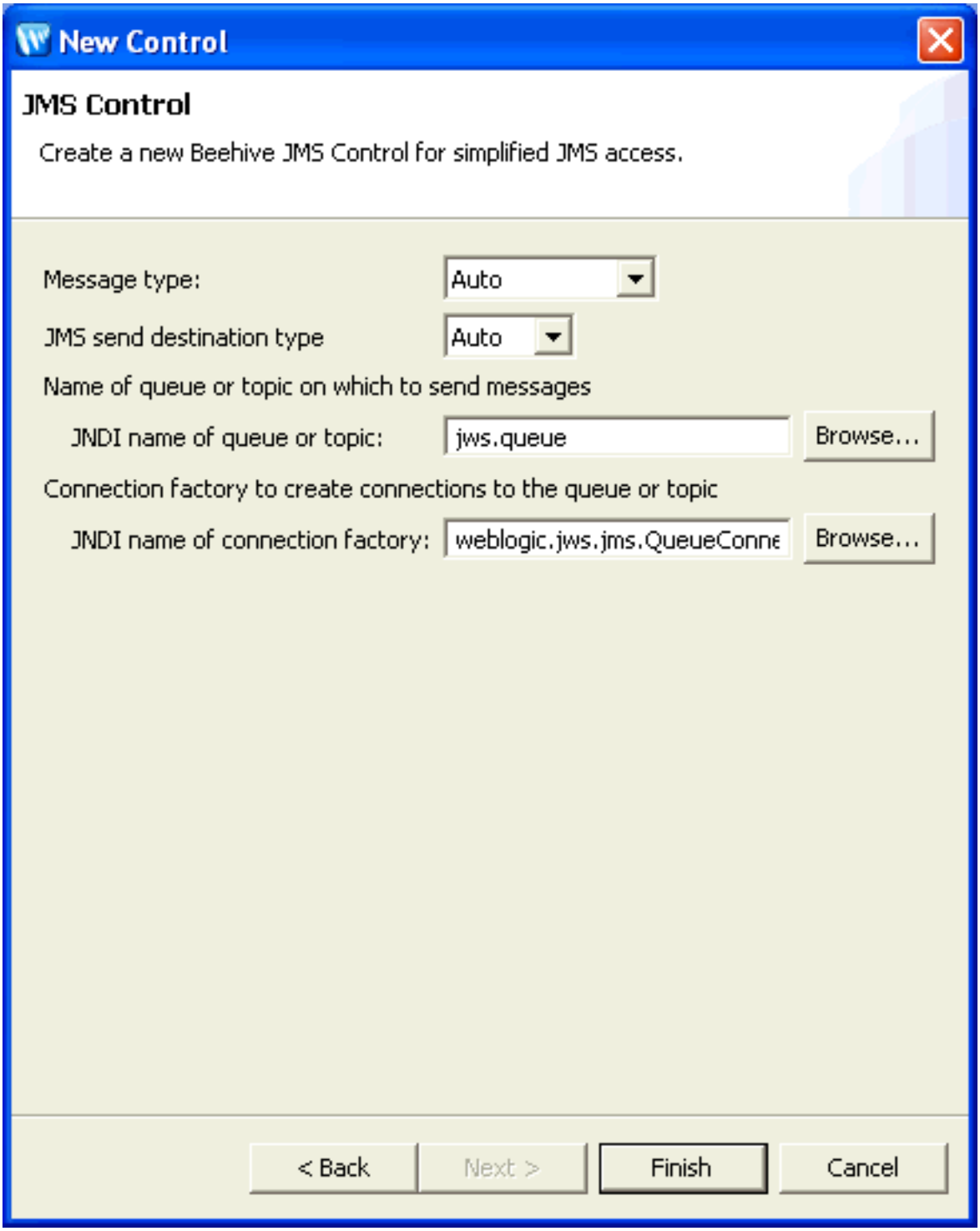
- From any perspective, select **File > New > Other > Controls > JMS Control**.
- In the **J2EE** perspective, select **File > New > JMS Control**.
- In the **Page Flow** perspective, right-click the **Referenced Controls** node on the **Page Flow Explorer** tab, and select **Add Control > New System Control > JMS Control > Ok**.
- From the source editor window, right click and choose **Insert > Control** then expand **New System Control** and choose **JMS Control**.

