



# BEA WebLogic Workshop™ Help

Version 8.1 SP4  
December 2004

# Copyright

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

## Restricted Rights Legend

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

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

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

## Trademarks or Service Marks

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

All other trademarks are the property of their respective companies.

# Table of Contents

<b>WebLogic Workshop Tutorials.....</b>	<b>1</b>
<b>The Getting Started Tutorials.....</b>	<b>2</b>
<b>Getting Started: Web Services.....</b>	<b>3</b>
<b>Getting Started: Web Service Core Concepts.....</b>	<b>5</b>
<b>Step 1: Create a Web Service Project.....</b>	<b>8</b>
<b>Step 2: Implement Synchronous Communication.....</b>	<b>12</b>
<b>Step 3: Implement Asynchronous Communication.....</b>	<b>15</b>
<b>Summary: Getting Started Web Services Tutorial.....</b>	<b>22</b>
<b>Getting Started: Web Applications.....</b>	<b>23</b>
<b>Getting Started: Web Application Core Concepts.....</b>	<b>24</b>
<b>Step 1: Create a Web Application Project.....</b>	<b>26</b>
<b>Step 2: Submitting Data.....</b>	<b>29</b>
<b>Step 3: Processing Data.....</b>	<b>39</b>
<b>Step 4: Displaying Results.....</b>	<b>45</b>
<b>Summary: Getting Started Web Application Tutorial.....</b>	<b>48</b>
<b>Getting Started: Java Controls.....</b>	<b>49</b>
<b>Getting Started: Java Control Core Concepts.....</b>	<b>50</b>
<b>Step 1: Create a Control Project.....</b>	<b>53</b>
<b>Step 2: Define the Database Control.....</b>	<b>57</b>
<b>Step 3: Define the Custom Control.....</b>	<b>62</b>
<b>Step 4: Test the Controls.....</b>	<b>67</b>
<b>Summary: Getting Started Control Tutorial.....</b>	<b>71</b>
<b>Getting Started: Enterprise JavaBeans.....</b>	<b>72</b>

# Table of Contents

<b>Getting Started: EJB Core Concepts.....</b>	<b>73</b>
<b>Step 1: Create an EJB Project.....</b>	<b>76</b>
<b>Step 2: Define the Entity Bean.....</b>	<b>80</b>
<b>Step 3: Define the Session Bean.....</b>	<b>87</b>
<b>Step 4: Test the EJBs.....</b>	<b>92</b>
<b>Summary: Getting Started EJB Tutorial.....</b>	<b>97</b>
<b>Getting Started: Portals.....</b>	<b>98</b>
<b>Getting Started: Portal Core Concepts.....</b>	<b>100</b>
<b>Step 1: Create a Portal Project.....</b>	<b>104</b>
<b>Step 2: Organize Content in a Portal.....</b>	<b>109</b>
<b>Step 3: Personalize a Web Application.....</b>	<b>117</b>
<b>Summary: Getting Started Portal Tutorial.....</b>	<b>127</b>
<b>Tutorial: Java Control.....</b>	<b>128</b>
<b>Step 1: Begin the Investigate Java Control.....</b>	<b>131</b>
<b>Step 2: Add a Database Control.....</b>	<b>147</b>
<b>Step 3: Add a Web Service Control.....</b>	<b>156</b>
<b>Step 4: Add a JMS Control.....</b>	<b>164</b>
<b>Step 5: Add an EJB Control.....</b>	<b>171</b>
<b>Step 6: Add Support for Cancellation and Exception Handling.....</b>	<b>175</b>
<b>Step 7: Transport Security.....</b>	<b>181</b>
<b>Step 8: Compile the Java Control and Add a Property.....</b>	<b>189</b>
<b>Summary: Java Control Tutorial.....</b>	<b>197</b>
<b>Tutorial: Web Services.....</b>	<b>198</b>

# Table of Contents

<b>Step 1: Begin the Investigate Web Service.....</b>	<b>200</b>
<b>Step 2: Add Support for Asynchronous Communication.....</b>	<b>208</b>
<b>Step 3: Add Support for Synchronous Communication.....</b>	<b>217</b>
<b>Step 4: Add Support for XML Mapping.....</b>	<b>222</b>
<b>Step 5: Add a Script for XML Mapping.....</b>	<b>231</b>
<b>Step 6: Web Service Security.....</b>	<b>235</b>
<b>Summary: Web Service Tutorial.....</b>	<b>244</b>
<b>Tutorial: Page Flow.....</b>	<b>246</b>
<b>Step 1: Begin the Page Flow Tutorial.....</b>	<b>249</b>
<b>Step 2: Processing Data with Page Flows.....</b>	<b>255</b>
<b>Step 3: Create the JSP Pages.....</b>	<b>264</b>
<b>Step 4: Role-Based Security.....</b>	<b>273</b>
<b>Summary: Page Flow Tutorial.....</b>	<b>281</b>
<b>Tutorial: Building Your First Business Process.....</b>	<b>282</b>
<b>Tutorial: Building Your First Data Transformation.....</b>	<b>283</b>
<b>Portal Tutorials.....</b>	<b>286</b>
<b>Tutorial: Building Your First Portal.....</b>	<b>287</b>
<b>Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server.....</b>	<b>288</b>
<b>Step 2: Create a Portal.....</b>	<b>289</b>
<b>Tutorial: Changing a Portal's Look &amp; Feel and Navigation.....</b>	<b>291</b>
<b>Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server.....</b>	<b>292</b>
<b>Step 2: Change the Portal Look &amp; Feel and Navigation.....</b>	<b>293</b>
<b>Tutorial: Showing Personalized Content in a Portlet.....</b>	<b>295</b>

## Table of Contents

<b>Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server.....</b>	<b>297</b>
<b>Step 2: Create a User Profile Property Set.....</b>	<b>298</b>
<b>Step 3: Create Two Users.....</b>	<b>299</b>
<b>Step 4: Load Content.....</b>	<b>300</b>
<b>Step 5: Create a JSP.....</b>	<b>301</b>
<b>Step 6: Add Content Selectors to the JSP.....</b>	<b>302</b>
<b>Step 7: Create a Portlet with the JSP.....</b>	<b>305</b>
<b>Step 8: Test the Personalized Portlet.....</b>	<b>306</b>
<b>Tutorial: Creating a Login Control Page Flow Using the Wizard.....</b>	<b>308</b>
<b>Step 1: Create a Page Flow Using the Wizard.....</b>	<b>309</b>
<b>Step 2: Place the Page Flow in a Portlet.....</b>	<b>313</b>
<b>Tutorial: Using Page Flows Inside Portlets.....</b>	<b>319</b>
<b>Step 1: Create a Portal Application.....</b>	<b>320</b>
<b>Step 2: Create a Simple Navigation Page Flow.....</b>	<b>321</b>
<b>Step 3: Create Portlets that Communicate.....</b>	<b>327</b>
<b>Step 4: Create an EJB Control Page Flow portlet.....</b>	<b>334</b>
<b>Tutorial: Enterprise JavaBeans.....</b>	<b>337</b>
<b>Part One: Building Your First EJBs.....</b>	<b>340</b>
<b>Step 1: Begin the EJB Tutorial.....</b>	<b>341</b>
<b>Step 2: Import an EJB from a JAR.....</b>	<b>345</b>
<b>Step 3: Create a Session Bean.....</b>	<b>349</b>
<b>Step 4: Run a Web Application.....</b>	<b>353</b>
<b>Part Two: Defining an Entity Relationship.....</b>	<b>358</b>

## Table of Contents

<b>Step 5: Create a New Entity Bean.....</b>	<b>359</b>
<b>Step 6: Update the Band Entity Bean.....</b>	<b>365</b>
<b>Step 7: Update the Session Bean.....</b>	<b>369</b>
<b>Step 8: Run a Web Application.....</b>	<b>371</b>
<b>Step 9: Build and Run a Web Service Application.....</b>	<b>373</b>
<b>Part Three: Adding a Message–Driven Bean.....</b>	<b>376</b>
<b>Step 10: Create a Message–Driven Bean.....</b>	<b>378</b>
<b>Step 11: Update the Recording Bean.....</b>	<b>382</b>
<b>Step 12: Update the Music Bean.....</b>	<b>384</b>
<b>Step 13: Run a Web Application.....</b>	<b>389</b>
<b>Summary: Enterprise JavaBean Tutorial.....</b>	<b>391</b>

# WebLogic Workshop Tutorials

WebLogic Workshop includes tutorials that walk you through the process of creating each of the core component types of WebLogic Workshop applications.

## Topics Included in This Section

### The Getting Started Tutorials

The Getting Started tutorials are a series of independent tutorials that provide a quick step-by-step introduction to the core components of WebLogic Workshop.

#### Tutorial: Java Controls

Provides a step-by-step guide to building a custom Java control.

#### Tutorial: Web Services

Provides a step-by-step guide to building Java Web Services.

#### Tutorial: Page Flows

Provides an introduction to developing web applications based on Java Page Flows.

#### Tutorial: Your First Business Process

Provides step-by-step instructions to build a basic integration application.

#### Tutorial: Your First Data Transformation

Provides step-by-step instructions to building an XML-to-XML data transformation in an integration application.

#### Tutorial: Enterprise JavaBeans

Provides step-by-step instructions to developing Enterprise JavaBeans.



# The Getting Started Tutorials

The Getting Started tutorials is a series of short tutorials that offers a basic step-by-step introduction to the core J2EE components and how you can develop these components with WebLogic Workshop. Each tutorial is self-contained, and can be run independently of any of the other tutorials. You can do any or all of these tutorials in any order you want.

## Topics Included in This Section

Getting Started: Web Services

Provides a step-by-step introduction to building Web Services.

Getting Started: Web Applications

Provides a step-by-step introduction to building Web Applications.

Getting Started: Controls

Provides a step-by-step introduction to building Java controls.

Getting Started: Enterprise JavaBeans

Provides a step-by-step introduction to building EJBs.

Getting Started: Portal

Provides a step-by-step introduction to building Portal applications.

Related Topics

The Getting Started Samples

# Getting Started: Web Services

## The Problem with Making Applications Interact

Today, companies rely on thousands of different software applications each with their own role to play in running a business. To name just a few, database applications store information about customers and inventories, web applications allow customers to browse and purchase products online, and sales tracking applications help business identify trends and make decisions for the future.

These different software applications run on a wide range of different platforms and operating systems, and they are implemented in different programming languages. As a result, it is very difficult for different applications to communicate with one another and share their resources in a coordinated way.

Take, for example, a company that has its customer data stored in one application, its inventory data stored in another application, and its purchasing orders from customers in a third. Until now, if this company wanted to integrate these different systems, it had to employ developers to create custom bridging software to allow the different applications to communicate with one another. However, these sorts of solutions are often piecemeal and time consuming. As soon as a change is made to one application, corresponding changes have to be made to the applications linked to it and to the bridges that link the applications together.

Another solution would be for a human being to bridge the communication gap between two applications, an inefficient and error-prone approach.

## The Web Services Solution

To solve the problem of application-to-application communication, businesses need a standardized way for applications to communicate with one another over networks, no matter how those applications were originally implemented. Web services provide exactly this solution by providing a standardized method of communication between software applications. With a standardized method of communication in place, different applications can be integrated together in ways not possible before. Different applications can be made to call on each other's resources easily and reliably, and the different resources that applications already provide can be linked together to provide new sorts of resources and functionality.

Moreover, application integration becomes much more flexible because web services provide a form of communication that is not tied to any particular platform or programming language. The interior implementation of one application can change without changing the communication channels between it and the other applications with which it is coordinated. In short, web services provide a standard way to expose an application's resources to the outside world so that any user can draw on the resources of the application.

## What this Tutorial Teaches

In this tutorial you will learn the basic concepts behind building web services with WebLogic Workshop. You will create a web service that receives a name and returns a greeting. Specifically, you will learn how to build a synchronous method that will receive the name and return the greeting, and how to build an asynchronous, buffered method that will receive the name and invoke a buffered callback method to return a greeting.

Related Topics

Building Web Services

Getting Started: Web Services

## WebLogic Workshop Tutorials

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



# Getting Started: Web Service Core Concepts

This topic will familiarize you with the basic concepts of web service development.

## What is a Web Service?

A web service is a set of functions packaged into a single entity that is available to other systems on a network. The network can be a corporate intranet or the Internet. Other systems can call these functions to request data or perform an operation. Because they rely on basic, standard technologies which most systems provide, they are an excellent means for connecting distributed systems together.

## Standards-Based

Web services are built on standard technologies such as HTTP and XML. All web service messages are exchanged using a standard XML-messaging protocol known as SOAP, and web service interfaces are described using documents in the WSDL standard. These standards are all completely agnostic of the platform on which the web services were built. In short, the following standards are used:

- **WSDL.** A Web Service Description Language (WSDL) file describes how the web service is operated and how other software applications can interface with the web service. Think of a WSDL file as the instruction manual for a web service explaining how a user can draw on the resources provided by the web service.
- **XML.** Extensible Markup Language (XML) messages provide the common language by which different applications can talk to one another over a network. To operate a web service a user sends an XML message containing a request for the web service to perform some operation; in response the web service sends back another XML message containing the results of the operation.
- **SOAP.** Typically XML messages are formatted according to SOAP syntax. Simple Object Access Protocol (SOAP) specifies a standard format for applications to call each other's methods and pass data to one another. Note that other non-SOAP forms of XML messages are possible, depending on the specific requirements of the web service. (The type of XML message and the specific syntax required is given in the web service's WSDL file.)
- **HTTP & JMS.** To make web services accessible to other applications across networks, web services receive requests and send responses using widely used protocols such as HyperText Transfer Protocol (HTTP) and Java Message Service (JMS).

## Asynchrony

Many business processes take more than a few moments to complete; it could take a few minutes, even longer, or sometimes response time is variable and hard to predict. This is a problem for a synchronous relationship, in which a client invokes a method of the web service and is blocked from continuing its own processes until it receives the return value from the web service.

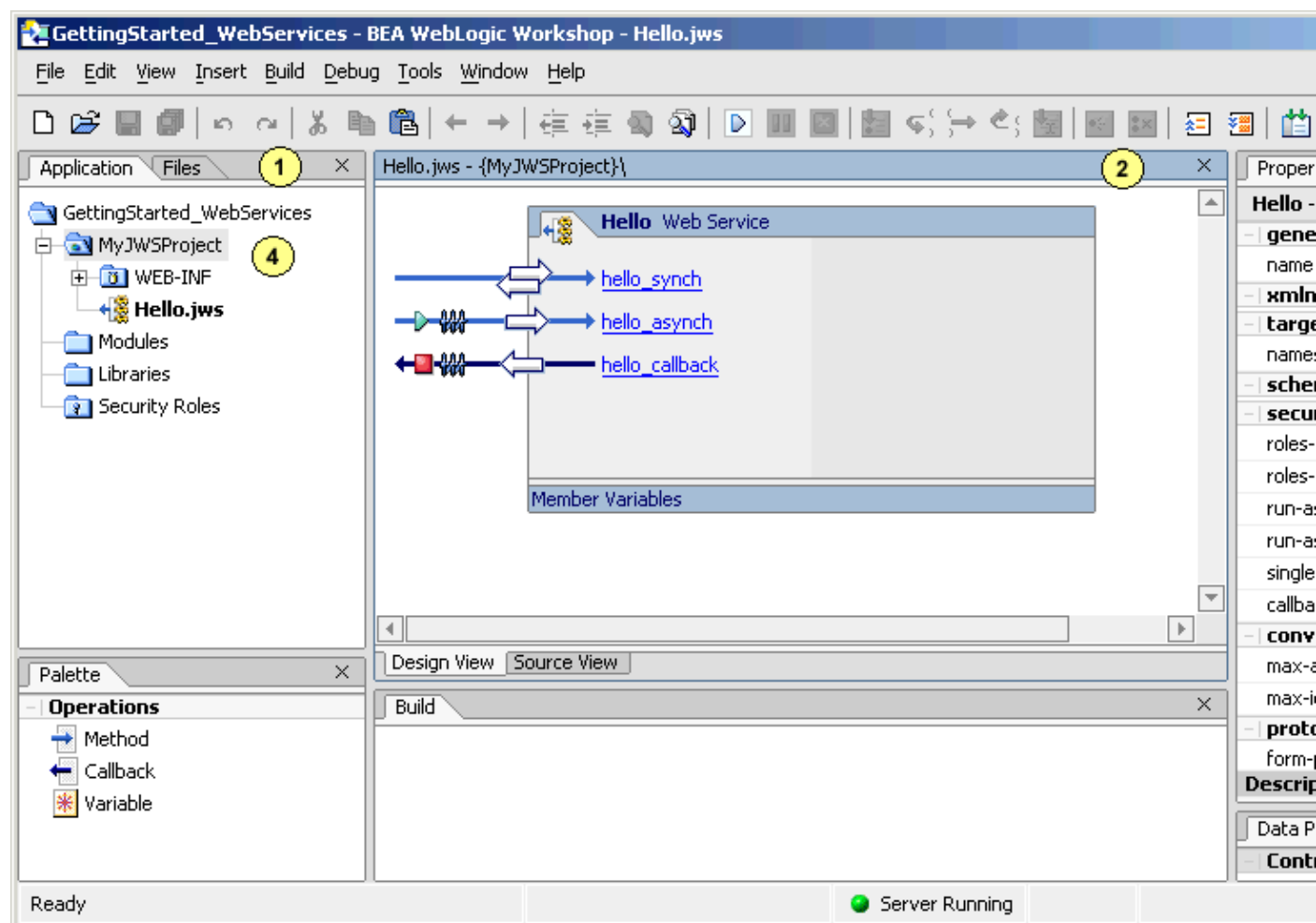
In WebLogic Workshop you can use asynchrony to overcome this problem. In asynchronous communication, the client communicates with the web service in such a way that it is not forced to halt its processes while the web service produces a response. In particular, the client invokes a method of the web service that immediately returns a simple acknowledgement to the client, thereby allowing the client to continue its own processes. When the web service has produced a response, it sends a callback method to the client containing that response. Asynchronous web services use conversations to correlate messages and manage state between message exchanges. This will ensure that the web service sends its response back to the correct client.

## Buffering

When a conversational web service's request method is called by multiple clients at the same time, a client might have to wait until its call, however short, has been processed by the web service. In addition, when a web service has completed processing multiple requests and needs to send multiple callback messages, it must wait for a message to be processed by the client before it can send the next callback. For a conversational web service, and especially for a web service receiving high-volume traffic, it is recommended to always add buffers to the web service's methods and callbacks. When clients call a buffered method, the call is stored in a buffer and the client does not have to wait for the web service to handle the call. When a web service sends a callback, the message is stored in a buffer and the web service does not have to wait until the client processes the callback.

## Designing Web Services with WebLogic Workshop

Web services are developed using WebLogic Workshop, a visual tool for designing J2EE applications. The image below shows the finished tutorial.



The **Application** tab (area 1) shows the application's source files. A web service is stored as a file with a JWS extension.

## WebLogic Workshop Tutorials

The main work area (area 2) shows the Design View or the Source view of the component you are building. In the above picture a web service is shown in Design View.

The ***Property Editor*** (area 3) allows you to see and set properties of a web service or web service method selected in the main area.

A web service project folder holds one or more web services used in the application. In this tutorial you will develop a web service in a web service project folder called MyJWSProject (area 4).

Related Topics

Getting Started with Web Services

Designing Asynchronous Interfaces

Using Buffering to Create Asynchronous Methods and Callbacks

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



# Step 1: Create a Web Service Project

In this step you will create the application that you will be working with throughout this tutorial. In WebLogic Workshop, an application contains one or more projects. For this application you will create a web service project to hold the web service you are going to build.

The tasks in this step are:

- To Start WebLogic Workshop
- To Create a New Application and Select a WebLogic Server Domain
- To Create a New Web Service Project
- To Start WebLogic Server

To Start WebLogic Workshop

If you have an instance of WebLogic Workshop already running, you can skip this step.

... on Microsoft Windows

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

- From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**WebLogic Workshop 8.1**.

...on Linux

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

- Open a file system browser or shell.
- Locate the **Workshop.sh** file at the following address:

```
$HOME/BEA/weblogic81/workshop/Workshop.sh
```

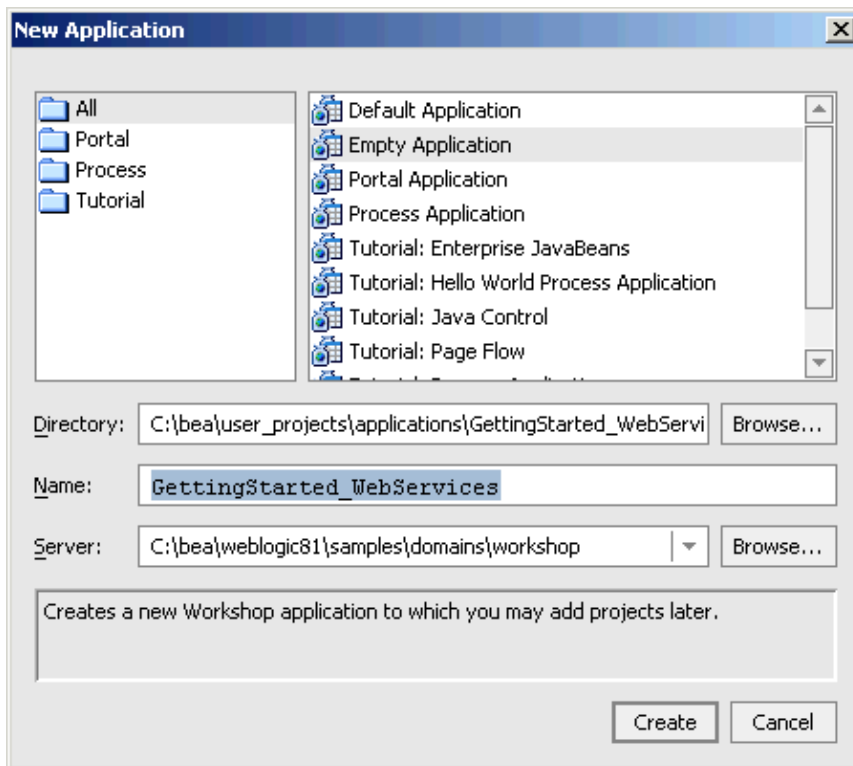
- In the command line, type the following command:

```
sh Workshop.sh
```

To Create a New Application and Select a WebLogic Server Domain

To create the web service project, you must first create the application to which it will belong:

1. From the **File** menu, select **New**—>**Application**. The **New Application** dialog appears.
2. In the **New Application** dialog, in the upper left-hand pane, select **All**.  
In the upper right-hand pane, select **Empty Application**.  
In the **Directory** field, use the **Browse** button to select a location to save your source files. A suggested location is BEA\_HOME\user\_projects\applications\GettingStarted\_WebServices.  
In the **Name** field, enter GettingStarted\_WebServices.  
In the **Server** field, from the drop-down list, select BEA\_HOME\weblogic81\samples\domains\workshop.

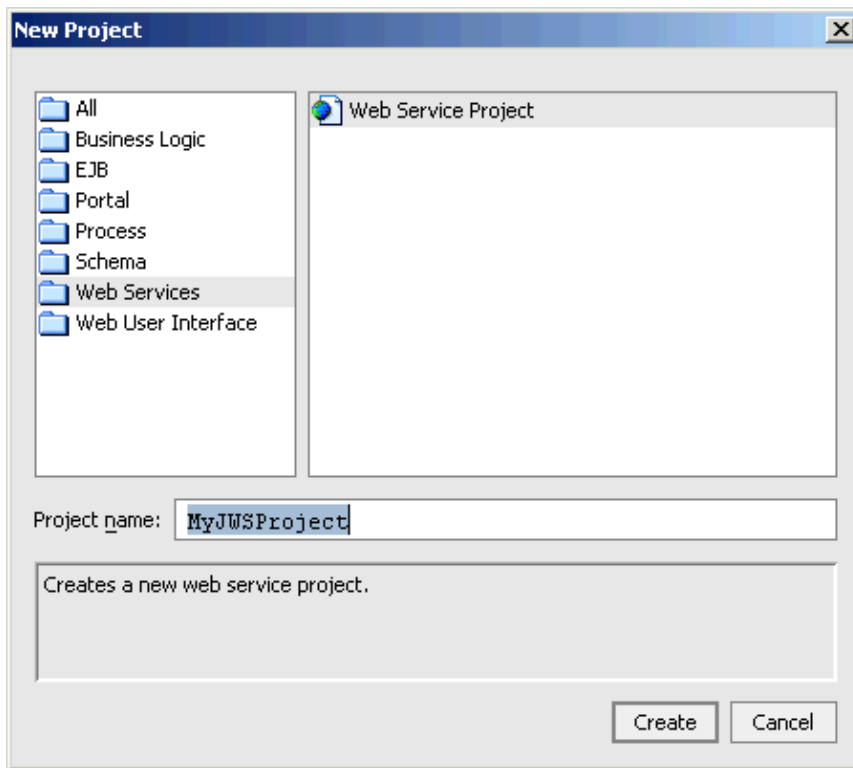


3. Click **Create**.

To Create a New Web Service Project

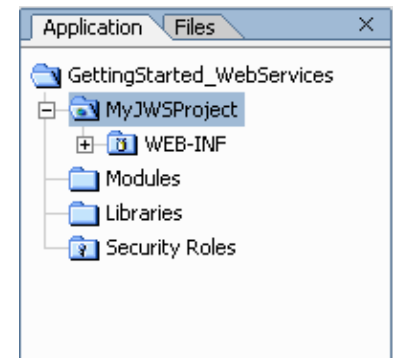
1. In the **Application** tab, right-click the GettingStarted\_webServices folder and select **New-->Project**.
2. In the **New Project** dialog, in the upper left-hand pane, confirm that **Web Services** is selected.  
In the upper right-hand pane, select **Web Service Project**.  
In the **Project Name** field, enter MyJWSProject.





3. Click **Create**.

The **Application Tab** should now look like the picture shown to the right.



The top-level folder *GettingStarted\_WebServices* is the containing folder for the entire application. All of the source files for your application exist in this folder.

The folder *MyJWSProject* is a project folder. An application can contain any number of project folders and there are many different kinds of project folders, including Web projects, Web Services projects, Schema projects, and so forth. *MyJWSProject* is a web services project. A web service project always contains the **WEB-INF** folder. This folder stores JSP tag libraries, configuration files, compiled class files, and other runtime resources.

The **Modules** folder is for the storage of stand-alone applications (packaged as WAR and JAR files) that can be deployed parallel with your web service, if you wish.

The **Libraries** folder is for the storage of resources that are to be used across multiple projects, provided the resources are packaged as JAR files. For example, if you had a control or an EJB that you wanted to re-use across all your projects, you would store it in the Libraries folder.

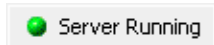
## WebLogic Workshop Tutorials

The **Security Roles** folder lets you define security roles and test users for your web application. You can test the security design of your web application by logging in as a test user.

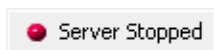
### To Start WebLogic Server

Since you will be deploying and running your Java control on WebLogic Server, it is helpful to have WebLogic Server running during the development process.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.

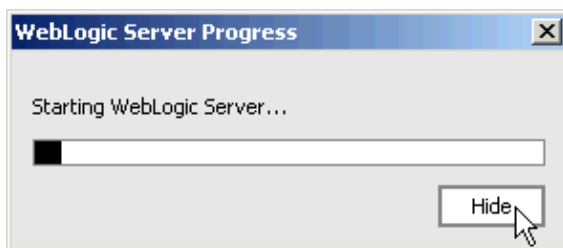


If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow these instructions to start WebLogic Server:

- From the **Tools** menu, choose **WebLogic Server**—>**Start WebLogic Server**.
- On the **WebLogic Server Progress** dialog, you may click **Hide** and continue to the next task.



### Related Topics

[Getting Started with Web Services](#)

[How Do I...? Web Service Topics](#)

[The WebLogic Workshop Development Environment](#)

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



## Step 2: Implement Synchronous Communication

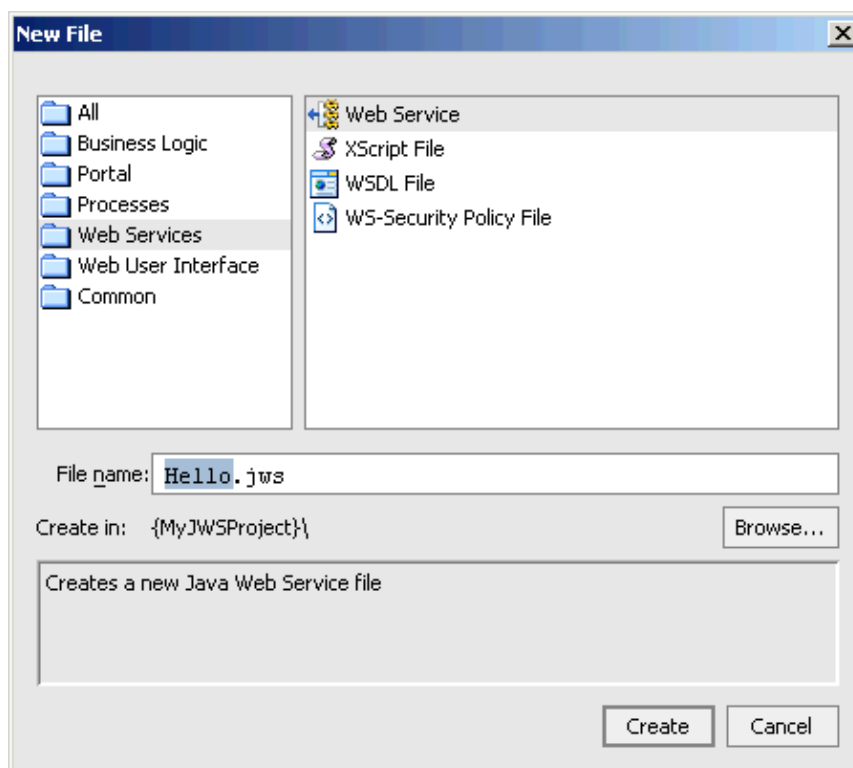
In this step you will create the web service and you will add a synchronous method to this web service. Remember that when a client invokes a synchronous method, it is blocked from continuing its own processes until it receives the return value from the web service.

The tasks in this step are:

- Create the Web Service
- Add a Method
- Test the Web Service

### Create the Web Service

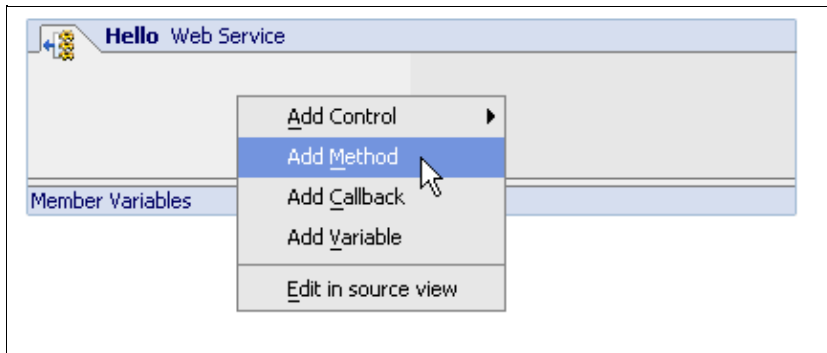
1. Right-click *MyJWSProject* in the *Application* tab, and select *New-->Web Service*.
2. In the *New File* dialog, ensure that Web Service is selected and enter the file name *Hello.jws*.



3. Click *Create*.

### Add a Method

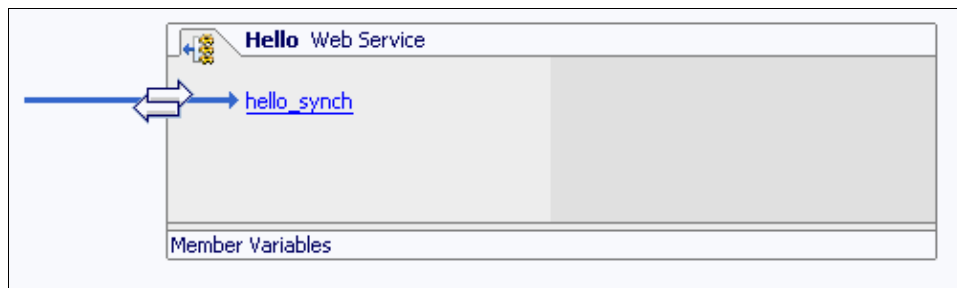
1. Right-click the web service in the main area and select *Add Method*.



2. The method `newMethod1` appears. Rename this method `hello_sync`. If you step off the method before you have finished renaming it, right-click the method and select **Rename**.
3. Click the `hello_sync` method to go to Source View and modify the method as shown in red below:

```
/**
 * @common:operation
 */
public String hello_sync(String name)
{
    return "Hello, " + name + "!";
}
```

You have now created a web service with the method `hello_sync`. This method is considered synchronous because when the method is invoked, the invoking client has to wait until the method has finished processing and returns a `String` result. A synchronous method that returns a result to the invoking client is depicted in Design View as a straight blue arrow pointing to the right, with white arrows in both directions on top of the blue arrow.



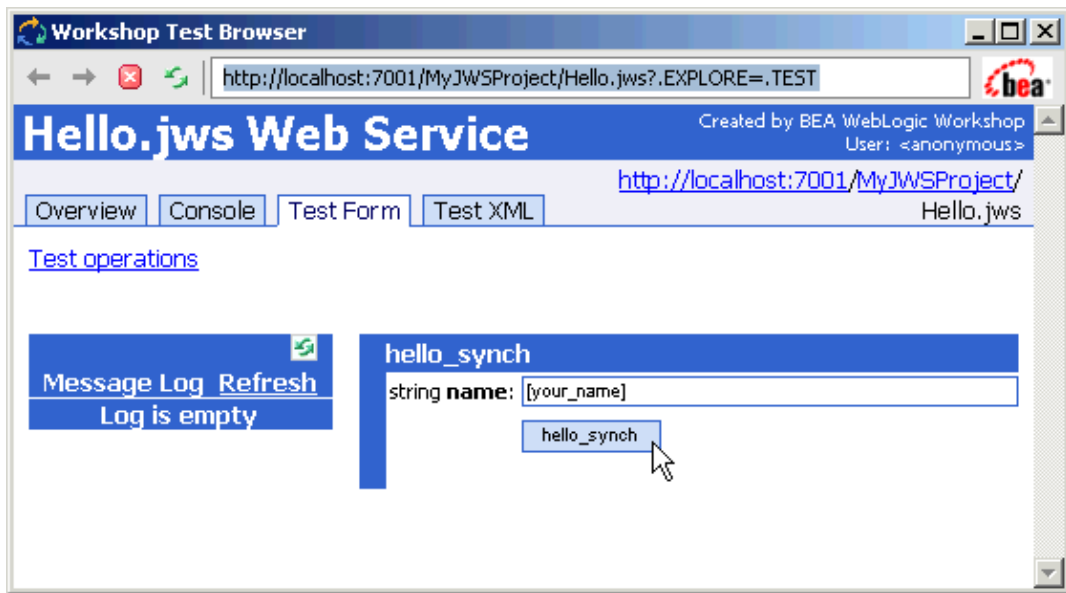
## Test the Web Service

Next you will test the web service to examine its behavior.

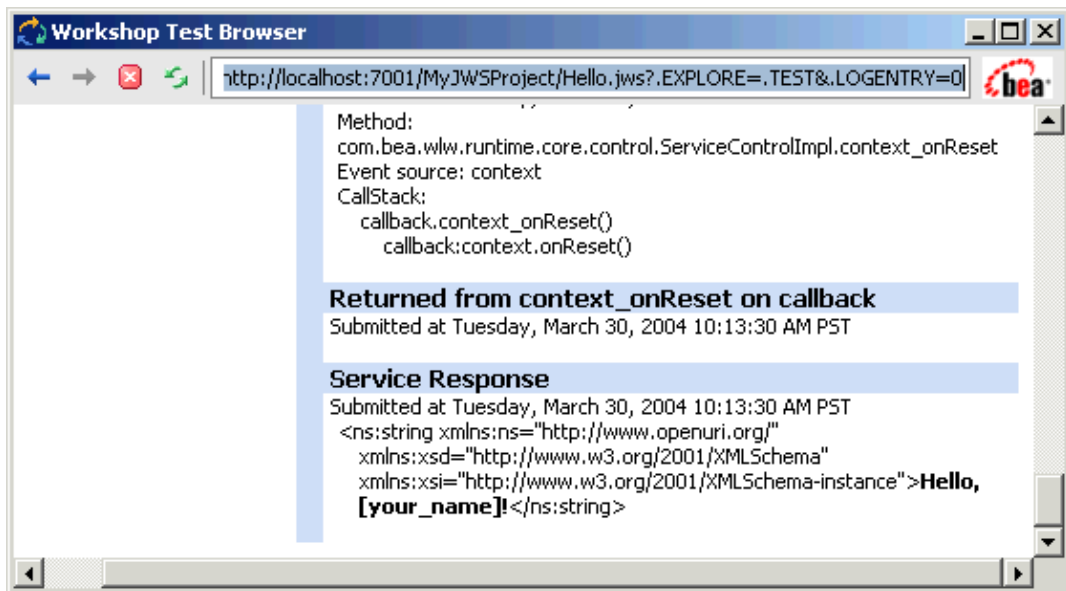
1. Click the **Start** button, shown below:



2. When the **Workshop Test Browser** launches, in the **Name** field, enter `[your_name]` and click **hello**.



3. Examine the result returned by the `hello_synch` method. Scroll down to the **Service Response** section and notice that the method returned *Hello, [your\_name]!*



4. Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



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



## Step 3: Implement Asynchronous Communication

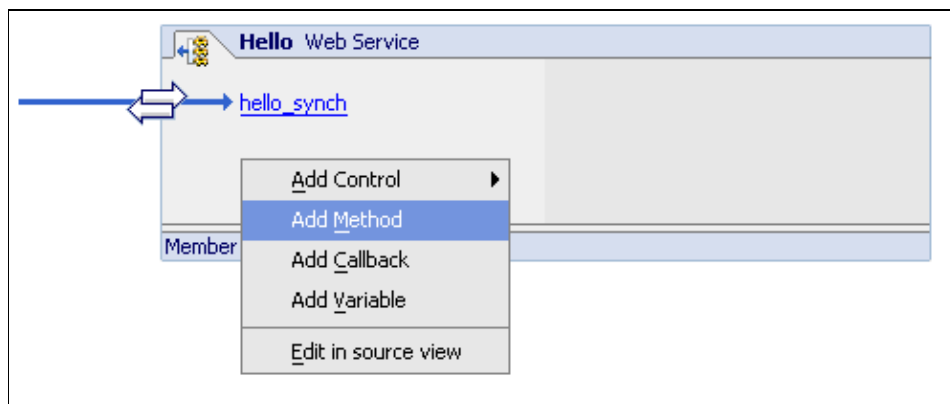
In this step you will enable asynchronous communication for the web service. You will add an asynchronous method that immediately returns a simple acknowledgement to the client, thereby allowing the client to continue its own processes. You will also add a callback method that returns the response to the client. Finally, you will add buffers to the method and the callback to further minimize the time the client or web service has to wait while the method invoked or callback returned is processed by the receiving party.

The tasks in this step are:

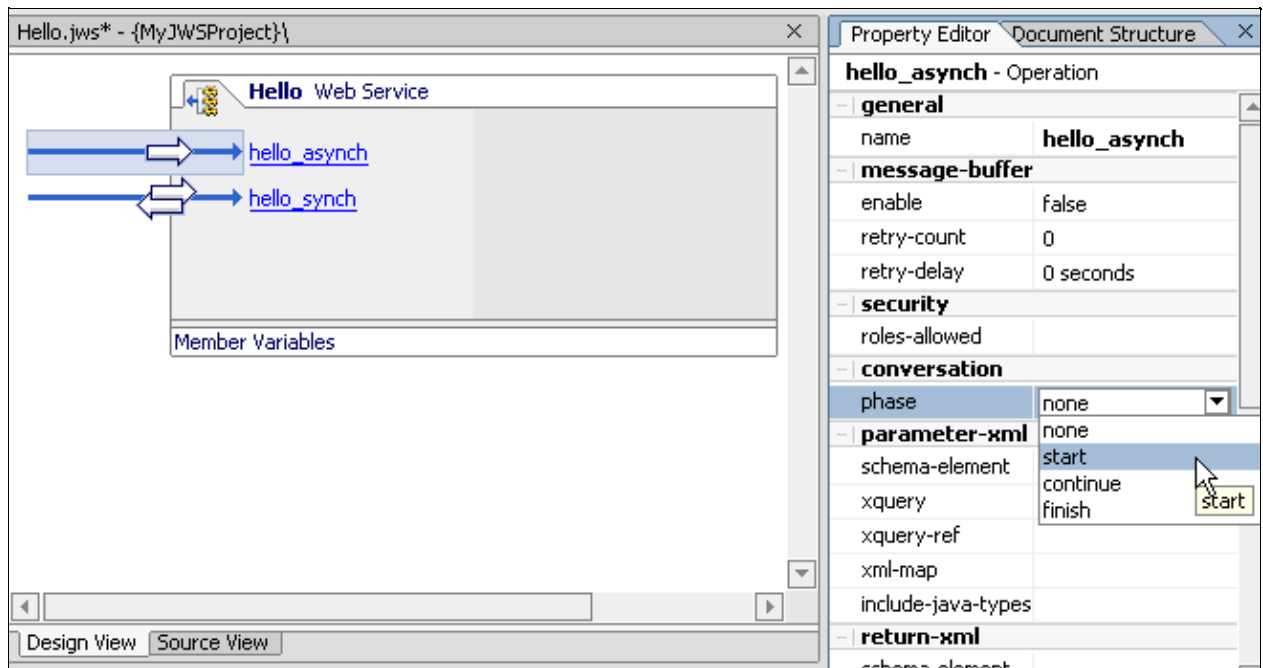
- Add a Method
- Add a Callback
- Edit the Method
- Add Buffers
- Test the Web Service

### Add a Method

1. Right-click the web service in the main area and select **Add Method**.



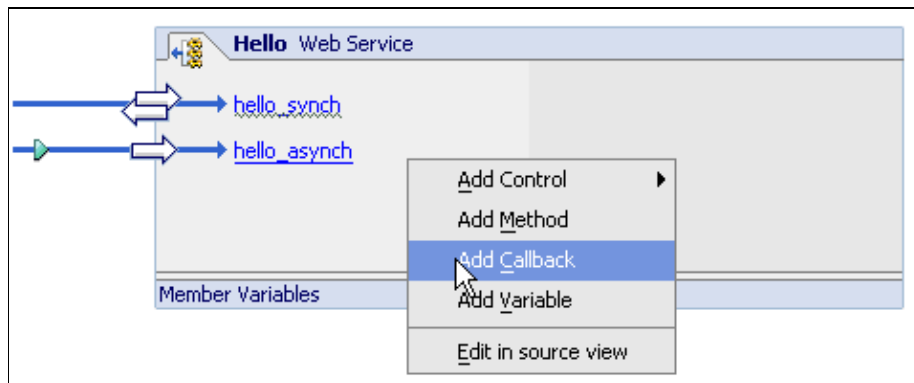
2. The method newMethod1 appears. Rename this method hello\_async. If you step off the method before you have finished renaming it, right-click the method and select **Rename**.
3. Make sure that hello\_async is still selected and go to the **Property Editor**. Locate the conversation section and for the phase attribute, select **start**.



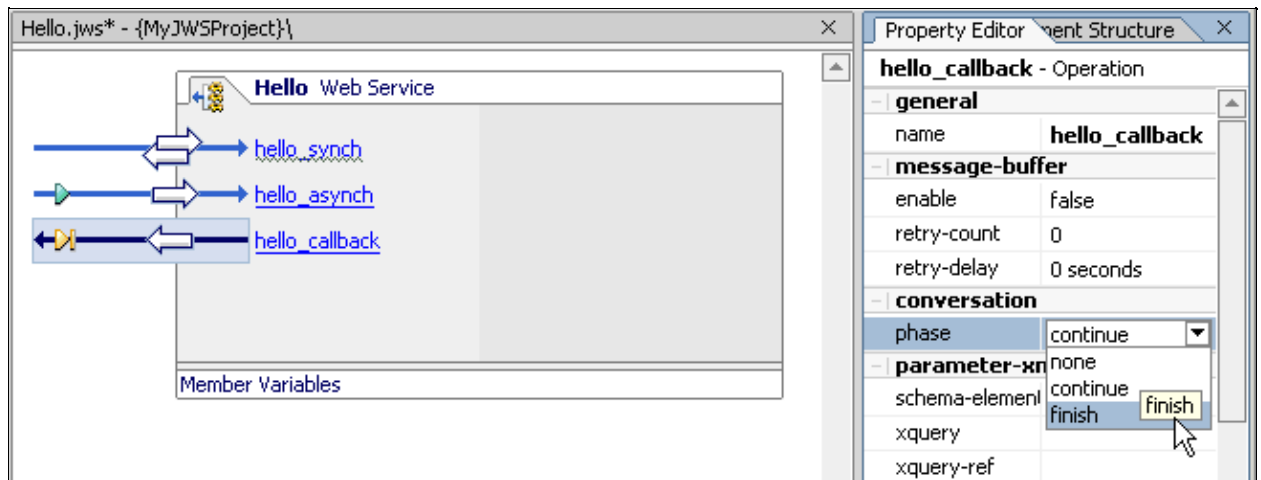
A green start icon will appear on the blue arrow in front of the hello\_async method.

## Add a Callback

1. Right-click the web service in the main area and select **Add Callback**.



2. The method newCallback1 appears. Rename this method hello\_callback. If you step off the method before you have finished renaming it, right-click the method and select **Rename**.
3. Make sure that hello\_callback is still selected and go to the **Property Editor**. Locate the conversation section and for the phase attribute, select **finish**.

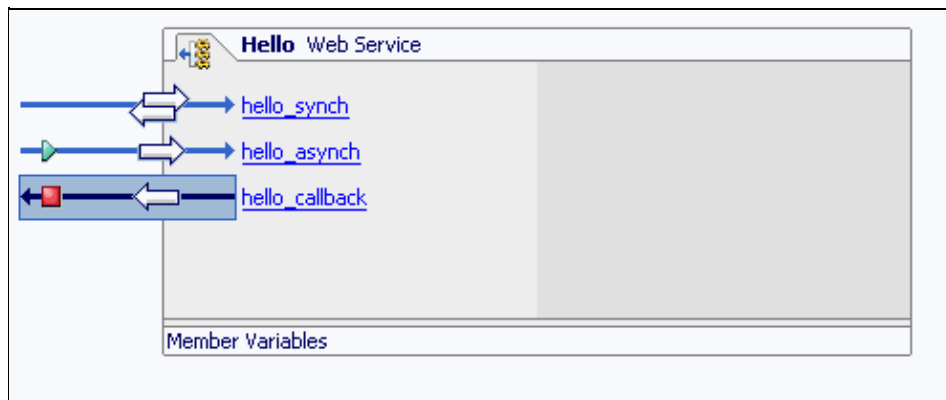


A red stop icon will appear on the blue arrow in front of the `hello_async` method.

- Click `hello_callback` to go to Source View and modify the method as shown in red below:

```
public interface Callback extends com.bea.control.ServiceControl
{
    /**
     * @jws:conversation phase="finish"
     */
    void hello_callback(String message);
}
```

In Design View the web service should now look like this:



## Edit the Method

Now you will edit the `hello_async` method to invoke the `hello_callback` callback.

- In Design View, click `hello_async` to go to Source View and modify the method as shown in red below:

```
/**
 * @common:operation
 * @jws:conversation phase="start"
 */
public void hello_async(String name)
{
    String greeting;
```



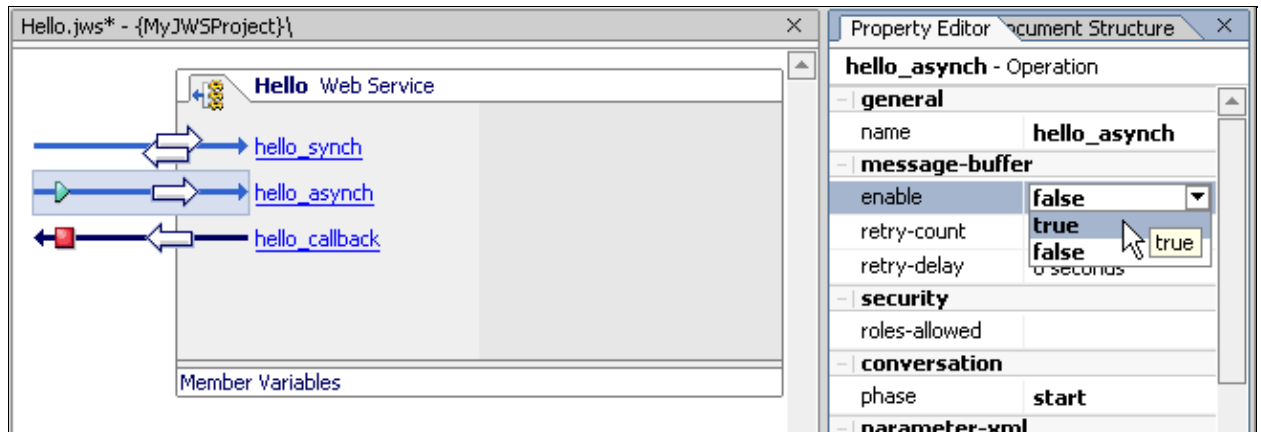
```

greeting = "Hello, " + name + "!";
this.callback.hello_callback(greeting);
}

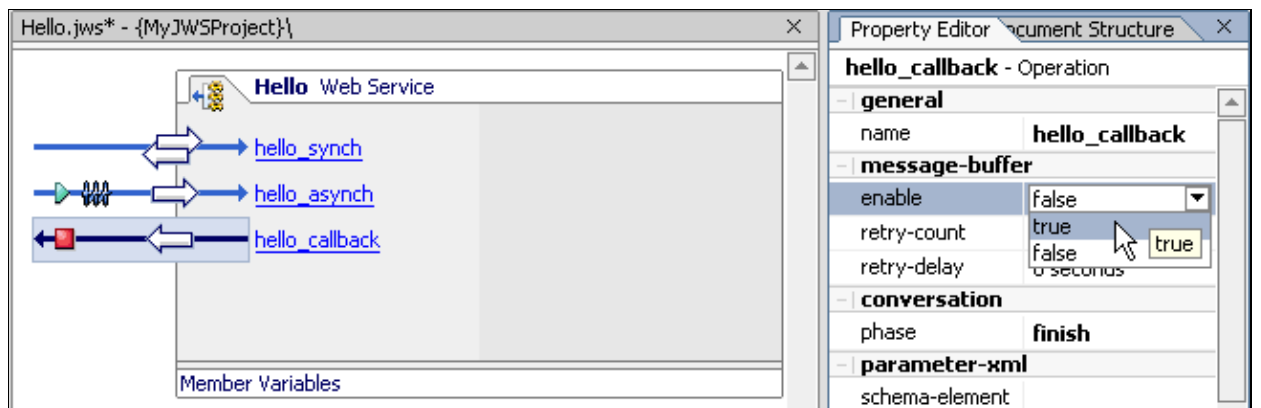
```

## Add Buffers

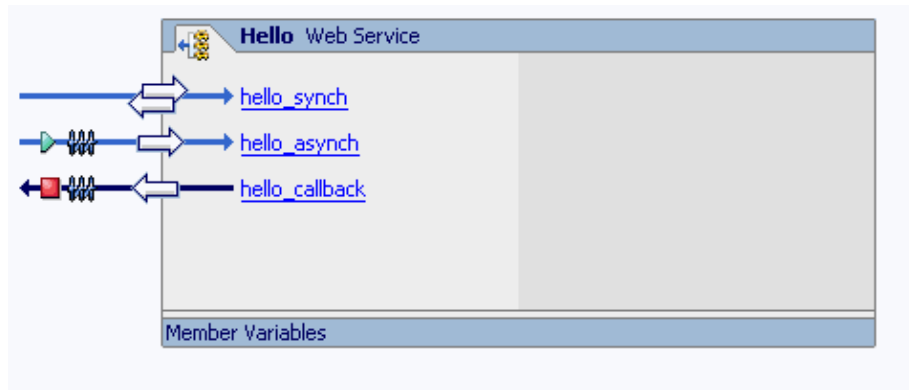
1. Click the Design View tab.
2. In Design View, select the `hello_async` method by selecting the blue arrow in front of the method.
3. In the **Property Editor**, locate the *message-buffer* section and set the `enable` attribute to `true`. Notice that a buffer icon appears on the blue arrow.



4. In Design View, select the `hello_callback` method by selecting the blue arrow in front of the method.
5. In the **Property Editor**, locate the *message-buffer* section and set the `enable` attribute to `true`. Notice that a buffer icon appears on the blue arrow.



The completed web service should look like this:



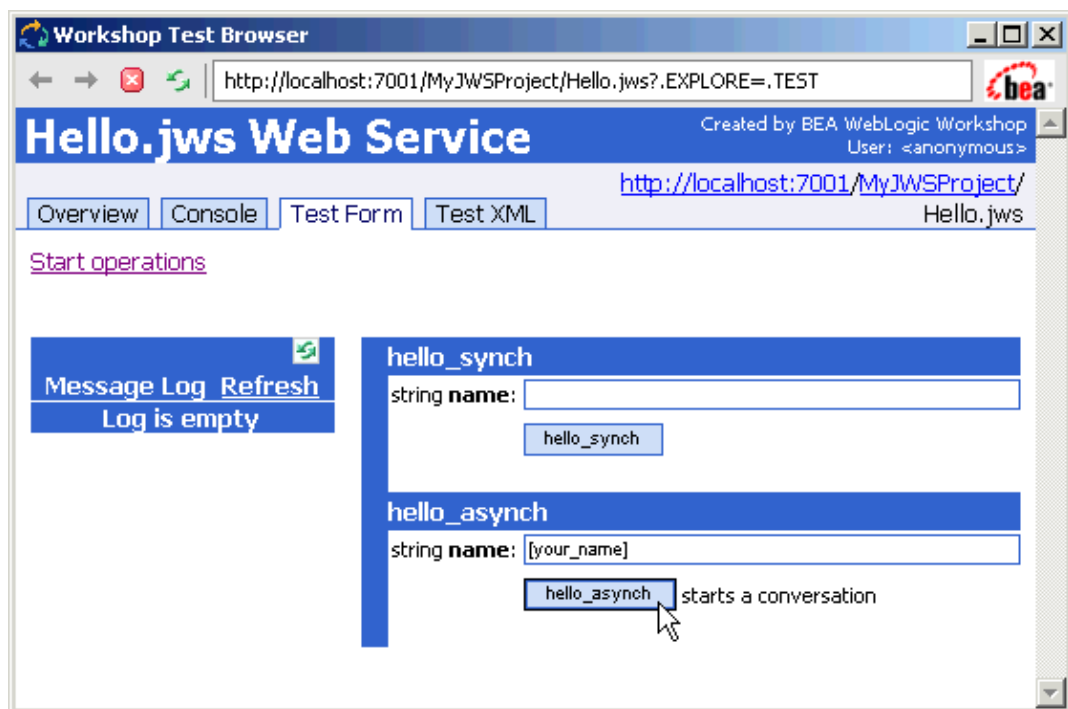
## Test the Web Service

Next you will test the web service to examine its behavior.

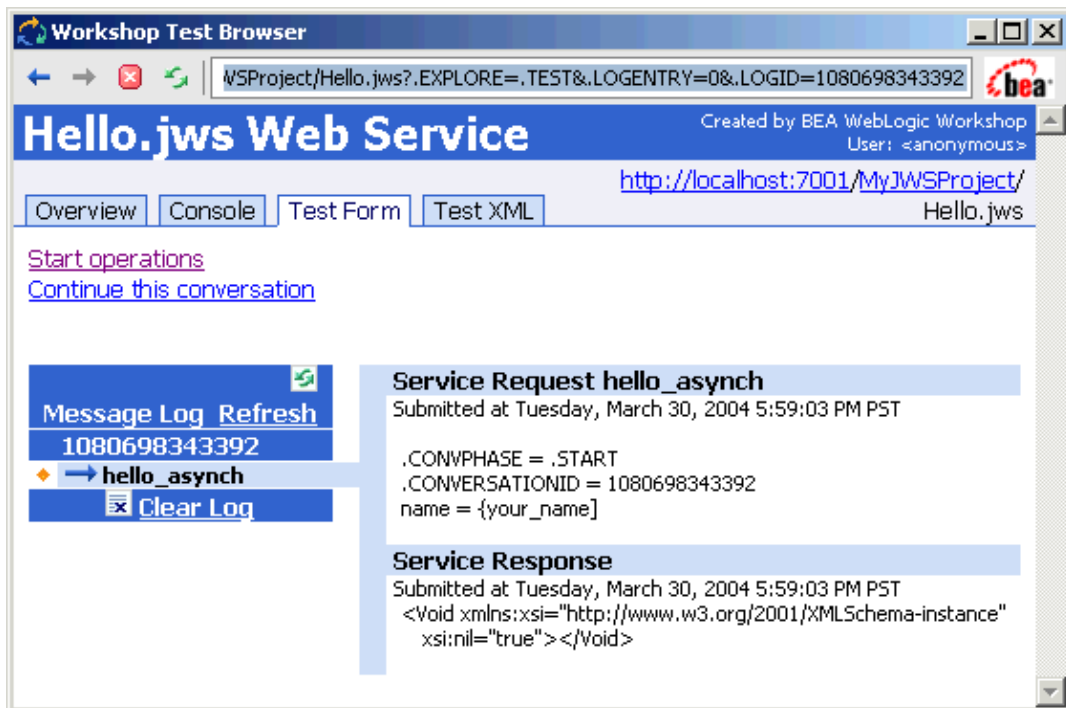
1. Click the **Start** button, shown below:



2. When the **Workshop Test Browser** launches, in the **Name** field, enter [your\_name] and click **hello\_async**.



3. Notice that the method returns immediately. Scroll down to the **Service Response** section and notice that the method did not return anything. Also notice that the conversation has started.



- Click the **Continue this conversation** link. Notice in the **Message Log** that the the hello\_callback method has been invoked and that the conversation is finished. Also notice that no greeting is returned as a return value of the method; instead the callback method sends the greeting as a parameter to the invoking client.



- Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



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



# Summary: Getting Started Web Services Tutorial

This tutorial introduced you to the basics of building web services with WebLogic Workshop.

## Concepts and Tasks Introduced in This Tutorial

- Using a web service to provide standards-based communication
- Using asynchrony and buffering to prevent the invoking client from having to halt its process

Now that you have finished the basic web service tutorial, you can continue with one of the other basic tutorials to learn more about the other core components you can develop in WebLogic Workshop. For more information, see The Getting Started Tutorials. If you want to learn more about web service development, go to the advanced Tutorial: Web Services. You can learn more about the web service design at Building Web Services.

### Related Topics

The Getting Started Tutorials

Tutorial: Web Services

Building Web Services

Designing Asynchronous Interfaces

Using Buffering to Create Asynchronous Methods and Callbacks

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



# Getting Started: Web Applications

## The Problem with 'Old-Style' Web Applications

Many web application developers are constantly faced with the problem of successfully making changes to a web site. These developers use web application technology that doesn't allow an easy separation of the user interface and the logic to interpret user input. Their web applications use CFM, ASP, or JSP pages in which HTML code, JavaScript, and server-side code are all embedded in the same page. Because the server-side logic needed to process a web page is stored directly in the page, similar logic is often reduplicated throughout the various web pages instead of being coded once in a central location. As a result, if you need to change the way data is interpreted, you need to make these changes in all the web pages that implement that logic.

This problem is compounded by the fact that this 'old-style' web application technology does not come with an easy way to get a visual overview of how the various web pages relate to each other. A web developer either needs to rely on his/her own memory to remember the flow through a web site or needs to use a mapping tool to take a snapshot of the relations between the pages in a web application. Because a snapshot is a static reflection, it is outdated as soon as changes to the web application have been made.

## The Page Flow Solution

WebLogic Workshop uses Page Flows that make it easy to separate the user interface from the navigation and business logic. A page flow consists of JSP pages that contain the user interface elements and a controller (JPF) file that contains instructions on how data provided by a user is processed and what page will be returned to the user next. A page flow provides a visual overview of the web application that enables you to see how the various JSP pages relate to each other and allows you quickly build the overall architecture of the web application.

## What this Tutorial Teaches

In this tutorial you will learn the basic concepts behind building web applications with WebLogic Workshop. You will create a page flow that allows a user to enter his/her name and receive a greeting. Specifically, you will learn how to (1) build a JSP page on which a user can enter input, (2) build a controller file that receives this data, processes it, and forwards a results to another JSP page, and (3) build a second JSP page that displays data.

Related Topics

Developing Web Applications

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



# Getting Started: Web Application Core Concepts

This topic will familiarize you with the basic concepts of page flow design and development.

## What is a Page Flow?

A page flow application is a web application with a special architecture. It contains the following elements.

### JSP Files

The JSP files contains the presentation elements of a page flow applications. In other words, it contains the user interface displayed in a browser. The web application can have as many JSP pages as necessary, and each JSP page can contain all the elements of any regular JSP page. A JSP page in a page flow typically contains HTML elements and netui tags, which are custom tags provided in WebLogic Workshop.

### Controller Files

The controller file contains the business logic and the navigation logic of a web application. You can have exactly one controller file for your web application, or you can have many controller files with each controller file implementing the business and navigation logic of a part of the web application. A controller file typically contains numerous *Action Methods*. When a user sends a request to the server, for instance by hitting a **Submit** button on a web page, a specific action method contains the business logic describing how to process this user request. The action method also contains the navigation logic describing what JSP page should be returned to the user next. A controller file can be read in a special graphical view called the *Flow View* that shows the navigational flow through the web application.

### Data Binding

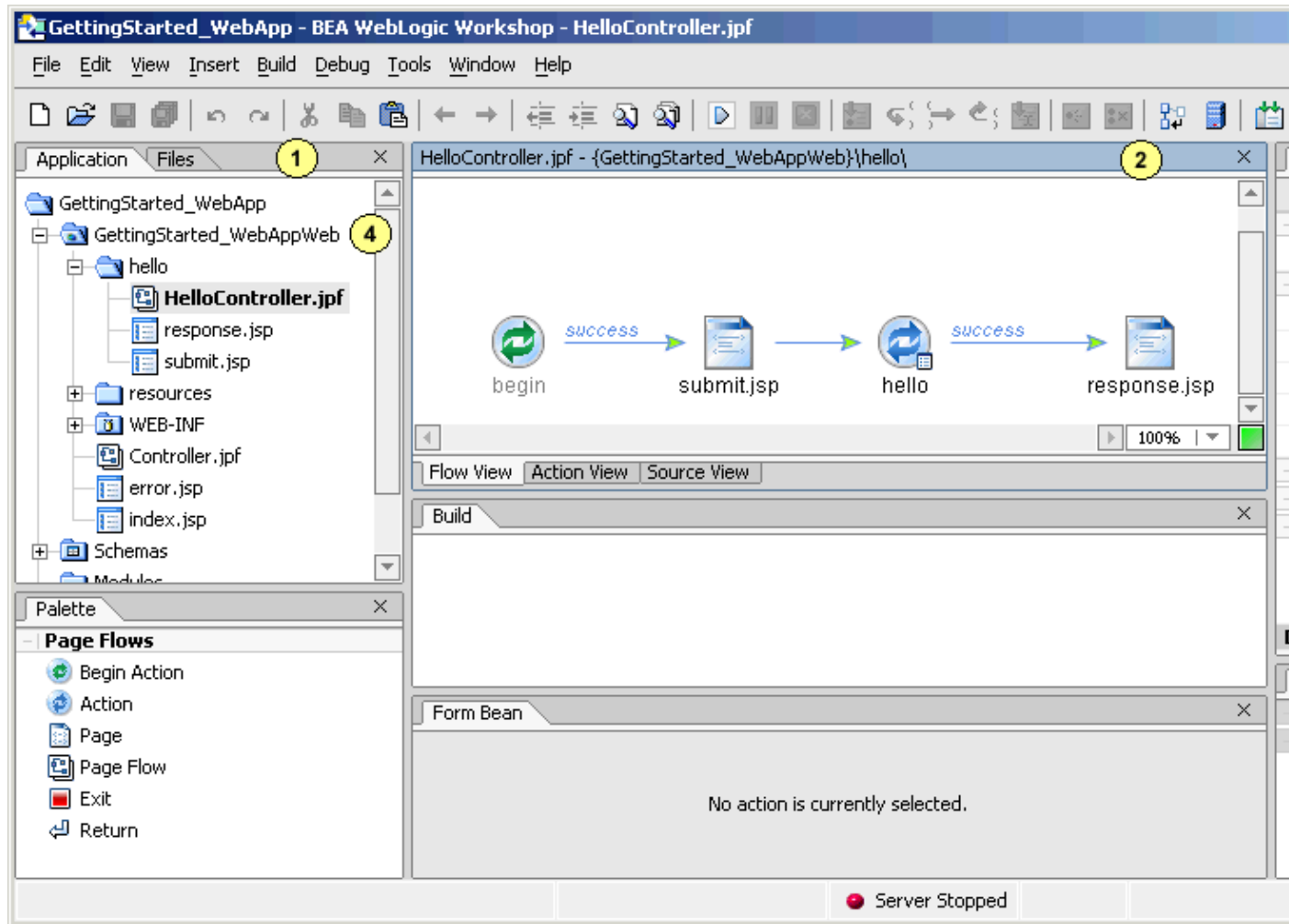
A user request typically contains data that the user provided by filling in fields on a JSP page. When a user submits this data, for instance his name and age, it must be properly received by the controller file such that it is unequivocally clear what data corresponds to his name and what data corresponds to his age. The process of properly tying user input entered in a field to a certain property is called data binding.

Conversely, a JSP page often needs to display data. This data is not static but might be read from a database, received earlier from a user, or computed on the basis of other dynamic data. Again, data binding must be done correctly such that a certain presentation element on a JSP page displays the correct data.

There are various methods to accomplish data binding. User input is typically handled using a form bean, which contains a set of properties corresponding to the data entered in the fields on a page. For instance, a form bean might have a String name property to store the name, an int age property to store the age, and so forth. To display data, you can use a form bean or you can add parameters to a URL, to name just two examples. In this tutorial you will make the data to be displayed an attribute of a request object, which is a special object readable by a JSP page.

## Designing Page Flows with WebLogic Workshop

Page flows are developed using WebLogic Workshop, a visual tool for designing J2EE applications. The image below shows the finished tutorial.



The **Application** tab (area 1) shows the application's source files.

The main work area (area 2) shows several views of the page flow components you are building. In the above picture a controller (JPF) file is shown in Flow View. The Flow View shows the navigational flow of the web application.

The **Property Editor** (area 3) allows you to see and set properties of the page flow component opened in the main area.

A web project folder holds all the page flow components required to run a web application. In this tutorial you will develop the page flow application in a web project folder called GettingStarted\_WebApp (area 4).

## Related Topics

### Guide to Building Page Flows

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





# Step 1: Create a Web Application Project

In this step you will create the application that you will be working with throughout this tutorial. In WebLogic Workshop, an application contains one or more projects that hold the various components of the application. In this tutorial you will create a web project for the page flow.

The tasks in this step are:

- To Start WebLogic Workshop
- To Create a Web Application
- To Start WebLogic Server

## To Start WebLogic Workshop

### ... on Microsoft Windows

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

1. From the *Start* menu, choose *Programs*—>*WebLogic Platform 8.1*—>*QuickStart*.
2. In the *QuickStart* dialog, click *Experience WebLogic Workshop 8.1*.

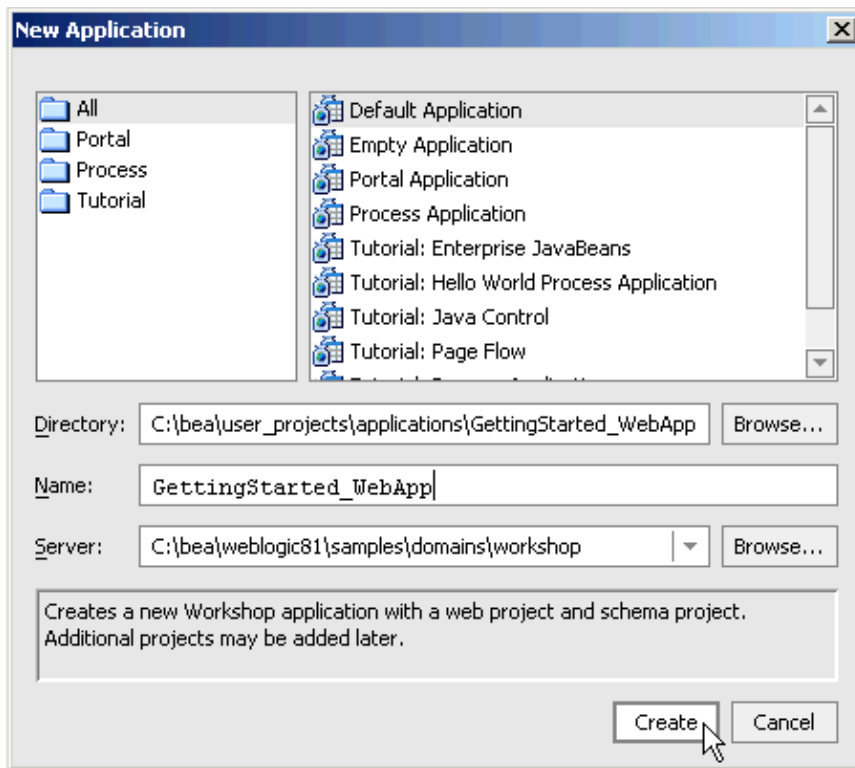
### ...on Linux

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

1. Open a file system browser or shell.
2. Locate the **Workshop.sh** file at the following address:  
\$HOME/bea/weblogic81/workshop/Workshop.sh
3. In the command line, type the following command:  
sh Workshop.sh

## To Create a Web Application

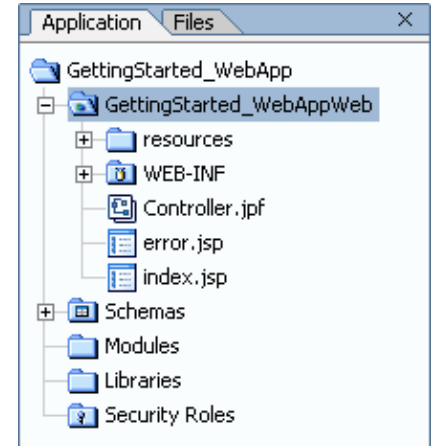
1. From the *File* menu, select *New*—>*Application*.
2. In the *New Application* dialog, in the upper left-hand pane, select *All*,  
in the upper left-hand pane, select *Default Application*,  
in the *Name* field, enter GettingStarted\_WebApp,  
in the *Directory* field, use the *Browse* button to select a location to save your source files. A suggested location is BEA\_HOME\user\_projects\applications\GettingStarted\_WebApp is selected,  
in the *Server* field, select BEA\_HOME\weblogic81\samples\domains\workshop.



3. Click **Create**.

When you create a new Default Application, Workshop creates the application structure shown to the right.

The top-level folder *GettingStarted\_WebApp* is the containing folder for the entire application. All of the source files for your application exist in this folder.



The folder *GettingStarted\_WebAppWeb* is a project folder. An application can contain any number of project folders and there are many different kinds of project folders, including Web projects, Web Services projects, Schema projects, and so forth. *GettingStarted\_WebAppWeb* is a Web project, created by default when you created the default application in the last dialog. Web projects contain the following common resources for web applications:

- The **resources** folder stores commonly reused JSP elements, including CSS files and JSP templates.
- The **WEB-INF** folder stores JSP tag libraries, configuration files, compiled class files, and other runtime resources.
- The **Controller.jspf**, **error.jsp**, and **index.jsp** are the components of the default parent page flow. You may use this page flow as the master error handler and/or the master navigational controller for other

page flows in your portal.

The **Modules** folder is for the storage of stand-alone applications (packaged as WAR and JAR files) that can be deployed parallel with your web application, if you wish.

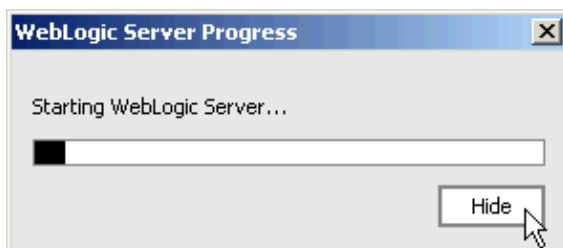
The **Libraries** folder is for the storage of resources that are to be used across multiple projects, provided the resources are packaged as JAR files. For example, if you had a control or an EJB that you wanted to re-use across all your projects, you would store it in the Libraries folder.

The **Security Roles** folder lets you define security roles and test users for your web application. You can test the security design of your web application by logging in as a test user.

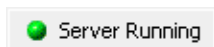
## To Start WebLogic Server

In order to run and test your page flow application, it must first be deployed on WebLogic Server. For convenience you will start WebLogic Server now, so you can test your page flow as you design it.

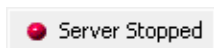
1. From the **Tools** menu, select **WebLogic Server**—>**Start WebLogic Server**.
2. On the **Start up Progress** dialog, you may click **Hide** and continue to the next task.



You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed.



If WebLogic Server is not running, a red ball is displayed.



You are now ready to begin designing your web application.

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



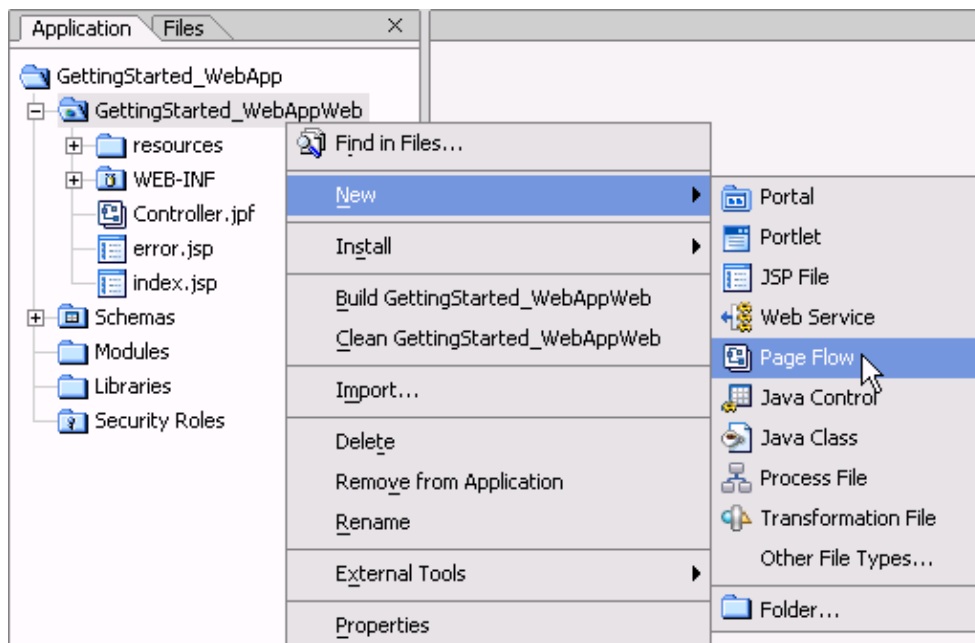
## Step 2: Submitting Data

In this step you will set up the basic page flow application, create the JSP page on which users will enter their name, and create the form bean to hold the user input. The tasks in this step are:

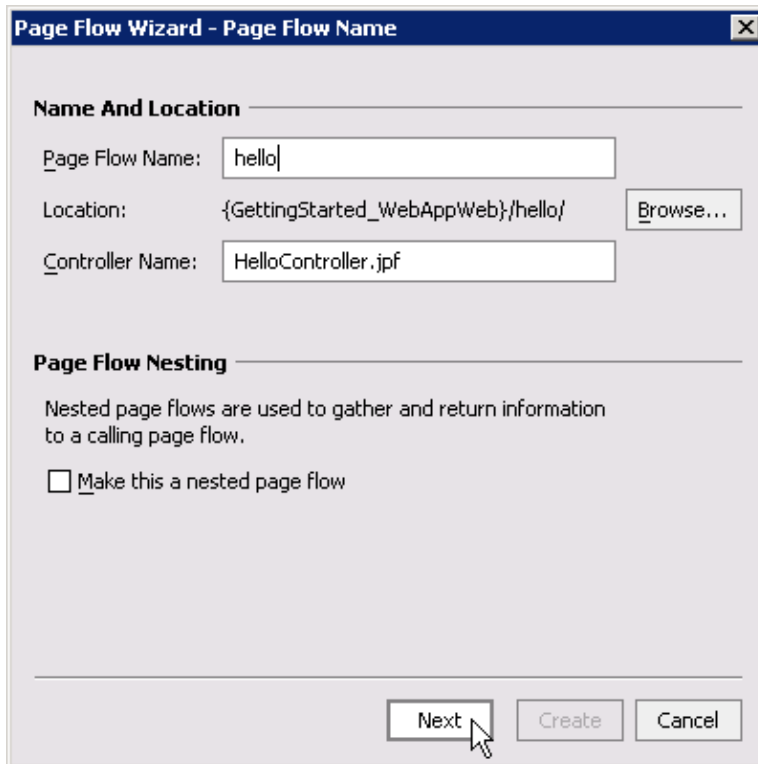
- Create a Page Flow
- Make a Submission Page, a Form Bean, and a Method
- Test the Web Application

### Create a Page Flow

1. On the *Application* tab, right-click *GettingStarted\_WebAppWeb* folder and select *New*—>*Page Flow*.



2. In the *Page Flow Wizard–Page Flow Name* dialog, in the *Page Flow Name* field, enter hello. Click *Next*.



**Page Flow Wizard - Page Flow Name**

**Name And Location**

Page Flow Name:

Location:

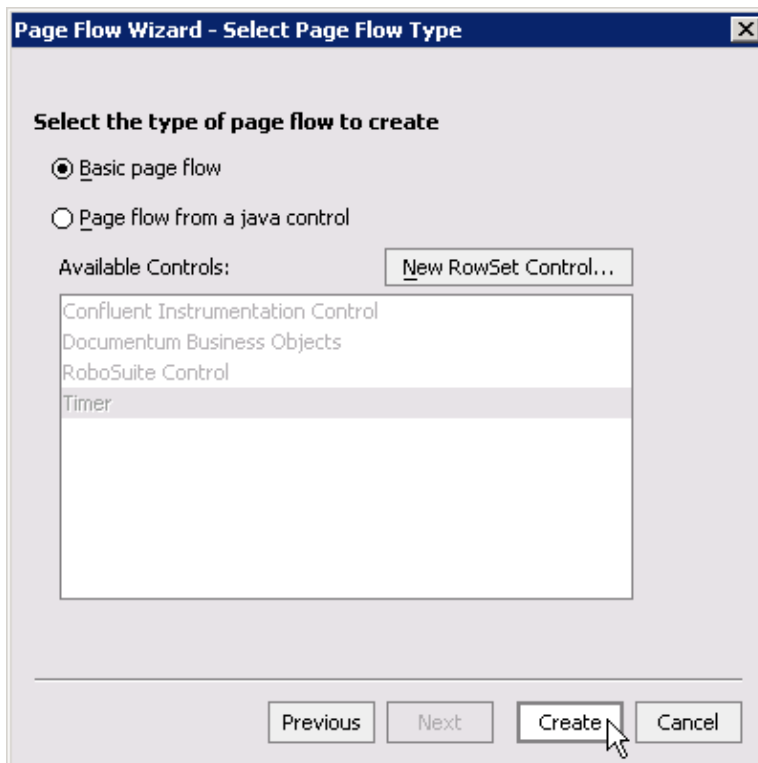
Controller Name:

**Page Flow Nesting**

Nested page flows are used to gather and return information to a calling page flow.

☐ Make this a nested page flow

3. On the *Page Flow Wizard–Select Page Flow Type* dialog, click *Create*.



**Page Flow Wizard - Select Page Flow Type**

**Select the type of page flow to create**

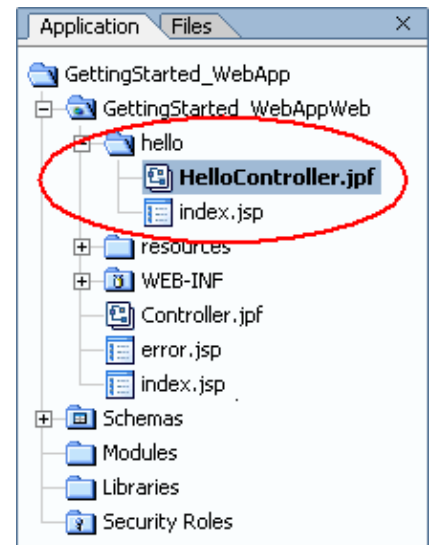
☒ Basic page flow

☐ Page flow from a java control

Available Controls:

- Confluent Instrumentation Control
- Documentum Business Objects
- RoboSuite Control
- Timer

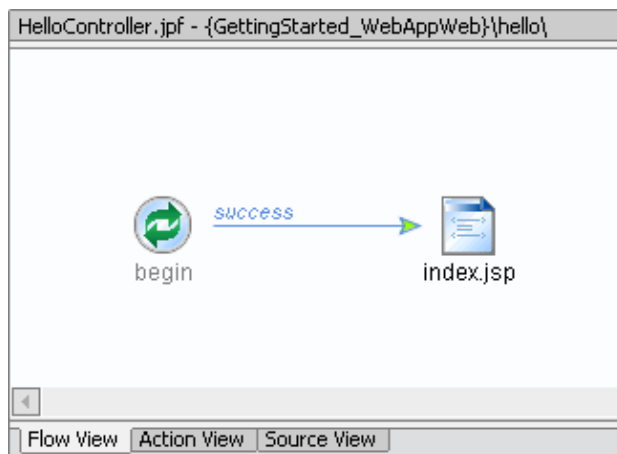
The new page flow that is created resides in a folder named hello and consists of a JSP file named index.jsp and a Controller file named HelloController.jpf. The JSP page is the default home page of the Page Flow. The Controller file contains Java methods for processing data and for forwarding users to different JSP pages.



This controller file is opened in the main area in Flow View as is shown below. The Flow View is a pictorial view of the Controller file, which shows the relationships between the JSP pages and the methods of the page flow.

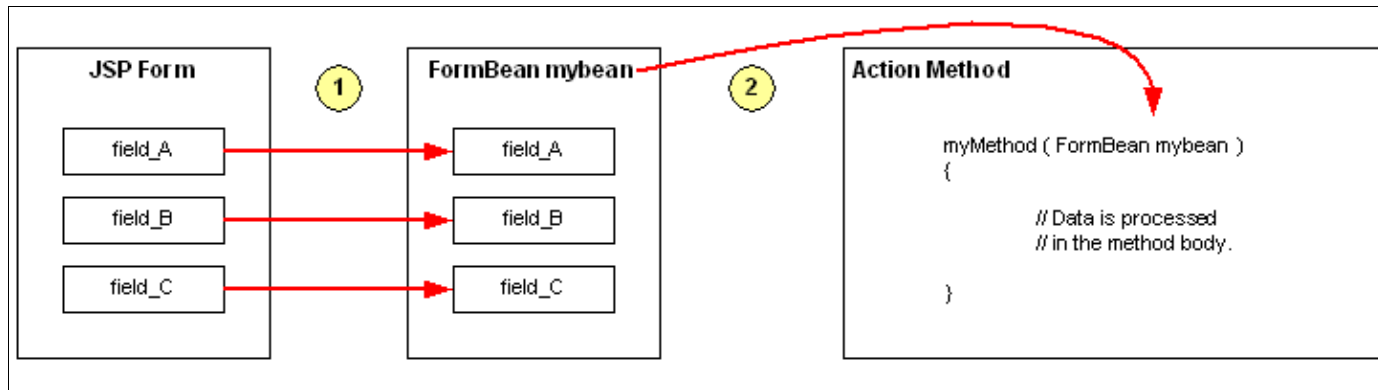
In this case, the Flow View depicts:

- A method, named begin
- An arrow, indicating that the begin method forwards users to the index.jsp page
- A JSP page named index.jsp

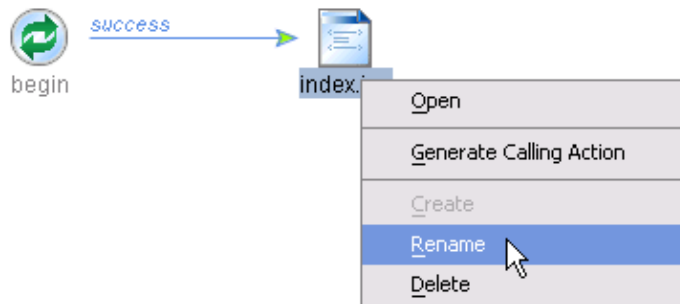


## Make a Submission Page, a Form Bean, and a Method

In this step you will create a JSP page where users can submit data. You will also create a form bean and a method to handle the submission. The submission process has two steps: (1) the submitted data is loaded into matching fields in a form bean and (2) the form bean is passed to a method for processing.



1. In **Flow View**, right-click the icon *index.jsp* and select **Rename**.



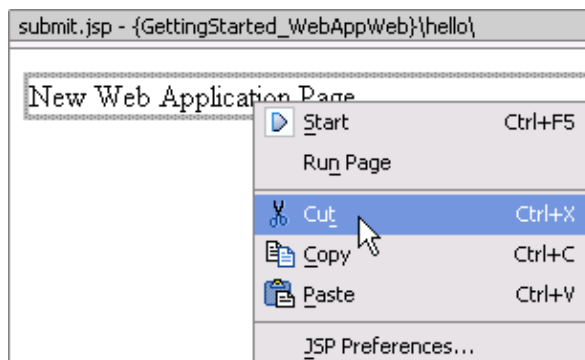
2. In the field provided, enter submit and press the **Enter** key.



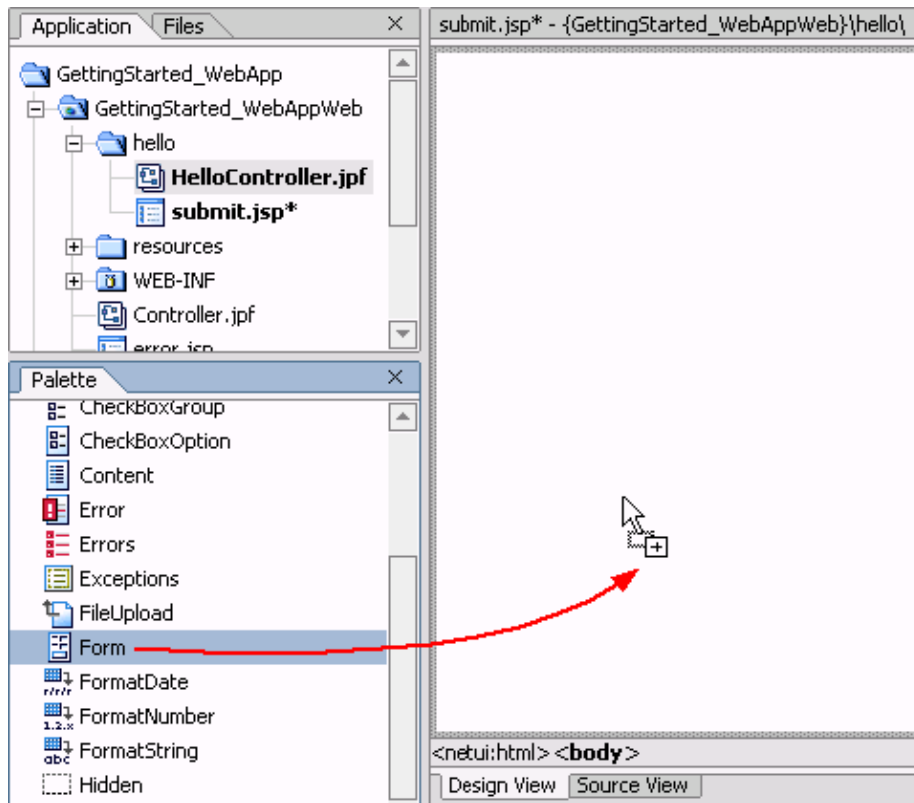
3. Double-click the icon *submit.jsp* to open this page in the main area.



4. Right-click the text *New Web Application Page* and select **Cut**.

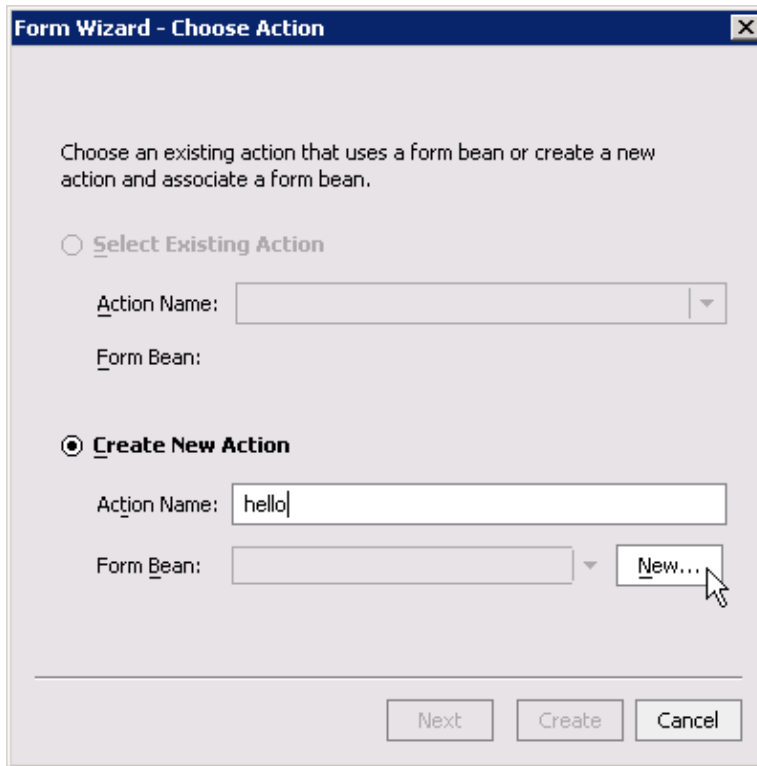


- From the **Palette** tab, drag and drop the **Form** icon into **Design View**. The **Form Wizard – Choose Action** dialog appears.



- Note.* If the **Palette** tab is not shown, go to the **View** menu and select **Windows—>Palette**.
- In the **Form Wizard – Choose Action** dialog, in the **Action Name** field, enter hello.





**Form Wizard - Choose Action**

Choose an existing action that uses a form bean or create a new action and associate a form bean.

☐ Select Existing Action

Action Name:

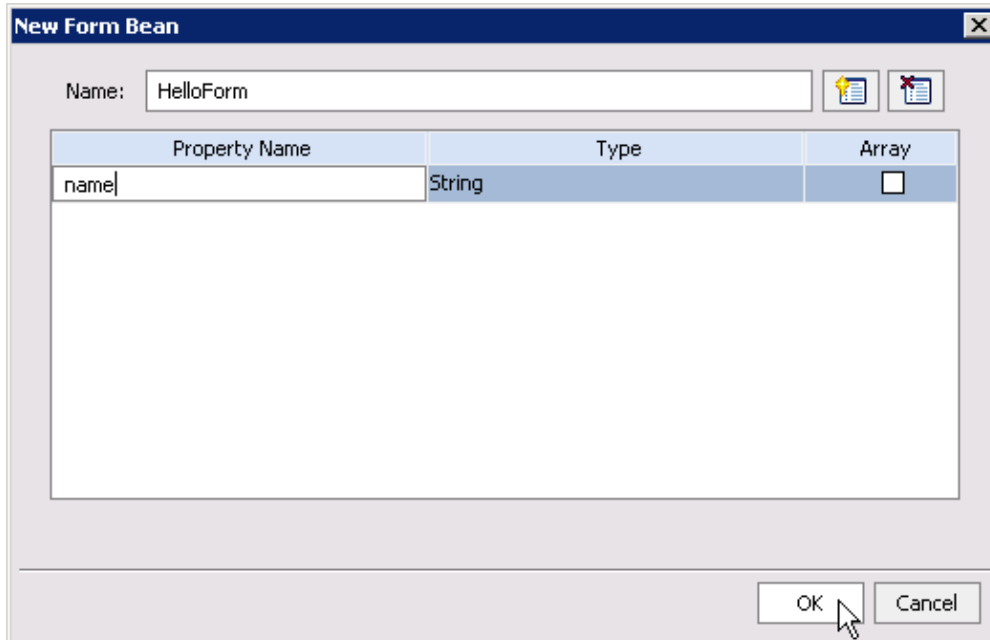
Form Bean:

☒ Create New Action

Action Name:

Form Bean:

7. Click **New...** to bring up the **New Form Bean** dialog.
8. In the **New Form Bean** dialog, in the **Property Name** field, enter name. You have now defined a name property in the form bean of type String.