How To Use This Dialog

In **Enter or select the parent folder**, select to the directory location, where the new control is be created. If the desired directory does not already exit, you may enter the directory path to create the desired directory.



Provided that WebLogic Server is running, you can browse for available JMS queues, topics and connection factories.



Related Topics

none.

Service Control Generation Wizard

Use this dialog to create a new service control.

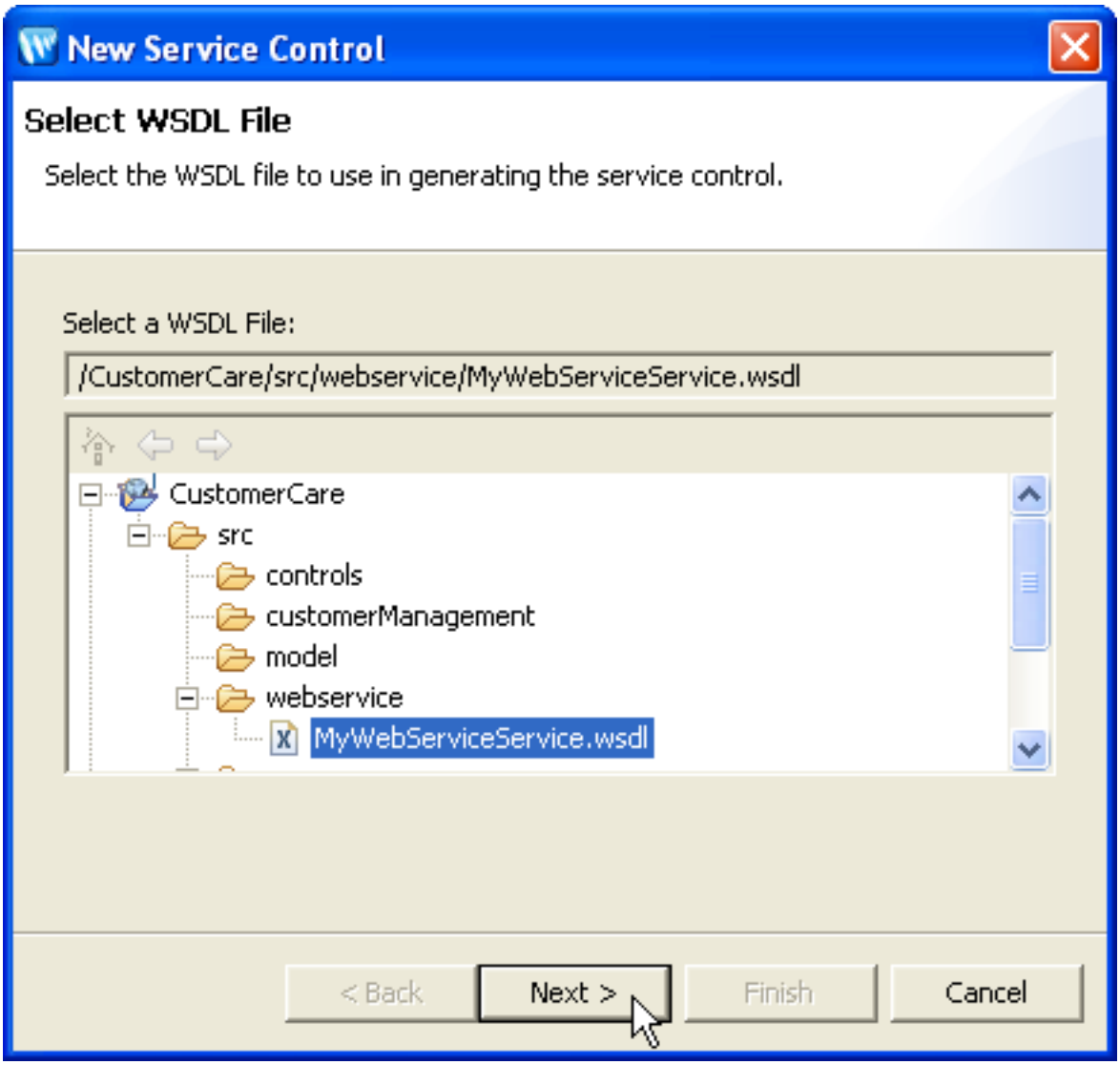
To create a service control using this dialog, you must first have a local copy of the WSDL file for the target web service.

How To Open This Dialog

To open this dialog, right-click a folder in the **Project Explorer** view and select **New > Service Control**.

How To Use This Dialog

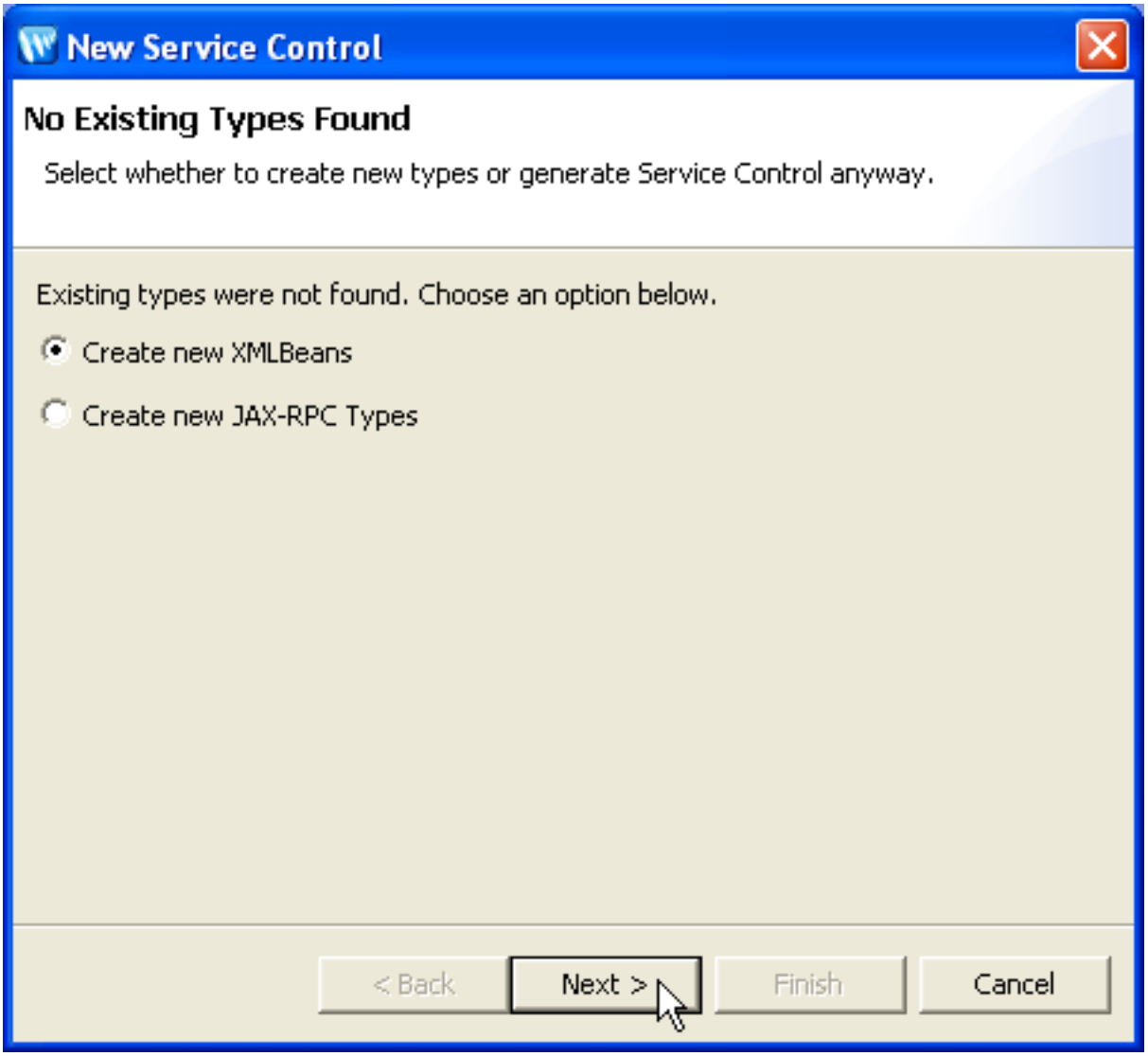
On the first page, navigate to the WSDL file for the target web service.