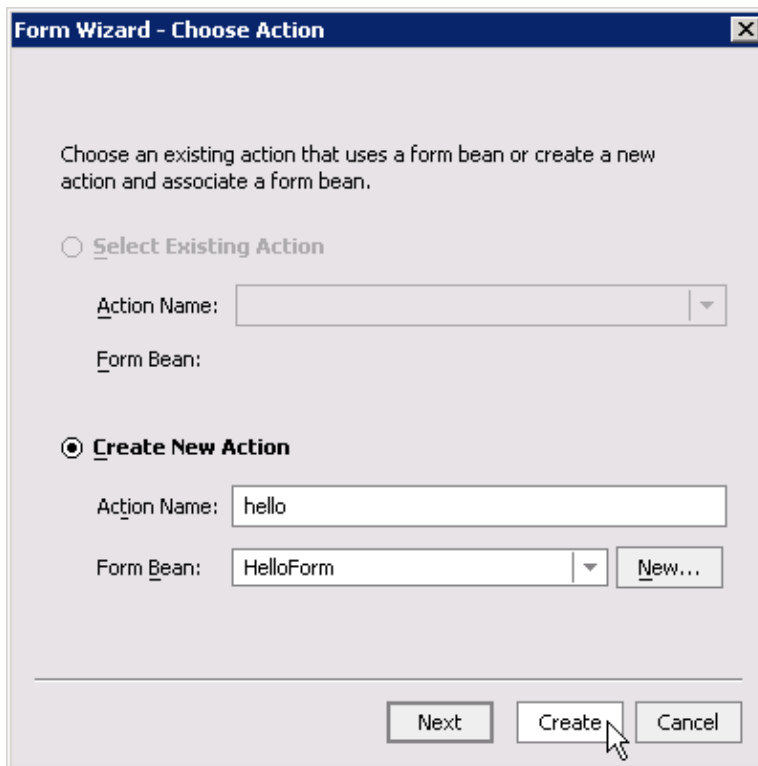


**New Form Bean**

Name:

Property Name	Type	Array
<input type="text" value="name"/>	String	<input type="checkbox"/>

9. Click **Ok** to return to the **Form Wizard – Choose Action** dialog.
10. In the **Form Wizard – Choose Action** dialog, click **Create**.



Form Wizard - Choose Action

Choose an existing action that uses a form bean or create a new action and associate a form bean.

☐ Select Existing Action

Action Name:

Form Bean:

☒ Create New Action

Action Name:

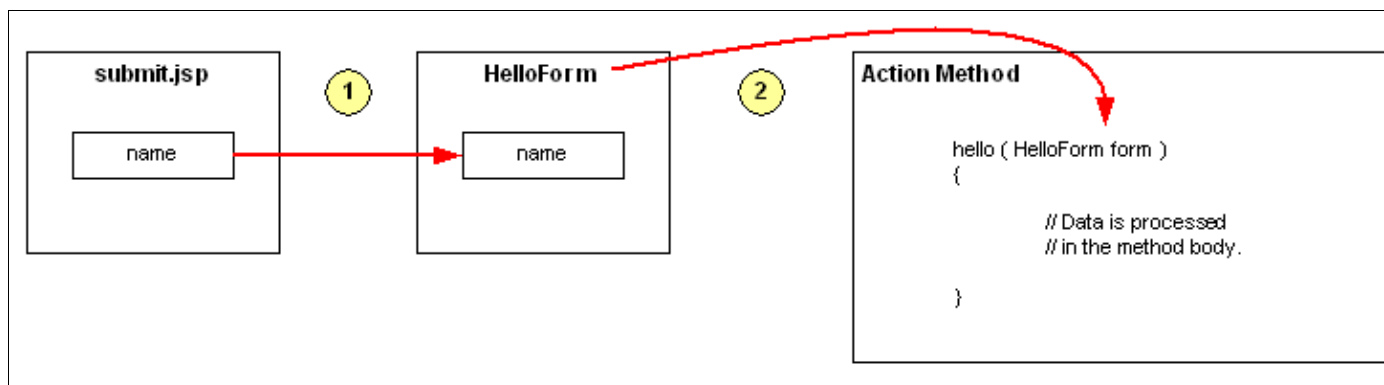
Form Bean:

11. Press **Ctrl+S** to save your work.

As this point you have added three new elements to the Page Flow:

- a set of JSP tags has been added to the JSP page submit.jsp
- an action method hello has been added to the Controller file HelloController.jpf. You can verify this by opening this file in Flow View. (To view the Controller file in Flow View, double-click HelloController.jpf and click the **Flow View** tab.)
- a form bean HelloForm has been added to the Controller file HelloController.jpf. This form bean is associated with the hello method. You can verify this by opening this file in Flow View and selecting this action method. The form bean is displayed in the **Form Bean** tab. (If this tab is not shown, go to the **View** menu and select **Windows-->Form Bean**.)

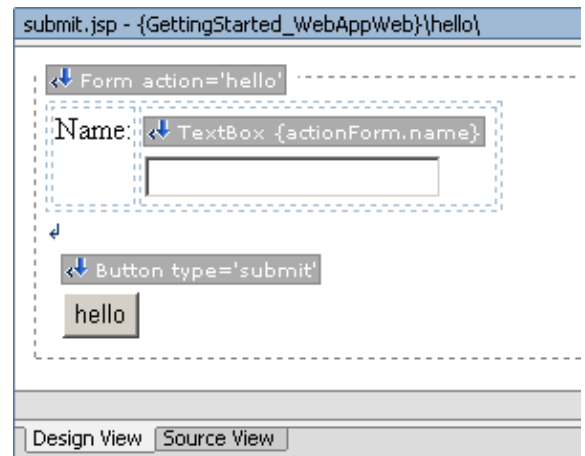
These three elements work together to submit data in the following way: (1) the submitted name field is bound to the name field in the HelloForm Form Bean and (2) the Form Bean is passed to the method hello for processing.



You can deduce this submission process from the Design View of submit.jsp and the Flow View of HelloController.jpf. The next two sections explain how to read these views.

## Reading the Design View of submit.jsp

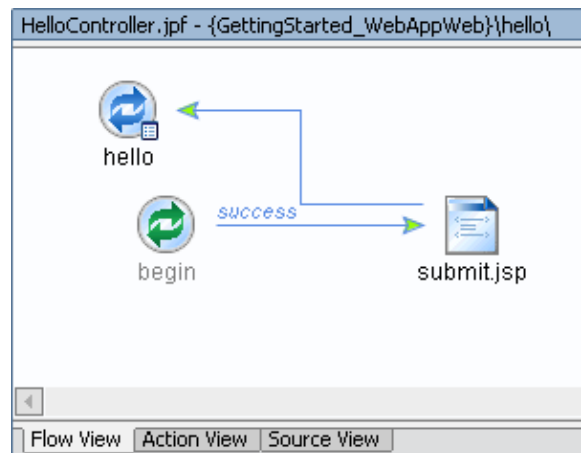
After completing the Form Wizard, the Design View of submit.jsp should look like the image to the right. Design View depicts the added JSP tags as three gray icons labeled **Form**, **TextBox**, and **Button**:



- Notice that the Form icon includes the text **action='hello'**. This indicates that the form passes data to the hello action method in the controller file
- Notice that the TextBox icon includes the text **{actionForm.name}**. This indicates that the TextBox data is bound to the name property of the form bean that is passed to the hello method

## Reading the Flow View of HelloController.jpf

The same submission process can be deduced from the Flow View of the Controller file. Notice the following graphical elements in Flow View:



- an **icon** labeled hello: this represents the action method hello
- a **small box** in the lower right-hand corner of the hello icon: this indicates that a form bean is passed to hello
- an **arrow** pointing from the submit.jsp icon to the hello icon: this indicates that data is submitted from the JSP, loaded into a Form Bean, and then passed to the method hello.

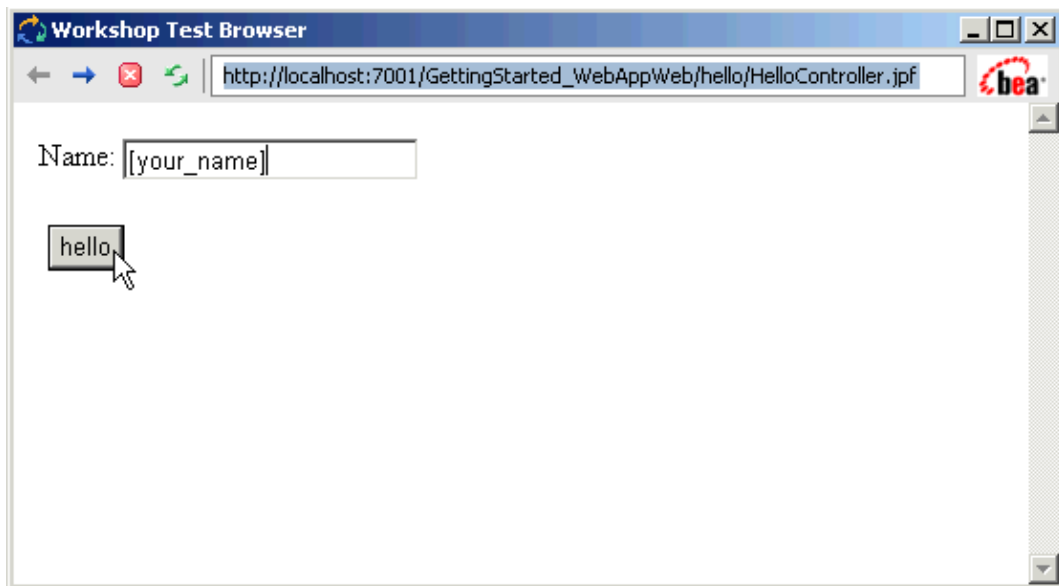
## Test the Web Application

Let's now test the page flow to make sure you have correctly executed the above steps.

1. Make sure that the HelloController.jspf file is displayed in the main area.
2. Click the **Start** button, shown below:



3. When the **Workshop Test Browser** launches, in the **Name** field, enter [your\_name] and click **hello**.



The following error is displayed in the **Workshop Test Browser**.



The reason an error is displayed is because the hello method does not contain instructions on the next

## WebLogic Workshop Tutorials

JSP page that needs to be called. You will correct this situation in the next step.

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



## Step 3: Processing Data

In this step you will learn how to process data submitted from users and forward users to a page where the results are displayed. The tasks in this step are:

- Edit the hello Method to Construct a Message
- Edit the hello Method to Forward the User
- Rearrange the Icons in Flow View
- Test the Web Application

### Edit the hello Method to Construct a Message

First you will edit the hello action method to implement the business logic for handling the user input. Specifically, you will construct a String message that takes the name from the form bean and you will make this message available to the next page. The message is made available to the next page by making it an attribute (property) of the request object, which can be read by the next JSP page.

1. Make sure that HelloController.jsp is displayed in Flow View in the main area.
2. In Flow View, double-click the **hello icon** to see the source code for this action.



3. In Source View, edit the hello method so it appears as follows. The added code is shown below in red:

```
/**
 * @jpf:action
 */
protected Forward hello(HelloForm form)
{
    // Construct a message from the submitted name.
    String message = "Hello, " + form.getName() + "!";

    // Place the message on the request object.
    getRequest().setAttribute("message", message);

    return new Forward("success");
}
```

4. Press **Ctrl+S** to save your work.

### Edit the hello Method to Forward the User

Now you will enhance the hello method to implement the navigation logic. Specifically, you will modify the action method to call the response.jsp page.

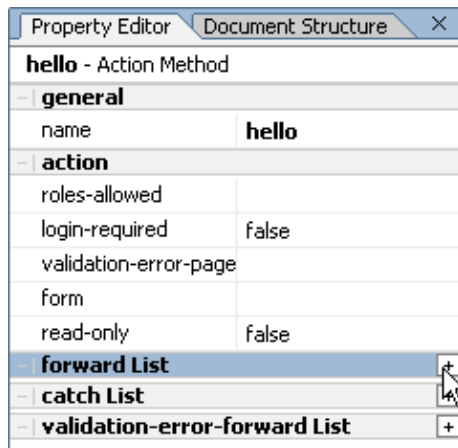
1. In **Source View**, place the cursor somewhere within the signature of the **hello** method.

```
/**
 * @jpf:action
 */
protected Forward hello(HelloForm form)
{
    // Construct a message from the submitted name.
    String message = "Hello, " + form.getName() + "!";

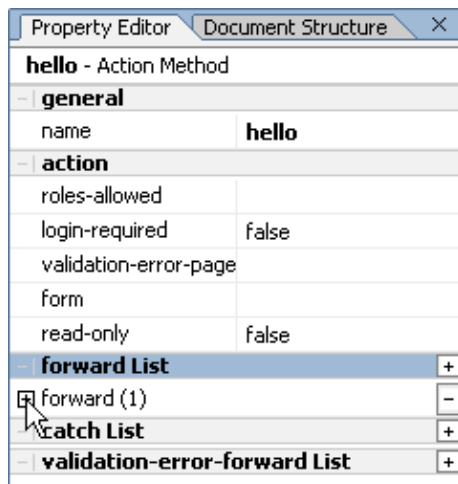
    // Place the message on the request object.
    getRequest().setAttribute("message", message);

    return new Forward("success");
}
```

2. In the *Property Editor*, click the *plus sign* to the right of the section labeled *forward List*.



3. In the *Property Editor*, click the *plus sign* to the left of *forward(1)*.



4. In the *Property Editor* in the section below *forward(1)*, in the *name* property, enter success, and press the *Enter* key.

Property Editor		Document Structure	X
<b>hello - Action Method</b>			
- <b>general</b>			
name	hello		
- <b>action</b>			
roles-allowed			
login-required	false		
validation-error-page			
form			
read-only	false		
- <b>forward List</b> +			
[-] forward (1) -			
name	success		
path			
return-action			
redirect	false		
return-to			
return-form-type			
return-form			
- <b>catch List</b> +			
- <b>validation-error-forward List</b> +			

5. In the *Property Editor* in the section below *forward(1)*, in the *path* property, enter `response.jsp`, and press the *Enter* key.

Property Editor		Document Structure	X
<b>hello - Action Method</b>			
- <b>general</b>			
name	hello		
- <b>action</b>			
roles-allowed			
login-required	false		
validation-error-page			
form			
read-only	false		
- <b>forward List</b> +			
[-] forward (1) -			
name	success		
path	response.jsp		
return-action			
redirect	false		
return-to			
return-form-type			
return-form			
- <b>catch List</b> +			
- <b>validation-error-forward List</b> +			

6. Press **Ctrl+S** to save your work.

The hello method is now complete and should look like this in Source View.



```

/**
 * @jpf:action
 * @jpf:forward path="response.jsp" name="success"
 */
protected Forward hello(HelloForm form)
{
    // Construct a message from the submitted name.
    String message = "Hello, " + form.getName() + "!";

    // Place the message on the request object.
    getRequest().setAttribute("message", message);

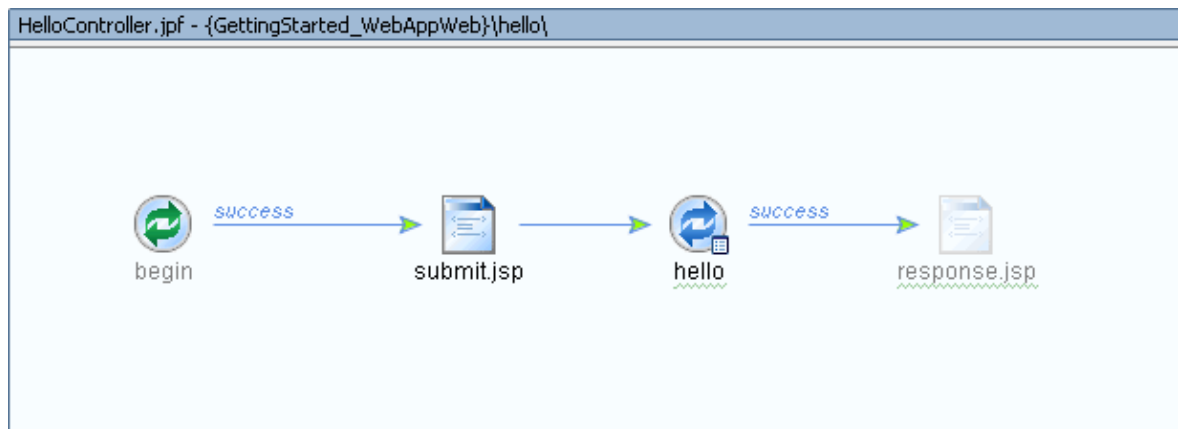
    return new Forward("success");
}

```

## Rearrange the Icons in Flow View

Let's rearrange the icons in Flow View so that it becomes easier to read the flow.

1. Click the **Flow View** tab.
2. Rearrange the icons so that they appear as follows.



Notice that the response.jsp icon is grey out. This indicates that the JSP page response.jsp is referred to in the Controller file, but does not yet exist in the Page Flow folder.

3. Press **Ctrl+S** to save your work.

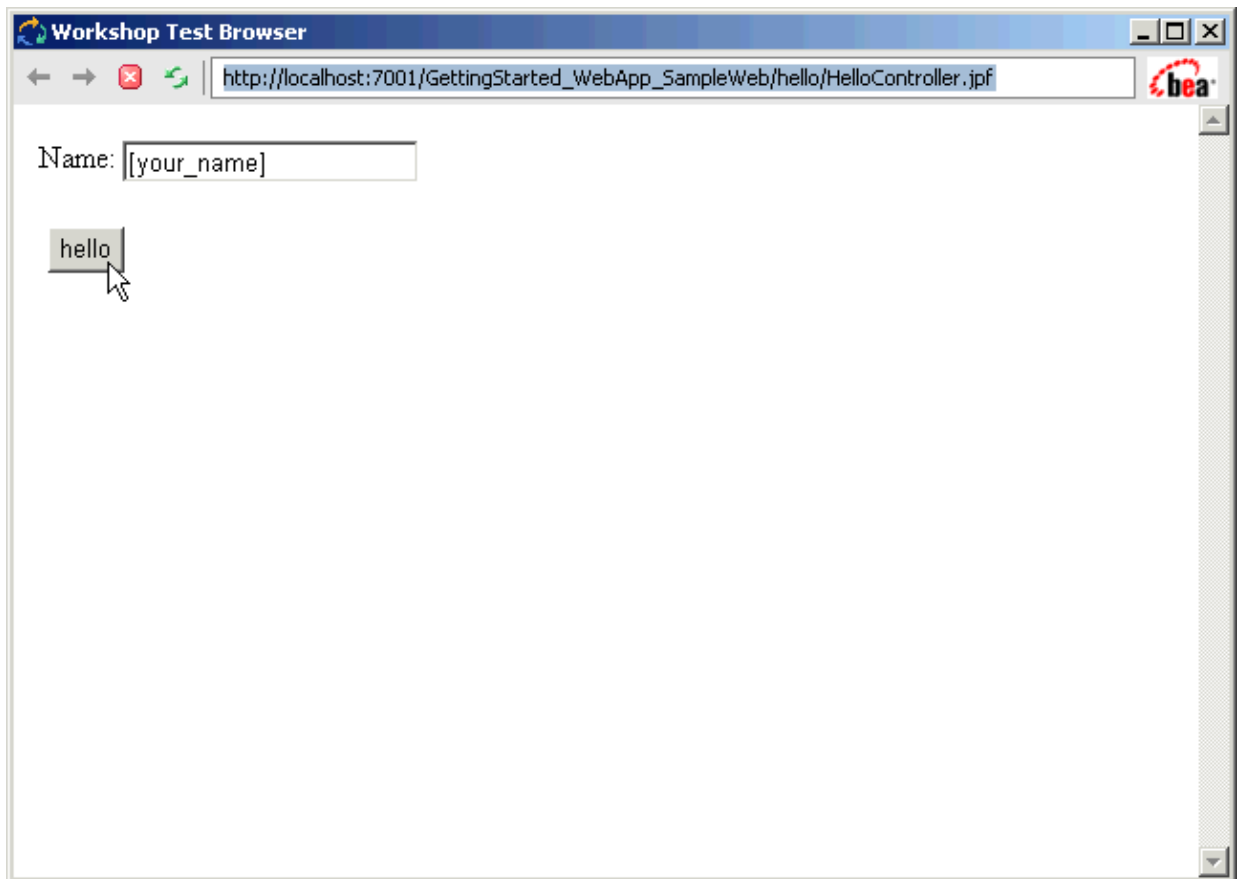
## Test the Web Application

Let's now test the page flow to make sure you have correctly executed the above steps.

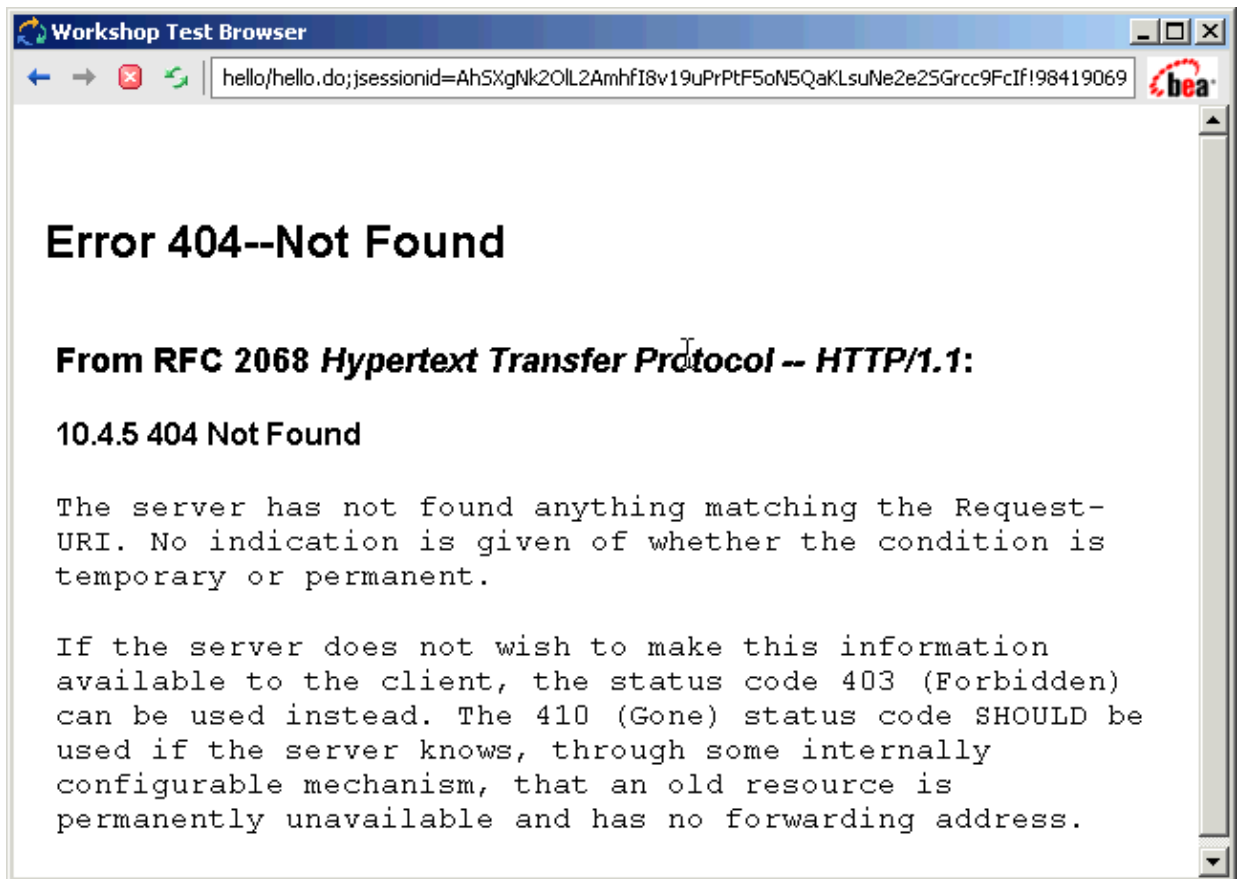
1. Make sure that the HelloController.jfp file is displayed in the main area.
2. Click the **Start** button, shown below:



3. When the **Workshop Test Browser** launches, in the **Name** field, enter [your\_name] and click **hello**.



The following error is displayed in the *Workshop Test Browser*.



The reason an error is displayed is because the hello method forwards the user to the response.jsp page but that does not yet exist. You will correct this situation in the next step.

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



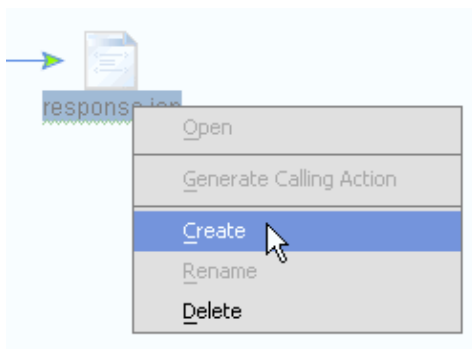
## Step 4: Displaying Results

In this step you will learn how to display data to a user. The tasks in this step are:

- Create the Response Page
- Edit the Page
- Test the Web Application

### Create the Response Page

- In *Flow View*, right-click the *response.jsp icon* and select *Create*.

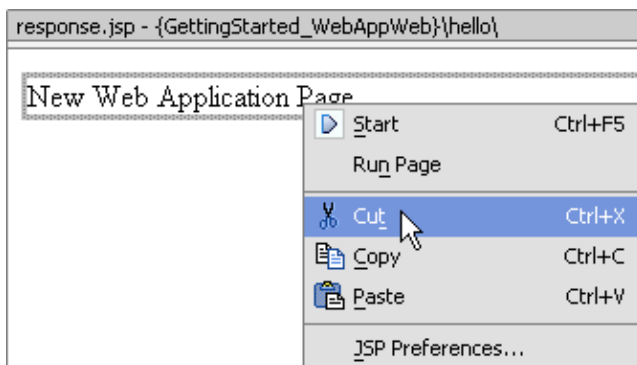


### Edit the Page

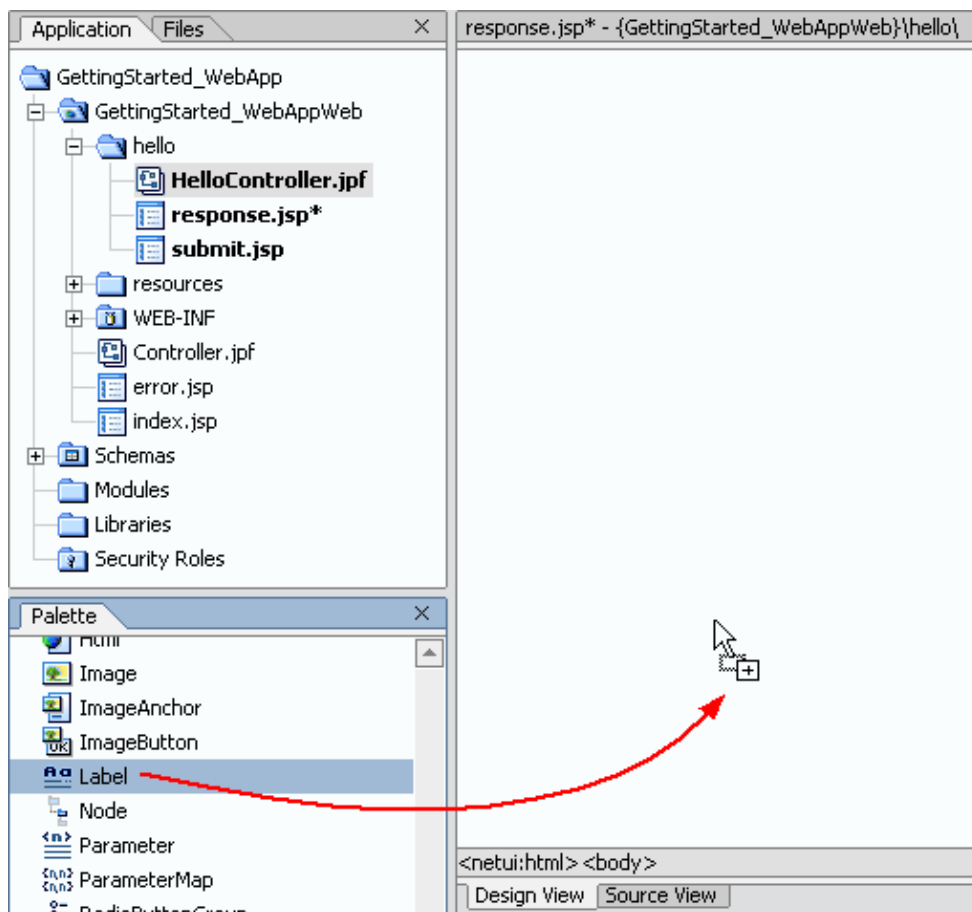
1. In *Flow View*, double-click the *response.jsp icon* to open this page in Design View.



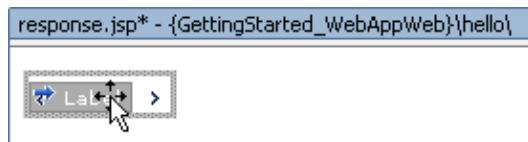
2. In Design View, right click the text *New Web Application Page* and select *Cut*.



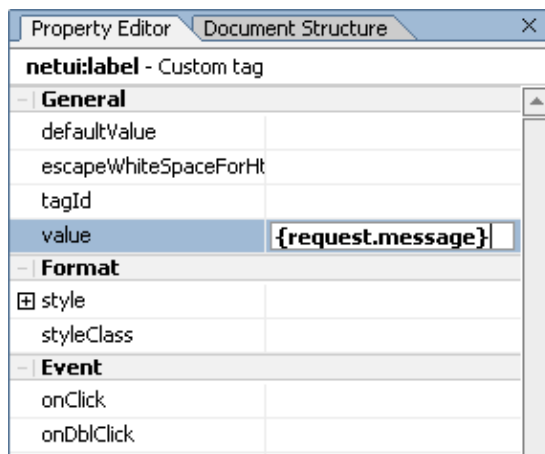
3. From the *Palette* tab, drag and drop the *Label* icon into *Design View*.



4. Click the gray **label icon** to ensure that it is selected.



5. In the **Property Editor**, in the section labeled **General**, in the **value** property, enter `{request.message}`, and press the **Enter** key. This will cause the String message, which is created by the hello method, to be displayed on the page.



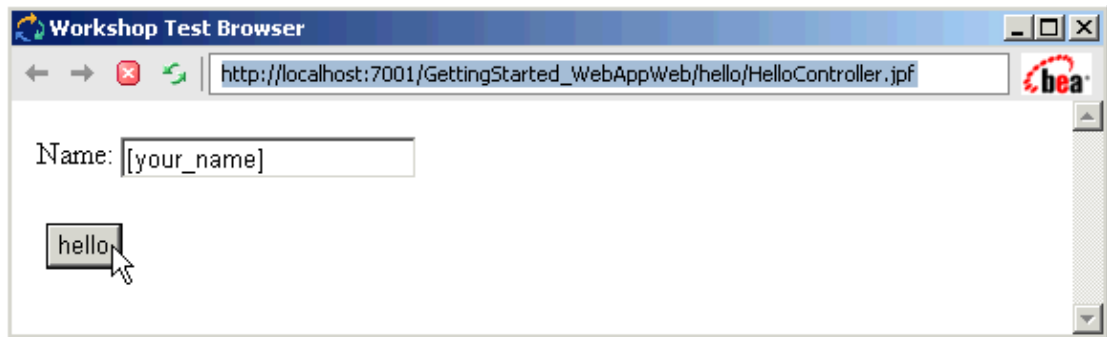
6. Press **Ctrl+S** to save your work.

## Test the Web Application

1. On the *Application* tab, double-click the file *HelloController.jspf*.
2. Click the *Start* button, shown below:



3. When the *Workshop Test Browser* launches, in the *Name* field, enter [your\_name], and click *hello*.



4. Your data is submitted to the hello method, a message is constructed, and that message is displayed on the response.jsp page.



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



# Summary: Getting Started Web Application Tutorial

This tutorial introduced you to the basics of building Page Flows with WebLogic Workshop.

## Concepts and Tasks Introduced in This Tutorial

- Using page flows to separate user interface from business and navigation logic
- Data binding

Now that you have finished the basic web application tutorial, you can continue with one of the other basic tutorials to learn more about the other core components you can develop in WebLogic Workshop. For more information, see The Getting Started Tutorials. If you want to learn more about page flow development, go to the advanced Tutorial: Page Flow. You can learn more about the page flow design at Guide to Building Page Flows.

### Related Topics

[The Getting Started Tutorials](#)

[Tutorial: Page Flow](#)

[Guide to Building Page Flows](#)

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



# Getting Started: Java Controls

## The Problem of Organizing Business Logic

When you model a real-world problem in a software application, you are faced with modeling the behavior of multiple business objects including the rules that govern their interactions. To model the business objects, you need some way to persistently represent their properties, that is, store the information in a database. To model the rules, you need to be able to break the business logic into smaller subtasks that comprise a complex interaction, and you need to have some way to encapsulate these subtasks such that they can be reused to model other interactions. For instance, if your software application is an online solution for a customer buying books or CDs, you need a way to save information regarding customers, books, CDs, credit cards, and so forth. In addition, buying a CD and buying a book are two tasks that have common subtasks, such as the use of a credit card to pay for the purchased item. You want to be able to separate this subtask and reuse it in both purchase scenarios.

## The Java Control Solution

WebLogic Workshop provides Java controls that make it easy for you to encapsulate business logic and to access enterprise resources such as databases, legacy applications, and external web services. Java controls that access enterprise resources are known as built-in controls. An example of a built-in control is a database control, which specializes in querying relational databases to access stored information.

Java controls that are specifically used to encapsulate business logic are called custom controls. A custom control can invoke other custom controls that handle parts of the larger task. For instance, a custom control that handles a book purchase might invoke another custom control that handles payment of a product by credit card. Also, a custom control can invoke built-in controls, such as a database control to retrieve the data required to execute the business task. For instance, a custom control that handles a book purchase might invoke a database control to verify that the book is still in stock.

## What This Tutorial Teaches

In this tutorial you will learn the basics of using a custom control to encapsulate business logic, and using a database control to query information stored in relational databases. You will create two controls to model a simplified login procedure. The procedure consists of a user submitting a name and receiving a greeting that is based on the number of times the user has submitted his/her name previously. To model this procedure you will create (1) a database control that creates records in a database regarding the number of visits by a user, and (2) a custom control that receives the user input, queries the database control to find out the status of the user, and returns a status-specific greeting.

Related Topics

Getting Started with Java Controls

Click the arrow to navigate to the next step.





# Getting Started: Java Control Core Concepts

This topic will familiarize you with the basic concepts of Java control design and development.

## What is a Java control?

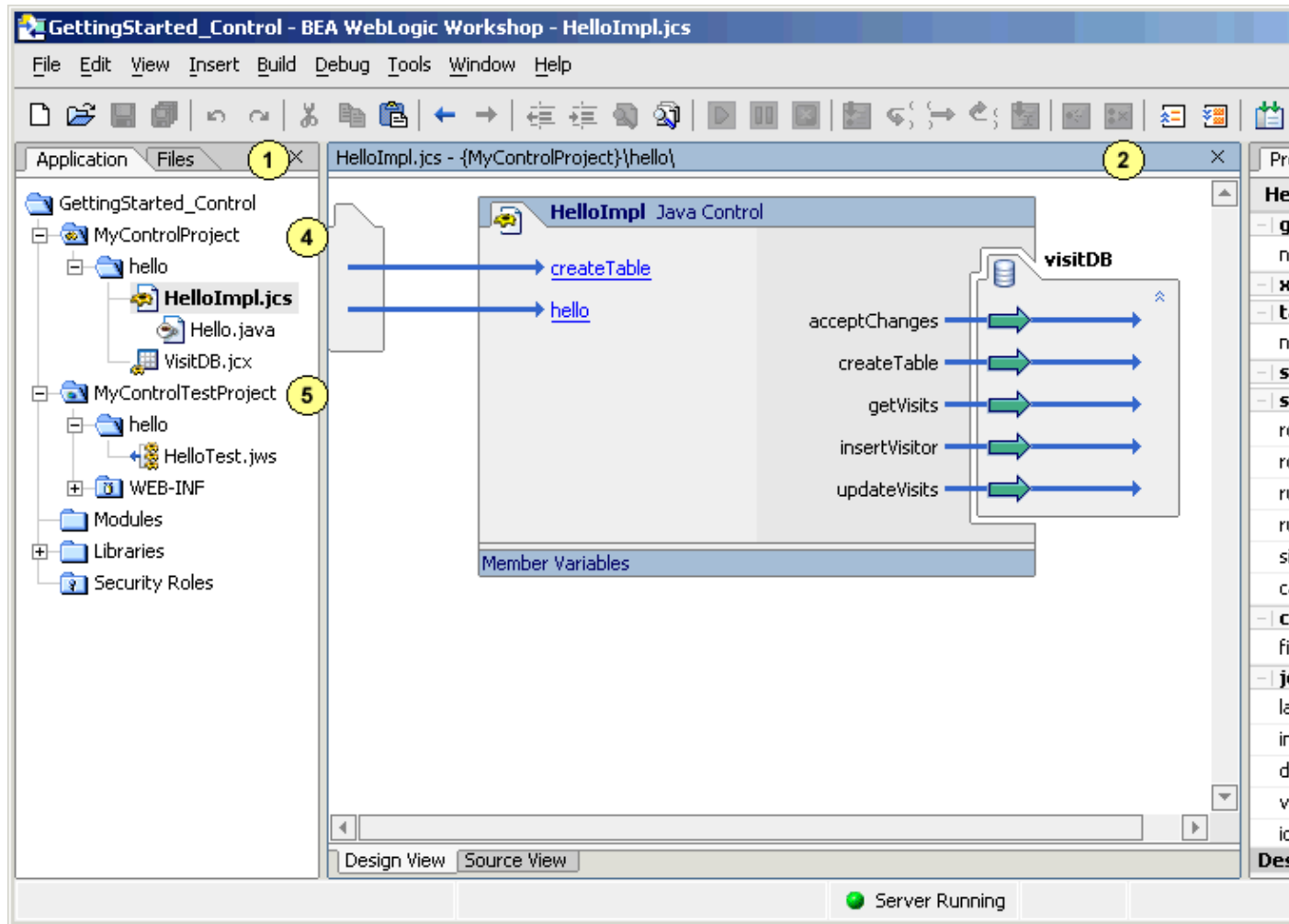
Java controls are reusable components that are optimized to access enterprise resources and execute business tasks. In this tutorial you will see how to use custom controls and one type of built-in control called the database control. Other built-in controls are discussed in the advanced Tutorial: Java Control. For an overview of Java controls, see *Working with Java Controls*.

A database control makes it easy to access a relational database. A database control method is associated with an SQL query against a database. When a database method is invoked, the corresponding SQL query is executed to retrieve data, perform operations like inserts and updates, and even make structural changes to the database. In this tutorial you will define methods for all these operations, that is, you will create methods to create a database table, insert a new record, update an existing record, and read a record. The database control automatically handles the work of connecting to the database, so you don't have to understand JDBC to work with a database.

Custom controls model business logic and are often an intermediary between your client application, such as a web application, and built-in controls. For instance, a *Purchase* custom control might take an 'add to shopping cart' request for a particular product through a web page, use a database control to query whether this product is still in stock, and return a response to the web application that the product is not available, is backordered, or is available and has been successfully added to the cart.

## Developing Java Controls with WebLogic Workshop

Java controls are developed using WebLogic Workshop, a visual tool for designing J2EE applications. The image below shows the finished tutorial.



The **Application** tab (area 1) shows the application's source files. A custom control is stored in two files, that is a .jcs and a .java file. When you define a custom control, you only work in the .jcs file; WebLogic Workshop automatically synchronizes the associated .java file. A database control, like most built-in controls, is stored in a file with a .jcx extension.

The main work area (area 2) shows the Design or Source View of the components you are building, in this case the Design View of a custom control.

The **Property Editor** (area 3) allows you to see and set properties of the Java control under development.

A Java control project folder contains Java controls that are developed for reuse throughout the application as well as in other applications. A Java control project is like an organizational unit; when building controls you build all the Java controls in a project and the resulting files are wrapped in a single JAR file with the same name as the project. You can have one or multiple control projects in an application. In this tutorial you will develop all Java controls using one Java control project folder called MyControlProject (area 4). It is also possible to develop a Java control directly in a web or web service project; in that case the Java controls are meant to be used only in the web or web service project and not throughout the application or across applications.

Other components of the application are built in other projects. In this tutorial you will build a test web service, which you will use as a client application to test the controls. Web services are created in a web

## WebLogic Workshop Tutorials

service project folder, called MyControlTestProject in this tutorial (area 5).

### Related Topics

[Getting Started with Java Controls](#)

[Overview: Database Controls](#)

[Building Custom Java Controls](#)

Click the arrow to navigate to the next step.



# Step 1: Create a Control Project

In this step you will create the application that you will be working with throughout this tutorial. In WebLogic Workshop, an application contains one or more projects. You will create two projects, namely a control project for the Java controls and, during a later step, a web service project for the test web service used to test the Java controls.

The tasks in this step are:

- To Start WebLogic Workshop
- To Create a New Application and Select a WebLogic Server Domain
- To Create a New Control Project
- To Start WebLogic Server

To Start WebLogic Workshop

If you have an instance of WebLogic Workshop already running, you can skip this step.

... on Microsoft Windows

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

- From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**WebLogic Workshop 8.1**.

...on Linux

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

- Open a file system browser or shell.
- Locate the **Workshop.sh** file at the following address:

```
$HOME/BEA/weblogic81/workshop/Workshop.sh
```

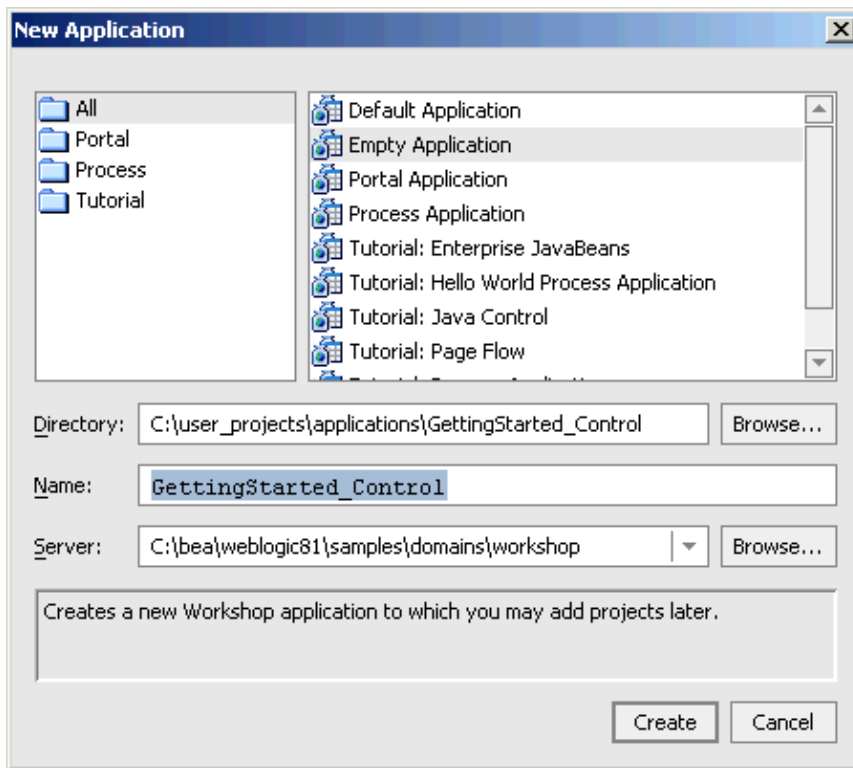
- In the command line, type the following command:

```
sh Workshop.sh
```

To Create a New Application and Select a WebLogic Server Domain

To create the control project, you must first create the application to which it will belong:

1. From the **File** menu, select **New**—>**Application**. The **New Application** dialog appears.
2. In the **New Application** dialog, in the upper left-hand pane, select **All**.  
In the upper right-hand pane, select **Empty Application**.  
In the **Directory** field, use the **Browse** button to select a location to save your source files. A suggested location is `BEA_HOME\user_projects\applications\GettingStarted_Control`.  
In the **Name** field, enter `GettingStarted_Control`.  
In the **Server** field, from the drop-down list, select `BEA_HOME\weblogic81\samples\domains\workshop`.

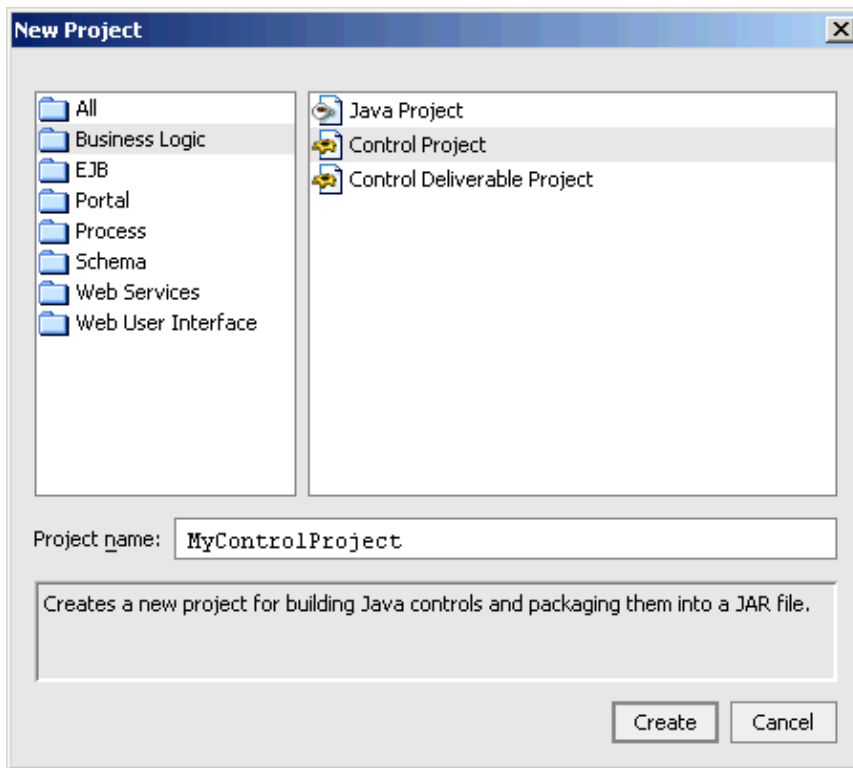


3. Click **Create**.

To Create a New Control Project

You will need a control project to contain the Java controls that you are going to create. A control project can be thought of as a unit of work; all controls in a given project will be built together and will be packaged in the same JAR file.

1. In the **Application** tab, right-click the GettingStarted\_Control folder and select **New-->Project**.
2. In the **New Project** dialog, in the upper left-hand pane, confirm that **Business Logic** is selected. In the upper right-hand pane, select **Control Project**. In the **Project Name** field, enter MyControlProject.

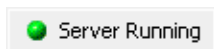


3. Click **Create**.

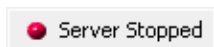
To Start WebLogic Server

Since you will be deploying and running your Java control on WebLogic Server, it is helpful to have WebLogic Server running during the development process.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.

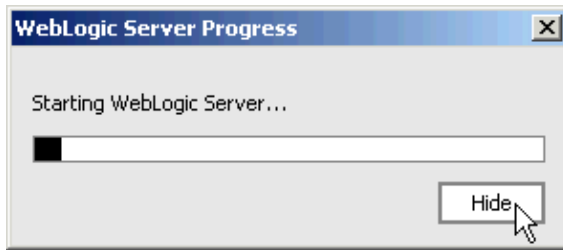


If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow these instructions to start WebLogic Server:

- From the **Tools** menu, choose **WebLogic Server**—>**Start WebLogic Server**.
- On the **WebLogic Server Progress** dialog, you may click **Hide** and continue to the next task.



### Related Topics

[Getting Started with Java Controls](#)

[How Do I...? Java Control Topics](#)

[The WebLogic Workshop Development Environment](#)

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



## Step 2: Define the Database Control

In this step you will create the database control *VisitDB*, which runs queries against a database table containing information about visitors and number of prior visits. A database control queries a database table to store, find, and update information.

The tasks in this step are:

- To Create a Package
- To Create a Database Control
- To Define Methods
- Build the Database Control

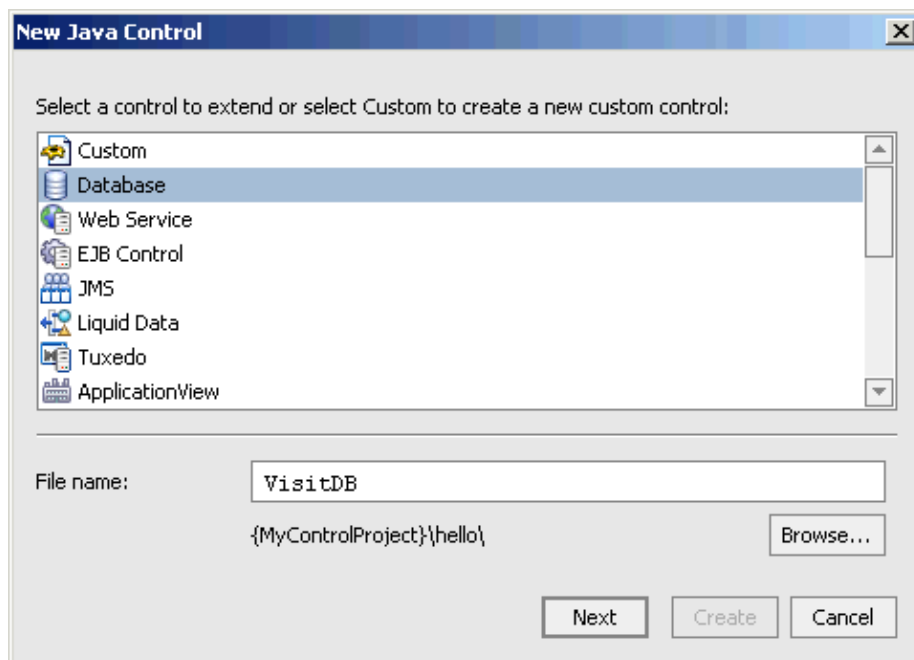
To Create a Package

All Java controls (and all Java classes) must be part of a Java package.

1. Right-click the **MyControlProject** folder in the **Application** tab, and select **New-->Folder**.
2. Enter the folder name *hello*.
3. Click **OK**.

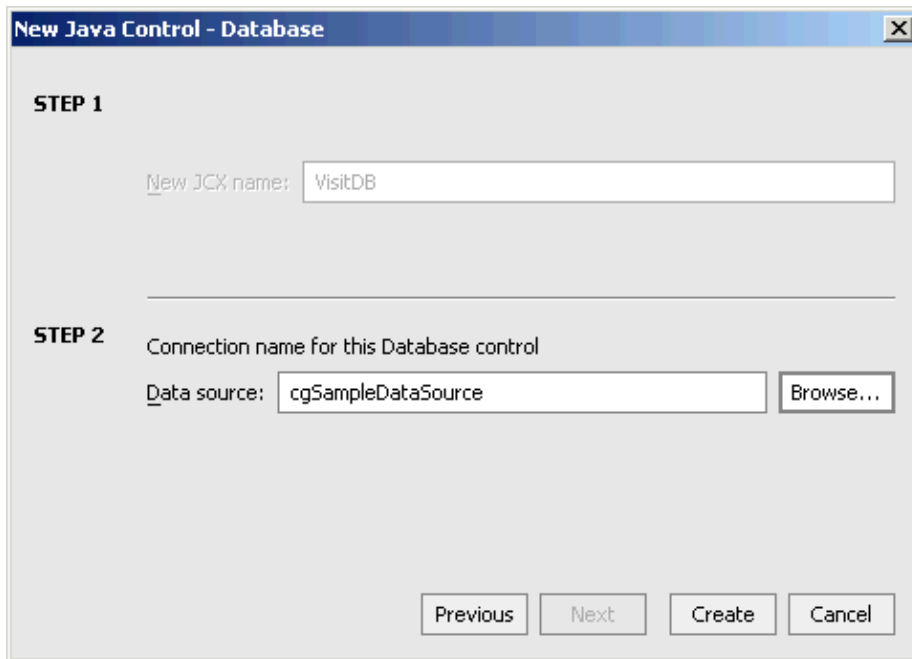
To Create a Database Control

1. Right-click the **hello** folder in the **Application** tab, and select **New-->Java Control**.
2. In the **New Java Control** dialog, select **Database** and enter the file name *VisitDB*.



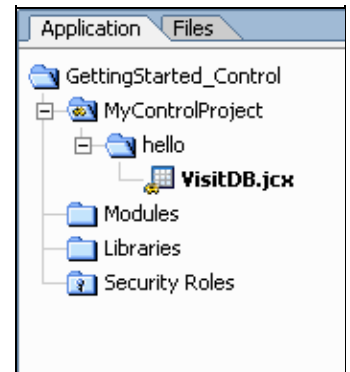
3. Click **Next**.
4. In **STEP 2**, make sure that *cgSampleDataSource* is selected. If this is not the case, click the **Browse** button to select it. If this data source is not given as an option, you selected the wrong domain in the previous step when you created the application.





5. Click **Create**.

The database control opens in Design View. The **Application** tab should now look like the picture on the right.



The top-level folder *GettingStarted\_Control* is the containing folder for the entire application. All of the source files for your application exist in this folder.

The folder *MyControlProject* is the control project folder. An application can contain any number of project folders and there are many different kinds of project folders, including Web projects, Web Services projects, Schema projects, and so forth.

The **Modules** folder is for the storage of stand-alone applications (packaged as WAR and JAR files) that can be deployed parallel with your web service, if you wish.

The **Libraries** folder is for the storage of resources that are to be used across multiple projects, provided the resources are packaged as JAR files. For example, if you had a control or an EJB that you wanted to re-use across all your projects, you would store it in the Libraries folder.

The **Security Roles** folder lets you define security roles and test users for your web application. You can test the security design of your web application by logging in as a test user.

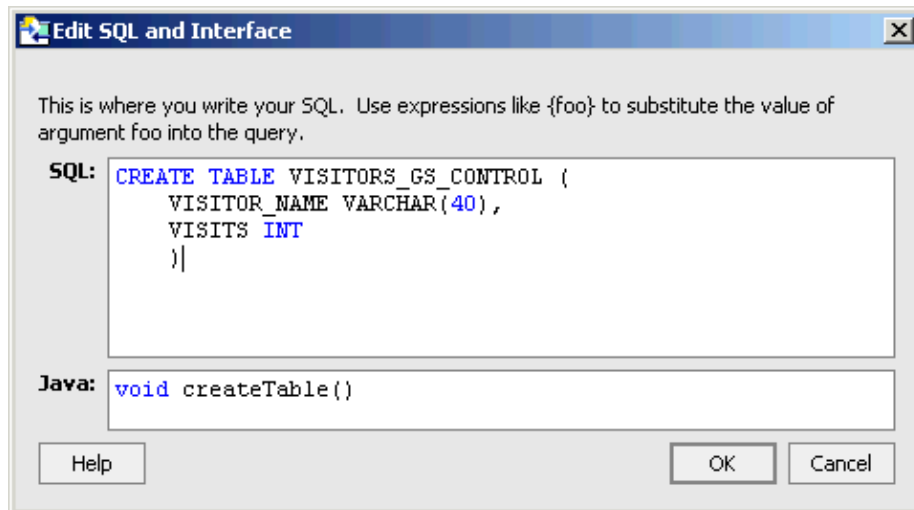
## To Define Methods

Next you will define the methods of the database control that, when invoked, execute a SQL query. A database control method consists of two parts, namely the method signature and the SQL query that is executed by the method. The first method you are going to create is called `createTable`, and it executes the SQL query that will create the database table `VISITORS_GS_CONTROL` used by the database control.

1. Right-click `VisitDB` in design view, and select **Add Method**. A method `newMethod1` appears.
2. Rename this method `createTable`. If you step off the method prematurely, right-click the method and select **Rename**.
3. Right-click the method and select **Edit SQL**. The **Edit SQL and Interface** dialog appears.
4. In the **SQL** field, enter the following SQL query expression:

```
CREATE TABLE VISITORS_GS_CONTROL (
    VISITOR_NAME VARCHAR(40),
    VISITS INT
)
```

The dialog should now look like this



5. Click **OK**.

In the preceding steps you learned how to add a method to a database control. Next you are going to add three additional methods in the same manner. These methods are:

- `getVisits`. This method takes the visitor's name as an argument and returns the number of previous visits by this visitor
- `insertVisitor`. This method takes the visitor's name and adds the name to the database
- `updateVisits`. This method takes the visitor's name and a number of visits as arguments and updates the database to store the number of visits for this visitor

1. Right-click `VisitDB` in design view, and select **Add Method**. A method `newMethod1` appears. Rename this method `getVisits`.
2. Right-click the `getVisits` method and select **Edit SQL**. In the **SQL** field of the **Edit SQL and Interface** dialog, enter the following SQL query expression:

```
SELECT VISITS FROM VISITORS_GS_CONTROL WHERE VISITOR_NAME = {name}
```

In the *Java* field of the *Edit SQL and Interface* dialog, change the signature of the method to:

```
int getVisits(String name)
```

Click **OK**.

3. Right-click VisitDB in design view, and select **Add Method**. A method newMethod1 appears. Rename this method insertVisitor.
4. Right-click the insertVisitor method and select **Edit SQL**. In the *SQL* field of the *Edit SQL and Interface* dialog, enter the following SQL query expression:

```
INSERT INTO VISITORS_GS_CONTROL (VISITOR_NAME, VISITS) VALUES ({name}, 1)
```

In the *Java* field of the *Edit SQL and Interface* dialog, change the signature of the method to:

```
int insertVisitor(String name)
```

Click **OK**.

5. Right-click VisitDB in design view, and select **Add Method**. A method newMethod1 appears. Rename this method updateVisits.
6. Right-click the updateVisits method and select **Edit SQL**. In the *SQL* field of the *Edit SQL and Interface* dialog, enter the following SQL query expression:

```
UPDATE VISITORS_GS_CONTROL SET VISITS = {visits} WHERE VISITOR_NAME = {name}
```

In the *Java* field of the *Edit SQL and Interface* dialog, change the signature of the method to:

```
int updateVisits(String name, int visits)
```

Click **OK**.

7. Save your results.

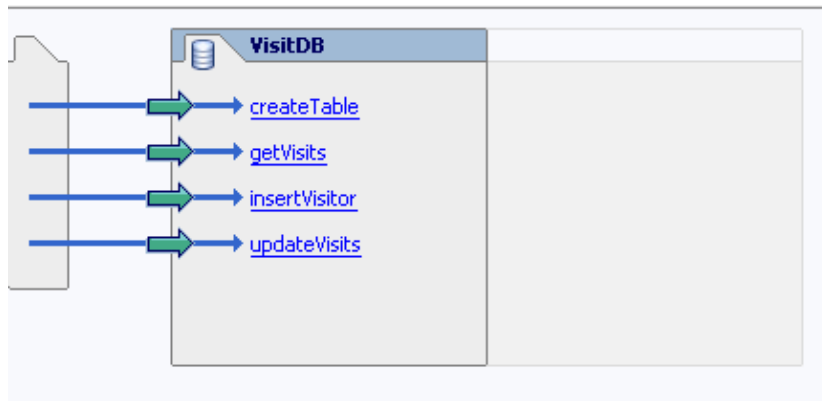
As you might have noticed above, arguments in the method are passed to the SQL query, and the results returned by the query are in turn returned by the method. For example, the method getVisits takes a String name as an argument. This name is passed to the SQL query, as shown below in bold:

```
SELECT VISITS FROM VISITORS_GS_CONTROL WHERE VISITOR_NAME = {name}
```

This query returns a number, which is returned as an int value by the method.

**Note.** The insertVisitor and updateVisits methods return the values 0 and 1, indicating whether the insertion/updating of a record failed or was successful. In this tutorial you will not explicitly test the returned values to determine success or failure.

The VisitDB control should now look like this in Design View:



### Build the Database Control

Now let's build the database control to make sure it is defined correctly.

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View**—>**Windows**—>**Build**.
2. In the **Application** pane, right-click MyControlProject, and select **Build MyControlProject**.
3. Monitor the **Build** window and verify that the build completes correctly
4. Open the **Libraries** folder in the **Application** pane, and notice that the file MyControlProject.jar has been created.

If you encounter build errors, verify that the various SQL queries and Java methods have been created correctly.

### Related Topics

#### Database Control

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



## Step 3: Define the Custom Control

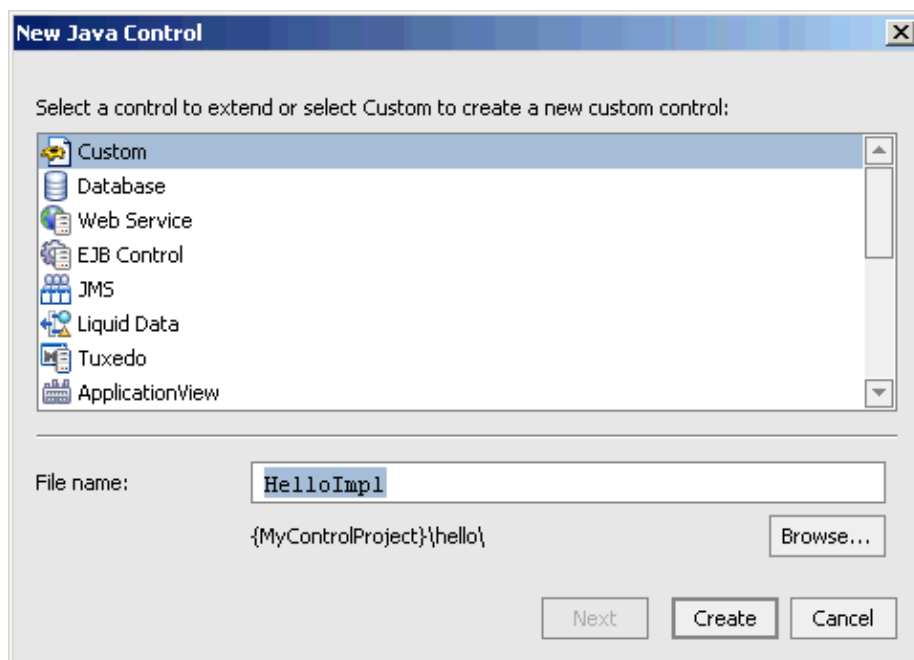
In this step you will create the custom control *Hello*. The custom control will have two methods, `createTable` and `hello`. The `createTable` method will be called only once by the client application and calls the corresponding method on the database control *visitDB* to create the table to be used in this application. The `hello` method can be invoked repeatedly by the client application. This method takes a visitor's name as an argument, in turn calls the *visitDB* database control to determine how many times the visitor has visited before, and returns a greeting that takes into account the number of prior visits. In other words, the custom control implements the business logic that governs a particular interaction.

The tasks in this step are:

- To Create a Custom Control
- To Add the Database Control
- To Add the Methods
- Build and Deploy

To Create a Custom Control

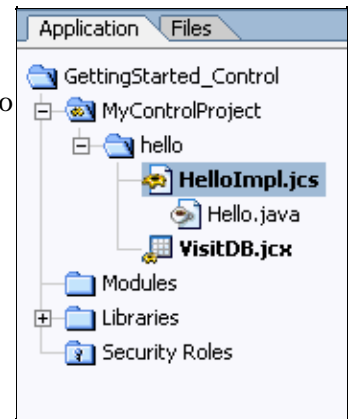
1. Right-click the *hello* folder in the *Application* tab and select *New*—>*Java Control*. The *New Java Control* dialog appears.
2. Make sure that Custom is selected in the top panel and enter the file name `HelloImpl`.



3. Click *Create*.

## WebLogic Workshop Tutorials

The files `HelloImpl.jcs` and `Hello.java` now appear in the **Application** tab above `VisitDB.jcx` in the **hello** folder. As you build your Java control you will work in the JCS file. WebLogic Workshop automatically updates the code in JAVA file to reflect the changes you made in the JCS file. In other words, you should not edit the JAVA file directly.

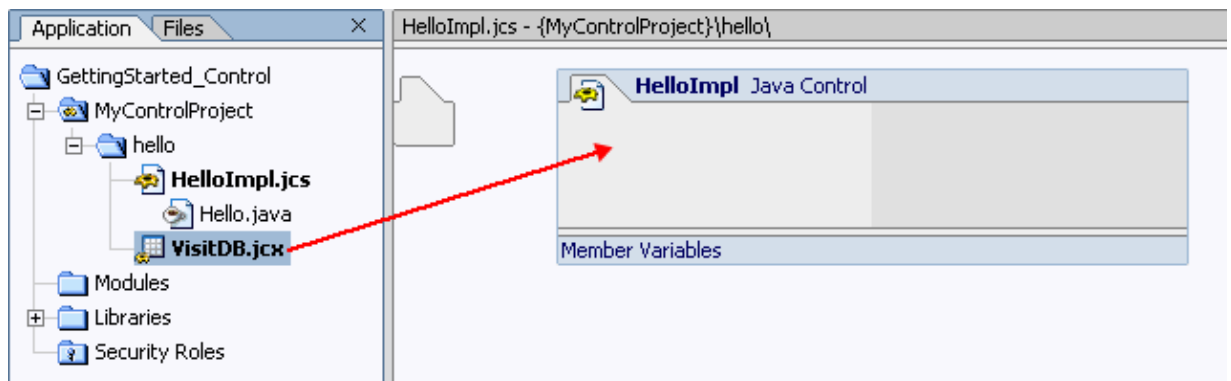


Also notice that the custom control opens in Design View in the main area of the IDE.

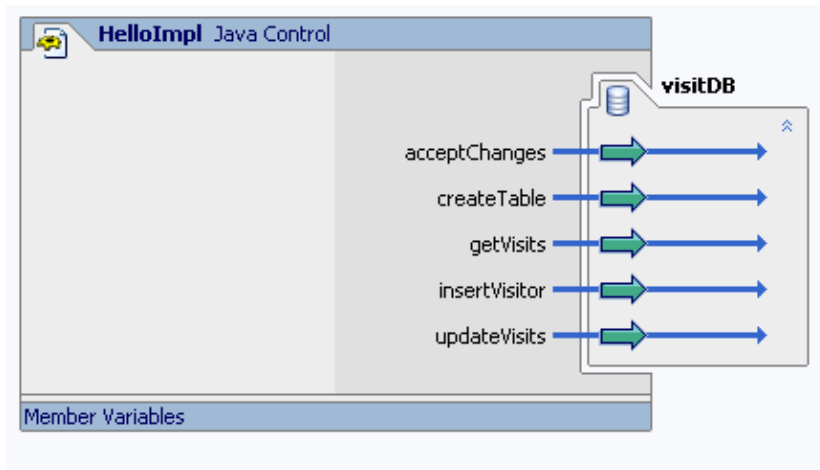
To Add the Database Control

The *Hello* custom control will use the *VisitDB* database control to determine if a visitor is new or returning. In order for the custom control to invoke the database control's methods, a database control instance must be defined for the custom control. To do so, you simply drag and drop the database control into the custom control's Design View.

1. Make sure the Hello custom control is opened in Design View.
2. In the **Application** tab, locate `VisitDB.jcx`, and drag and drop it onto Hello.



The `VisitDB` database control has now been added to the Hello custom control.



### To Add the Methods

Now you will implement the methods `createTable` and `hello` to create a table and to implement the business logic that computes a response to a user on the basis of his/her previous number of visits.

1. Ensure that the Hello control is displayed in Design View.
2. Right-click Hello and choose **Add Method**. A method called `newMethod1` appears.
3. Rename the method `createTable`. If you step off the method and need to rename it, right-click the method and select **Rename**.
4. Click the `createTable` method to go to Source View and modify the method as shown in red below:

```
/**
 * @common:operation
 */
public void createTable()
{
    visitDB.createTable();
}
```

5. Return to Design View, right-click Hello and choose **Add Method**. A method called `newMethod1` appears.
6. Rename the method `hello`. If you step off the method and need to rename it, right-click the method and select **Rename**.
7. Click the component method to go to Source View and modify the method as shown in red below:

```
/**
 * @common:operation
 */
public String hello(String name)
{
    int visits = visitDB.getVisits(name) + 1;

    if(visits == 1)
    {
        visitDB.insertVisitor(name);
        return "Hello, " + name + "!";
    }
    else if(visits == 2)
    {
        visitDB.updateVisits(name, visits );
    }
}
```

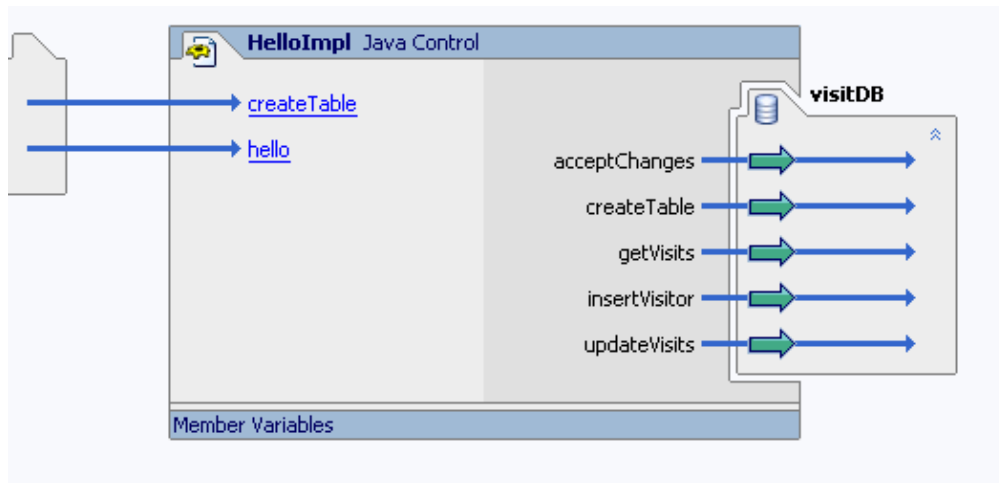
## WebLogic Workshop Tutorials

```
        return "Hello again, " + name + "!";
    }
    else
    {
        visitDB.updateVisits( name, visits );
        return "Hello, " + name + "! This is visit number " + visits + ".";
    }
}
```

8. Save your results.

In the hello method, the visitDB's getVisits method is called to determine the number of previous visits, and this number is updated by one. If the user is not known in the database, the value 0 is returned by the getVisits method, the visitor is entered into the database, and a greeting is returned. In all other cases the number of visits is updated for this known visitor, and a different greeting is returned to the invoking client application. Also, visitors who have one prior visit will be returned a different message than visitors who have multiple prior visits.

When you return to Design View, the Hello control should look like this:



### Build and Deploy

Now let's build the control project and make sure both controls are defined correctly.

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View**—>**Windows**—>**Build**.
2. In the **Application** pane, right-click MyControlProject, and select **Build MyControlProject**.
3. Monitor the **Build** window and verify that the build completes correctly
4. Open the **Libraries** folder in the **Application** pane, and notice that the file MyControlProject.jar has been recreated.

If you encounter build errors, verify that the hello and createTable methods have been created correctly.

### Related Topics

#### Building Custom Java Controls

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





## Step 4: Test the Controls

In this step you are going to test the two controls and examine whether their behavior is as intended. To do so, you must create a client application, such as a web application, to invoke the custom control. Here you will build a test web service instead.

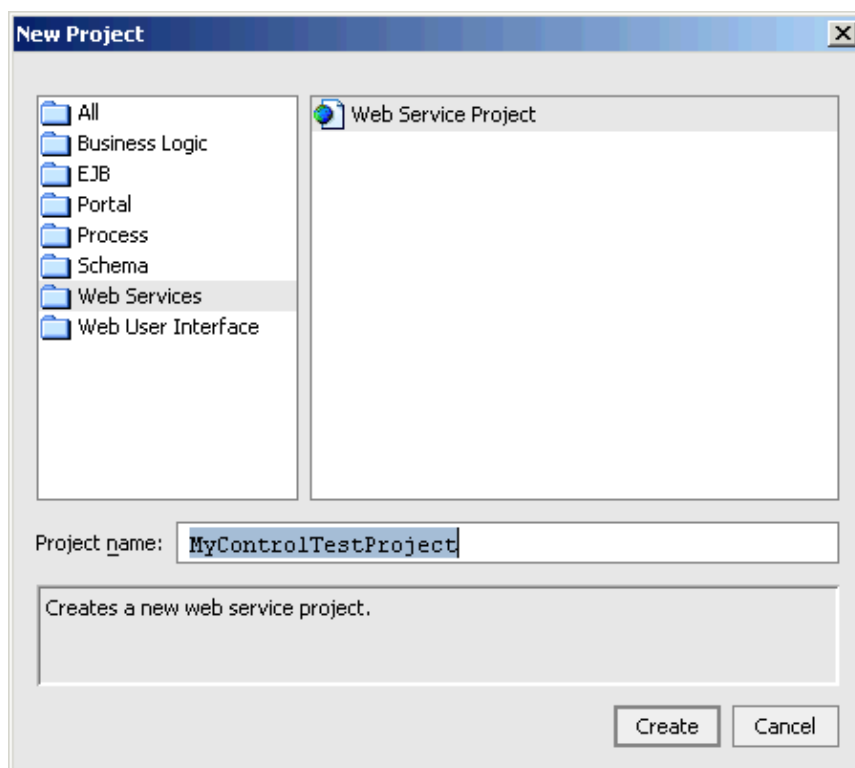
The tasks in this step are:

- To Create a Web Service Project
- To Generate the Test Web Service
- To Run the Test Web Service

To Create the Web Service Project

Web service development is done in a separate project in the application.

1. In the **Application** tab, right-click the **GettingStarted\_Control** folder and select **New—>Project**.
2. In the **New Project** dialog, in the upper left-hand pane, select **Web Services**.  
In the upper right-hand pane, select **Web Service Project**.  
In the **Project Name** field, enter MyControlTestProject.



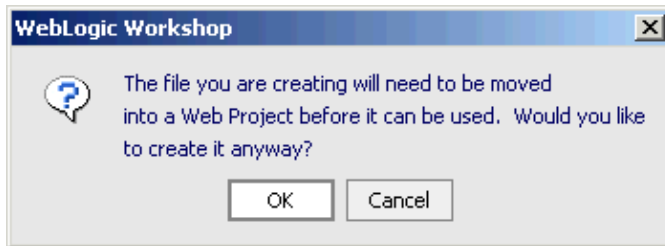
3. Click **Create**.
4. In the Application tab, right-click **MyControlTestProject** and select **New—>Folder**. Enter **hello**.
5. Click **OK**.

To Generate the Test Web Service

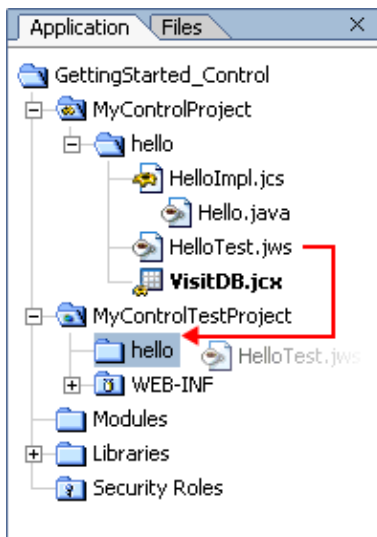
## WebLogic Workshop Tutorials

A test web service is a web service that exposes the methods of a control. In this particular case the test web service is a client application that calls the methods on the custom control Hello. To learn more about web services, see the Getting Started Tutorial: Web Services.

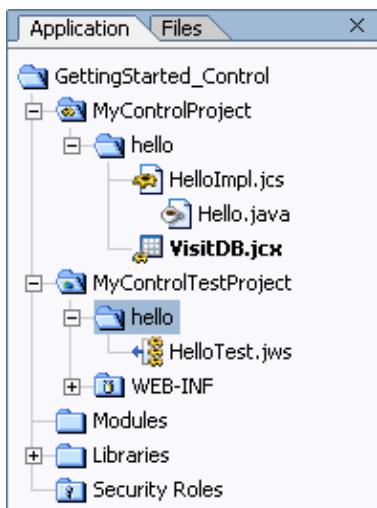
- In the **Application** tab, right-click HelloImpl.jcs and select **Generate Test JWS File (Stateless)**. The following message appears:



- Click **OK**. The file HelloTest.jws is created.
- In the **Application** tab, drag and drop HelloTest.jws to the **hello** folder in MyControlTestProject.



- The test web service HelloTest has now been moved to the web service project. The **Application** pane should look like this:



## WebLogic Workshop Tutorials

- In the **Application** tab, double-click HelloTest.jws to open the test web service in Design View



To Run the Test Web Service

Let's run the test web service and test the behavior of the Java controls.

1. Make sure the HelloTest web service is open.
2. Click the **Start** button.



Workshop launches the Workshop Test Browser.

3. If this is the first time you have ever run this application, click the **createTable** button to make the table the database control is going to query.
4. Scroll down to the **Service Response** section, and notice that no exceptions are thrown.
5. Scroll back up and click the Test Operations link.
6. Enter your name in the String field and click the **hello** button.
7. Scroll down to the **Service Response** section, and notice that the response is Hello, <your name>!

In the preceding two steps the web service invoked the hello method on the custom control. The control called the getVisits method of the VisitDB database control to find out the number of previous visits. Because you were running this test for the first time, this method returned 0, your name was added to the database using the database control's insertVisitor method, and the Hello control returned the appropriate response.

8. Click the Test Operations link, enter your name again, click the hello button, and notice that the response is Hello again, <your name>!

In this step the Hello control again invoked the getVisits method of the VisitDB database control to find out the number of previous visits. This method returned 1, the database record was updated to reflect that this was your second visit using the VisitDB's updateVisits method, and the Hello control returned the appropriate response.

9. Click the Test Operations link, enter your name again, click the hello button, and notice that the response is Hello <your name>! This is visit number 3.
10. Repeat the test with a different name and observe the outcome.
11. Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



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



# Summary: Getting Started Control Tutorial

This tutorial introduced you to the basics of working with Java Controls in WebLogic Workshop.

## Concepts and Tasks Introduced in This Tutorial

- An introduction to Java controls, and the problems Java controls solve
- What is a database control, and how do you define a database control in WebLogic Workshop?
- What is a custom control, and how do you define a custom control in WebLogic Workshop?

Now that you have finished the basic Java control tutorial, you can continue with one of the other basic tutorials to understand the other core components you can develop in WebLogic Workshop. For more information, see [The Getting Started Tutorials](#). If you want to learn more about Java control development, go to the advanced [Tutorial: Java Control](#). You can learn more about the Java control technology at [Working with Java Controls](#).

Related Topics

[The Getting Started Tutorials](#)

[Tutorial: Java Control](#)

[Working with Java Controls](#)

Click the arrow to navigate through the tutorial:



# Getting Started: Enterprise JavaBeans

## The Problem of Organizing Business Logic

When you model a real-world problem in a software application, you are faced with modeling the behavior of multiple business objects including the rules that govern their interactions. To model the business objects, you need some way to represent their properties. To model the rules, you need to be able to break the business logic into smaller subtasks that comprise a complex interaction, and you need to have some way to encapsulate these subtasks such that they can be reused to model other interactions. For instance, if your software application is an online solution for a customer buying books or CDs, you need a way to represent customers, books, CDs, credit cards, and so forth. In addition, buying a CD and buying a book are two tasks that have common subtasks, such as the use of a credit card to pay for the purchased item. You want to be able to separate this subtask and re-use it in both cases.

## The Enterprise JavaBean Solution

The Enterprise JavaBeans (EJB) technology solves this problem by providing specialized server-side components to handle data and to encapsulate business logic. Entity beans are the EJBs that specialize in representing business objects, such as customers, books, CDs, and credit cards. An entity bean interacts with a database to ensure that business object properties are stored in a persistent manner. Session beans are the EJBs that specialize in encapsulating business logic and governing the interactions between business objects. Furthermore, these EJBs operate in an EJB container on the server. The EJB container takes care of managing other processes that are crucial for an application to work successfully but are not an integral part of the application design. For example, the EJB container takes care of transaction management such that when an error occurs during the online purchase of a book, the customer does not end up being charged for a book that he will not receive. In other words, if a transaction fails, the EJB container makes sure that any changes made during that transaction (such as charging a credit card) are rolled back.

## What This Tutorial Teaches

In this tutorial, you will learn the basics of using EJBs to encapsulate business logic and represent business objects. You will create two EJBs to model the server-side logic of a simplified login procedure. The procedure consists of a user submitting a name, and receiving a greeting that is based on the number of times the user has submitted his/her name previously. To model this procedure you will create (a) an entity bean that creates records in a database regarding the number of visits by a user, and (b) a session bean that will receive the user input, query the entity bean to find out the status of this user, and return a status-specific greeting.

### Related Topics

[Getting Started with EJB Project](#)

Click the arrow to navigate to the next step.



# Getting Started: EJB Core Concepts

This topic will familiarize you with the basic concepts of EJB design and development.

## What is an EJB?

EJBs are server-side components that are optimized to handle data and execute business tasks. There are three types of EJBs: entity beans, session beans, and message-driven beans. In this tutorial you will only learn about entity beans and session beans. Message-driven beans are discussed in the advanced Tutorial: Enterprise JavaBeans.

Entity beans model business objects. For instance, a *Customer* entity bean defines its properties and has methods to get and set the customer's name, age, and so forth. In addition, the Customer bean will likely have methods to create a new customer, delete a customer and find customers with certain properties. An entity bean ensures that the properties of a business object are mapped to a database table, so that these properties can be stored, updated and retrieved. What is interesting about using an entity bean is that you do not issue the SQL commands to interact with the database. The database-specific commands are entirely done in the background by the EJB container. This for instance means that to change the customer's age in the database, you only need to change this property, for instance, by calling the `setAge(newAge)` method on the Customer bean instance; the EJB container will ensure that the corresponding record in the database is updated to reflect this change.

Session beans model business logic and are often an intermediary between your client application, such as a web application, and entity beans. For instance, a *Purchase* session bean might take an 'add to shopping cart' request for a particular product through a web page, query an entity bean to find out whether this product is still in stock, and return a response to the web application that the product is not available, or is backordered, or is available and has been successfully added to the cart. There are two types of session beans, stateless and stateful. A stateful session bean maintains conversational state. In other words, a stateful session bean remembers the calling client application from one method to the next. Stateless session beans do not maintain conversational state. Stateless session beans are more commonly used and will be addressed in this tutorial.

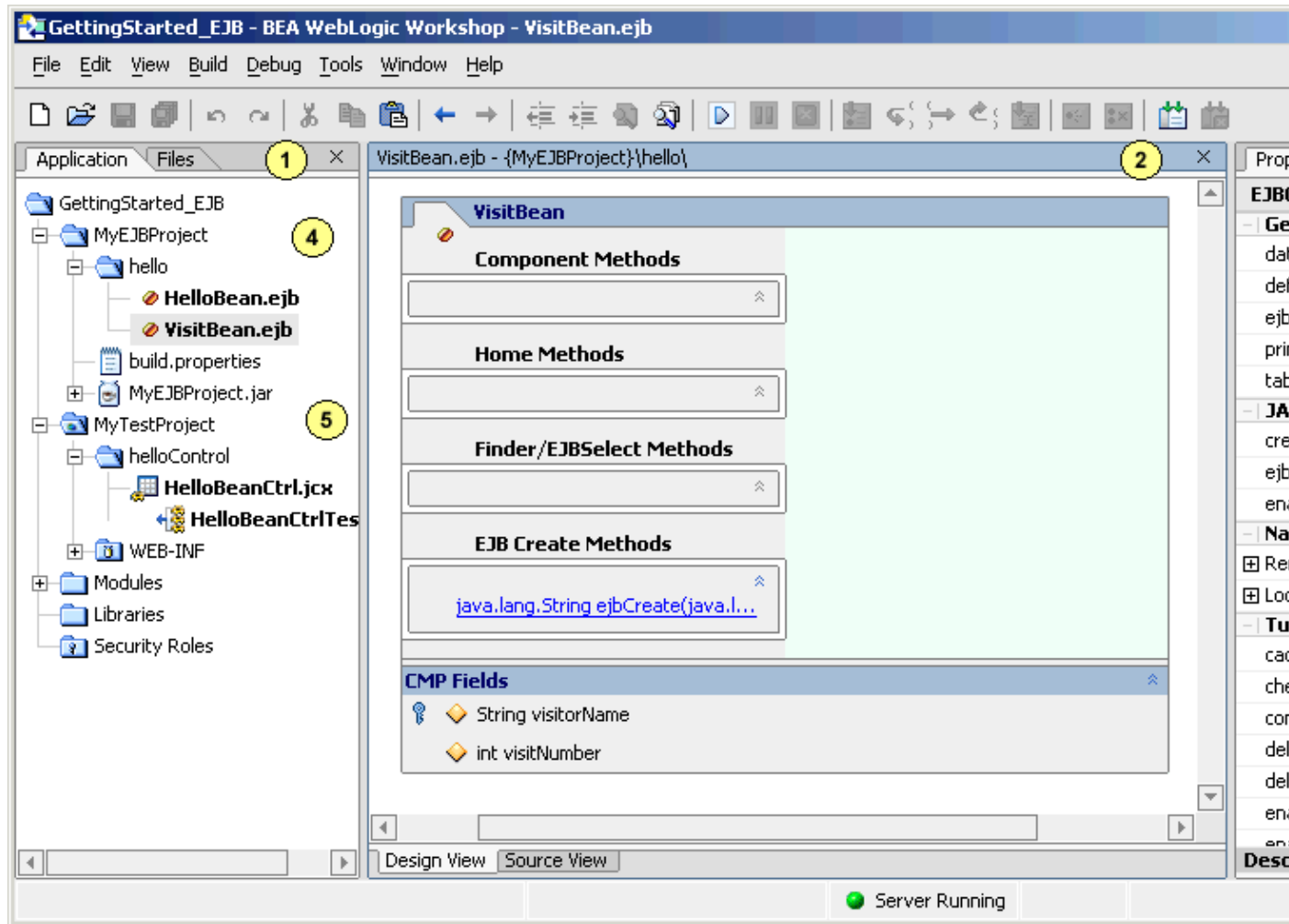
All EJBs must adhere to a certain set of design specifications (as set by the Java Community Process). For instance, all session and entity beans must have a home interface containing methods to obtain a reference to a (new or existing) bean instance. These beans must also have a business interface containing the methods to get or set data, and execute business logic. These interfaces can be local, meaning that only other objects in the same application can call these methods, remote, meaning that objects in the same and other applications can call these methods, or both.

According to the EJB design specifications, the definitions of the bean class and the various interfaces need be stored in separate Java files (and the corresponding compiled .class files), which means that a developer needs to keep the various files and definitions synchronized. In WebLogic Workshop you use only one file to develop the bean class and interfaces. When you build this file, the various Java files for the bean class and the interfaces are automatically generated for you, releasing you of the task to keep this information synchronized.

## Developing EJBs with WebLogic Workshop

EJBs are developed using WebLogic Workshop, a visual tool for designing J2EE applications. The image below shows the finished tutorial.





The **Application** tab (area 1) shows the application's source files. Enterprise JavaBeans are stored as .ejb files, which is a special Java file with an .ejb extension.

The main work area (area 2) shows the Design and Source View of the components you are building. In the above picture an entity bean is shown in Design View. You can develop EJBs in Design View with the assistance of dialogs and wizards and/or you can add source code directly to the Source View.

The **Property Editor** (area 3) allows you to see and set properties of the EJB shown in the main area as well as properties of the EJB project.

An EJB project folder holds EJBs that are developed in an application. An EJB project is like an organizational unit; when building EJBs you build all the EJB in a project and the resulting files are wrapped in a JAR file with the same name as the EJB project. You can have one or multiple EJB projects in an application. In this tutorial you will develop all EJBs using one EJB Project folder called MyEJBProject (area 4).

Other components of the application are built in other projects. In this tutorial you will build a test web service, which you will use as a client application to test the EJBs. Web services are created in a web service project folder, called MyTestProject in this tutorial (area 5).

Related Topics

## Getting Started with EJB Project

Click the arrow to navigate to the next step.



# Step 1: Create an EJB Project

In this step you will create the application that you will be working with throughout this tutorial. In WebLogic Workshop, an application contains one or more projects. You will create two projects, namely an EJB project for the Enterprise JavaBeans and, during a later step, a web service project for the test web service used to test the EJBs.

The tasks in this step are:

- To Start WebLogic Workshop
- To Create a New Application and Select a WebLogic Server Domain
- To Create a New EJB Project
- To Start WebLogic Server

To Start WebLogic Workshop

If you have an instance of WebLogic Workshop already running, you can skip this step.

... on Microsoft Windows

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

- From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**WebLogic Workshop 8.1**.

...on Linux

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

- Open a file system browser or shell.
- Locate the **Workshop.sh** file at the following address:

```
$HOME/BEA/weblogic81/workshop/Workshop.sh
```

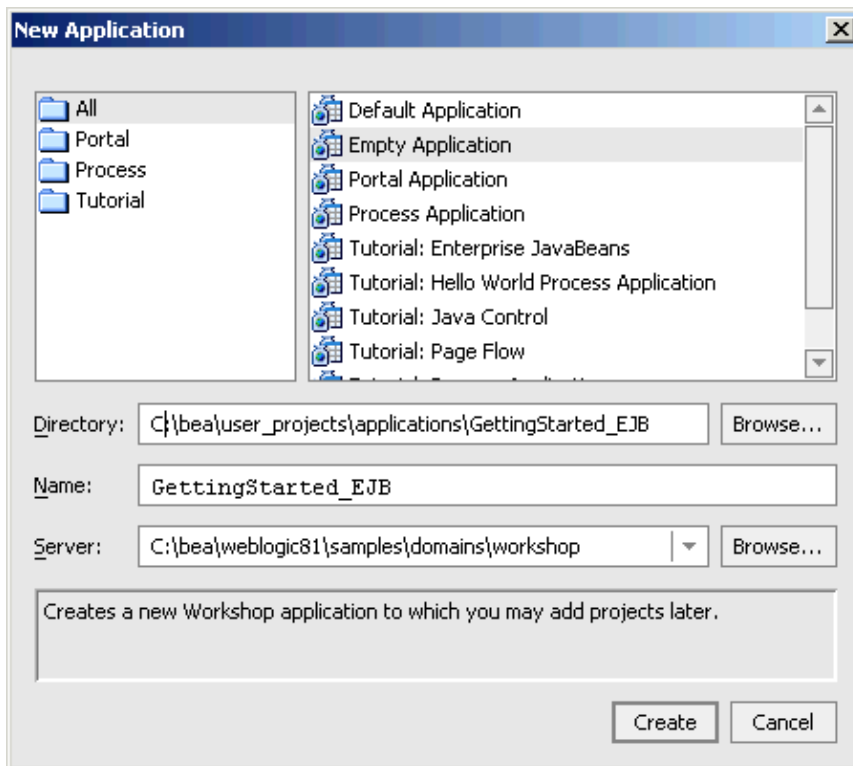
- In the command line, type the following command:

```
sh Workshop.sh
```

To Create a New Application and Select a WebLogic Server Domain

To create the EJB project, you must first create the application to which it will belong:

1. From the **File** menu, select **New**—>**Application**. The **New Application** dialog appears.
2. In the **New Application** dialog, in the upper left-hand pane, select **All**.  
In the upper right-hand pane, select **Empty Application**.  
In the **Directory** field, use the **Browse** button to select a location to save your source files. A suggested location is [BEA\_HOME]\user\_projects\applications\GettingStarted\_EJB.  
In the **Name** field, enter GettingStarted\_EJB.  
In the **Server** field, from the drop-down list, select  
BEA\_HOME\weblogic81\samples\domains\workshop.

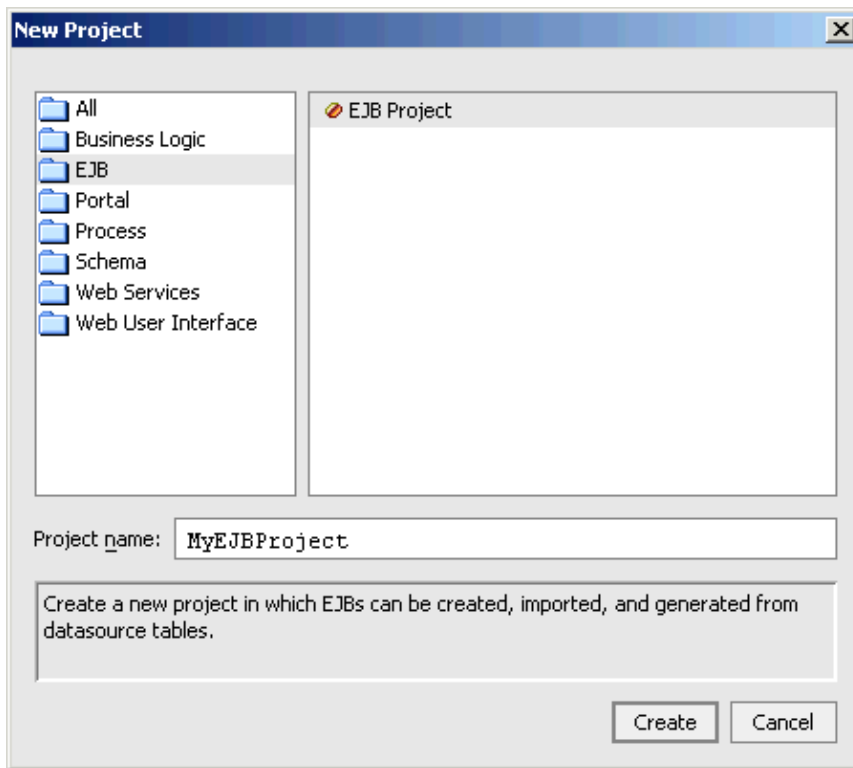


3. Click **Create**.

To Create a New EJB Project

You will need an EJB project to contain the EJBs that you are going to create. An EJB project can be thought of as a unit of work; all EJBs in a given project will be build together and will be packaged in the same JAR file.

1. In the **Application** tab, right-click the GettingStarted\_EJB folder and select **New**—>**Project**.
2. In the **New Project** dialog, in the upper left-hand pane, confirm that **EJB** is selected.  
In the upper right-hand pane, select **EJB Project**.  
In the **Project Name** field, enter MyEJBProject.

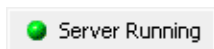


3. Click **Create**.

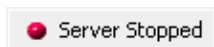
#### To Start WebLogic Server

Since you will be deploying and running your EJBs on WebLogic Server, it is helpful to have WebLogic Server running during the development process.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.

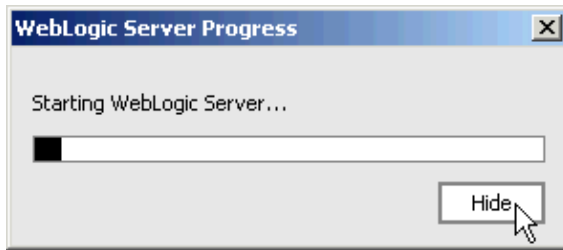


If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow these instructions to start WebLogic Server:

- From the **Tools** menu, choose **WebLogic Server-->Start WebLogic Server**.
- On the **WebLogic Server Progress** dialog, you may click **Hide** and continue to the next task.



Related Topics

[Getting Started with EJB Project](#)

[How Do I...? EJB Topics](#)

[The WebLogic Workshop Development Environment](#)

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



## Step 2: Define the Entity Bean

In this step you will create the entity bean *Visit*, which will hold information about visitors and the number of prior visits. An entity bean interacts with a database table to store, find, and update information.

The tasks in this step are:

- To Create a Package
- To Create a Entity Bean
- To Change the Database Table
- To Define CMP Fields
- To Add EJBCreate
- Build and Deploy

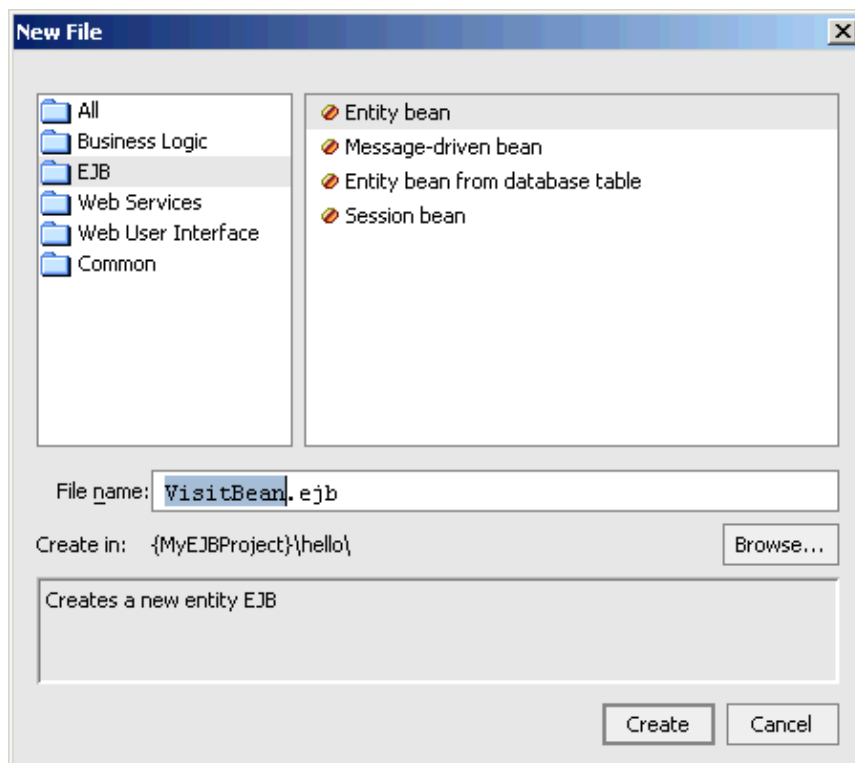
To Create a Package

All EJBs (and all Java classes) must be part of a Java package.

1. Right-click the **MyEJBProject** folder in the **Application** tab, and select **New**—>**Folder**.
2. Enter the folder name *hello*.
3. Click **OK**.

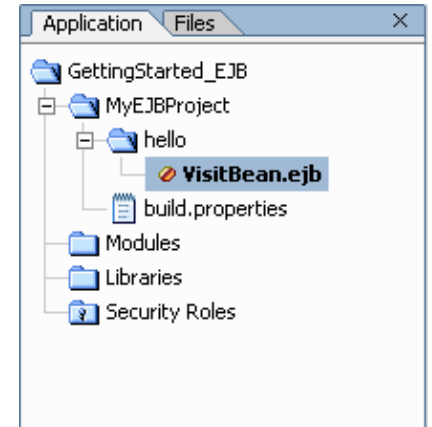
To Create an Entity Bean

1. Right-click the **hello** folder in the **Application** tab, and select **New**—>**Entity Bean**.
2. Enter the file name *VisitBean.ejb*.



3. Click **Create**.

The entity bean opens in Design View. The **Application** tab should now look like the picture on the right.



The top-level folder *GettingStarted\_EJB* is the containing folder for the entire application. All of the source files for your application exist in this folder.

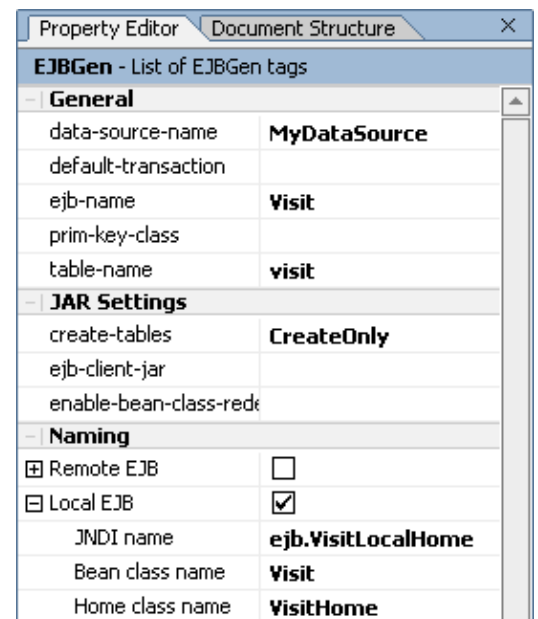
The folder *MyEJBProject* is the EJB project folder. An application can contain any number of project folders and there are many different kinds of project folders, including Web projects, Web Services projects, Schema projects, and so forth.

The **Modules** folder is for the storage of stand-alone applications (packaged as WAR and JAR files) that can be deployed parallel with your web service, if you wish.

The **Libraries** folder is for the storage of resources that are to be used across multiple projects, provided the resources are packaged as JAR files. For example, if you had a control or an EJB that you wanted to re-use across all your projects, you would store it in the Libraries folder.

The **Security Roles** folder lets you define security roles and test users for your web application. You can test the security design of your web application by logging in as a test user.

To Change the Database Table





When a new entity bean is created, a number of properties are automatically set for you. These properties are listed in the **Property Editor**. For example, notice that the *ejb-name* is *Visit*, the *data-source-name* is *MyDataSource*, the name of the database table used to store the data is *visit*, and that the *create-tables* setting is set to *CreateOnly*. The latter setting means that when you build and deploy this EJB, the table will be automatically created for you. To learn more about the other table creation options set with this property, see `@ejbgen:jar-settings` Annotation. Notice that the *create-tables* setting is part of the JAR settings. In other words, for every entity bean created as part of this project, the same table creation setting applies.

Also notice that the names of the bean's local interfaces are already set. The home interface class, set in the Home class name property, is called *VisitHome*, while the business interface, set in the Bean class name property, is *Visit*. Finally, the JNDI name, used to locate the entity bean, is *ejb.VisitLocalHome*. JNDI naming is discussed in detail in the next step. To learn more about the entity bean architecture, see *Getting Started with Entity Beans*.

Let's use the **Property Editor** to make two changes. First, the data source name for the Visit bean does not match the name of the data source that is used in the workshop domain. Second, let's rename the database table to better reflect that it is part of this tutorial:

1. From the Tools menu, select **WebLogic Server**—>**DataSource Viewer**. The **DataSource Viewer** dialog appears.
2. Notice that one of the data sources is called *cgSampleDataSource*. Click **OK**.

**Note.** If you cannot find *cgSampleDataSource*, verify that the correct workshop domain was selected in Step 1 of the tutorial.

3. In the **Property Editor**, locate *data-source-name* and change the entry to *cgSampleDataSource*.
4. In the **Property Editor**, locate *table-name* and the table name to *visit\_GS\_EJB*.

When you build and deploy the Visit bean, the table with the table name *visit\_GS\_EJB* will be created for you, and the bean will interact with this database table to persist data about visits.

### To Define CMP Fields

Now you define the container-managed persistence (CMP) fields for the Visit bean. CMP fields are virtual fields that are not defined in the entity bean itself but are mapped to columns in the database table. The entity bean code only implements the accessor (getter and setter) methods to read and write the data stored in these fields. In WebLogic Workshop these accessor methods are by default added to the bean's local business interface for you.

You need to define two fields that hold the name of the visitor and the number of visits. The name of the visitor will also be used as the primary key. The primary key is the unique index in a database table. In other words, you make the assumption that each visitor's name is unique and that you can use it to uniquely identify what stored data about the number of visits belongs to him/her.

1. Ensure that the Visit bean is displayed in Design View.
2. Right-click the VisitBean and choose **Add CMP field**. A CMP field called String field1 appears in the **CMP Fields** section.
3. Rename the field String visitorName. If you step off the field and need to rename it, right-click the field and select **Change declaration**.
4. Right-click the field String visitorName and select **Primary Key**. A key icon appears in front of the field definition. The yellow diamond indicates that the method *getVisitorName* and *setVisitorName* will be declared in the local business interface.

5. Make sure the visitorName declaration in Design View is still selected and verify in the **Property Editor** tab, which now displays the properties for this field, that the database **column** is *visitorName*. If this is not the case, make this change now.
6. Right-click the VisitBean again, and choose **Add CMP field**. A CMP field called String field1 appears in the **CMP Fields** section.
7. Rename the field int visitNumber. If you step off the field and need to rename it, right-click the field and select **Change declaration**.
8. Select the visitNumber declaration in Design View and verify in the **Property Editor** tab that the database **column** is *visitNumber*. If this is not the case, make this change now.
9. Save your results.
10. Go to source view and examine the code. Notice that the getter and setter methods have been defined for these properties but that there are no variables declared in the bean class to hold the actual property values.

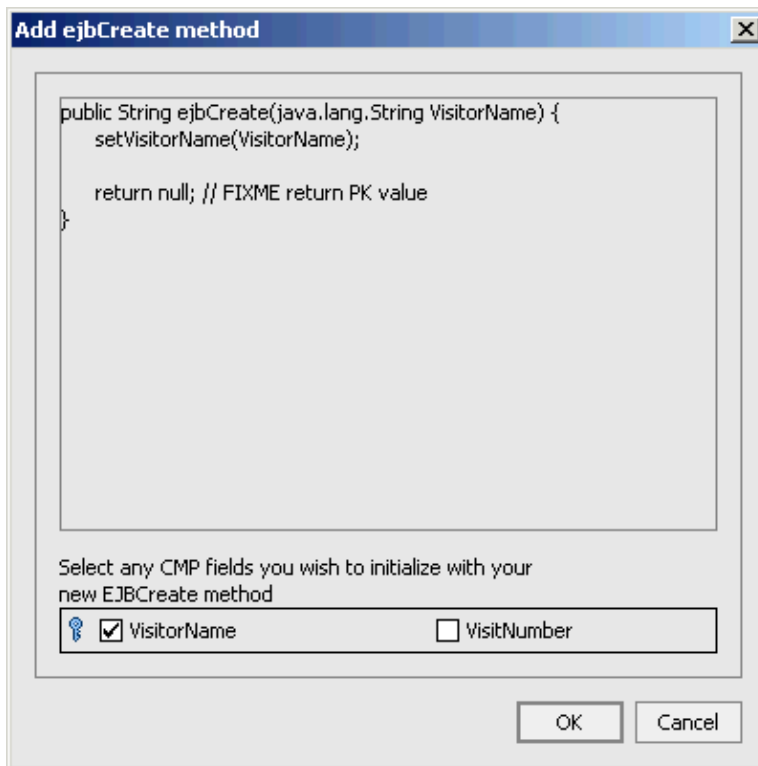
Now you have defined two CMP fields and their accessor methods. Also, you have mapped these fields to columns with the same names in the database table. (The name of the CMP field and the corresponding column name can be different, but for consistency reasons you have given them the same name here.) Finally, you have assigned one of these fields to function as the primary key.

### To Add EJBCreate

Now you will add an ejbCreate method, which when invoked creates a new Visit bean and adds the corresponding record to the database table. To create a new instance, you must minimally provide the primary key to unique identify the record.

In addition to creating the ejbCreate method in the bean class, a corresponding create method must be defined in the bean's home interface. When you follow the steps below to define the ejbCreate method, WebLogic Workshop automatically defines the corresponding create method in the home interface for you.

1. Ensure that the Visit bean is displayed in Design View.
2. Right-click the Visit bean and choose **Add EJBCreate**. The **Add ejbCreate method** dialog appears.



3. Click **OK**. The method you created appears in the *EJB Create Methods* section.
4. Click the ejbCreate method to go to Source View, and add the following line shown in red:

```
public java.lang.String ejbCreate(java.lang.String VisitorName)
{
    setVisitorName(VisitorName);
    setVisitNumber(1);

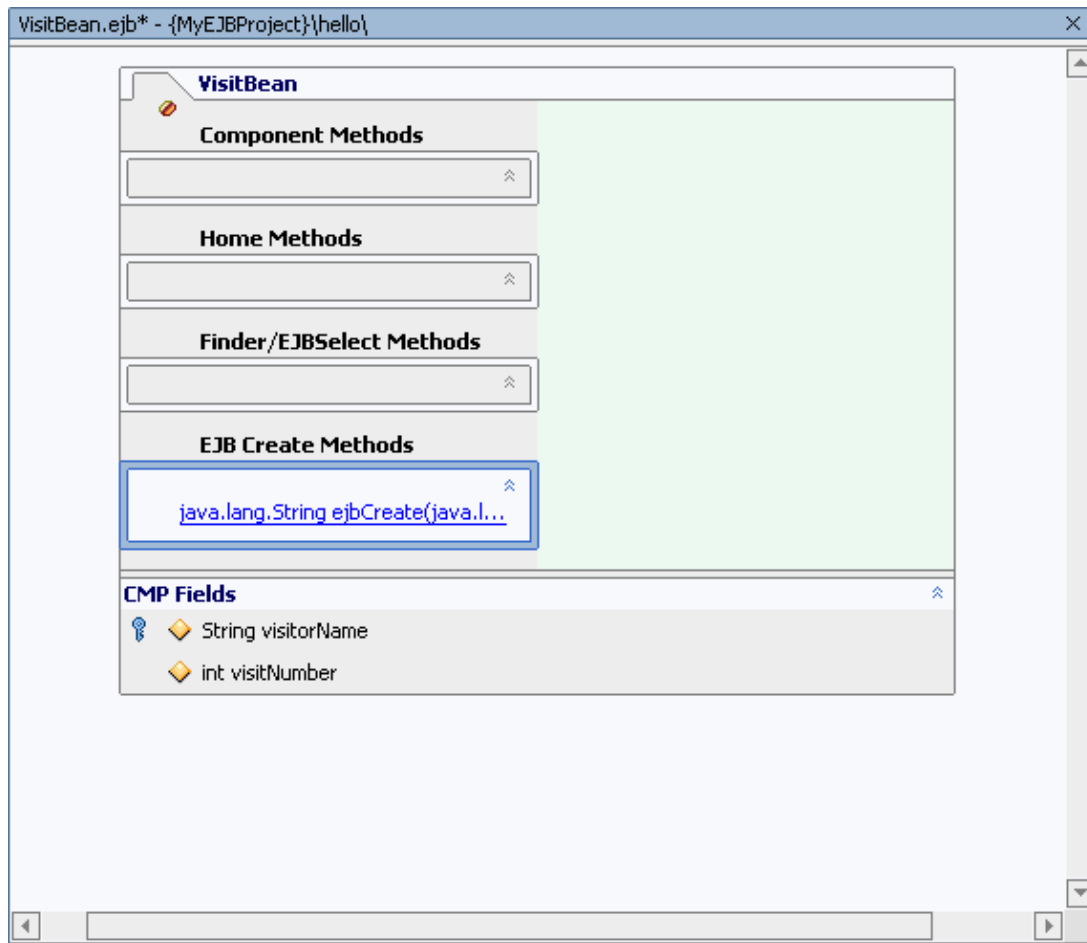
    return null; // FIXME return PK value
}
```

5. Save your results.

When a new Visit bean instance is created using this ejbCreate method, the name of the visitor, passed as an argument in the method, is entered in the database. Also, it is noted that this is the visitor's first visit. In the next step of the tutorial you will see how to invoke this method.

An entity bean can have multiple ejbCreate methods. For instance, you could create another ejbCreate method for the Visit bean that would take a String name and an int number as arguments, and use these to set the name and visit number for a new bean instance. Each ejbCreate method must be unique, that is, it must have a unique set of parameters as arguments. For instance, for the Visit bean you cannot create a second ejbCreate method that has a String parameter as its sole argument.

When you go back to Design View, the Visit bean should look like this:



### Build and Deploy

Now let's build the entity bean to make sure it is defined correctly. If the Visit bean builds successfully, it will be automatically deployed on the server.

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View**—>**Windows**—>**Build**.
2. In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is created. You will see this JAR appear in the MyEJBProject folder.
4. After the build completes, the EJB deploys: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The EJB is now deployed on the server.

If you encounter build errors, verify that the various methods and interfaces have been created correctly. If you encounter a deployment error, make sure that the data source and database table settings are set correctly.

Optionally you can expand MyProjectEJB.jar and the contained hello folder. In this folder you will see (among other files) the Java classes for the bean and its interfaces, and the corresponding compiled .class classes.

### Related Topics

### Getting Started with Entity Beans

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



## Step 3: Define the Session Bean

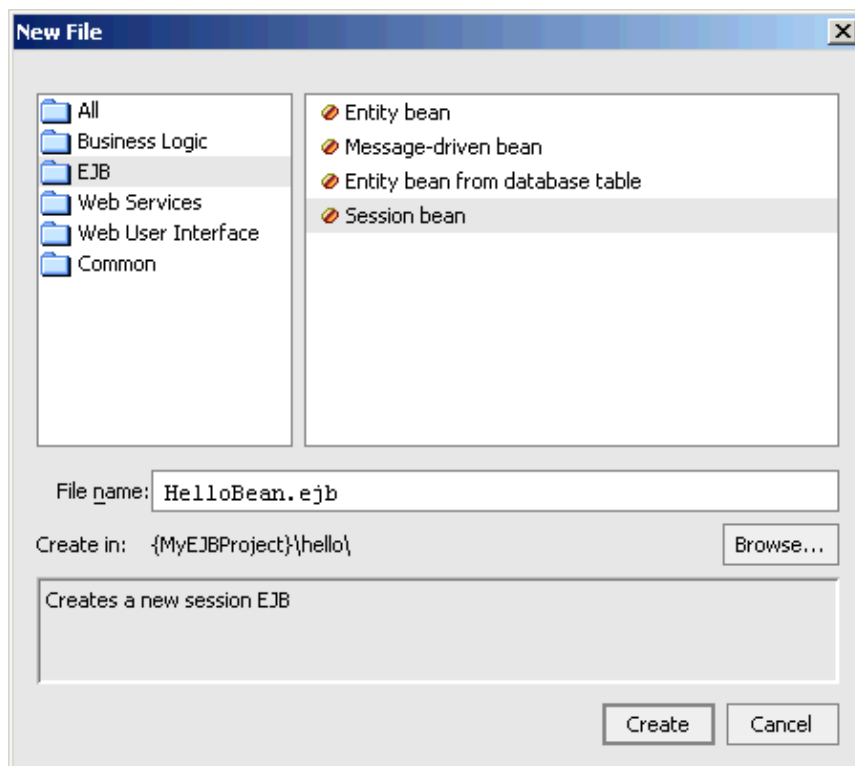
In this step you will create the (stateless) session bean *Hello*. The session bean will have one method, *hello*, which is invoked by a client application and takes a visitor's name as an argument. The session bean in turns calls the Visit entity bean to determine whether this visitor is known, and if so, how many times the visitor has visited before, and returns an appropriate greeting to the client application. In other words, the session bean implements the business logic that governs a particular interaction.

The tasks in this step are:

- To Create a Session Bean
- To Reference the Entity Bean
- To Add an `ejb-local-ref` Tag
- To Add a Component Method
- Build and Deploy

To Create a Session Bean

1. Right-click the *hello* folder in the *Application* tab, and select *New*--->*Session Bean*.
2. Enter the file name `HelloBean.ejb`.



3. Click *Create*.

The file `HelloBean.ejb` should now appear in the **Application** tab above `VisitBean.ejb` in the *hello* folder. Also notice that the bean opens in Design View, and that the **Property Editor** shows that a number of properties have been automatically set. In particular notice that the remote business interface, set in the Bean class name property, is `Hello`, the remote home interface, set in the Home class name property, is `HelloHome`, and that the remote JNDI name is `ejb.HelloRemoteHome`. JNDI naming is discussed in more detail below.

Naming	
<input checked="" type="checkbox"/> Remote EJB	<input checked="" type="checkbox"/>
JNDI name	<b>ejb.HelloRemoteHome</b>
Bean class name	<b>Hello</b>
Home class name	<b>HelloHome</b>
<input type="checkbox"/> Local EJB	<input type="checkbox"/>

When you go to Source View, you will notice that an `ejbCreate` method with no arguments has already been defined. This method is called to create a new session bean instance. A stateless session bean, like the `Hello` bean, will always have this `ejbCreate` method. Unlike entity beans, a stateless session bean cannot have additional `ejbCreate` methods.

### To Reference the Entity Bean

The `Hello` bean will use the `Visit` bean to determine if a visitor is new or returning. In order for the `Hello` bean to invoke the `Visit` bean's methods, it must first locate and obtain a reference to the bean's home interface. To do so, you modify the `Hello` bean's `ejbCreate` method to include code that obtains this reference.

1. Select the Source View tab.
2. Locate the `ejbCreate` method in `HelloBean.ejb`, and modify its definition as shown in red below:

```
public void ejbCreate() {
    try {
        javax.naming.Context ic = new InitialContext();
        visitHome = (VisitHome)ic.lookup("java:comp/env/ejb/MyVisitBean");
    }
    catch (NamingException ne) {
        throw new EJBException(ne);
    }
}
```

3. Locate the import section at the top of the file and add the following import statements, or use Alt+Enter when prompted by the IDE to automatically add these import statements:

```
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

4. Go to the beginning of the class definition and define `visitHome` as shown in red below:

```
public class HelloBean
    extends GenericSessionBean
    implements SessionBean
{
    private VisitHome visitHome;

    public void ejbCreate() {
        ...
    }
}
```

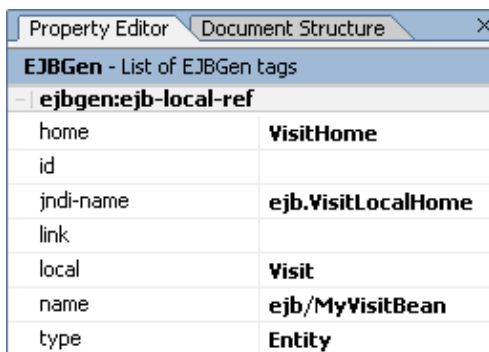
When the `ejbCreate` method executes, the `Visit` bean is looked up using JNDI API, a standard naming and directory API. This API is described in more detail below. (More information can also be found in many books on J2EE and at <http://java.sun.com>.) Notice that the `lookup` method returns a reference to the local home interface of the `Visit` bean, which is called `VisitHome` (as you saw in the previous step), and stores this reference in the `visitHome` variable.

## To Add an ejb-local-ref Tag

The Java Naming and Directory Interface (JNDI) API provides a set of methods to map an easy-to-remember name to the actual location of an object. It works essentially the same as other naming services you are already familiar with, such as the telephony naming system. A phone directory lists phone numbers, which are easy-to-remember names to call certain telephones. Each phone number is mapped to an actual phone somewhere, such that when you call the number the appropriate phone starts ringing. The telephone company does this mapping for you; all you need to remember is the telephone number and the context in which this phone number is relevant, meaning whether you are calling locally and don't need an area code, whether you do need an area code, or whether you are calling internationally and need a country code too.

In the above ejbCreate method the name ejb/MyVisitBean is used to refer to the Visit bean. To make the lookup procedure successful, you must map this name to the actual EJB location, in this case the Visit bean. This mapping is done in an XML file called the deployment descriptor. However, in WebLogic Workshop you don't modify the deployment descriptor directly but use ejbgen tags inserted in the ejb file instead. To map this name to the actual Visit bean definition, you need to insert an `ejbgen:ejb-local-ref` tag.

1. Ensure that the Hello bean is displayed in Design View.
2. Right-click the Design View, and select **Insert EJB Gentag**—>**ejb-local-ref**. A new `ejbgen:ejb-local-ref` section appears in the **Property Editor**.
3. In the Property Editor, make the following changes:



EJBGen - List of EJBGen tags	
<b>ejbgen:ejb-local-ref</b>	
home	<b>VisitHome</b>
id	
jndi-name	<b>ejb.VisitLocalHome</b>
link	
local	<b>Visit</b>
name	<b>ejb/MyVisitBean</b>
type	<b>Entity</b>

By adding this tag, you specify that the reference ejb/MyVisitBean as used in the Hello bean's code maps to an entity bean with the JNDI name ejb.VisitLocalHome, the local business interface name Visit, and the local home interface VisitHome. (Remember that you defined these names for the entity bean in the previous step.) Notice that this mapping uses another JNDI name reference, ejb.VisitLocalHome, which is the Visit bean reference used in the entire EJB container (the latter reference is automatically mapped for you). This concatenation of mapping is similar to the concatenation of country code, area code, and local phone number in the telephony naming system.

The ejb/MyVisitBean reference is only valid within the context of the Hello bean. If you were to create another bean that refers to the Visit bean, you might name that reference ILikeThisVisitBean, and map this name to the Visit bean location using a `ejbgen:ejb-local-ref` tag in the new bean's definition

## To Add a Component Method

Now you will implement the business logic to receive a client's request and return a response.

1. Ensure that the Hello bean is displayed in Design View.



2. Right-click the Hello bean and choose **Add Component Method**. A component method called void method() appears in the **Component Methods** section.
3. Rename the method `String hello(String visitorName)`. If you step off the method and need to rename it, right-click the method and select **Rename**.
4. Click the component method to go to Source View and modify the method as shown in red below:

```

/**
 * @ejbgen:remote-method
 */
public String hello(String visitorName)
{
    Visit theVisit;
    int visitNumber;

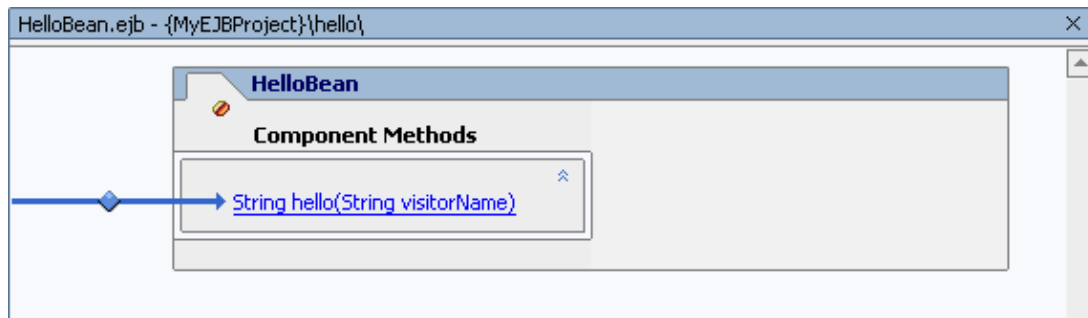
    try {
        // Find the visitor in the database
        theVisit = visitHome.findByPrimaryKey(visitorName);
    }
    catch(FinderException fe) {
        try {
            // Name of visitor not found. Enter the visitor in the database
            visitHome.create(visitorName);
            return "Hello, " + visitorName + "!";
        }
        catch(CreateException ce) {
            throw new EJBException(ce);
        }
    }
    // visitor was found. Find the number of previous visits
    visitNumber = theVisit.getVisitorNumber();
    theVisit.setVisitorNumber(visitNumber + 1);
    if(visitNumber == 1) {
        return "Hello again, " + visitorName + "!";
    }
    else {
        return "Hello, " + visitorName + "! This is visit number " + theVisit.getVisi
    }
}

```

5. Save your results.

In this method, you use the reference stored in the `visitHome` variable to find the record corresponding to the visitor name using the `findByPrimaryKey` method. You did not explicitly define this method for the `Visit` bean, because this method was automatically generated for you and defined in the bean's home interface by the EJB container. If the name of the visitor cannot be found, a `FinderException` will be thrown and caught. When caught, a new record will be created for this new visitor and a welcoming message is returned. If the visitor's name was found in the database, his/her number of previous visits is obtained, the number of visits is increased by one, and a message is returned to the invoking client application. Visitors who have one prior visit will be returned a different message than visitors who have multiple prior visits.

When you return to Design View, the Hello bean should look like this:



### Build and Deploy

Now let's build both beans to make sure they are defined correctly. Again, once compiled these beans will be automatically deployed on the server.

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View—>Windows—>Build**.
2. In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is recreated. You will see this Jar disappear and re-appear in the MyEJBProject folder.
4. After the build completes, the EJB deploys: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The EJB is are deployed on the server.

If you encounter build errors, verify that the hello method has been created correctly.

### Related Topics

#### Getting Started with Session Beans

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



## Step 4: Test the EJBs

In this step you are going to test the EJBs and examine whether their behavior is as intended. Because EJBs are server-side components, you must first create a client application, such as a web application. Here you will build a test web service instead. This requires generating an EJB control for the EJB, and generating a test web service for the EJB control.

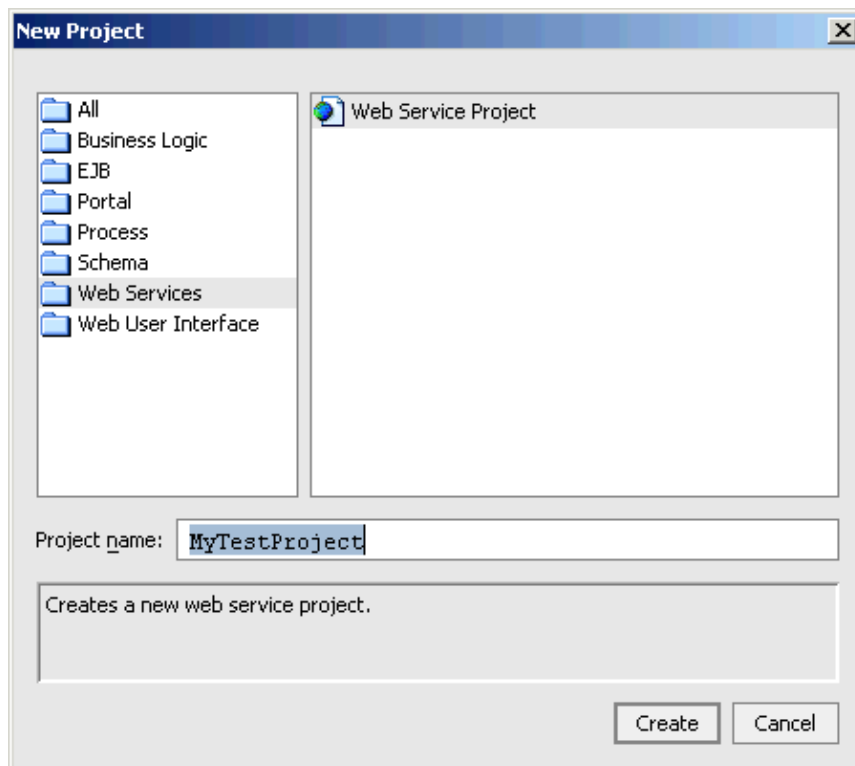
The tasks in this step are:

- To Create a Web Service Project
- To Create the EJB Control
- To Generate the Test Web Service
- To Run the Test Web Service

To Create the Web Service Project

Web service development is done in a separate project in the application.

1. In the *Application* tab, right-click the *GettingStarted\_EJB* folder and select *New-->Project*.
2. In the *New Project* dialog, in the upper left-hand pane, select *Web Services*.  
In the upper right-hand pane, select *Web Service Project*.  
In the *Project Name* field, enter MyTestProject.



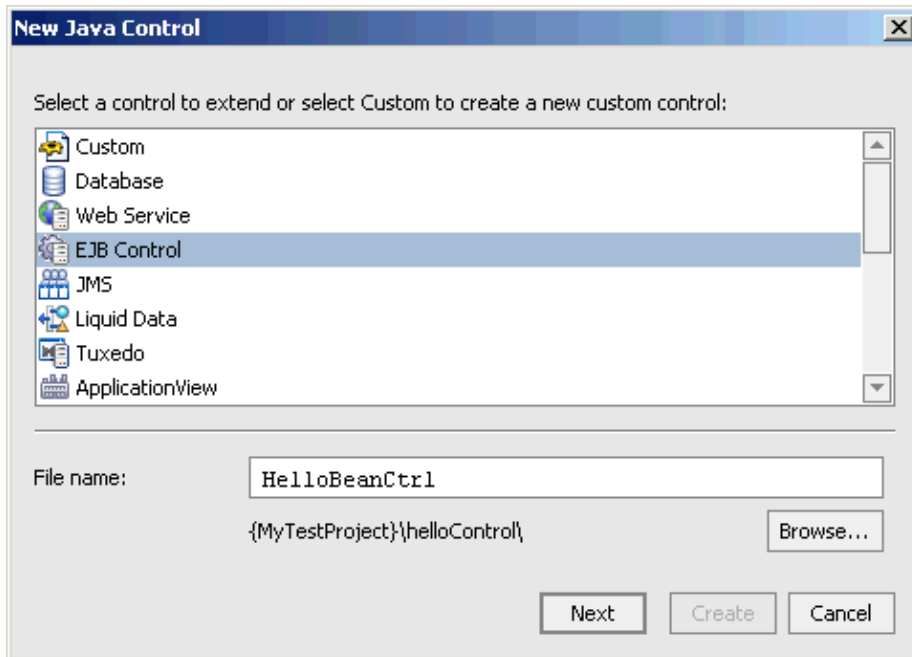
3. Click *Create*.

To Create the EJB Control

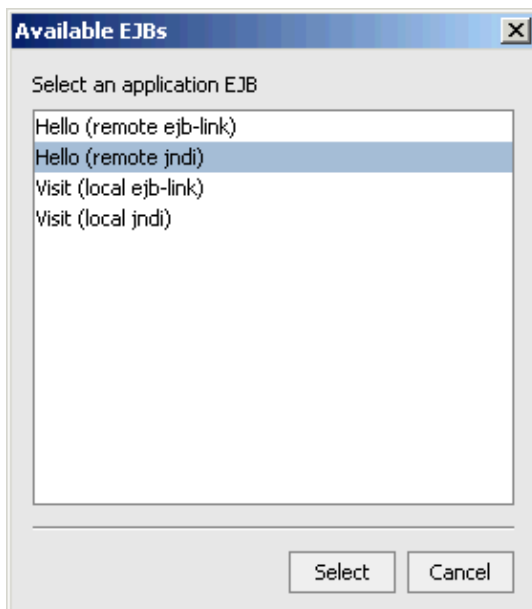
## WebLogic Workshop Tutorials

When you want to invoke a session bean from a client application, you must first look up the session bean in the EJB container's JNDI registry and obtain a reference to the EJB's home interface, before you can create an EJB instance and invoke the EJB's business methods. You can do this by calling the standard JNDI API methods, or you can use EJB controls that do this for you. In this tutorial you will create an EJB control for the Hello session bean.

1. Right-click MyTestProject and select **New**—>**Folder**. The **Create New Folder** dialog appears.
2. Enter helloControl. Click **OK**.
3. Right-click helloControl and select **New**—>**Java Control**. The **New Java Control** dialog appears.
4. Select EJB Control. In the **File name** field, enter HelloBeanCtrl.



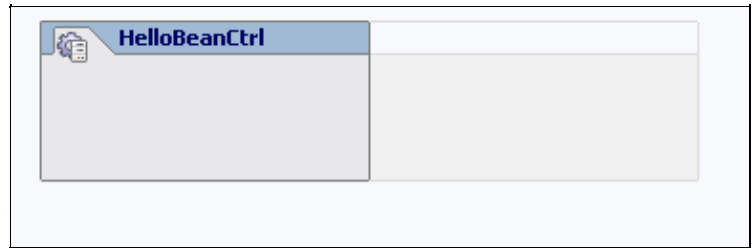
5. Click **Next** to go to the next page
6. Click **Browse application EJBs** and select Hello (remote jndi).



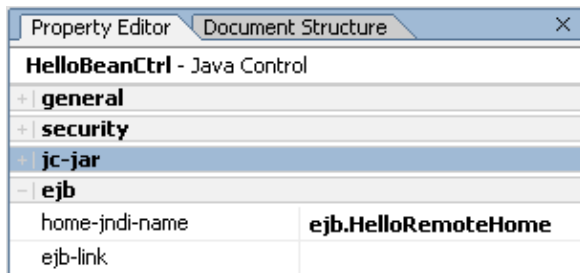
7. Click **Select**.

8. Click **Create**.

The EJB control HelloBeanCtrl is created and appears in Design View. Notice that there are no methods defined in the EJB Control. An EJB control is only used to obtain a reference to the EJB. It does not define methods of its own, but simply provides a window to the EJB it exposes. To see exactly which methods are exposed, you need to use the EJB control in a client application such as the test web service that you will generate next.



If you examine the **Property Editor**, you will see that the EJB control uses the Hello bean's remote JNDI name `ejb.HelloRemoteHome` to reference its remote interfaces. Remember that this JNDI name was defined when you created the session bean and can be used to look up the EJB within the context of the EJB container.

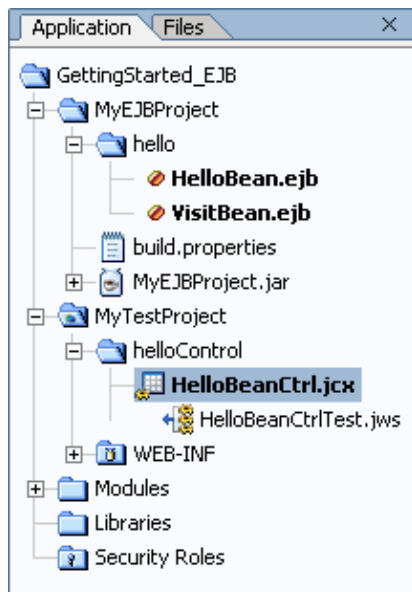


To learn more about EJB controls, see the help topic EJB Controls. To learn more about Java controls in general, see Getting Started Tutorial: Java Controls

To Generate the Test Web Service

- In the **Application** tab, right-click `HelloBeanCtrl.jcx` and select **Generate Test JWS File (Stateless)**

The test web service `HelloBeanCtrlTest` is generated. The **Application** pane should now look like this:



- Double-click HelloBeanCtrlTest.jws to open it.

A test web service is a web service that exposes the methods of a Java control. In this particular case the test web service is a client application that calls the methods on the EJB control `helloBeanCtrl`, which in turn exposes the methods of the Hello bean. Remember that the Hello bean has two methods, namely the `create` method to obtain a reference to a Hello bean instance, and the component method `hello`. When you use an EJB control, you do not have to call the `create` method first to obtain a reference, because the EJB control will handle this automatically for you. Instead you can simply invoke the `hello` method directly. To learn more about web services, see the Getting Started Tutorial: Web Services.



### To Run the Test Web Service

Let's run the test web service and test the behavior of the EJBs.

1. Make sure the HelloBeanCtrlTest Web Service is open.
2. Click the **Start** button.



Workshop launches the Workshop Test Browser.

3. Enter your name in the String field and click the `hello` button.
4. Scroll down to the **Service Response** section, and notice that the response is Hello, <your name>!

In the preceding two steps the web service invoked the **hello** method on the Hello bean via the EJB control. The Hello bean invoked the `findByPrimary` method of the Visit bean to find out if your name is stored in the database. Because you are running this test for the first time, your name could not be found. A record with your name was created in the database and the Hello bean sent the appropriate response.

5. Click the Test Operations link.
6. Enter your name again, click the **hello** button, and notice that the response is Hello again, <your name>!

In this step the Hello bean again invoked the `findByPrimary` method of the Visit bean to find out if your name is stored in the database. Your name was found, the database record was updated to reflect that this was your second visit, and the Hello bean sent the appropriate response.

7. Click the Test Operations link, enter your name again, click the **hello** button, and notice that the response is Hello <your name>! This is visit number 3.
8. Repeat the test with a different name and observe the outcome.
9. Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



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



# Summary: Getting Started EJB Tutorial

This tutorial introduced you to the basics of building Enterprise JavaBeans with WebLogic Workshop.

## Concepts and Tasks Introduced in This Tutorial

- An introduction to the J2EE and EJB architecture, and the problems EJBs solve
- What are entity beans, and how do you develop these in WebLogic Workshop?
- What are session beans, and how do you develop these in WebLogic Workshop?
- The use of ejbgen tags to do all the development of an EJB in one file
- Understanding JNDI naming services, the JNDI API, and EJB controls

Now that you have finished the basic EJB tutorial, you can continue with one of the other basic tutorials to learn more about the other core components you can develop in WebLogic Workshop. For more information, see The Getting Started Tutorials. If you want to learn more about EJB development, go to the advanced Tutorial: Enterprise JavaBeans. You can learn more about the EJB technology at Developing Enterprise JavaBeans.

Related Topics

The Getting Started Tutorials

Tutorial: Enterprise JavaBeans

Developing Enterprise JavaBeans

Click the arrow to navigate through the tutorial:





# Getting Started: Portals

## The Problem of Organizing Web Content

Suppose your company needs to manage a suite of different web applications: a product catalog for customers, a human resources site for employees, and an information site for business partners. The different applications need to be organized so that users can easily navigate the content and the applications need to be personalizable so that different kinds of users can access different kinds of content. Customers need to access the product catalog, but they shouldn't be allowed to browse the employee intranet.

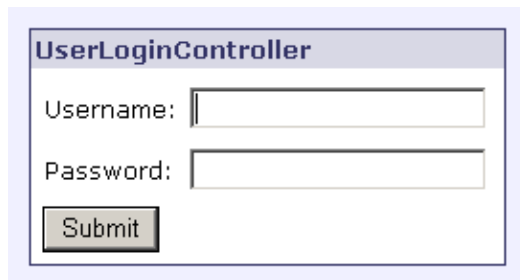
## The Portal Solution

Portals solve this problem by giving you a way to organize all your web applications and web-based content into a single web site. Portals give you a way to visually arrange different applications into a single web site. Using tabs and windows, and other navigational devices, you can arrange disparate content into a visual whole that is easy to navigate. Portals also let you personalize web content. Portal's personalization features allow you (1) to filter content depending on the characteristics of the user, (2) to make applications respond to individual patterns of use, and (3) to allow users to personalize the look and feel of the Portal.

## What This Tutorial Teaches

In the following tutorial you will learn the basics of organizing and personalizing web content using Weblogic Portal. You will create a Portal that surfaces two applications: one application allows users to login into the Portal, the other application allows users to select a favorite color.

### *The Login Application*



**UserLoginController**

Username:

Password:

### *The Favorite Color Application*



**favoriteColorController**

Your favorite color is Red:

Choose a favorite color:

You will also use Portal personalization features to remember the user's favorite color while the user is logged out of the Portal.

The tutorial is divided into three steps:

- In the first step you will create an empty Portal.
- In the second step you will populate the Portal with application content and layout the Portal's navigation scheme. You will learn the basics of organizing Portal content and navigation.
- In the third step you will learn how to make different application work together in a Portal. You will make the favorite color application accessible only to logged in users. You will also learn how to access the data in a user profile and utilize that data to personalize a web application.

## Related Topics

Portal Overview

Click the arrow below to navigate through the tutorial:



# Getting Started: Portal Core Concepts

This topic will familiarize you with the basic concepts of Portal design and development.

## What is a Portal?

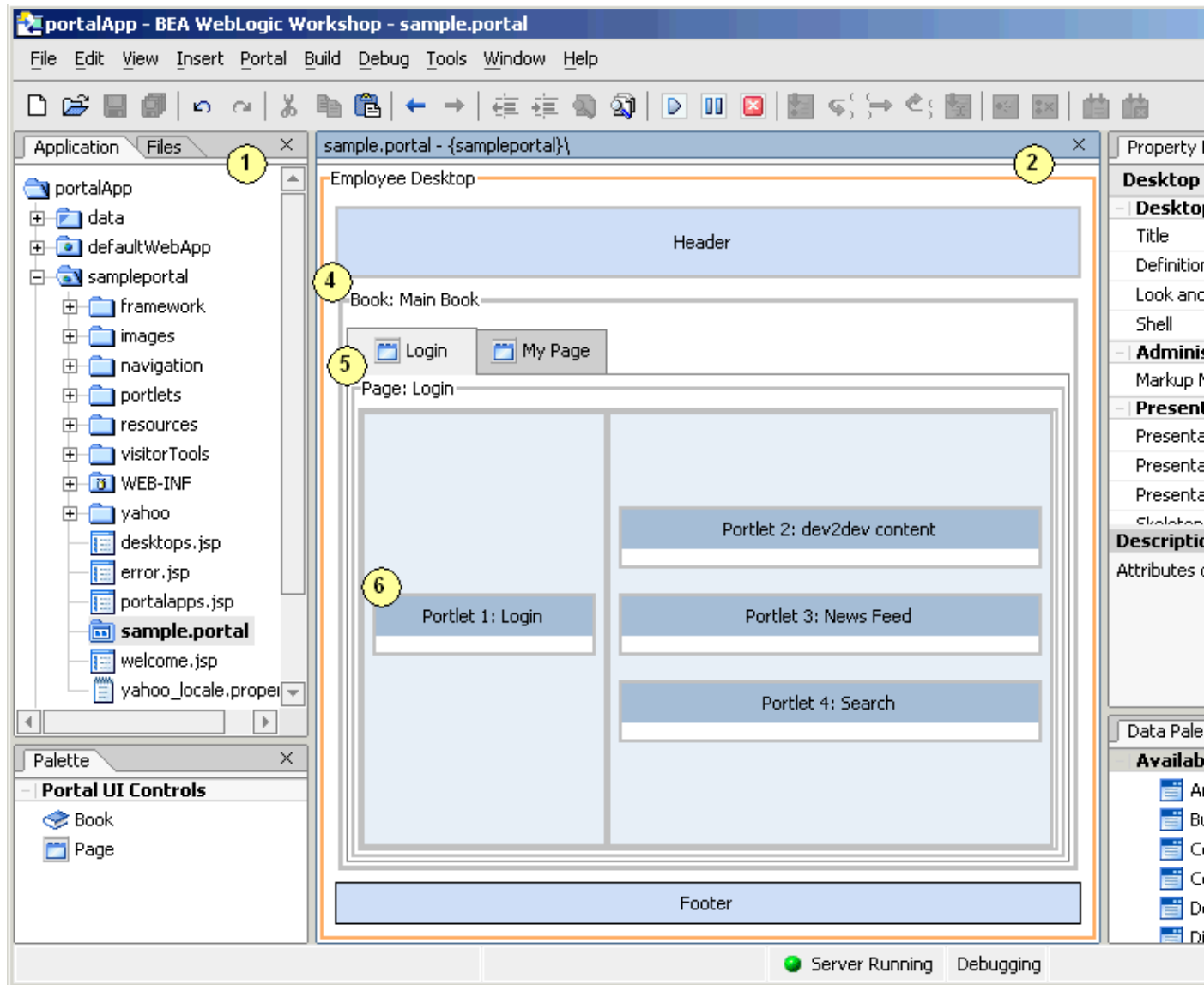
A Portal is a web application that presents and organizes disparate web content in a personalizable manner.

A Portal can present a wide variety of different web content from static HTML files to complex web applications, all of them organized into a single web site. A good example Portal is the Sample Portal, which contains portlets for logging in, news feeds, collaboration, and content management.

Personalizing the Portal might be as simple as allowing users to choose a color scheme or as complex as presenting different Portal content depending on whether the user is a customer, employee, or manager.

## Designing Portals with WebLogic Workshop

Portals are designed using WebLogic Workshop, a visual tool for designing Java and web applications. The image below shows a Portal being designed within WebLogic Workshop. WebLogic Workshop allows you to design a Portal by dragging and dropping components and setting properties on those components.



The **Application tab** (area 1) shows the Portal's source files. A Portal is stored as a .portal file, an XML file with the .portal extension. When a browser is pointed at a .portal file, the Portal is rendered as HTML.

The **main work area** (area 2) contains a picture of your Portal. You add content to the Portal by dropping and dragging elements into the main work area.

The **Property Editor** (area 3) allows you to set properties on Portal elements. When you select a Portal element in the main work area, its associated properties appear in the Property Editor.

## Organizing Content

When you have lots of different kinds of content, you need some way to organize it into a coherent whole. Portals let you organize your content using common visual and navigational tools devices such as tabs, menus and windows. The following list describes the basic devices used to organize web content within a Portal. Consult the image above to see how these organizational tools are rendered in the Workshop design environment.

A **book** (area 4) is a high-level organizational container for web content. By default a book is rendered in a web browser as a series of tabs, but you may choose to render them as a list of links, or in some other way. A book can contain other books or pages. A Portal can contain as many books as you like, but each Portal has one main book, which is the top-level container for all the content in the Portal.

A **page** (area 5) is a low-level organizational container for web content. By default a page is rendered as a single tab in a web browser. A page is typically divided into different columns and windows. These windows contain the actual web content of the Portal.

Web content is surfaced through **Portlets** (area 6). Portlets are embedded windows in your Portal that behave much like application windows: they are maximizable, minimizable, and closable. A Portlet can contain something simple, like a static HTML page, or something complex, like a feature rich web application.

## Personalization Features

The following list describes some of the ways that you can personalize the content of a Portal

### *Desktops*

A desktop is a user-specific view of the Portal content. A single Portal can support many different desktops, each designed with a specific kind of user in mind. A customer-specific desktop might show that content that appeals especially to customers, such as a product catalog, a shopping cart, and a wish-list. An employee-specific desktop might show a broader view of the Portal content, such as an inventory tracking application, an employee directory, or other sensitive information that should not be exposed to the general public.

### *User Controls*

You can personalize the behavior of an individual application within the Portal by using a user control. A user control allows you to store and retrieve information about an individual user. The information might include the last date the user visited the Portal, the user's preferences, or a user profile.

### *Campaigns*

Campaigns allow you to expose users to web content based on specific rules. The content might be advertising in the form of a pop-up window, email, or a web banner. The rules you use might be based on the user's demographic profile, purchasing history, or navigational patterns inside the Portal.

## Administration of Portals

Portals are web applications that are constantly evolving. After a Portal has been designed and put into production on the web, often a new need arises for different content, or old content becomes outdated. For these reasons, there needs to be a way to change the Portal without an extensive re-design process.

This is where the Portal Administrator steps in. The Portal Administrator Console lets you perform common administration and design tasks at runtime without having to put the Portal through another design process. For example, a Portal Administrator can add new desktops, pages, and portlets, or change the behavior of a campaign.

## WebLogic Workshop Tutorials

Portal Administrators also manage the Portal's users. They can create or delete user accounts, or they can edit the user's entitlements within the Portal.

### Related Topics

Portal Overview

Click the arrow below to navigate through the tutorial:



# Step 1: Create a Portal Project

In this step you will set up the design environment for a Portal application. You will build your Portal using WebLogic Workshop, a visual tool for designing Portals.

You will also create a Portal project. Portal projects contain supporting resources for Portal web applications including JSP tag libraries, skins, and personalization tools.

The tasks in this step are:

- To Start WebLogic Workshop
- To Create a Portal Application
- To Create a Portal Project
- To Start WebLogic Server

## To Start WebLogic Workshop

### ... on Microsoft Windows

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

1. From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**QuickStart**.
2. In the **QuickStart** dialog, click **Experience WebLogic Workshop 8.1**.

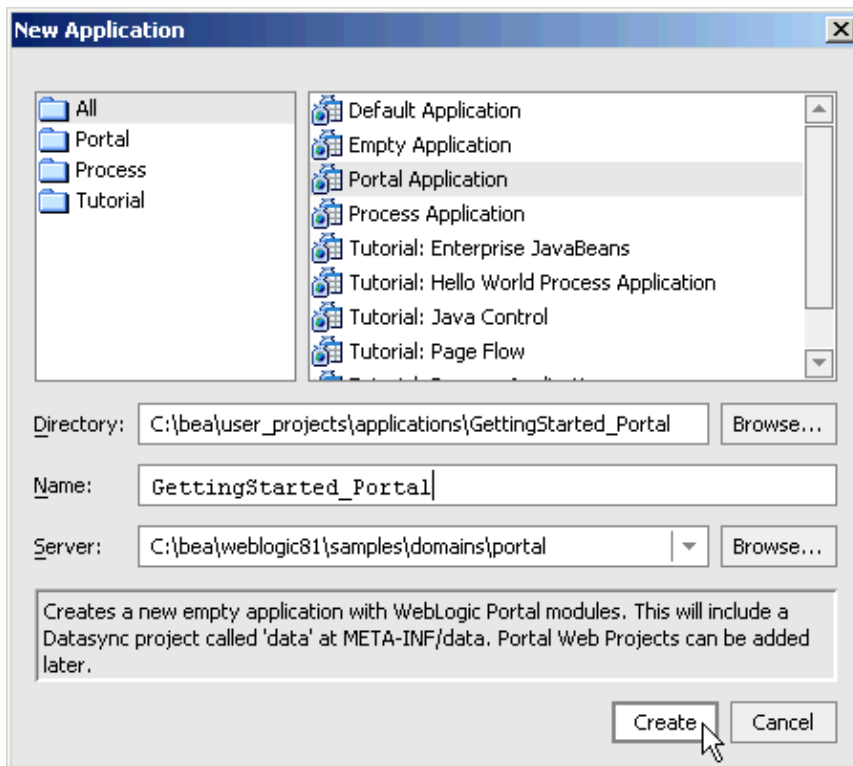
### ...on Linux

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

1. Open a file system browser or shell.
2. Locate the **Workshop.sh** file at the following address:  
`$HOME/BEA/weblogic81/workshop/Workshop.sh`
3. In the command line, type the following command:  
`sh Workshop.sh`

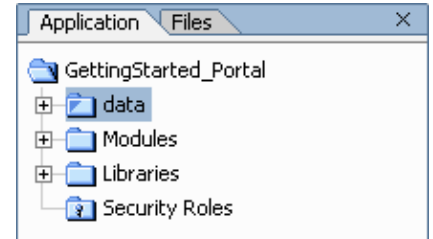
## To Create a Portal Application

1. From the **File** menu, select **New**—>**Application**.
2. In the **New Application** dialog, in the upper left-hand pane, select **All**.  
In the upper right-hand pane, select **Portal Application**.  
In the **Name** field, enter `GettingStarted_Portal`.  
In the **Directory** field, confirm that `[BEA_HOME]\user_projects\applications\GettingStarted_Portal` is selected.  
In the **Server** field, select `[BEA_HOME]\weblogic81\samples\domains\portal`.



3. Click **Create**.

When you create a new Portal Application, Workshop creates the file structure shown to the right.



The **data** folder contains a Datasync project. A Datasync project contains personalization services that can be used in Portals, including user profiles, events, session properties, campaigns, and others.

The **Modules** folder contains stand-alone applications (packaged as WAR and JAR files) that can help you develop and administer your portal. The primary application is the WebLogic Administration Portal (adminPortal.war), which is used to administer a Portal both at design time and runtime.

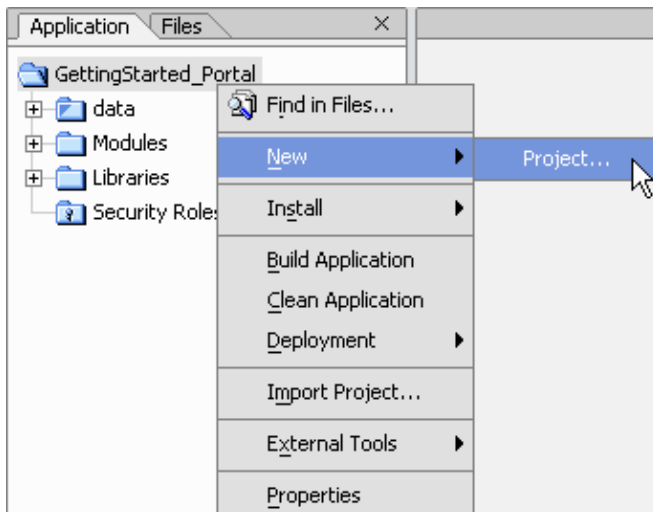
The **Libraries** folder contains resources commonly used in Portal applications, such as the User Provider Control, which lets you store and retrieve information about the current user of the Portal.

The **Security Roles** folder lets you define security roles and test users for your Portal. You can test the security design of your Portal by logging in as a test user.

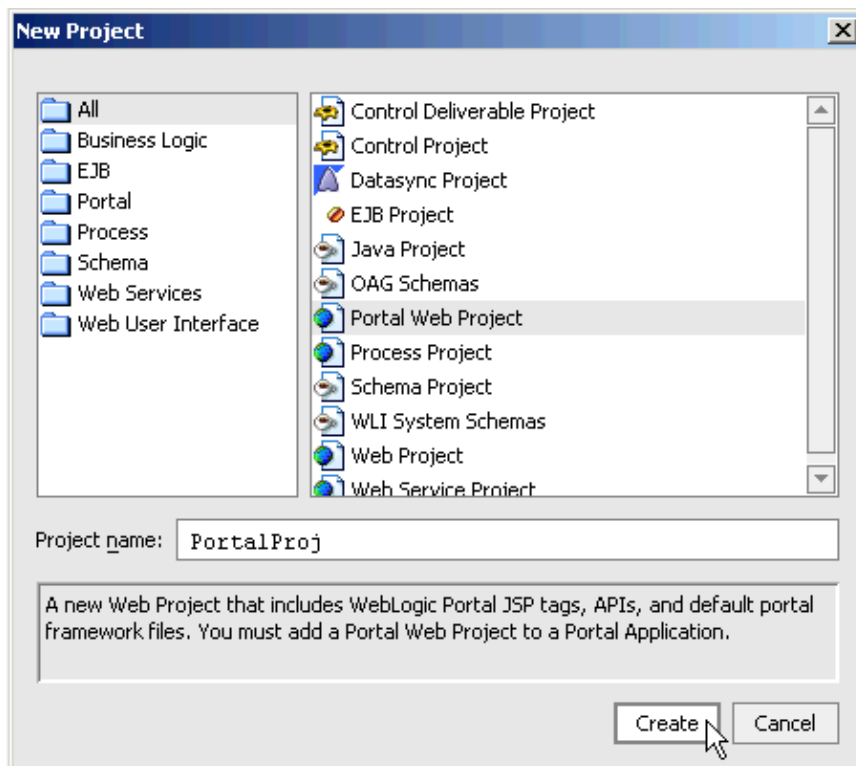
## To Create a Portal Project

1. Right-click the **GettingStarted\_Portal** folder and select **New-->Project...**



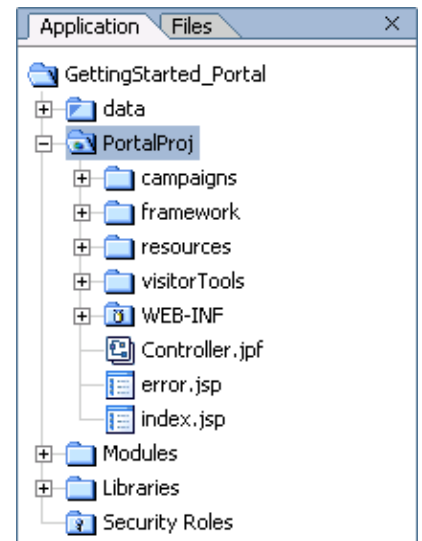


2. In the **New Project** dialog, in the upper left-hand pane, confirm that **All** is selected. In the upper right-hand pane, select **Portal Web Project**. In the **Project Name** field, enter PortalProj.



3. Click **Create**.

When you create a new Portal project Workshop creates the file structure shown to the right.



The ***campaigns*** folder is provided as a convenient place to store your campaign rules. Campaigns let you target users with content (general web content, emails, advertisement, etc.) based on fine-grained rules. For example, a campaign can present a user with web advertisement based on the user's purchasing history, demographics, navigational patterns, or any number of criteria.

The ***framework*** folder contains resources for determining the look and feel of your Portal, including skins and themes.

The ***resources*** folder stores commonly re-used JSP elements, including CSS files and JSP templates.

The ***visitorTools*** folder stores an application that lets users set their own Portal preferences. If you wish, you can include this application in your Portal.

The ***WEB-INF*** folder stores JSP tag libraries, configuration files, compiled class files, and other runtime resources.

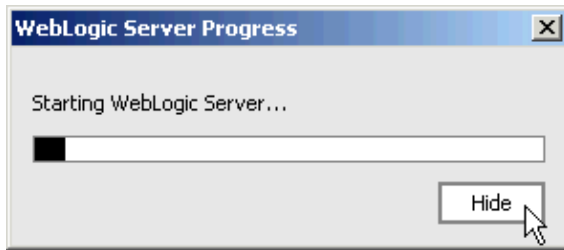
The ***Controller.jspf***, ***error.jsp***, and ***index.jsp*** are the components of the default parent page flow. You may use this page flow as the master error handler and/or the master navigational controller for other page flows in your portal.

## To Start WebLogic Server

In order to run and test your Portal application, it must first be deployed on WebLogic Server. For convenience you will start WebLogic Server now, so that you can test your Portal as you design it.

1. From the ***Tools*** menu, select ***WebLogic Server***—>***Start WebLogic Server***.
2. On the ***WebLogic Server Progress*** dialog, you may click ***Hide*** and continue to the next task.

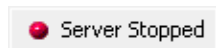
## WebLogic Workshop Tutorials



You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed.



If WebLogic Server is not running, a red ball is displayed.



You are now ready to begin designing your Portal.

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



## Step 2: Organize Content in a Portal

In this step you will create a Portal and import two web applications into the Portal.

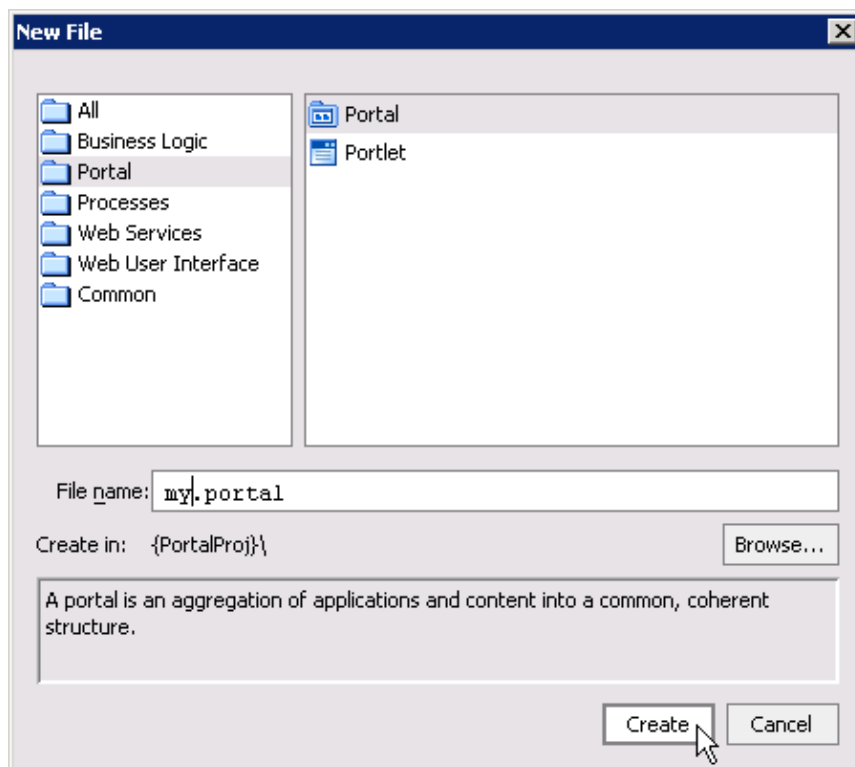
The tasks in this step are:

- To Create a Portal File
- To Add a Page
- To Import Two Web Applications
- To Place the Web Applications in the Portal
- To Test the Portal

### To Create a Portal File

In this step you will create a Portal file, an XML file with the .portal file extension. The .portal file is the central defining file of a Portal. It refers to all of the major components of the portal: the desktops, books, pages, portlets, etc. When users visit the Portal, they will point their browsers at this file:  
<http://localhost:7001/PortalProj/my.portal>.

1. On the **Application** tab, right-click the **PortalProj** folder, and select **New-->Portal**.
2. In the **New File** dialog, in the upper-left pane, confirm that **Portal** is selected.  
In the upper-right pane, confirm that **Portal** is selected.  
In the **File name** field, enter my.portal.



3. Click **Create**.

When you create a .portal file, a visual representation of the Portal is displayed in the main work area. The picture to the right shows the main visual components of a Portal.



A **desktop** is a user-specific view of the Portal content. A Portal can support many desktops. A single Portal might support an employee-specific desktop, a customer-specific desktop, and others, where each desktop exposes different kinds of users to different sets of content. For example, an employee-specific desktop would probably be a wider view of the Portal content compared to a customer-specific view. The employee desktop might include access to sensitive information that the company wouldn't want in the hands of customers or rival companies. Any part of a Portal can be included or excluded from a desktop, including a book, a page, a specific application, or an individual link.

Desktops can also define the look and feel attributes of a Portal. Desktops can be associated with a particular skin which defines the color scheme, fonts, and images used. Desktops also contain a header and footer: you can place images, text, or any web content in these areas to give consistency to the look and feel of a desktop.

A **book** gives you a way to organize your content and navigation in a hierarchical manner. Books can contain other books or pages. In a browser, a book is rendered as a set of tabs or links. Each Portal contains a main book called by default "Main Page Book".

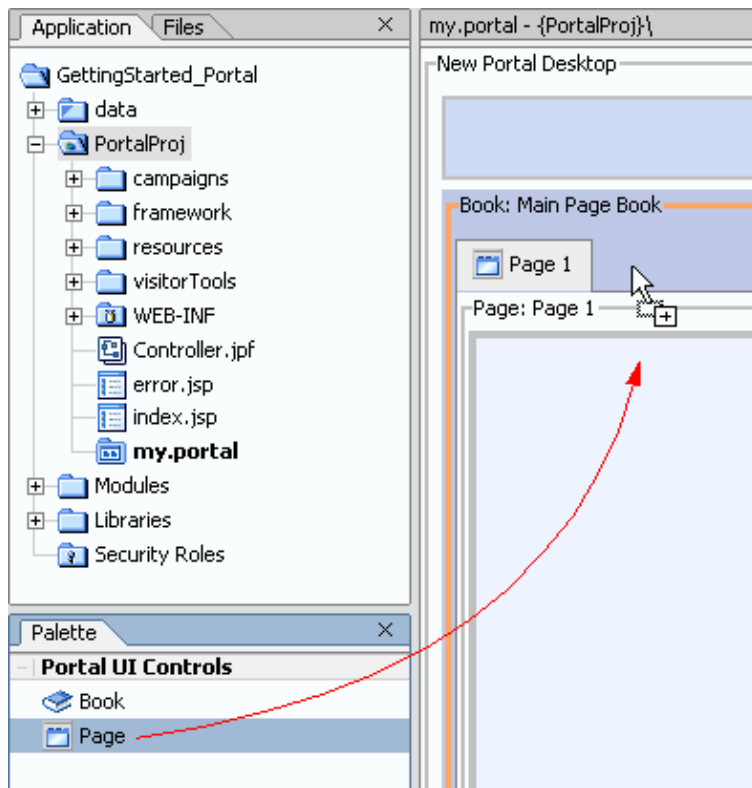
A **page** consists of a set of columns and/or windows that organize the actual content of your Portal. You navigate to a page by clicking on an individual tab or a link.

## To Add a Page

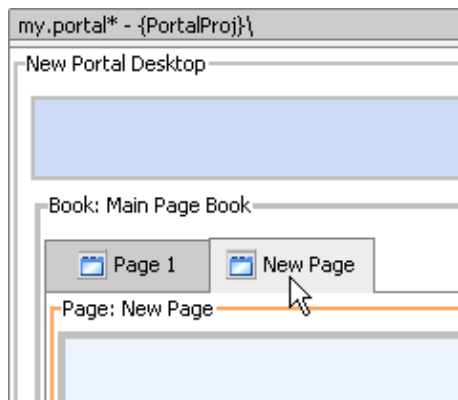
In this step you will add a second page to the Portal's main book. When the Portal is rendered in a browser, the two pages will appear as two clickable tabs. You add a new page by dragging and dropping the new page into the main work area. You will also set properties on the new page using the Property Editor.

1. From the **Palette** tab, drag and drop the **Page** icon to the area directly to the right of the **Page 1** tab.

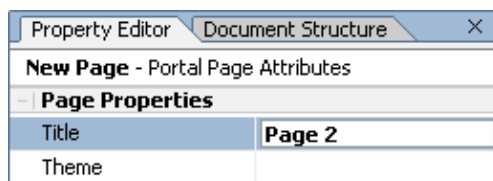
(If you do not see the **Palette** tab, select **View**—>**Windows**—>**Palette**.)



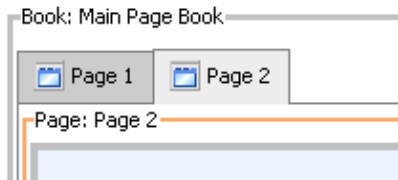
2. In the main work area, select **New Page**.



3. On the **Property Editor** tab, in the **Title** property, enter Page 2 and press the **Enter** key.



When you enter a value in the Property Editor, note the change that occurs in the main work area: the title of the page has changed from "New Tab" to "Page 2".



4. Press **Ctrl+S** to save your work.

## To Import Two Web Applications

In this task you will import the source code for two applications into your Portal Project. These two applications are web applications built using WebLogic Workshop's Page Flow technology. To learn more about Page Flow web applications see the Getting Started Tutorial: Web Applications.

1. On the **Application** tab, right-click the **PortalProj** folder and select **Import...**
2. In the **Import Files to Project 'PortalProj'** dialog navigate to and select the portlets folder located in [BEA\_HOME]/weblogic81/samples/platform/tutorial\_resources/GettingStarted\_Portal.
3. Click **Import**.

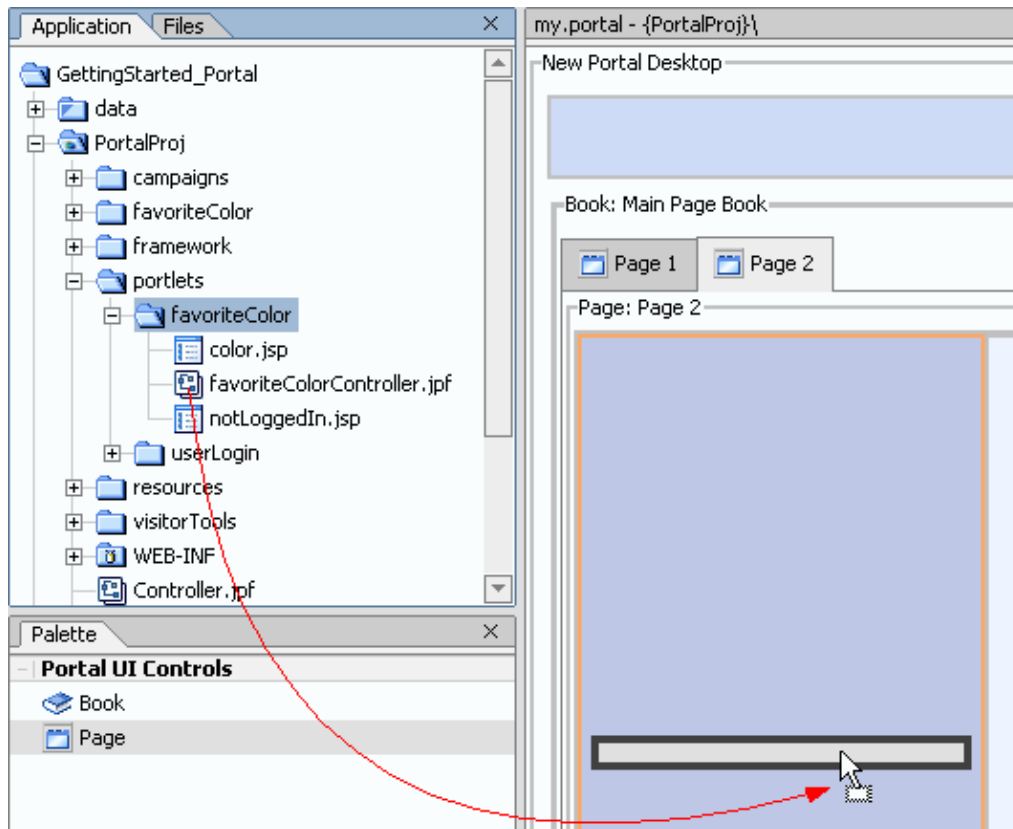
## To Place the Web Applications in the Portal

Portlets are windows that surface web applications and other web content. A Portlet can contain content that is as simple as an HTML file or as complex as a large web application. In the following step you will add two Portlets to your Portal, one will surface the login application, the other will surface the favorite color application.

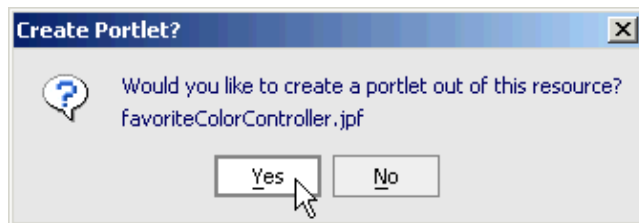
Note that Portlets are automatically created for you whenever you drag and drop new content into the Portal.

1. On the **Application** tab, navigate to the file GettingStarted\_Portal/PortalProj/portlets/favoriteColor/favoriteColorController.jspf.
2. On the **Application** tab, drag and drop the file **favoriteColorController.jspf** onto the left column of **Page 2**.

(Note that each page is by default divided into two columns. These columns are provided as a convenience for arranging your Portal content. To change the number of columns, use the **Layout Type** property on the **Property Editor** tab.)

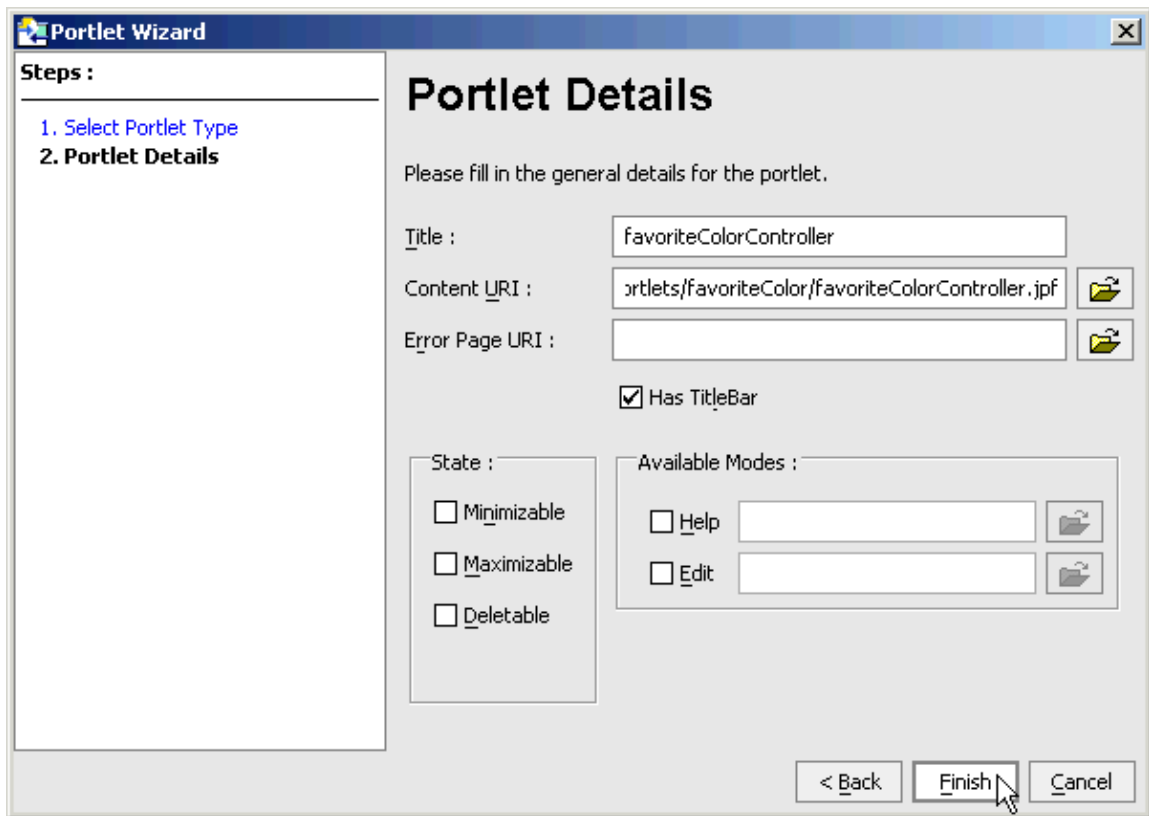


3. In the *Create Portal* dialog, click *Yes*.

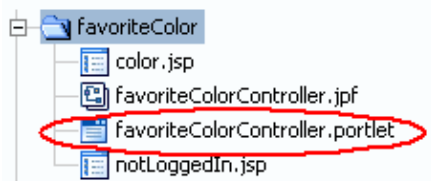


4. In the *Portlet Wizard* dialog, click *Finish*.

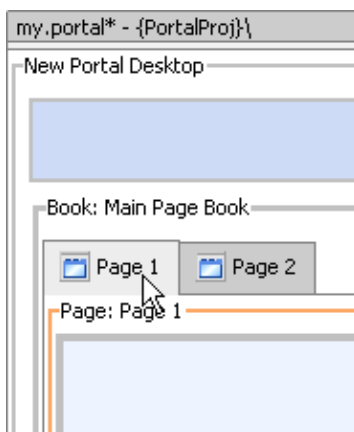




Note that a new .portlet file has been created in the favoriteColor folder.



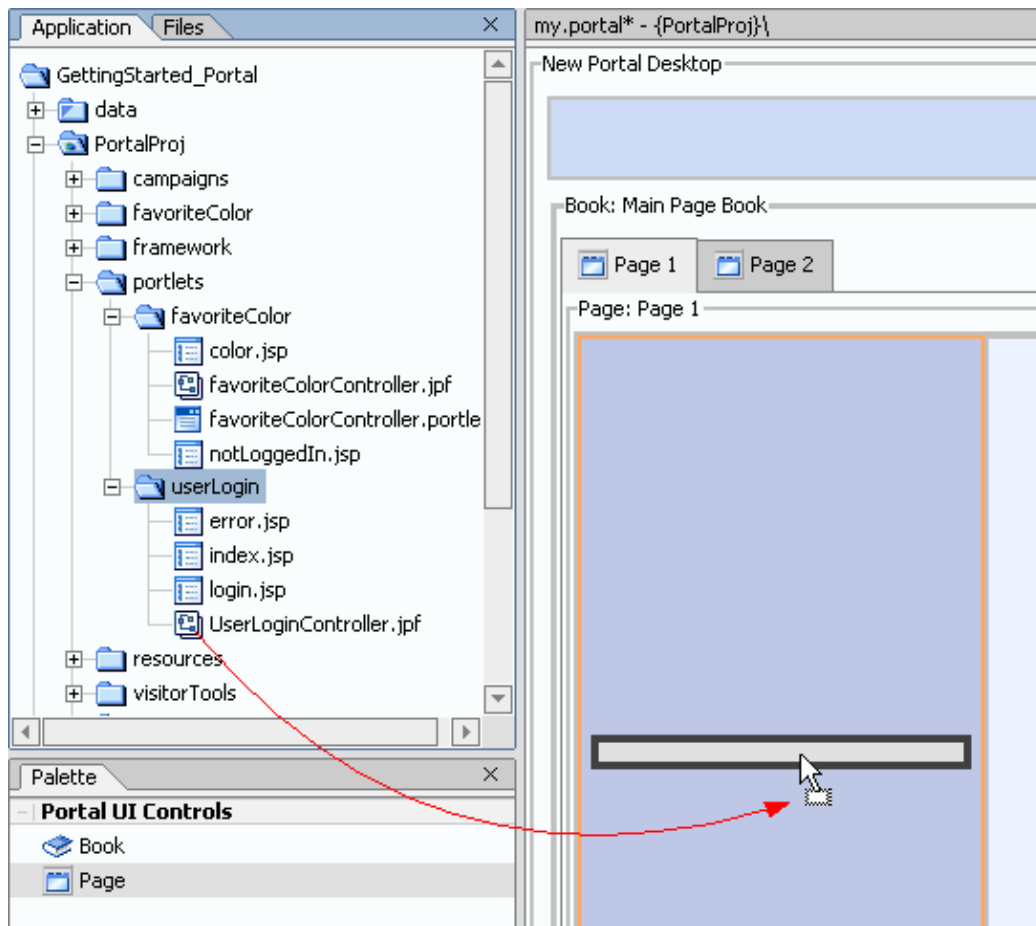
5. In the main work area, click the tab labeled **Page 1**.



On the **Application** tab, locate the file

GettingStarted\_Portal/PortalProj/portlets/userLogin/UserLoginController.jspf.

6. On the **Application** tab, drag and drop the file **UserLoginController.jspf** onto the left column of **Page 1**.



7. In the *Create Portal* dialog, click *Yes*.
8. In the *Portlet Wizard* dialog, click *Finish*.
9. Press **Ctrl+S** to save your work.

You now have a Portal that surfaces two web applications.

## To Test the Portal

In this task you will test the Portal using the Workshop Test Browser. The Workshop Test Browser can be used to test any web application you build with WebLogic Workshop.

There are two things to note as you test the behavior of the Portal at this point:

1. The login and favorite color applications do not communicate with one another. In particular, the favorite color application has no way of knowing whether the user is logged in or not.
2. The favorite color application has no way of remembering a user's favorite color once the user has logged out.

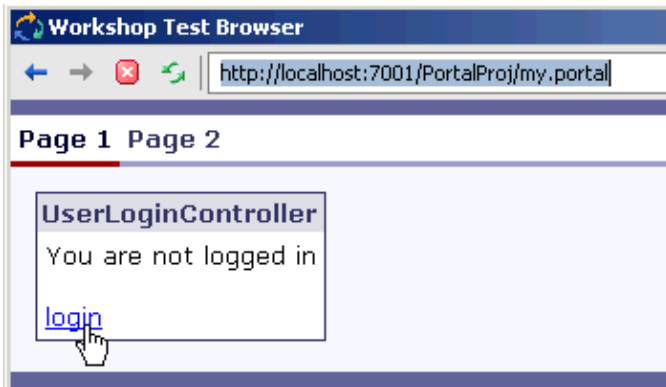
In the next step of the tutorial you will correct both of these problems. You will make the favorite color application accessible only to logged in users and you will provide a way for the favorite color application to store a user's favorite color while the user is logged out.

## WebLogic Workshop Tutorials

1. Click the Start button, shown below.



2. When the **Workshop Test Browser** launches, click the **Page 2** tab. (Notice that you can access the favorite color application even though you have not logged in.)
3. Click the **Page 1** tab.



4. Login in with the username / password: **weblogic / weblogic**.
5. Once you have logged in, click the **Page 2** tab, and select a favorite color.
6. Log out of the Portal. (Click the **Page 1** tab and click **logout**.)
7. Log back into the Portal. Click the **Page 2** tab. (Notice that the application has failed to remember the favorite color selected since you last logged in.)

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



## Step 3: Personalize a Web Application

So far, you have a Portal which surfaces two applications. But the Portal lacks any personalization features. The favorite color application has no way to know when a user is logged in and it has no way of remembering a user's favorite color after the user has logged out of the Portal.

In this step you will make the favorite color application capable of remembering the user's favorite color after the user has logged out. You will add a User Profile Control to the application, a control which can save information about individual Portal users. The User Profile Control saves the information in a database (a database entirely managed by the control) and accesses the information using a set of property/value pairs.

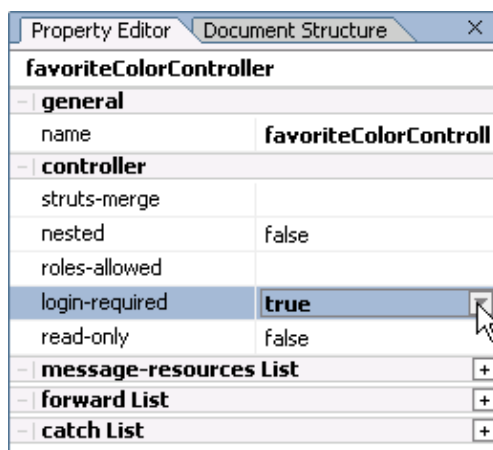
The tasks in this step are:

- To Edit the Application to Require Login
- To Add a User Profile Control
- To Edit the Web Application to Use the Control
- To Test the Portal

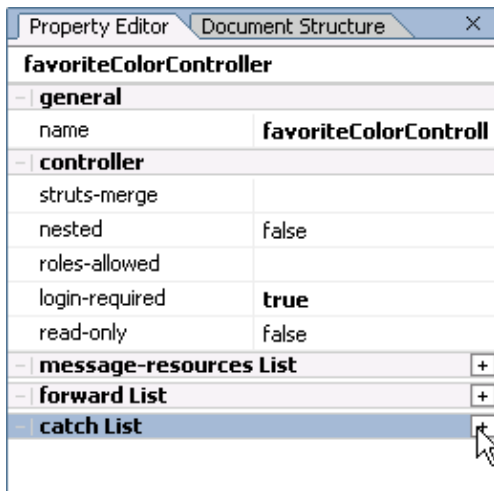
### To Edit the Application to Require Login

In this step you will edit the application so that it is accessible only to logged in users. You will accomplish this by setting properties on the favorite color application. First, you will set the application's login-required property to true. This will cause the application to throw a `NotLoggedInException` anytime a non-logged in user tries to access the application. Second, you will edit the application's catch property. The catch property provides error handling instructions to the application. In this case the application will handle the exception by sending non-logged in users to a JSP page that instructs them that they must first log on to use the application.

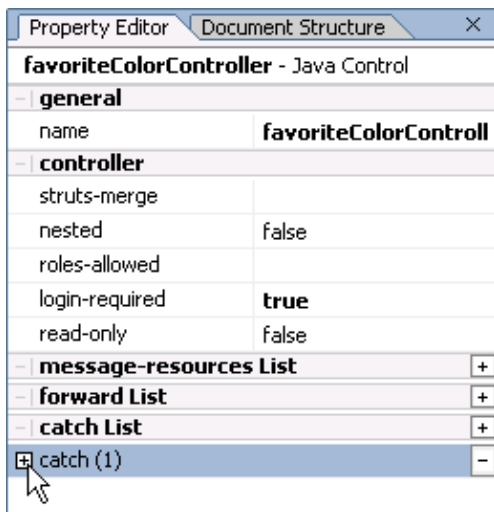
1. On the **Application** tab, double-click the file *favoriteColorController.jspf* to open the file.
2. On the **Property Editor** tab, set the *login required* property to true.



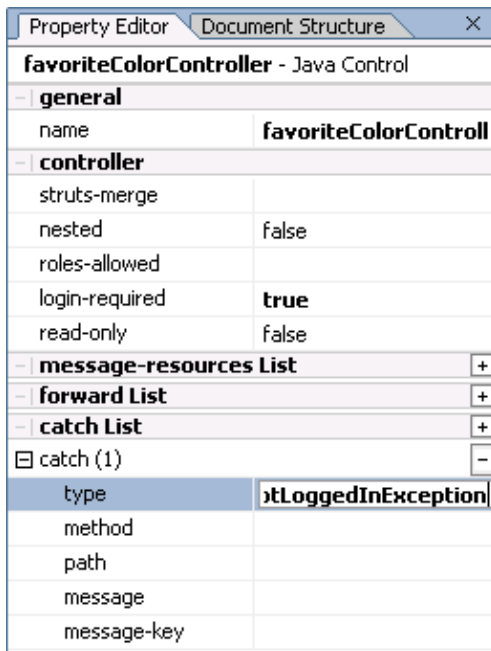
3. On the **Property Editor** tab, click the *plus sign* to the right of the *catch List* property.



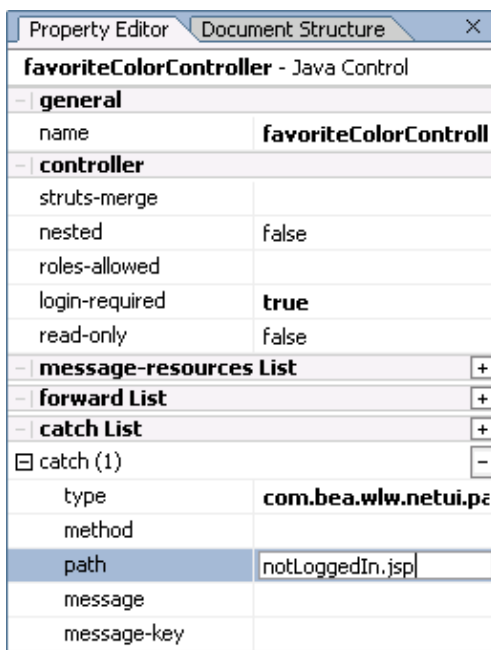
4. On the **Property Editor** tab, click the *plus sign* to the left of **catch(1)**.



5. In the **type** property, enter `com.bea.wlw.netui.pageflow.NotLoggedInException` and press the **Enter** key.



6. In the **path** property, enter notLoggedIn.jsp and press the **Enter** key.

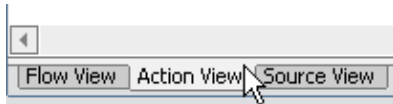


7. Press **Ctrl+S** to save your work.

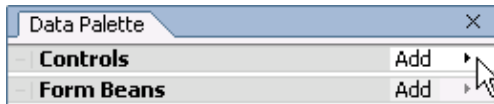
## To Add a User Profile Control

In this task you will add a User Profile Control to the favorite color application. A User Profile Control provides access to a user's User Profile: a store of information that is maintained by the Portal. The User Profile stores information about the user as a set of property/values pairs. Most importantly the User Profile persists after a user has logged out of the Portal. In this case you will save the user's favorite color in the User Profile.

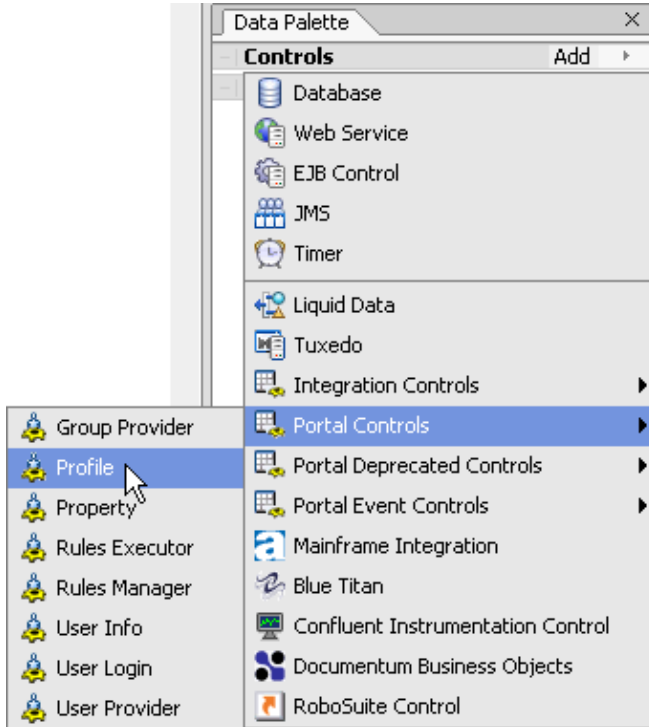
1. In the main work area, click the **Action View** tab.



2. On the **Data Palette** tab, to the right of the **Controls** heading, click **Add**.



3. From the menu, select **Portal Controls**—>**Profile**.

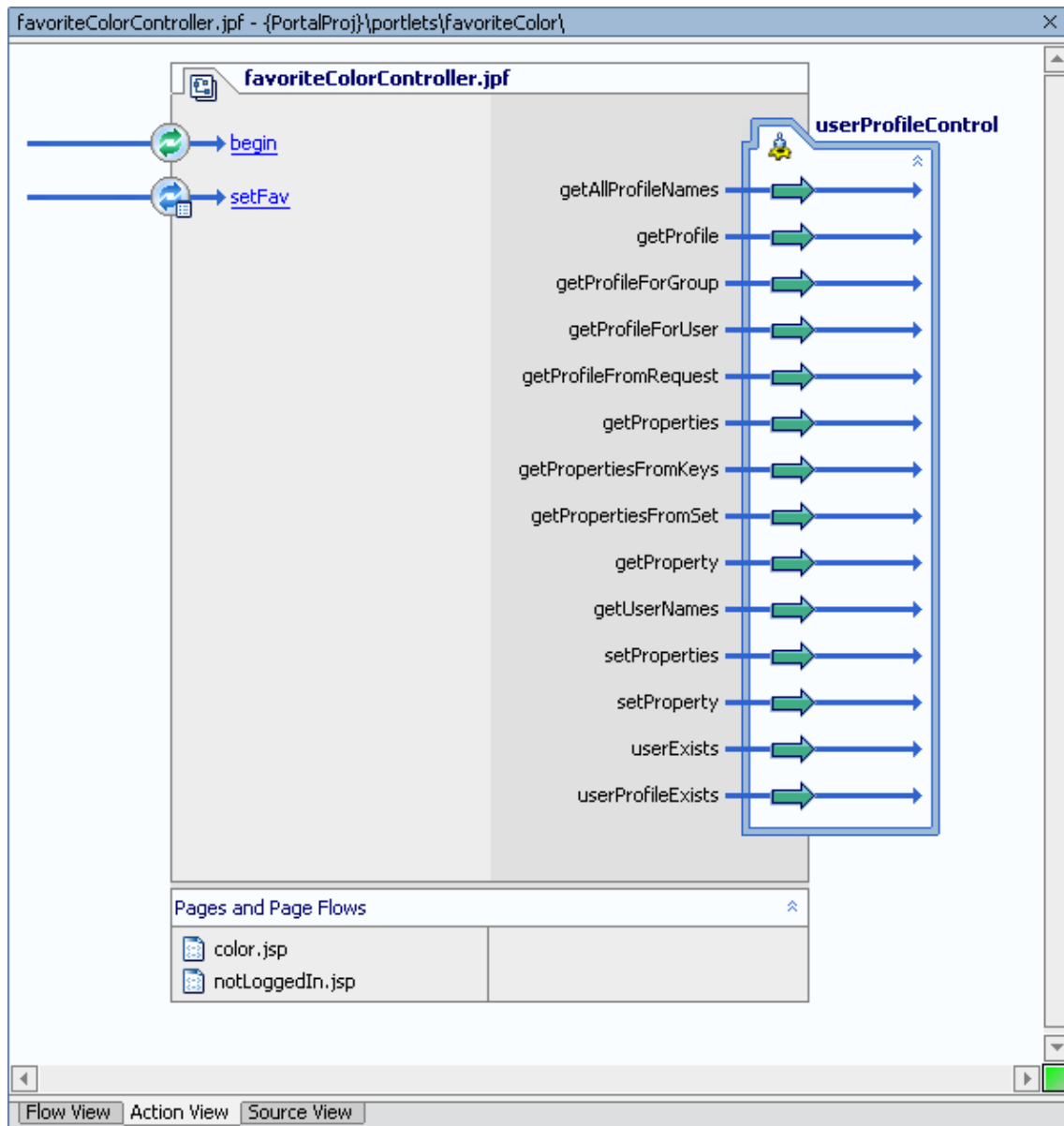


4. In the **Insert Control – User Profile Control** dialog, enter `userProfileControl`.



5. Click **Create**.
6. Press **Ctrl+S** to save your work.

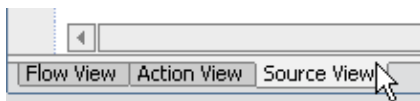
The User Profile Control you have added contains methods for storing user information (`setProperties` and `setProperty`) and methods for retrieving user information (such as `getProfile` and `getProfileForUser`). You can use these methods to store and retrieve user preferences and other information over a history of visits to the Portal.



## To Edit the Web Application to Use the Control

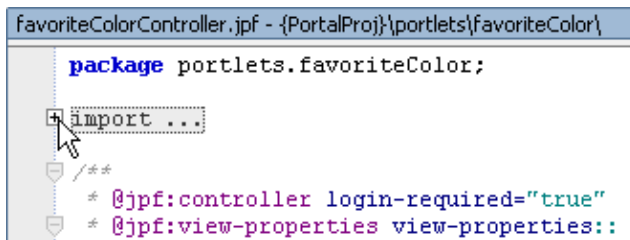
In this task you will edit the favorite color application code to take advantage of the User Profile Control. When the user selects a favorite color, it will be saved in the User Profile. When the user revisits the Portal, the favorite color will be retrieved from the User Profile.

1. In the main work area, click the *Source View* tab.



2. At the top of the file, click the *plus sign* to the left of the *import...* area.

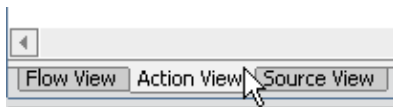




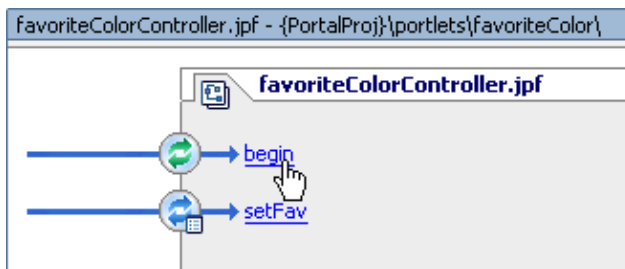
3. Edit the set of import statements so it looks like the following. Code to add is shown in red.