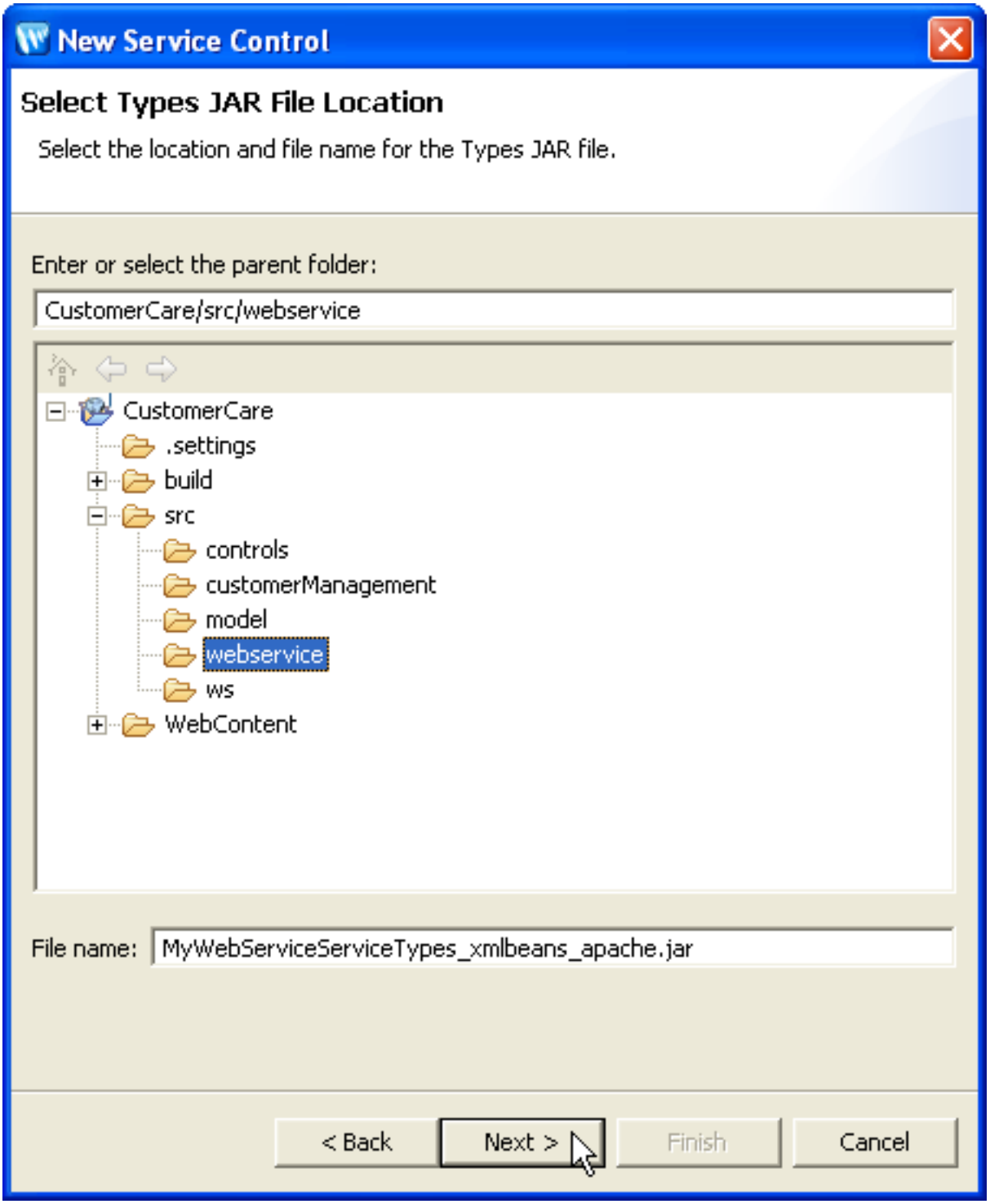
On the second page, select the content types in the generated JAR file. Depending on your

selection, either JAX-RPC or XMLBean types will be generated.

This page will not appear if your service has only simple types or if the necessary types are already available.