```
import com.bea.wlw.netui.pageflow.FormData;
import com.bea.wlw.netui.pageflow.Forward;
import com.bea.wlw.netui.pageflow.PageFlowController;
import com.bea.p13n.usermgmt.profile.ProfileWrapper;
```

4. In the main work area, click the **Action View** tab.



5. In the main work area, click the link text for the **begin** method.



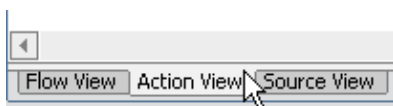
6. Edit the source code for the begin method so it appears as follows. Code to add appears in red:

```
/**
 * @jpf:action
 * @jpf:forward name="showColor" path="color.jsp"
 */
protected Forward begin()
    throws Exception
{
    // Get the current user's profile.
    ProfileWrapper pw = userProfileControl.getProfileFromRequest( getRequest() );

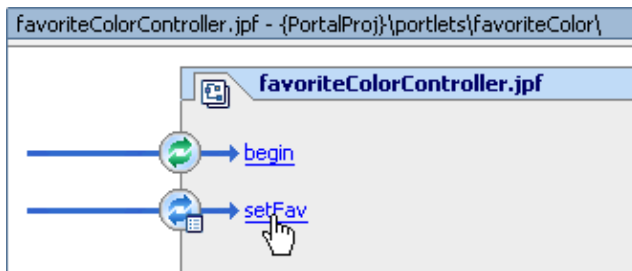
    // If the favorite color retrieved from the user profile is not null...
    if( pw.getPropertyAsString( "favorites", "favoriteColor" ) != null )
    {
        // ...load the favorite color into the member variable _fav
        _fav = pw.getPropertyAsString( "favorites", "favoriteColor" );
    }

    // Render the color.jsp page
    return new Forward( "showColor" );
}
```

7. In the main work area, click the **Action View** tab.



8. In the main work area, click the link text for the **setFav** method.



9. Edit the source code for the setFav method so it appears as follows. Code to add appears in red:

```
/**
 * @jpf:action
 * @jpf:forward name="showColor" path="color.jsp"
 */
protected Forward setFav(SelectFavForm form)
    throws Exception
{
    // Get the current user's profile.
    ProfileWrapper pw = userProfileControl.getProfileFromRequest( getRequest() );

    // Save the selected color in the user's profile.
    // This color will be read from the profile on the
    // user's next visit to the portal.
    pw.setProperty( "favorites",
                    "favoriteColor",
                    form.getFavoriteColor() );

    // Load the selected color into the global variable _fav.
    // The JSP page color.jsp reads the display color from _fav.
    _fav = form.getFavoriteColor();

    // Render the JSP page color.jsp.
    return new Forward( "showColor" );
}
```

10. Press **Ctrl+S** to save your work.

## To Test the Portal

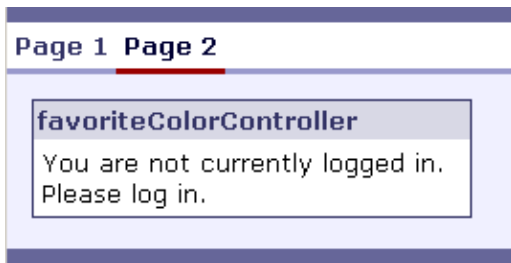
As you test the Portal, notice it has two new capabilities:

1. The favorite color application is accessible only to logged in users
  2. The favorite color application remembers a user's favorite color while the user is logged out
1. On the **Application** tab, double-click **my.portal** to display the file in the main work area.
  2. Click the **Start** button, shown below.



3. When the **Workshop Test Browser** launches, click the **Page 2** tab.

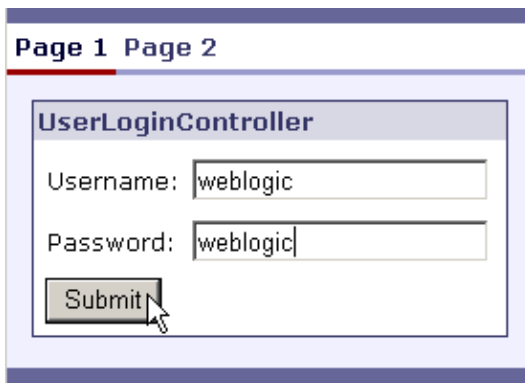
Notice that the user must be logged in to access the Favorite Color application.



4. Click the **Page 1** tab and click the **login** link.

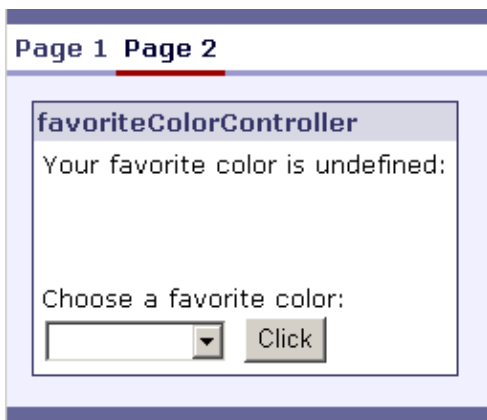


5. Login with the username/password combination **weblogic/weblogic**.

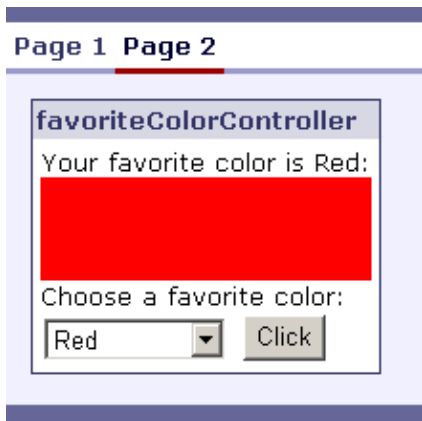


6. Click the **Page 2** tab.

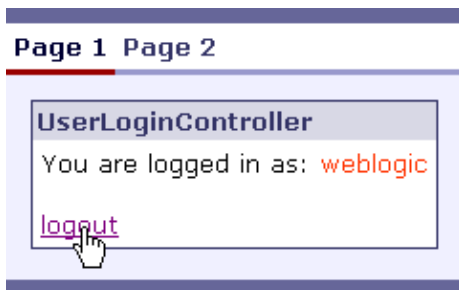
Notice that the Favorite Color Application is visible once the user is logged in.



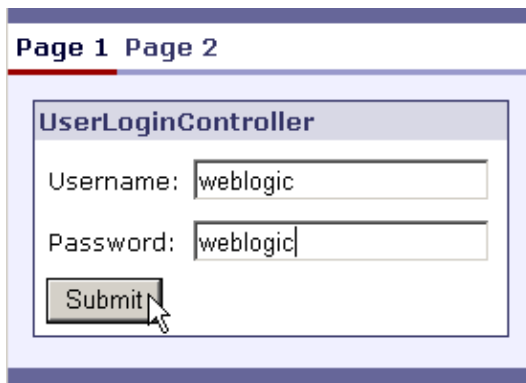
7. Select a favorite color and click **Click**.



8. Click the **Page 1** tab and click the **logout** link.



9. Login again with the username/password combination **weblogic/weblogic**.



10. Click the **Page 2** tab.

Notice that the Portal remembered the user's favorite color while the user was logged out.



### Related Topics

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



# Summary: Getting Started Portal Tutorial

This topic reviews the concepts introduced in the tutorial and provides links to more information on Portals.

## Concepts Introduced in the Tutorial

### Organizing Content

- Portals let you organize a variety of web content into a single, easy to navigate web site. For more information on organizing content in a Portal see [Assembling Portal Applications](#).

There are extensive tools for determining the look and feel of a Portal. For more information, see [Look & Feel Architecture](#).

### Personalization

- The User Profile Control can be used to personalize the user's Portal experience. For more information on using the User Profile Control see [User Profile Control](#).

This tutorial only serves as a first introduction to personalization. Portal personalization also include the following features:

- There are a variety of other controls that can be used to personalize the Portal experience. For more information see [Using Portal Controls](#).
- Desktops are an important personalization feature for different kinds of users. For information on filtering content using desktops see [How Do I: Create a Desktop?](#) For information on using the Visitor Tools application to allow users to set the look and feel of the Portal themselves see [Adding Visitor Tools to Portals](#).
- Campaigns let you target users with content based on user properties, behavior, events, and other fine-grained criteria. For more information on campaigns see [Creating Campaigns](#).
- Using a Unified User Profile takes your User Profile to the next level. A Unified User Profile lets you draw user information from external sources, such as LDAP servers, legacy systems and databases. For more information on Unified User Profiles see [Setting up Unified User Profiles](#).

### Portlets

- Portlets are containers that hold web content within a Portal. For more information on how to customize Portlet behavior see [Customizing Portlets](#).

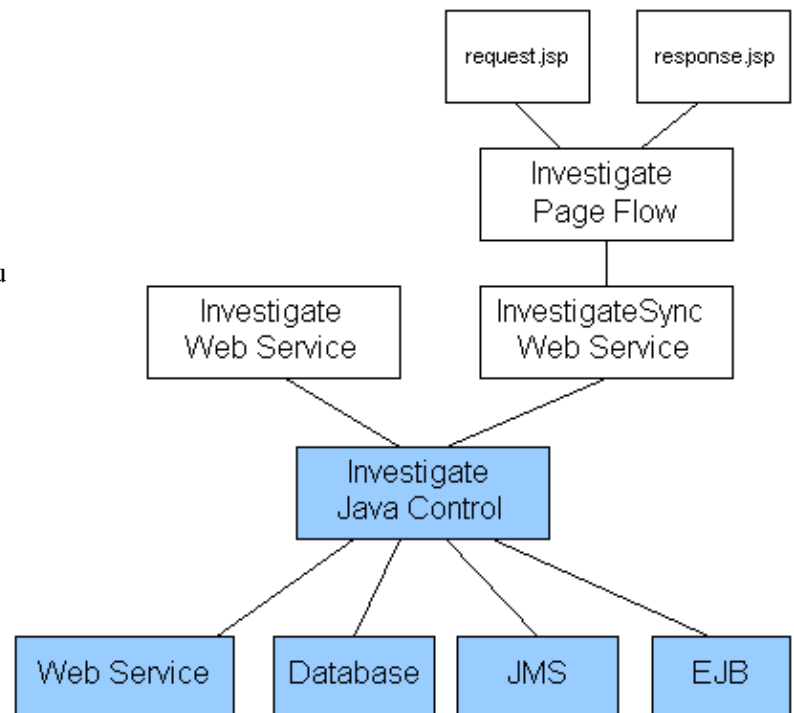
Click one of the following arrows to navigate through the tutorial:



# Tutorial: Java Control

## The Big Picture

The Java control tutorial is the first in a series of three tutorials that builds the Investigate Application, an application designed to assemble a credit-worthiness report on loan applicants. (**Note:** you do **not** need to execute the tutorial series in order. You may skip ahead to the web services tutorial or page flow tutorial if you wish.)



This, the first tutorial in the series, builds the core of the application: the Investigate Java control. The Investigate Java control consults a variety of components to assemble the credit-worthiness report. The Investigate Java control and its components are shown in blue in the diagram to the right. The remaining two tutorials focus on different ways to access the core functionality encapsulated by the Investigate Java control.

The second tutorial in the series, Tutorial: Web Service, builds the web services tier of the application.

The third tutorial in the series, Tutorial: Page Flow, builds a web-based user interface for the Investigate Application.

## Tutorial Goals

Through this tutorial, you will learn how to create and test a Java control with WebLogic Workshop. Along the way, you will also learn how to create methods that expose a control's functionality, become acquainted with WebLogic Workshop's support for asynchronous communication and loose coupling, and learn how to use WebLogic Workshop's built-in Java controls to help you connect to backend resources and to speed the design process.

## Tutorial Overview

The Java control you build with this tutorial collects credit-related information about a loan applicant, computes a credit worthiness score, and returns the combined information to the client.

There are six actors in this scenario:

## WebLogic Workshop Tutorials

- The clients of your Java control. Clients of Investigate are loan or credit application processing systems. An example might be a credit card application processing system at a department store. These systems will use Investigate to determine the applicant's credit worthiness. They will supply you with the applicant's taxpayer ID, and your control will compute and return a credit score.
- The Investigate Java control. Investigate receives a taxpayer ID as part of requests from clients and respond with information about the applicant's credit worthiness.
- A credit card reporting web service with information about the applicant's credit history. You should imagine that the credit card reporting web service is an external application accessible over the internet. But for the purposes of the tutorial, this web service will be deployed on the same server as the Investigate Java control.
- A database containing bankruptcy information about credit applicants.
- A credit scoring application, exposed through the Java Message Service (JMS), designed to calculate a credit score based on information you've collected.
- An Enterprise Java Bean (EJB) designed to provide a credit rating based on the score.

This tutorial guides you through the process of adding functionality in increments, and shows you how to test your Java control as you build it.

### Steps in This Tutorial

#### Step 1: Begin the Investigate Java Control — 25 minutes

In this step you build and run your first WebLogic Workshop Java control. You start WebLogic Workshop and WebLogic Server, then create the access points clients use to request and retrieve credit reports.

#### Step 2: Add a Database Control — 20 minutes

You add a control for access to a database, then query the database for bankruptcy information.

#### Step 3: Add a Web Service Control — 25 minutes

You use a Web service control to invoke another web service, collecting credit card data on the applicant. This step also uses XMLBeans to parse the response from the Web service.

#### Step 4: Add a JMS Control — 20 minutes

You will take the data you have gathered and request help from an in-house application to retrieve a credit score.

#### Step 5: Add an EJB Control — 20 minutes

You will send the credit score to an Enterprise Java Bean (EJB) that will use it to measure the applicant's credit risk.

#### Step 6: Add Support for Cancellation and Exception Handling — 20 minutes

You enhance Investigate to handle a client's desire to cancel the request, an overly long wait for a response from the credit card service, and exceptions thrown from your service's methods.

#### Step 7: Security — 20 minutes

#### Tutorial: Java Control



## WebLogic Workshop Tutorials

You modify the Investigate Java control so it can access a web service over SSL (Secure Socket Layer).

Step 8: Compile the Java Control and Add a Property — 25 minutes

You add a property to the Investigate Java control so that developers using the control can change the way it functions and you package the control as a JAR file for general distribution.

Summary: Java Control Tutorial

You review the concepts and technologies covered in the tutorial. This step provides additional links for more information about each area covered.

To begin the tutorial, see Step 1: Begin the Investigate Java Control.

Click the arrow to navigate to the next step.



# Step 1: Begin the Investigate Java Control

In this step, you create a simple Java control that assembles credit–worthiness reports on loan applicants. You will focus on the public interface that is presented to users rather than the underlying implementation code that assembles the credit report. The public interface you build works as follows: users begin by requesting a credit report from the Investigate Java control, then the Investigate Java control builds a credit report based on those requests, and finally sends a completed report back to the user through a *callback*.

Callbacks are part of your Java control's support for *asynchronous* communication with its users. By supporting asynchronous communication, users are not forced to halt execution of their own processes while the credit report is being prepared. For detailed information of callbacks and asynchronous communication, see Designing Asynchronous Interfaces.

The tasks in this step are:

- To start WebLogic Workshop
- To create a new application and select a WebLogic server domain
- To create a new project
- To start WebLogic Server
- To create the Investigate Java control
- To create the Applicant class
- To add a member variable
- To add a method
- To add a message buffer
- To specify a parameter and method body
- To add a callback
- To add code to invoke the callback after the credit–worthiness report has been assembled
- To generate a test client for the Investigate Java control
- To test the Investigate Java control

To Start WebLogic Workshop

If you have an instance of WebLogic Workshop already running, you can skip this step.

... on Microsoft Windows

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

1. From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**QuickStart**.
2. In the **QuickStart** dialog, click **Experience WebLogic Workshop 8.1**.

...on Linux

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

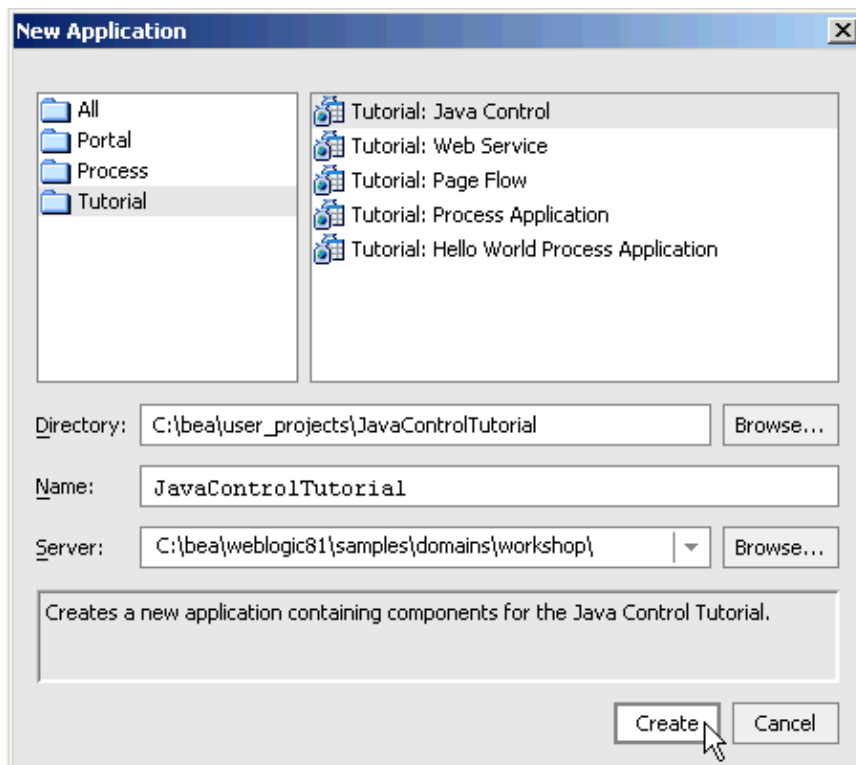
1. Open a file system browser or shell.
2. Locate the **Workshop.sh** file at the following address:  
`$HOME/beanlogic81/workshop/Workshop.sh`
3. In the command line, type the following command:  
`sh Workshop.sh`

## WebLogic Workshop Tutorials

### To Create a New Application and Select a WebLogic Server Domain

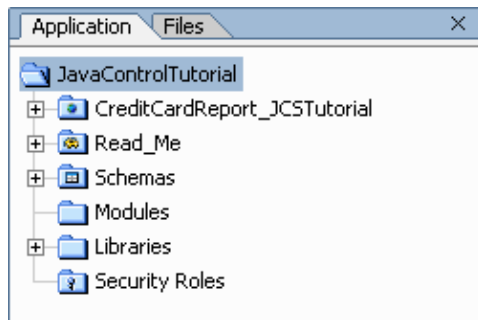
The Investigate Java control uses four different resources to assemble a credit report: (1) a web service, (2) a database, (3) an EJB application, and (4) a message driven EJB application. In this task you will create a new application that contains all four of these resources. The application you create below already contains the web service resource. The server domain you select below contains the remaining three resources.

1. From the **File** menu, select **New** --> **Application**. The **New Application** dialog appears.
2. In the **New Application** dialog, in the upper left-hand pane, select **Tutorial**.  
In the upper right-hand pane, select **Tutorial: Java Control**.  
In the **Directory** field, use the **Browse** button to select a location to save your source files. (The folder you choose is up to you, the default location is shown in the illustration below.)  
In the **Name** field, enter JavaControlTutorial.  
In the **Server** field, from the dropdown list, select  
BEA\_HOME\weblogic81\samples\domains\workshop.



3. Click **Create**.

A new application is created. The application contains six sub-folders named **CreditCardReport\_JCSTutorial**, **Read\_Me**, **Schemas**, **Modules**, **Libraries**, and **Security Roles**.

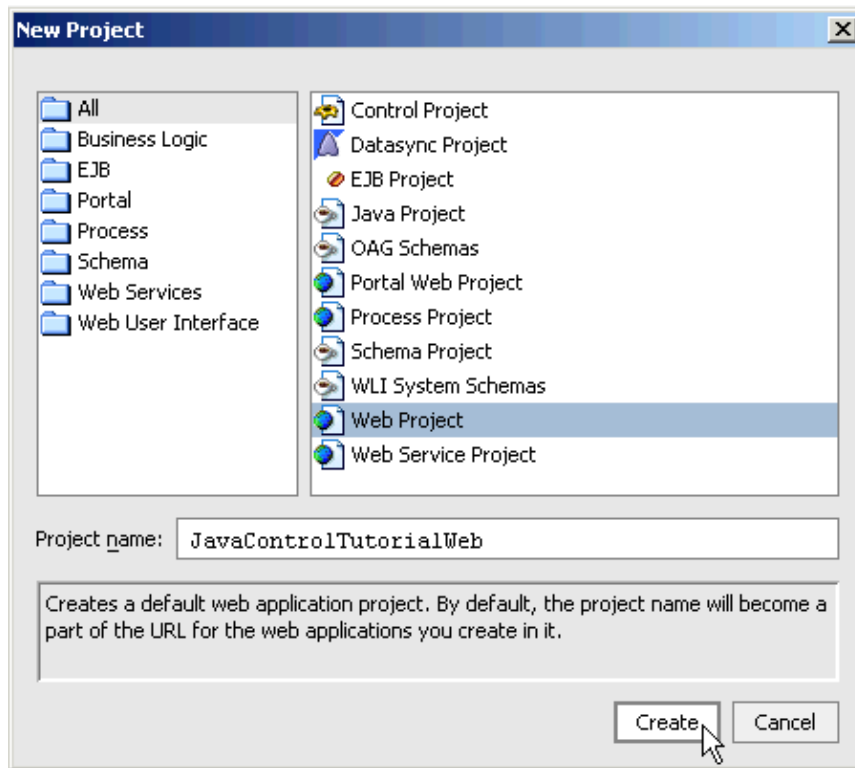


The folder ***CreditCardReport\_JCSTutorial*** contains a web service that your Java control will utilize. This web service offers credit card information on loan applicants. The ***Read\_Me*** folder contains a message directing users to this tutorial. This message is provided in case a user should open a new Java Control Tutorial Application by accident. The ***Schemas*** folder contains schema files used for parsing XML documents. The ***Modules*** folder is provided with every WebLogic application: use this folder to store stand-alone applications. The ***Libraries*** folder is used to store JAR files and other application resources. The ***Security Roles*** folder is used to define the security roles used in your application.

## To Create a New Project

For the sake of convenience, you will develop your Java control within a web application project instead of a Java control project. This way, it is easier to generate and use Workshop's testing support. When the Investigate Java control is complete, you will move the source code to a Java control project for final packaging.

1. On the ***Application*** tab, right-click the ***JavaControlTutorial*** folder and select ***New-->Project***.
2. In the ***New Project*** dialog, in the upper left-hand pane, confirm that ***All*** is selected.  
 In the upper right-hand pane, select ***Web Project***.  
 In the ***Project Name*** field, enter ***JavaControlTutorialWeb***.

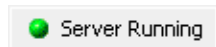


3. Click **Create**.

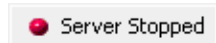
To Start WebLogic Server

Since you will be deploying and running your Java control on WebLogic Server, it is helpful to have WebLogic Server running during the development process.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.



If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow the instruction below to start WebLogic Server.

- **To start WebLogic Server:** from the **Tools** menu, choose **WebLogic Server**—>**Start WebLogic Server**.

**Note:** on the **WebLogic Server Progress** dialog, you may click **Hide** and continue to the next task.

To Create the Investigate Java Control

You are now ready to begin development of your Java control.

1. On the **Application** tab, right-click the **JavaControlTutorialWeb** folder and select **New**—>**Folder**.

2. In the **Create New Folder** dialog, enter `investigateJCS`, as shown in the following illustration.

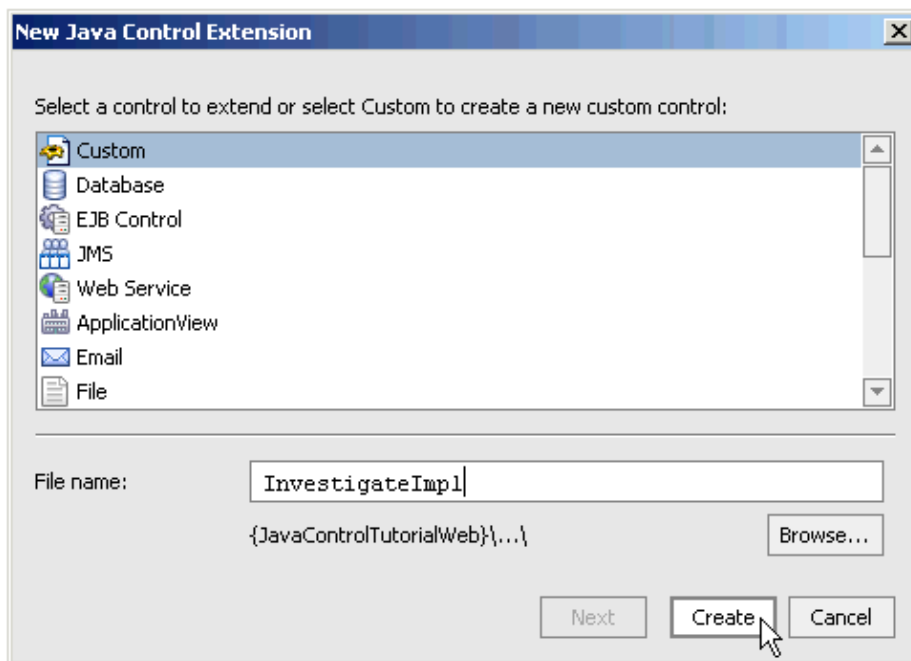


3. Click **OK**.

A new folder named *investigateJCS* is created inside the *JavaControlTutorialWeb* folder.

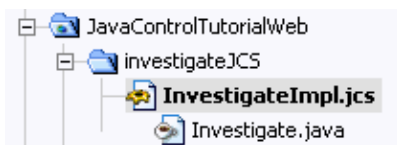
4. Right-click the *investigateJCS* folder and select **New—>Java Control**.
5. In the **New Java Control Extension** dialog, select **Custom**.

In the **File Name** field, enter `InvestigateImpl`. ("Impl" indicates that this file contains the implementation code for your control.)



6. Click **Create**.

Note that *two* files are created: a JCS (Java Control Source) file, and a Java file.



The JCS file contains the implementation code for your Java control, the Java file contains the public interface for the underlying implementation code. The Java file is the public face of your Java control: it is essentially a list of methods that users can call to access the functionality contained within the underlying implementation code. For the most part, you do not have to directly edit the Java file, because WebLogic Workshop automatically maintains the Java file as you edit the JCS file.

### To Create the Applicant Class

The code you add below, the Applicant class, contains fields for storing the information about credit applicants, such as the applicant's name, currently available credit, and so on. Each time your Java control is invoked a new instance of the Applicant class is created to store data on a particular applicant.

(Note that the Applicant class implements the Serializable interface. This allows WebLogic Workshop to save a particular Applicant object to disk while it is assembling a credit-worthiness report. This is important for two reasons. First, assembling a credit-worthiness report may take a long time, so it is convenient to be able to save data to disk instead of keeping it in memory. Second, the Investigate Java control will ultimately be exposed through a web service access point and web services *require* that the data they transport over the wire be serializable.)

1. On the **Application** tab, right-click the *investigateJCS* folder and select **New** --> **Java Class**. The **New File** dialog appears.
2. In the **New File** dialog, in the upper right-hand pane, confirm that **Java Class** is selected. In the **File name** field enter Applicant.java. Click **Create**.  
A new Java file, Applicant.java, is created.
3. Edit Applicant.java to look like the following. Add the elements appearing in red.

```
package investigateJCS;

public class Applicant implements java.io.Serializable
{
    /*
     * This long allows for versioning of this class.
     */
    static final long serialVersionUID = 1L;

    public String taxID;
    public String firstName;
    public String lastName;
    public boolean currentlyBankrupt;
    public int availableCCCredit;
    public int creditScore;
    public String approvalLevel;
    public String message;

    public Applicant() {}
}
```

4. Press **Ctrl+S** to save Applicant.java, then press **Ctrl+F4** to close the file.

### To Add a Member Variable

Each time a new report is requested from the Investigate Java control, a new instance of the Applicant object is created to store the report data. The following code constructs a new Applicant object each time a new report is requested.

1. Confirm that *InvestigateImpl.jcs* is displayed in the main work area. (To display any file in the main work area, double-click its icon on the **Application** tab.)
2. Click the **Source View** tab.
3. Add the following code to body of InvestigateImpl.

## WebLogic Workshop Tutorials

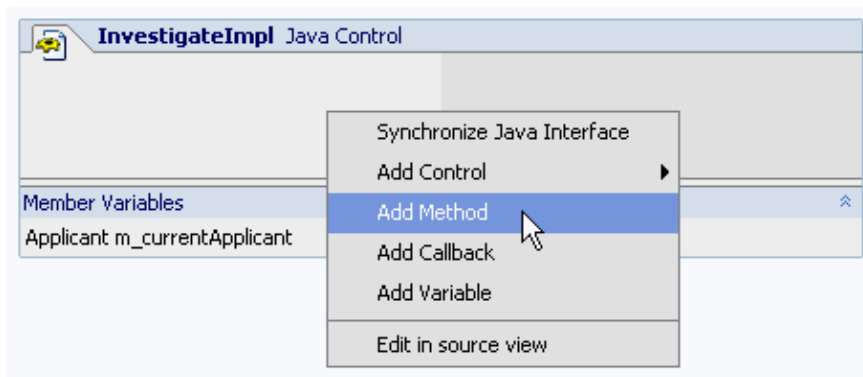
```
/*  
 * Construct a new instance of the Applicant class to store  
 * the credit report data.  
 */  
public Applicant m_currentApplicant = new Applicant();
```

4. Press **Ctrl+S** to save your work.

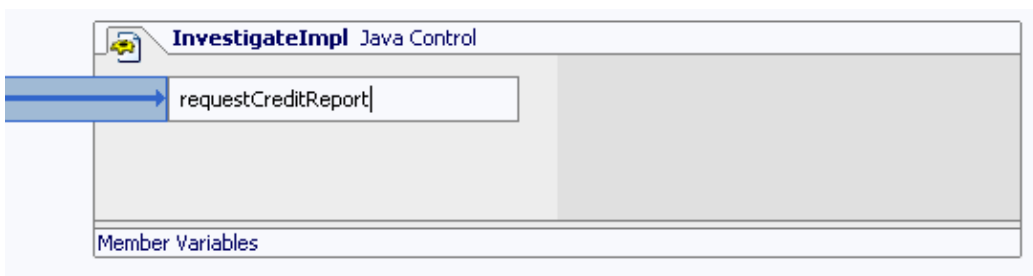
### To Add a Method

In this task you add a method through which users will request a new credit report. When users invoke this method, it starts off the process of assembling a credit report. (When the credit report is done, the report will be sent back to the user through a *callback method*, added below.)

1. Confirm that **InvestigateImpl.jcs** is displayed in the main work area.
2. Click the **Design View** tab.
3. In **Design View**, right-click the picture of your Java control and select **Add Method**.



4. In the space provided, replace newMethod1 with requestCreditReport and press **Enter**.



**Note:** If you switched from WebLogic Workshop to another application (for example, to read this topic), the method name may no longer be available for editing. To re-open it for editing, right click its name, select **Rename**, type requestCreditReport, and then press **Enter**.

5. Press **Ctrl+S** to save your work.

### To Add a Message Buffer

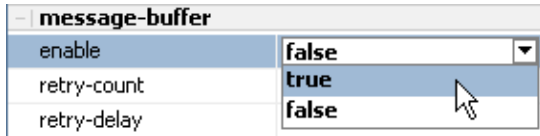
By adding a message buffer to a method, the method immediately returns a simple method invocation acknowledgement to the user. This allows the user to continue with its own processes while the requestCreditReport method executes.



1. In **Design View** select the **requestCreditReport** method. You select a method by clicking the arrow icon next to the method's name.



2. On the **Property Editor** tab, in the section labeled **message buffer**, select the **enable** property. Using the drop-down list, set the value to **true**.



A buffer icon is added to the method's arrow icon, as shown below.



## To Specify a Parameter and Method Body

The Investigate Java control assembles a credit report based on a subject's tax identification number. In this task you will add a parameter to the requestCreditReport method through which you can specify the subject's tax identification number.

You will also add code to the method body that assembles a credit report on the applicant. For the time being the assembly process is not very informative about the applicant, but as you progress in the tutorial, the assembly process will become more complex.

1. In **Design View**, click the name of the **requestCreditReport** method, as shown in the following illustration.



Clicking the method's name in Design View, displays the method's source code in Source View.

Note the two Javadoc annotations that appear above the requestCreditReport method.

```
/**
 * @common:operation
 * @common:message-buffer enable="true"
 */
public void requestCreditReport()
{
}
```

The Javadoc annotation `@common:operation` indicates that this method is part of the public interface of `InvestigateImpl.js`. When this annotation is present, WebLogic Workshop includes this method as part of the public interface listed in `Investigate.java`. In Design View, this annotation is depicted by the blue arrow icon.

## WebLogic Workshop Tutorials

The Javadoc annotation `@common:message-buffer` indicates that this method immediately returns an acknowledgment to the user after he invokes the method. In Design View this annotation is depicted by the buffer icon.

2. Edit the ***requestCreditReport*** method declaration and body so that they appear as follows. Code to add appears in red. Do *not* edit the annotations above the method declaration.

```
public void requestCreditReport(String taxID)
{
    /*
     * Assemble a credit worthiness report on the applicant.
     */
    m_currentApplicant.taxID = taxID;
    m_currentApplicant.firstName = "unknown";
    m_currentApplicant.lastName = "unknown";
    m_currentApplicant.approvalLevel = "unknown";
    m_currentApplicant.message = "There was insufficient data available to assemble a report";
}
```

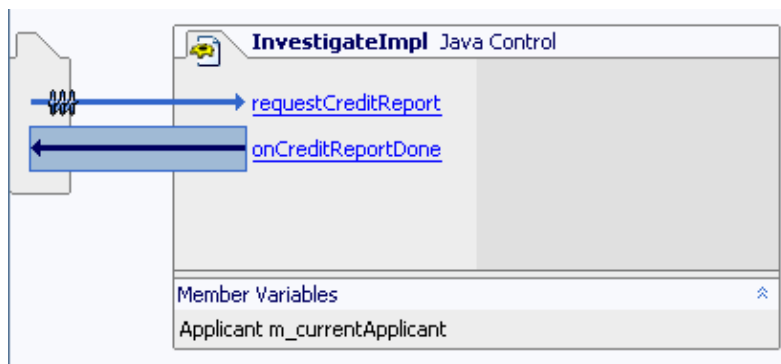
3. Press ***Ctrl+S*** to save your work.

### To Add a Callback

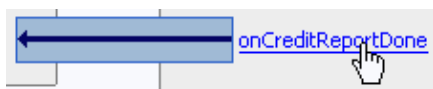
When the Investigate Java control has assembled a complete credit report it sends the report to the user through a *callback method*. In this task you will add the callback method through which users will receive the completed credit report. You will also add a parameter to your callback method. When you add a parameter to a callback method, the parameter you add will be sent to the client of your Java control.

1. Confirm that ***InvestigateImpl.jcs*** is displayed in the main work area.
2. Click the ***Design View*** tab.
3. In ***Design View***, right-click the picture of your Java control and select ***Add Callback***.
4. In the space provided, replace `newCallback1` with `onCreditReportDone` and press ***Enter***.

A callback is added to your Java control.



5. Click the name of the callback ***onCreditReportDone***.



Clicking a callback opens the control's interface file (a Java file).

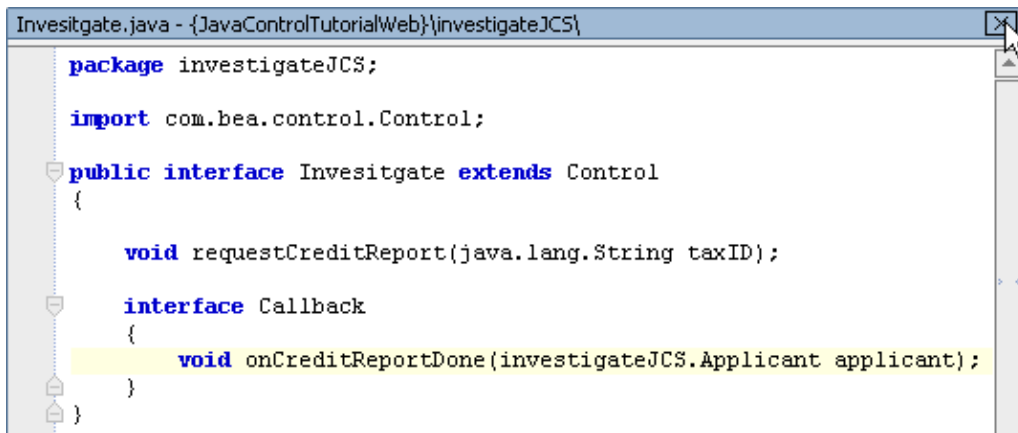
6. Edit the Callback interface so it appears as follows.

```
interface Callback
{
```

## WebLogic Workshop Tutorials

```
    void onCreditReportDone(investigateJCS.Applicant applicant);  
}
```

7. Press **Ctrl+S** to save your work.
8. Click the **X** in the upper right-hand corner of the main work area to close the Investigate.java file.



### Add Code to Invoke the Callback After the Credit-worthiness Report Has Been Assembled

At this point the Investigate Java control has (1) a method through which a client can request a credit-worthiness report (`requestCreditReport`) and (2) a callback methods which sends the credit-worthiness report to the client (`onCreditReportDone`). In this following task you will connect these two methods, so that when the credit-worthiness report is complete, it will be sent to the client.

1. Confirm that **InvestigateImpl.jcs** is displayed in the main work area.
2. If necessary, click the **Design View** tab.
3. Click the name of the **requestCreditReport** method.



4. Edit the source code for the **requestCreditReport** method so it appears as follows. Code to add appears in red.

```
public void requestCreditReport(String taxID)  
{  
    /*  
     * Assemble a credit-worthiness report on the applicant.  
     */  
    m_currentApplicant.taxID = taxID;  
    m_currentApplicant.firstName = "unknown";  
    m_currentApplicant.lastName = "unknown";  
    m_currentApplicant.approvalLevel = "unknown";  
    m_currentApplicant.message = "There was insufficient data available to assemble a  
  
    /*  
     * Send the credit-worthiness report to the client via a callback.  
     */  
    callback.onCreditReportDone(m_currentApplicant);  
}
```

5. Press **Ctrl+S** to save your work.

As it is now written, the Investigate Java control works like this: the user begins the process by invoking the `requestCreditReport` method, supplying a tax identification number as a method parameter. The `requestCreditReport` then immediately returns a request acknowledgement to the user (this is the role of the message buffer). Next the Investigate Java control assembles a credit report on the applicant. Finally the completed credit report is sent back to the user through the callback `onCreditReportDone`.

### To Generate a Test Client for the Investigate Java Control

In this task you will generate a client to test your Java control. The test client is a web service that allows you to invoke the methods of the Investigate Java control and view the results of those method invocations.

1. On the **Application** tab, right-click *InvestigateImpl.jcs* and select **Generate Test JWS File (Conversational)**.

A new web service, *InvestigateTest.jws*, is created and placed in the *investigateJCS* folder.

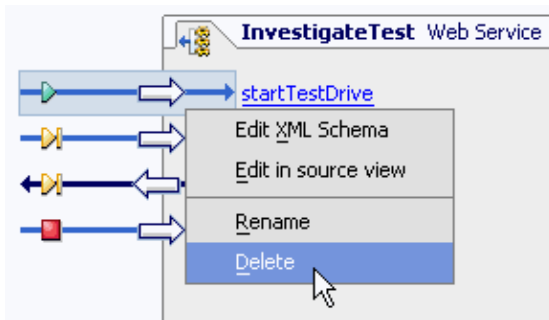
2. On the **Application** tab, double-click *InvestigateTest.jws* to open the file.



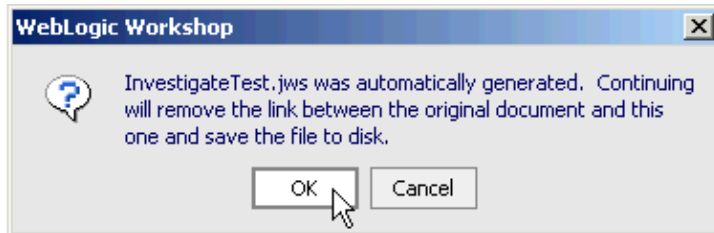
When you auto generate a test web service, WebLogic Workshop creates a set of web service methods corresponding to the methods of your Java control plus two more: a `startTestDrive` method and a `finishTestDrive` method. These two extra methods are used to start and finish a web service *conversation*. Conversations are used to group together the different method calls and callbacks into a single session. This is especially useful if you make multiple requests for credit reports: it lets you associate each invocation of the `requestCreditReport` method with the corresponding credit report returned by the callback `onCreditReportDone`. The green, yellow, and red icons on the web service methods indicate the *phases* of the conversation: a green icon indicates that the method starts a new conversation, a yellow icon indicates that the method continues an already existing conversation, and a red icon indicates that the method ends an already existing conversation.

Autogenerated test web services are optimized for Java controls with complex public interfaces. In a typical test cycle you would start a new conversation by invoking the `startTestDrive` method, continue the conversation by invoking the methods corresponding to your Java control, and finally end the conversation by invoking the `finishTestDrive` method. But our Java control has a simple public interface, so you will modify the test web service to make it easier to use.

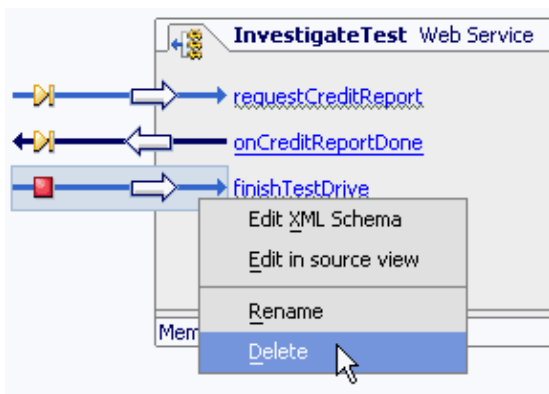
3. On the **Design View** tab, right-click the *startTestDrive* method and select **Delete**.



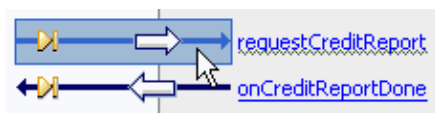
4. In the dialog warning you that you are attempting to edit an auto generated file, click **OK**.



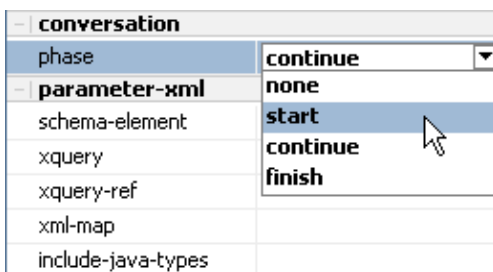
5. On the **Design View** tab, right-click the *finishTestDrive* method and select **Delete**.



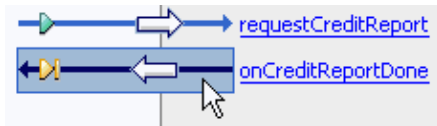
6. On the **Design View** tab, select the *arrow icon* for the *requestCreditReport* method.



7. On the **Property Editor** tab, in the section labeled **conversation**, find the **phase** property. Using the drop-down list, set the value to **start**.



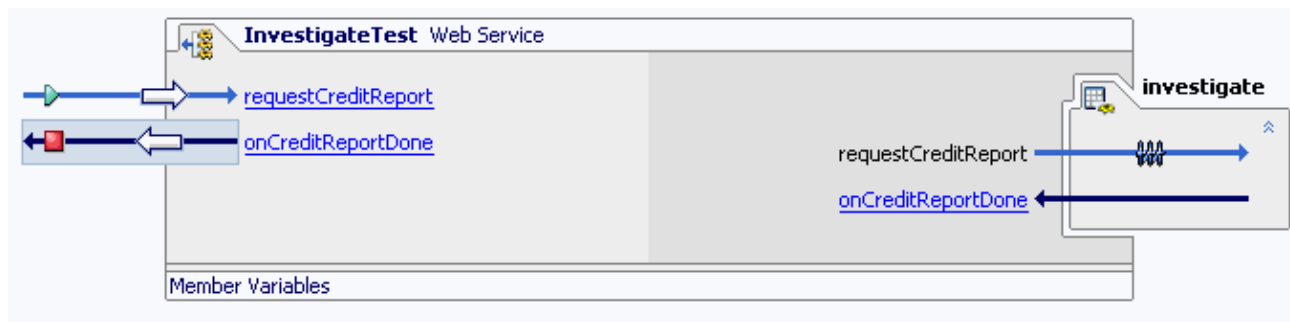
8. On the **Design View** tab, select the *arrow icon* for the *onCreditReportDone* method.



9. On the **Property Editor** tab, in the section labeled **conversation**, find the **phase** property. Using the drop-down list, set the value to **finish**.

<b>conversation</b>	
phase	continue
parameter-xml	none
schema-element	continue
xquery	finish
xquery-ref	
xml-map	
include-java-types	

Design View should look like the following.



10. Click the **Source View** tab.  
 11. Edit the **requestCreditReport** method so it looks like the following. Add the code appearing in red.

```

/**
 * <p>Use the following taxID's to test the Investigate web service.</p>
 *
 * <blockquote>111111111<br>
 * 222222222<br>
 * 333333333<br>
 * 444444444<br>
 * 555555555<br>
 * 123456789</blockquote>
 *
 * @common:operation
 * @jws:conversation phase="start"
 */
public void requestCreditReport(java.lang.String taxID)
{ investigate.requestCreditReport(taxID); }

```

12. Press **Ctrl+S** to save your work.

### To Test the Investigate Java Control

In the following task you will use the test web service to invoke the methods of the Investigate Java control. The test web service also lets you view detailed information about the methods while they are executing.

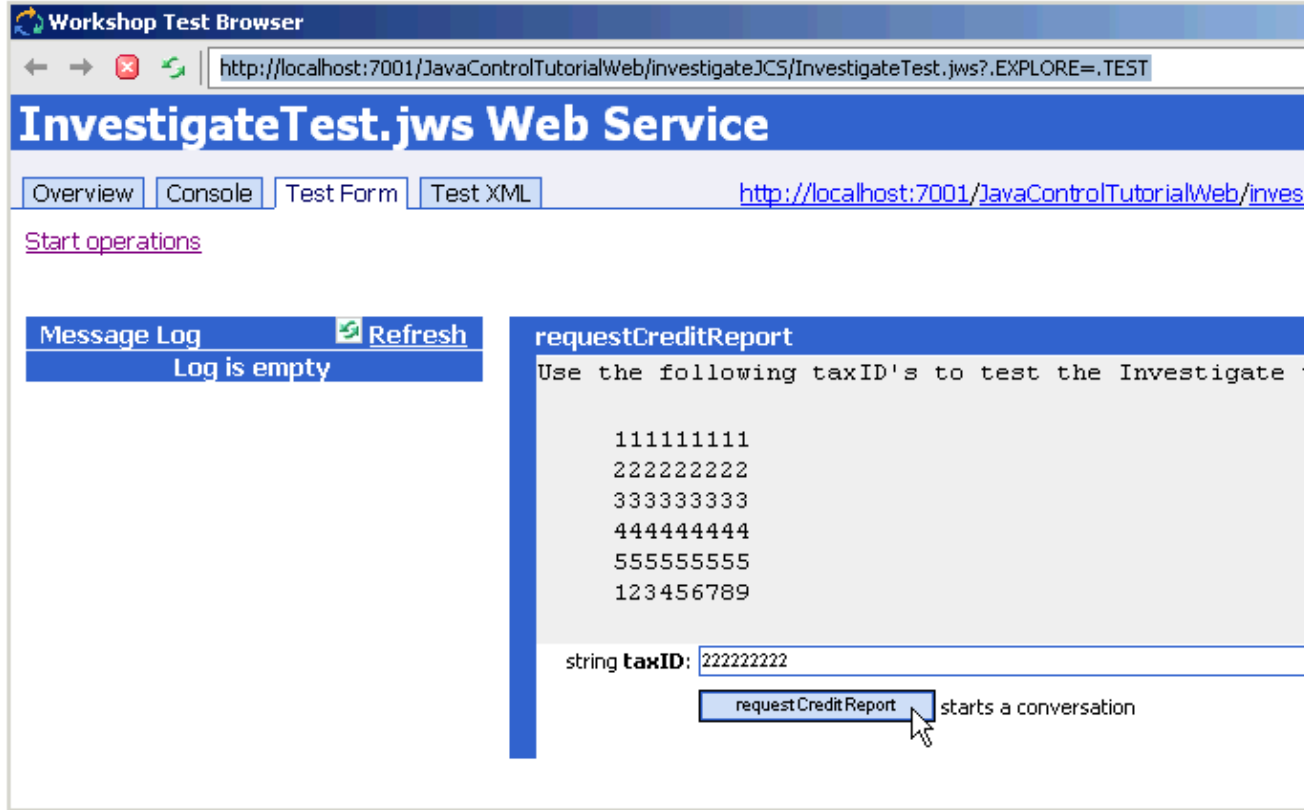
1. Confirm that **InvestigateTest.jws** is displayed in the main work area.

- Click the **Start** button.

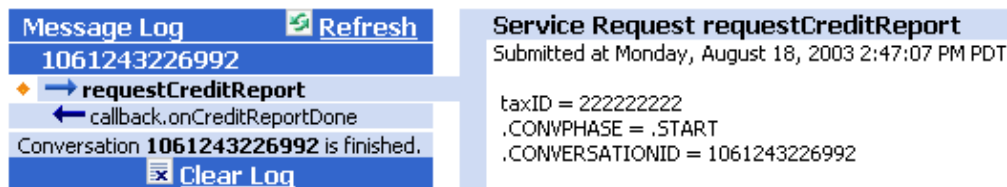


Workshop builds InvestigateTest.jws and launches the Workshop Test Browser.

- In the **Workshop Test Browser**, in the **taxID** field, enter some value and click the **requestCreditReport** button.



- Click **Refresh** until the **callback.onCreditReportDone** appears in the **Message Log**.



In the image above, the **Message Log** contains information about the method invocation **requestCreditReport** and the callback **onCreditReport**. The right-hand column, underneath the heading **Service Request requestCreditReport**, gives detailed information about the time the **requestCreditReport** method was invoked, the parameter that was entered, the conversation phase, and the conversation ID that was assigned to this method invocation.

In the left-hand column, note the number appearing directly above **requestCreditReport**. This is the conversation ID that groups together the **requestCreditReport** method and the **onCreditReport** callback. This conversation ID appearing in your Test Browser will be different than the one illustrated here: this is because a new conversation ID is generated each time the **requestCreditReport** method is invoked.

The right-hand column (not fully pictured here, see your Test Browser) gives you the run-time implementation details about how the web service and the Investigate Java control interact. (As you can see, from a run-time perspective their interaction is fairly complex. But from an application development perspective you do not have to worry about these details, they are taken care of by the WebLogic Workshop runtime.) The most important part of this interaction is the invocation of the `requestCreditReport` method on the Investigate Java control and the acknowledgment it immediately returns to the web service client (illustrated below). Recall that the acknowledgment is returned because you placed a message buffer on the `requestCreditReport` method.

**Operation requestCreditReport**  
Submitted at Monday, August 18, 2003 2:47:07 PM PDT  
Method: investigateJCS.InvestigateImplTest.requestCreditReport  
Arguments:  
taxID : 22222222  
CallStack:  
requestCreditReport()

**Returned from requestCreditReport**  
Submitted at Monday, August 18, 2003 2:47:07 PM PDT

5. In the *Message Log*, click *callback.onCreditReportDone*.

**Message Log** Refresh

1061243226992  
→ requestCreditReport  
◆ ← **callback.onCreditReportDone**  
Conversation 1061243226992 is finished.  
 Clear Log

**Client Callback**  
Submitted at Monday, August 18, 2003 2:47:07 PM PDT

<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
<SOAP-ENV:Header>  
<CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversationID">1061243226992</conversationID>  
</CallbackHeader>  
</SOAP-ENV:Header>  
<SOAP-ENV:Body>  
<ns:onCreditReportDone xmlns:ns="http://www.openuri.org/">  
<ns:applicant>  
<ns:taxID>22222222</ns:taxID>  
<ns:firstName>unknown</ns:firstName>  
<ns:lastName>unknown</ns:lastName>  
<ns:currentlyBankrupt>false</ns:currentlyBankrupt>  
<ns:availableCCCredit>0</ns:availableCCCredit>  
<ns:creditScore>0</ns:creditScore>  
<ns:approvalLevel>unknown</ns:approvalLevel>  
<ns:message>There was insufficient data available to as  
</ns:applicant>  
</ns:onCreditReportDone>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>

Notice that the results are displayed in the form of an XML document. This is because the test web service converts the Applicant object into an XML message before it is displayed in the test browser. (Note that web services *always* send XML messages to their clients, typically in a SOAP format.)

Also notice the `<conversationID>` element. This is the same conversation ID that was assigned to the `requestCreditReport` method when the conversation began.

6. Return to *WebLogic Workshop* and press the *Stop* button to close the Test Browser.



Related Topics



Designing Asynchronous Interfaces

Building Custom Java Controls

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



## Step 2: Add a Database Control

In this step you add a database control to the Investigate Java control. The database control provides access to a database containing bankruptcy and other information about credit applicants.

The database control you add is a pre-built control provided by WebLogic Workshop. Pre-built controls act as interfaces to common data resources, such as databases, web services, Java Message Services, and Enterprise Java Beans. Pre-built controls free you from having to design a new interface each time you want to connect to one of these kinds of resources.

The tasks in this step are:

- To add the Record class
- To create a database control file
- To add a method
- To associate a SQL query with the method
- To edit the Investigate Java control code to incorporate the database control file
- To test the Investigate Java control using the debugger

### To Add the Record Class

The Record class is a Java object that represents an individual record within a database. In particular, it represents an individual record of the BANKRUPTCIES table. In the following tasks you will create a database control file that queries the BANKRUPTCIES table and then returns a Record object containing the results of the query.

1. Right-click the *investigateJCS* folder and select *New*—>*Java Class*.
2. In the *New File* dialog, in the *File Name* field enter Record.java.
3. Click *Create*.
4. Edit the Record.java file so it appears as follows. Code to add appears in red.

```
package investigateJCS;

public class Record
{
    static final long serialVersionUID = 1L;

    public String firstname;
    public String lastname;
    public String taxID;
    public boolean currentlyBankrupt;
}
```

5. Press **Ctrl+S** to save your work. Press **Ctrl+F4** to close Record.java.

### To Create a Database Control File

In this task you will create a database control file, called BankruptciesDatabase.jcx. "JCX" stands for "Java Control Extension". A JCX file *extends* one of Workshop's pre-built control classes, in this case the com.bea.control.DatabaseControl class, which allows easy access to a database.

## WebLogic Workshop Tutorials

1. On the **Application** tab, double-click **InvestigateImpl.jcs** to display the file in the main work area. (Throughout this tutorial make sure that you do not confuse InvestigateImpl.jcs and InvestigateTest.jws. They are easy to confuse because of their similar names.)
2. Click the **Design View** tab.
3. From the **Insert** menu, choose **Controls**—>**Database**.

The **Insert Control** dialog appears.

4. Enter values as shown in the following illustration:

**Insert Control - Insert Database**

**STEP 1** Variable name for this control: bankruptciesDB

**STEP 2** I would like to :

☐ Use a Database control already defined by a JCX file

JCX file:  Browse...

☒ Create a new Database control to use.

New JCX name: BankruptciesDatabase

☐ Make this a control factory that can create multiple instances at runtime

**STEP 3** Connection name for this Database control

Data source: cgSampleDataSource Browse...

Create Cancel

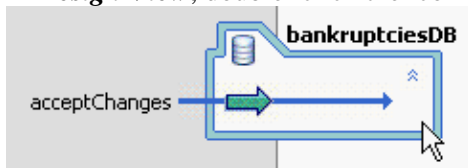
5. Click **Create**.

A new JCX file called BankruptciesDatabase.jcx is created in the *investigateJCS* folder and a new database control icon appears in **Design View**.

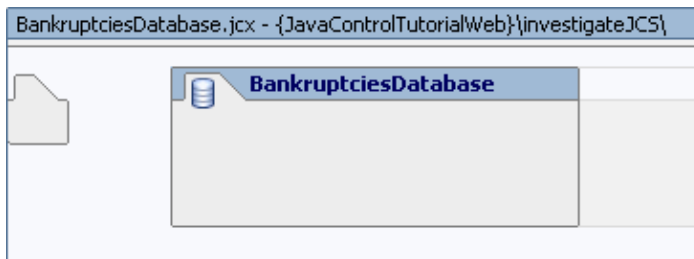
### To Add a Method

In this task, you will add a method to the database control file. This method will ultimately be able to query a database and return a Record object containing the results of the query.

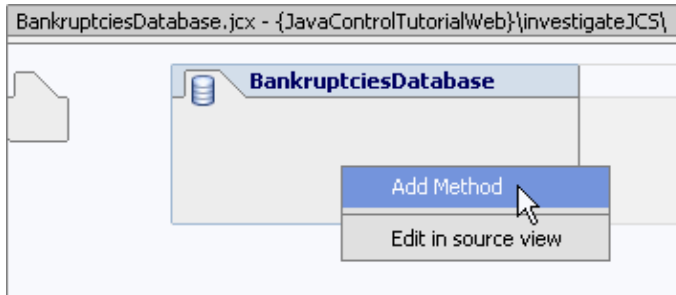
1. In **Design View**, double-click the icon of the database control.



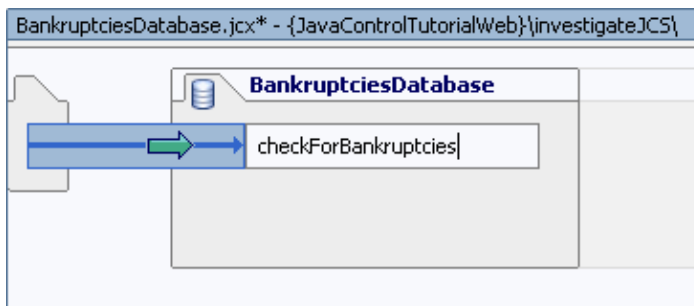
The database control file, BankruptciesDatabase.jcx, appears in **Design View**.



2. Right-click the picture of the database control and select **Add Method**.



3. In the space provided, replace newMethod1 with checkForBankruptcies and press **Enter**.

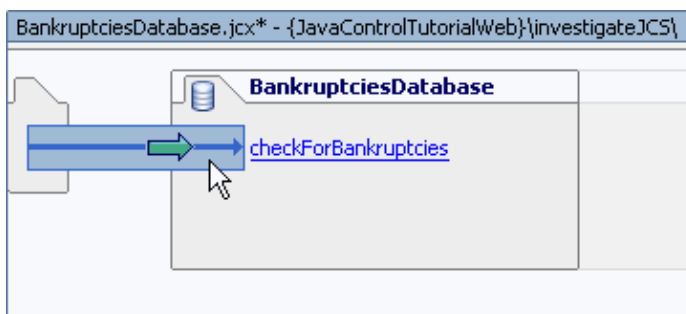


A new method is added to the database control file.

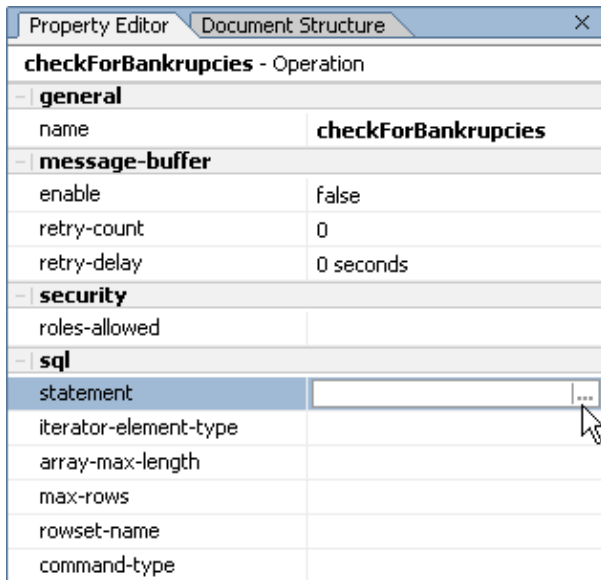
## To Associate a SQL Query with the Method

In this task you will associate a SQL query with the checkForBankruptcies method, so that when the method is invoked, it will query the BANKRUPTCIES table. The results of the query will be returned by the method as a return value.

1. Click the blue arrow icon for the **checkForBankruptcies**.



2. On the **Property Editor** tab, in the section labeled **SQL**, find the property named **statement**. To the far right side of the value field, click the **elipses** symbol.



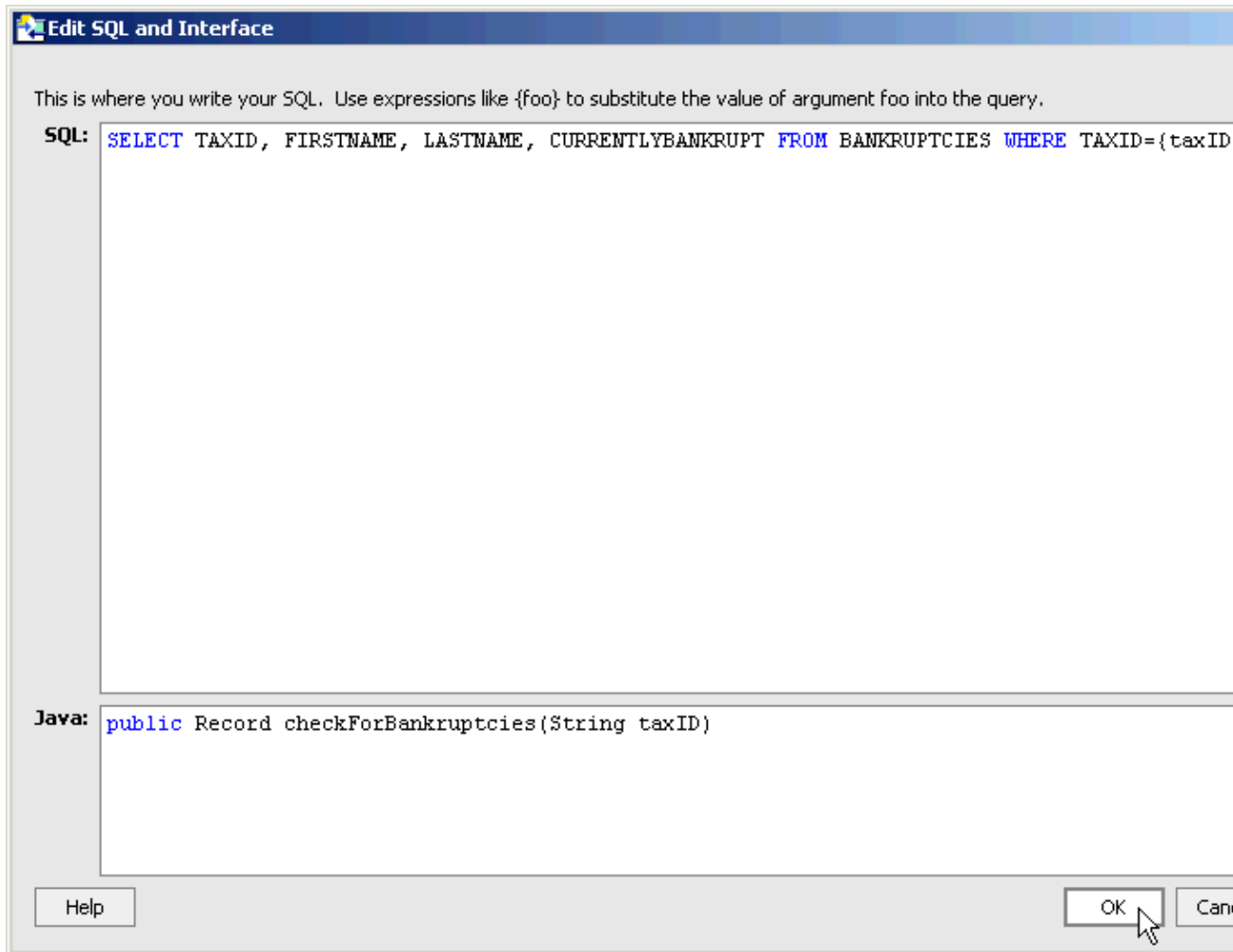
3. In the *Edit SQL and Interface* dialog, enter values as shown in the illustration. Note that the illustration does not contain selectable text. As a convenience, use the selectable text shown in red. Also: to enter text into the dialog, select the target pane and press **Ctrl+V**.

Enter into the *SQL* pane...

```
SELECT TAXID, FIRSTNAME, LASTNAME, CURRENTLYBANKRUPT FROM BANKRUPTCIES WHERE TAXID={taxID}
```

Enter into the *Java* pane...

```
public Record checkForBankruptcies(String taxID)
```



4. Click **Ok**.
5. Click the name of the method *checkForBankruptcies*.



The source code for the `checkForBankruptcies` method should appear as follows.

```
/**
 * @jcs:sql statement="SELECT TAXID, FIRSTNAME, LASTNAME, CURRENTLYBANKRUPT FROM BANKRUPTCIES WHERE TAXID={taxID}"
 */
public Record checkForBankruptcies(String taxID);
```

The `checkForBankruptcies` method works in the following way. (1) A client, in this case the Investigate Java control, invokes the `checkForBankruptcies` method by passing in an applicant's `taxID` as a parameter. (2) The `checkForBankruptcies` method then uses the `taxID` parameter to query the

BANKRUPTCIES table with a SQL query, appearing in the @jc:sql annotation. (3) Finally, the checkForBankruptcies method automatically constructs a Record object based on the results returned from the database and returns the Record object to the client. (The checkForBankruptcies method is able to automatically construct a Record object from the results of the database query because the field names of the BANKRUPTCIES table match those of the Record class.)

6. Press **Ctrl+S** to save your work and press **Ctrl+F4** to close BankruptciesDatabase.jcx.

To Edit the Investigate Java Control to Incorporate the Database Control File

In this task you will modify the Investigate Java control to use the database control. You will edit the method requestCreditReport to invoke the database control method checkForBankruptcies. The data returned by the checkForBankruptcies method will be stored in the member variable m\_currentApplicant.

1. If InvestigateImpl.jcs is not displayed, then, from the **Window** menu, select **InvestigateImpl.jcs**.
2. Click the **Source View** tab.
3. Edit the **requestCreditReport** method to look like the following. (Delete the entire body of the requestCreditReport method and replace it with the code shown in red below.)

```
public void requestCreditReport(String taxID)
{
    /*
    * Assemble a credit worthiness report on the applicant.
    */
    // m_currentApplicant.taxID = taxID;
    // m_currentApplicant.firstName = "unknown";
    // m_currentApplicant.lastName = "unknown";
    // m_currentApplicant.approvalLevel = "unknown";
    // m_currentApplicant.message = "There was insufficient data available to assemble
    */
    * Send the credit worthiness report to the client via a callback.
    */
    // callback.onCreditReportDone(m_currentApplicant);

    m_currentApplicant.taxID = taxID;

    /*
    * Retrieve data from the database and store it in the rec object.
    */
    Record rec = bankruptciesDB.checkForBankruptcies(taxID);

    /*
    * If the database returns substantial data, then store that data
    * in the m_currentApplicant object.
    */
    if(rec != null)
    {
        m_currentApplicant.firstName = rec.firstname;
        m_currentApplicant.lastName = rec.lastname;
        m_currentApplicant.currentlyBankrupt = rec.currentlyBankrupt;

        /*
        * Send the credit report to the client via a callback.
        */
        callback.onCreditReportDone(m_currentApplicant);
    }
    /*
    * If the database does not return substantial data, notify the client
    * that there is a problem.
    */
}
```

```

        */
    else
    {
        m_currentApplicant.message = "No data could be found on the applicant. Please
        try again."

        /*
        * Send the the error message to the client via a callback.
        */
        callback.onCreditReportDone(m_currentApplicant);
    }
}

```

4. Press **Ctrl+S** to save your work.

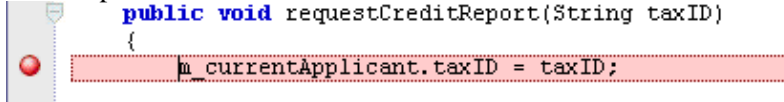
### To Test the Web Service Using the Debugger

Next you will test your Java control using the debugger. The debugger allows you to set breakpoints in your code and track how your code is running line-by-line. Setting a breakpoint in your code will cause the Java Virtual Machine to halt execution of your code immediately before the breakpoint, allowing you to step through your code beginning at that point.

1. Confirm that *InvestigateImpl.jcs* is displayed in the main work area. If necessary, click the **Source View** tab.
2. Place the cursor on the first line of code within the method `requestCreditReport`. (The first line of code is `"m_currentApplicant.taxID = taxID;"`)
3. Click the Toggle Breakpoint button on the toolbar, as shown here:



A breakpoint is set as shown below.



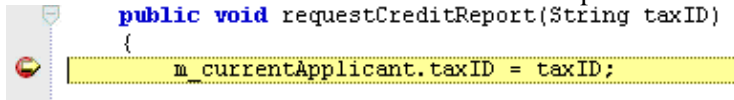
4. On the **Application** tab, double-click *InvestigateTest.jws*.
5. Press the **Start** button, shown below.



6. In the **Workshop Test Browser**, in the *taxID* field, enter the 9 digit number 222222222 and click the **requestCreditReport** button.

**Note:** The bankruptcies database contains data on 6 individuals. The taxIDs of these individuals are 123456789, 111111111, 222222222, 333333333, 444444444, and 555555555. Use these six taxIDs to test your Java control throughout the tutorial.

7. Note that the execution of code halts at the breakpoint as shown below.



8. On the **Locals** tab, expand the entries for *this* and *m\_currentApplicant*, as shown below.



Locals Watch Streams Immediate	
Name	Value
this	investigateJCS.InvestigateImpl #8205
bankruptciesDB	\$Proxy15 #8207
callback	\$Proxy14 #8208
m_currentApplicant	investigateJCS.Applicant #8209
approvalLevel	<NULL>
availableCCCredit	0
creditScore	0
currentlyBankrupt	false
firstName	<NULL>
lastName	<NULL>
message	<NULL>
taxID	<NULL>
taxID	"222222222"

- Click the Step Over button on the toolbar, shown here:



Each time you press the Step Over button, Workshop executes the current line of code and moves to the next line of code (within the current file).

```


public void requestCreditReport(String taxID)
{
    m_currentApplicant.taxID = taxID;

    /**
     * Retrieve data from the database and store it in the rec object.
     */
    Record rec = bankruptciesDB.checkForBankruptcies(taxID);
}

```

- Click the Step Over button until the web service finishes executing. The fields of the m\_currentApplicant object are modified as data is retrieved from the database. As you click the Step Over button, watch the values of m\_currentApplicant change on the Locals tab of the Debug Window. (Execution will stop after a total of 8 clicks of the Step Over button.)
- Return to the **Test Browser** and click **Refresh**.
- Under **Message Log**, click **callback.onCreditRepotDone**. Notice that the database has provided three new pieces of information about the applicant: a first name, last name, and his bankruptcy status.

## WebLogic Workshop Tutorials


**Message Log**  **Refresh**

**1061490580883**

→ requestCreditReport

◆ ← **callback.onCreditReportDone**


Conversation **1061490580883** is finished.

 **Clear Log**

### Client Callback

Submitted at Thursday, August 21, 2003 11:29:49 AM PDT

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1061490580883</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditReportDone xmlns:ns="http://www.openuri.org/">
      <ns:applicant>
        <ns:taxID>222222222</ns:taxID>
        <ns:firstName>John</ns:firstName>
        <ns:lastName>Smith</ns:lastName>
        <ns:currentlyBankrupt>false</ns:currentlyBankrupt>
        <ns:availableCCCredit>0</ns:availableCCCredit>
        <ns:creditScore>0</ns:creditScore>
      </ns:applicant>
    </ns:onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

13. Return to WebLogic Workshop and press **Ctrl+F9** to clear the breakpoint from InvestigateImpl.jcs.
14. Click the **Stop** button  to close the **Workshop Test Browser**

Related Topics

Debugging Your Application

Database Control

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



## Step 3: Add a Web Service Control

In this step you will add a web service control to the Investigate Java control. A web service control allows your Java control to communicate with a web service. This web service control is, like the database control in the previous step, a pre-built control provided by WebLogic Workshop. In this case, you will add a control for the Credit Card Report web service, a web service that provides credit card data on loan applicants. You should imagine that the Credit Card Report web service is an independent application that exists externally out on the Internet; but for the purposes of this tutorial, it has been included as a project within the Tutorial: Java Control application.

Because Credit Card Report is a web service, it returns an XML document to its clients. (By definition, web services return XML documents, typically SOAP XML documents.) In this task you will learn how to process these XML documents using XMLBean classes.

The tasks in this step are:

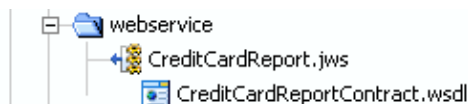
- To generate a WSDL file from a JWS file
- To generate a web service control from a WSDL file
- To add the web service control file to the Investigate Java control
- To add code to invoke the web service
- To add code to handle callbacks from the web service
- To test the Investigate Java control

To Generate a WSDL File from a JWS File

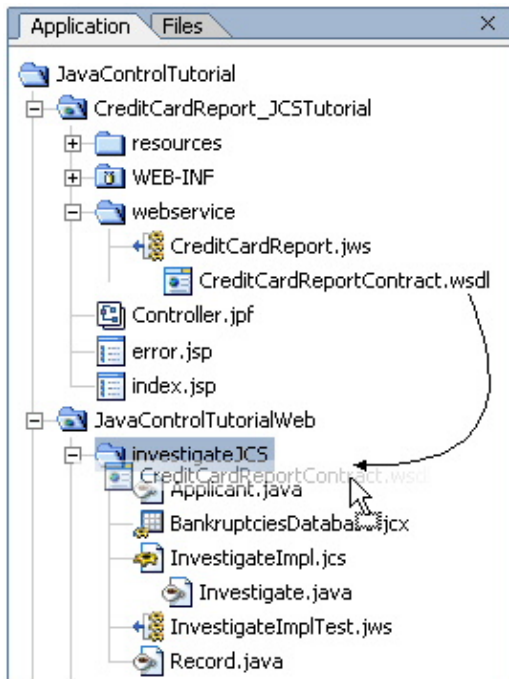
Web services advertise their functionality through WSDL files. WSDL files are like instruction manuals for a web service: they set out the available functionality of a web service and how to access that functionality. In this step you will generate a WSDL file for the Credit Card Report web service. In the next task you will generate a web service control from the WSDL file.

1. On the **Application** tab, navigate to **JavaControlTutorial/CreditCardReport\_JCSTutorial/webservice/CreditCardReport.jws**.
2. On the **Application** tab, right-click the **CreditCardReport.jws** file and select **Generate WSDL File**.

A new WSDL file, **CreditCardReportContract.wsdl**, is generated and placed underneath **CreditCardReport.jws**.



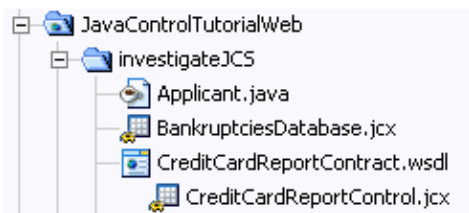
3. Hold the mouse button down over **CreditCardReportContract.wsdl** and drag it into the **JavaControlTutorial/JavaControlTutorialWeb/investigateJCS** folder.



To Generate a Web Service Control from a WSDL file

- On the **Application** tab, right-click **CreditCardReportContract.wsdl** and select **Generate Service Control**.

A new web service control file, **CreditCardReportControl.jcx**, is created and placed in the **investigateJCS** folder.



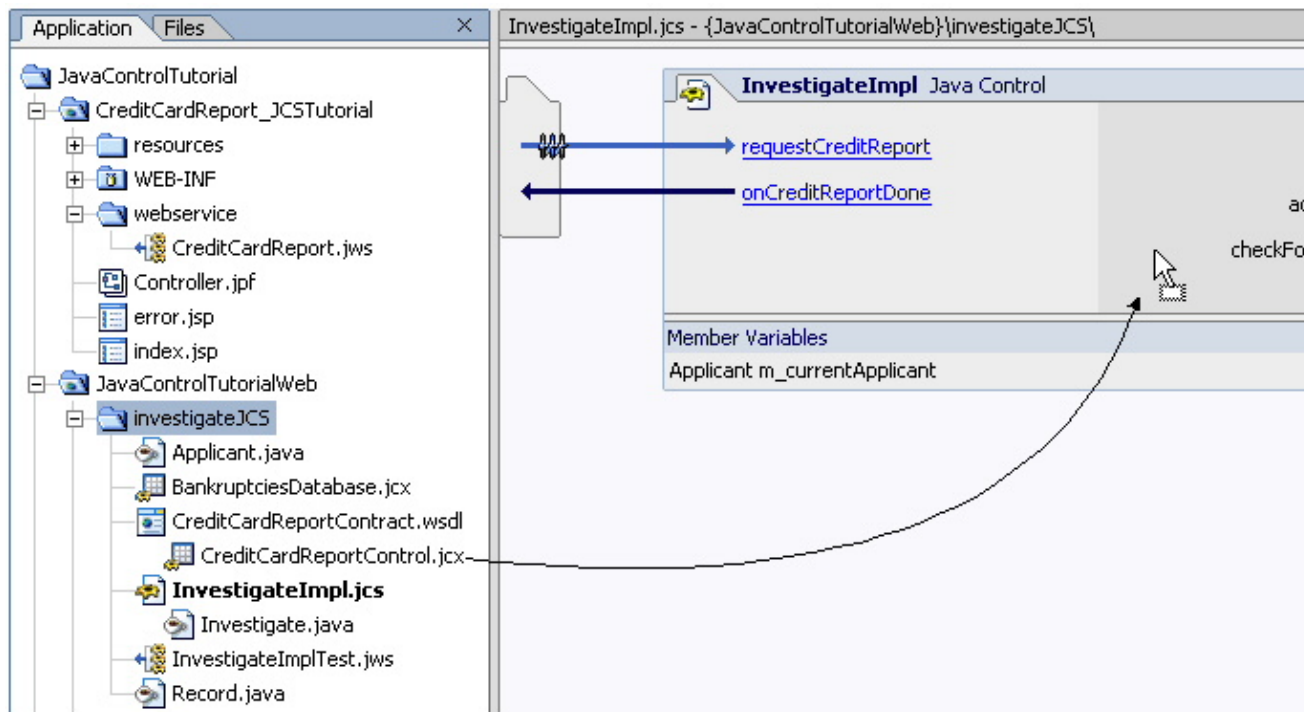
Note that the **CreditCardReportControl.jcx** file is a pre-built Workshop control that allows easy access to the **CreditCardReport** web service.

To Add a Web Service Control to The Investigate Java Control

In this task you will add the web service control **CreditCardReportControl.jcx** to the **Investigate Java Control**.

1. On the **Application** tab, double-click **InvestigateImpl.jcs**.
2. Click the **Design View** tab, if necessary. (Note make sure that **InvestigateImpl.jcs** is displayed in Design View and not **InvestigateTest.jws**. It is very easy to confuse a Java control and its test web service.)
3. From the **Application** Tab, click **CreditCardReportControl.jcx** and drag and drop it into **Design View**. A new web service control is added to the **Investigate Java control**.

## WebLogic Workshop Tutorials



The web service control is added to the Investigate Java control.



4. Press **Ctrl+S** to save your work.

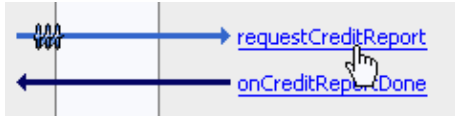
To Add Code that Invokes the Web Service

In this task you will add code that invokes the Credit Card Report web service. When the Credit Card Report web service is ready to provide data to the Investigate Java control, it will send it through a callback. Note that

## WebLogic Workshop Tutorials

data provided by the Credit Card Report web service is in the form of an XML document. In the next task you will modify the Investigate Java control so it can process XML documents.

1. Confirm that **InvestigateImpl.jcs** is displayed in the main work area.
2. Click name of the **requestCreditReport** method.



3. Edit the source code for the **requestCreditReport** method as shown below. Code to add is shown in red. Make sure that you remove the lines of code shown crossed out.

```
public void requestCreditReport(String taxID)
{
    m_currentApplicant.taxID = taxID;

    /*
     * Retrieve data from the database and store it in the rec object.
     */
    Record rec = bankruptciesDB.checkForBankruptcies(taxID);
    /*
     * If the database returns substantial data, then store that data
     * in the m_currentApplicant object.
     */
    if(rec != null)
    {
        m_currentApplicant.firstName = rec.firstname;
        m_currentApplicant.lastName = rec.lastname;
        m_currentApplicant.currentlyBankrupt = rec.currentlyBankrupt;

        /*
         * Send the credit report to the client via a callback.
         */
        callback.onCreditReportDone(m_currentApplicant);

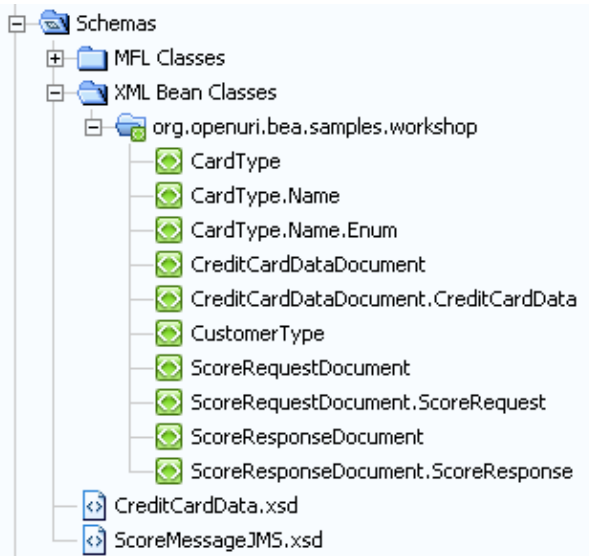
        /*
         * Invoke the Credit Card Report web service.
         * Results from the web service will be provided via a callback.
         */
        creditCardReportControl.getCreditCardData(taxID);
    }
    /*
     * If the database does not return substantial data, notify the client
     * that there is a problem.
     */
    else
    {
        m_currentApplicant.message = "No data could be found on the applicant. Please
        /*
         * Send an error message to the client via a callback.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }
}
```

### To Add Code to Handle Callbacks from the Web Service

When your Java control receives a callback from the Credit Card Report web service, it needs to know what to do with the data received. In this task you will add code to handle the XML document delivered by the Credit

Card Report web service. The code you add does three things: first, extracts the relevant data from the XML document. Second, it stores the extracted data in the member variable `m_currentApplicant`. Third, it sends the applicant profile back to the client using the callback `onCreditReportDone`.

The first task, the extraction of data from the XML document, is accomplished with the help of a set of XMLBean classes (`CreditCardDataDocument` and others), classes that know how to parse the XML documents returned by the web service. These XMLBean classes can be viewed in the **Schemas** project.



These XMLBean classes were produced by compiling a schema file (XSD file). Schema files are like "meta XML" documents. Schema files are like ordinary XML documents in that they obey ordinary XML syntax; but they are special in that they define the general shape of other XML documents, called "instances" of the schema file. When you compile a schema file in Workshop, XMLBean classes are produced that know how to parse any instances of the schema file. (Schema files can be compiled in a Schema type project.)

In this case, the XML messages returned by the Credit Card Report web service are instances of the schema file `CreditCardData.xsd`. When `CreditCardReport.xsd` is compiled, a set of XMLBean classes (`CardType`, `CardType.Name`, etc.) is produced that know how to parse any of the XML instance documents returned by the Credit Card Report web service.

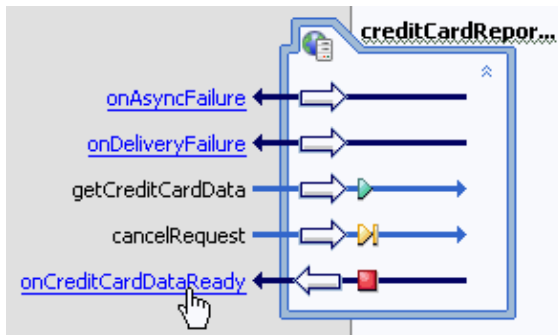
1. Confirm that ***InvestigateImpl.jcs*** is displayed in the main work area. If necessary, click the **Source View** tab.
2. Add the following import statements to the top of the `InvestigateImpl.jcs` file, directly underneath the package declaration.

Note that these are the XMLBean classes that know how to parse the XML documents returned by the Credit Card Report web service.

```
import com.bea.xml.XmlException;
import org.openuri.bea.samples.workshop.CardType;
import org.openuri.bea.samples.workshop.CustomerType;
import org.openuri.bea.samples.workshop.CreditCardDataDocument.CreditCardData;
import org.openuri.bea.samples.workshop.CreditCardDataDocument;
```

3. Click the **Design View** tab.

- Click the link text for the callback handler *onCreditCardDataReady*.



- Edit the callback handler *creditCardReportControl\_onCreditCardDataReady* to look like the following. Add the code appearing in red.