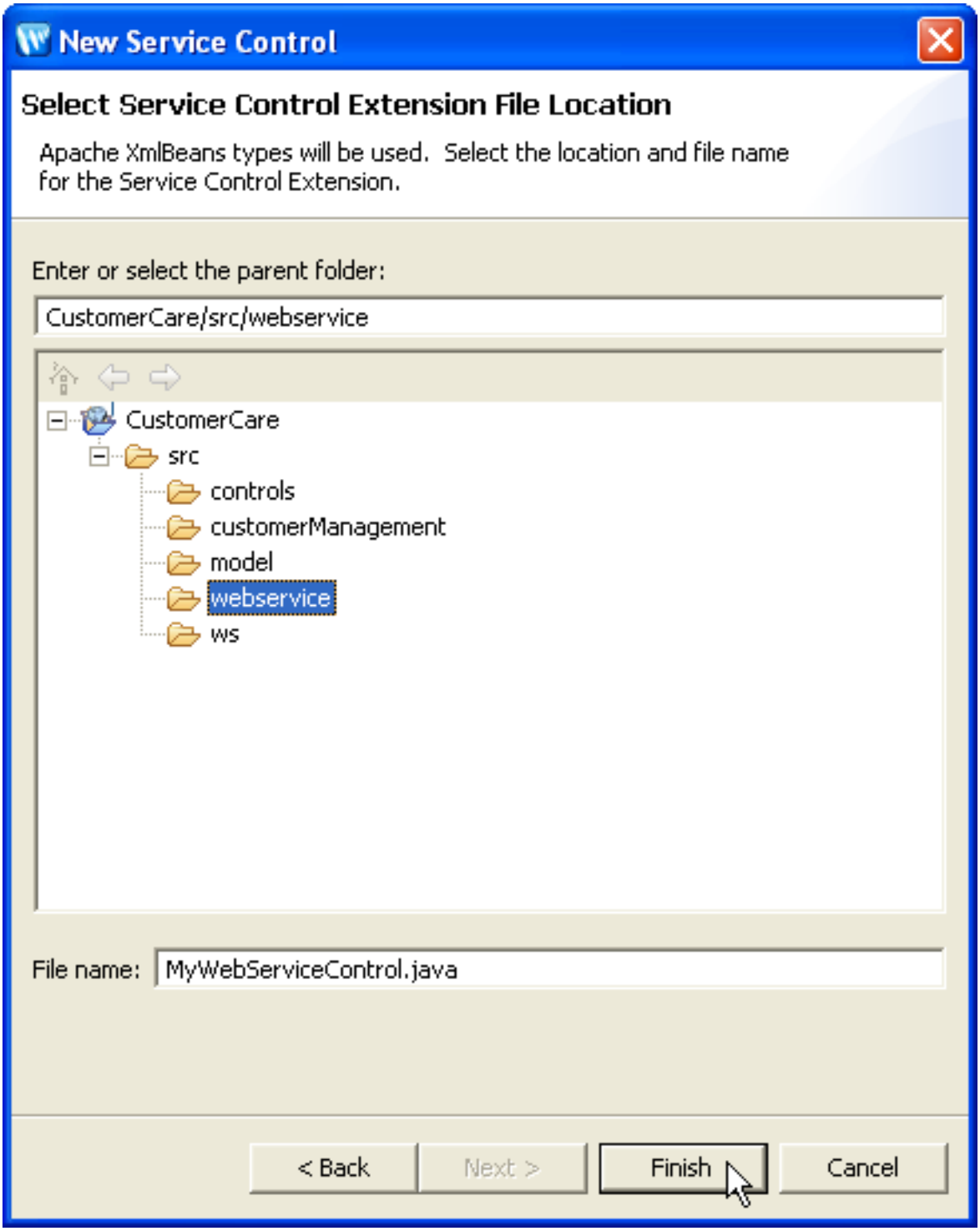


On the third page, select the location where you wish the JAR file to be saved. By default, the JAR will be saved to WEB-INF/lib (in a web or web service project) or APP-INF/lib (in a utility project). If you choose a different location, you must ensure that the chosen location is on the classpath.



On the fourth page, in the **Enter or select the parent folder** field, enter the directory location where the new service control is be created. If the desired directory does not already exit, you may type the directory path to create the desired directory.