```
public void creditCardReportControl_onCreditCardDataReady(java.lang.String cardData)
{
    try
    {
        /*
         * When the XML document containing the credit card report is delivered,
         * extract the relevant credit card data, and store it in m_currentApplicant
         */
        /*
         * Note that the data is extracted with the aid of XMLBean classes, classes
         * produced from the compilation of the schema file CreditCardData.xsd.
         */
        CreditCardDataDocument cardInfo = CreditCardDataDocument.Factory.parse(cardData);
        CustomerType[] customer = cardInfo.getCreditCardData().getCustomerArray();
        CardType[] cards = customer[0].getCardArray();
        for(int i = 0; i < cards.length; i++)
        {
            m_currentApplicant.availableCCCredit += cards[i].getAvailableCredit();
        }

        /*
         * Send the complete report back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }
    catch(XmlException xe)
    {
        /*
         * If there is a problem with extracting the credit card info,
         * store an error message in m_currentApplicant.
         */
        m_currentApplicant.message = "There was an error retrieving the credit card data";

        /*
         * Send the the error message back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }
}
```

- Press **Ctrl+S** to save your work.

## To Test the Investigate Java Control

- On the *Application* tab, double-click *InvestigateTest.jws*.
- On the *toolbar*, click the *Start* button, shown here:





Workshop builds InvestigateTest.jws and launches the Test Browser.

3. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 22222222 and click the **requestCreditReport** button.

**Note:** Use one of the following (9 digit) taxID's to test your Java control throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click **Refresh** until **callback.onCreditReportDone** appears in the **Message Log**.
5. Under **Message Log**, click **callback.onCreditReportDone**.

Message Log	Refresh
1061503547455	
→ requestCreditReport	
investigate:creditCardReportControl.getCreditCardData→	
investigate:creditCardReportControl.onCreditCardDataReady←	
◆ ← <b>callback.onCreditReportDone</b>	
Conversation 1061503547455 is finished.	
Clear Log	

#### Client Callback

Submitted at Thursday, August 21, 2003 3:06:06 PM PDT

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader
      xmlns="http://www.openuri.org/2002/04/soap/conver
      <conversationID>1061503547455</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditReportDone xmlns:ns="http://www.openur
      <ns:applicant>
        <ns:taxID>22222222</ns:taxID>
        <ns:firstName>John</ns:firstName>
        <ns:lastName>Smith</ns:lastName>
        <ns:currentlyBankrupt>false</ns:currentlyBankrup
        <ns:availableCCCredit>2000</ns:availableCCCred
        <ns:creditScore>0</ns:creditScore>
      </ns:applicant>
    </ns:onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that a new piece of information has been added to the applicant profile:

<availableCCCredit>2000</availableCCCredit>. This information was provided by the Credit Card Report web service.

6. To see the entire credit card report provided by the web service, under **Message Log**, click the entry **investigate:CreditCardReportControl.onCreditDataReady**.

Message Log	Refresh
1061503547455	
→ requestCreditReport	
investigate:creditCardReportControl.getCreditCardData→	
◆ <b>investigate:creditCardReportControl.onCreditCardDataReady</b> ←	
← callback.onCreditReportDone	
Conversation 1061503547455 is finished.	
	Clear Log

**External Service Callback****investigate:creditCardReportControl.on**

Submitted at Thursday, August 21, 2003 3:06:06 PM

```

<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/creditCardReportControl/7fdf">
      <conversationID>[1061503547455]
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditCardDataReady xmlns:ns="http://www.openuri.org/creditCardReportControl/7fdf">
      <ns:cardData><wor:credit-card-data
        xmlns:wor="http://openuri.org/creditCardReportControl/7fdf"
        name="MasterCard" number="1111 1111 1111 1111"
        credit>1000</available-credit><credit-used></card><card name="Visa" number="4444 4444 4444 4444">
        <available-credit>1000</available-credit><credit-used>500</credit-used></card></ns:cardData>
      </ns:onCreditCardDataReady>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>

```

Notice that the web service returns the credit card data in the form of an XML document. It is this XML document that the XMLBeans parse when they extract the credit card data.

7. Return to WebLogic Workshop, and click the **Stop** button  to close the **Workshop Test Browser**

Related Topics

Web Service Control

Getting Started with XMLBeans

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



## Step 4: Add a JMS Control

In this step, you will add functionality to generate a credit score for the loan applicant.

To generate the credit score, you will use an in-house, legacy credit scoring application. (The legacy application is already running on WebLogic Server as part of the Workshop sample domain: this is the server you selected when you created the Tutorial: Java Control application.) To make this legacy application available to other components, it has been exposed through a Java messaging service (JMS). You will send an XML message to the credit scoring application containing loan applicant data, and receive an XML message containing a credit score.

Messaging systems are often used to create bridges between otherwise incompatible software components. Messaging is also useful when asynchronous communication is needed, since the requesting component doesn't have to wait for a response in order to continue working. Here, asynchrony is useful because you have no way of knowing how long the credit scoring application will take to respond to your request. For more information about messaging, see Overview: Messaging Systems and JMS.

The tasks in this step are:

- To add a JMS control
- To add code to construct an XML message
- To add code to invoke the JMS control
- To edit the callback handler to parse the XML response
- To test the Investigate Java Control

To Add a JMS Control

1. On the **Application** tab, double click **InvestigateImpl.jcs**. (Don't confuse your Java control with its test web service!)
2. In necessary, click the **Design View** tab.
3. From the **Insert** menu select **Controls --> JMS**.  
The **Insert Control – JMS** dialog appears.
4. Enter values as shown in the following illustration.

**Note:** Use the **Browse** buttons in the dialog below to fill in the *send-jndi-name*, *receive-jndi-name*, and *connection-factory* fields.

**Note:** The value for the *connection-factory* field is not fully displayed by the illustration below. If you enter the value manually instead of using the Browse button, you must enter **weblogic.jws.jms.QueueConnectionFactory**.

**Insert Control - JMS**

**STEP 1** Variable name for this control:

**STEP 2** I would like to :

☐ Use a JMS control already defined by a JCX file

JCX file:

☒ Create a new JMS control to use.

New JCX name:

☐ Make this a control factory that can create multiple instances at runtime

**STEP 3** Message type:

JMS send destination type:

Name of topic on which to send messages

send-jndi-name:

JMS receive destination type:

Name of topic on which to receive messages

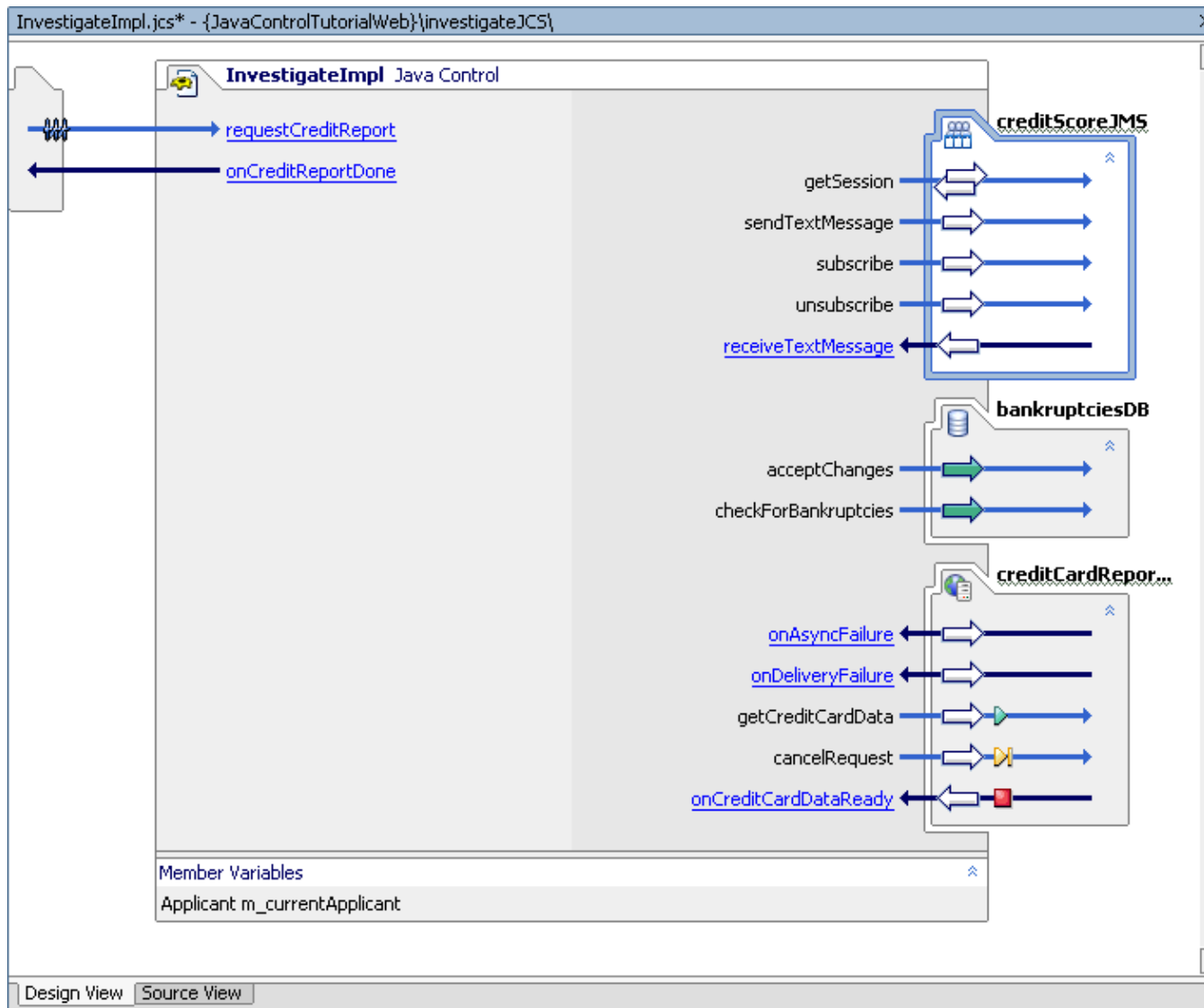
receive-jndi-name:

Connection factory to create connections to the topic

connection-factory:

5. Click **Create**.

Your Java control now includes the creditScoreJMS control, as shown in the following illustration.



### To Add Code to Construct an XML Message

The credit score application exposed over JMS is designed to expect an incoming XML message of the following shape.

```
<score_request>
  <credit_remaining>[integer_value]</credit_remaining>
  <is_bankrupt>[boolean_value]</is_bankrupt>
</score_request>
```

In this task you will add a method that constructs a message of this shape using XMLBeans compiled from the schema file ScoreMessageJMS.xsd, a schema file already included in the Schemas project.

1. Confirm that *InvestigateImpl.jcs* is displayed in the main work area.
2. Click the **Source View** tab.
3. Add the following import statements to the list of import statements at the top of InvestigateImpl.jcs, directly underneath the package declaration.

```
import org.openuri.bea.samples.workshop.ScoreRequestDocument;
```

### Step 4: Add a JMS Control

## WebLogic Workshop Tutorials

```
import org.openuri.bea.samples.workshop.ScoreRequestDocument.ScoreRequest;
import org.openuri.bea.samples.workshop.ScoreResponseDocument;
```

**Note:** these XMLBean classes know how to parse and construct the XML documents that form the JMS traffic between the Investigate Java control and the legacy credit scoring application. These XMLBean classes were produced by the compilation of the schema file ScoreMessageJMS.xsd.

4. Add the following method to the body of *InvestigateImpl.jcs*.

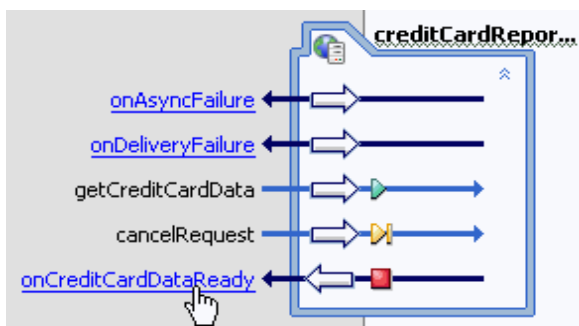
```
/**
 * This method uses the XMLBean ScoreRequestDocument (compiled from the XML schema
 * ScoreMessageJMS.xsd) to construct a string of the following form.
 *
 * <score_request>
 *   <credit_remaining>[integer_value]</credit_remaining>
 *   <is_bankrupt>[boolean_value]</is_bankrupt>
 * </score_request>
 */
private String makeMessageToJMS(int availableCredit, boolean currentlyBankrupt)
{
    ScoreRequestDocument reqDoc = ScoreRequestDocument.Factory.newInstance();
    ScoreRequest req = reqDoc.addNewScoreRequest();
    req.setCreditRemaining((short)availableCredit);
    req.setIsBankrupt(currentlyBankrupt);
    String str = reqDoc.xmlText();
    return reqDoc.xmlText();
}
```

5. Press **Ctrl+S** to save your work.

To Add Code to Invoke the JMS Control

In this task you will add code to invoke the JMS control.

1. Confirm that *InvestigateImpl.jcs* is displayed in the main work area.
2. Click the **Design View** tab.
3. Click the link text for the *onCreditCardDataReady* callback handler.



4. Edit the *creditCardReportControl\_onCreditCardDataReady* callback handler code so it appears as follows. Code to edit appears in red. Make sure to delete the lines of code shown crossed out below.

```
public void creditCardReportControl_onCreditCardDataReady(java.lang.String cardData)
{
    try
    {
        /*
         * When a report is delivered, extract the relevant credit card info,
         * and store it in m_currentApplicant
         */
        CreditCardDataDocument cardInfo = CreditCardDataDocument.Factory.parse(cardData);
```

## WebLogic Workshop Tutorials

```

CustomerType[] customer = cardInfo.getCreditCardData().getCustomerArray();
CardType[] cards = customer[0].getCardArray();
for(int i = 0; i < cards.length; i++)
{
    m_currentApplicant.availableCCCredit += cards[i].getAvailableCredit();
}

/*
 * Send the complete report back to the client.
 */
// callback.onCreditReportDone(m_currentApplicant);

}
catch(XmlException xe)
{
    /*
     * If there is a problem with extracting the credit card info,
     * store an error message in m_currentApplicant.
     */
    m_currentApplicant.message = "There was an error retrieving the credit card

    /*
     * Send an error message back to the client.
     */
    callback.onCreditReportDone(m_currentApplicant);
}

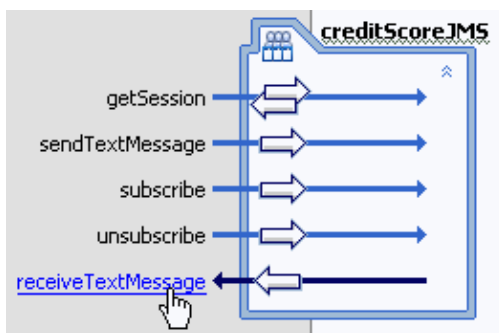
/*
 * Construct and send an XML message through the JMS control
 * to the credit scoring application.
 */
String messageToJMSControl = makeMessageToJMS(m_currentApplicant.availableCCCredit);
creditScoreJMS.subscribe();
creditScoreJMS.sendTextMessage(messageToJMSControl);
}

```

To Edit the Callback Handler to Parse the XML Response

In this task you will use the XBeans classes to parse the XML response from the JMS control and extract the credit score.

1. Click the **Design View** tab.
2. Click the link text for the **receiveTextMessage** callback handler.



3. Edit the **creditScoreJMS\_receiveTextMessage** callback handler code so that it appears as follows:

```

public void creditScoreJMS_receiveTextMessage(java.lang.String payload)
{
    try

```

## WebLogic Workshop Tutorials

```
{
    /*
     * Extract the the credit score from the JMS response string
     * and store it in m_currentApplicant.
     */
    ScoreResponseDocument scoreInfo = ScoreResponseDocument.Factory.parse(payload);
    m_currentApplicant.creditScore = scoreInfo.getScoreResponse().getCalculatedScore();

    /*
     * Send the complete report back to the client.
     */
    callback.onCreditReportDone(m_currentApplicant);
}
catch(XmlException e)
{
    m_currentApplicant.message = "There was a problem retrieving the credit score";

    /*
     * Send the the error message back to the client.
     */
    callback.onCreditReportDone(m_currentApplicant);
}
}
```

4. Click **Ctrl+S** to save your work.

### To Test the Investigate Java Control

1. From the **Window** menu, select **InvestigateTest.jws**.
2. Click the Start button, shown here:



Workshop builds InvestigateTest.jws and launches the Workshop Test Browser.

3. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 555555555 and click the **requestCreditReport** button.

**Note:** Use one of the following (9 digit) taxID's to test your Java control throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click **Refresh** until callback.onCreditReportDone appears in the **Message Log**. Click **callback.onCreditReportDone**.

Note that a new piece of information appears in the applicant profile:

<creditScore>810</creditScore>. This new piece of information was returned by the legacy credit scoring application exposed over JMS.



## Client Callback

Submitted at Thursday, August 21, 2003 3:48:31 PM PDT

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1061506104823</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditReportDone xmlns:ns="http://www.openuri.org/">
      <ns:applicant>
        <ns:taxID>555555555</ns:taxID>
        <ns:firstName>Marnie</ns:firstName>
        <ns:lastName>Smithers</ns:lastName>
        <ns:currentlyBankrupt>false</ns:currentlyBankrupt>
        <ns:availableCCCredit>10000</ns:availableCCCredit>
        <ns:creditScore>810</ns:creditScore>
      </ns:applicant>
    </ns:onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5. Return to WebLogic Workshop, and click the **Stop** button  to close the **Workshop Test Browser**.

Related Topics

JMS Control

Getting Started with XMLBeans

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



## Step 5: Add an EJB Control

In this task you will add an Enterprise Java Bean control to the Investigate Java control. This control provides accesses to an EJB (already deployed on the server as part of the Workshop samples domain) that calculates an approval rating on credit applicants. The tasks in this step are:

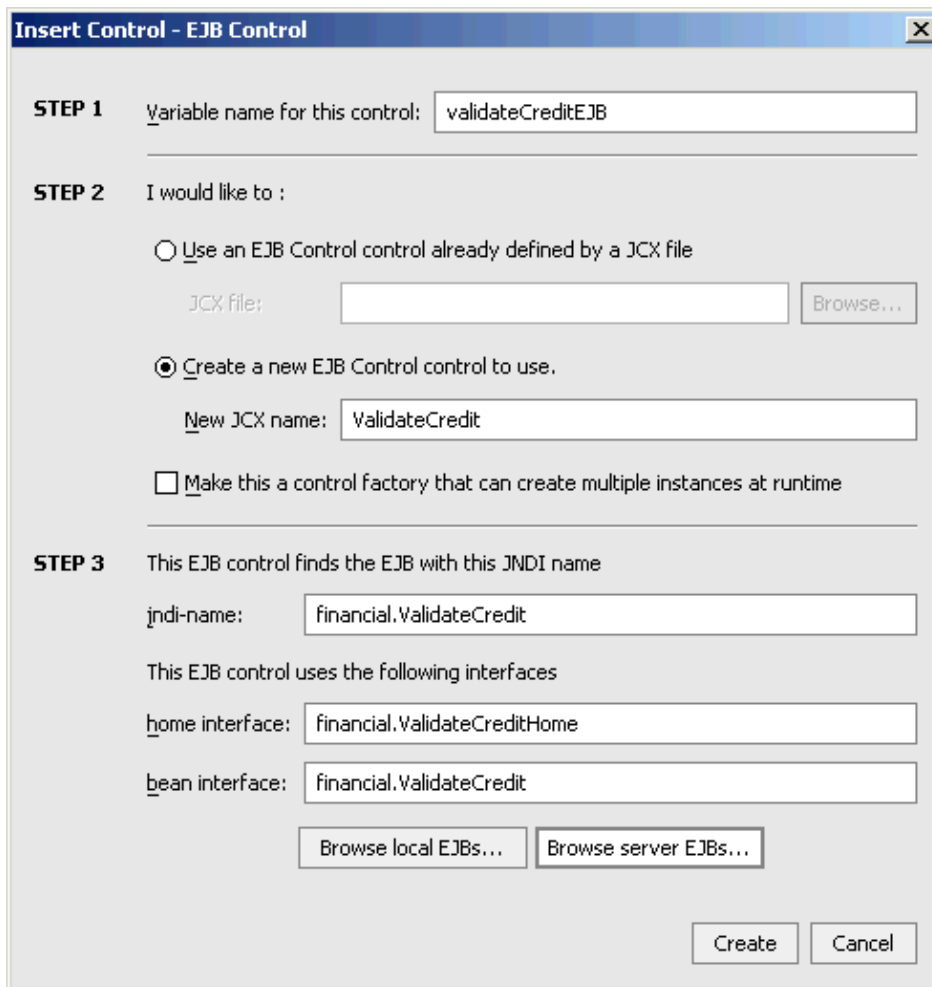
- To add an EJB control
- To add code that requests credit validation
- To test the Investigate Java control

### To Add an EJB Control

The ValidateCredit EJB you will be accessing is designed to take the credit score and return a response about the applicant's credit worthiness. To access the ValidateCredit bean, you will add an EJB control.

**Note:** In order for you to add an EJB control to a web service project, the compiled home and remote interfaces for the EJB must be in the project also. The JAR file containing the home and remote interfaces is already within the *Libraries* folder of the application.

1. On the *Application* tab, double-click *InvestigateImpl.jcs* and click the *Design View* tab.
2. From the *Insert* menu, choose *Controls* → *EJB Control*. The *Insert Control – EJB Control* dialog appears.
3. Enter values as shown in the following illustration. Use the *Browse server EJBs* button to fill in the *jndi-name* values required in Step 3 of the dialog. When you select *financial.ValidateCredit* from the list provided and click *Select*, the home interface and bean interface values will be added automatically.



**Insert Control - EJB Control**

**STEP 1** Variable name for this control:

**STEP 2** I would like to :

☐ Use an EJB Control control already defined by a JCX file

JCX file:

☒ Create a new EJB Control control to use.

New JCX name:

☐ Make this a control factory that can create multiple instances at runtime

**STEP 3** This EJB control finds the EJB with this JNDI name

jndi-name:

This EJB control uses the following interfaces

home interface:

bean interface:

4. Click **Create**.

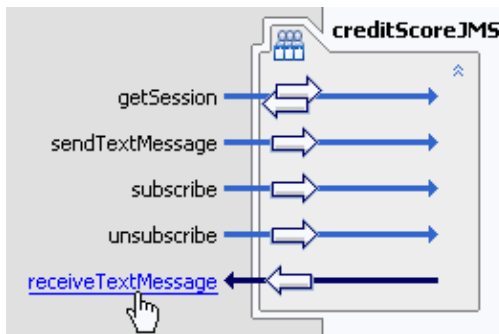
WebLogic Workshop adds an EJB control to your design, as shown below. The control shows that the EJB exposes two methods: create and validate. The validate method is the one you will use (all EJBs expose a create method that is used by WebLogic Server; you will not need to call this).



Now you need to connect the score received through the creditScore JMS control to the validateCredit EJB control. You will do this by updating the JMS control's callback handler. The code you add will send the credit score to the EJB for validation.

To Add Code that Requests Credit Validation

1. Confirm that **InvestigateImpl.jcs** is displayed in the main work area.
2. Click the **Design View** tab.
3. Click the link text of the **receiveTextMessage** callback handler. Its source code appears in Source View.



4. Edit the **creditScoreJMS\_receiveTextMessage** callback handler so that it appears as follows. Code to edit appears in red. Make sure to delete the code shown crossed out.

```

public void creditScoreJMS_receiveTextMessage(java.lang.String payload)
{
    try
    {
        /*
         * Extract the the credit score from the JMS response string
         * and store it in m_currentApplicant.
         */
        ScoreResponseDocument scoreInfo = ScoreResponseDocument.Factory.parse(payload);
        m_currentApplicant.creditScore = scoreInfo.getScoreResponse().getCalculatedScore();

        /*
         * Send the complete report back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
        */
    }
    catch(XmlException e)
    {
        m_currentApplicant.message = "There was a problem retrieving the credit score";

        /*
         * Send the the error message back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }

    try
    {
        /*
         * Pass the credit score to the EJB's validate method. Store the value returned
         * with other applicant data.
         */
        m_currentApplicant.approvalLevel = validateCreditEJB.validate(m_currentApplicant);

        /*
         * Send the complete report back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }
    catch(java.rmi.RemoteException e)
    {
        m_currentApplicant.message = "There was a problem retrieving the approval level";

        /*
         * Send an error message back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }
}

```

- }  
5. Press **Ctrl+S** to save your work.

To Test the Investigate Java control

1. On the **Application** tab, double-click **InvestigateTest.jws** to display the file.
2. Click the Start button, shown here:



Workshop builds InvestigateTest.jws and launches the Workshop Test Browser.

3. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 555555555 and click the **requestCreditReport** button.

**Note:** Use one of the following (9 digit) taxID's to test your Java control throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click **Refresh** until **callback.onCreditReportDone** appears in the **Message Log**.
5. When **callback.onCreditReportDone** appears, click it. Your screen should resemble the following:

```

Client Callback
Submitted at Thursday, August 21, 2003 4:09:37 PM PDT
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1061507376401</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditReportDone xmlns:ns="http://www.openuri.org/">
      <ns:applicant>
        <ns:taxID>555555555</ns:taxID>
        <ns:firstName>Marnie</ns:firstName>
        <ns:lastName>Smithers</ns:lastName>
        <ns:currentlyBankrupt>false</ns:currentlyBankrupt>
        <ns:availableCCCredit>10000</ns:availableCCCredit>
        <ns:creditScore>810</ns:creditScore>
        <ns:approvalLevel>Applicant is a low risk.</ns:approvalLevel>
      </ns:applicant>
    </ns:onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Notice the new element in the applicant profile `<approvalLevel>Applicant is a low risk.</approvalLevel>`. This element was provided by the EJB credit validating application.

Related Topics

EJB Control

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



## Step 6: Add Support for Cancellation and Exception Handling

Working through the first steps of this tutorial has given you a sense of how a Java control works. In ideal conditions, it does what it is supposed to do. But as it now stands, it isn't quite ready for prime time. Here are a few possible problems:

- The client may want to cancel the credit report before Investigate has sent back a response. Providing this ability is especially important in asynchronous exchanges, which may be long-lived.
- Investigate's dependency on the credit card web service (which is accessed asynchronously) may make the overall response time to the client quite long.
- You have no way of knowing how long it will take the credit card service to respond to your requests. To better manage this, you can use a timer to set a limit on how the resource may take to respond.
- Operation methods (such as requestCreditReport) may throw an uncaught exception.

Below you will enhance the Investigate Java control to better handle these possible problems.

This tasks in this step are:

- To add a method so that clients can cancel operations
- To add a Timer Control to limit the time allowed for response
- To handle exceptions thrown from operation methods
- To test the Investigate Java Control

To Add a Method so that Clients Can Cancel Operations

1. On the **Application** tab, double-click **InvestigateImpl.jcs**.
2. Click the **Design View** tab.
3. From the **Insert** menu, select **Method**.
4. In the field that appears, type `cancelInvestigation` and press Enter.
5. Click the link text ***cancelInvestigation*** to view its code in Source View.
6. Edit the `cancelInvestigation` method code so it appears as follows. Code to add is shown in red.

```
/**
 * @common:operation
 */
public void cancelInvestigation()
{
    /*
     * Cancel the request to the credit card company because it is now unnecessary.
     */
    creditCardReportControl.cancelRequest();
    /*
     * Use the callback to send a message to the client. Note that this also ends
     * the conversation because the callback's conversation phase property is set to
     */
    m_currentApplicant.message = "Investigation canceled.";
    callback.onCreditReportDone(m_currentApplicant);
}
```

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

To Add a Timer Control to Limit the Time Allowed for Response

It can be difficult to predict how long an asynchronous resource will take to respond to a request. For example, Investigate's call to the credit card report web service may take hours. Here, you will add a way to limit the amount of time the credit card's web service has to respond. Using a Timer control, you can specify an amount of time after which the operation should be canceled.

1. Confirm that *InvestigateImpl.jcs* is displayed in the main work area.
2. Click the **Design View** tab.
3. From the **Insert** menu, select **Controls-->Timer**. The **Insert Control** dialog appears.
4. Enter values as shown in the following illustration:

**Insert Control - Insert Timer**

**STEP 1** Variable name for this control:

---

**STEP 2** Amount of time before the timer fires its onTimeout method the first time

timeout:

Examples: 5 seconds  
5 s  
5 hours 3 minutes  
3 days 2 hours 41 minutes

(OPTIONAL)  
Interval at which the timer will fire its onTimeout method thereafter

repeats-every:

Examples: 5 seconds  
5 s  
5 hours 3 minutes  
3 days 2 hours 41 minutes

☐ Make this a control factory that can create multiple instances at runtime

These values specify that the Timer control will send its onTimeout callback five minutes after the timer starts.

5. Click **Create**.
6. Click the link text for the *requestCreditReport* method to view its code in Source View.
7. Add the code shown in red:

```
public void requestCreditReport(String taxID)
{
    m_currentApplicant.taxID = taxID;

    /**
     * Retrieve data from the database and store it in the rec object.
     */
    Record rec = bankruptciesDB.checkForBankruptcies(taxID);
    /**
     * If the database returns substantial data, then store that data
     * in the m_currentApplicant object.
     */
    if(rec != null)
    {
        m_currentApplicant.firstName = rec.firstname;
        m_currentApplicant.lastName = rec.lastname;
    }
}
```

## WebLogic Workshop Tutorials

```

m_currentApplicant.currentlyBankrupt = rec.currentlyBankrupt;

/**
 * Invoke the Credit Card Report web service.
 * Results from the web service will be provided via a callback.
 */
creditCardReportControl.getCreditCardData(taxID);

/*
 * Start the timer. If the credit card report is not
 * received within 5 minutes, the conversation will be finished
 * and the client will be notified that
 * there was a problem.
 */
creditCardReportTimer.start();
}
/**
 * If the database does not return substantial data, notify the client
 * that there is a problem.
 */
else
{
    m_currentApplicant.message = "No data could be found on the applicant. Please
    /*
     * Send the error message to the client via a callback.
     */
    callback.onCreditReportDone(m_currentApplicant);
}
}

```

This starts the timer as soon as the credit card service is called. If the credit card service does not respond by the time the timeout duration is reached, then the timer's onTimeout callback handler is invoked.

8. Click the **Design View** tab to view to Design View for InvestigateImpl.jcs.
9. Click the link text **onCreditCardDataReady** (the callback handler for the web service control) to view its code in Source View.
10. Edit the code for **creditCardReportControl\_onCreditCardDataReady** so it appears as follows. Add the code appearing in red.

```

public void creditCardReportControl_onCreditCardDataReady(java.lang.String cardData)
{
    /*
     * Stop the timer that was started when the Credit Card Report web service
     * was called, because the callback from the web service has been received.
     */
    creditCardReportTimer.stop();

    try
    {
        /*
         * When a report is delivered, extract the relevant credit card info,
         * and store it in m_currentApplicant
         */
        CreditCardDataDocument cardInfo = CreditCardDataDocument.Factory.parse(cardData);
        CustomerType[] customer = cardInfo.getCreditCardData().getCustomerArray();
        CardType[] cards = customer[0].getCardArray();
        for(int i = 0; i < cards.length; i++)
        {
            m_currentApplicant.availableCCCredit += cards[i].getAvailableCredit();
        }
    }
}

```



```

    }
    catch(XmlException xe)
    {
        /*
         * If there is a problem with extracting the credit card info,
         * store an error message in m_currentApplicant.
         */
        m_currentApplicant.message = "There was an error retrieving the credit card

        /*
         * Send the the error message back to the client.
         */
        callback.onCreditReportDone(m_currentApplicant);
    }

    /*
     * Construct and send an XML message through the JMS control
     * to the credit scoring application.
     */
    String messageToJMSControl = makeMessageToJMS(m_currentApplicant.availableCCCreditScoreJMS.subscribe();
    creditScoreJMS.sendTextMessage(messageToJMSControl);
}

```

11. Click the **Design View** tab to view InvestigateImpl.jcs in Design View.
12. On the Timer control, click the link text for the **onTimeout** callback handler to view its source code.
13. Edit the **onTimeout** callback handler source code so that it appears as follows. Code to add appears in red.

```

public void creditCardReportTimer_onTimeout(long time)
{
    /**
     * Because the Credit Card Report web service has not calledback in the time all
     * cancel the request for a report from the web service
     */
    creditCardReportControl.cancelRequest();

    /**
     * Cancel the current investigation.
     */
    cancelInvestigation();
}

```

14. Press **Ctrl+S** to save your work.

### To Handle Exceptions Thrown from Operation Methods

Unhandled exceptions thrown from operations (such as methods exposed to clients) can leave the client hanging and the Java control may simply continue operating on the system, unnecessarily using resources.

In this step, you will add code to implement a handler for the `ControlContext.onException` callback. The callback handler receives the exception object thrown from the operation, the name of the method that threw the exception, and the parameters passed to the method.

WebLogic Workshop provides a way for you to preserve information about exceptions by logging it in a text file. You will add exception handling code to log the exception, then send an appropriate response to the client.

1. If InvestigateImpl.jcs is not displayed in Source View, click the **Source View** tab.

## WebLogic Workshop Tutorials

2. Add the following import statement to the list of import statements at the top of the InvestigateImpl.jcs directly underneath the package declaration.

```
import com.bea.wlw.util.Logger;
```

3. Add the following annotation and declaration to the body of InvestigateImpl.jcs.

```
/**
 * @common:context
 */
ControlContext context;
```

4. In Source View, add the following onException callback handler to the body of InvestigateImpl.jcs.

```
public void context_onException(Exception e, String methodName, Object[] arguments)
{
    /*
     * Create a logger variable to use for logging messages. Assigning it the
     * "Investigate" category name will make it easier to find its messages
     * in the log file.
     */
    Logger logger = context.getLogger("*** Investigate ***");

    /*
     * Log an error message, giving the name of the method that threw the
     * exception and a stack trace from the exception object.
     * The message will be written to the jws.log file located
     * in the server root folder, in this case:
     * BEA_HOME/weblogic81/samples/domains/workshop
     */
    logger.error("Exception in " + methodName + ": " + e);

    /*
     * Send a callback to the client with notification that an error has occurred.
     */
    m_currentApplicant.message = "Unable to respond to request at this time. Please  
callback.onCreditReportDone(m_currentApplicant);
}
```

5. Press **Ctrl+S** to save your work.

By default the log file is located in the server's domain root folder, in this case the Workshop domain root folder:

BEA\_HOME/weblogic81/samples/domains/workshop/workshop.log

An entry in the log will resemble the following:

```
21 Aug 2003 16:38:26,610 ERROR *** Investigate ***: Exception in requestCreditReport: com.bea.c
Code:java.io.FileNotFoundException
String:Response: '401: Unauthorized xxx' for url: 'https://weblogic:wrong_password@localhost:70
Detail:
END SERVICE FAULT [ServiceException]
```

To test the Investigate Java Control

Try the following test scenarios through InvestigateTest.jws.

## WebLogic Workshop Tutorials

- Enter a random string when you invoke `requestCreditReport`, such as "qwe123qwe". (**Note:** this will not raise an error in `workshop.log`, instead a message will be sent to the client saying that no data could be found.)
- Test the timer control by changing its timeout value to 1 second. Unless your computer is very fast, 1 second should not be enough time for the Credit Card Report web service to send a callback before the timeout. After changing the timeout value, run `InvestigateImplTest.jws`, wait one second, then click the **Refresh** link. The message log will resemble the following.



- Test the exception handler by throwing an exception from somewhere in your code. Then check the `workshop.log` file to see the message that results.

### Related Topics

#### Timer Control

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



## Step 7: Transport Security

There is no need to secure the Investigate Java control against Internet traffic, because it can only be called directly by other local components. But it is necessary to secure Investigate's communications with other applications across the Internet. In this step you will secure one of those communication routes: the communication route between Investigate and the Credit Card Report web service.

The tasks in this step are:

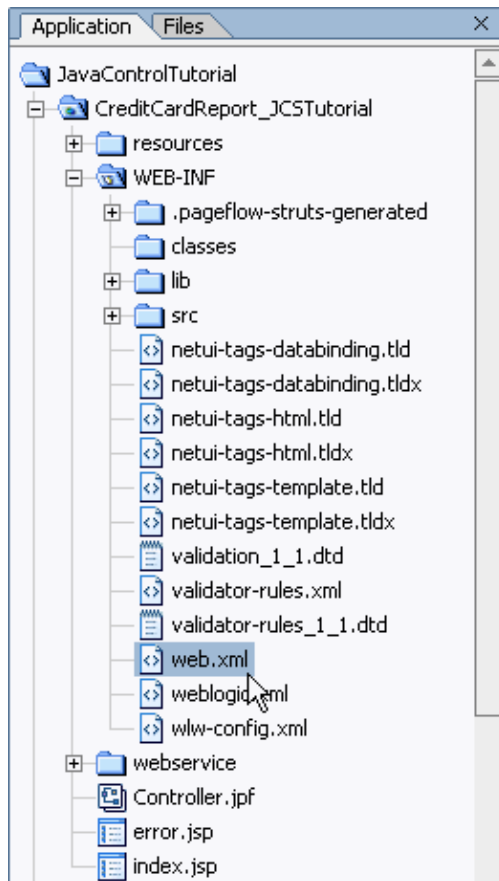
- To configure CreditCardReport.jws for SSL
- To configure CreditCardReport.jws for basic authentication
- To edit Investigate to use the right password to access CreditCardReport.jws
- To Edit the Credit Card Report Control File
- To Test the Investigate Java Control

To Configure CreditCardReport.jws for SSL

You configure a web resource for SSL through the web.xml and weblogic.xml files, located in your application's WEB-INF directory.

By placing the following <security-constraint> element in web.xml, all communication with the resource is encrypted, ensuring the confidentiality of the communication.

1. On the *Application* tab, navigate to the folder *JavaControlTutorial/CreditCardReport\_JCSTutorial/WEB-INF* and open the *web.xml* file.



2. Add the following `<security-constraint>` XML element to `web.xml`. Make sure that you add the `<security-constraint>` element after the final `</taglib>` end tag, but before the final `</web-app>` end tag. Code to add is shown in red.

```

<taglib>
  <taglib-uri>netui-tags-template.tld</taglib-uri>
  <taglib-location>/WEB-INF/netui-tags-template.tld</taglib-location>
</taglib>

<security-constraint>
  <display-name>
    Security Constraint for the Credit Card Report web service
  </display-name>
  <web-resource-collection>
    <web-resource-name>CreditCardReport.jws</web-resource-name>
    <description>A web service secured by SSL and basic authentication</description>
    <url-pattern>/webservice/CreditCardReport.jws</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>

</web-app>

```

To configure `CreditCardReport.jws` for basic authentication

Basic authentication requires that clients provide a valid username and password to access a web resource. (The registration processes by which users originally acquire a username and password is an important consideration that is beyond the scope of this tutorial.) Also, the `<auth-constraint>` element you add below requires that users be members of the `RegisteredCreditCardUsers` role to access the Credit Card Report web service. Whereas SSL ensures that communications are confidential, basic authentication ensures that you know the identity of your clients. In this case, basic authentication ensures that Credit Card Report knows its client, the Investigate Java control, really *is* the Investigate Java control, and not another potentially malicious third party.

1. Confirm that ***web.xml*** is displayed in the main work area.
2. Edit the `<security-constraint>` element so it appears as follows. Also add a `<security-role>` element immediately after the `<security-constraint>` element. Code to add appears in red.

```
<taglib>
  <taglib-uri>netui-tags-template.tld</taglib-uri>
  <taglib-location>/WEB-INF/netui-tags-template.tld</taglib-location>
</taglib>

<security-constraint>
  <display-name>
    Security Constraint for the Credit Card Report web service
  </display-name>
  <web-resource-collection>
    <web-resource-name>CreditCardReport.jws</web-resource-name>
    <description>A web service secured by SSL and basic authentication</description>
    <url-pattern>/web-service/CreditCardReport.jws</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>RegisteredCreditCardReportUsers</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>

<security-role>
  <description>Users who have successfully completed
    Credit Card Report's registration process and have been given a username
    and password</description>
  <role-name>RegisteredCreditCardReportUsers</role-name>
</security-role>

</web-app>
```

3. Press **Ctrl+S** to save your work.

Next you grant one user the role of `RegisteredCreditCardReportUser`. The user, "weblogic", is a preconfigured administrative user in WebLogic Server. The password for this user is "weblogic".

4. On the **Application** tab double-click ***JavaControlTutorial/CreditCardReport\_JCSTutorial/WEB-INF/weblogic.xml*** to open the file. Add the following `<security-role-assignment>` element immediately after the opening `<weblogic-web-app>` tag. Code to add appears in red.

## WebLogic Workshop Tutorials

```
<weblogic-web-app>

  <security-role-assignment>
    <role-name>RegisteredCreditCardReportUsers</role-name>
    <principal-name>weblogic</principal-name>
  </security-role-assignment>

  <jsp-descriptor>
    <!-- Comment the jspServlet param out to go back to weblogic's jspc -->
    <jsp-param>
      <param-name>jspServlet</param-name>
      <param-value>weblogic.servlet.WlwJSPServlet</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </jsp-param>
  </jsp-descriptor>

  <url-match-map>
    weblogic.servlet.utils.SimpleApacheURLMatchMap
  </url-match-map>

</weblogic-web-app>
```

5. Press **Ctrl+S** to save your work.

To Edit Investigate to Use the Correct Username and Password to Access CreditCardReport.jws

Now the Credit Card Report web service requires that its clients provide a valid username and password and that those clients be assigned the role of RegisteredCreditCardReportUsers. In this task you will edit Investigate to provide the correct username and password when accessing Credit Card Report.

1. On the **Application** tab, double-click **InvestigateImpl.jcs**.
2. Edit the **requestCreditReport** method so it appears as follows. Code to add is shown in red.

```
public void requestCreditReport(String taxID)
{
    m_currentApplicant.taxID = taxID;

    /*
     * Retrieve data from the database and store it in the rec object.
     */
    Record rec = bankruptciesDB.checkForBankruptcies(taxID);
    /*
     * If the database returns substantial data, then store that data
     * in the m_currentApplicant object.
     */
    if(rec != null)
    {
        m_currentApplicant.firstName = rec.firstname;
        m_currentApplicant.lastName = rec.lastname;
        m_currentApplicant.currentlyBankrupt = rec.currentlyBankrupt;

        /*
         * Set the username and password necessary to access the Credit
         * Card Report web service.
         * Then invoke the getCreditCardData method.
         * Results from the web service will be provided via a callback.
         */
    }
}
```

## WebLogic Workshop Tutorials

```
    */
    creditCardReportControl.setUsername("weblogic");
    creditCardReportControl.setPassword("weblogic");
    /*
     * Invoke the Credit Card Report web service.
     * Results from the web service will be provided via a callback.
     */
    creditCardReportControl.getCreditCardData(taxID);

    /*
     * Start the timer. If the credit card report is not
     * received within 5 minutes, the conversation will be finished
     * and the client will be notified that
     * there was a problem.
     */
    creditCardReportTimer.start();
}
/*
 * If the database does not return substantial data, notify the client
 * that there is a problem.
 */
else
{
    m_currentApplicant.message = "No data could be found on the applicant. Please
    /*
     * Send the error message to the client via a callback.
     */
    callback.onCreditReportDone(m_currentApplicant);
}
}
```

3. Press **Ctrl+S** to save your work.

### To Edit the Credit Card Report Control File

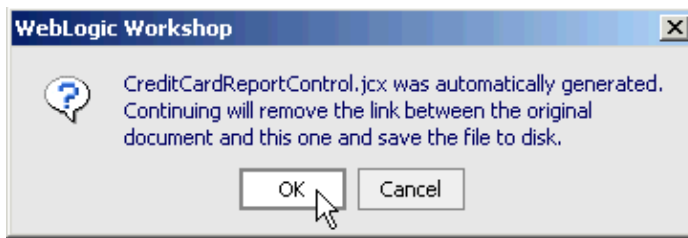
If you were to test your Java control now, it would not succeed in invoking the Credit Card Report web service. This is because the Credit Card Report currently listens for clients on the HTTPS port 7002, but the Investigate Java control sends its requests through the HTTP port 7001. In this task you will configure the Credit Card Report control (used by the Investigate Java control) to send communications through port 7002.

1. On the **Application** tab double-click **CreditCardReportControl.jcx** to display the file.
2. Click the **Source View** tab to view the source code for CreditCardReportControl.jcx.
3. Edit the **@jc:location** annotation so it appears as follows. Code to edit appears in red.

```
/**
 * @jc:location http-url="https://localhost:7002/CreditCardReport_JCSTutorial/webservice
 * @jc:wsdl file="#CreditCardReportWsd1"
 */
public interface CreditCardReportControl extends com.bea.control.ControlExtension, com.b
```

When you try to modify this control file, Workshop will warn you that you are trying to edit an autogenerated file. Click **Yes** when Workshop asks you if want to edit this file.





4. Press **Ctrl+S** to save your work and press **Ctrl+F4** to close the file.

#### To Test the Investigate Java Control

In this task you will perform two tests. First you will test to see if the right password results in successful communication, then you will test to see if the wrong password results in failed communication.

#### Testing for Successful Communication

1. On the **Application** tab, double-click **InvestigateTest.jws**.
2. Click the Start button, shown here:



Workshop builds InvestigateTest.jws and launches the Test Browser.

3. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 555555555 and click the **requestCreditReport** button.

**Note:** Use one of the following (9 digit) taxID's to test your Java control throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click **Refresh** until **callback.onCreditReportDone** appears in the **Message Log**.
5. When **callback.onCreditReportDone** appears, click it. Your screen should resemble the following:

#### Client Callback

Submitted at Thursday, August 21, 2003 4:09:37 PM PDT

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1061507376401</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditReportDone xmlns:ns="http://www.openuri.org/">
      <ns:applicant>
        <ns:taxID>555555555</ns:taxID>
        <ns:firstName>Marnie</ns:firstName>
        <ns:lastName>Smithers</ns:lastName>
        <ns:currentlyBankrupt>false</ns:currentlyBankrupt>
        <ns:availableCCCredit>10000</ns:availableCCCredit>
        <ns:creditScore>810</ns:creditScore>
        <ns:approvalLevel>Applicant is a low risk.</ns:approvalLevel>
      </ns:applicant>
    </ns:onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

#### Testing for Failed Communication

1. From the **Window** menu, select **InvestigateImpl.jcs** and click the **Source View** tab.

## WebLogic Workshop Tutorials

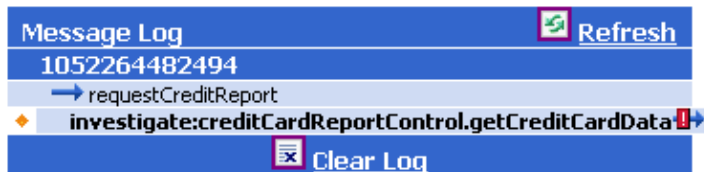
2. Edit the relevant portion of the **requestCreditReport** method to look like the following. Note that you are intentionally providing the wrong password when calling Credit Card Report.

```
/**
 * Set the username and password necessary to access the Credit
 * Card Report web service.
 * Then invoke the getCreditCardData method.
 * Results from the web service will be provided via a callback.
 */
creditCardReportControl.setUsername("weblogic");
creditCardReportControl.setPassword("wrong_password");
/**
 * Invoke the Credit Card Report web service.
 * Results from the web service will be provided via a callback.
 */
creditCardReportControl.getCreditCardData(taxID);
```

3. From the **Window** menu, select **InvestigateTest.jws** to open the file.
4. Click the Start button, shown here:



5. Workshop builds InvestigateTest.jws and launches the Test Browser.
6. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 555555555 and click the **requestCreditReport** button.
7. Click **Refresh** until **investigate:creditCardReportControl.getCreditCardData** appears in the **Message Log**. Your screen should resemble the following:



The red exclamation mark next to the method invocation arrow indicates that an error has occurred. In this case, the `getCreditCardData` method throws an error because Investigate provided the wrong password to the Credit Card Report web service.

8. Note the error message in the **External Service Response** section on the **Test Browser**

### External Service Response

Submitted at Friday, Aug 21, 2003 16:38:26 PM PST

External Service Failure: Response: '401: Unauthorized xxx' for url:

'https://weblogic:wrong\_password@localhost:7002/CreditCardReport\_JCSTutorial/webservice/CreditCardRep

If you check the file `BEA_HOME/weblogic81/samples/domains/workshop/workshop.log` you will see the following entry.

```
21 Aug 2003 16:38:26,610 ERROR *** Investigate ***: Exception in requestCreditReport: com
Code: java.io.FileNotFoundException
String: Response: '401: Unauthorized xxx' for url: 'https://weblogic:wrong_password@locall
Detail:
END SERVICE FAULT [ServiceException]
```

## WebLogic Workshop Tutorials

Note that the error is interpreted as a "FileNotFoundException" because the the Investigate Java control thinks of the URL

"https://weblogic:wrong\_password@localhost:7002/CreditCardReport\_JCSTutorial/webservice/CreditCardReport" as a file resource. When the Investigate Java control cannot access the file, it throws a file not found exception, although the real cause of the problem is the wrong password.

9. Don't forget to open *InvestigateImpl.jcs* and replace "wrong\_password" with the correct password: "weblogic".

### Related Topics

#### Transport Security

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



## Step 8: Compile the Java Control and Add a Property

In the previous steps you built a control that encapsulates some useful business logic. In this step, you'll compile the Investigate Java control into a JAR file for greater portability.

For many needs, it's perfectly fine to use the control's source files directly within the same project as the control's container or client, as is the case with Investigate's web service container, *InvestigateTest.jws*. But what if you're designing a control that might be useful from many containers? It could get tedious to copy a set of source files from project to project. To solve this problem we will package the control in a JAR file that's much easier to copy into different containers.

You begin the process of packaging by moving the control into its own control project. A control project includes files and annotations that support the control as a separately distributable package. You can even develop the sources for multiple controls within a single project, and package them all in a single JAR file for distribution.

Before you compile the Investigate Java control, you will add one more layer of functionality. You will add a property to the Investigate control, which will allow developers using the control to change the way it functions.

You'll start by creating a project in which to further develop your Java control's source files.

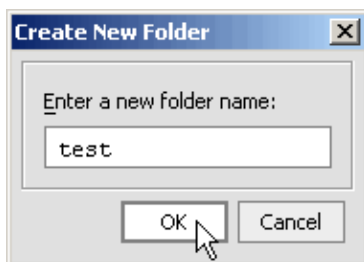
The tasks in the step are:

- To move Investigate to a Control Project
- To set Investigate's properties
- To create a new property for Investigate
- To edit Investigate to use the new property
- To compile the Investigate control
- To test the Investigate Java control

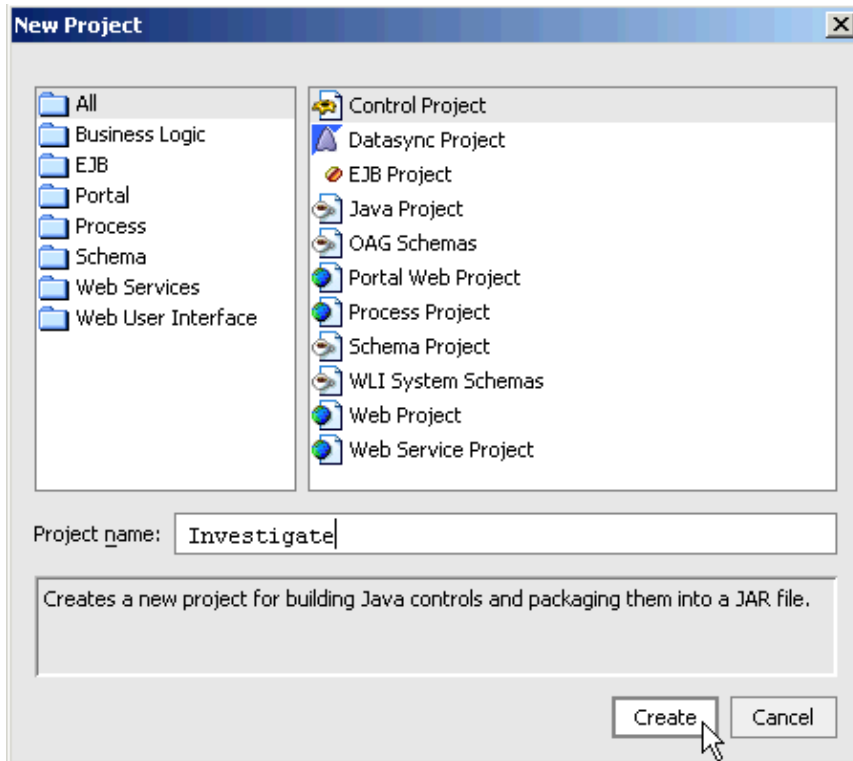
To Move Investigate to a Control Project

In this task you will move two items. First you will move the test web service *InvestigateTest.jws* into its own folder. Second you will move the Investigate Java control into its own Control project. It is necessary to separate the Investigate Java control from its test web service because web services aren't enabled in Control projects.

1. On the **Application** tab, right click the *JavaControlTutorialWeb* folder and select **New-->Folder**.
2. In the **Create New Folder** dialog, enter test.

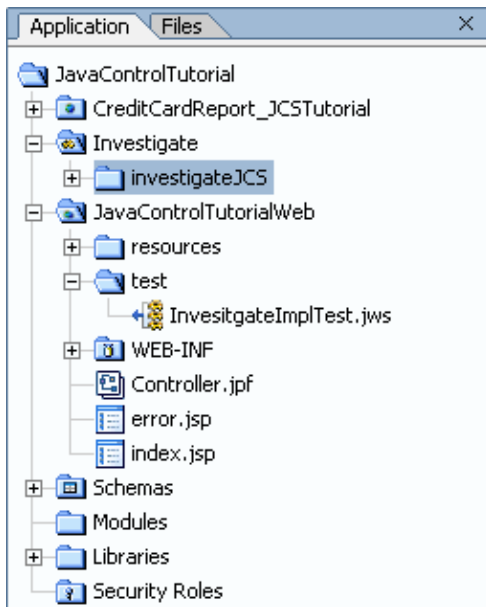


3. Click **Ok**.
4. On the **Application** tab, hold the mouse button down over the test web service **InvestigateTest.jws** and drag it into the folder **JavaControlTutorialWeb/test**.
5. On the **Application** tab, right-click the **JavaControlTutorial** folder and choose **New-->Project**.
6. In the **New Project** dialog, in the upper left-hand pane, confirm that **All** is selected.  
In the upper right-hand pane, select **Control Project**.  
In the **Project Name** field, enter **Investigate**.



7. Click **Create**.  
WebLogic Workshop adds a new control project called **Investigate**.
8. On the **Application** tab, hold down the mouse button over folder **investigateJCS**, drag it into the **Investigate** project folder.

The Application tab should look like the following.

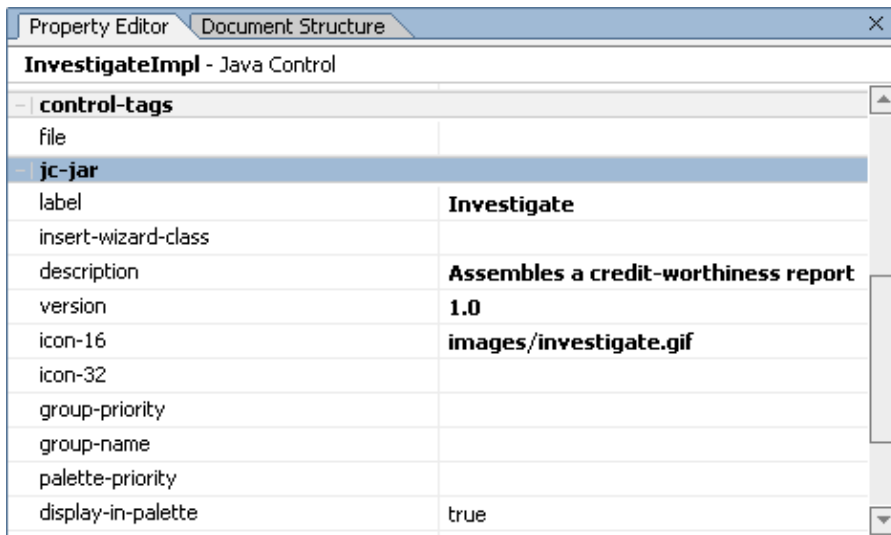


To Set Investigate's Properties

1. Open the *investigateJCS* folder you just moved, then double-click *InvestigateImpl.jcs*. Click the **Design View** tab.
2. Make sure that the control's design is selected by clicking its title bar in Design View.



3. On the **Property Editor** tab, in the section labeled *jc-jar*, edit the property attributes as shown in the following figure. (You may want to re-size the Property Editor tab to make it easier to work with.)



These properties provide information that WebLogic Workshop will present to a developer using the Investigate Java control. The *label* corresponds to the control's name on menus and palettes; the *icon-16* attribute value is a relative path to the image associated with the control in the IDE. Finally, if you look at the InvestigateImpl.jcs source code, you'll see that the values for these attributes have been written into the source as annotation attributes.

```
/**
 * @editor-info:code-gen control-interface="true"
 * @jcs:jc-jar icon-16="images/investigate.gif" version="1.0" description="Assembles a credit-wor
 */
public class InvestigateImpl implements Investigate, ControlSource
```

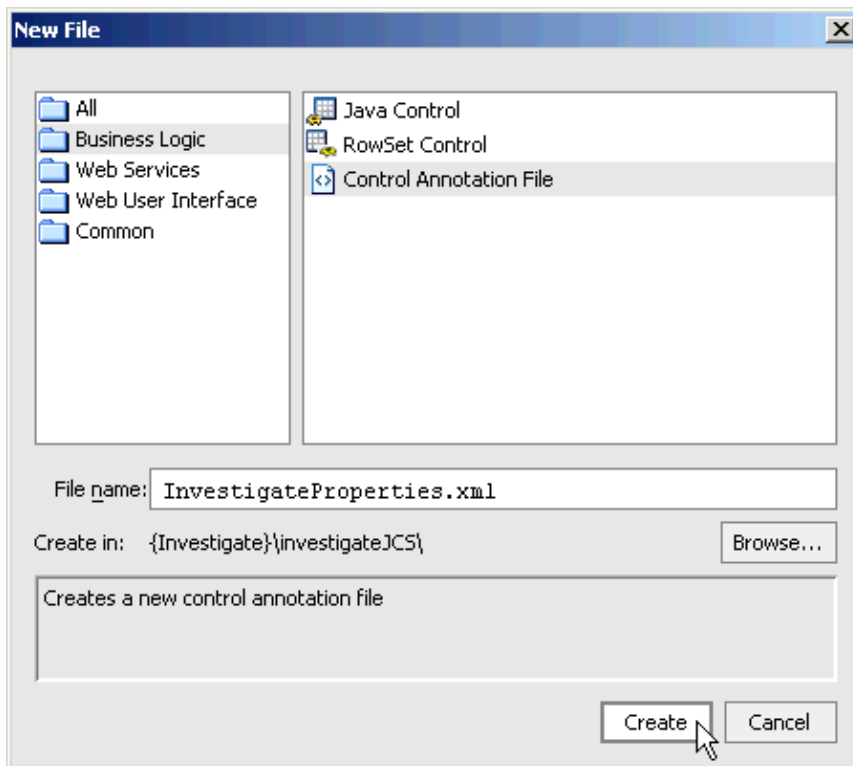
4. Press **Ctrl+S** to save your work.
5. Right-click the *investigateJCS* folder and select **New—>Folder**.
6. In the **Create New Folder** dialog, type *images* and click **OK**.
7. Holding down the mouse button over the image below, drag it to WebLogic Workshop's Application tab, and drop it in the *images* folder you just created.



Before packaging and testing the control project, you'll make one more enhancement to the control. With a little more code, you'll make it possible for an application developer using your control to set the phone number users should call if they run into an error. You'll start by defining a property for the Investigate Java control. To do this, you'll create an InvestigateProperties.xml file to hold a description of the control's new property.

#### To Create a New Property for Investigate

1. In the **Application** tab, right-click the *investigateJCS* folder and select **New—>Other File Types**.
2. In the **New File** dialog, in the left-hand pane, click **Business Logic**.  
In the right-hand pane, click **Control Annotation File**.  
In the File name box, type InvestigateProperties.xml.



3. Click **Create**.

WebLogic Workshop creates the new XML file and displays it in the main work area. It contains a simple XML declaration and sample tags.

4. Edit InvestigateProperties.xml so it appears as follows. Overwrite any XML elements already appearing in the file.

```
<?xml version="1.0" ?>
<control-tags xmlns="http://www.bea.com/2003/03/controls/">
  <control-tag name="error-message">
    <description>Properties relating to the error message.</description>
    <attribute name="phone-number" required="false">
      <description>
        Sets the phone number to call in case of an error.
      </description>
      <type>
        <text/>
      </type>
      <default-value>(555) 555-5555</default-value>
    </attribute>
  </control-tag>
</control-tags>
```

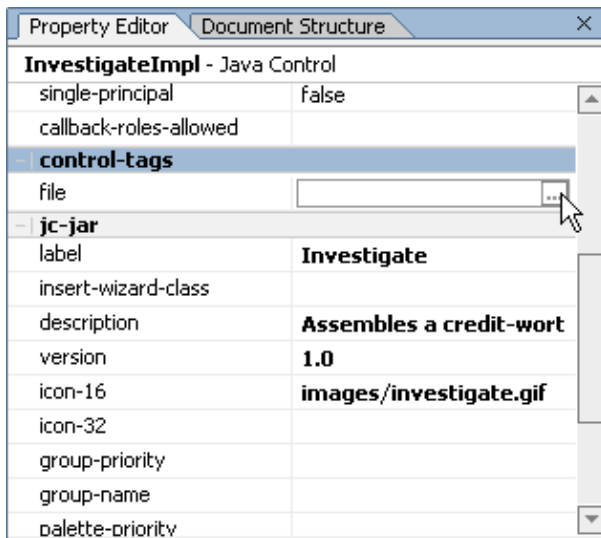
The <control-tag> element has a name attribute that specifies what the property will be called (error-message). The <description> element describes the property. The <attribute> element defines an attribute for the property: phone-number. The <attribute> element's "required" attribute is set to false, meaning that it's not necessary for a developer using the control to actually set the attribute in order for the control to work. As a result, the <default-value> element must be included here so that WebLogic Workshop knows what value to use if the developer doesn't explicitly set it. Finally, the <type> element specifies the attribute's type, similar to a data type. WebLogic Workshop will use this <type> value to do a bit of validation when the developer tries to set its value.

5. Press **Ctrl+S** to save your work. Press **Ctrl+F4** to close InvestigateProperties.xml.

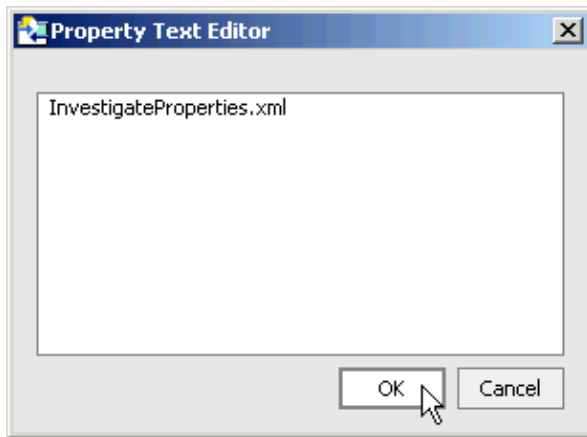


## To Edit Investigate to Use the New Property

1. On the **Application** tab, double-click **InvestigateImpl.jcs**.
2. Click the **Design View** tab, if necessary.
3. On the **Property Editor** tab, in the section labeled **control-tags**, to the right of the property named **file**, click the **ellipses** symbol.



4. In the **Property Text Editor**, enter InvestigateProperties.xml.



5. Click **Ok**.
6. Click the **Source View** tab.
7. Add the following code to the the body of InvestigateImpl.jcs.

```
private String _phoneNumber;
```

8. Add the following code to the top of the requestCreditReport method. Code to add appears in red.

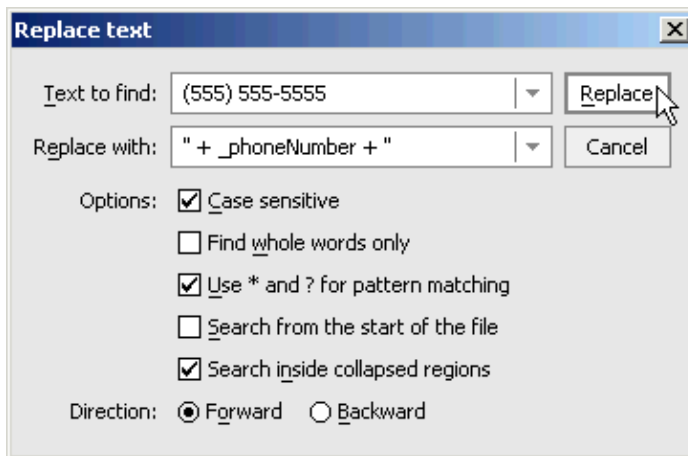
```
/**
 * @common:operation
 * @common:message-buffer enable="true"
 */
public void requestCreditReport(String taxID)
{
    _phoneNumber = context.getControlAttribute("jc:error-message", "phone-number");
```

## WebLogic Workshop Tutorials

```
m_currentApplicant.taxID = taxID;

/*
 * Retrieve data from the database and store it in the rec object.
 */
Record rec = bankruptciesDB.checkForBankruptcies(taxID);
```

9. Press **Ctrl+R**.
10. In the **Replace Text** dialog, in the **Text to find** field, enter **(555) 555-5555**  
In the **Replace with** field, enter **" + \_phoneNumber + "**

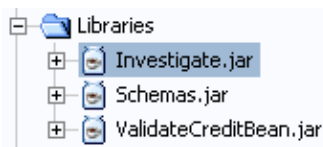


11. Click **Replace**.
12. In the **Confirm Replace** dialog click **All**.
13. Press **Ctrl+S** to save your work.

### To Compile the Investigate Control

- On the **Application** tab, right-click the **Investigate** project folder and select **Build Investigate**.

The Investigate project is compiled, packaged as a JAR file named **Investigate.jar** and placed in the **Libraries** folder. If you open the **Libraries** folder on the **Application** tab, you will see the following:



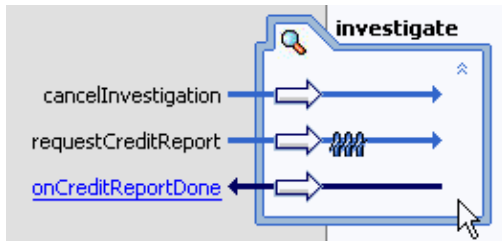
Note that this is a distributable JAR file: place it in the Libraries folder of any application to use the functionality of the Investigate Java control. (Note that any application that uses the JAR file must also have access to the Database, web service, JMS application, and EJB application that are used by the Java control.)

Also note that once you have compiled a Java control, WebLogic Workshop uses the JAR version by default, instead of working directly from the source code contained in the control project. For this reason, you could remove the control project from your application, and your test web service would still run successfully, because it would use the JAR version of the Java control.

### To Test the Investigate Java Control

1. On the **Application** tab, double-click **InvestigateTest.jws**.

2. Click the **Design View**.
3. Select the icon for the Investigate control. (Note that the GIF file has been incorporated into the new icon.)



4. On the **Property Editor** tab, in the section labeled **error-message**, locate the **phone-number** property, and enter (123) 456-7890.
5. Click the Start button, shown here:



Workshop builds TestClient.jws and launches the Test Browser.

6. In the **Workshop Test Browser**, in the **taxID** field, enter **wrong\_taxid** and click the **requestCreditReport** button.

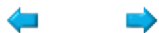
Note that you are intentionally entering an invalid taxID to cause an error message. When the callback arrives you should see the following message.

```

Client Callback
Submitted at Thursday, August 21, 2003 5:28:57 PM PDT
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1061512137565</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns:onCreditReportDone xmlns:ns="http://www.openuri.org/">
      <ns:applicant>
        <ns:availableCCCredit>0</ns:availableCCCredit>
        <ns:currentlyBankrupt>false</ns:currentlyBankrupt>
        <ns:creditScore>0</ns:creditScore>
        <ns:taxID>wrong_taxid</ns:taxID>
        <ns:message>No data could be found on the applicant. Please call (123) 456-7890 for assistance.
        </ns:message>
      </ns:applicant>
    </ns:onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
    
```

Note the value of the `<ns:message>` element has incorporated the phone number you specified: (123) 456-7890.

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



# Summary: Java Control Tutorial

This tutorial introduced you to the basics of building Java controls.

This topic lists ideas this tutorial introduced. You may also find it useful to look at the following:

- For links to topics about the basics of using built-in Java controls in WebLogic Workshop, see [Using Built-In Java Controls](#).
- For links to topics about building your own custom Java controls, see [Building Custom Java Controls](#).

## Concepts and Tasks Introduced in This Tutorial

- Java controls are built from a Java control source file (JCS) and an interface file (JAVA) .
- You can use Java controls to encapsulate complex functionality.
- Java controls can communicate asynchronously with other components.
- Java controls can be compiled into JAR files for general distribution to other containers.
- Control property attribute values are stored as Javadoc annotations in source code.

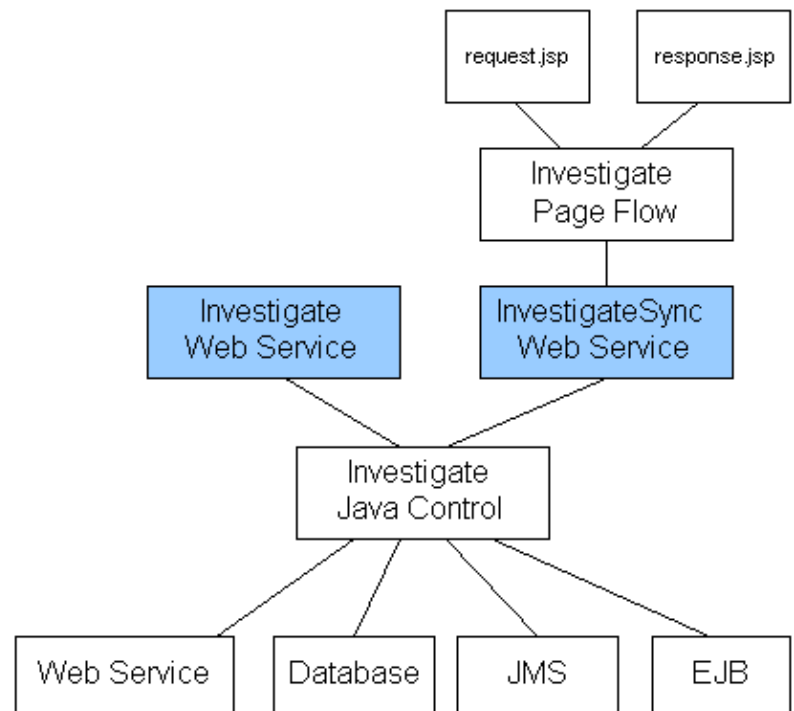
Click one of the following arrows to navigate through the tutorial:



# Tutorial: Web Services

## The Big Picture

The web service tutorial is the second in a series of three tutorials, which builds the Investigate Application. The Investigate Application is designed to assemble a credit-worthiness report on loan applicants. (*Note:* you do **not** need to execute the tutorial series in order. You may proceed with this tutorial, even if you have not completed the previous tutorial in the series. Or you may skip ahead to the next tutorial in the series if you wish.)



The first tutorial in the series, the Tutorial: Java Control, builds the core of the application: the Investigate Java control, which consults a variety of components to compile the credit-worthiness report.

This, the second tutorial in the series, builds the web services tier of the application. You will build two web service access points to the functionality encapsulated by the Investigate Java control. These two web service access points are shown in blue in the diagram to the right.

The third tutorial in the series, the Tutorial: Page Flow, builds a web-based user interface for the Investigate Application.

## Tutorial Goals

This tutorial will teach you how to create and test a web service with WebLogic Workshop. Along the way, you will also learn how to create methods that expose a service's functionality, become acquainted with WebLogic Workshop's support for asynchronous communication and loose coupling, and learn how controls speed the design process.

## Tutorial Overview

### Steps in This Tutorial

Step 1: Begin the Investigate Web Service — 20 minutes

## WebLogic Workshop Tutorials

In this step you build and run your first WebLogic Workshop web service. You start WebLogic Workshop and WebLogic Server, then create a web service that has a single method.

### Step 2: Add Support for Asynchronous Communication — 20 minutes

You will work around a problem inherent in communication across the web: the problem of network latency. Network latency refers to the time delays that often occur when transmitting data across a network.

### Step 3: Add Support for Synchronous Communication — 20 minutes

You add support for clients that cannot hear callbacks.

### Step 4: Add Support for XML Mapping — 25 minutes

You will add an XML transformation to your web service.

### Step 5: Add a Script for XML Mapping — 15 minutes

Modify the XML transformation with a call to a script file.

### Step 6: Web Service Security — 20 minutes

You secure your web services with SSL and web service security (WSSE security).

### Summary: Web Service Tutorial

You review the concepts and technologies covered in the tutorial. This step provides additional links for more information about each area covered.

To begin the tutorial, see Step 1: Begin the Investigate Web Service.

Click the arrow to navigate to the next step.



# Step 1: Begin the Investigate Web Service

In this step, you will use WebLogic Workshop to create a simple, synchronous web service called the Investigate web service. You will also deploy and run Investigate on WebLogic Server.

The tasks in this step are:

- To start WebLogic Workshop
- To create a new application
- To start WebLogic Server
- To create a new web service
- To add a method
- To specify a parameter and return value
- To test the Investigate Web Service

To Start WebLogic Workshop

... on Microsoft Windows

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

1. From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**QuickStart**.
2. In the **QuickStart** dialog, click **Experience WebLogic Workshop 8.1**.

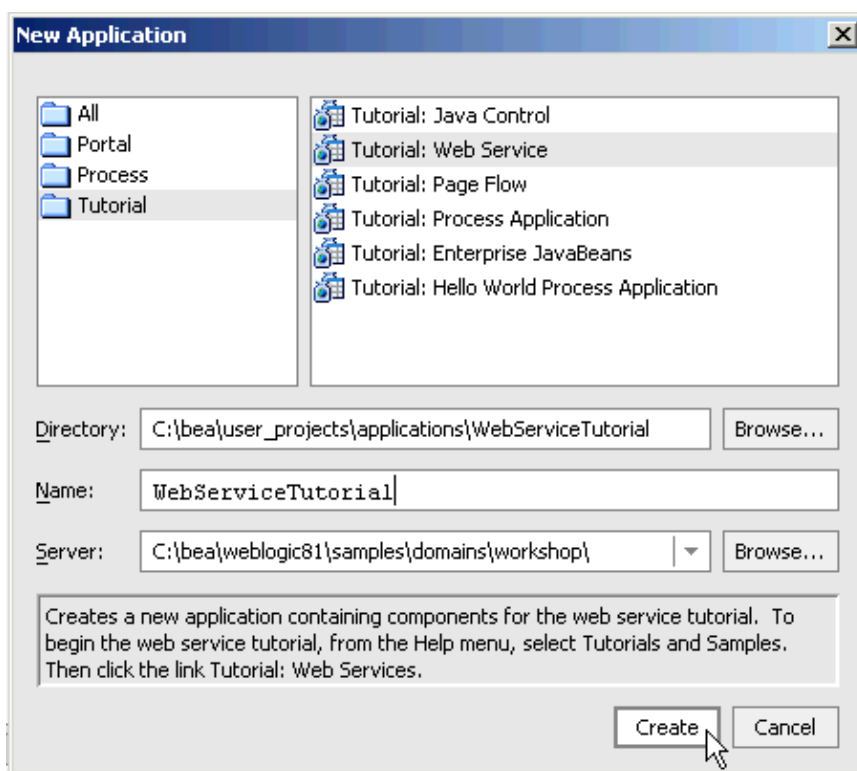
...on Linux

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

1. Open a file system browser or shell.
2. Locate the **Workshop.sh** file at the following address:  
`$HOME/bea/weblogic81/workshop/Workshop.sh`
3. In the command line, type the following command:  
`sh Workshop.sh`

To Create a New Application

1. From the **File** menu, select **New** —> **Application**. The **New Application** dialog appears.
2. In the **New Application** dialog, in the upper left-hand pane, select **Tutorial**.  
In the upper right-hand pane, select **Tutorial: Web Service**.  
In the **Directory** field, use the **Browse** button to select a location to save your source files. (The folder you choose is up to you. The default location is shown in the illustration below.)  
In the **Name** field, enter **WebServiceTutorial**.  
In the **Server** field, from the dropdown list, select **BEA\_HOME\weblogic81\samples\domains\workshop**.



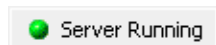
3. Click **Create**.

A new Tutorial: Web Service application is created. Note that this application contains a number of resources that you will use in this tutorial. The most important resource is the Investigate Java control. For details on the structure of and functionality of the Investigate Java control see Tutorial: Java control.

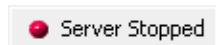
### To Start WebLogic Server

Once you have created a new web service tutorial application, you must ensure that WebLogic Server is running while you build your web service.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.



If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow the instruction below to start WebLogic Server.

- **To start WebLogic Server:** from the **Tools** menu, choose **WebLogic Server**—>**Start WebLogic Server**.

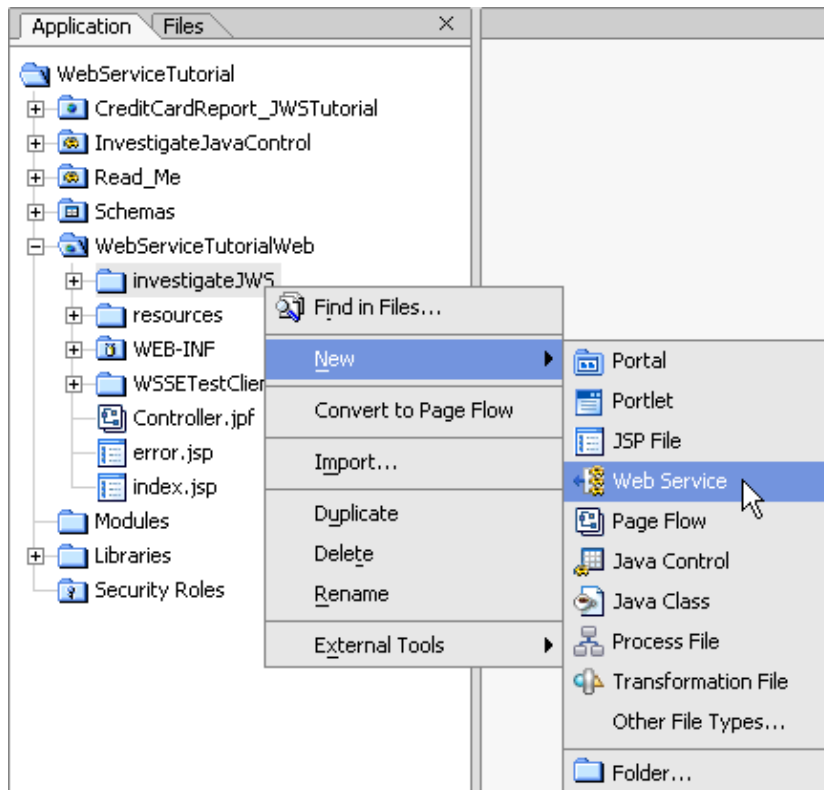
**Note:** On the **Start up Progress** dialog, you may click **Hide** and continue to the next task.



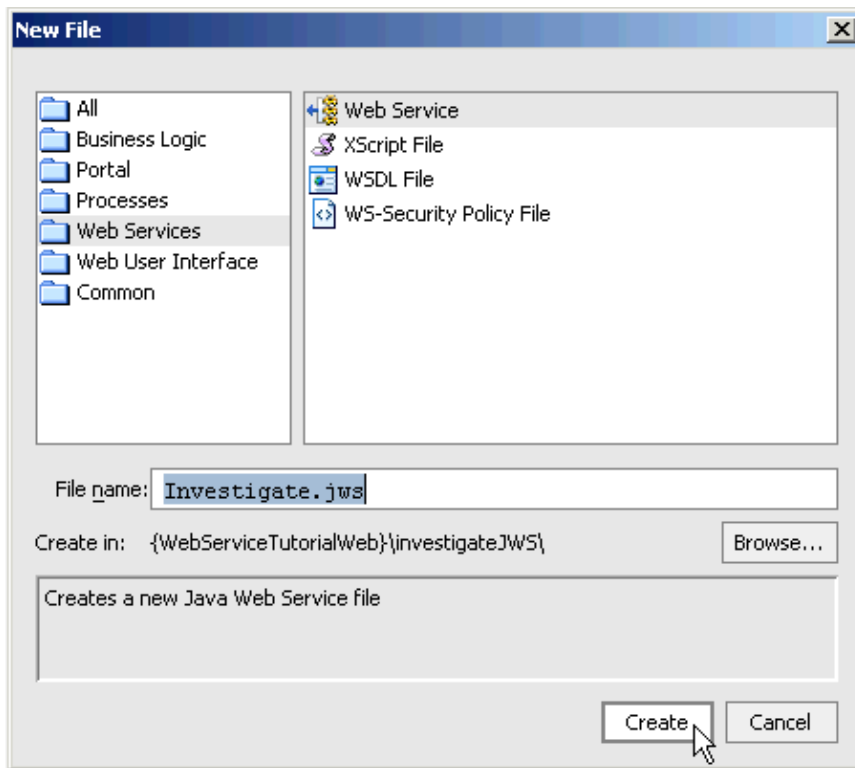
### To Create a New Web Service

In this step you will create a new web service called Investigate.jws. The code for a web service is contained within a JWS file ("JWS" stands for "Java Web Service"). A JWS file is a JAVA file in that it contains code for an ordinary Java class. But, because a file with a JWS extension contains the implementation code intended specifically for a web service class, the extension gives it special meaning in the context of WebLogic Server, where your web service will be deployed.

1. On the **Application** tab, right-click the folder **WebServiceTutorial/WebServiceTutorialWeb/investigateJWS** folder and select **New-->Web Service**.

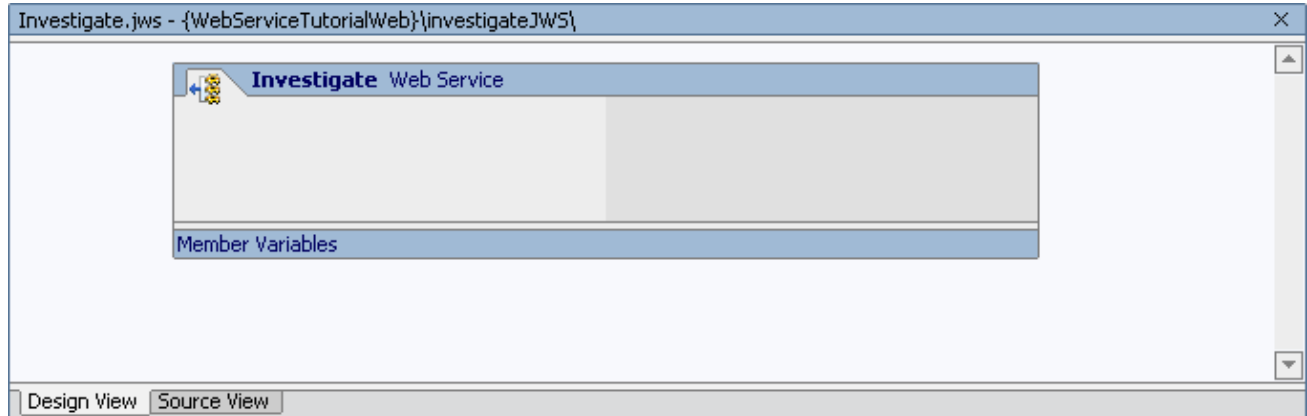


2. In the **New File** dialog, in the upper left-hand pane, confirm that **Web Services** is selected. In the upper right-hand pane, confirm that **Web Service** is selected. In the **File name** field, enter Investigate.jws.



3. Click **Create**.

A new JWS file, Investigate.jws, is created and displayed in the main work area in Design View.

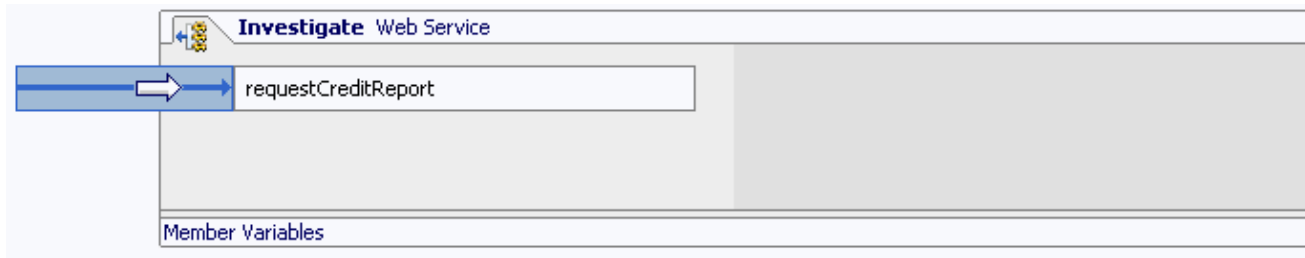


#### To Add a New Method

Web services expose their functionality through methods, methods that clients invoke when they want to request something from the web service. In this case, clients will invoke a method to request credit reports. The service you are building will eventually collect credit information from other sources. But for now, to keep things simple, you will provide a method that returns a simple report immediately.

1. Confirm that Investigate.jws is displayed in the main work area. (Throughout this tutorial, to display a file in the main work area, double-click its icon on the **Application** tab.)
2. Click the **Design View** tab.
3. From the **Insert** menu, select **Method**.

4. In the space provided, replace newMethod1 with requestCreditReport and press **Enter**.



**Note:** If you switched from WebLogic Workshop to another application (for example, to read this topic), the method name may no longer be available for editing. To re-open it for editing, right-click its name, select **Rename**, type requestCreditReport, and then press **Enter**.

## To Specify a Parameter and Return Value

In this task you will complete the requestCreditReport method by adding a parameter and a return value.

1. In Design View, click the link text for the **requestCreditReport** method, as shown in the following illustration.



The web service switches from Design View to Source View, with the cursor placed in front of the requestCreditReport method, as shown below.

```
/**
 * @common:operation
 */
public void requestCreditReport()
{
}
```

Note: the method has a Javadoc comment (Javadoc comments always begin with `/**` and end with `*/`). Javadoc was originally introduced as a way to specify inline documentation that could be extracted and presented as HTML. WebLogic Workshop uses a variety of Javadoc annotations to indicate how web service code should be treated by WebLogic Server at runtime. In this case, the Javadoc annotation `"@common:operation"` indicates that the method should be exposed to clients as accessible over the internet.

2. Edit the **requestCreditReport** method body so it appears as follows. Code to edit appears in red.

```
/**
 * @common:operation
 */
public String requestCreditReport(String taxID)
{
    return "Applicant " + taxID + " is approved.";
}
```

As you can see, this is not a very interesting credit evaluation process. In the next step of the tutorial, you will implement a more precise evaluation process.

3. Press **Ctrl+S** to save your work.

As it is now written, the **requestCreditReport** method works like this: the client invokes the method by supplying a tax identification number, the method then immediately returns a String to the client saying that the applicant with the supplied tax identification number was approved.

Before you go on to test your web service, click the **Design View** tab.



The long blue arrow indicates that the `requestCreditReport` method is an operation: a method of your web service that is exposed to clients. (The long blue arrow is the graphical equivalent of the Javadoc annotation `@common:operation`.)

The white arrow pointing to the right indicates that the `requestCreditReport` method takes one or more parameters. In this case, the `requestCreditReport` takes one parameter: the String `taxID`.

The white arrow pointing to the left indicates that the method returns some data. In this case, the method returns the String value "Applicant " + `taxID` + " is approved."

### To Test the Investigate Web Service

You can test and debug code with the **Workshop Test Browser**, a browser-based tool through which you can call methods of your web service.

1. Click the Start button, shown in the following illustration.



Clicking the Start button prompts WebLogic Workshop to build your project, checking for errors along the way. WebLogic Workshop then launches a web browser through which you can test your web service. The following illustration shows the Test Form page that appears after you click the Start button.

The screenshot shows the 'Investigate.jws Web Service' test interface. It has tabs for 'Overview', 'Console', 'Test Form', and 'Test XML'. The 'Test Form' tab is active, showing a 'requestCreditReport' method. There is a text input field for 'string taxID:' and a 'request Credit Report' button. A 'Message Log' section on the left shows 'Log is empty' and a 'Refresh' button. The URL bar shows 'http://localhost:7001/WebServiceTutorial/Web/investigateJWS/'.

As you can see, this page provides tabs for access to other pages of the Test View. Test Form and Test XML provide alternative ways to specify the content of the call to your service's methods. Overview and Console include additional information and commands useful for debugging a service.

2. If it is not already selected, click the **Test Form** tab.
3. In the **string taxID** field, enter a value. (This value can be any string value.)
4. Click **requestCreditReport**.

The browser refreshes to display a summary of your request parameters and your service's response, as shown here:

**Message Log** Refresh

requestCreditReport

Clear Log

**Service Request requestCreditReport**  
Submitted at Thursday, May 8, 2003 8:57:14 AM PDT

taxID = 123456789

**Operation requestCreditReport**  
Submitted at Thursday, May 8, 2003 8:57:14 AM PDT  
Method: investigateJWS.Investigate.requestCreditReport  
Arguments:  
taxID : 123456789  
CallStack:  
requestCreditReport()

**Returned from requestCreditReport**  
Submitted at Thursday, May 8, 2003 8:57:14 AM PDT  
Return value: Applicant123456789 is approved.

**Service Response**  
Submitted at Thursday, May 8, 2003 8:57:14 AM PDT  
<string xmlns="http://www.openuri.org" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">Applicant123456789 is approved.</string>

Under **Service Request**, the summary displays the essence of what was sent by the client (= you) when the method was called. It shows the parameter values passed with the method call, as in the following example:

taxID = 123456789

Under **Service Response**, the summary displays what was returned by the Investigate web service. The response is formatted as a fragment of Extensible Markup Language (XML), this is the payload of the SOAP message returned by the web service.

To the left of the request and response is the **Message Log** area. This area lists a separate entry for each test call to the service method. Entries in the Message Log correspond to your service's methods, updating with a new entry for each test you make.

To try this out, click the **Test operations** link to return to the original Test Form page, then enter a new value in the **string taxID** box and click **requestCreditReport**. When the page refreshes, request

## WebLogic Workshop Tutorials

and response data corresponding to your second test is displayed as a second entry in the Message Log. You can click each log entry to view the data for the corresponding test. Click ***Clear Log*** to empty the list of log entries and start fresh.

5. Return to WebLogic Workshop, and click the ***Stop*** button (shown below) to close the ***Workshop Test Browser***.



### Related Topics

#### Applications and Projects

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



## Step 2: Add Support for Asynchronous Communication

In this step you will construct a web service entry point for the Investigate Java control (the Investigate Java control is the subject of Tutorial: Java Controls). Making a web service entry point for the Investigate Java control solves one of the problems inherent in communication across the web: the problem of network *latency*. Network latency refers to the time delays that often occur when transmitting data across a network.

This can be a particular problem on the internet, which has highly unpredictable performance characteristics and uncertain reliability. There may be additional latency if it takes a long time for a web service to process a request (for example, if a person "behind" the web service must review the applicant's credit before a response can be returned).

So far, the design of the Investigate web service forces the client calling the `requestCreditReport` method to halt its own processes and wait for the response from the web service. This is called a *synchronous* relationship, in which the client software invokes a method of the web service and is blocked from continuing its own processes until that method returns a value.

You will solve the latency problem by enabling your web service to communicate with its clients *asynchronously*. In asynchronous communication, the client communicates with the web service in such a way that it is *not* forced to halt its processes while the web service produces a response. In particular, you will add a method to the web service that immediately returns a simple acknowledgement to the client, thereby allowing the client to continue its own processes. You will also add a callback that returns the full results to the client at a later time. Finally, you will implement support for conversations that enable your web service to remember which client to send the full response to via the callback.

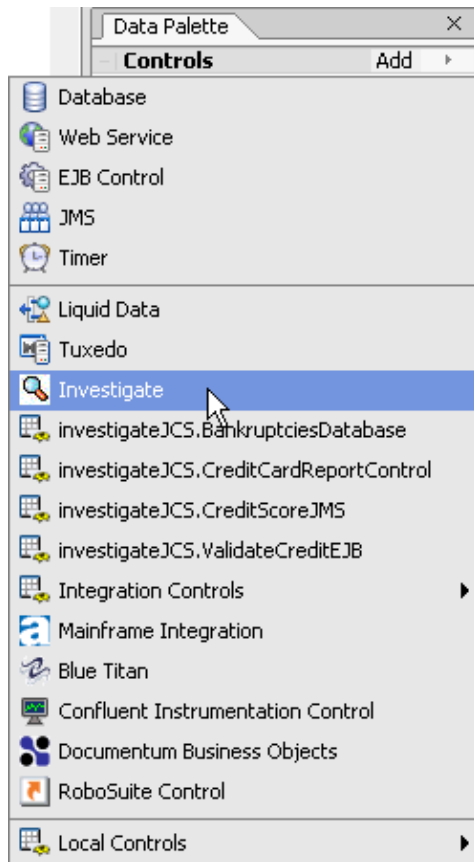
The tasks in this step are:

- To add the Investigate Java control
- To edit the `requestCreditReport` method
- To add a callback
- To edit the callback handler
- To add a buffer and conversation support
- To test the Investigate web service

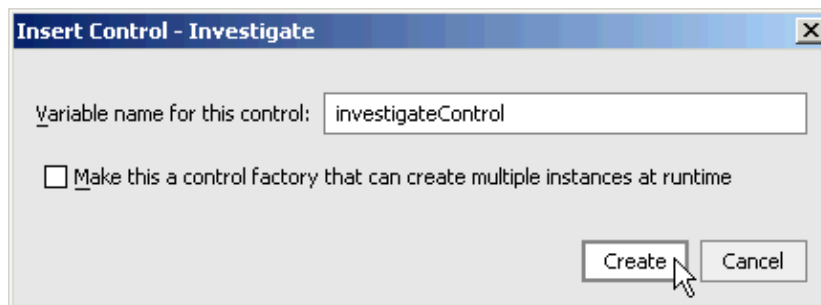
Add the Investigate Java Control

In this task you will add the functionality provided by the Investigate Java control.

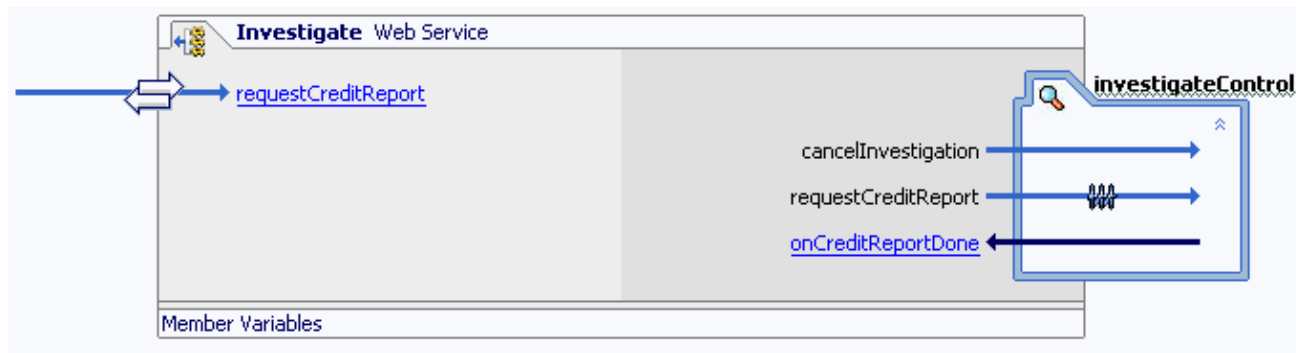
1. Confirm that ***Investigate.jws*** is displayed in the main work area.
2. Click the ***Design View*** tab.
3. Click ***View-->Windows-->Data Palette***.
4. On the ***Data Palette***, in the section labeled ***Controls***, select ***Add-->Investigate***.



5. In the **Insert Control – Insert Investigate** dialog, enter `investigateControl` and click **Create**.



The Investigate Java control is added to your web service.



To Edit the requestCreditReport Method



## WebLogic Workshop Tutorials

In this task you will edit your web service's `requestCreditReport` method so that (1) it returns an acknowledgement to the invoking client without returning any information about the applicant and (2) invokes the `requestCreditReport` method on the Investigate Java control. The Investigate Java control will respond by sending a callback containing the credit report to your web service.

1. Confirm that *Investigate.jws* is displayed in the main work area.
2. If necessary, click the *Design View* tab.
3. Click the link text for the *requestCreditReport* method.



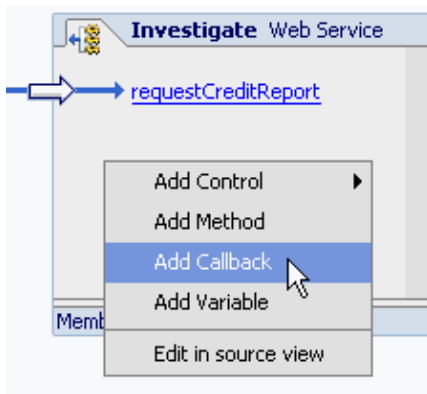
4. Edit the *requestCreditReport* method code so that it appears as follows. Code to edit appears in red.

```
/**
 * <p>Use the following taxID's to test the Investigate web service.</p>
 *
 * <blockquote>111111111<br>
 * 222222222<br>
 * 333333333<br>
 * 444444444<br>
 * 555555555<br>
 * 123456789</blockquote>
 *
 * @common:operation
 */
public void requestCreditReport(String taxID)
{
    /*
     * Invoke the requestCreditReport method on the
     * Investigate Java control passing in the taxID
     * as a parameter.
     */
    investigateControl.requestCreditReport(taxID);
}
```

### To Add a Callback

Once the web service gets the credit report from the Investigate Java control, you want to be able to pass the credit report along to the client. The callback you add below will send the credit report to your web service's clients.

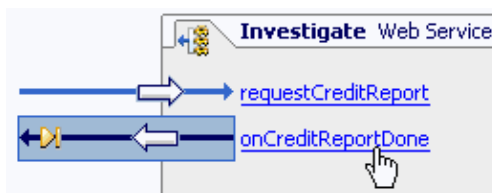
1. Confirm that *Investigate.jws* is displayed in the main work area.
2. Click the *Design View* tab.
3. Right-click the picture of the Investigate web service and select *Add Callback*.



4. In the input field that appears, replace newCallback1 with onCreditReportDone and press **Enter**.



5. Click the link text for the **onCreditReportDone** callback.



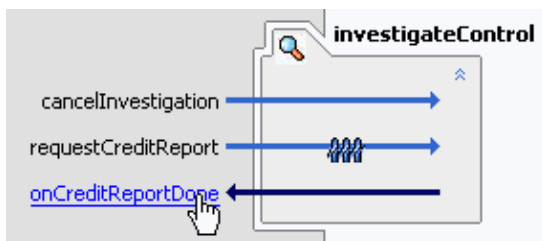
6. Edit the code for onCreditReportDone so that it appears as follows. Add the code appearing in red.

```
public interface Callback extends com.bea.control.ServiceControl
{
    void onCreditReportDone(investigateJCS.Applicant applicant);
}
```

#### To Edit the Callback Handler

When the Investigate Java control sends a callback message to your web service, your web service needs to know how to handle the message. In this task you will edit the handler for callbacks sent from the Investigate Java control.

1. Confirm that **Investigate.jws** is displayed in the main work area.
2. Click the **Design View** tab.
3. Click the link text for the investigate Java Control callback handler **onCreditReportDone**.



4. Edit the callback handler **investigateControl\_onCreditReportDone** to look like the following. Code to edit appears in red.

## WebLogic Workshop Tutorials

```
public void investigateControl_onCreditReportDone(investigateJCS.Applicant m_currentApplicant)
{
    /*
     * When the callback message from the Investigate control arrives,
     * then pass the message along to the Investigate web service callback
     * onCreditReportDone.
     */
    callback.onCreditReportDone(m_currentApplicant);
}
```

**Note:** Callback handlers always have a name composed of the name of the control, plus the underscore character, plus the name of the callback method being handled. So "investigateControl\_onCreditReportDone" means: the handler for investigateControl's callback method onCreditReportDone.

The request/response cycle now works like this: (1) a client invokes your web service's requestCreditReport method; (2) your web service then invokes the Investigate Java control's method requestCreditReport; (3) when the Investigate Java control has assembled the credit report, it send a callback to your web service; (4) your web service's callback handler investigateControl\_onCreditReportDone invokes your web service's callback onCreditReportDone; (5) finally your web service's callback onCreditReportDone sends the complete credit report to the original client.

### To Add Buffer and Conversation Support to the Web Service

Now you are ready to add a buffer and conversation support to your web service.

The buffer, which you will add to the requestCreditReport method, serves two purposes. First, it saves client requests in a message queue which prevents requests from being lost during server outages. Second, the buffer immediately returns an acknowledgement to the client, which allows the client to continue its own processes without waiting for the full response from the web service.

Adding conversation support to your web service lets the client know for sure that the result your service sends back through the onCreditReportDone callback is associated with the request the client originally made. If the same client makes two independent calls with the same taxpayer ID (say, for separate loan applications from the same applicant), how will it know which finished report corresponds to which request? With all the interaction back and forth between the client and your web service, your service needs a way to keep things straight.

Also, if the server goes down (taking the m\_applicant member variable with it), the data that the client sent as a parameter is lost. The service not only loses track of the client's request, but of who the client was in the first place.

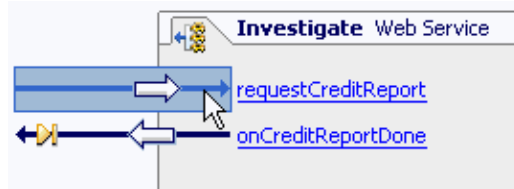
You can solve these problems by associating each exchange of information (request, response, and any interaction needed in between) with a unique identifier—an identifier known to both the client and the service.

In web services built with WebLogic Workshop, you do this by adding support for a conversation. For services participating in a conversation, WebLogic Server stores state-related data, such as the member variable you added, on the computer's hard drive and generates a unique identifier to keep track of the data and of which responses belong with which requests. This infrastructure remains in place as long as the conversation is ongoing. When the conversation is finished, the resources allocated to the infrastructure are released.

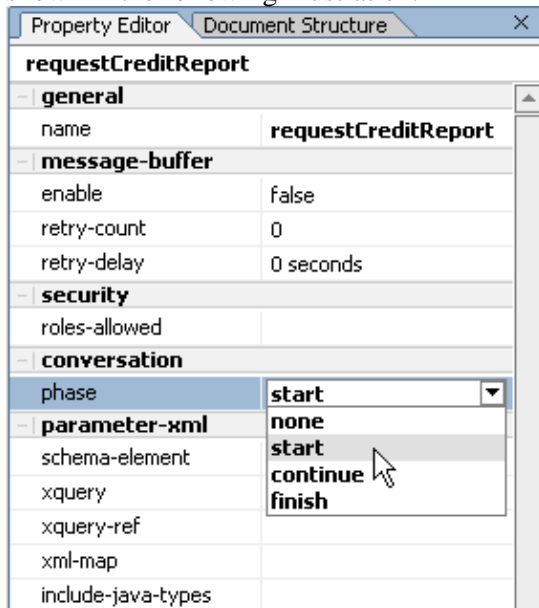
When you add methods to a web service that supports conversations, you can indicate whether each method

starts, continues or finishes a conversation. Adding support for a conversation is very easy. With methods that represent both the first step of the transaction (`requestCreditReport`) and the last step (`onCreditReportDone`), you want to indicate when the conversation starts and when it finishes. To add a buffer and conversation support to your web service follow the steps below.

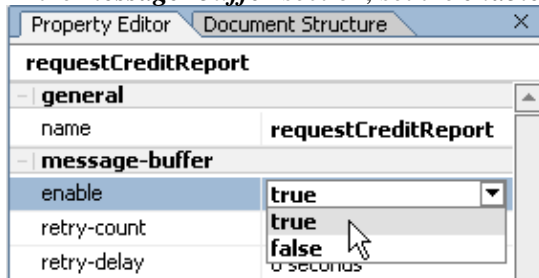
1. Confirm that *Investigate.jws* is displayed in the main work area.
2. Click the **Design View** tab.
3. Click the arrow icon associated with the *requestCreditReport* method.



4. On the **Property Editor** tab, in the section labeled *conversation*, set the *phase* property to *start*, as shown in the following illustration.



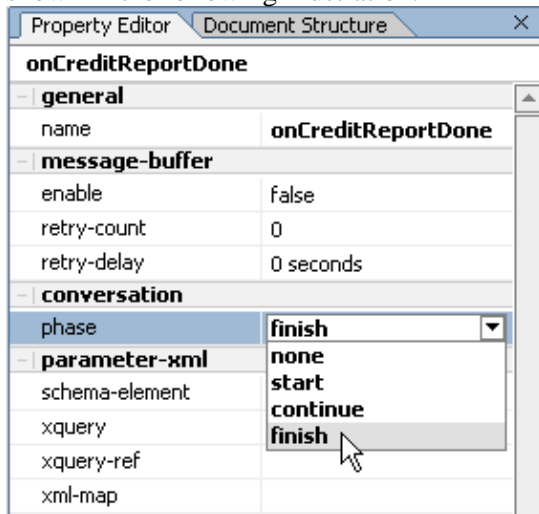
5. In the *message-buffer* section, set the *enable* property to *true*, as shown in the following illustration.



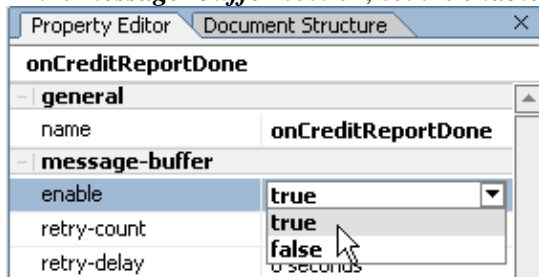
6. In **Design View**, click the arrow icon next to your web service's *onCreditReportDone* callback.



- On the **Property Editor** tab, in the section labeled **conversation**, set the **phase** property to **finish**, as shown in the following illustration.

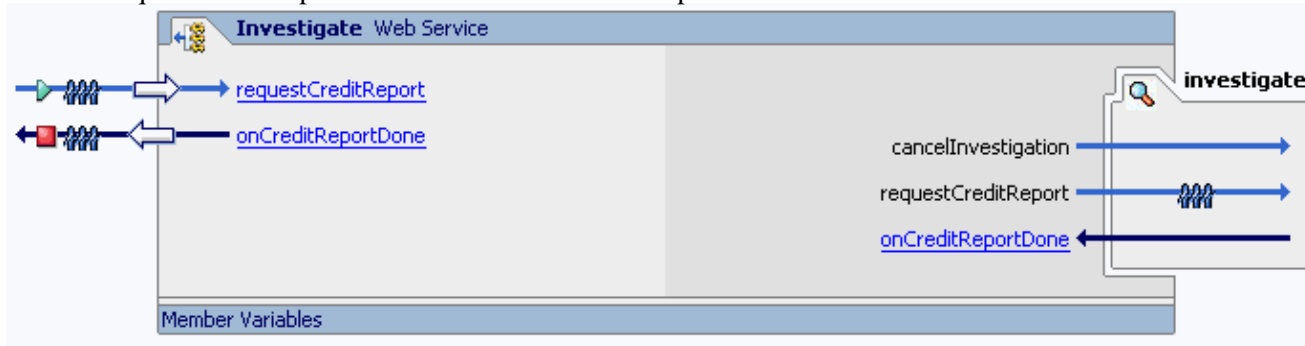


- In the **message-buffer** section, set the **enable** property to **true**, as shown in the following illustration.



- Press **Ctrl+S** to save your work.

After modifying your web service through the **Property Editor**, the Design View picture of your web service should look like the following image. Notice the new conversation and buffer icons associated with the requestCreditReport method and the onCreditReportDone callback.



## To Test the Investigate Web Service

Now you are ready to compile and test your web service.

- Confirm that **Investigate.jws** is displayed in the main work area.
- Click the Start button, shown below.



Your web service compiles and Workshop Test Browser launches.

3. In the **taxID** field enter the 9 digit number 222222222.

**Note:** Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click the **requestCreditReport** button.

The Test Browser refreshes to display a summary of the request parameters sent by the client and your service's response, as shown here:

**Service Request requestCreditReport**  
Submitted at Thursday, May 8, 2003 10:50:49 AM PDT

taxID = 222222222  
.CONVPHASE = .START  
.CONVERSATIONID = 1052416249913

**Service Response**  
Submitted at Thursday, May 8, 2003 10:50:49 AM PDT  
<Void xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true">  
</Void>

**Note:** The **Service Request** section displays what the client (in this case, the Workshop Test Browser) sent to the Investigate web service to invoke the requestCreditReport method. Note the conversation identification number, this is a unique number generated for each time a conversation starting method is invoked. The web service uses this id to associate the initial client request and the callback response it will later send back to the client.

The **Service Response** section shows what the Investigate web service has sent back (so far) to the client. In this case it sends back an XML message, **<Void xmlns:xsi=...**, serving as an acknowledgement to the client that its request has been received.

5. Click **Refresh** until the **Message Log** shows an entry for **callback.onCreditReportDone**.
6. Click **callback.onCreditReportDone** to view the contents of the callback sent from the Investigate web service to the client, as shown here:

**Message Log**
 Refresh

1061515863626  
→ requestCreditReport  
investigateControl:creditCardReportControl.getCreditCardData→  
investigateControl:creditCardReportControl.onCreditCardDataReady←  
♦ ← **callback.onCreditReportDone**  
Conversation 1061515863626 is finished.  
 Clear Log

**Client Callback**  
Submitted at Thursday, August 21, 2003 6:31:05 PM P

<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-in" >  
<SOAP-ENV:Header>  
<CallbackHeader xmlns="http://www.openuri.org" >  
<conversationID>1061515863626</conversationID>  
</CallbackHeader>  
</SOAP-ENV:Header>  
<SOAP-ENV:Body>  
<ns:onCreditReportDone xmlns:ns="http://www.openuri.org" >  
<ns:applicant>  
<ns:availableCCCredit>2000</ns:availableCCCredit>  
<ns:currentlyBankrupt>false</ns:currentlyBankrupt>  
<ns:lastName>Smith</ns:lastName>  
<ns:firstName>John</ns:firstName>  
<ns:approvalLevel>Applicant is a high risk</ns:approvalLevel>  
<ns:creditScore>560</ns:creditScore>  
<ns:taxID>222222222</ns:taxID>  
</ns:applicant>  
</ns:onCreditReportDone>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>

The **Client Callback** section displays the SOAP message that the client receives, including data about the credit applicant. The two other entries in the **Message Log** show some of the activity of the Investigate Java control as it assembles the report on the credit applicant.

## WebLogic Workshop Tutorials

7. Return to WebLogic Workshop, and click the **Stop** button (shown below) to close the **Workshop Test Browser**.



Related Topics

Overview: Conversations

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



## Step 3: Add Support for Synchronous Communication

The Investigate web service sends callbacks to its clients. But not all clients can hear callbacks, so not all clients can use the Investigate web service as an access point to the Investigate Java control. In this step you will build a *synchronous* web service interface that allows non-callbackable clients to access the Investigate Java control.

The tasks in this step are:

- To create a web service
- To add the Investigate Java control
- To add two member variables
- To add polling logic
- To add code to finish the conversation
- To Test InvestigateSync.jws

To Create a Web Service

1. On the **Application** tab, right-click the *investigateJWS* folder and select **New-->Web Service**.
2. In the **New File** dialog, in the **File name** field, enter InvestigateSync.jws, and click **Create**.

To Add the Investigate Java Control

1. On the **Data Palette**, in the section labeled **Controls**, select **Add-->Application Controls-->Investigate**.
2. In the **Insert Control – Insert Investigate** dialog, enter investigateControl and click **Create**.  
The Investigate Java control is added to your web service.

To Add Two Member Variables

In this step you will add two member variables to the InvestigateSync web service: one to store the credit report returned by the Investigate Java control, the other to record the status of the investigation process.

1. Click the **Source View** tab to view the source code for InvestigateSync.jws
2. Add the following code to the body of InvestigateSync.jws. Code to add is shown in red.

```
/*
 * m_applicant stores the credit report returned by the Investigate Java control
 * for retrieval at a latter time by the client.
 */
public investigateJCS.Applicant m_applicant = new investigateJCS.Applicant();

/*
 * m_isReportReady is set to true when the credit report from the Investigate Java control
 * has been received, signaling the client to retrieve the report.
 */
public boolean m_isReportReady = false;
```

To Add Polling Logic



In this task you will add the methods through which the client will request and collect the credit report from the InvestigateSync web service. These methods are designed to work in the following way: (1) the client invokes the web service's requestCreditReport method. (2) The web service pass the request onto the Investigate Java control and waits for a callback containing the credit report. (3) When the callback from the Investigate Java control arrives the Investigate web service stores the report in the member variable m\_applicant and sets m\_isReportReady to true. (4) If the client has been checking the value of m\_isReportReady (through the isReportReady method), it now knows that it can collect the credit report through the getCreditReport method.

These methods are collectively referred to as "polling logic" because the client polls the methods, asking for the status of the credit report.

1. If InvestigateSync.jws is not displayed in Source View, click the **Source View** tab.
2. Add the following methods to the body of InvestigateSync.jws. Code to add is shown in red.

```

/**
 * Calling this method requests a credit report from the Investigate Java control.
 *
 * @common:operation
 * @common:message-buffer enable="true"
 * @jws:conversation phase="start"
 */
public void requestCreditReport(String taxID)
{
    investigateControl.requestCreditReport(taxID);
}

/**
 * After making a request for credit report, clients check this method periodically
 * to see if the report is ready. When this method returns true, clients should inv
 * the getCreditReport method to retrieve the credit report.
 *
 * @common:operation
 * @jws:conversation phase="continue"
 */
public boolean isReportReady()
{
    return m_isReportReady;
}

/**
 * Clients collect the completed credit report by invoking this method.
 *
 * @common:operation
 * @jws:conversation phase="continue"
 */
public investigateJCS.Applicant getCreditReport()
{
    /**
     * If m_isReportReady is true (i.e., if the callback handler investigateControl_
     * has been invoked) return the results to the client.
     * If m_isReportReady is false, tell the client to check back later for the resu
     */
    if(m_isReportReady)
    {
        return m_applicant;
    }
    else

```

## WebLogic Workshop Tutorials

```
{
    m_applicant.message = "The report is not yet ready. Please check back later";
    return m_applicant;
}

}

/**
 * This method allows the client to cancel the request for a credit report.
 *
 * @common:operation
 * @jws:conversation phase="finish"
 * @jws:message-buffer enable="true"
 */
public void cancelInvestigation()
{
    investigateControl.cancelInvestigation();
}

/**
 * When the callback handler is invoked by the Investigate Java control,
 * store the credit report in the member variable m_applicant
 * and set m_isReportReady to true.
 */
public void investigateControl_onCreditReportDone(investigateJCS.Applicant applicant)
{
    m_applicant = applicantReport;

    m_isReportReady = true;
}
}
```

3. Press **Ctrl+S** to save your work.
4. Click the **Design View** tab. Your web service should look like the following illustration.



### To Add Code to End the Conversation

There is one problem yet to be solved: how will the web service's conversation ever end, since the client will typically never invoke a method marked with the annotation `@jws:conversation phase="finish"`? You will solve this problem by adding an API call to the `getCreditReport` method.

1. If `InvestigateSync.jws` is not displayed in Source View, click the **Source View** tab.

2. Add the following code to the body of InvestigateSync.jws

```
/**
 * @common:context
 */
com.bea.control.JwsContext context;
```

3. Edit the getCreditReport method to look like the following. Code to add appears in red.

```
public investigateJCS.Applicant getCreditReport()
{
    /*
     * If m_isReportReady is true (i.e., if the callback handler investigateControl_
     * has been invoked) return the results to the client.
     * If m_isReportReady is false, tell the client to check back later for the result.
     */
    if(m_isReportReady)
    {
        /*
         * Calling context.finishConversation ends the current conversation.
         * Although the call to context.finishConversation() appears before the return
         * statement, it is executed after the return statement.
         */
        context.finishConversation();
        return m_applicant;
    }
    else
    {
        m_applicant.message = "The report is not yet ready. Please check back later";
        return m_applicant;
    }
}
```

4. Press **Ctrl+S** to save your work.

To Test InvestigateSync.jws

The Workshop Test Browser is an appropriate client for testing InvestigateSync.jws because it is a client that cannot hear callbacks from web services.

1. Confirm that **InvestigateSync.jws** is displayed in the main work area.
2. Click the Start button, shown below.



Your web service compiles and Workshop Test Browser launches.

3. In the **taxID** field, enter 22222222 and click the **requestCreditReport** button.

**Note:** Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click **Continue this Conversation**.
5. Click the **isReportReady** button.

If the **isReportReady** method returns **false**, then go back to step 4 above.

If the **isReportReady** method returns **true**, then continue to step 6 below.

6. Click **Continue the Conversation**.

## WebLogic Workshop Tutorials

7. Click the ***getCreditReport*** button.

The complete credit report appears in the ***Service Response*** section.

8. Return to WebLogic Workshop, and click the ***Stop*** button (shown below) to close the ***Workshop Test Browser***.



9. Press ***Ctrl+F4*** to close the ***InvestigateSync.jws*** web service.

### Related Topics

#### Using Polling as an Alternative to Callbacks

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



## Step 4: Add Support for XML Mapping

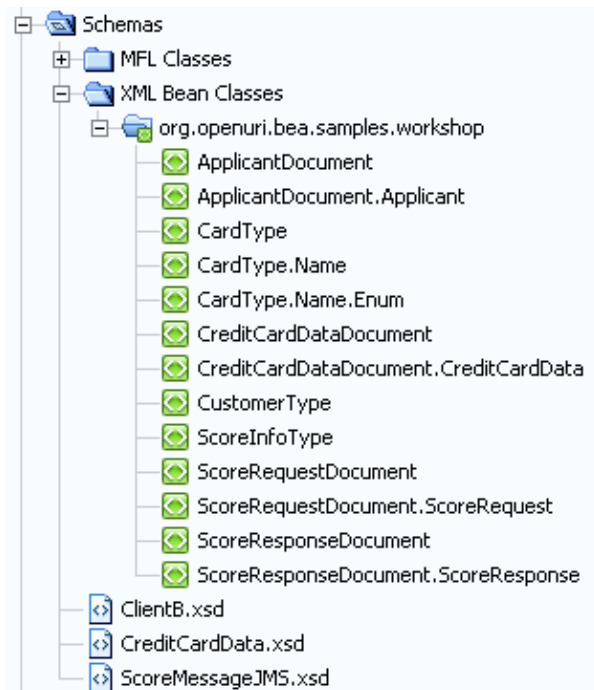
Currently your web services return XML reports of the following form.

```
<applicant>
  <availableCCCredit>2000</availableCCCredit>
  <currentlyBankrupt>false</currentlyBankrupt>
  <lastName>Smith</lastName>
  <firstName>John</firstName>
  <approvalLevel>Applicant is a high risk.</approvalLevel>
  <creditScore>560</creditScore>
  <taxID>22222222</taxID>
</applicant>
```

But what if a client asked you to return an XML report of the following form?

```
<applicant id="22222222">
  <name_last>Smith</name_last>
  <name_first>John</name_first>
  <bankrupt>false</bankrupt>
  <balance_remaining>2000</balance_remaining>
  <risk_estimate>Applicant is a high risk.</risk_estimate>
  <score_info>
    <credit_score>560</credit_score>
    <us_percentile>19.0</us_percentile>
  </score_info>
  <message>Credit Report complete.</message>
</applicant>
```

In this step you will construct a credit report to your client's specifications with the aid of the schema file ClientB.xsd and a set of corresponding XMLBeans classes, ApplicantDocument and ApplicantDocument.Applicant, which are located in the *Schemas* folder.



When a schema file is placed in the Schemas project, WebLogic Workshop produces a set of XMLBean Java classes corresponding to the schema. These XMLBean classes let you construct or parse XML documents that are instances of the original schema. In this case, you will use the XMLBean classes to construct an XML document of the form desired by your client.

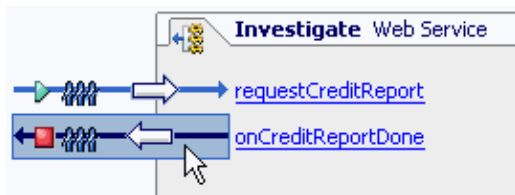
The tasks in this step are:

- to add an XQuery map
- to test the Investigate web service

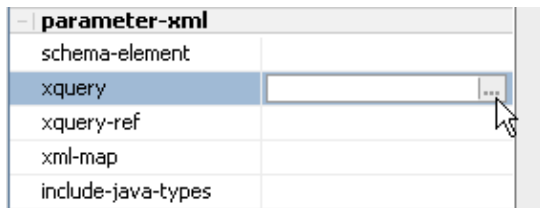
### To Add an XQuery Map

In this task you will place an XQuery map on the callback `onCreditReportDone`. The XQuery map will reshape the XML document currently sent back to clients into an XML document that conforms to the schema `ClientB.xsd`.

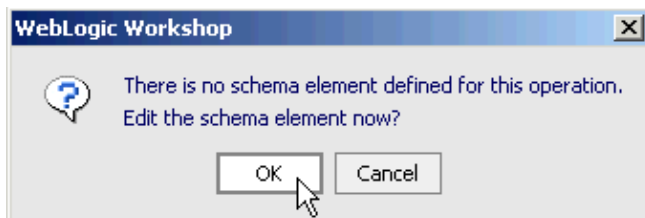
1. From the **Window** menu, select **Investigate.jws**.
2. Click the **Design View** tab.
3. Select the arrow icon for the **onCreditReportDone** callback.



4. On the **Property Editor**, in the section labeled **parameter-xml**, locate the **xquery** property. Click the **ellipses** on the far right-hand side of the value field, as shown in the following illustration.

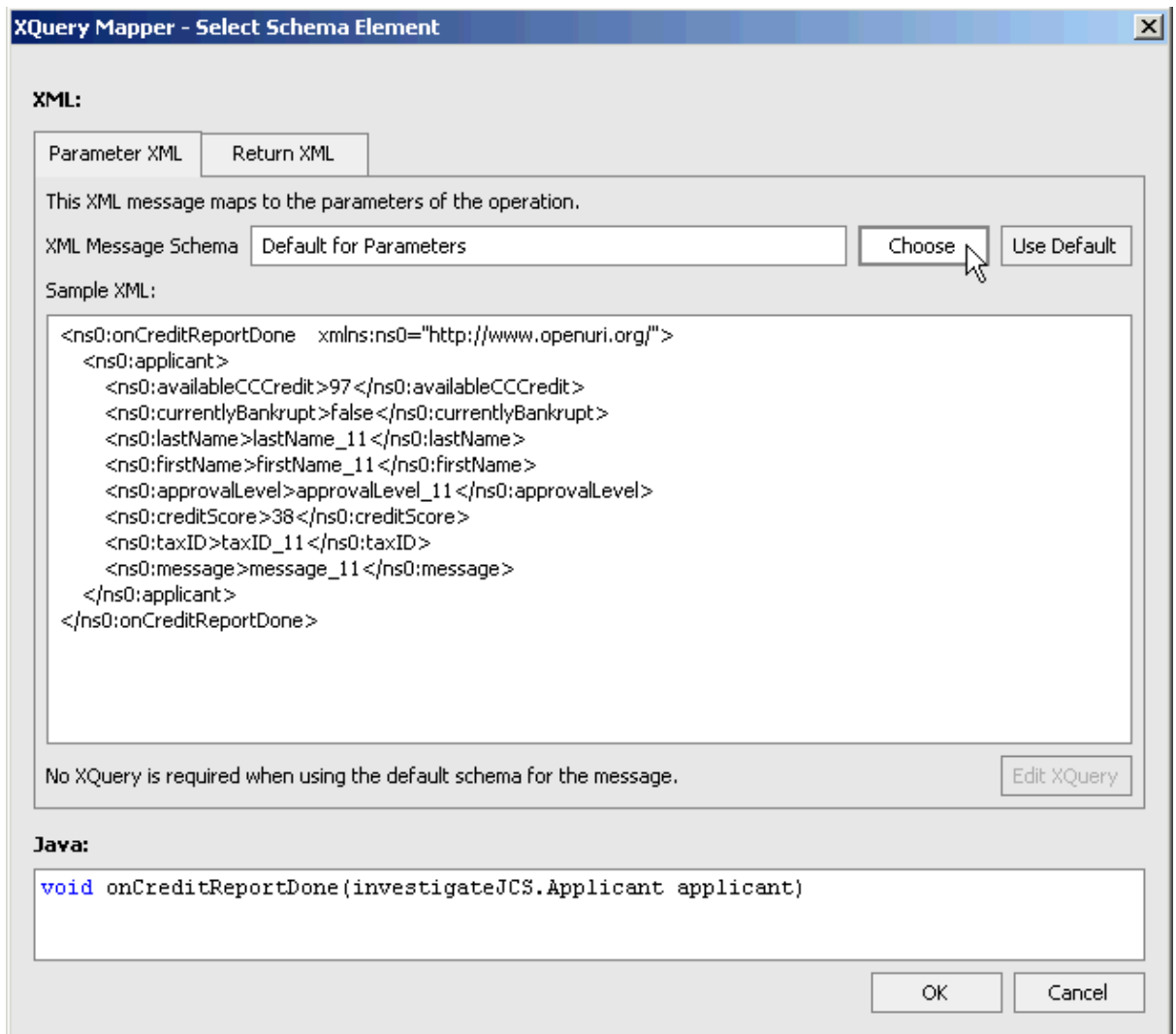


5. Click **Yes**, when asked if you would like to define a schema for this operation.

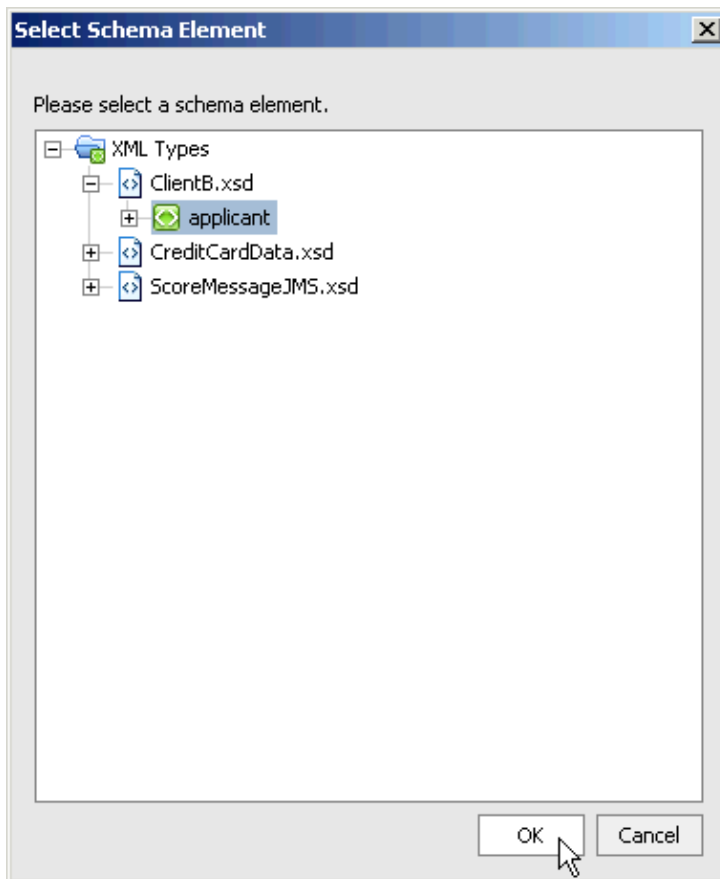


6. In the **XQuery Mapper – Select Schema Element** dialog, click **Choose**.

(Look under **Sample XML**. Note that the method `onCreditReportDone` already has a default schema associated with it. This is the schema that the method currently uses to convert `Applicant` objects into XML messages.)



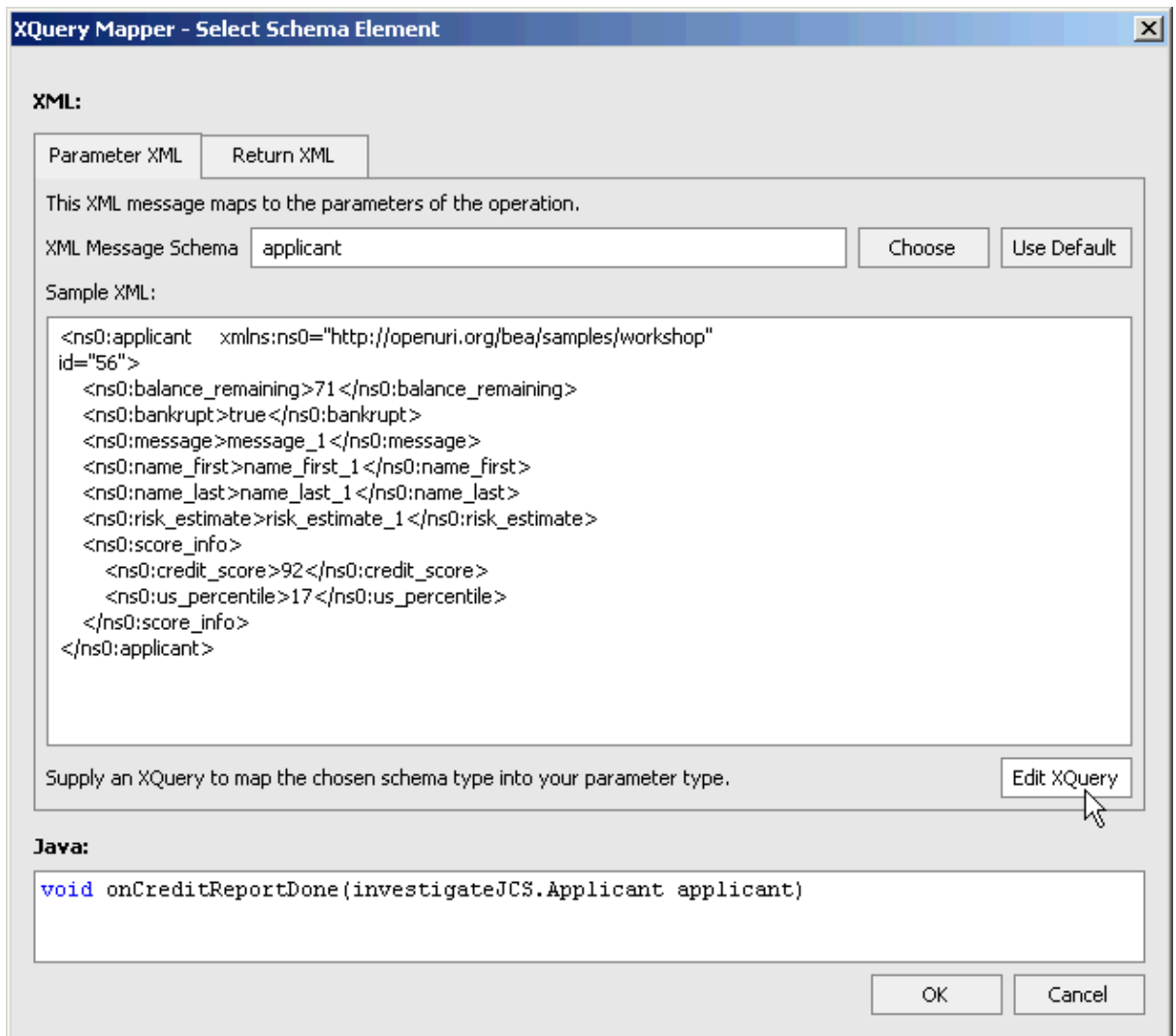
7. In the *Select Schema Element* dialog, select *XML Types*-->*ClientB.xsd*-->*applicant* and click *Ok*.



(In the *XQuery Mapper – Select Schema Element*, look under *Sample XML*. Notice that the new schema, *ClientB.xsd*, is now associated with *onCreditReportDone* method.)

8. In the *XQuery Mapper – Select Schema Element* dialog, click *Edit XQuery*.

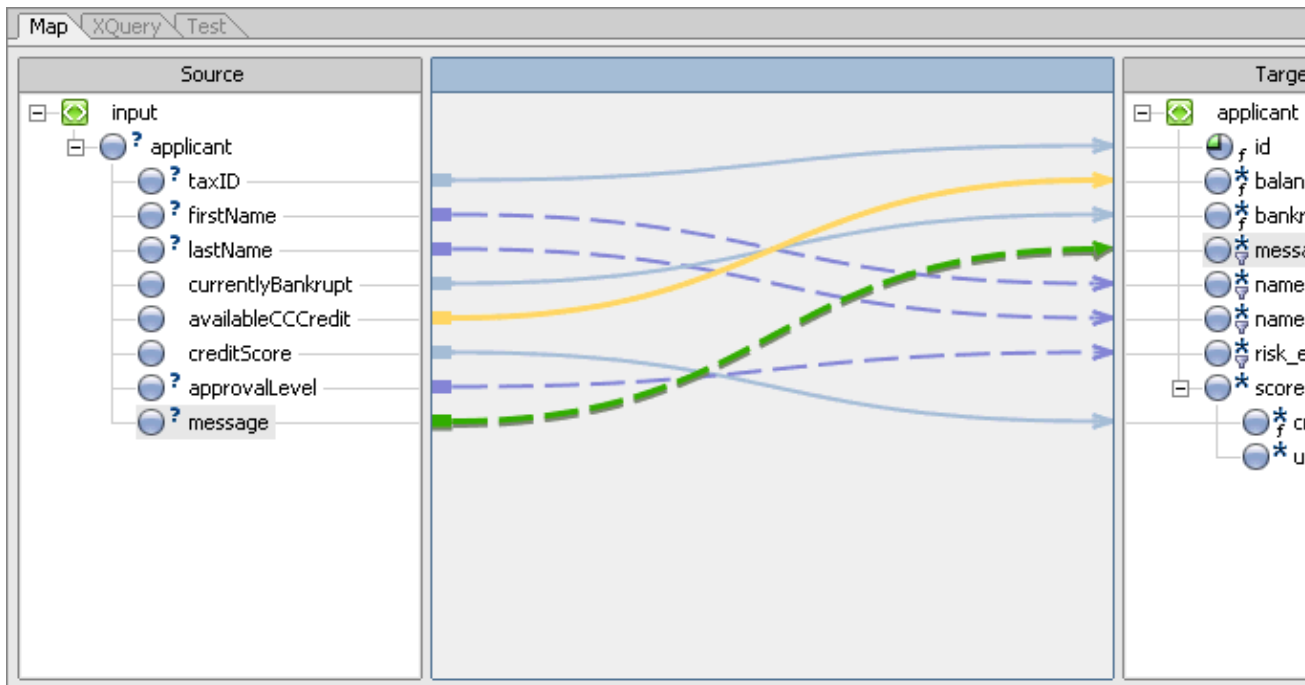




9. In the *XQuery – Edit XQuery* dialog, map the XML elements by dropping and dragging icons from the *Source Schema* pane to the *Target Schema* pane. Form the following matching pairs:

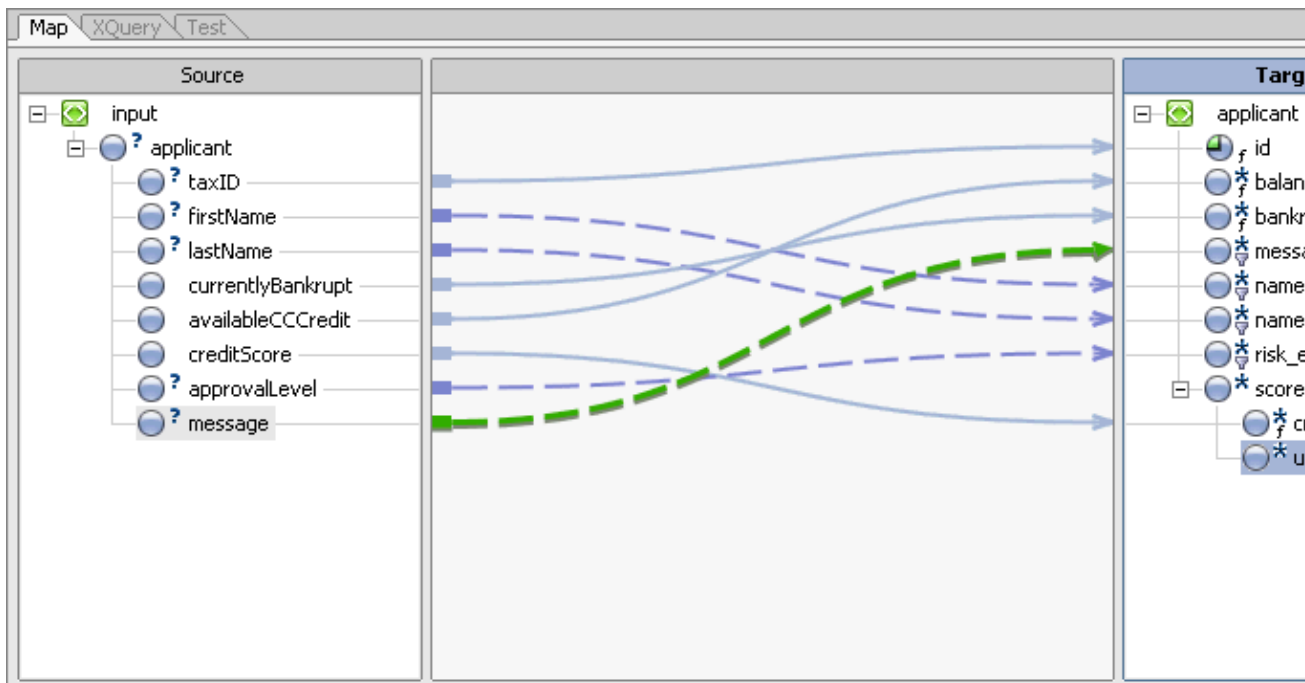
<i>Source Schema</i>	<i>Target Schema</i>
taxID	id
firstname	name_first
lastName	name_last
currentlyBankrupt	bankrupt
availableCCCredit	balance_remaining
creditScore	credit_score
approvalLevel	risk_estimate
message	message

The *Map* tab of The XQuery Mapper – Edit XQuery dialog should like the following illustration.



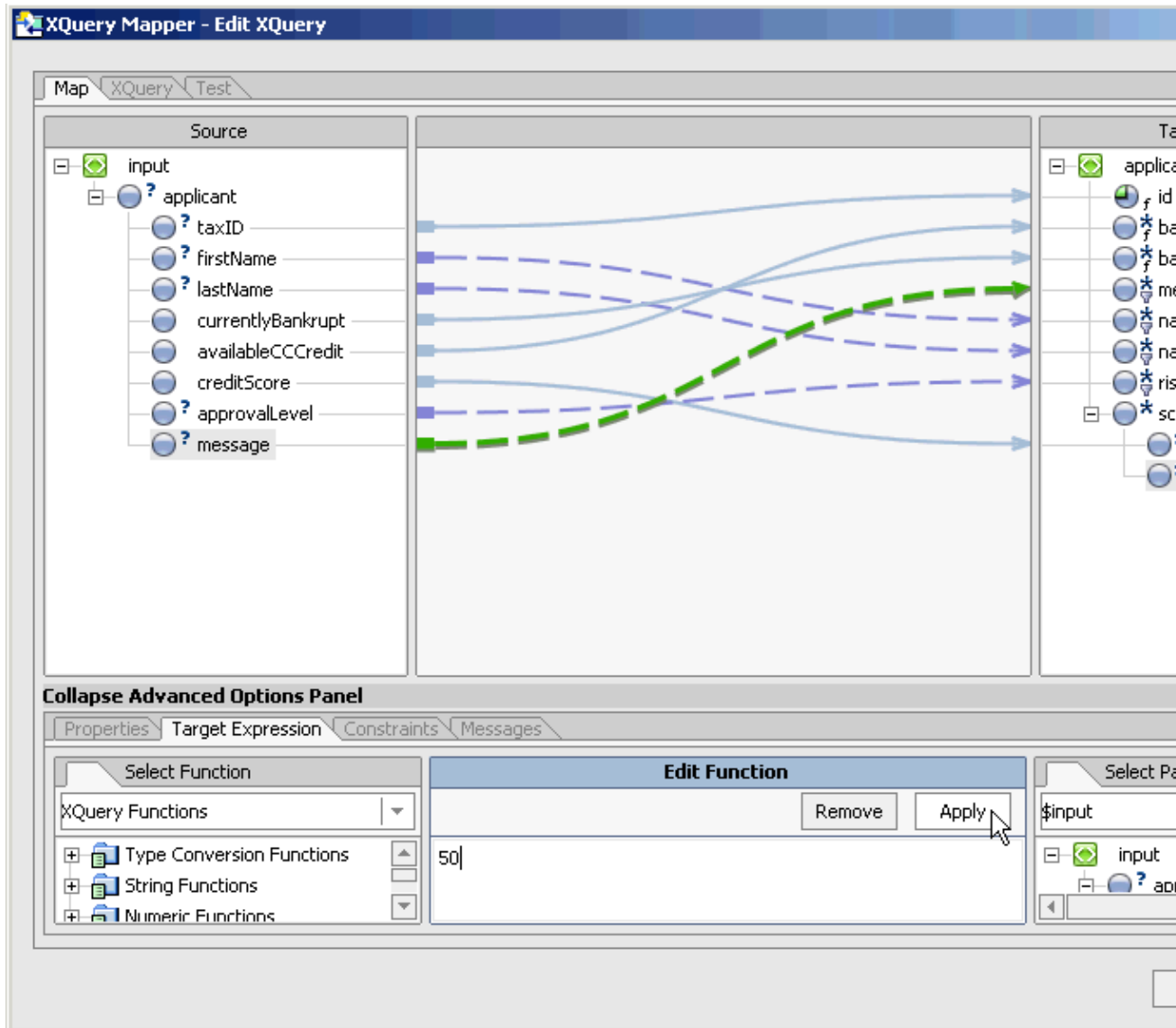
(Solid lines indicate a data conversion function is applied to the mapping. For example the mapping `availableCCCredit`—>`balance_remaining` also includes a data conversion from the `int` data type to the `short` data type. Dotted lines indicate that no data conversion function is applied.)

10. On the *XQuery Mapper – Edit XQuery* dialog, in the *Target Schema* pane, select the *us\_percentile* element.



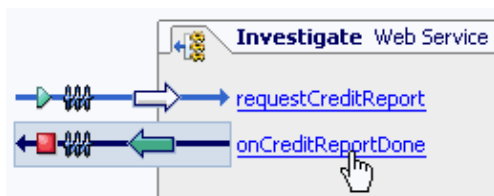
11. In the *Advanced Options Panel*, click the *Target Expression* tab.
12. In the *Edit Function* pane, enter the value 50 and click *Apply*.

(This is a default value. You will add a script to calculate the `<us_percentile>` element in the next step of the tutorial.)



13. Click **Ok** to close the *XQuery Mapper – Edit XQuery* dialog.
14. Click **Ok** to close the *XQuery Mapper – Select Schema Element* dialog.
15. In **Design View**, click the name of the *onCreditReportDone* callback.

(Notice that the arrow icon for the *onCreditReportDone* callback has a green color. This indicates that there is an XQuery map now associated with this callback.)



The callback *onCreditReportDone* now has the following XQuery map associated with it.

```

public interface Callback extends com.bea.control.ServiceControl
{
    /**
     * @common:message-buffer enable="true"
     * @jws:conversation phase="finish"
     * @jws:parameter-xml schema-element="ns0:applicant" xquery::
     * declare namespace ns0 = "http://www.openuri.org/"
     * declare namespace ns1 = "http://openuri.org/bea/samples/workshop"
     *
     * <ns1:applicant id = "{ xs:int( data($input/ns0:applicant/ns0:taxID) ) }">
     *   <ns1:balance_remaining>{ xs:short( data($input/ns0:applicant/ns0:availableCCCredit) ) }
     *   <ns1:bankrupt>{ data($input/ns0:applicant/ns0:currentlyBankrupt) }</ns1:bankrupt>
     *   {
     *     for $message in $input/ns0:applicant/ns0:message
     *     return
     *       <ns1:message>{ data($message) }</ns1:message>
     *   }
     *   {
     *     for $firstName in $input/ns0:applicant/ns0:firstName
     *     return
     *       <ns1:name_first>{ data($firstName) }</ns1:name_first>
     *   }
     *   {
     *     for $lastName in $input/ns0:applicant/ns0:lastName
     *     return
     *       <ns1:name_last>{ data($lastName) }</ns1:name_last>
     *   }
     *   {
     *     for $approvalLevel in $input/ns0:applicant/ns0:approvalLevel
     *     return
     *       <ns1:risk_estimate>{ data($approvalLevel) }</ns1:risk_estimate>
     *   }
     *   <ns1:score_info>
     *     <ns1:credit_score>{ xs:short( data($input/ns0:applicant/ns0:creditScore) ) }</ns1:
     *     <ns1:us_percentile>50</ns1:us_percentile>
     *   </ns1:score_info>
     * </ns1:applicant>
     * ::
     */
    void onCreditReportDone(InvestigateJCS.Applicant applicant);
}

```

In the XQuery map, there are two schemas at work: (1) the default schema automatically created for the Applicant object and (2) the new schema which you are imposing, i.e., ClientB.xsd. When an Applicant object is being prepared as an XML document for transport over the wire, XQuery expressions ( such as `xs:int(data($input/ns0:applicant/ns0:taxID))` ) select values from the default XML schema and insert them into the imposed schema ( such as `<ns1:applicant id="...">` ). The result is an XML document conforming to the imposed schema.

16. Press **Ctrl+S** to save your work.

Your web service's `onCreditReportDone` callback now sends XML messages that conform to the ClientB.xsd schema.

To Test the Investigate Web Service

1. Confirm that *Investigate.jws* is displayed in the main work area.
2. Click the **Start** button, shown below.



3. Your web service compiles and Workshop Test Browser launches.

## WebLogic Workshop Tutorials

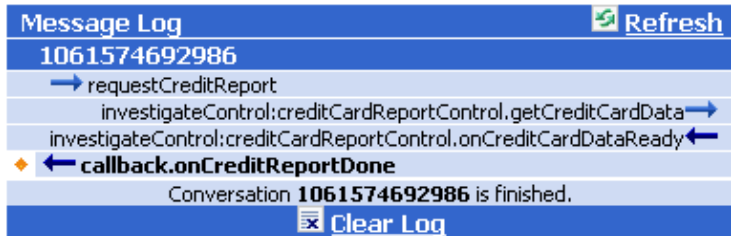
4. In the **taxID** field, enter 222222222 and click the **requestCreditReport** button.

**Note:** Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

5. Click **Refresh** until the **Message Log** shows an entry for **callback.onCreditReportDone**.
6. Click **callback.onCreditReportDone**.

Note that the shape of the XML message conforms to the schema ClientB.xsd.



Message Log Refresh

1061574692986

→ requestCreditReport

investigateControl:creditCardReportControl.getCreditCardData →

investigateControl:creditCardReportControl.onCreditCardDataReady ←

← callback.onCreditReportDone

Conversation 1061574692986 is finished.

Clear Log

### Client Callback

Submitted at Friday, August 22, 2003 10:51:35 AM PDT

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org"
      <conversationID>1061574692986</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <wor:applicant id="222222222" xmlns:wor="http://www.openuri.org">
      <wor:balance_remaining>2000</wor:balance_remaining>
      <wor:bankrupt>false</wor:bankrupt>
      <wor:message></wor:message>
      <wor:name_first>John</wor:name_first>
      <wor:name_last>Smith</wor:name_last>
      <wor:risk_estimate>Applicant is a high risk</wor:risk_estimate>
      <wor:score_info>
        <wor:credit_score>560</wor:credit_score>
        <wor:us_percentile>50</wor:us_percentile>
      </wor:score_info>
    </wor:applicant>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

7. Return to WebLogic Workshop, and click the **Stop** button (shown below) to close the **Workshop Test Browser**.



## Related Topics

### Getting Started with XMLBeans

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



## Step 5: Add a Script for XML Mapping

In the previous step, you added an XML map to your web service, but one part of the map was left unfinished. The `<us_percentile>` element, which gives the applicant's percentile ranking, currently displays a default value of 50% in all cases.

```
<applicant id="22222222">
  <name_last>Smith</name_last>
  <name_first>John</name_first>
  <bankrupt>false</bankrupt>
  <balance_remaining>2000</balance_remaining>
  <risk_estimate>Applicant is a high risk.</risk_estimate>
  <score_info>
    <credit_score>560</credit_score>
    <us_percentile>50</us_percentile>
  </score_info>
  <message>Credit Report complete.</message>
</applicant>
```

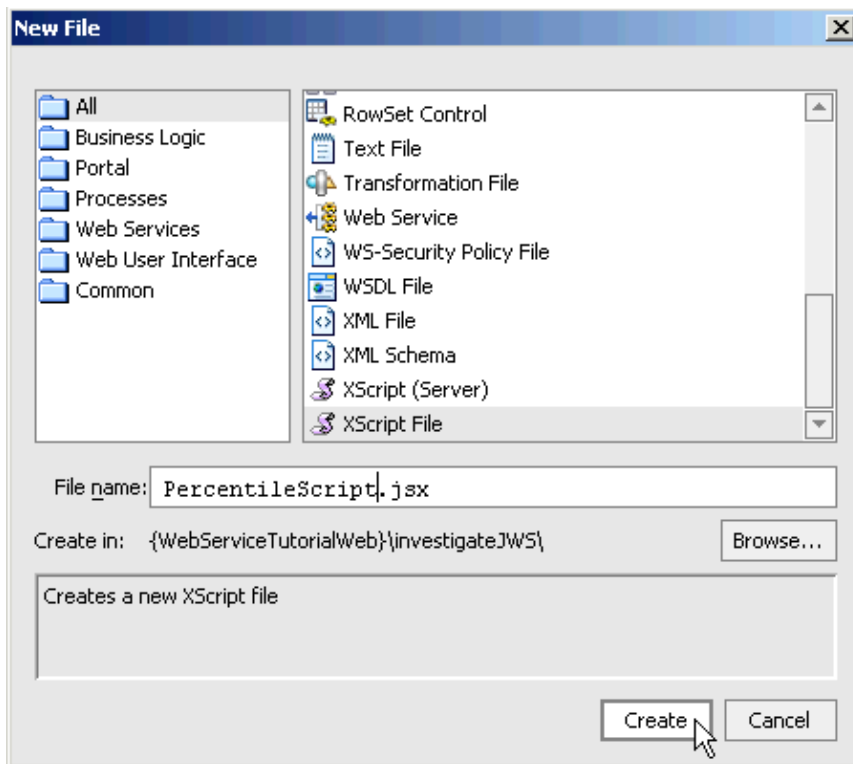
In the following step you will finish the XML map by calculating a value for the `<us_percentile>` element. To do this, you will write a function in ECMAScript (also known as JavaScript), and then refer to the function from the XML map.

The tasks in this step are:

- To add a script file for use with mapping
- To edit the XQuery map
- To test the Investigate Web Service

To Add a Script File for Use with Mapping

1. Right-click the *investigateJWS* folder and select *New-->Other File Types*. The *New File* dialog opens.
2. Enter values as shown in the following illustration:



3. Click **Create**. This adds a JSX file called PercentileScript.jsx. WebLogic Workshop displays the new file in Source View.
4. Paste the following code into **PercentileScript.jsx**. Overwrite all of the code already in the file.

```

/* A function to calculate the applicant's credit score percentile. */
function calcPercentile (score) {
    var percentile = 0;
    var scoreLowEnd = 300;
    var number = 3162.5;
    percentile = Math.ceil((Math.pow(score, 2) - Math.pow(scoreLowEnd, 2))/(2 * number));
    return percentile;
}

```

5. Press **Ctrl+S** to save your work. Press **Ctrl+F4** to close PercentileScript.jsx.

#### To Edit the XQuery Map to Call Script

In this task you will edit the XQuery map to use PercentileScript.jsx to fill in the value of the `<us_percentile>` element.

1. If Investigate.jws is not displayed, from the **Window** menu, select **Investigate.jws**. Click the **Source View** tab.
2. Edit the XQuery map to look like the following. Code to edit appears in red.

```

public interface Callback extends com.bea.control.ServiceControl
{
    /**
     * @common:message-buffer enable="true"
     * @jws:conversation phase="finish"
     * @jws:parameter-xml schema-element="ns0:applicant" xquery::
     * declare namespace ns0 = "http://www.openuri.org/"
     * declare namespace ns1 = "http://openuri.org/bea/samples/workshop"
     *

```


## WebLogic Workshop Tutorials

```
* <ns1:applicant id = "{ xs:int( data($input/ns0:applicant/ns0:taxID) ) }">
*   <ns1:balance_remaining>{ xs:short( data($input/ns0:applicant/ns0:availableBalance) ) }
*   <ns1:bankrupt>{ data($input/ns0:applicant/ns0:currentlyBankrupt) }</ns1:bankrupt>
*   {
*     for $message in $input/ns0:applicant/ns0:message
*     return
*       <ns1:message>{ data($message) }</ns1:message>
*   }
*   {
*     for $firstName in $input/ns0:applicant/ns0:firstName
*     return
*       <ns1:name_first>{ data($firstName) }</ns1:name_first>
*   }
*   {
*     for $lastName in $input/ns0:applicant/ns0:lastName
*     return
*       <ns1:name_last>{ data($lastName) }</ns1:name_last>
*   }
*   {
*     for $approvalLevel in $input/ns0:applicant/ns0:approvalLevel
*     return
*       <ns1:risk_estimate>{ data($approvalLevel) }</ns1:risk_estimate>
*   }
*   <ns1:score_info>
*     <ns1:credit_score>{ xs:short( data($input/ns0:applicant/ns0:creditScore) ) }
*     <ns1:us_percentile> {investigateJWS.PercentileScript.calcPercentile(data($input/ns0:applicant/ns0:creditScore))}
*   </ns1:score_info>
* </ns1:applicant>
* ::
*/
void onCreditReportDone(investigateJCS.Applicant applicant);
}
```

Note that the value of the <us\_percentile> element is based on the value of the <credit\_score> element: when the calcPercentile() function is called, the value of the <credit\_score> element is passed in as a parameter.

3. Press **Ctrl+S** to save your work.

### To Test the Investigate Web Service

1. Confirm that **Investigate.jws** is open and displayed in the main work area.
2. Press **Ctrl+F5** run Investigate.jws. (Pressing Ctrl+F5 is equivalent to clicking the Start button: )

Your web service compiles and Workshop Test Browser launches.

3. In the **Workshop Test Browser**, in the **taxID** field enter the 9 digit number 222222222.
- Note:** Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click the **requestCreditReport** button.
5. Click **Refresh** until the **Message Log** shows an entry for **callback.onCreditReportDone**.

Note the value of the <us\_percentile> element.



### Client Callback

Submitted at Wednesday, August 20, 2003 9:23:50 AM PDT

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1061396627667</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <wor:applicant id="22222222" xmlns:wor="http://openuri.org/bea/samples/workshop">
      <wor:balance_remaining>2000</wor:balance_remaining>
      <wor:bankrupt>false</wor:bankrupt>
      <wor:message></wor:message>
      <wor:name_first>John</wor:name_first>
      <wor:name_last>Smith</wor:name_last>
      <wor:risk_estimate>Applicant is a high risk.</wor:risk_estimate>
      <wor:score_info>
        <wor:credit_score>560</wor:credit_score>
        <wor:us_percentile>36.0</wor:us_percentile>
      </wor:score_info>
    </wor:applicant>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

### Related Topics

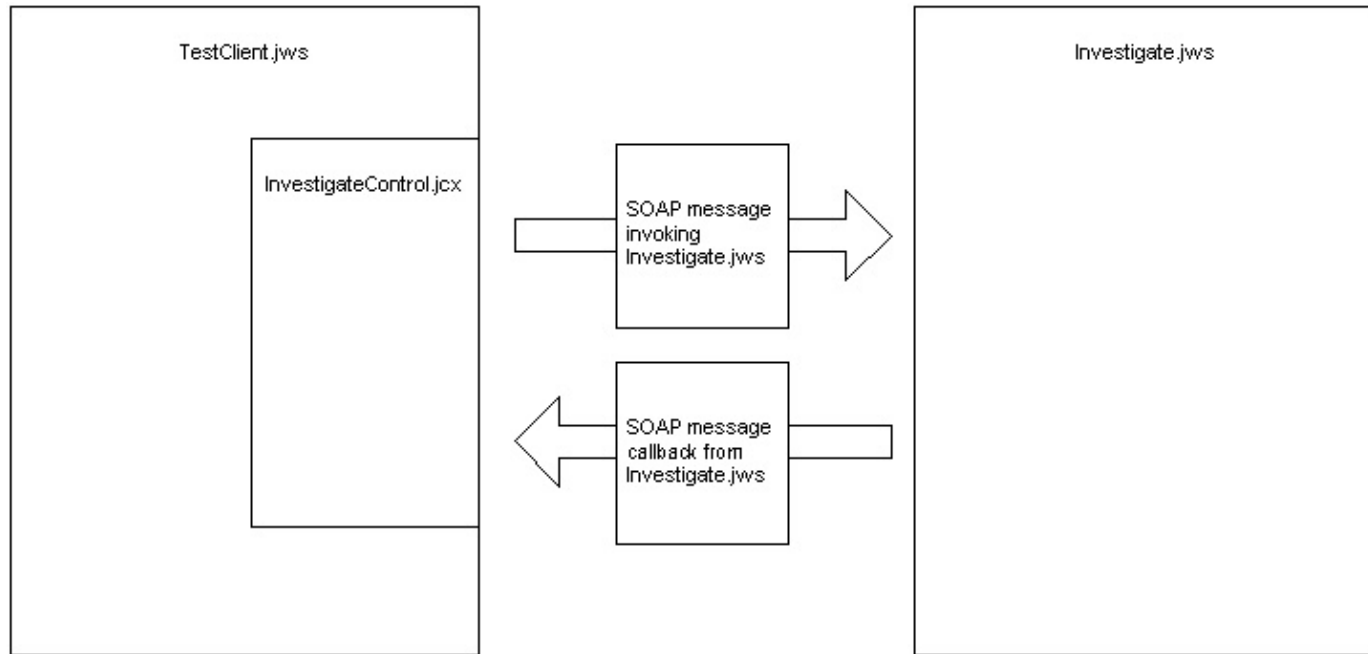
#### Handling XML with ECMAScript Extensions

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

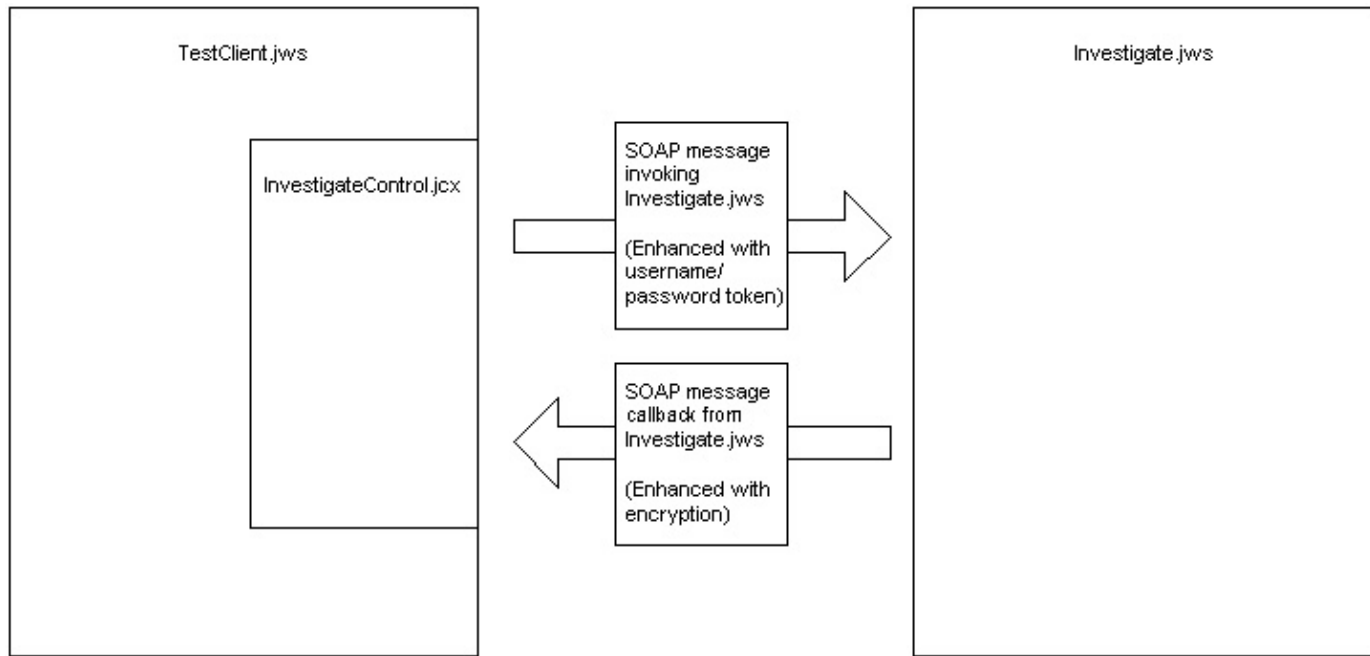


## Step 6: Web Service Security

In this task you will use Web Service Security, sometimes referred to with the acronyms "WS–Security" or simply "WSSE", to secure the communications between the Investigate Web Service and a client web service TestClient.jws. (TestClient.jws is located in the folder WebServiceTutorialWeb/WSSETestClient/.) The two web services communicate through SOAP messages, as shown in the following diagram.



The communication will be secured by enhancing the SOAP messages with username/password tokens and encryption, as shown in the following diagram. TestClient's requests for a credit report will include a username and password, allowing the Investigate web service can authenticate TestClient. The credit report returned by Investigate will be encrypted before it is sent out over the wire, ensuring that the credit report is confidential.



These security enhancements are specified in a web service security policy file, or "WSSE file". WSSE files are XML files that specify the security behavior of your web service or web service control.

**Note:** that WSSE security is not the only option for securing a web service. SSL and Basic Authentication are other important options; for an example, see Java Control Tutorial Step 7: Transport Security.

The tasks in this step are:

- To associate the WSSE policy file with Investigate web service
- To associate a WSSE policy file with the Investigate control file
- To Test the Investigate web service

### To Associate a WSSE Policy with the Investigate Web Service

In this task you will apply a web service security policy file (WSSE file) to the Investigate web service. This policy file specifies how incoming and outgoing SOAP message should be treated.

Before you begin this task, on the **Application** tab, double-click the file **WebServiceTutorialWeb/investigateJWS/Investigate.wsse** to view the file.

```
Investigate.wsse - {WebServiceTutorial\Web}\InvestigateJWS\
<?xml version="1.0" ?>
<wsSecurityPolicy xsi:schemaLocation="WSSecurity-policy.xsd"
  xmlns="http://www.bea.com/2003/03/wsse/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!--
  Incoming SOAP messages to Investigate.jws must
  be accompanied by a username/password token.
  -->
  <wsSecurityIn>
    <token tokenType="username"/>
  </wsSecurityIn>

  <!--
  Encrypt the completed credit reports with the recipient's
  public key for their trip across the wire.
  Note that the encrypted SOAP message can only be decrypted
  by the recipient's matching private key.
  -->
  <wsSecurityOut>
    <encryption>
      <encryptionKey>
        <alias>client1</alias>
      </encryptionKey>
    </encryption>
  </wsSecurityOut>

  <keyStore>
    <keyStoreLocation>samples_mycompany.jks</keyStoreLocation>
    <keyStorePassword>password</keyStorePassword>
  </keyStore>
</wsSecurityPolicy>
```

WSSE files contain two main elements. (1) The <wsSecurityIn> element specifies the kinds of security requirements that must be met by incoming SOAP messages. (2) The <wsSecurityOut> element specifies the sorts of security enhancements that should be added to outgoing SOAP messages.

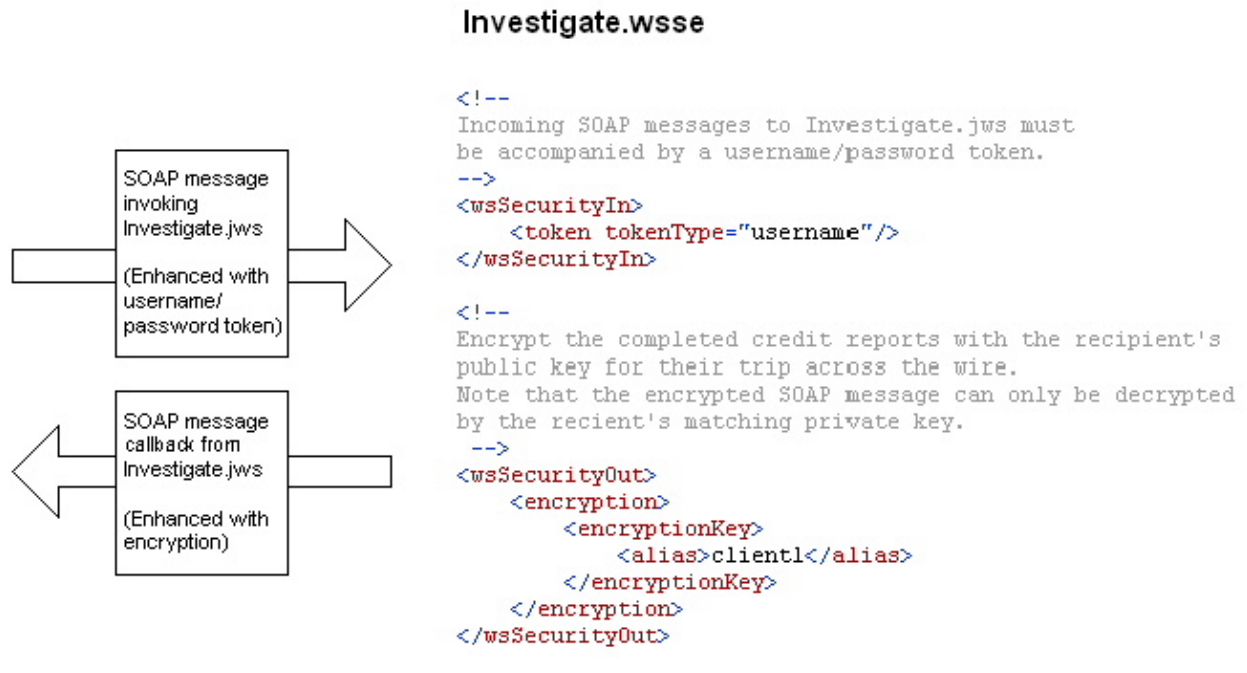
The <wsSecurityIn> specifies that incoming SOAP messages must be accompanied by a valid username and password pair. If a SOAP message is sent to Investigate.jws without a valid username/password pair, then the SOAP message will be rejected by the web service and go unprocessed. The reason for setting up this security requirement is that it is important to know who is asking for credit reports: the Investigate web service doesn't want to be sending out credit reports to anonymous users. Note that this security strategy requires a registration mechanism to be fully implemented. This registration mechanism is outside the scope of this tutorial. But it would go something like this: candidate users would be required to provide some information about themselves: an email, a phonenumber, etc., and, upon verification of this information, the user would be given a username/password with which to login to the Investigate web service. For testing purposes we will use a test user already known to the WebLogic Security framework.

The <wsSecurityOut> element specifies that outgoing SOAP messages are to be encrypted before they are sent out over the wire. This will ensure that the credit report will remain confidential and can only be read by the intended recipient.

The <keyStore> element points to a Java KeyStore file (JKS file). This file contains the message recipient's public key with which the SOAP messages are encrypted. When data is encrypted in this way, only the matching private key, known only to the message recipient, can successfully decrypt the message, ensuring

confidentiality of communication.

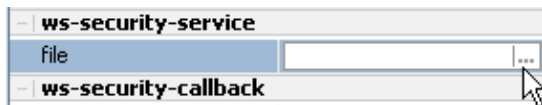
The follow diagram shows how Investigate.wsse controls the SOAP traffic coming into and out of Investigate.jws.



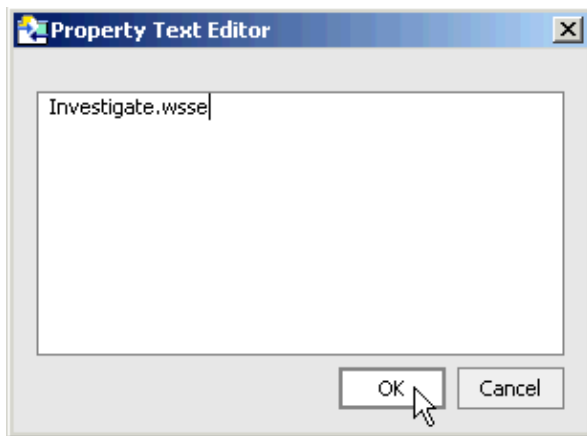
1. From the **Window** menu select *Investigate.jws*.
2. Click the **Design View** tab.
3. In **Design View**, select the picture of the Investigate web service. You can select the web service by clicking the picture's title bar, as shown below.



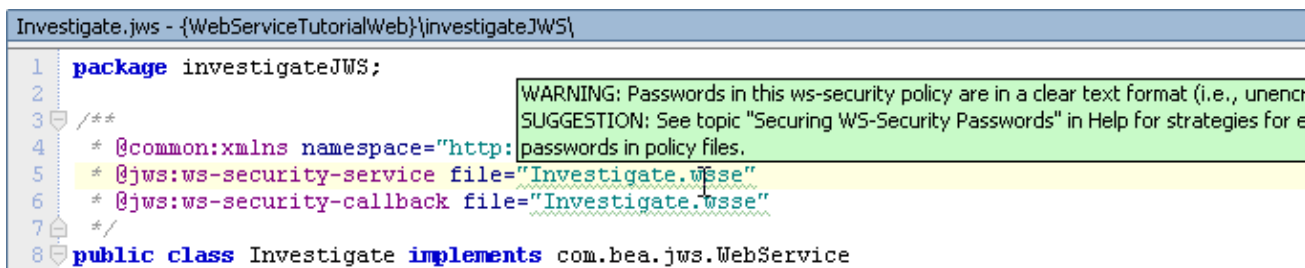
4. On the **Property Editor** tab, in the section labeled **ws-security-service**, on the right side of the *file* property. Click the *ellipses* button.



5. In the **Property Text Editor** dialog, enter *Investigate.wsse*, and click *Ok*.



6. On the **Property Editor** tab, in the section labeled **ws-security-callback**, on the right side of the **file** property, click the **ellipses** button.
7. In the **Property Text Editor** dialog, enter **Investigate.wsse**, and click **Ok**.
8. Press **Ctrl+S** to save your work.
9. Click the **Source View** tab.
10. Hover the mouse pointer over the green wavy line underneath "`@jws:ws-security-service` file=`Investigate.wsse`"



Notice the tool tip that appears. This tool tip is warning you that your WSSE files are not encrypted. When you deploy a WSSE file to a production server, the passwords within the WSSE files appear in plain text. By leaving the passwords in plain text, you run the risk of allowing anyone with local access to the production server to view those passwords. To mitigate this risk, WebLogic Workshop has provided utilities to encrypt the passwords within WSSE files. (For detailed information on these utilities see the Help topic *Securing WS-Security Passwords*.) Note that it is not possible to successfully encrypt and run a WSSE file within a development environment, so we will ignore these warnings.

#### To Associate a WSSE Policy with the Investigate Web Service Control

In this task you will place a security policy on *InvestigateControl.jcx*, not on *TestClient.jws*. The reason you place a policy on the control file is because you are primarily interested in changing the way that *TestClient.jws* communicates with the particular web service *Investigate.jws*, but you are not interested in changing the way *TestClient.jws* communicates with any and all web services. This is a typical pattern in WSSE: to secure the communication between a client web service and its target web service, you place security policies on the target web service and on the *target web service control file* used by the client web service, but you leave the client web service without a policy of its own.

Before you complete this task double-click the file *WebServiceTutorialWeb/WSSETestClient/InvestigateControl.wsse* to view the file.

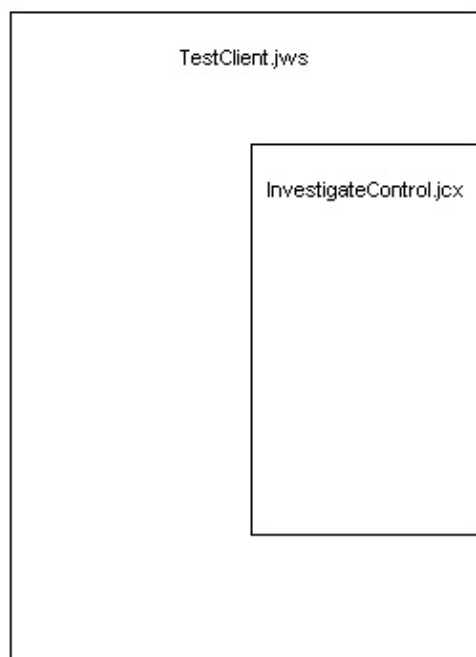
```
InvestigateControl.wsse - {WebServiceTutorial\Web}\WSETestClient\
<?xml version="1.0" ?>
<wsSecurityPolicy xsi:schemaLocation="WSSecurity-policy.xsd"
  xmlns="http://www.bea.com/2003/03/wsse/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!--
  SOAP messages sent to the Investigate web service
  are enhanced with a username/password token.
  -->
  <wsSecurityOut>
    <userNameToken>
      <userName>weblogic</userName>
      <password type="TEXT">weblogic</password>
    </userNameToken>
  </wsSecurityOut>

  <!--
  Incoming SOAP messages must be encrypted.
  Decrypt the SOAP messages with the recipient's private key.
  -->
  <wsSecurityIn>
    <encryptionRequired>
      <decryptionKey>
        <alias>client1</alias>
        <password>password</password>
      </decryptionKey>
    </encryptionRequired>
  </wsSecurityIn>

  <keyStore>
    <keyStoreLocation>samples_client.jks</keyStoreLocation>
    <keyStorePassword>password</keyStorePassword>
  </keyStore>
</wsSecurityPolicy>
```

Notice that InvestigateControl.wsse has a security policy to complement Investigate.wsse. Incoming SOAP messages must be encrypted; and outgoing SOAP messages are enhanced with a username and password before they are sent out over the wire. The following diagram shows how InvestigateControl.wsse controls the SOAP traffic coming into and out of the control file InvestigateControl.jcx.



## InvestigateControl.wsse

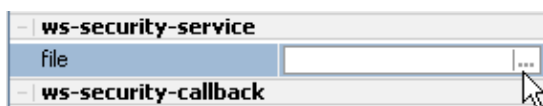
```

<!--
SOAP messages sent to the Investigate web service
are enhanced with a username/password token.
-->
<wsSecurityOut>
  <userNameToken>
    <userName>weblogic</userName>
    <password type="TEXT">weblogic</password>
  </userNameToken>
</wsSecurityOut>

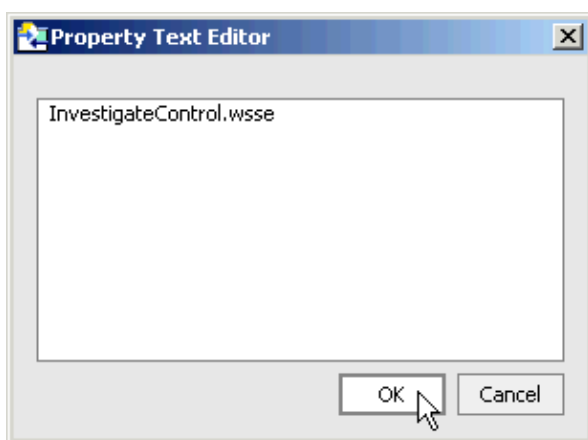
<!--
Incoming SOAP messages must be encrypted.
Decrypt the SOAP messages with the recipient's private key.
-->
<wsSecurityIn>
  <encryptionRequired>
    <decryptionKey>
      <alias>client1</alias>
      <password>password</password>
    </decryptionKey>
  </encryptionRequired>
</wsSecurityIn>

```

1. On the **Application** tab, double-click the control file *WebServiceTutorial/WebServiceTutorialWeb/WSSETestClient/InvestigateControl.jcx*.
2. If InvestigateControl.jcx is not displayed in Design View, click the **Design View** tab.
3. On the **Property Editor** tab, in the section labeled **ws-security-service**, on the right side of the **file** property, click the **ellipses** button.



4. In the **Property Text Editor** dialog, enter *InvestigateControl.wsse*, and click **Ok**.



5. On the **Property Editor** tab, in the section labeled **ws-security-callback**, on the right side of the **file** property, click the **ellipses** button.
6. In the **Property Text Editor** dialog, enter *InvestigateControl.wsse*, and click **Ok**.
7. Press **Ctrl+S** to save your work. Press **Ctrl+F4** to close the InvestigateControl.jcx file.

To Test the Investigate Web Service



## WebLogic Workshop Tutorials

In this task you will perform two tests. First you will test to see if the right passwords result in successful communication, then you will test to see if the wrong passwords result in failed communication.

### *Testing for Successful Communication*

1. On the **Application** tab, double-click **TestClient.jws**.
2. Press **Ctrl+F5** run TestClient.jws.

Your web service compiles and the Workshop Test Browser launches.

3. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 222222222.

**Note:** Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

4. Click the **requestCreditReport** button.
5. Click **Refresh** until the **Message Log** shows an entry for **callback.onCreditReportDone**.

### *Testing for Failed Communication*

1. On the **Application** tab, double-click **InvestigateControl.wsse**.
2. Edit the the <wsSecurityOut> element to look like the following. Edit the code appearing in red. Note that you are intentionally entering the wrong password.

```
<wsSecurityOut>
  <userNameToken>
    <userName>weblogic</userName>
    <password type="TEXT">wrong_password</password>
  </userNameToken>
</wsSecurityOut>
```

3. On the **Application** tab, double-click **TestClient.jws**.
4. Press **Ctrl+F5** to run TestClient.jws
5. In the **Workshop Test Browser**, in the **taxID** field, enter the 9 digit number 222222222.
6. Click the **requestCreditReport** button.

Note that the invocation of the Investigate web service fails with the following exception.

#### **Exception**

```
Submitted at Friday, August 22, 2003 1:57:41 PM PDT
Exception in requestCreditReport
com.bea.control.ServiceControlException: SERVICE FAULT:
Code:com.bea.wlw.runtime.jws.wssecurity.exception.WLWWSSEEException
String:java.security.UnrecoverableKeyException: Cannot recover key
Detail:
END SERVICE FAULT
at com.bea.wlw.runtime.core.control.ServiceControlImpl.invoke(ServiceControlImpl.jcs:896)
at com.bea.wlw.runtime.core.dispatcher.DispMethod.invoke(DispMethod.java:373)
at com.bea.wlw.runtime.core.container.Invocable.invoke(Invocable.java:420)
at com.bea.wlw.runtime.core.container.Invocable.invoke(Invocable.java:393)
at com.bea.wlw.runtime.jcs.container.JcsProxy.invoke(JcsProxy.java:390)
at $Proxy36.requestCreditReport(Unknown Source)
at WSESTestClient.TestClient.requestCreditReport(TestClient.jws:22)
```

7. Remember to re-edit the InvestigateControl.wsse file so that it sends the correct password.

### Related Topics

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



# Summary: Web Service Tutorial

This topic lists ideas this tutorial introduced, along with links to topics for more information. You may also find it useful to look at the following:

- For an overview of WebLogic Workshop, see [Building Web Services with WebLogic Workshop](#).
- For a links to information on tasks you can accomplish in WebLogic Workshop, see the [How Do I?](#) topics.
- For a list of the samples provided, see [Samples](#).

## Concepts and Tasks Introduced in This Tutorial

- You interact with web service source files through a WebLogic Workshop project. You use projects to group files associated with a web service or related web services.  
For more information about projects, see [Applications and Projects](#).
- As you develop web services, you test and debug code on a running instance of WebLogic Server.
- You design a web service as you might make a drawing of it and its relationships with clients and other resources. WebLogic Workshop provides a Design View you can use to generate code to start with as you create your design.
- The source file for a web service is a JWS file, which is automatically recognized as a web service when deployed with WebLogic Server.
- You expose the functionality of a web service by creating methods within a JWS file. You can add the method in Design View, setting properties to specify its characteristics, including powerful server features.
- To test a service, you use the Test View, a dynamically generated HTML page through which you can invoke service methods with specific parameter values.
- You can use asynchronous communications to handle latency issues inherent on the Internet, and sometimes with web services in general. Through asynchrony, you avoid situations in which client software is blocked from proceeding until it receives the results of its requests.  
For an introduction to asynchrony and the WebLogic Workshop tools that support it, see [Designing Asynchronous Interfaces](#).
- Callbacks provide a useful way to return results to clients where asynchronous communication is needed. Callbacks require an agreement that the client will implement a means to handle the callback a service makes.

For more details, see [Using Callbacks to Notify Clients of Events](#).

- You can use conversations to maintain consistent state even between disparate asynchronous exchanges. Conversations make it possible to correlate these exchanges with the original request and the client that made it.  
For more information, see [Designing Conversational Web Services](#) and [Getting Started: Using Asynchrony to Enable Long-Running Operations](#).
- When you need to handle or control the specific shape of XML messages your service exchanges with other components, you can use Workshop's extensive XML tools such as XQuery and XMLBeans.

For more on Workshop's XML tools, see [Getting Started with XMLBeans](#) and [Transforming XML Messages with XQuery Maps](#).

- By supporting polling, your web service can offer an alternative to clients that aren't capable of receiving callbacks. With polling, a client can periodically ask your web service if the response to its request is ready yet.

## WebLogic Workshop Tutorials

For information about polling, see [Using Polling as an Alternative to Callbacks](#).

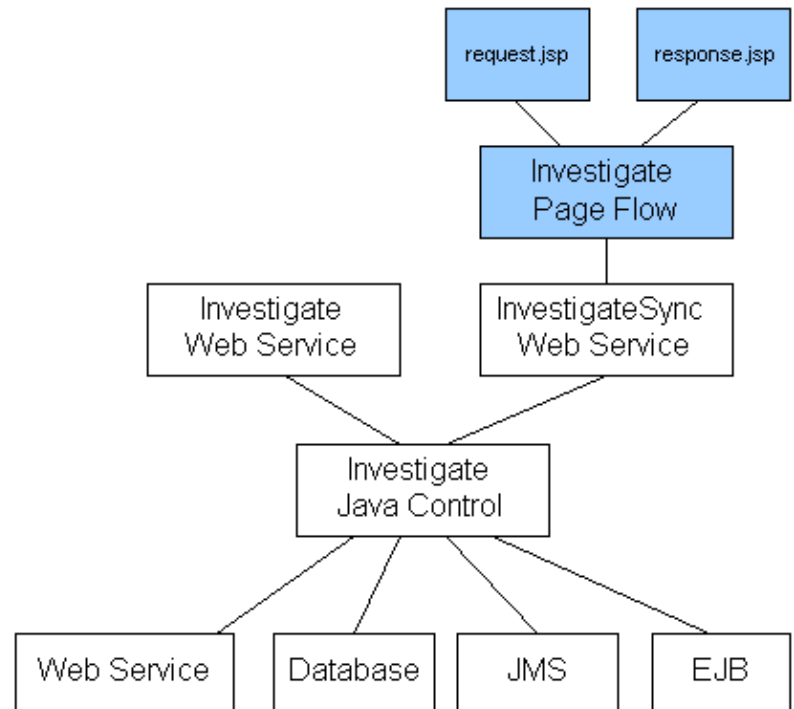
- You can make your web service secure with WS–Security or more traditional transport security. For more detail, see [Web Service Security and Transport Security](#).

# Tutorial: Page Flow

The following tutorial demonstrates advanced capabilities of Page Flows. For an introductory tutorial see Getting Started: Web Applications.

## The Big Picture

The page flow tutorial is the last in a series of three tutorials, which builds the Investigate Application, an application designed to assemble a credit-worthiness report on a loan applicant. (**Note:** you do **not** need to execute the tutorial series in order. You may proceed with this tutorial, even if you have not completed the previous tutorials in the series.)



The first tutorial in the series, the Tutorial: Java Control, builds the core of the application: the Investigate Java control, which consults a variety of components to assemble the credit-worthiness report.

The second tutorial in the series, the Tutorial: Web Service, builds the web services tier of the application.