In the **File name** field, enter the desired name of the service control. The default name is the name of the WSDL appended with "Control".



Related Topics

Types JAR File Generation Wizard

New Web Service From WSDL Wizard

Select Control Dialog

Use this dialog to select an existing control or create a new control.

How To Open This Dialog

You can open the dialog in one of the following ways:

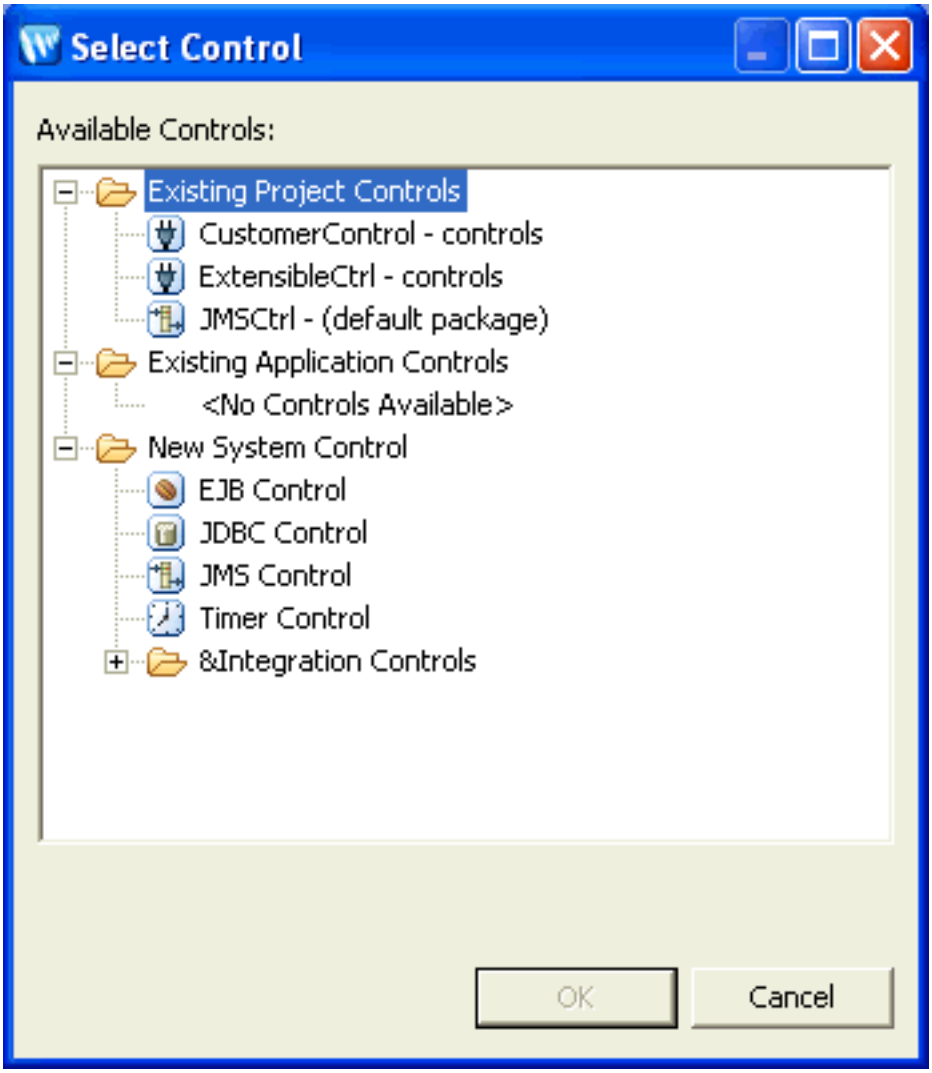
- From the **Page Flow** perspective, right-click the **Referenced Controls** node on the **Page Flow Explorer** tab and select **Add Control**.
- From the **J2EE** or **Page Flow** perspectives, right-click anywhere within a Java source file and select **Insert > Control**.

How To Use This Dialog

The **Existing Project Controls** node displays the controls that reside in the current project.

The **Existing Application Controls** node displays the controls that reside in other projects in the same workspace. Only controls from utility projects and projects dependent on the current project are displayed.

The **New System Control** node displays wizards for creating a new control.



Related Topics

none.