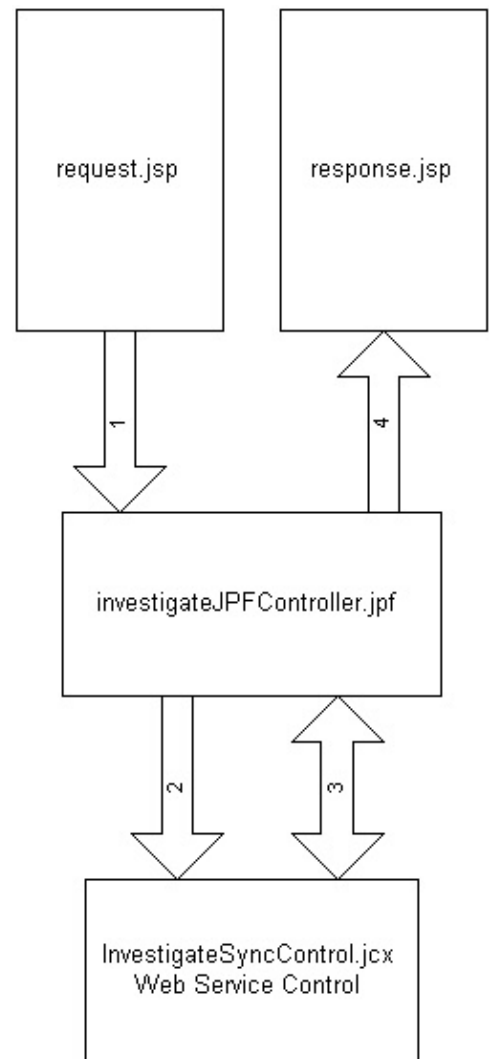
This, the third tutorial in the series, builds the Investigate Web Application, a web-based user interface that lets users request and receive credit reports through a web browser. The components built are shown in blue in the diagram to the right.

## Tutorial Goals

In this tutorial, you will learn:

- How to create a web application's user interface with JSPs
- How to access a web application's back-end data resources with Java controls
- How to coordinate navigation, user data input, and back-end data resources with page flow files
- How to use page flow technology to separate data processing and data presentation
- How to secure a web application with role-based security.

## Tutorial Overview



In this tutorial you will build a web-based user interface for the Investigate credit reporting web service, a service that provides credit profiles on loan applicants. As you progress in the tutorial, you will learn how to access data with web service controls, how to present data with JSPs, and how to control user navigation, user requests, and back-end resources with page flows.

The user interface you build consists of main three main layers (see the diagram to the right):

1. The user-facing, presentation layer consists of two JSP pages: ***request.jsp*** and ***response.jsp***. Users request and view credit reports through these pages.
2. The middle, coordinating layer consists of Java page flow file: ***investigateJPFCController.jspf***. This file controls user navigation between JSP pages, how user requests are handled, and how credit reports are retrieved and displayed to the user.
3. The back-end, data resource layer consists of a web service control file: ***InvestigateSyncControl.jcx***. This file provides access to the Investigate web service, which, along with the Investigate Java control, assembles the credit reports.

The numbered arrows represent a typical request-response cycle. (1) Users submit a request for a credit report to the page flow file *investigateJPFCController.jspf*. (2) This request is passed on to the web service control *InvestigateSyncControl.jcx*. (3) After the web service control has had time to assemble the credit report, it is retrieved and stored in the page flow file *investigateJPFCController.jspf*. (4) Finally, the credit report is

displayed to the user on the response.jsp page.

### Steps in This Tutorial

#### Step 1: Begin the Investigate Page Flow

In this step you will deploy the back-end resources that assemble credit reports.

#### Step 2: Processing Data with Page Flows

In this step you will (1) create a page flow file (JPF file), which controls user requests and retrieval of credit reports, and (2) create a web service control, which lets you access the resources that provides the credit reports.

#### Step 3: Create the JSP pages

In this step you will create the JSP pages that let users request and view credit reports.

#### Step 4: Role-Based Security

In this step you will set role restrictions on the Investigate page flow.

#### Summary: Page Flow Tutorial

#### Related Topics

#### Getting Started with Page Flows

Click the arrow to navigate to the next step in the tutorial.



# Step 1: Begin the Page Flow Tutorial

In this step you will create the Investigate page flow. A page flow is a Java class (with the JPF file extension) that controls how your web application functions and what it does. Page flows control all of the major features of a web application: how users navigate from page to page, user requests, and access to the web application's back-end resources.

Page flows control the features of a web application through the use of Action methods. An Action method may do something simple, such as forward a user from one JSP page to another; or it may do something complex, such as receive user input from a JSP page, calculate and/or retrieve other data based on the user input, and forward the user to a JSP page where the results are displayed.

The page flow you create in this step contains one simple Action method. This simple navigational Action method forwards users to the index.jsp page. In the next step, you will create a more complex Action method.

The tasks in this step are:

- To start WebLogic Workshop
- To create a new application
- To start WebLogic Server
- To create a new page flow
- To test the page flow

To Start WebLogic Workshop

... on Microsoft Windows

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

1. From the **Start** menu, choose **Programs**—>**WebLogic Platform 8.1**—>**QuickStart**.
2. In the **QuickStart** dialog, click **Experience WebLogic Workshop 8.1**.

...on Linux

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

1. Open a file system browser or shell.
2. Locate the **Workshop.sh** file at the following address:  
`$HOME/bea/weblogic81/workshop/Workshop.sh`
3. In the command line, type the following command:  
`sh Workshop.sh`

Create a New Application

The application you create in this task contains resources for the page flow tutorial. The resources consist of the Investigate Java control (and its sub-components) and the Investigate web service. The Investigate Java control is a custom Java control that assembles a credit-worthiness report on loan applicants. The Investigate web service is a web service that gives web access to the Investigate Java control. For a detailed descriptions of these resources see Tutorial: Java Control and Tutorial: Web Service.



## WebLogic Workshop Tutorials

1. From the **File** menu, select **New**—>**Application**
2. In the **New Application** dialog, in the upper left-hand pane, select **Tutorial**.

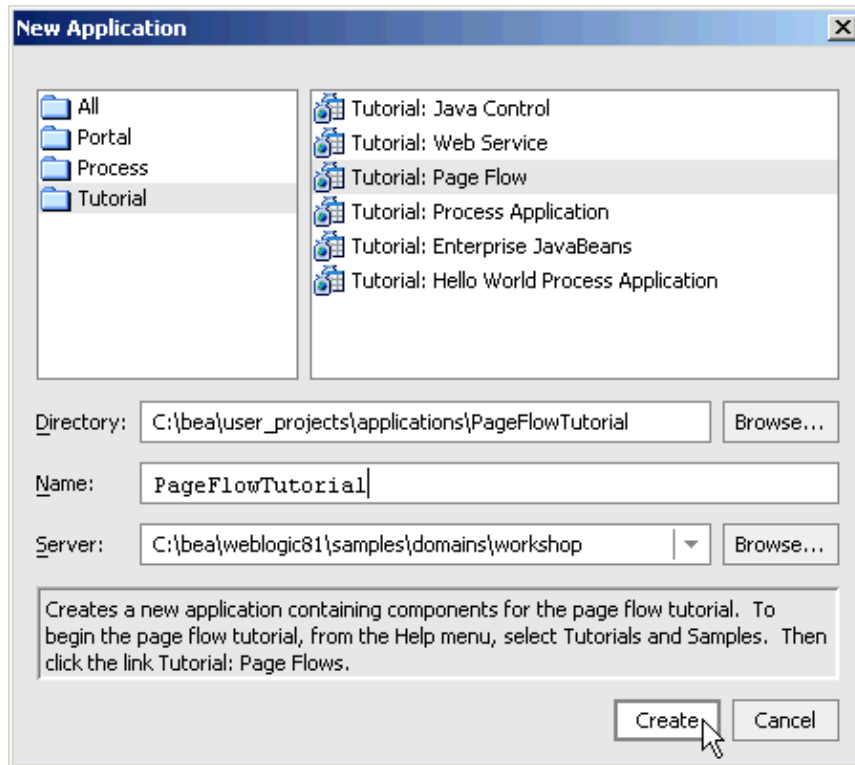
In the upper right-hand pane, select **Tutorial: Page Flow**.

In the **Directory** field, use the **Browse** button to select a location to save your source files. (The folder you choose is up to you. The default location is shown in the illustration below.)

In the **Name** field, type PageFlowTutorial.

In the **Server** field, from the dropdown list, select

BEA\_HOME\weblogic81\samples\domains\workshop.

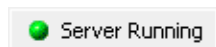


3. Click **Create**.

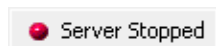
### To Start WebLogic Server

As you develop the Investigate page flow, you will deploy it to WebLogic Server to test its functionality. In this task you will start WebLogic Server to ensure that it is available during the development process.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.



If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow the instruction below to start WebLogic Server.

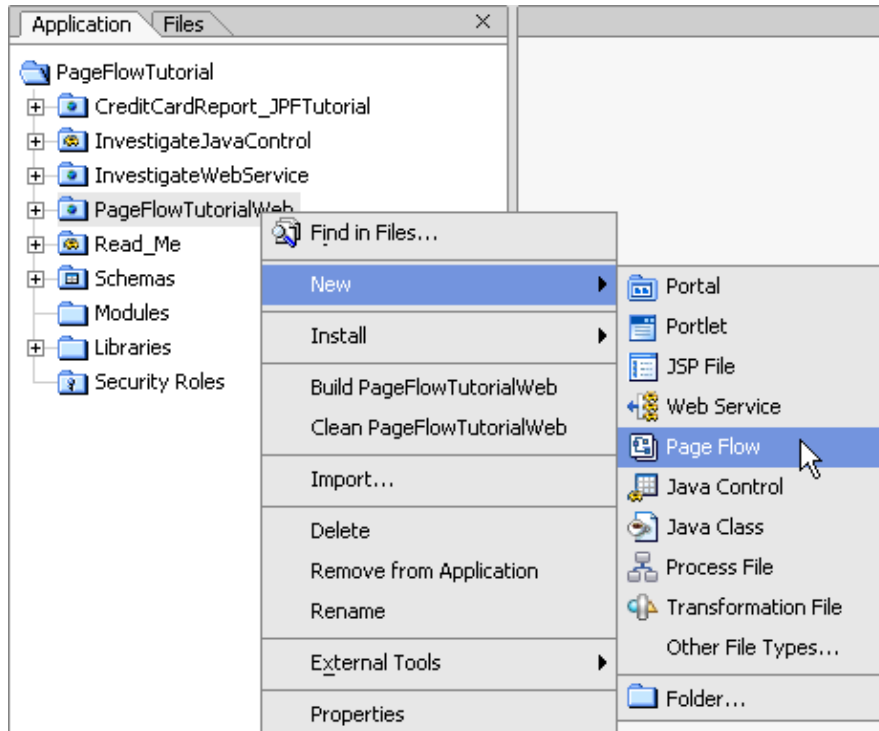
- **To start WebLogic Server:** from the **Tools** menu, choose **WebLogic Server**—>**Start WebLogic**

## Server.

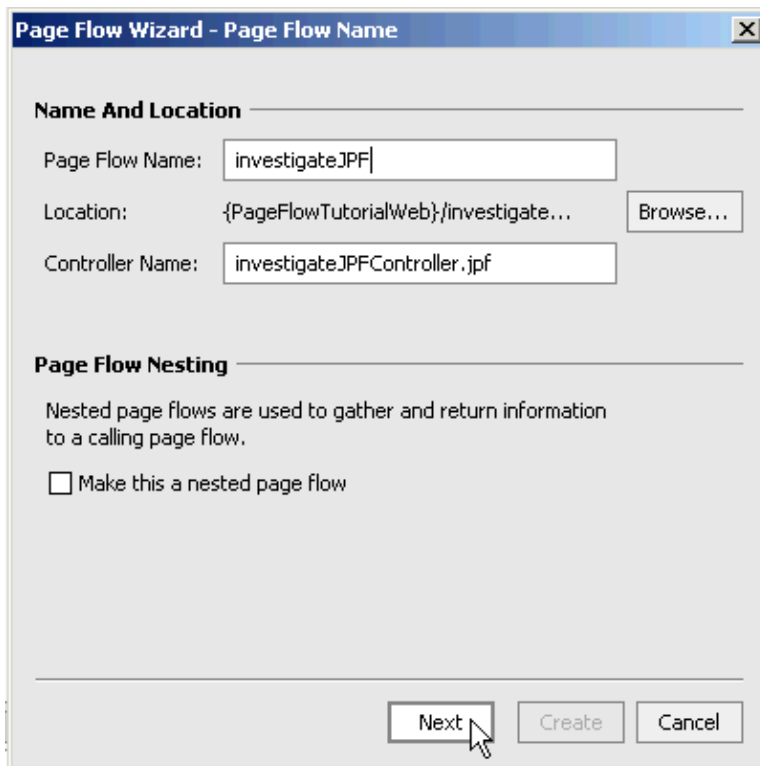
To Create a New Page Flow

You will develop your page flow within a Web Application project. This project has been pre-loaded into WebLogic Workshop as part of the new application you created above.

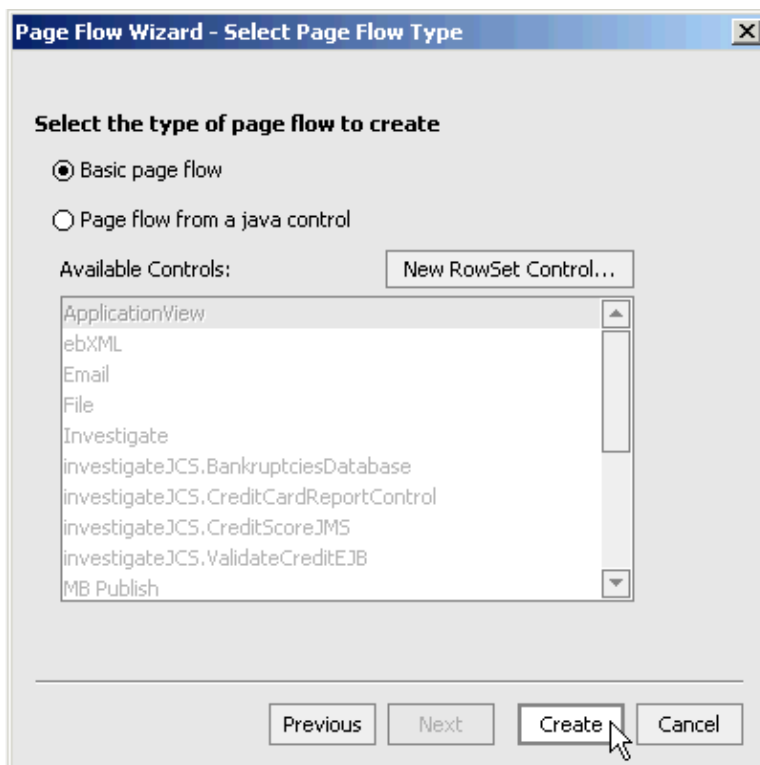
1. On the Application tab, right click the **PageFlowTutorialWeb** folder and select **New-->Page Flow**.



2. In the **Page Flow Wizard – Page Flow Name** dialog, in the **Page Flow Name** field, enter **investigateJPF**, and click **Next**.



3. In the *Page Flow Wizard – Select Page Flow Type* dialog, confirm that *Basic page flow* is selected.

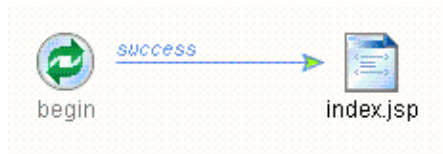


4. Click *Create*.

A new *page flow*, *investigateJPFController.jpf*, is created along with one *JSP page*, *index.jsp*. When you create a new page flow, it is automatically displayed in Flow View, a graphical representation of your page flow.

## WebLogic Workshop Tutorials

Flow View currently shows one Action method, named begin, and one JSP page, named index.jsp. The arrow, labeled "success", that points from the Action method to the JSP page tells you that the begin method is a navigational Action method: when the method is executed, it forwards users to the index.jsp page. (Not all Action methods are navigational: Action methods can do anything that a Java method can do.)



Every page flow must have a begin method. The begin method is the first method executed when your page flow is invoked by a web browser. When a user points a web browser at the page flow's URL

`http://localhost:7001/PageFlowTutorialWeb/investigateJPf/investigateJPfController.jspf`

the begin method will be executed and the user will be forwarded to the index.jsp page.

To see the source code for the begin method, double-click the icon for the begin method.



The source code for the begin method is displayed in Source View.

```
/**
 * This method represents the point of entry into the pageflow
 * @jpf:action
 * @jpf:forward name="success" path="index.jsp"
 */
protected Forward begin()
{
    return new Forward( "success" );
}
```

Note that the begin method has two annotations within the Javadoc comment above the method signature: @jpf:action and @jpf:forward. Javadoc was originally introduced as a way to specify inline documentation that could be extracted and presented as HTML. WebLogic Workshop uses a variety of Javadoc annotations to indicate how code should be treated at compile time and at run time. The @jpf:action annotation declares that the method should be compiled as an Action method and that it should implement all of the functionality of an Action method. The @jpf:forward annotation declares that the method has navigational functionality. In this case, the @jpf:forward is able to forward users to the index.jsp page.

The begin method works as follows. (1) When the method is invoked, it returns a Forward object with the parameter "success". (2) The WebLogic Workshop runtime then searches for a @jpf:forward annotation that defines the destination corresponding to the parameter "success", in this case, the index.jsp page. When it finds the destination, it forwards the user to the index.jsp page.

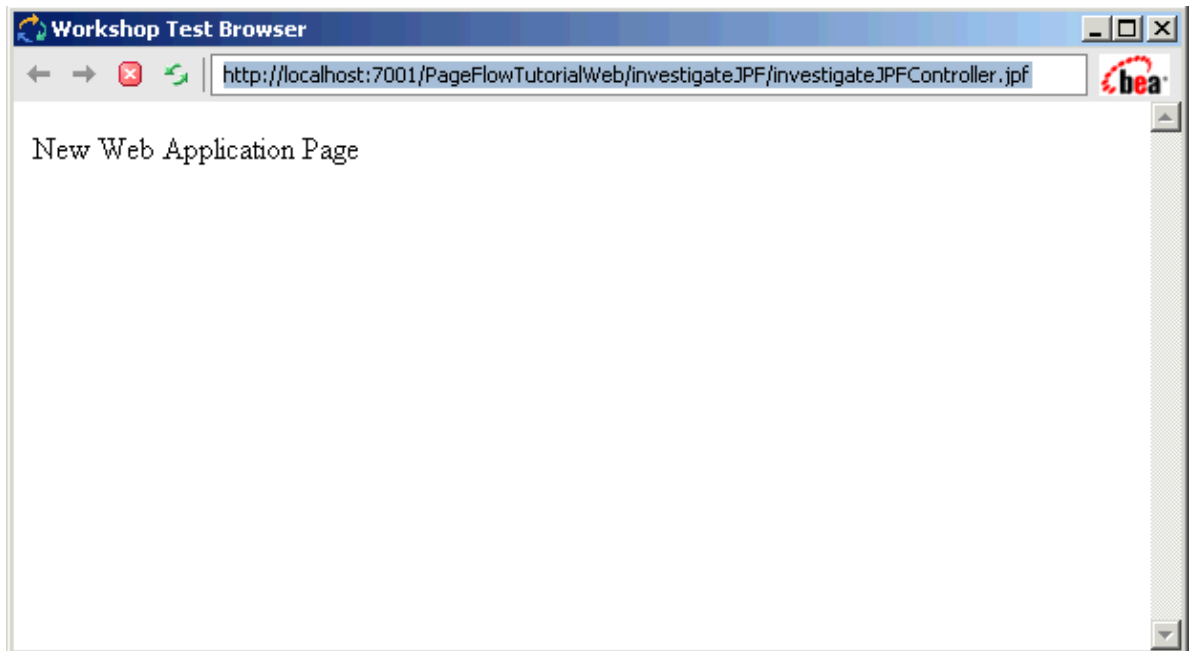
To Test the Page Flow

## WebLogic Workshop Tutorials

1. Confirm that investigateJPFCController.jpf is displayed in the main work area. (Throughout this tutorial, to display a file in the main work area, double-click its icon on the **Application** tab.)
2. Click the **Start** button, shown below.



The page flow builds and the Workshop Test Browser launches. Notice the address bar of the Test Browser points to the page flow file investigateJPFCController.jpf, but the browser displays the index.jsp page. When you point a browser at a page flow file, it executes its begin() method. In this case the begin() method forwards the user to the index.jsp page.



Click the arrow to navigate to the next step in the tutorial.



## Step 2: Processing Data with Page Flows

In this step you will create the data processing elements of your page flow. The data processing of your page flow consists of three parts: (1) accepting user requests for a credit report, (2) the retrieval of credit reports from a web service, and (3) the storage of the credit reports within the page flow.

(1) The first task is accomplished through a **Form Bean**. Form Beans act as intermediaries between the requests that users enter into JSP pages and your page flow. The data that users submit is typically stored in a Form Bean, which is then passed to an Action method for further processing.

(2) The second task is accomplished through an **Action method** which requests and retrieves credit reports from a web service.

(3) The third task is accomplished through a **member variable**. Member variables are Java objects that you place on your page flow, typically to store some data content. In this case you will store the credit reports retrieved from the web service, so that they can be displayed to the user.

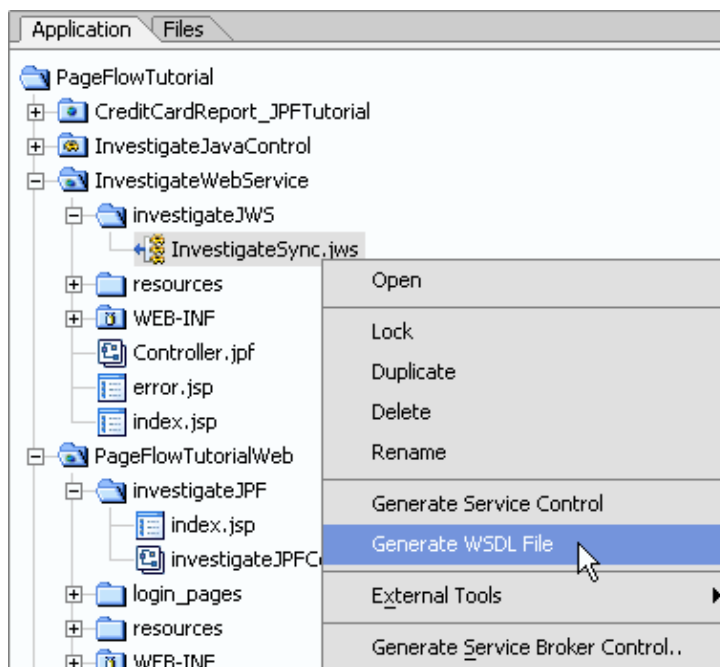
The tasks in this step are:

- To generate a WSDL file and a web service control
- To add a web service control
- To add an Action method and Form Bean
- To edit the Form Bean
- To edit the Action method
- To add a member variable
- To re-arrange the Flow View icons

To Generate a WSDL File and a Web Service Control

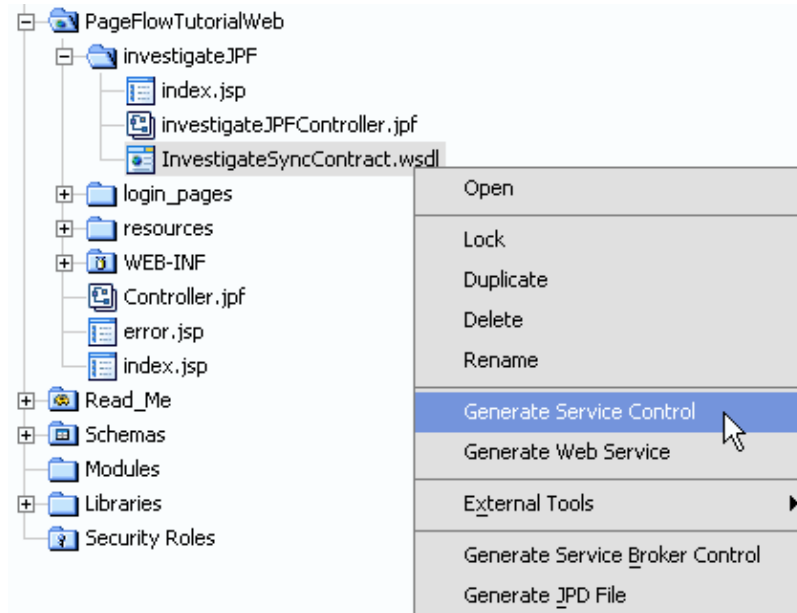
In this step you will create a control for the InvestigateSync web service, which allows your page flow to request and receive credit-worthiness reports from the web service.

1. On the **Application** tab, within the folder **PageFlowTutorial/InvestigateWebService/investigateJWS**, right-click the web service **InvestigateSync.jws** and select **Generate WSDL File**.



The WSDL file *InvestigateSyncContract.wsdl* is created and placed into the folder *investigateJWS*.

2. On the **Application** tab, hold the mouse button down over the *InvestigateSyncContract.wsdl* file and drag it into the *PageFlowTutorial/PageFlowTutorialWeb/investigateJPF* folder.
3. On the **Application** tab, right-click the WSDL file *InvestigateSyncContract.wsdl* and select **Generate Service Control**.



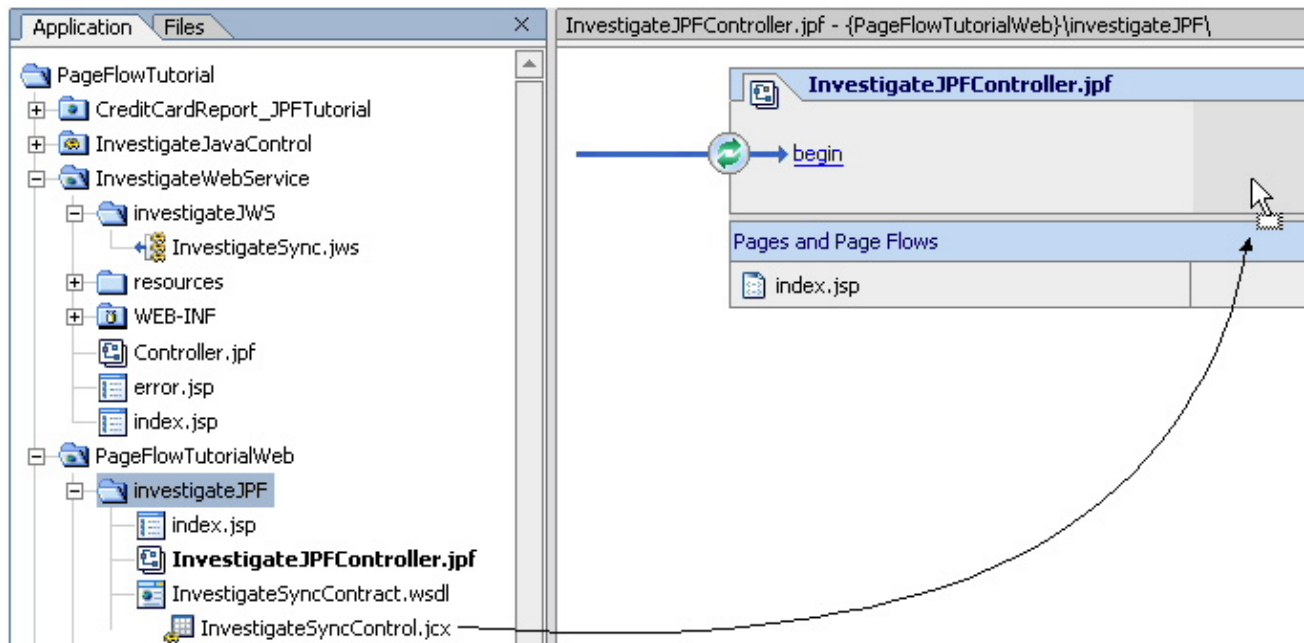
The control file, *InvestigateSyncControl.jcx*, is created and placed into the folder *investigateJPF*.

## To Add a Web Service Control

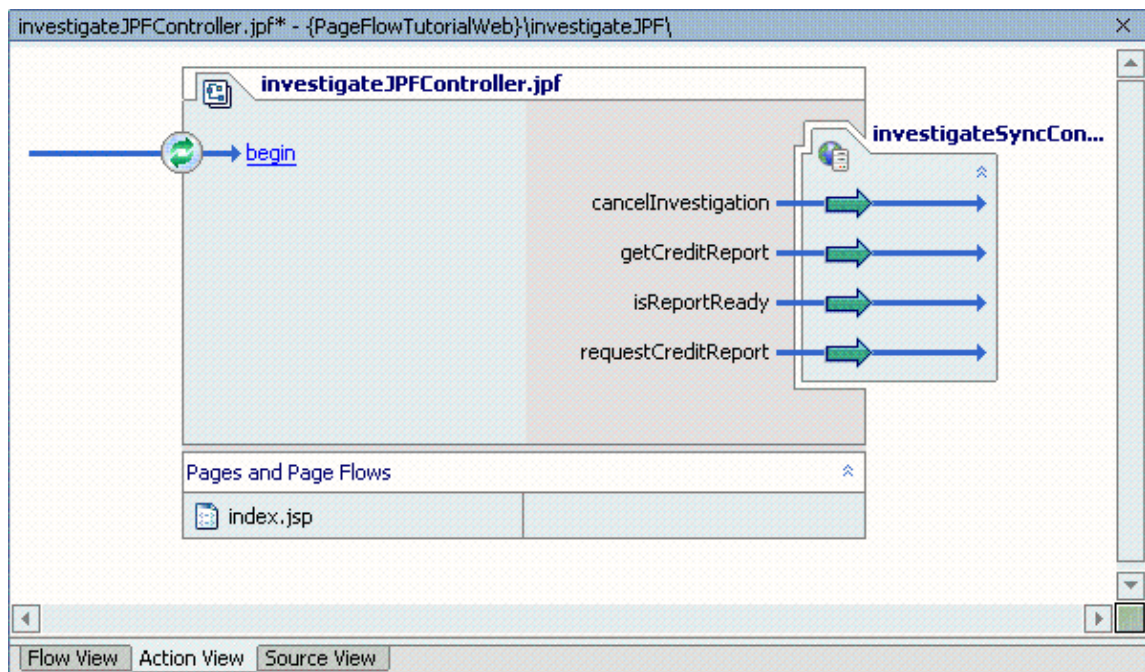
In this task you connect the page flow and the web service through the web service control you created in the previous task.

## WebLogic Workshop Tutorials

1. Confirm that the *investigateJPFCtrloller.jspf* file is displayed in the main work area.
2. Click the **Action View** tab.
3. On the **Application** tab, click the *InvestigateSyncControl.jcx* file and drag and drop it into **Action View**.



A new web service control is add to your page flow.



Action View shows you the controls and their methods available to your page flow file. In this case, one control is available containing four methods: `cancelInvestigation`, `getCreditReport`, `isReportReady`, and `requestCreditReport`.



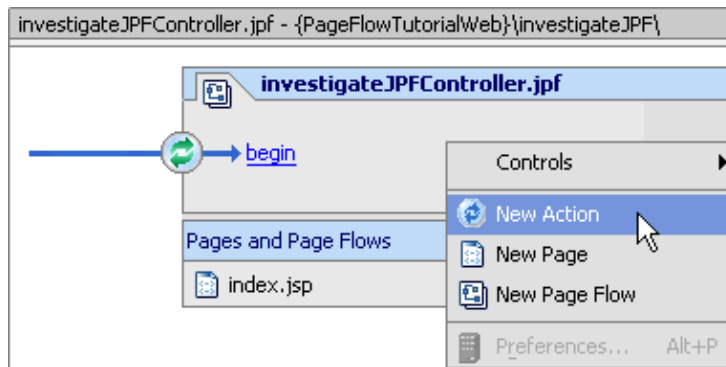
4. Press **Ctrl+S** to save your work.

### To Add an Action Method and a Form Bean

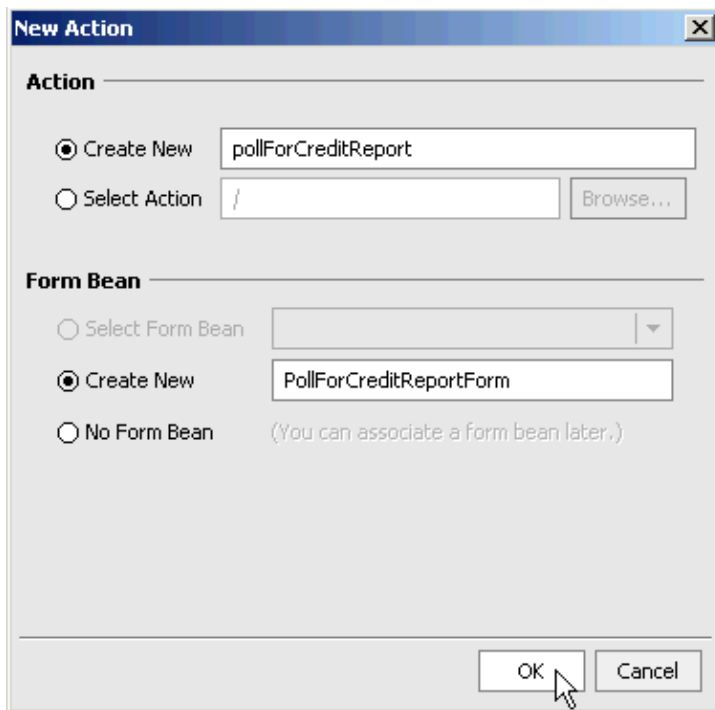
In this task you will add an Action method that processes user requests for credit reports. This Action method receives user input, retrieves a credit report from the InvestigateSync web service, and then forwards the user to a JSP page where the credit report is displayed.

You will also add a Form Bean. Form Beans are used to separate the data presentation layer from the data processing layer of your web application. Form Beans are used as *intermediaries* between the data that users input into JSP pages and the Action methods. Instead of submitting the data directly to the Action method, the data is first *databound* to a Form Bean and this Form Bean is then passed as a parameter to the Action method. This helps to separate the presentation layer (i.e., the user input forms on the JSP pages) and the processing layer (i.e., the Action methods), because the Form Bean can play a flexible, intermediary role.

1. Confirm that the *investigateJPFCController.jspf* file is open and displayed.
2. Click the **Action View** tab.
3. Right-click within the grey area of the page flow picture and select **New Action**.



4. In the **New Action** dialog, in the **Action** section, in the **Create New** field enter pollForCreditReport. In the **Form Bean** section, check **Create New** and enter PollForCreditReportForm.



5. Click **Ok**.

A new Action method is added to your page flow file.

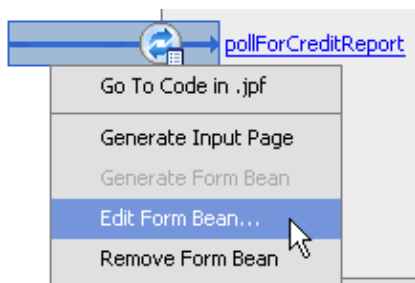


Note the Action method icon displays a small box on its lower right-hand corner. This indicates that the Action method takes a Form Bean parameter. Form Beans are used as intermediaries between user input and an Action method. When a user submits data from a JSP page, this data is loaded into a Form Bean, and then this Form Bean is passed as a parameter to an Action method.

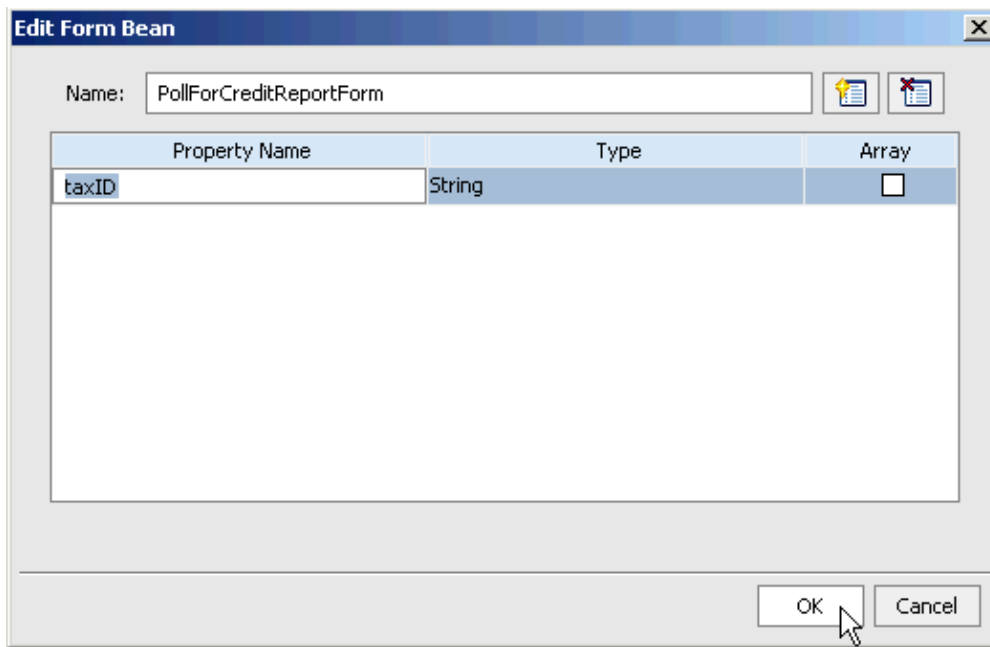
#### Edit the Form Bean

In this task you will add a taxID field to the Form Bean. This field will carry the data that users input into the JSP page (a page not yet created).

1. In **Action View**, right-click the icon for the method **pollForCreditReport** and select **Edit Form Bean**.



2. In the **Edit Form Bean** dialog, in the **Property Name** field, enter **taxID**.



3. Click **Ok**.

Before you move on to the next task, click the **Source View** tab, to see the code that has been generated. You have just added one Action method and one Form Bean shown below.

Notice that the Action method pollForCreditReport takes the Form Bean parameter PollForCreditReportForm form. Also notice that the Form Bean takes the form of an ordinary Java class with fields, and setter and getter methods on those fields.

```
/**
 * @jpf:action
 */
protected Forward pollForCreditReport(PollForCreditReportForm form)
{
    return new Forward("success");
}

/**
 * FormData get and set methods may be overwritten by the Form Bean editor.
 */
public static class PollForCreditReportForm extends FormData
{
    private String taxID;

    public void setTaxID(String taxID)
    {
        this.taxID = taxID;
    }

    public String getTaxID()
    {
        return this.taxID;
    }
}
```

To Edit the Action Method

In this task you will add code to the pollForCreditReport Action method. The code you add has two purposes:

(1) it polls the web service for a credit report and (2) it controls user navigation.

Polling is a form of synchronous communication between a client and a web service. When a client polls a web service it first asks the web service to begin some process. The client then waits some predetermined amount of time after which it asks the web service if the process is complete. If the process is complete then the client retrieves the result of the process; if the process is not complete, then the client continues to wait for the process to be completed.

1. Confirm that *investigateJPFController.jpf* is displayed in the main work area.
2. Click the **Action View** tab.
3. On the **Action View** tab, click the link text for the *pollForCreditReport* method.



The source code for the *pollForCreditReport* method is displayed in **Source View**.

4. Edit the *pollForCreditReport* to look like the following. Code to edit appears in red. (Note that the code below mentions the "response.jsp" page, which will be added in the next tutorial step.)

```
/**
 * @jpf:action
 * @jpf:forward name="response" path="response.jsp"
 */
protected Forward pollForCreditReport(PollForCreditReportForm form)
    throws java.lang.InterruptedException
{
    /**
     * Request a credit report from the InvestigateSync
     * web service.
     */
    investigateSyncControl.requestCreditReport(form.taxID);

    /**
     * Poll the isReportReady method 20 times, once a second,
     * to check if the result is ready.
     */
    for(int i=0; i<20; i++)
    {
        /**
         * If the report is ready, get the credit report,
         * store it in m_applicant, and forward the user to
         * response.jsp.
         */
        if(investigateSyncControl.isReportReady() == true)
        {
            m_applicant = investigateSyncControl.getCreditReport();
            return new Forward("response");
        }
        else
        {
            /**
             * If the report is not ready, wait one second before
             * polling again.
             */
            Thread.sleep(1000,0);
        }
    }
}
```

```

/*
 * The credit report was not delivered in the time allowed.
 * Store a message in the applicant object and Forward the user
 * to the response.jsp page.
 */
m_applicant.message = "The Investigate Web Service did not respond it the time a
return new Forward("response");
}

```

5. Press **Ctrl+S** to save your work. (Note that you will see red jagged lines under some of the elements of this code. This indicates that the compiler does not recognize these elements. In the next task you will remove these red lines.)

### To Add a Member Variable

In this step you add a member variable to your page flow file. This member variable will store the credit report provided by the InvestigateSync web service.

1. Confirm that the *investigateJPFCController.jpf* file is open and displayed.
2. Click the **Source View** tab.
3. Add the following code to the body of *investigateJPFCController*. (Note that the comment attached to the code below mentions the "response.jsp" page. This JSP page will be added in the next step of the tutorial.)

```

/**
 * The m_applicant object stores the credit report provided
 * by the InvestigateSync web service.
 * The tag on response.jsp is data bound to m_applicant.
 */
public InvestigateSyncControl.Applicant m_applicant;

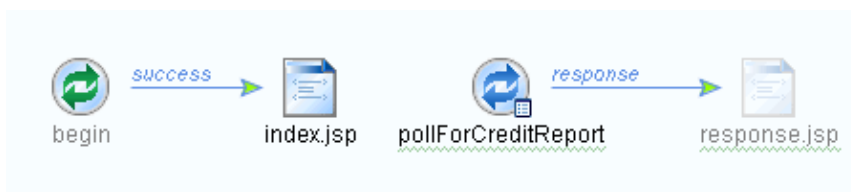
```

4. Click **Ctrl+S** to save your work.

### Re-arrange the Flow View Icons

This task does not change the functionality of your page flow, but it does make it easier to understand how the different parts of the page flow are related to one another.

1. Confirm that *investigateJPFCController.jpf* is displayed in the main work area.
2. Click the **Flow View** tab.
3. By selecting and dragging Action method and JSP icons, arrange the elements as shown below.



Notice that the JSP page *response.jsp* is greyed out. This indicates that *response.jsp* is referred to by the page flow file, but that it does not yet exist within the page flow folder. In the next task you will create the *response.jsp* page within the page flow folder.

## WebLogic Workshop Tutorials

Also notice that there is no way for a user to interact with the pollForCreditReport Action. In the next step you will add a JSP page which will allow the user to submit requests to the pollForCreditReport Action.

### Related Topics

[A Detailed Page Flow Example](#)

[Using Form Beans to Encapsulate Data](#)

[Using Polling as an Alternative to Callbacks](#)

[Tutorial: Web Services, Step 3: Add Support for Synchronous Communication](#)

Click the arrow to navigate to the next step in the tutorial.



## Step 3: Create the JSP Pages

In this step you complete the presentation components of your page flow. The presentation components consist of two JSP pages: (1) `request.jsp` allows users to request a credit report and (2) `response.jsp` displays the completed credit report to the user.

Page flows allow you to separate how your web application *processes* data from how it *presents* that data to users. When data processing is separated from data presentation, it's much easier to re-use the data processing aspects of the application in other contexts and to manage any future changes to your web application. Page flows link data processing and data presentation by means of *databinding*. Databinding is accomplished through the use of a special library of JSP tags that allow you to associate the presentation of data on the JSP pages with the data processing inside the page flow file. The special library of JSP tags begin with the prefix "netui-databinding".

The JSP pages you create in this step demonstrate two cases of databinding: (1) databinding when a user *submits* data, (2) databinding when data is *displayed* to the user.

(1) The `request.jsp` page demonstrates databinding when a user *submits* data to a page flow. In this case, the data that a user inputs into the `request.jsp` page is databound to a Form Bean. The Form Bean acts as an intermediary between the submitted data and the Action method that consumes the data. This helps to separate the presentation and processing layers, because changes to one layer do not necessitate changes to the other. Instead, developers need only change the Form Bean that intermediates between the two layers.

(2) The `response.jsp` page demonstrates databinding for *display* to the user. In this case, you databind to a Java object in order to render it as HTML in the browser. In particular, you will bind to the credit report (= the `m_applicant` object) to render it as HTML on the `response.jsp` page.

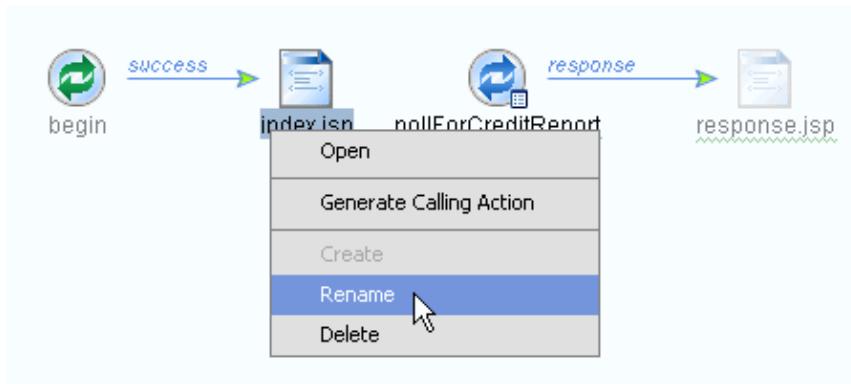
The tasks in this step are:

- To create the request page
- To create the response page
- To test the user interface

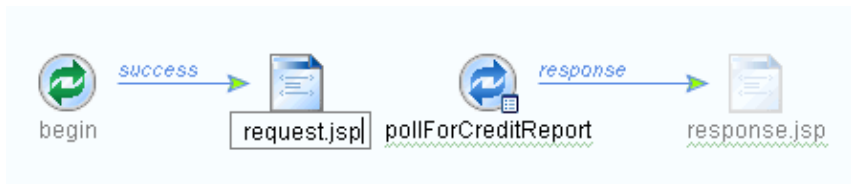
### To Create the Request Page

In this task you will rename the `index.jsp` page and add a user input form to the page. The form you add is databound to the `PollForCreditReportForm` Form Bean. The Form Bean is used as an intermediary between the data submitted by the user and the Action method that processes that data.

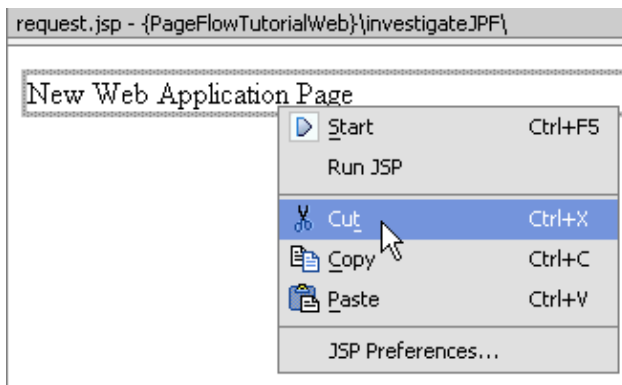
1. Confirm that the page flow file *`investigateJPFController.jspf`* is displayed in the main work area.
2. Click the ***Flow View*** tab.
3. Right-click the ***index.jsp icon*** and select ***Rename***.



4. In the editable field provided, replace the text `index.jsp` with the text `request.jsp` and press **Enter**.

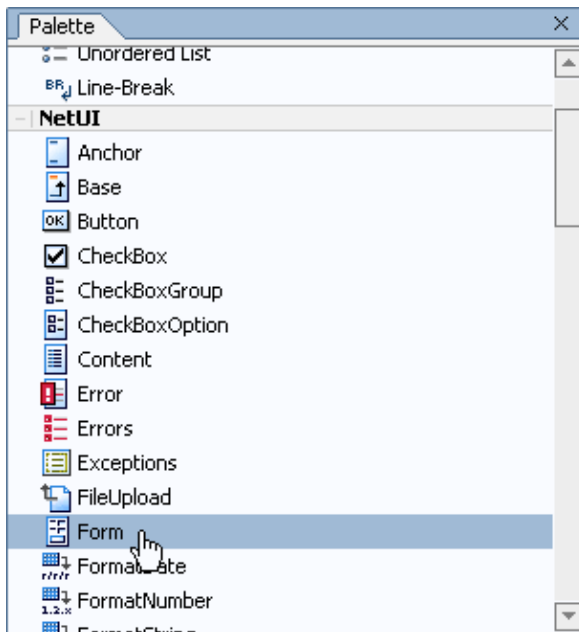


5. Double-click the ***request.jsp*** icon to display the JSP file in the main work area in Design View.  
 6. On the **Design View** tab, right-click the text **New Web Application Page** and select **Cut**.



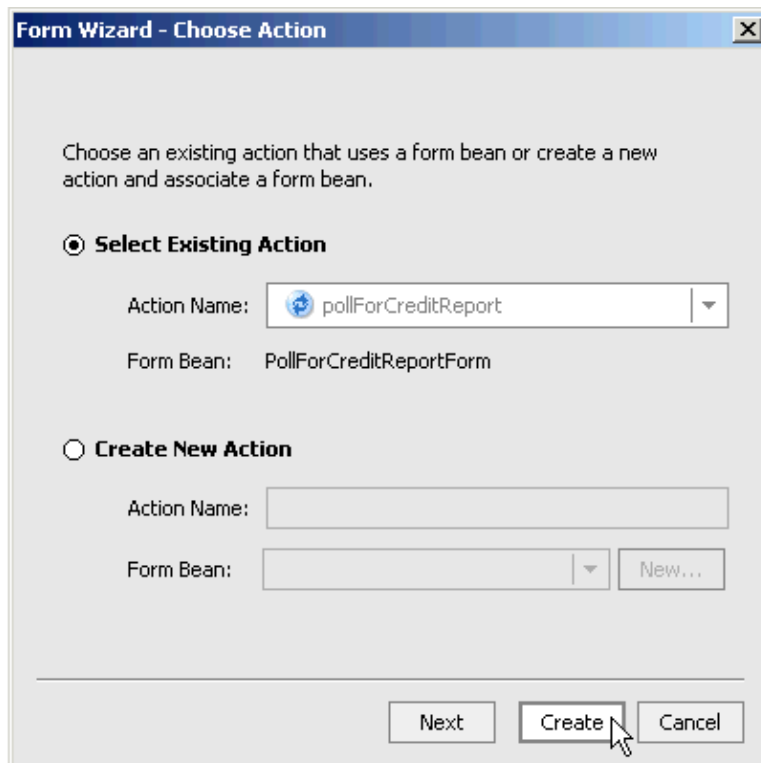
7. On the **Palette** tab, in the section labeled **NetUI**, select the **Form** icon and drag it into **Design View**.





The **Form Wizard** dialog appears.

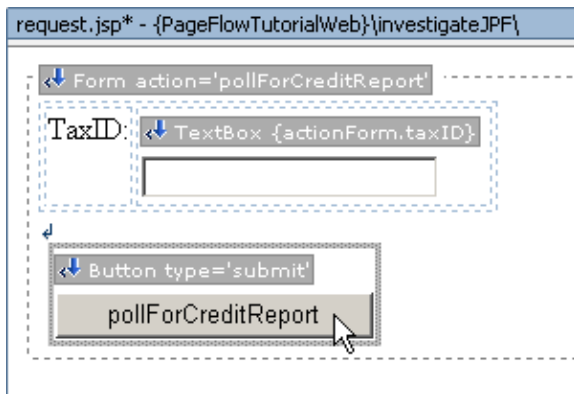
8. In the **Form Wizard – Choose Action** dialog, confirm that **Select Existing Action** is selected and that the **Action Name** field displays **pollForCreditReport**.



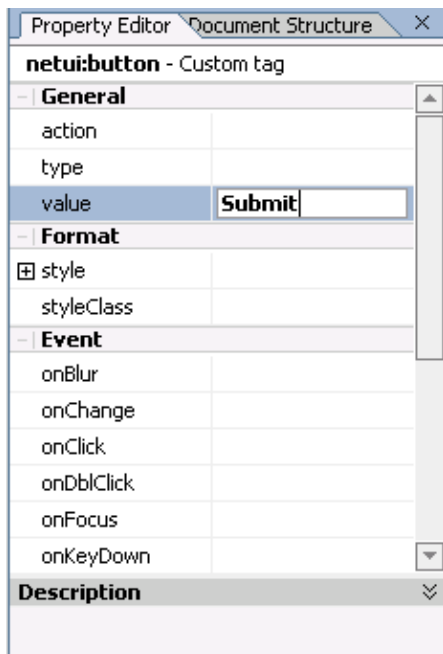
9. Click **Create**.

A user input form is added to the request.jsp page.

10. In **Design View**, select the gray **pollForCreditReport** button underneath the input box.

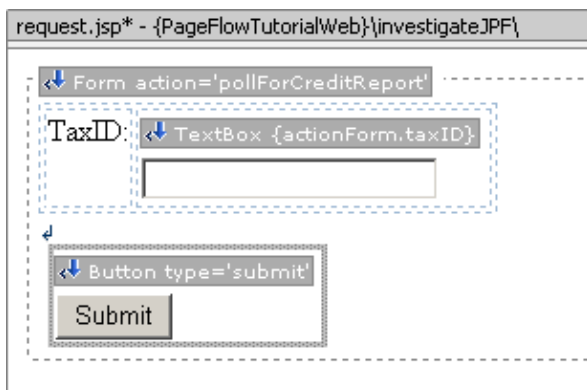


11. On the **Properties Editor** tab, in the **value** field and enter Submit (overwriting the text pollForCreditReport).



12. Press **Enter**.

Note the change that takes place in **Design View**.



13. Click the **Source View** tab to view the code you have just added to request.jsp.

## WebLogic Workshop Tutorials

```
<!--Generated by WebLogic Workshop-->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
  <head>
    <title>
      Web Application Page
    </title>
  </head>
  <body>
    <netui:form action="pollForCreditReport">
      <table>
        <tr valign="top">
          <td>TaxID:</td>
          <td>
            <netui:textBox dataSource="{actionForm.taxID}"/>
          </td>
        </tr>
      </table>
      <br/><netui:button value="Submit" type="submit"/>
    </netui:form>
  </body>
</netui:html>
```

When a user submits this `<netui:form>`, two things occur: (1) the user's data is databound to a new instance of the Form Bean `PollForCreditReportForm` and (2) this Form Bean instance is passed as a parameter to the method `pollForCreditReport`.

The databinding is specified by the data binding expression `{actionForm.taxID}`. *`{actionForm....}`* refers to the Form Bean; *`{actionForm.taxID}`* refers to the `taxID` field of the Form Bean.

The target method is specified by the attribute `action="pollForCreditReport"`. The ***action*** attribute specifies that the form submits its values to the `pollForCreditReport` method.

The following diagram shows how these expressions in `request.jsp` refer to elements in the page flow file.

**request.jsp****InvestigateJPFFCont**

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
  <head>
    <title>
      Web Application Page
    </title>
  </head>
  <body>
    <netui:form action="pollForCreditReport">
      <table>
        <tr valign="top">
          <td>TaxID:</td>
          <td>
            <netui:textBox dataSource="(actionForm.taxID)" />
          </td>
        </tr>
      </table>
      <br/><br/>
      <netui:button value="Submit" type="submit"/>
    </netui:form>
  </body>
</netui:html>

```

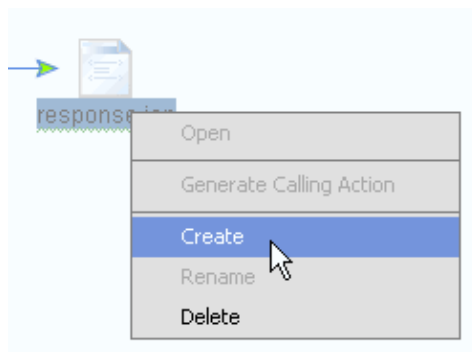
**protected** Forward pol.**public static class** I**private** String ta**public void** setTa**this**.taxID =**public** String get**return** this.t

14. Press **Ctrl+S** to save your work, then press **Ctrl+F4** to close request.jsp.

## To Create the Response Page

In this task you will create a JSP page to display the credit report. You will render the credit report as an HTML table.

1. Confirm that *investigateJPFFController.jspf* is displayed in the main work area.
2. If necessary, click the **Flow View** tab.
3. Right-click the *response.jsp icon* and select **Create**.



A new JSP page, response.jsp, is created in the investigateJPF folder.

4. Double-click the *response.jsp icon*.

The response.jsp page is displayed in the main work area in **Design View**.

5. Click the **Source View** tab. Edit the source code so it looks like the following. Overwrite any code that already exists in the response.jsp file.

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
  <head>
    <title>
      Investigate: Credit Report
    </title>
  </head>
  <body>
    <table border="1">
      <tr>
        <td colspan="2" bgcolor="#ccccff">
          Credit Report
        </td>
      </tr>
      <tr>
        <td>Tax ID Number:</td>
        <%--
          The following netui:label tags use different databinding expressions to
          different fields on the m_applicant object. The m_applicant object is a
          of the page flow file investigateJPFCController.jspf
        --%>
        <td><netui:label value="{pageFlow.m_applicant.taxID}"/></td>
      </tr>
      <tr>
        <td>First Name</td>
        <td><netui:label value="{pageFlow.m_applicant.firstName}"/></td>
      </tr>
      <tr>
        <td>Last Name</td>
        <td><netui:label value="{pageFlow.m_applicant.lastName}"/></td>
      </tr>
      <tr>
        <td>Available Credit Card Balance</td>
        <td><netui:label value="{pageFlow.m_applicant.availableCCCredit}"/></td>
      </tr>
      <tr>
        <td>Bankrupt?</td>
        <td><netui:label value="{pageFlow.m_applicant.currentlyBankrupt}"/></td>
      </tr>
      <tr>
        <td>Credit Score</td>
        <td><netui:label value="{pageFlow.m_applicant.creditScore}"/></td>
      </tr>
      <tr>
        <td>Approval Level</td>
        <td><netui:label value="{pageFlow.m_applicant.approvalLevel}"/></td>
      </tr>
      <tr>
        <td>Message</td>
        <td><netui:label value="{pageFlow.m_applicant.message}"/></td>
      </tr>
    </table>
    <p><a href="request.jsp">Request another credit report</a>
  </body>
</netui:html>

```

## WebLogic Workshop Tutorials

Note the `<netui:label>` tags in the code above. Each `<netui:label>` tag uses a databinding expression to refer to different fields of the `m_applicant` object, a member variable of the page flow file `investigateJPFController.jpf`.

The expression `{pageFlow....}` refers to the current pageFlow file: `investigateJPFController.jpf`.

The expression `{pageFlow.m_applicant}` refers to the member variable `m_applicant` on the current page flow file.

The expressions `{pageFlow.m_applicant.taxID}`, `{pageFlow.m_applicant.firstName}`, etc. refer to different fields of the `m_applicant` object. (The different fields of the `m_applicant` object are specified by its data type: `InvestigateSyncControl.Applicant`. Open the `InvestigateSyncControl.jcx` file to see this type.)

6. Press **Ctrl+S** to save your work, then press **Ctrl+F4** to close `response.jsp`.

### To Test the User Interface

1. Confirm that ***investigateJPFController.jpf*** is displayed in the main work area.
2. Click the ***Start*** button, shown below.



Your page flow is built and the Workshop Test Browser launches.

3. In the ***Workshop Test Browser***, in the ***TaxID*** field, enter the 9 digit number 333333333 and click ***Submit***.



**Note:** Use one of the following (9 digit) taxID's to test your page flow:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.

The `response.jsp` page displays the results.

Credit Report	
Tax ID Number:	333333333
First Name	Jane
Last Name	Doe
Available Credit Card Balance	4000
Bankrupt?	true
Credit Score	660
Approval Level	Applicant is a moderate risk.
Message	

[Request another credit report](#)

#### Related Topics

Using Data Binding in Page Flows

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



## Step 4: Role-Based Security

In this step you will secure your page flow application using role-based security. You will restrict access to the page flow to only those users who have been granted the Investigators role.

Any time you implement a role-based security strategy you must do two things: you must set up (1) an authentication procedure and (2) an authorization procedure.

An *authentication* procedure verifies that an incoming user *is who he says he is*. In this case, the authentication procedure will be based on a username/password challenge. A user who wants to access the Investigate page flow will be asked to provide a matching username and password.

An *authorization* procedure decides whether a given (already authenticated) user has been granted the necessary permissions to access a given resource. In this case, the authorization procedure will be based on role membership. We will modify the Investigate page flow so that only users in the Investigators role can access the page flow.

Note that the authorization procedure assumes that the authentication procedure has already been completed. This is because a user's role membership can be decided only once his identity has been established by the authentication procedure.

The tasks in this step are:

- to set up an authentication procedure
- to set up an authorization procedure
- to add a test user
- to test the Investigate page flow

### Set up an Authentication Procedure

In this task you will place a security constraint on the Investigate page flow, a constraint that requires clients to provide a valid username and password before they can access the Investigate page flow. You will use a set of login JSP pages that are already included in the page flow application. The security constraint specifies that users who try to access the Investigate page flow will be redirected to the login pages. If they successfully login in they will be directed back to the page flow.

This sort of authentication procedure is sometimes called "Form-based" security, because of the HTML form submitted from the login JSP pages. (Note that this sort of authentication procedure should not be used to log in users *once they have entered* the page flow. This is because page flows do not support redirection away from and back to the navigational Action methods within the page flow. But this kind of authentication procedure can be used to login users *before* they enter the page flow.)

1. On the **Application** tab, double-click the file **PageFlowTutorial/PageFlowTutorialWeb/WEB-INF/web.xml**.
2. Add the following `<security-constraint>`, `<login-config>` and `<security-role>` elements to the **web.xml** file. Add the elements immediately before the closing `</web-app>` tag. Code to add is shown in red. Do not add code shown in black.

```
<taglib>
  <taglib-uri>netui-tags-template.tld</taglib-uri>
  <taglib-location>/WEB-INF/netui-tags-template.tld</taglib-location>
```



</taglib>

<!--

The following <security-constraint> element specifies which web resource should be protected. It also specifies that only users granted the Users role can access the resource. Note that this role restriction is "toothless" because any successfully logged in user will automatically pass the role restriction (since all logged in users are members of the Users group, which is mapped to the Users role).

The purpose of the role restriction is its side effect: it causes WebLogic Server to protect the web resource.

-->

<security-constraint>

    <web-resource-collection>

        <web-resource-name>The Investigate Page Flow</web-resource-name>

        <description>

            Protect the following web resources with a username / password challenge

        </description>

        <url-pattern>/investigateJPF/investigateJPFController.jsp</url-pattern>

        <http-method>GET</http-method>

        <http-method>POST</http-method>

    </web-resource-collection>

    <auth-constraint>

        <description>

            These are the roles who have access the web resources above.

        </description>

        <role-name>

            Users

        </role-name>

    </auth-constraint>

    <user-data-constraint>

        <description>

            This is how the user data must be transmitted.

        </description>

        <transport-guarantee>NONE</transport-guarantee>

    </user-data-constraint>

</security-constraint>

<!--

Log users in with "Form-based" authentication.

Use the JPS pages login.jsp and fail\_login.jsp

to login candidate users.

-->

<login-config>

    <auth-method>FORM</auth-method>

    <realm-name>default</realm-name>

    <form-login-config>

        <form-login-page>/login\_pages/login.jsp</form-login-page>

        <form-error-page>/login\_pages/fail\_login.jsp</form-error-page>

    </form-login-config>

</login-config>

<!--

The following <security-role> element declares the Users role so that it can be mapped to the users principal. For the mapping see weblogic.xml.

-->

<security-role>

    <description>

        The Users role is mapped to the users group, a pre-defined principal in WebLogic default authentication provider. For the mapping see weblogic.xml.

    </description>

```

        <role-name>
            Users
        </role-name>
    </security-role>

</web-app>

```

3. On the **Application** tab, double-click the **WEB-INF/weblogic.xml** file.
4. Add the following <security-role-assignment> element to weblogic.xml. Make sure to add the code directly after the opening <weblogic-web-app> tag. Code to add is shown in red. Do not add code shown in black.

```

<weblogic-web-app>

    <!--
    Grants the Users role to clients who possess the users principal.

    The users group is a pre-defined principal in the WebLogic Server default authentication.
    All clients who log in with a valid username/password pair automatically
    possess the users principal.

    Since logged in clients automatically possess the users principal,
    and the <security-role-assignment> below grants the Users role to anyone possessing
    the users principal, anyone who logs in with a valid username/password pair
    is automatically granted the Users role.
    -->
    <security-role-assignment>
        <role-name>Users</role-name>
        <principal-name>users</principal-name>
    </security-role-assignment>

    <jsp-descriptor>
        <!-- Comment the jspServlet param out to go back to weblogic's jspc -->
        <jsp-param>
            <param-name>jspServlet</param-name>
            <param-value>weblogic.servlet.WlwJSPServlet</param-value>
        </jsp-param>
        <jsp-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </jsp-param>
    </jsp-descriptor>

    <url-match-map>
        weblogic.servlet.utils.SimpleApacheURLMatchMap
    </url-match-map>
</weblogic-web-app>

```

In web.xml, the <auth-constraint> element, standing for *authorization* constraint, states that clients must be granted the Users role in order to access the Investigate page flow. It may seem strange to set up an *authorization* procedure at this point. Wasn't the idea to set up an *authentication* procedure? There is a trick here: the presence of the <auth-constraint> element causes WebLogic Server to *both* authenticate and authorize the client. WebLogic Server first asks the candidate user for a username/password pair; second WebLogic Server applies the authorization procedure called for by the <auth-constraint>.

Also note that any client who passes the username/password challenge automatically passes the authorization procedure. This is because any client with a valid username/password pair is

## WebLogic Workshop Tutorials

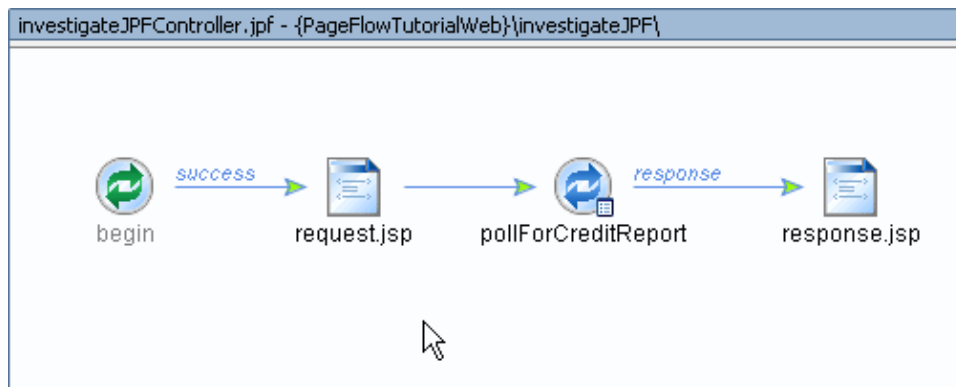
automatically granted the users principal by WebLogic Server (users is a pre-configured principal in WebLogic Server) and the users principle is mapped to the Users role in weblogic.xml. So at this point we have set up a authentication procedure that challenges clients for a username and password and a "toothless" authorization procedure that an authenticated client will automatically pass. We will set up the "real" authorization procedure below.

5. Press **Ctrl+S** to save your work. Press **Ctrl+F4** twice to close web.xml and weblogic.xml.

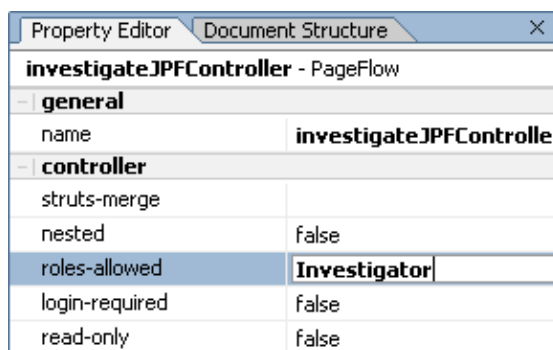
### Set up an Authorization Procedure

In this task you will place an authorization constraint on the Investigate page flow, such that only those users who have been granted the role of Investigator will be able to gain access to the Investigate page flow.

1. Confirm that **investigateJPFCController.jspf** is displayed in the main work.
2. Click the **Flow View** tab, if necessary.
3. In **Flow View**, click a spot within the white space to confirm that the page flow file as a whole is selected.



4. On the **Property Editor** tab, in the **Roles Allowed** property, enter Investigator.

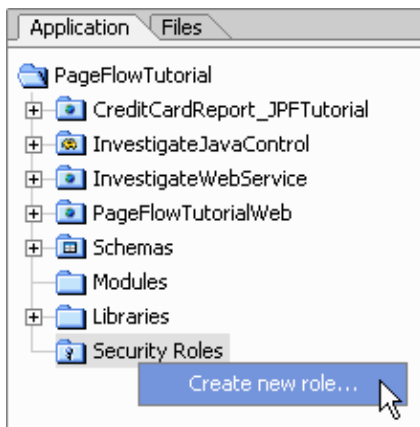


5. Press **Enter**.
6. Press **Ctrl+S** to save your work.

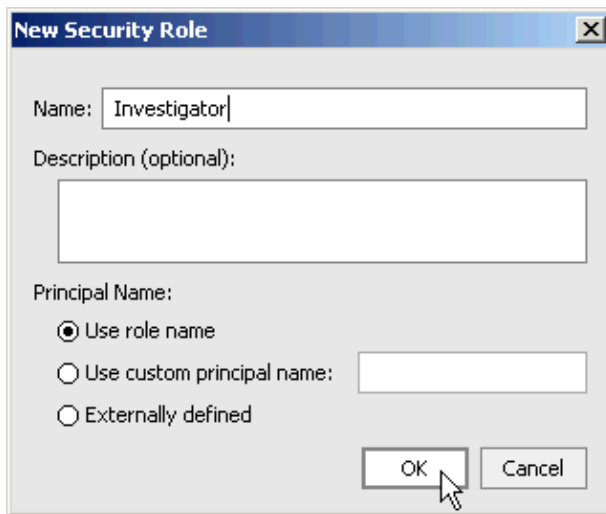
Declare the Investigator Security Role, Add a Test User, and Map this User to the Investigator Security Role

In this task you do three things. First, you will declare the security role Investigator. This role is a *scoped-role*. It is scoped at the application level. Application-scoped roles are declared in the META-INF/application.xml file. (This file does not appear on the **Application** tab by default, but, if you wish, you can see this file by opening PageFlowTutorial/META-INF/application.xml using your operating system.) Second, you will add a test user with the same name: Investigator. Third you will map this user to the security role. (The mapping is declared in the PageFlowTutorial/META-INF/weblogic-application.xml file)

1. On the **Application** tab right-click the **Security** folder and select **Create new role**.



2. In the **New Security Role** dialog, in the **Name** field, enter Investigator. In the section labeled **Principal Name**, ensure that Use role name is selected.



3. Click **Ok**.

The New Security Role dialog has three effects:

- (1) A new (application–scoped) role is added to the application.
- (2) A new user is added to the WebLogic Workshop security framework. When "Use role name" is selected, this user has the same name as the role. By default, the password for this user is password.
- (3) The user is mapped to the role.

(For more information on creating security users and roles with the New Security Role dialog, see the help topic [How Do I: Create An Application–Scoped Security Role?](#))

Test the Investigate Page Flow

### ***Test for Successful Authentication and Successful Authorization***

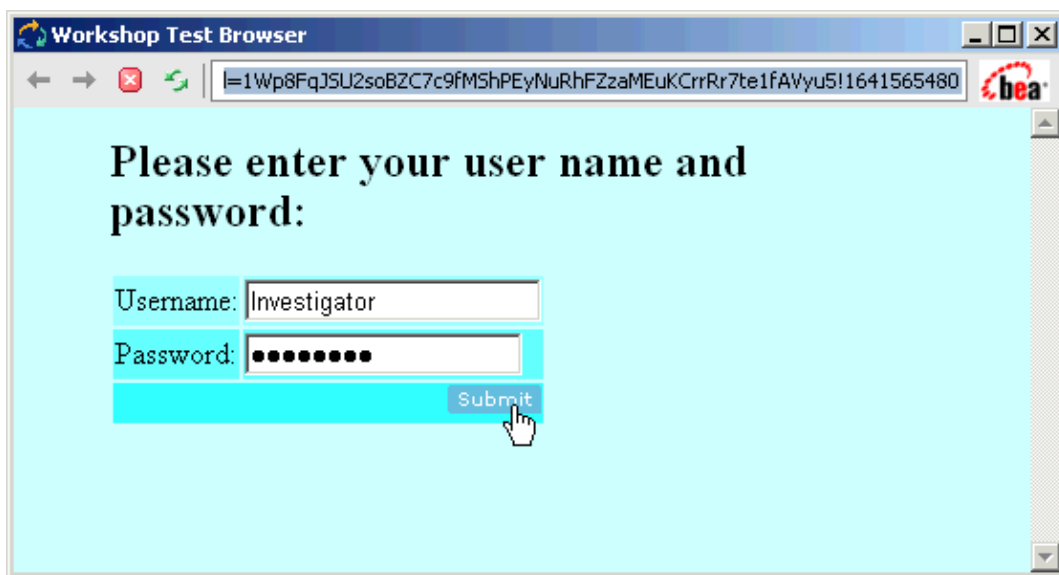
In this test you will ensure that users who have been granted the role of Investigator can pass both the

authentication procedure and authorization procedure.

1. Confirm that *investigateJPFController.jspf* is displayed in the main work area.
2. Right-click the folder *PageFlowTutorialWeb* and select *Build PageFlowTutorialWeb*. (It is a good idea to re-compile your web application project after making changes to the configuration files web.xml and weblogic.xml. This helps ensure that the web application properly deploys to WebLogic Server.)
3. Click the *Start* button.

The Investigate page flow builds and the login.jsp page launches.

4. In the *login.jsp* page, in the *Username* field, enter Investigator.  
In the *Password* field, enter password.  
Click *Submit*.



5. In the *Workshop Test Browser*, in the *TaxID* field, enter the 9 digit number 222222222 and click *Submit*.

The credit report is retrieved and displayed.

6. Close the *Workshop Test Browser*.

### *Test for Successful Authentication but Failed Authorization*

In this test you will ensure that users who have not been granted the role of Investigator do not pass the authorization process. You will do this by logging in as the user 'weblogic'. This user will pass the authentication procedure, since it is a user known to the WebLogic Security authentication provider. But 'weblogic' has not been granted the Investigator role, so it will fail the authorization procedure.

1. Confirm that *investigateJPFController.jspf* is displayed in the main work area.
2. Click the *Start* button.

The Investigate page flow builds and the login.jsp page launches.

3. In the *login.jsp* page, in the *Username* field, enter **weblogic**.  
In the *Password* field, enter **weblogic**.  
Click *Submit*

The Workshop Test Browser displays the following error.

## Page Flow Error - Unsatisfied Role Restriction

<b>Page Flow:</b>	/investigateJPF/investigateJPFController.jspf
<b>Action:</b>	begin

Action **begin** requires the user to be in one of the following roles: investigator.

4. Close the *Workshop Test Browser*.

### Test For Failed Authentication and Failed Authorization

In this test you will ensure that users that are unknown to the WebLogic Server authentication provider cannot successfully login.

1. Confirm that *investigateJPFController.jspf* is displayed in the main work area.
2. Click the *Start* button.

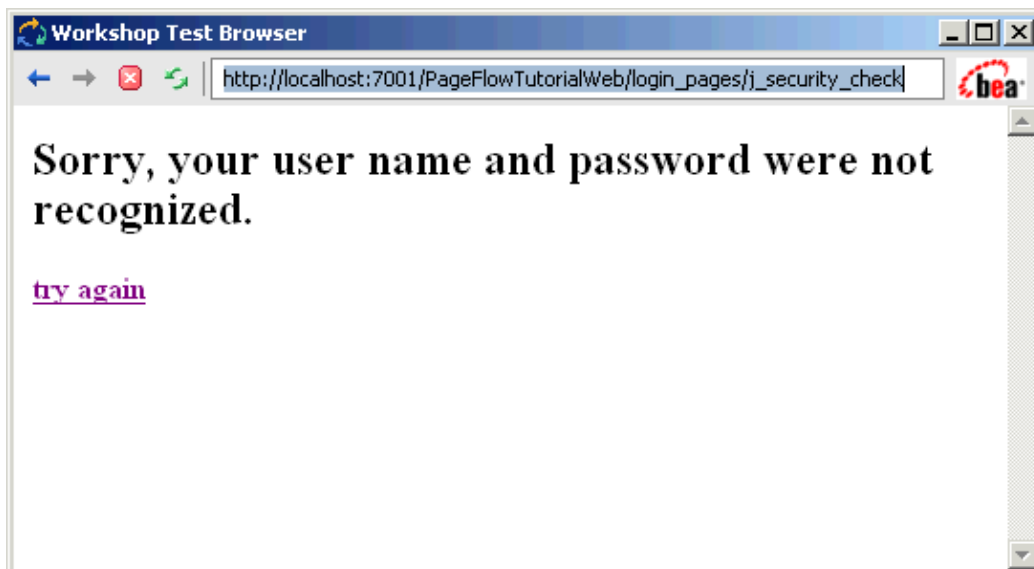
The Investigate page flow builds and the login.jsp page launches.

3. In the *login.jsp* page, in the *Username* field, enter **wrong\_user**.

In the *Password* field, enter **wrong\_password**.

Click *Submit*

The Workshop Test Browser displays the failed\_login.jsp page.



Related Topics

Role-Based Security

Handling Exceptions in Page Flows

Step 4: Role-Based Security

## WebLogic Workshop Tutorials

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



# Summary: Page Flow Tutorial

This tutorial introduced you to the basics of building web applications with WebLogic Workshop page flows.

## Concepts and Tasks Introduced in This Tutorial

- JSP files make up the presentation layer of a web application
- JPF files contain the code, individual Action methods, that determines the major features of a Workshop web application: how users navigate from page to page, and how data moves around the application.
- User input data is data bound to Form Beans before the data is submitted to an Action method.
- Built-in Workshop Java controls (web service controls, database controls, etc.) and custom Java controls (JCS files) provide access to the back-end data resources of a web application.
- You can use the <netui...> tag library to data bind to Java objects and render them as HTML.

### Related Topics

#### Getting Started with Page Flows

Click the arrow to navigate through the tutorial:





# Tutorial: Building Your First Business Process

WebLogic Integration's business process management (BPM) functionality enables the integration of diverse applications and human participants, as well as the coordinated exchange of information between trading partners outside of the enterprise.

This tutorial provides a tour of the features available to design business processes in the WebLogic Workshop graphical design environment. It describes how to create a business process that orchestrates the processing of a *Request for Quote*.

## Tutorial Goals

The goal of the tutorial is to provide the steps to create and test a business process using the graphical environment provided in WebLogic Workshop. It includes:

- Designing communication nodes in a business process that is, creating the interface between your business process and its clients and resources. Clients of business processes can be any other resources or services that invoke business processes to perform one or more operations.
- Designing the interactions with clients, including creating the methods that expose your business process's functionality.
- Designing the interactions with resources using controls. WebLogic Platform controls make it easy to access enterprise resources, such as databases, Enterprise Java Beans (EJBs), Web services, and other business processes (including those that use RosettaNet and ebXML business processes) from within your application.
- Handling XML, non-XML, and Java data types in the business process includes working with XML schemas and transforming data between disparate data types using the Transformation tool.
- Designing business processes to publish and subscribe to message broker channels.



# Tutorial: Building Your First Data Transformation

Data transformation is the mapping and conversion of data from one format to another. For example, XML data can be transformed from XML data valid to one XML Schema to another XML document valid to a different XML Schema. Other examples include the data transformation from non-XML data to XML data. This tutorial introduces the basics of building a data transformation by describing how to create and test a XML-to-XML data transformation using WebLogic Workshop.

In WebLogic Integration business processes, a data transformation transforms data using queries (written in the XQuery language). This tutorial describes the steps for building a query in the XQuery language a language defined by the World Wide Web Consortium (W3C) that provides a vendor independent language for the query and retrieval of XML data.

To learn about the XQuery language, see the XQuery 1.0: An XML Query Language Specification – W3C Working Draft 16 August 2002 at the W3C web site at the following URL:

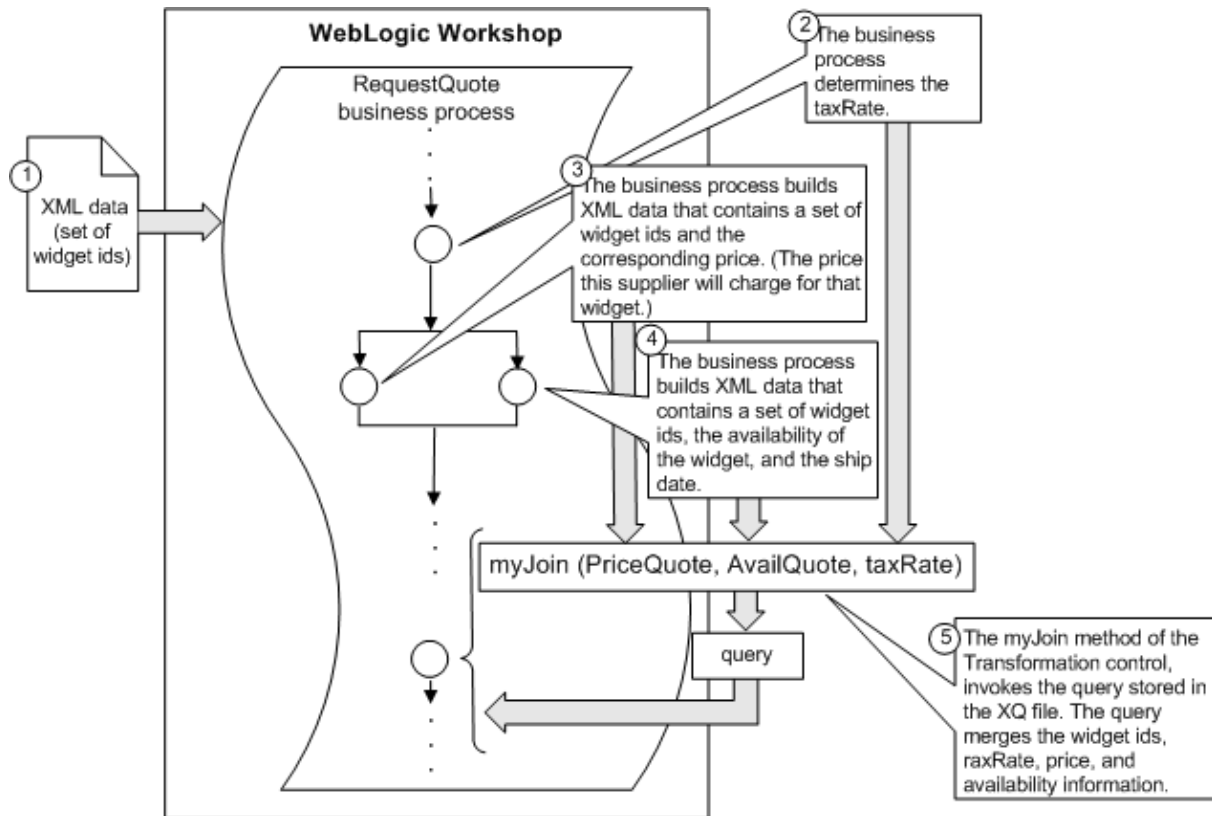
<http://www.w3.org/TR/2002/WD-xquery-20020816>

The WebLogic XQuery engine invoked by a business process conforms to the August 16, 2002 draft of the XQuery Specification.

To learn more about XML and XML Schemas, see Java and XML Basics.

The data transformation created in this tutorial is invoked in the RequestQuote business process. This business process is created to meet the business needs of an enterprise. The enterprise starts the business process as a result of receiving a Request for Quote from clients, checks the enterprise's inventory and pricing systems to determine whether the order can be filled, and sends a quote for the requested items to the client. To learn more about creating business processes and the RequestQuote business process, see *Tutorial: Building Your First Business Process*.

The following figure shows the flow of data in the RequestQuote business process of the Tutorial Process application.



The purpose of the RequestQuote business process is to provide price and availability information for a set of widgets. The flow of the data through the RequestQuote business process is represented by the following steps:

1. The business process receives the set of widget IDs.
2. The business process determines the tax rate for the shipment and puts the result in the taxRate float business process variable.
3. The business process gets the price of each of the requested widgets from a source and places the resulting XML data into the priceQuote business process variable. (This XML data is valid to the XML Schema in the PriceQuote.xsd file.)
4. The business process gets information about availability for the widgets from another source and places the resulting XML data into the availQuote business process variable. (This XML data is valid to the XML Schema in the AvailQuote.xsd file.)
5. The business process invokes the **Combine Price and Avail Quotes** node. The **Combine Price and Avail Quotes** node calls the myJoin Transformation method stored in the Transformation file called MyTutorialJoin.dtf file. The business process passes the values of the priceQuote, availQuote, and taxRate business process variables to the myJoin method. The myJoin method invokes the query written in the XQuery language and stored in the myJoin.xq file. The query merges all the price, availability, and tax rate information into a single set of XML data and returns the result as the return value of the myJoin method. The data returned from this myJoin method is valid to the XML Schema in the Quote.xsd file. After the myJoin method is invoked, the **Combine Price and Avail Quotes** node assigns the resulting XML data to the Quote business process variable.

## Tutorial Goals

The tutorial provides steps to create and test a transformation using the graphical environment provided in

WebLogic Workshop. Specifically, in this tutorial you will create the following:

- The **MyTutorialJoin** Transformation file.
- The myJoin Transformation method in the **MyTutorialJoin** Transformation file.
- The query invoked by the myJoin Transformation method. This query is stored in the XQ file called myJoin.xq.

## Steps in This Tutorial

Follow the steps in this tutorial to create and test a data transformation. Specifically, the steps include:

### Step 1: Getting Started

Describes how to load the prepackaged Tutorial Process Application.

### Step 2: Building the Transformation

Provides a step-by-step procedure to create and select source and target types for a Transformation method.

### Step 3: Mapping Elements and Attributes

Provides a step-by-step procedure to create mappings between source and target elements and attributes in a Transformation method.

### Step 4: Mapping Repeating Elements Creating a Join

Provides a step-by-step procedure to add a join between repeating elements to the Transformation method.



# Portal Tutorials

The following tutorials highlight the processes and features of portal development with the WebLogic Workshop Portal Extensions.

[Building Your First Portal](#)

[Changing a Portal's Look & Feel and Navigation](#)

[Showing Personalized Content in a Portlet](#)

[Creating a Login Control Page Flow](#)

[Using Page Flows Inside Portlets](#)

[Related Topics](#)

[Portal Samples](#)

# Tutorial: Building Your First Portal

This tutorial guides you through the brief process of creating a portal with working portlets. The tutorial takes about 15 minutes to complete.

## Tutorial Goals

At the end of this tutorial you will have built a fully functional portal in a short time with little effort.

## Tutorial Overview

The WebLogic Workshop Portal Extensions include a graphical Portal Designer that lets you surface application functionality easily and quickly in a sophisticated portal interface. Sample portlets included with the WebLogic Workshop Portal Extensions provide instant, reusable functionality for a portal, as this tutorial illustrates.

## Steps in This Tutorial

Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server — 5 minutes

In this step you start WebLogic Workshop, open the sample portal application, and start WebLogic Server.

Step 2: Create a Portal – 10 minutes

In this step you create a portal file for an existing portal project, add a page to the portal, add portlets to the pages, and view your finished portal.

Click the following arrow to navigate through the tutorial:



# Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server

In this step, you will start the development environment for building a portal.

The tasks in this step are:

- Start WebLogic Workshop
- Open the Sample Portal Application
- Start WebLogic Server

## Start WebLogic Workshop

- **Windows** – Choose *Start-->Programs-->BEA WebLogic Platform 8.1-->WebLogic Workshop*. The start script is located in the <BEA\_HOME>/weblogic81/workshop directory.

## Open the Sample Portal Application

The portal application you open contains all the necessary portal resources to complete the tutorial. In Java 2 Enterprise Edition (J2EE) terms, this application is an enterprise application that contains Web applications and related resources. Each Web application can contain multiple portals.

1. In the WebLogic Workshop menu, choose *File-->Open-->Application*.
2. In the Open *Workshop Application* dialog, select the <BEA\_HOME>\weblogic81\samples\portal\portalApp\portalApp.work file and click *Open*.

The portalApp application directory tree appears in the *Application* window.

## Start WebLogic Server

To develop portals and portal applications in WebLogic Workshop, WebLogic Server must be running on your development machine. For this tutorial, you will start the domain server used by the WebLogic Portal samples. The portalApp application you opened in the previous step contains all the necessary server configuration settings. To start WebLogic Server:

- In the WebLogic Workshop menu, choose *Tools-->WebLogic Server-->Start WebLogic Server*.

On Windows systems, you can bring up the command window from the Windows task bar to watch the startup progress. When the server starts, the WebLogic Workshop status bar shows the message "Server Running."

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



## Step 2: Create a Portal

In this step, you will create a portal file for an existing portal project, add a page to the portal, add portlets to the pages, and view your finished portal.

The tasks in this step are:

- Create a Portal File
- Add a Page to the Portal
- Add a Portlet to the Portal
- View the Portal

### Create a Portal File

A portal file is an XML file that contains all configuration information for a portal. The XML file is rendered graphically in the Portal Designer of the WebLogic Workshop Portal Extensions. As you build the portal in the graphical interface, the XML is generated automatically behind the scenes. To create a portal file:

1. In the **Application** window, right-click the *sampleportal* project and choose **New-->Portal**.
2. In the **New File** dialog, enter my.portal in the **File name** field. You must keep the file extension.
3. Click **Create**.

The new file is added to the sampleportal project, and a new portal file appears in the Portal Designer. The new portal has a header, footer, and a main body containing a default book and page. The Document Structure window displays the component hierarchy.

### Add a Page to the Portal

A new portal already contains a page. In this step you will add a second page to your portal and rename the tabs of both pages.

1. In the **Palette** window, drag the **Page** control into the Portal Designer next to the existing page tab. A page tab called "New Page" appears.
2. Click the **New Page** tab.
3. In the **Property Editor** window, change the page's **Title** property to My Page 2.
4. In the **Portal Designer**, click the **Page 1** tab.
5. In the **Property Editor** window, change the page's **Title** property to My Page 1 and press **Enter**.

### Add a Portlet to the Portal

In this step you will add a pre-built sample portlet from the sampleportal project to each page.

1. In the Portal Designer, click the **My Page 1** tab.
2. In the **Data Palette** window, drag the **Login to Portal** portlet into the left placeholder of **My Page 1**.
3. In the **Portal Designer**, click the **My Page 2** tab.
4. In the **Data Palette** window, drag the **RSS News Feed** into the left placeholder of **My Page 2** and the **Dev2Dev** portlet into the right placeholder.

**Note:** The RSS News Feed portlet requires an Internet connection to access the content feed.

5. Save the portal file.



### View the Portal

You can view your portal with the WebLogic Test Browser or with your default browser.

- **WebLogic Test Browser** – In the WebLogic Workshop toolbar, click the **Start** button (or press **Ctrl+F5**). Navigate between the portal pages using the **My Page 1** and **My Page 2** links.
- **Default Browser** – In the WebLogic Workshop menu, choose **Portal-->Open Current Portal**. Navigate between the portal pages using the **My Page 1** and **My Page 2** links.

Congratulations! You have built a portal.

### What You Can Do Next

The portal development lifecycle involves development with the WebLogic Workshop Portal Extensions and administration with the WebLogic Administration Portal. The tutorial you have just completed covers the development phase. The following steps instructions for starting a tutorial that covers the administration phase. The second phase of the portal-building process involves administering the portal you have created. To run a companion tutorial for administering a portal:

1. In the WebLogic Workshop menu, choose **Portal-->Open Portal Administration**.
2. Log in with Username: **weblogic** Password: **weblogic**.
3. When the Administration Portal appears, click **Show Help** in the upper left corner of the window.
4. In the Help window, click the **Tutorials** tab, select **Build Your First Portal**, and follow the tutorial.

### Related Topics

Developing Portal Applications

Building Portlets

Portal Samples

Click the following arrow to navigate through the tutorial:



# Tutorial: Changing a Portal's Look & Feel and Navigation

This tutorial shows you how easy it is to modify the look and feel of a portal and modify the navigation style used for the portal pages. The tutorial takes about 10 minutes to complete.

## Tutorial Goals

At the end of this tutorial you will have changed the look and feel and page navigation style of a portal.

## Tutorial Overview

This tutorial involves modifying two elements of a portal's physical appearance and behavior: look and feel, and navigation.

The WebLogic Workshop Portal Extensions provide a flexible, extensible architecture for controlling the look and feel and page navigation in a portal. A portal look and feel is made up of a skin (graphics, a cascading style sheet, and JavaScript functions) and skeletons (JSP files that determine the rendering the physical boundaries of individual portal components, such as desktops, pages, and portlets).

Ultimately the look and feel of a portal, and its navigation style, are determined by the portal administrator and end users. All look and feel and navigation resources that are available to developers in the WebLogic Workshop Portal Extensions are also available to delegated administrators in the WebLogic Administration Portal and to end users in the Visitor Tools when the portal is put into production.

When an administrator creates a desktop based on a .portal file in the WebLogic Administration Portal, that portal is decoupled from the development environment and can be modified as needed by the administrator, and in turn by end users when the portal is put into production. The initial look and feel and navigation settings you provide in development serve as the default settings for portal administrators and end users. This tutorial merely shows how easy it is to change look and feel and navigation elements in the development environment.

## Steps in This Tutorial

Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server — 5 minutes

In this step you start WebLogic Workshop, open the sample portal application, and start WebLogic Server.

Step 2: Change the Portal Look and Feel and Navigation — 5 minutes

In this step you change a portal's look and feel and page navigation style.

Click the following arrow to navigate through the tutorial:



# Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server

In this step, you will start the development environment for working with portals.

The tasks in this step are:

- Start WebLogic Workshop
- Open the Sample Portal Application
- Start WebLogic Server

## Start WebLogic Workshop

- **Windows** – Choose *Start-->Programs-->BEA WebLogic Platform 8.1-->WebLogic Workshop*. The start script is located in the <BEA\_HOME>/weblogic81/workshop directory.

## Open the Sample Portal Application

The portal application you open contains all the necessary portal resources to complete the tutorial. In Java 2 Enterprise Edition (J2EE) terms, this application is an enterprise application that contains Web applications and related resources.

1. In the WebLogic Workshop menu, choose *File-->Open-->Application*.
2. In the Open *Workshop Application* dialog, select the <BEA\_HOME>\weblogic81\samples\portal\portalApp\portalApp.work file and click *Open*.

The portalApp application directory tree appears in the *Application* window.

## Start WebLogic Server

To develop portals and portal applications in WebLogic Workshop, WebLogic Server must be running on your development machine. For this tutorial, you will start the domain server used by the WebLogic Portal samples. The portalApp application you opened in the previous step contains all the necessary server configuration settings. To start WebLogic Server:

- In the WebLogic Workshop menu, choose *Tools-->WebLogic Server-->Start WebLogic Server*.

On Windows systems, you can bring up the command window from the Windows task bar to watch the startup progress. When the server starts, the WebLogic Workshop status bar shows the message "Server Running."

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



## Step 2: Change the Portal Look & Feel and Navigation

In this step, you will change a portal's look and feel, change the navigation scheme for the portal pages, and view the results.

The tasks in this step are:

- Open the Sample Portal
- View the Portal in a Browser
- Add a Book to the Main Page Book
- Change the Portal's Look and Feel
- Change the Portal's Navigation
- View the Modified Portal in a Browser

### Open the Sample Portal

The Sample portal is included with the WebLogic Workshop Portal Extensions. It contains a set of pre-built sample portlets you can reuse in your own portals. You will change the Sample portal's look and feel and page navigation scheme.

1. In the *Application* window, expand the *sampleportal* folder.
2. Double-click *sample.portal*. The portal file opens in the Portal Designer.

### View the Portal in a Browser

In this step, view the existing look and feel and navigation of the Sample portal to see what it looks like before you change it.

1. In the WebLogic Workshop toolbar, click the *Start* button (or press *Ctrl+F5*) to view the portal in the Workshop Test Browser.

Click the page navigation tabs and note the behavior.

2. Close the WebLogic Test Browser.

### Add a Book to the Main Page Book

1. In the *Palette* window, drag the *Book* control into the Main Page Book area of the Portal Designer, next to *My Page*. A new book appears in the main page book.

Do NOT drag the Book control into a placeholder on one of the pages.

2. Click the *New Book* tab. Notice that it automatically contains a new page as well.

### Change the Portal's Look and Feel

1. In the *Document Structure* window, click *Desktop*.
2. In the *Property Editor* window, change the *Look and Feel* property from *avitek* to *Classic*.

### Change the Portal's Navigation

1. In the *Document Structure* window, click *Book: Main Page Book*.

## WebLogic Workshop Tutorials

2. In the **Property Editor** window, change the Navigation property from **Single Level Menu** to **Multi Level Menu**. Multi-level menus display links to nested books and pages with drop-down navigation.
3. Save the portal file.

View the Modified Portal in a Browser

You can view the portal with the WebLogic Test Browser or with your default browser.

- **WebLogic Test Browser** – In the WebLogic Workshop toolbar, click the **Start** button (or press **Ctrl+F5**).
- **Default Browser** – In the WebLogic Workshop menu, choose **Portal-->Open Current Portal**.

Use the page and book navigation links and note the different navigation behavior. The New Book link should provide a drop-down navigation menu to access the page it contains.

Congratulations! You have changed a portals look and feel and page navigation style.

Related Topics

Changing a Page Layout

Creating Look and Feels

Click the following arrow to navigate through the tutorial:



# Tutorial: Showing Personalized Content in a Portlet

This tutorial has you develop personalization functionality and surface that functionality in a portlet. The tutorial takes about 30 minutes to complete.

## Tutorial Goals

At the end of this tutorial you will have created a portlet that displays different personalized content to different users.

## Tutorial Overview

The WebLogic Workshop Portal Extensions include a graphical Portal Designer that lets you create and surface application functionality easily and quickly in a sophisticated portal interface. In this tutorial you will use the WebLogic Workshop Portal Extensions and the WebLogic Administration Portal to create the users, properties, content, and code that results in a complete personalization solution. You will perform minimal JSP development using Portal JSP tags in an easy-to-use, integrated interface.

## Steps in This Tutorial

Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server — 5 minutes

In this step you start WebLogic Workshop, open the sample portal application, and start WebLogic Server.

Step 2: Create a User Profile Property Set — 5 minutes

In this step you will create the user properties that uniquely identify authenticated users and determine what the users see.

Step 3: Create Two Users — 5 minutes

In this step you will create two test users and assign property values to them.

Step 4: Load Content— 5 minutes

In this step you will load content into the BEA Virtual Content Repository.

Step 5: Create a JSP — 1 minute

In this step you will create a JSP file.

Step 6: Add Content Selectors to the JSP — 10 minutes

In this step you will add JSP tags to the JSP to enable personalization.

Step 7: Create a Portlet with the JSP — 1 minute

In this step you will create a portlet out of the JSP and add it to an existing portal.

## WebLogic Workshop Tutorials

### Step 8: Test the Personalized Portlet — 5 minutes

In this step you will log in as both users to see the personalized portlet in action.

Click the following arrow to navigate through the tutorial:



# Step 1: Start WebLogic Workshop, Open an Application, and Start WebLogic Server

In this step, you will start the development environment for working with portals.

The tasks in this step are:

- Start WebLogic Workshop
- Open the Sample Portal Application
- Start WebLogic Server

## Start WebLogic Workshop

- **Windows** – Choose *Start-->Programs-->BEA WebLogic Platform 8.1-->WebLogic Workshop*. The start script is located in the <BEA\_HOME>/weblogic81/workshop directory.

## Open the Sample Portal Application

The portal application you open contains all the necessary portal resources to complete the tutorial. In Java 2 Enterprise Edition (J2EE) terms, this application is an enterprise application that contains Web applications and related resources.

1. In the WebLogic Workshop menu, choose *File-->Open-->Application*.
2. In the Open *Workshop Application* dialog, select the <BEA\_HOME>\weblogic81\samples\portal\portalApp\portalApp.work file and click *Open*.

The portalApp application directory tree appears in the *Application* window.

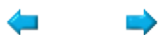
## Start WebLogic Server

To develop portals and portal applications in WebLogic Workshop, WebLogic Server must be running on your development machine. For this tutorial, you will start the domain server used by the WebLogic Portal samples. The portalApp application you opened in the previous step contains all the necessary server configuration settings. To start WebLogic Server:

- In the WebLogic Workshop menu, choose *Tools-->WebLogic Server-->Start WebLogic Server*.

On Windows systems, you can bring up the command window from the Windows task bar to watch the startup progress. When the server starts, the WebLogic Workshop status bar shows the message "Server Running."

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





## Step 2: Create a User Profile Property Set

In this step, you will create a user profile property set. This property set will contain properties that can be set for all users. Each user can have different property values. In this tutorial, property values determine the personalized content users see.

The tasks in this step are:

- Create a Property Set File
- Add a Property to the Property Set

### Create a Property Set File

1. In the **Application** window, expand the **data** project.
2. Right-click the **userprofiles** folder, and choose **New-->User Profile Property Set**.
3. In the **New File** dialog, enter **userpreferences.usr** in the **File name** field. You must keep the file extension.
4. Click **Create**. The **Property Set Designer** appears.

### Add a Property to the Property Set

1. In the **Palette** window, drag the **Single Restricted** icon into the Property Set Designer.
2. In the **Property Editor** window, enter the following text in the **Property Name** field: **Graphic Preference**.
3. Click the ellipsis icon [...] in the **Value(s)** field.
4. In the **Enter Property Value** dialog, enter **modern** in the top field and click **Add**. Enter **classic** in the top field and click **Add**.
5. Click **OK**.
6. Save and close the property set file.

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



## Step 3: Create Two Users

In this step, you will create two test users that have different preferences set in their user profiles. When you log in as each user at the end of the tutorial, each users will see different content displayed in the portlet based on their different user profile property values.

The tasks in this step are:

- Start the WebLogic Administration Portal
- Create Two Users and Set Their Profile Preferences

Start the WebLogic Administration Portal

1. In the WebLogic Workshop menu, choose **Portal**—>**Open Portal Administration**.
2. Log in with Username: **weblogic** Password: **weblogic**.
3. When the WebLogic Administration Portal appears, select **Users & Groups** under Users, Gropus, & Roles in the top Menu in the editor pane.

Create Two Users and Set Their Profile Preferences

1. In the Users & Groups resource tree, click **everyone (All Users)**.
2. Select the **Add Users** page in the editor pane.
3. Click **Create New User** in the main window.
4. In the Add a New User dialog that appears, enter modernuser, enter a password of password (which is the default), and click **Add New User**. A confirmation message appears at the top of the Add Users page.
5. Create a second user called classicuser with a password of password.
6. Select the **Edit Users** page.
7. In section **1** of the page, enter an asterisk (\*) in the Search field and click **Search**.
8. In section **2** of the page,click the **classicuser** name. The **Editing User** window appears.
9. Select the **Edit User Profile Values** page.
10. In the **Properties from property set** field, select **userpreferences**. This is the property set you created.
11. Expand the **Graphic Preference** property line. Change the Graphic Preference property value to **classic**, and click **Update Value**.
12. In the left resource tree, select **everyone (All Users)**.
13. In section **1** of the page, enter an asterisk (\*) in the Search field and click **Search**.
14. In section **2** of the page,click the **modernuser** name. The **Editing User** window appears.
15. On the **Edit User Profile Values** page, change the user's Graphic Preference to **modern**, and click **Update Value**.

You now have two users with different user profile preferences.

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



## Step 4: Load Content

In this step, you will load sample content into the Virtual Content Repository that will be used later in this tutorial. This step requires WebLogic Server to be running, as described in Step 1.

1. Open a command window and change to the following directory:

```
<BEA_HOME>\weblogic81\portal\bin
```

2. Enter the following command:

```
load_cm_data.cmd
```

or

```
sh load_cm_data.sh
```

The script loads sample content into the sample domain's Virtual Content Repository under the default "BEA Repository." You can view this sample content in the WebLogic Administration Portal using the following steps:

1. Choose **Portal**—>**Open Portal Administration** in WebLogic Workshop or by entering ***http://localhost:7001/portalAppAdmin*** in a browser.
2. Log in as weblogic/weblogic.
3. In the WebLogic Administration Portal, select ***Content*** in the top Menu in the editor pane.
4. In the left resource tree, expand the BEA Repository.

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



## Step 5: Create a JSP

In this step, you will create a Java Server Page (JSP) that will display the personalized content in a portlet.

The tasks in this step are:

- Create a JSP

Create a JSP

1. In WebLogic Workshop, right-click the *sampleportal* project in the *Application* window and choose *New-->JSP File*.
2. In the *New File* dialog, enter *myp13n.jsp* in the *File name* field. You must keep the file extension.
3. Click *Create*. A new JSP file appears.
4. Click the *Source View* tab at the bottom of the JSP Designer, and delete the text and tags that appear inside the *<body>* tag.
5. Click the *Design View* tab at the bottom of the JSP Designer.

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



## Step 6: Add Content Selectors to the JSP

In this step, you will add content selector JSP tags to the JSP, along with two other JSP tags to display the personalized content. In the process of adding the content selector JSP tags, you will also create content selectors (XML files) that contain the queries for retrieving content from the BEA Virtual Content Repository and the rules that trigger the queries to run.

The tasks in this step are:

- Add the <pz:contentSelector> Tag to the JSP
- Add the <es:forEachInArray> Tag to the JSP
- Add an Image Tag
- Create a Second Content Selector

Add the <pz:contentSelector> Tag to the JSP

1. In the JSP Designer, make sure the **Design View** tab at the bottom of the window is selected.
2. In the **Palette** window, locate the **Portal Personalization** category and drag the **Content Selector** tag into the JSP Designer.
3. In the **Property Editor** window, enter nodes in the **id** field.
4. In the **rule** field, click the ellipsis icon [...]. The **Select content selector** dialog appears.
5. Click **New content selector**, and click **Select**.
6. In the **New content selector** dialog, enter **modern** and click **OK**.
7. In the **Property Editor** window, click the → icon in the rule field. The **Content Selector Designer** appears.

Now you will define the query the content selector uses to retrieve content from the Virtual Content Repository.

8. In the **Content Selector Designer**, click the **[empty content search]** link. The **Content Search** window appears.
9. In the **Property set** field, select **Standard**.
10. In the **Property** field, make sure **cm\_binaryName** is selected, and click **Add**. The **Content Search Values** window appears.
11. In the **Comparison** field, select **contains**.
12. In the **Value** field, enter **college** and click **Add**.
13. Click **OK**. You are returned to the Content Search window. Click **OK**.
14. In the **Available Conditions** area of the Content Selector Designer, deselect any checkboxes that are selected, then select **The visitor has specific characteristics**.
15. In the top of the **Content Selector Designer**, click the **[characteristics]** link. The **Visitor Characteristics** window appears.
16. In the **Visitor property set** field, select **userpreferences**.
17. In the **Visitor property** field, make sure **Graphic Preference** is selected, and click **Add**. The **Visitor Characteristic Values** window appears.
18. In the **Comparison** field, make sure **is equal to** is selected.
19. In the **Value** field, select **modern**, click **Add**, and click **OK**.
20. In the **Visitor Characteristics** window, click **OK**.
21. Save and close the file.

Add the <es:forEachInArray> Tag to the JSP

1. Switch to the *myp13n.jsp* file if you are not already there.
2. In the **Palette** window, locate the **Portal Utilities** category, and drag the **For Each In Array** tag into the JSP Designer, to the right of the <pz:contentSelector> tag.
3. Click the **Source View** tab in the JSP designer and set the following properties on the For Each In Array tag:

```
<es:forEachInArray array="<%=nodes%>" id="node" type="com.bea.content.Node">
```

### Add an Image Tag

1. Switch back to **Design View** in the JSP designer.
2. In the **Palette** window, locate the **HTML** category, and drag the **Image** tag on top of the forEachInArray tag. In the ImageWizard dialog that appears, click **OK**. This should put the image tag inside the forEachInArray tag.
3. Go back to **Source View** in the JSP designer and add the following path to the image:  
">

### Create a Second Content Selector

1. In **Source View**, copy the block of JSP tags and the image reference you created, and paste the block just below itself.
2. In the copy of the JSP tags, change the <pz:contentSelector> tag's **rule** attribute to classic. You should now have two content selector blocks of code. The only difference between the two is the **rule** attribute value. One value is modern and the other is classic.
3. Switch to **Design View** in the JSP designer.
4. Select the second <pz:contentSelector> tag. It appears as **classic**.
5. In the **Property Editor** window, click the ellipsis icon [...] in the **rule** field. The **Select content selector** window appears.
6. Click **New content selector**, and click **Select**. The **New content selector** window appears.
7. Enter the name classic, and click **OK**.
8. In the **Property Editor** window, click the → icon in the **rule** field. The **Content Selector Designer** appears.
9. In the Content Selector Designer, click the [empty content search] link. The **Content Search** window appears.
10. In the **Property set** field, select **Standard**.
11. In the **Property** field, make sure **cm\_binaryName** is selected, and click **Add**. The **Content Search Values** window appears.
12. In the **Comparison** field, select **contains**.
13. In the **Value** field, enter IRACampaign and click **Add**.
14. Click **OK**. You are returned to the Content Search window. Click **OK**.
15. In the **Available Conditions** area of the Content Selector Designer, deselect any checkboxes that are selected, then select **The visitor has specific characteristics**.
16. In the top of the **Content Selector Designer**, click the [characteristics] link. The **Visitor Characteristics** window appears.
17. In the **Visitor property set** field, select **userpreferences**.
18. In the **Visitor property** field, make sure **Graphic Preference** is selected, and click **Add**. The **Visitor Characteristic Values** window appears.
19. In the **Comparison** field, make sure **is equal to** is selected.
20. In the **Value** field, select **classic**, click **Add**, and click **OK**.
21. In the **Visitor Characteristics** window, click **OK**.
22. Save and close the content selector file and the JSP file.

## WebLogic Workshop Tutorials

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



## Step 7: Create a Portlet with the JSP

In this step, you will drag the JSP into an existing portal to create a portlet out of it with the Portlet Wizard.

The tasks in this step are:

- Create a Portlet by Adding the JSP to a Portal

Create a Portlet by Adding the JSP to a Portal

1. In the **Application** window, expand the *sampleportal* project, and double-click *sample.portal* to open it in the Portal Designer.
2. In the **Application** window, drag *myp13n.jsp* below the **Login** portlet in the Portal Designer, and click **Yes** in the Create Portlet dialog that appears. The **Portlet Wizard** appears.
3. In the **Portlet Details** window, enter the following text in the **Title** field: My Personalized Portlet.
4. Click **Finish**. The new portlet appears below the Login to Portal Portlet.
5. Save the portal file.

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





## Step 8: Test the Personalized Portlet

In this step, you will test the portlet to see personalization in action.

The tasks in this step are:

- View the Sample Portal
- Log in as One User and View the Portlet
- Log in as the Other User and View the Portlet

### View the Sample Portal

You can view the portal with the WebLogic Test Browser or with your default browser.

- **WebLogic Test Browser** – In the WebLogic Workshop toolbar, click the **Start** button (or press **Ctrl+F5**).
- **Default Browser** – In the WebLogic Workshop menu, choose **Portal-->Open Current Portal**.

### Log in as One User and View the Portlet

1. In the **Login** portlet, log in with **Username**: classicuser **Password**: password.
2. View the updated contents of the personalized portlet.
3. Click **Logout**.

### Log in as the Other User and View the Portlet

1. Log in with **Username**: modernuser **Password**: password.
2. View the contents of the personalized portlet.

Congratulations! You have created a portlet that uses personalization.

### What You Can Do Next

The portal development lifecycle involves development with the WebLogic Workshop Portal Extensions and administration with the WebLogic Administration Portal. The tutorial you have just completed covers the development phase. The following steps instructions for starting a tutorial that covers the administration phase. The second phase of the portal personalization process involves administering the content selectors you have created. To run a companion tutorial for administering content selectors:

1. In the WebLogic Workshop menu, choose **Portal-->Open Portal Administration**.
2. Log in with Username: **weblogic** Password: **weblogic**.
3. When the Administration Portal appears, click **Show Help** in the upper right corner of the window.
4. In the Help window, click the **Tutorials** tab, select **Add Personalized Content to Your Portal**, and follow the tutorial.

### Related Topics

Developing Personalized Applications

Assembling Portal Applications

<pz:contentSelector> Tag

<es:forEachInArray> Tag

Click the following arrow to navigate through the tutorial:



# Tutorial: Creating a Login Control Page Flow Using the Wizard

This tutorial shows you how easy it is to add a Portal Control into a Page Flow. The tutorial takes about 20 minutes to complete.

At the end of this tutorial you will have created a Page Flow that includes a Portal Control, and that can be placed inside a portlet to expose the login functionality.

## Tutorial Overview

This tutorial introduces the use of Portal Controls with Page Flows, and should help familiarize you with databinding in forms.

**NOTE:** For instructions on adding the User Login Control without the wizard, consult the User Login Control help.

## Steps in This Tutorial

Step 1: Create a Portal Control Page Flow — 10 minutes

In this step you create a new Page Flow, which includes selecting the Login Control.

Step 2: Place the Page Flow in a Portlet— 10 minutes

In this step the new Page Flow is used to create a new portlet.

Click the following arrow to navigate through the tutorial:

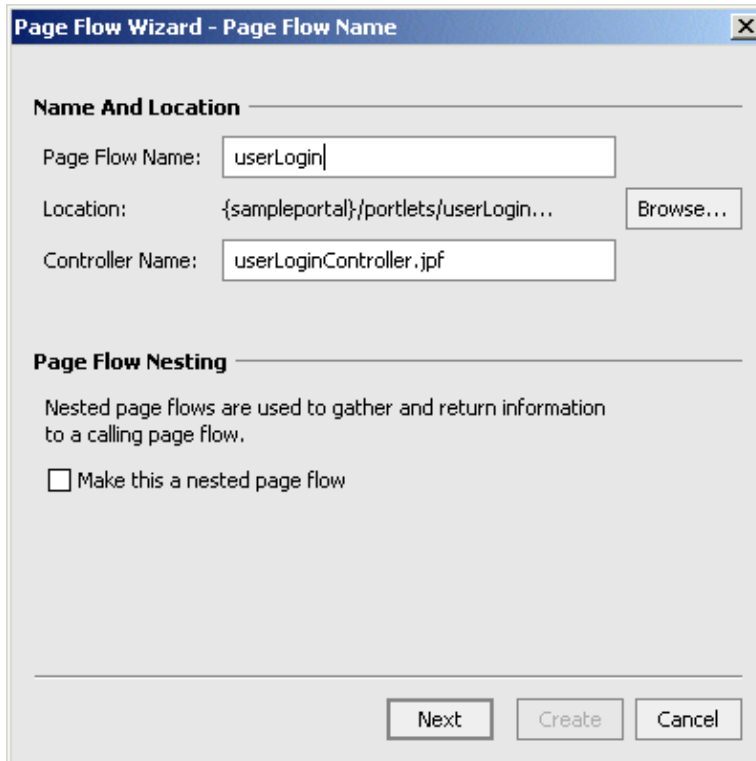


# Step 1: Create a Page Flow Using the Wizard

In this step, you will create a Portal Control Page Flow using the Page Flow Wizard.

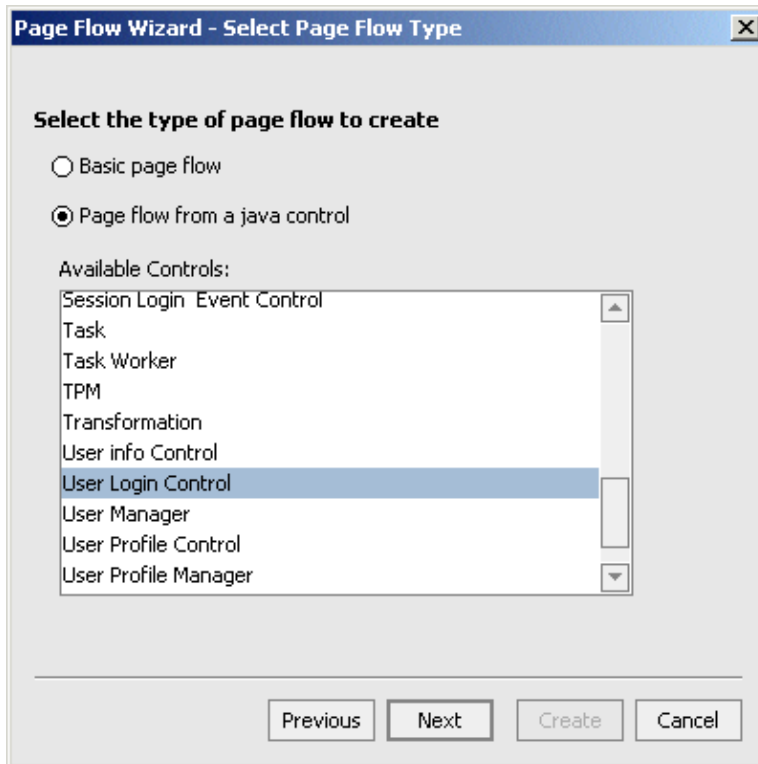
Create the Page Flow

1. From a Portal Web application project within WebLogic Workshop, create a new Page Flow by right-clicking on the portlets directory within the current Portal project and selecting **New** -> **Page Flow**.
2. The Page Flow Wizard appears. Name this new Page Flow "userLogin", and click **Next**.

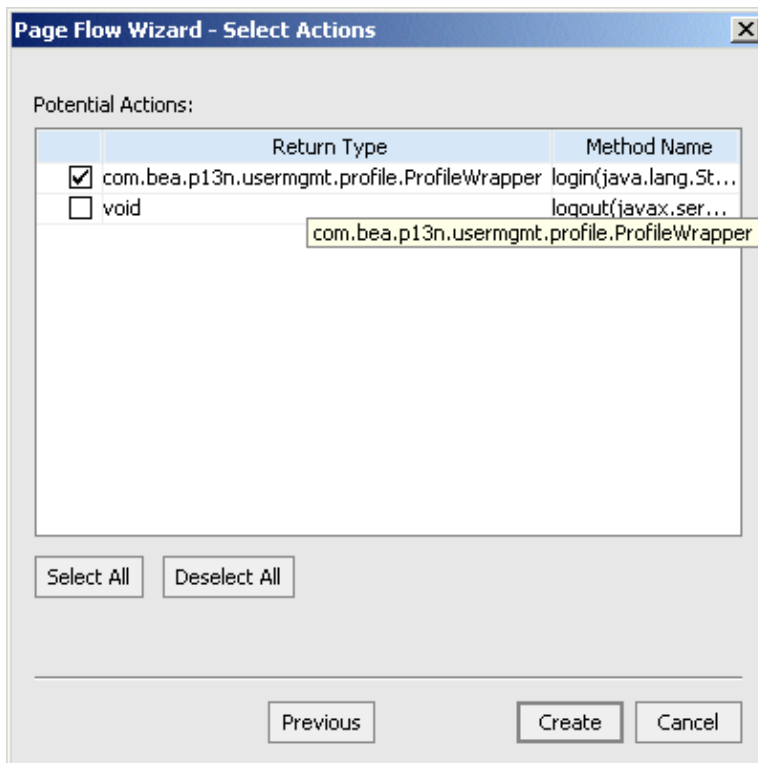


The screenshot shows the "Page Flow Wizard - Page Flow Name" dialog box. It has a title bar with a close button. The dialog is divided into two sections: "Name And Location" and "Page Flow Nesting". In the "Name And Location" section, there are three text input fields: "Page Flow Name:" containing "userLogin", "Location:" containing "{sampleportal}/portlets/userLogin...", and "Controller Name:" containing "userLoginController.jspf". A "Browse..." button is next to the Location field. In the "Page Flow Nesting" section, there is a text description: "Nested page flows are used to gather and return information to a calling page flow." and a checkbox labeled "Make this a nested page flow" which is currently unchecked. At the bottom of the dialog are three buttons: "Next", "Create", and "Cancel".

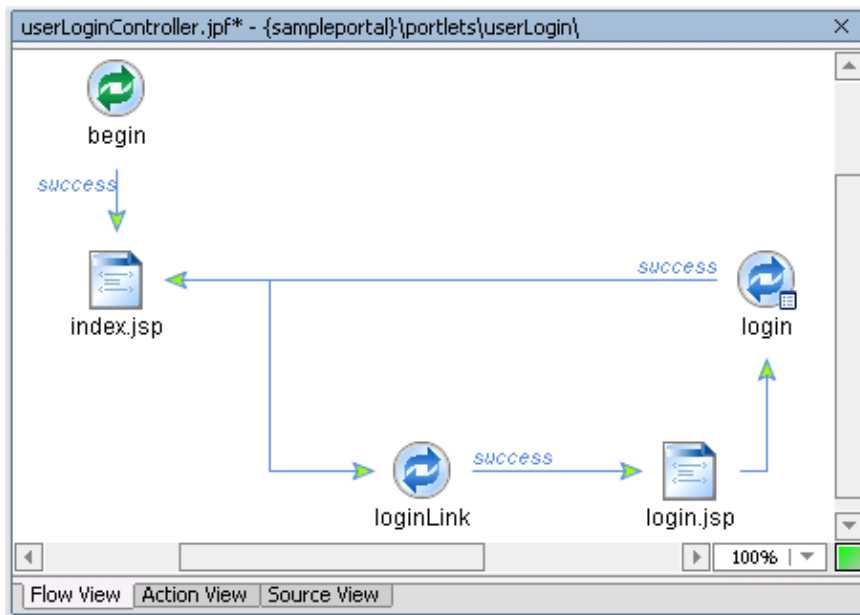
3. At the Select Page Flow Type prompt, select **Page Flow from a Java Control**, select the **User Login Control**, and Click **Next**.



4. From the Select Actions prompt, select the **login()** method, which returns the ProfileWrapper type. Click **Create**.



5. The resulting Page Flow should appear in Flow View.



6. Click **Ctrl+S** to save your work. Notice that in addition to the *userLoginController* page flow file, the wizard has also generated *index.jsp* and *login.jsp*.
7. The wizard has created a page flow with all the necessary elements in place, but they must be configured to fit your application. The pageflow needs to be modified to pass in the "request" to the methods. By default, the pageflow will create a formbean property for every parameter.
8. Open the *userLoginController.jpf* in **Source View**, and within the control method *login*, replace this code:

```
com.bea.p13n.usermgmt.profile.ProfileWrapper var = myControl.login(
aForm.username, aForm.password, aForm.request );
```

with this:

```
com.bea.p13n.usermgmt.profile.ProfileWrapper var = myControl.login(
aForm.username, aForm.password, super.getRequest() );
```

9. Now open the *userLoginController.jpf* in **Action View**, and from the Data Palette, open the control (it should be called *myControl*), and drag the *logout()* method into the Page Flow.

Double-clicking on the logout action will open the Source View to the code:

```
/**
 * @jpf:action
 */
protected Forward logout(LoginForm form)
{
    myControl.logout(form.getRequest());
    return new Forward( "success" );
}
```

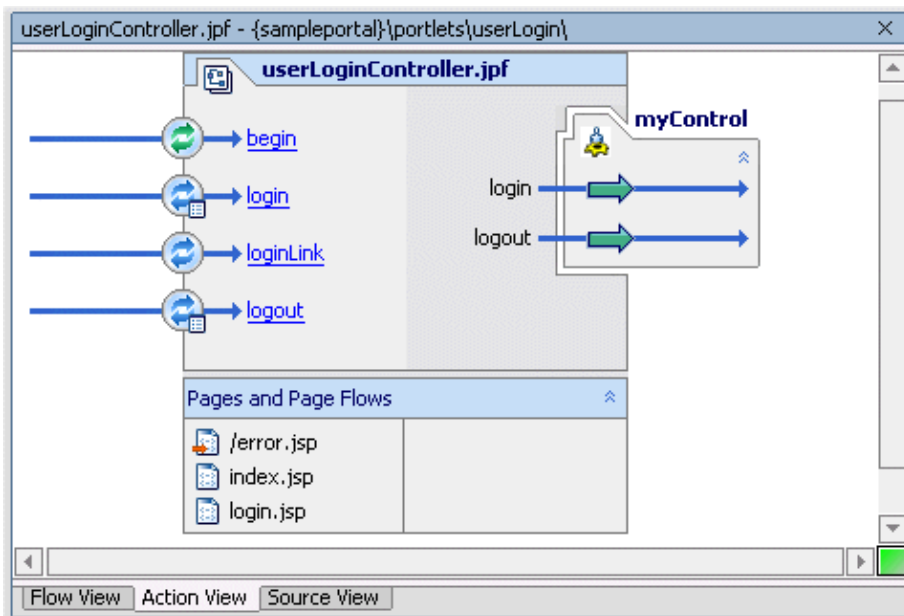
10. To the action method definition for this logout action, add the following line:

```
* @jpf:forward name="success" path="index.jsp"
```

11. In the same action code, change the (form.getRequest()); to (this.getRequest());
12. The resulting action method definition should look like this:

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="index.jsp"
 */
protected Forward logout(LoginForm form)
{
    myControl.logout(this.getRequest());
    return new Forward( "success" );
}
```

13. From the Action View, the userLoginController.jspf should now look like this:



14. Next, double-click on the **index.jsp** and open Source View. Just below the login form, add a logout button:

```
<netui:anchor action="loginLink">
login
</netui:anchor>
<netui:form action="logout">
<netui:button type="submit" value="logout" action="logout"></netui:button>
</netui:form>
```

The code in bold above places a button of type "submit" in the bottom of the page, associates it with the **logout** action, wraps it inside a netui:form that invokes the **action="logout"**.

15. Save the **index.jsp** file.

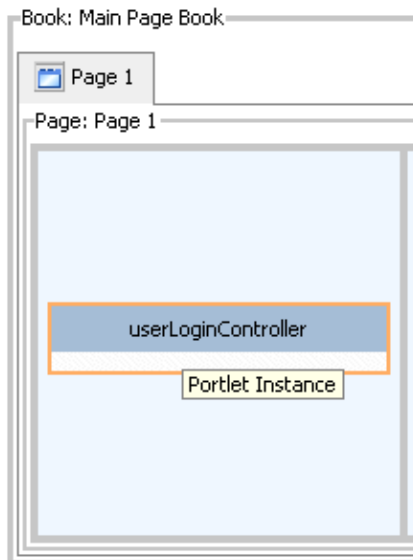
Click one of the following arrows to navigate through the tutorial:



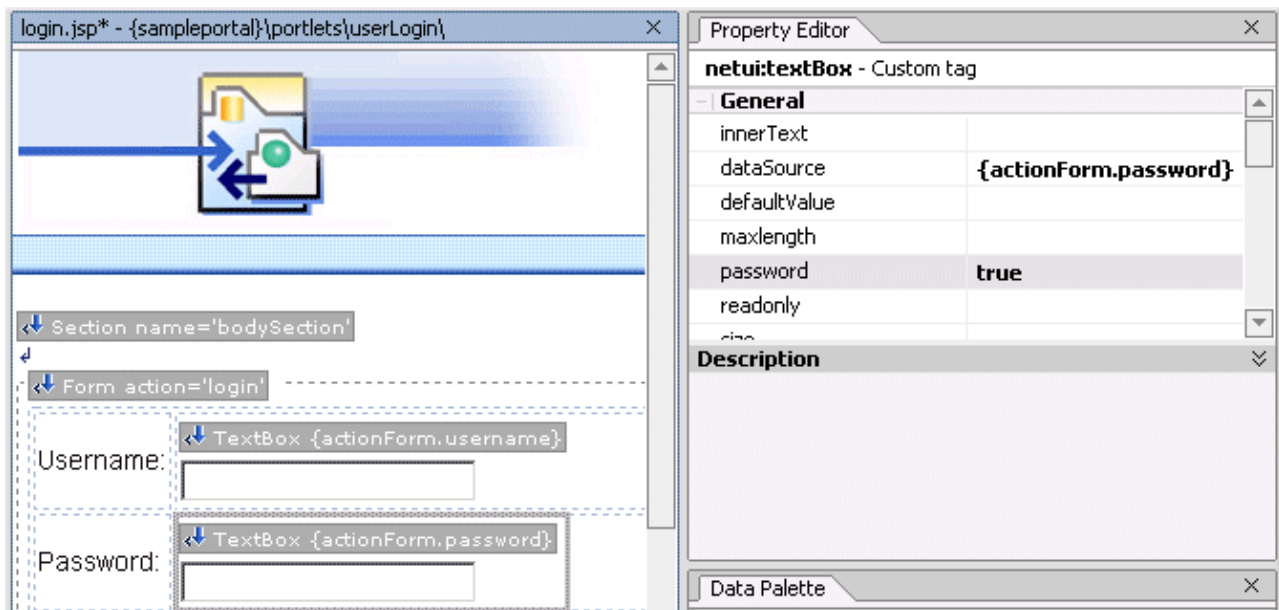
## Step 2: Place the Page Flow in a Portlet

In this step, you will use the Portlet Wizard to create a Portlet from the new Page Flow.

1. In the Portal Designer, drag the Page Flow file, called *userLoginController.jspf*, into a place holder in a portal. The Portlet Wizard appears.
2. From the Portlet Details screen, click **Finish** to create the *userLoginController* portlet and place it in the portal.



3. Open *login.jsp* in the Design View and select the Password text box. Then, in the Properties Editor, scroll down the list of General Properties to set Password equal to **Yes**. This will cause any input in this field to be masked.



4. For debugging purposes, the next step shows how to add some code to show which user is logged on. Open *index.jsp* in the Source View, adding the following code to the end, just before the last two lines:



## WebLogic Workshop Tutorials

```
<%Object res = request.getAttribute ( "results" );%><%= (res == null ? "<i>none</i><br/>" : ( res +  
" <br/>"))%>  
<% if (request.getRemoteUser() != null) { %>  
<BR>you are logged in as: <%=request.getRemoteUser()%>  
<br>  
<%  
}  
else  
{  
%>  
<BR>you are not logged in  
<%  
}  
%>
```

The index.jsp should read as follows:

```
<!--Generated by WebLogic Workshop-->  
<% @ page language="java"  
contentType="text/html;charset=UTF-8"%>  
<% @ taglib uri="netui-tags-databinding.tld"  
prefix="netui-data"%>  
<% @ taglib uri="netui-tags-html.tld" prefix="netui"%>  
<% @ taglib uri="netui-tags-template.tld"  
prefix="netui-template"%>  
<netui-template:template  
templatePage="/resources/jsp/template.jsp">  
<netui-template:setAttribute value="Index" name="title"/>  
<netui-template:section name="bodySection">  
<p class="pagehead">  
&nbsp;Page Flow: userLogin  
</p>  
<table width="100%" cellpadding="0" class="tablebody"  
cellspacing="0">  
<tr>  
<td valign="top">  
<table width="100%" class="tablebody">  
<tr class="tablehead">  
<td>Actions With No Parameters</td>  
</tr>  
</table>  
</td>  
<td valign="top">  
<table width="100%" class="tablebody">  
<tr class="tablehead">  
<td>Input Forms For Actions With Parameters</td>  
</tr>  
<tr>  
<td>  
<netui:anchor action="loginLink">
```

```

login
</netui:anchor>
<br>

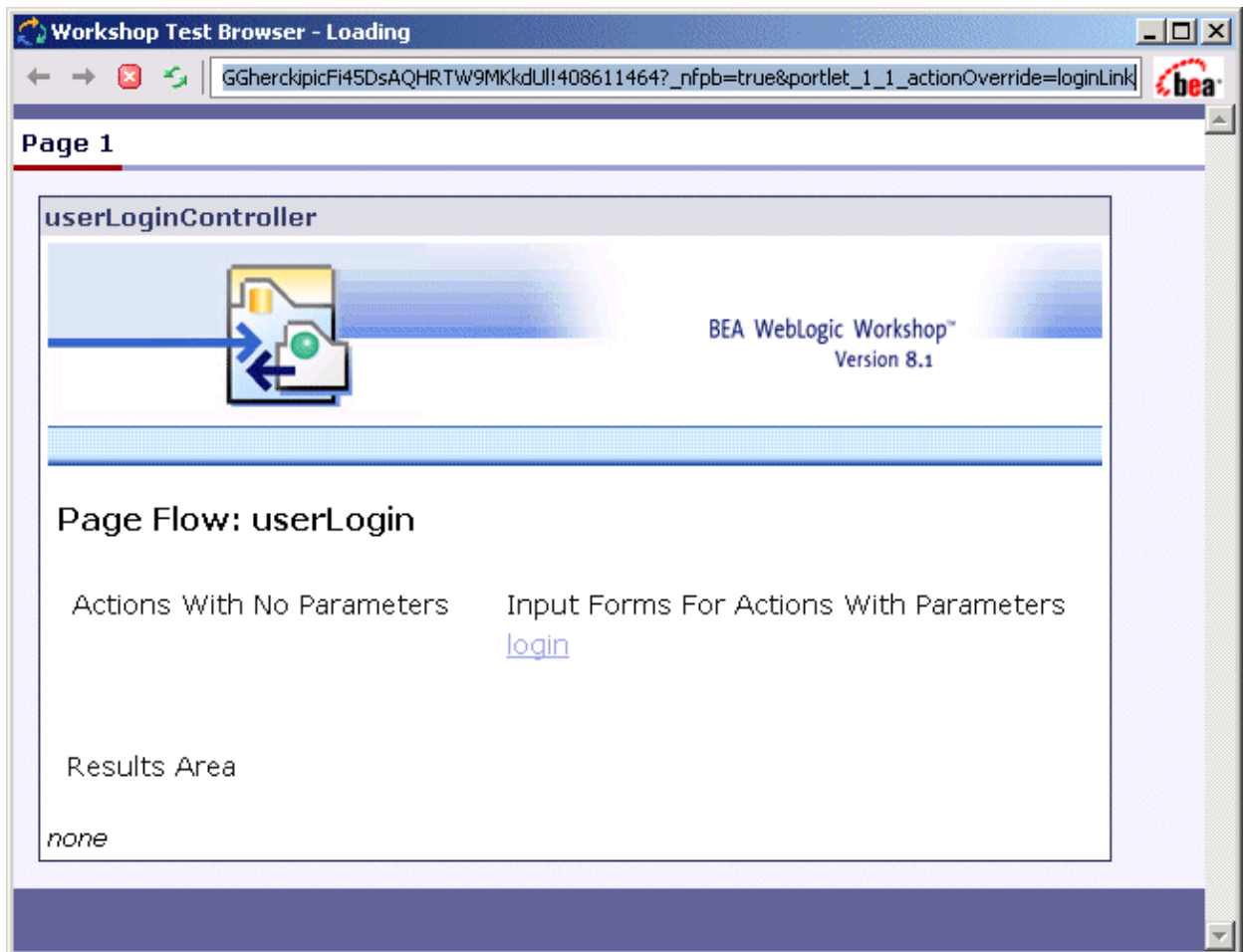
</td>
</tr>
</table>
<br/>
<br/>
</td>
</tr>
<tr class="tablehead">
<td align="left" colspan="2">
Results Area
</td>
</tr>
</table>
<br/>
<%Object res = request.getAttribute ( "results"
);%><%= (res == null ? "<i>none</i><br/>" : ( res +
"<br/>"))%>
<% if (request.getRemoteUser() != null) { %>
<BR>you are logged in as:
<%=request.getRemoteUser()%>
<br>
<netui:form action="logout">
<netui:button type="submit" value="logout"
action="logout"></netui:button>
</netui:form>

<%
}
else
{
%>
<BR>you are not logged in
<%
}
%>

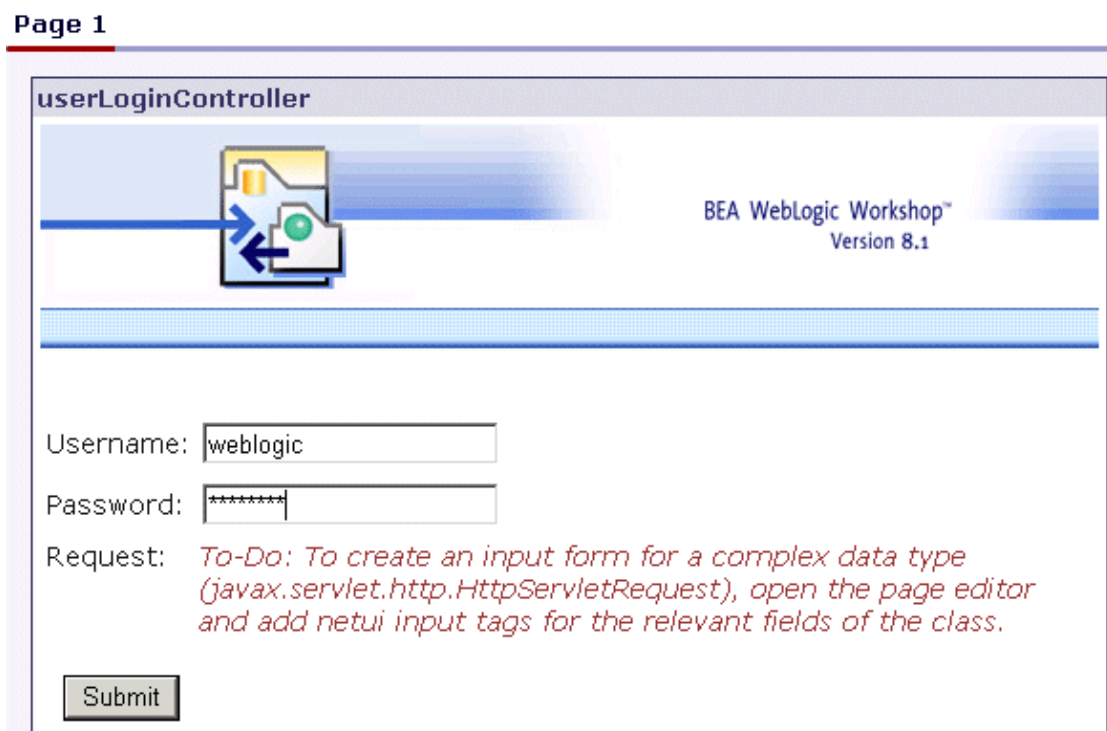
</netui-template:section>
</netui-template:template>

```

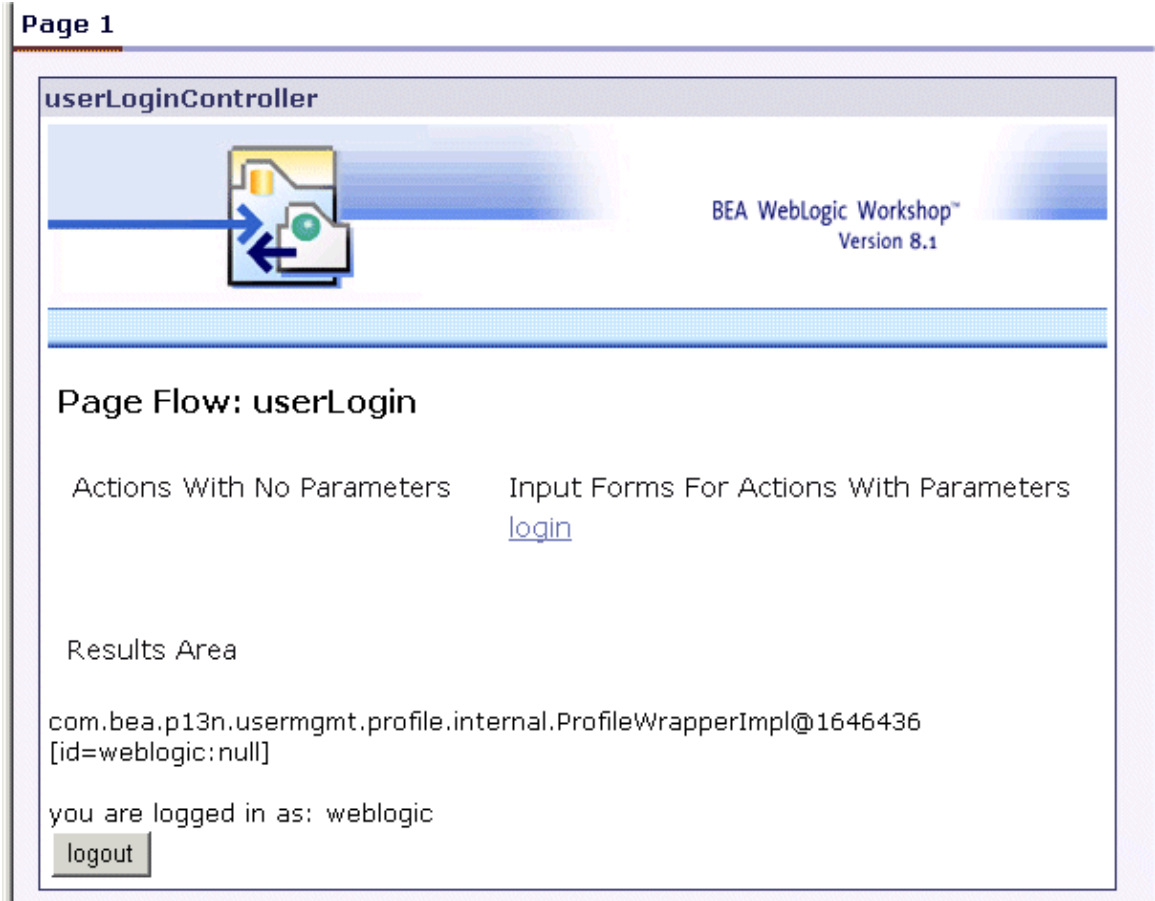
5. Click **Ctrl+S** to save your work.
6. In the WebLogic Workshop menu, choose **Portal-->Open Current Portal**.



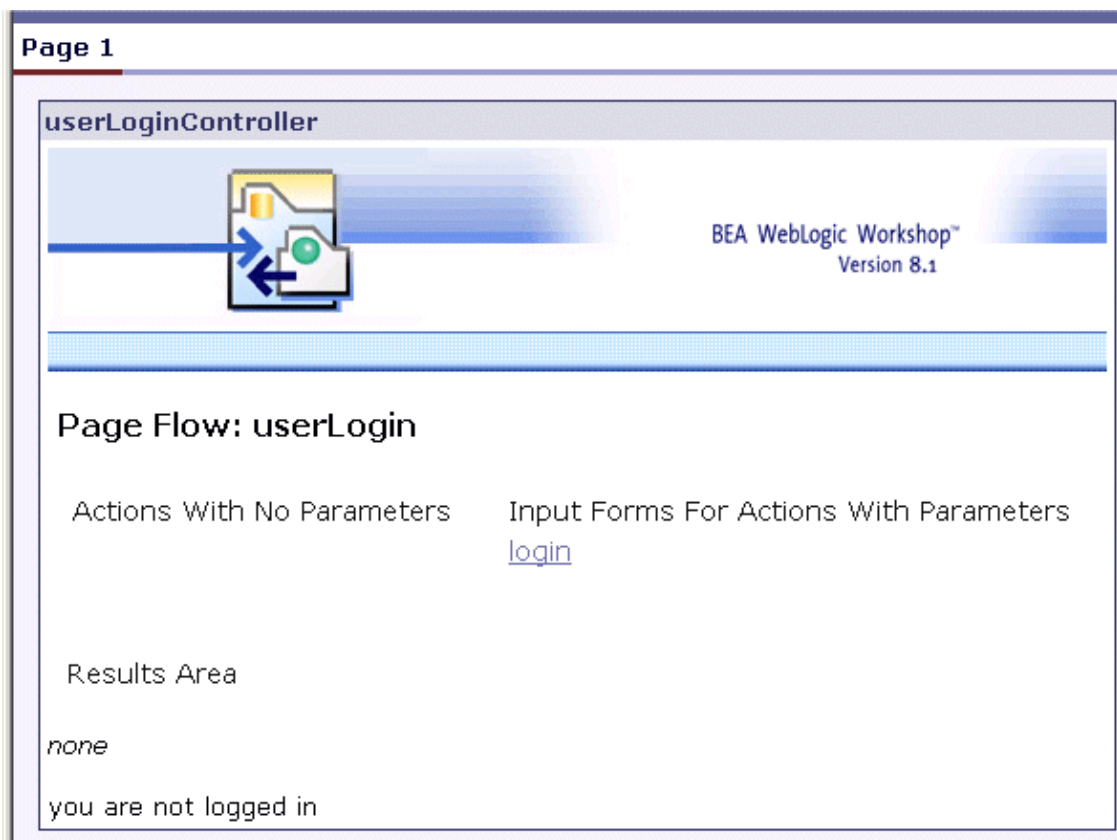
7. Click on Login, and at the login page, enter **weblogic/weblogic**. (The password field should be masked).



8. Verify that the next page displays the current logged-in user.



9. Click logout, and verify the page displays the "you are not logged in" message.



Congratulations! You have created a portlet that uses a Page Flow and an EJB control.

Related Topics

WebLogic Portal Tutorials

WebLogic Workshop Portal Extensions Samples

Click on the arrow below to go back in the tutorial:



# Tutorial: Using Page Flows Inside Portlets

This tutorial guides you through the process of learning how to use Page Flows inside portlets. The tutorial takes about 45 minutes to complete.

## Tutorial Goals

At the end of this tutorial you will have some familiarity with how to use Page Flows inside portlets.

## Tutorial Overview

The WebLogic Workshop Portal Extensions include a graphical Portal Designer that lets you surface application functionality easily and quickly in a sophisticated portal interface. Sample portlets included with the WebLogic Workshop Portal Extensions provide instant, reusable functionality for a portal, as this tutorial illustrates.

The portal development lifecycle involves development with the WebLogic Workshop Portal Extensions and administration with the WebLogic Administration Portal. This tutorial covers the development phase. After you have completed this tutorial, you will see instructions for starting a tutorial that covers the administration phase.

## Steps in This Tutorial

Step 1: Create a Portal Application — 5 minutes

In this step you Create a Portal Application, add a Portal Web Project, and add a Portal, and start WebLogic Server.

Step 2: Create a Simple Navigation Page Flow portlet – 10 minutes

In this step you create two simple .jpf files and their corresponding portlets.

Step 3: Create portlets that communicate. – 10 minutes

In this step you create a Page Flow portlet that listens to input from another Page Flow portlet.

Step 4: Create an EJB Control Page Flow portlet. – 10 minutes

In this step you create a Page Flow portlet, add a Personalization Control, and edit properties on the Control.

Click the following arrow to navigate through the tutorial:



# Step 1: Create a Portal Application

In this step, you will create a new Portal Application, add a Portal Web Project, and add a Portal.

The tasks in this step are:

- Create a new Portal Application
- Add a Portal

Create a new Portal Application

To create the necessary framework and resources for portal development, you must do one of two things:

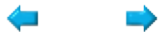
Create a new portal application and add a Portal Web Project to it.

Install Portal into an existing application and add a Portal Web Project to it.

Add a Portal

Create a Portal File to use as the layout framework to view your portlets. Name this *portal.portal*.

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



## Step 2: Create a Simple Navigation Page Flow

In this step, you will create a Page Flow that provides navigation from one JSP to another. Then, instead of designating the JSP files as the content url for the portlet, you will designate the .JPF file as the portlet's content node. The Page Flow supports JSP transitions within the portlet.

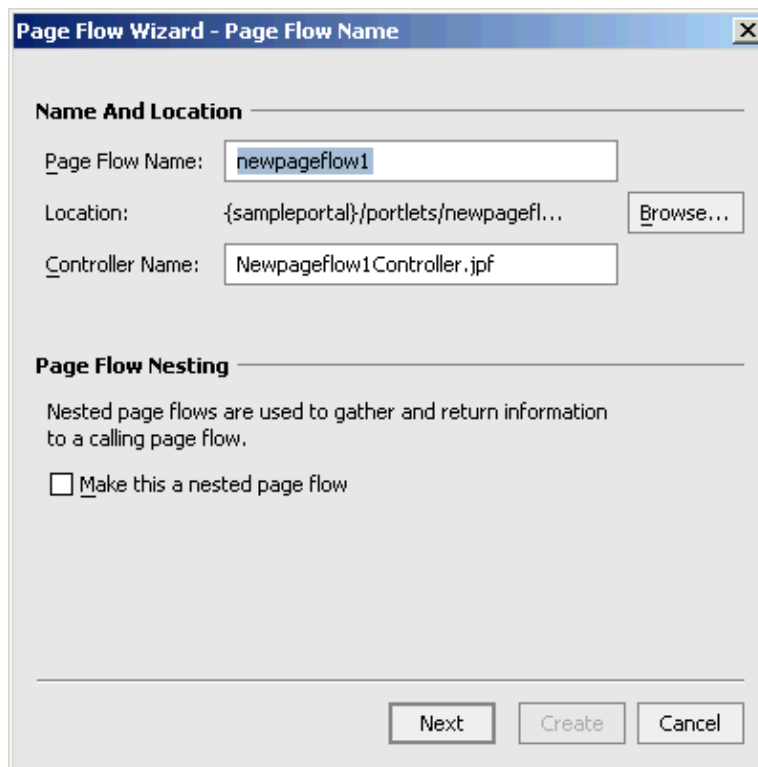
The tasks in this step are:

- Create a Page Flow Called simpleFlow
- Test the Page Flow in a portlet
- View the navigation portlets

Create a Page Flow called simpleFlow

1. Right-click a project in your web application and create a new folder; name the folder "portlets".
2. Right-click on the portlets folder and select **New>Page Flow**.

The Page Flow Name window appears.

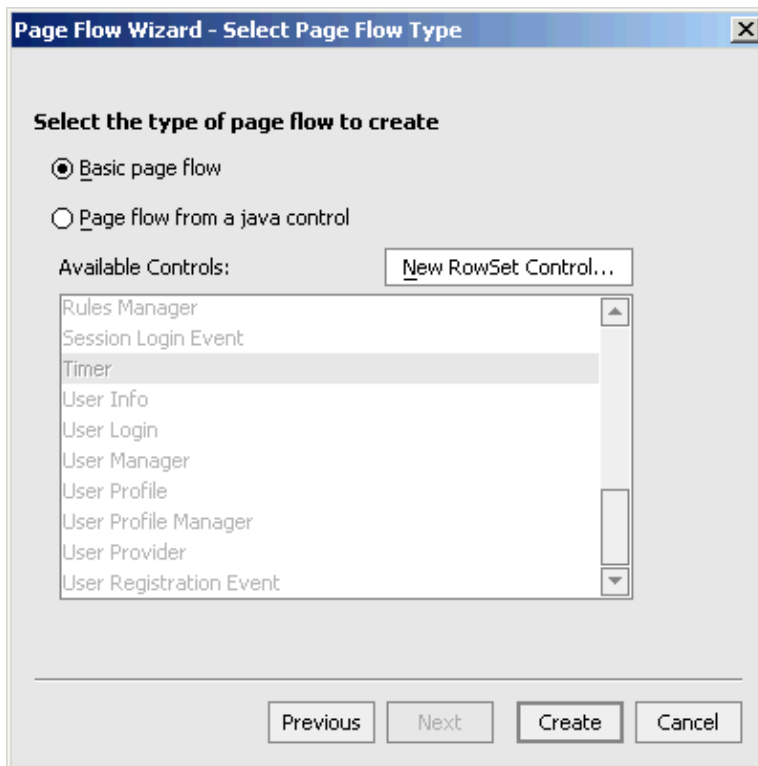


The screenshot shows the 'Page Flow Wizard - Page Flow Name' dialog box. It has a title bar with a close button. The dialog is divided into two sections: 'Name And Location' and 'Page Flow Nesting'. In the 'Name And Location' section, there are three text fields: 'Page Flow Name' with the value 'newpageflow1', 'Location' with the value '{sampleportal}/portlets/newpagefl...', and 'Controller Name' with the value 'Newpageflow1Controller.jspf'. There is a 'Browse...' button next to the Location field. In the 'Page Flow Nesting' section, there is a text box explaining that nested page flows are used to gather and return information to a calling page flow, and a checkbox labeled 'Make this a nested page flow' which is currently unchecked. At the bottom of the dialog, there are three buttons: 'Next', 'Create', and 'Cancel'.

3. Name the page flow "simpleFlow" and click **Next**.

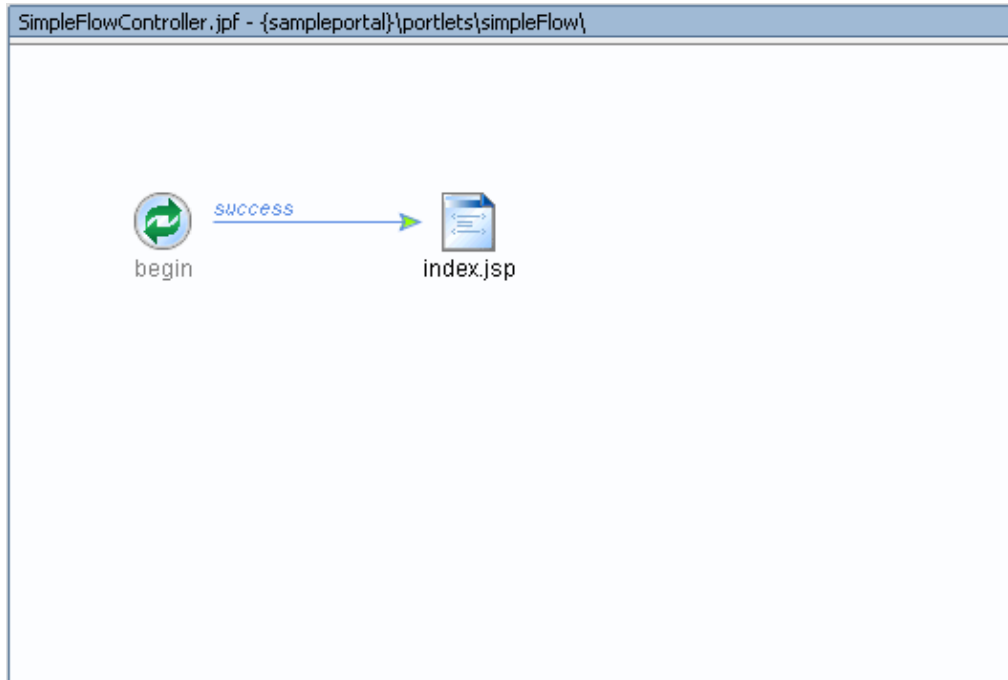
The Select Page Flow Type window appears.





4. Accept the default (**Basic page flow**) and click **Create**.

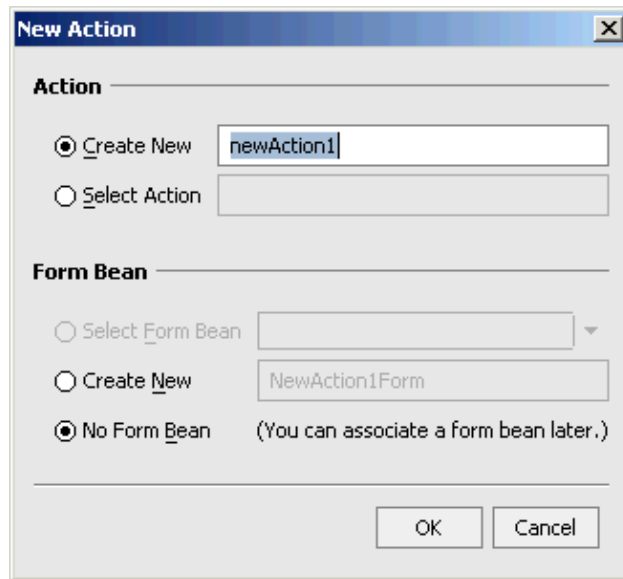
The page flow is generated and the PageFlow editor, in the Flow View, appears.



Note that the basic page flow has a **begin** action and an index.jsp presentation page.

5. Click **Page** in the palette, and drag a new page onto the PageFlow editor.
6. Name the page "page2.jsp".
7. Connect the two pages by doing the following:
  - a. Click **Action** in the palette and drag a new action onto the PageFlow editor.

The New Action window appears.



- b. Name the action "goToPage2" and click **OK**.

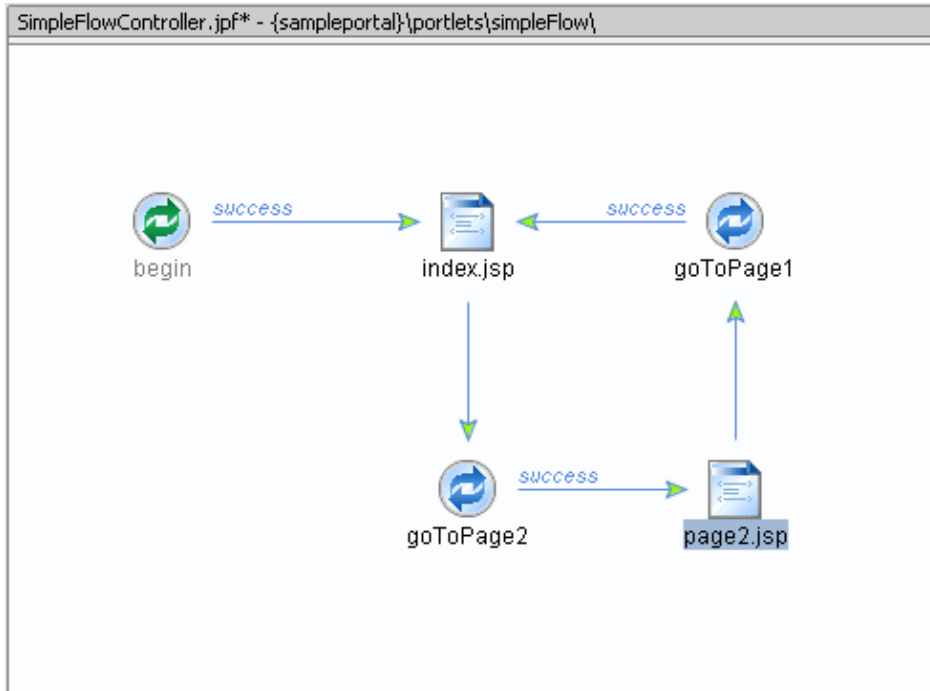
An icon for the *goToPage2* action appears in the Page Flow editor.

- c. Connect *index.jsp* to *goToPage2* by placing your pointer near the *index.jsp* icon, clicking to create the connection, and dragging to *goToPage2*.
- d. Connect *goToPage2* to *page2.jsp* (as described in step c., above). A red error indicator will appear under *index.jsp*. This will be fixed later in this procedure.
8. Create another action from the palette and name it "goToPage1" on the New Action window. Click **OK**.

The *goToPage1* action appears in the Page Flow editor.

9. Using the procedure described in step 7, connect the *goToPage1* action to *index.jsp* and then connect *page2.jsp* to *goToPage1*.
10. Correct the error indicators on *index.jsp* by doing the following:
  - a. Double-click *index.jsp* to open the JSP editor (ensure you are in the Source View).
  - b. Change the text within the <p> </p> elements to "PageFlow Page1".
  - c. Drag and drop the *goToPage2* action from the data palette onto *index.jsp*.
  - d. Save and close *index.jsp*.
11. Correct the error indicators *page2.jsp* by doing the following:
  - a. Double-click *page2.jsp* to open the JSP editor (ensure you are in the Source View).
  - b. Change the text within the <p> </p> elements to "PageFlow Page2".
  - c. Drag and drop the *goToPage1* action from the data palette onto *page2.jsp*.
  - d. Save and close *page2.jsp*.

Your page flow should now look like this:



12. Test the page flow in the test browser by doing the following:
  - a. Click the Start button (▶).

The page flow will appear in the Test Browser:



- b. Click *show goToPage2*.

The page2.jsp appears.

- c. Click *show goToPage1* to return to PageFlow page 1.

Test the Page Flow in a Portlet

1. Create a portal and drag simpleFlowController.jspf into it.

The Portlet Wizard appears.

2. Accept the Portlet Wizard defaults and click finish to create the portlet.
3. Save the portal and then view it in a browser by selecting .

JSP Examples

The JSPs should appear roughly as follows:

*index.jsp*

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
```

## WebLogic Workshop Tutorials

```
<head>
  <title>
    Web Application Page
  </title>
</head>
<body>
  <p>
    PageFlow page 1
    <netui:anchor action="goToPage2">show_goToPage2</netui:anchor>
  </p>
</body>
</netui:html>
```

### *page2.jsp*

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
  <head>
    <title>
      Web Application Page
    </title>
  </head>
  <body>
    <p>
      PageFlow Page 2
      <netui:anchor action="goToPage1">show_goToPage1</netui:anchor>
    </p>
  </body>
</netui:html>
```

### View the Navigation Portlet

1. Save all files and start the server.
2. Preview the portlet by navigating to ***http://<host>:<port>YourWebapppp/portal.portal***.

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



## Step 3: Create Portlets that Communicate

In this step, you will use Page Flows to create one portlet that listens to another portlet. This particular example shows two ways of passing information from one portlet to the next: using a Form or using the Request Parameter. They both use the Page Flow as the means of connecting two portlets. By configuring the portlets to listen to one another, forms and requests instantiated by a JSP in one Page Flow can be sent to both Page Flows. This allows the listening portlet to update itself.

The tasks in this step are:

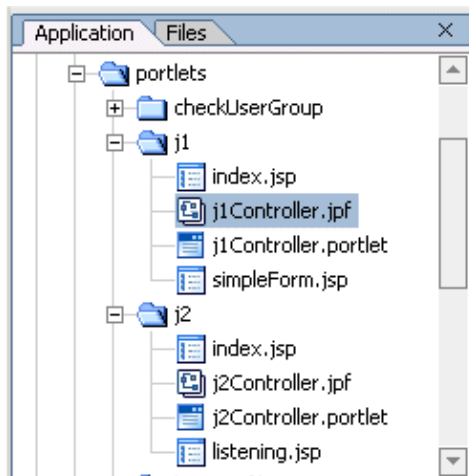
- Create two Page Flow portlets
- Create Actions, Additional JPS
- Place Portlets in Portal
- Set Properties on Portlets

Create two Page Flow portlets

In this task, two Page Flows are created within the portlets directory; one called *j1*, the other called *j2*.

1. Right-click on the portlets directory, and select *Create New Page Flow*.
2. Name the Page Flow *j1*, and select *Basic Page Flow* (with no Java Controls.)
3. Repeat this step, creating a Page Flow called *j2*.

**NOTE:** WebLogic Workshop automatically creates the Page Flow controller class, prepending the name of the Page Flow to the Controller.jpf filename.



Edit Page Flows

This example requires modifying the Page Flows so that they can be notified by the listening portlet. The *j1* Page Flow will receive form data from its *simpleForm.jsp*. The *j2* Page Flow (in a listening portlet) will also receive notifications, and store the data, so its portlet can retrieve and display it. \

**NOTE:** Use Flow View to set up actions, then touch up the code using the Source View.

### Edit j1 Page Flow

1. Open the *j1Controller.jspf* and make sure the package declaration is correct:

```
package portlets.j1;
```

2. Edit the begin method, adding a forward named simpleForm:

```
/**
 * @jpf:action
 * @jpf:forward name="simpleForm" path="simpleForm.jsp"
 */
public Forward begin()
{
    return new Forward( "simpleForm" );
}
```

3. Add the following three actions to the Page Flow:

```
/**
 * @jpf:action
 * @jpf:forward name="simpleForm" path="simpleForm.jsp"
 */
public Forward passString1( Form form )
{
    passedText = form.getText();
    return new Forward( "simpleForm" );
}

/**
 * @jpf:action
 * @jpf:forward name="simpleForm" path="simpleForm.jsp"
 */
public Forward passString2()
{
    return new Forward( "simpleForm" );
}

/**
 * @jpf:action
 * @jpf:forward name="simpleForm" path="simpleForm.jsp"
 */
public Forward passString3()
{
    return new Forward( "simpleForm" );
}
```

4. Make sure the following Form inner class appears at the end of the Page Flow file:

```
public static class Form extends FormData
{
    private String text;
    public void setText( String text )
    {
        this.text = text;
    }
}
```

## WebLogic Workshop Tutorials

```
public String getText()
{
    return this.text;
}
```

### Edit j2 Page Flow

1. Open the *j2Controller.jspf* and make sure the package declaration is correct:

```
package portlets.j2;
```

2. This Page Flow receives data from the *j1* portlet and stores it so it can be displayed in the *j2* portlet. directs the *j2* portlet to listen to the *j1* portlet. Add a variable declaration to the beginning of the class to hold the text from the *j1* Page Flow:

```
public class j2Controller extends PageFlowController
{
    public String thePassedText = "";
```

3. Edit the begin method, adding a forward named *listening*, which points to the *listening.jsp* we'll create later. The *listening.jsp* will display in the *j2* portlet the data received from the *j2* Page Flow.

```
/**
 * @jpf:action
 * @jpf:forward name="listening" path="listening.jsp"
 */
public Forward begin()
{
    return new Forward( "listening" );
}
```

4. The first action we'll now add to the Page Flow has the same signature as the *j1* *passString1* method. When the *j2* portlet is listening to *j1*, both the *j1.passString1* method and the *j2.passString1* methods are called.

```
/**
 * @jpf:action
 * @jpf:forward name="listening" path="listening.jsp"
 */
public Forward passString1(portlets.j1.j1Controller.Form form)
{
    thePassedText = form.getText();
    return new Forward( "listening" );
}
```

5. To illustrate other ways to pass strings between Page Flows, add *passString2* and *passString3* to the Page Flow for portlet *j2*.

- The *passString2* method uses a request to send the data.
- The *passString3* method uses the same portlet communication approach as *passString2*; the only difference is that the *j2* Page Flow sends data to its JSP portlet using an attribute.

```
/**
 * @jpf:action
 * @jpf:forward name="listening" path="listening.jsp"
 */
public Forward passString2()
{
    thePassedText = getRequest().getParameter("string2");
}
```



## WebLogic Workshop Tutorials

```
        return new Forward( "listening" );
    }

    /**
     * @jpf:action
     * @jpf:forward name="listening" path="listening.jsp"
     */
    public Forward passString3()
    {
        HttpServletRequest request = getRequest();
        attribValue = request.getParameter("string3");
        request.setAttribute("string3", attribValue);

        return new Forward( "listening" );
    }
}
```

### Create Additional JSPs

In this step, we'll create the "From" JSP for the *j1* portlet, and the "To" JSP used by the *j2* portlet.

### Create listening.jsp

1. In the Application palette, right-click on the *portlets/j2* folder and select *New JSP file*.
2. Name the file *listening.jsp*.
3. Open the source view, and insert the following code:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="netui-tags-html.tld" prefix="netui" %>
<h3>"To" JSP</h3>
Get text from jpf form = <b><netui:label value="{pageFlow.thePassedText}"/></b>
<br/>
<br/>
Using {request.string3} databinding = <b> <netui:label value="{request.string3}"/>
</b> <br/>
<% String attribName = "string3"; %>
Using request.getAttribute() = <%=request.getAttribute(attribName)%>
<br/>
<br/>
```

4. Save *listening.jsp*.

### Create simpleForm.jsp

1. In the Application palette, right-click on the *j1* folder and select *New JSP file*.
2. Name this new JSP file *simpleForm.jsp*.
3. Switch to Design View.
4. From the Netui Palette, drag a TextBox object onto the form, then a Button object right below it. Use the Form object to wrap around both of them. Initially, the source for this portion of the jsp will look something like this:

```
<netui:form action="none">
<netui:textBox></netui:textBox>
<netui:button>Submit</netui:button>
</netui:form>
```

5. *Edit the source of the fragment you created:* Add the *passString1* action to the form, and designate a *dataSource* attribute for the *textBox*:

## WebLogic Workshop Tutorials

```
<netui:form action="passString1">
<netui:textBox dataSource="text"/>
<netui:button>Submit</netui:button>
</netui:form>
```

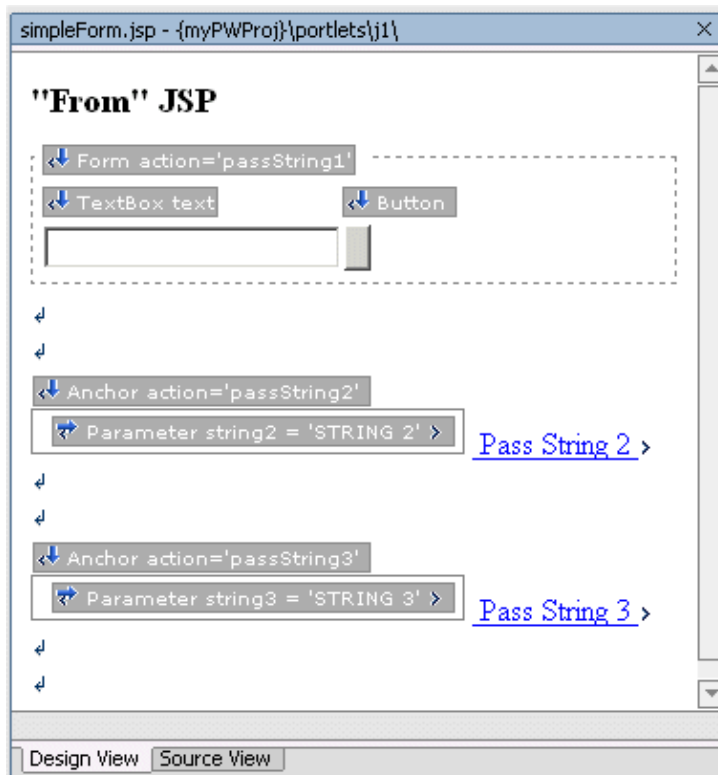
6. From the Netui Palette, drag an anchor element just below the textBox. The Form Wizard appears. Invoke the `passString2` action, labelling it "PassString2", or edit the source code as follows:

```
<netui:anchor action="passString2">Pass String 2
<netui:parameter name="string2" value="STRING 2"/>
</netui:anchor>
```

7. From the Netui Palette, drag one more anchor element just below the previous one, and edit the resulting code as follows:

```
<netui:anchor action="passString3">Pass String 3
<netui:parameter name="string3" value="STRING 3"/>
</netui:anchor>
```

8. The Design View of *simpleForm.jsp* should now look something like this:



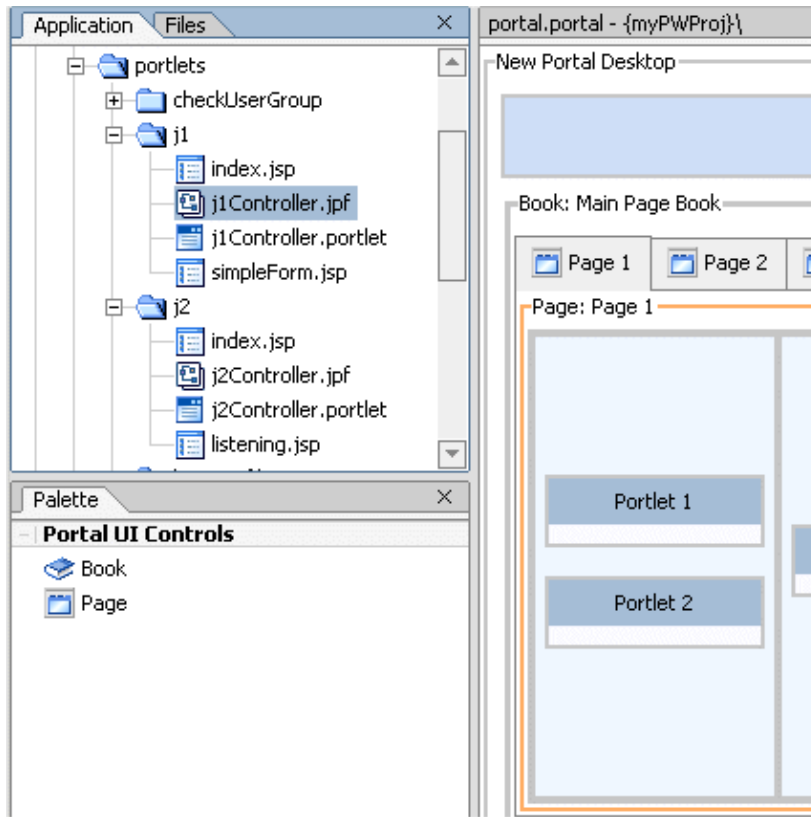
**NOTE:** As you edit the page flow, you can verify the navigation by opening a viewer that will preview the pages without the portal. To do this, click on the **Start** arrow from the WebLogic Workshop toolbar, or press **CTRL+F5**.

### Place Portlets in Portal

To view the portlets, they need to be placed in a portal.

## WebLogic Workshop Tutorials

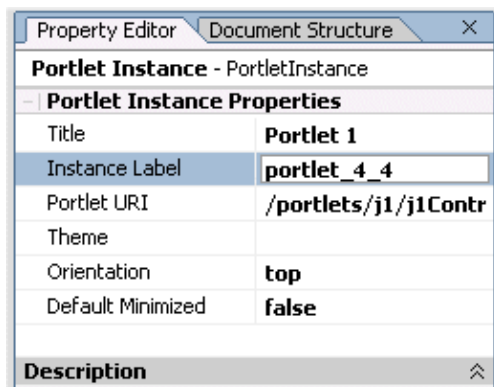
1. Open the portal created in Step 1 in Design View, select a placeholder, and drag the Page Flows into placeholders on a page, using the Portlet Wizard to create new portlets from each Page Flow.
2. Use the Property designer to edit the Title attribute for each portlet.
3. From the Portal designer, the portal should now look something like this:



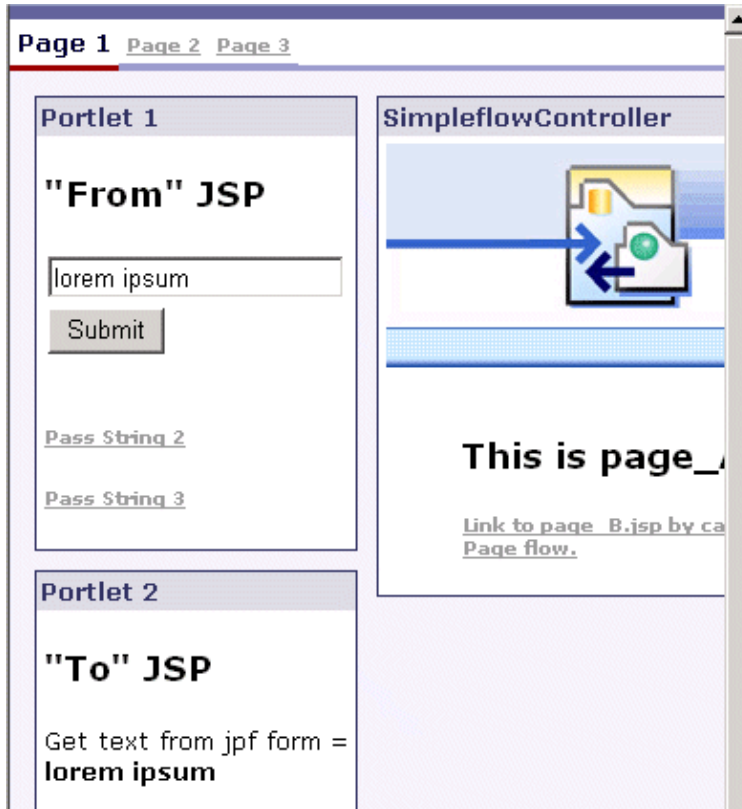
### Set Properties on Portlets

When a portlet is placed inside a portal, it is assigned an *instanceLabel* which the framework uses to keep track of individual portlet placement. In order to make this sample work, the `listenTo` attribute on portlet *j2* needs to be set to the `instanceLabel` of the *j1* portlet on your portal.

1. Open the portal, select the *j1* portlet and verify its `instanceLabel`. In this example, it is *portlet\_4\_4*.



2. Now open the **j2** portlet by double-clicking on it within the portal. The `listenTo` attribute for this portlet needs to be set to the `instanceLabel` for the **j1** portlet in your portal.
3. Save all files and start the server.
4. Preview the portal by navigating to ***http://<host>:<port>YourWebappp/portal.portal***.
5. The resulting portal should look something like this:



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



## Step 4: Create an EJB Control Page Flow portlet

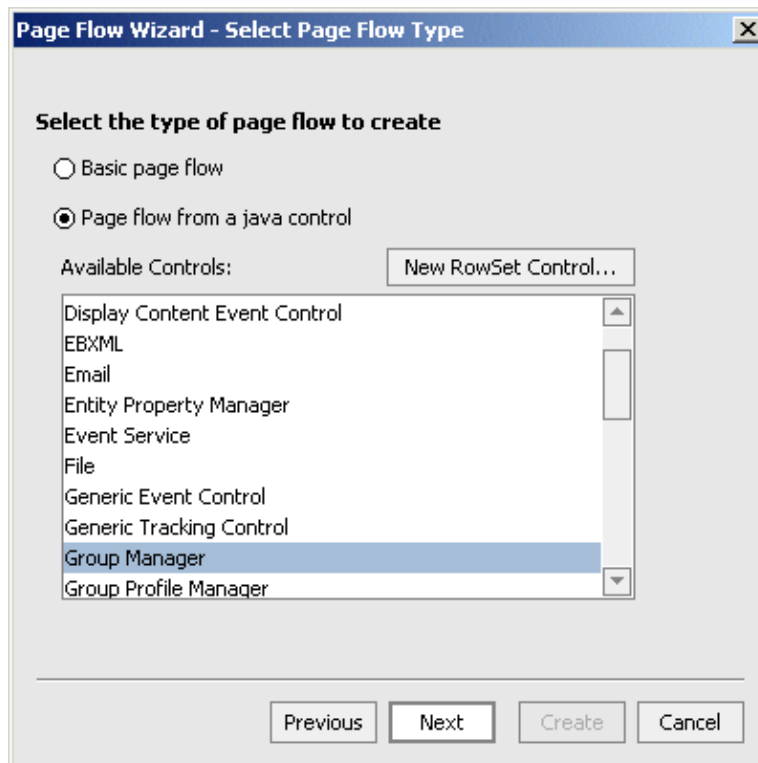
In this step you create a Page Flow portlet, add a Personalization Control, and edit properties on the Control.

The tasks in this step are:

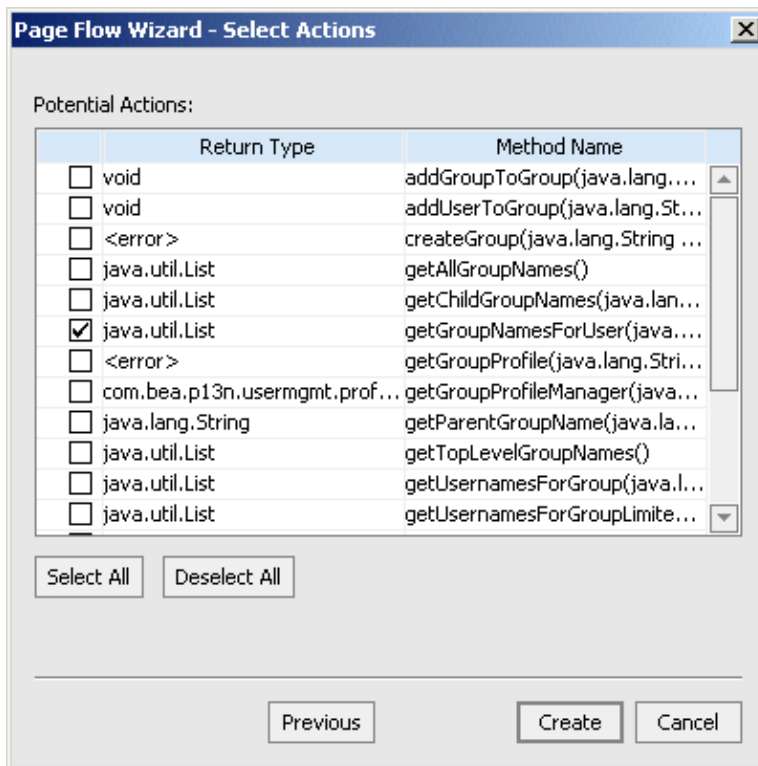
- Create an EJB Control Page Flow
- Place the Page Flow into a Portlet, Place Portlet in a Portal

Create a New Page Flow

1. From within the Project tab in your Web application, right-click on the portlets directory and create a new Page Flow. Name the Page Flow *checkuserGroup*.
2. In the Page Flow Wizard, select Page flow from a java control.
3. Select the *GroupManager* control. Click *Next*.



4. From the list of Potential Actions, select `getGroupNamesForUser`. Click *Create*.



5. The wizard creates an index.jsp page with a link to the `getGroupNamesForUser` action that invokes the control.


Place the Page Flow into a Portlet, Place Portlet in a Portal

1. With the Portal open in Design View, drag the ***checkUserGroupController.jpf*** into a placeholder in the portal.
2. The Portlet Wizard presents a prompt offering to create a portlet from this resource. Click ***Yes***.
3. The Portlet Details screen appears. Click ***Finish***.
4. Drag the resulting portlet into a placeholder in your portal.
5. Preview the portlet by navigating to ***http://<host>:<port>YourWebappp/portal.portal***.
6. Click on the ***getGroupNamesForUser*** link, enter a username and click ***Submit***.

Page 1 Page 2 **Page 3**

---

checkUserController




---


Username:

7. In this example, user *weblogic* is a member of the groups *Administrators* and *PortalSystemAdministrators*.

Page 1 Page 2 **Page 3**

---

checkUserController




---

**Page Flow: Check Groups for User**

Input Forms For Actions With Parameters  
[getGroupNamesForUser](#)

Results Area

[Administrators, PortalSystemAdministrators]

Click on the arrow below to go back in the tutorial:



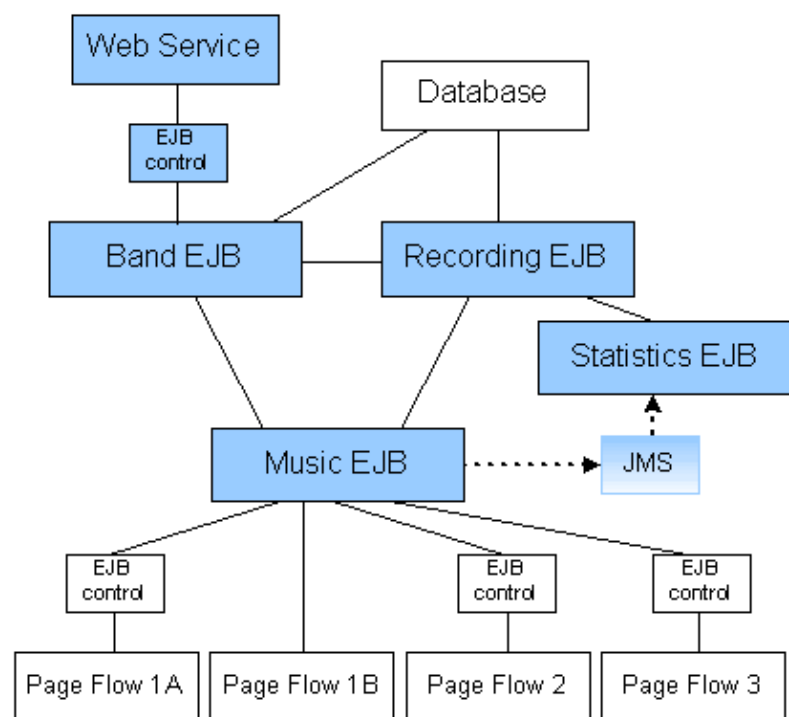
# Tutorial: Enterprise JavaBeans

## The Big Picture

In this tutorial you will learn how to develop an application that uses Enterprise JavaBeans technology. You will build the *EJBProjectTutorial* application, which uses session, entity, and message-driven beans to model a music site, registering information about bands, recordings, and recording ratings. The below diagram shows the components that you will build in blue. The components in white are predefined in the tutorial.

**Note.** If you are new to Enterprise JavaBeans technology, you might want to consider running the Getting Started: Enterprise JavaBeans tutorial first before continuing with this tutorial. The Getting Started Tutorials is a series of short tutorials that offers a basic step-by-step introduction to the core J2EE components and how you can develop these components with WebLogic Workshop

The tutorial consists of three parts. In Part One, you will create a new application, import an existing entity bean *Band* and develop a stateless session bean *Music*. You will also add component methods and EJB references. In the final section of Part One you will test the EJBs using two different page flows. Both page flows allow you to add bands to the database and receive a listing of known bands. One page flow uses an EJB control to locate and reference an EJB while the other page flow uses Java JNDI API directly embedded in a JSP page.



In Part Two of the tutorial you will develop a second entity bean *Recording*, and define a relationship between bands and recordings. you will use another page flow to add bands and recordings for bands, and receive listings of recordings and bands. You will also develop a test web service to run and test the Bands bean.

In Part Three of the tutorial you will develop a message-driven bean *Statistics* that creates ratings for recordings. You will learn how to send a message for this bean from the Music EJB and how to store the rating information the Statistics bean generates. You will use another page flow to test the EJBs.

## Tutorial Goals

In this tutorial you will learn how to develop session, entity, and message-driven beans. Along the way you will become acquainted with advanced features such as entity relationships. Also, this tutorial shows how to build a full-fledged application by including a page flow application, and demonstrates popular design



patterns used to model EJBs. The tutorial guides you through the process of adding functionality in increments and shows how to test your EJBs as you build them.

### Tutorial Overview

The following components are used in this tutorial:

- A *Band* entity bean, which holds information about bands.
- A *Recording* entity bean, which holds information about recordings.
- A *Music* session bean, which references the Band entity bean, and provides the interface to client applications.
- A *Statistics* message-driven bean, which processes messages sent by the Music bean.
- The client applications of the Music bean. In part one of the tutorial, two page flows invoke methods of the Music EJB to add a band and receive a listing of currently known bands. One of these page flows uses an EJB control to communicate with the Music bean. In part two of the tutorial, a page flow invokes the methods of the Music bean to add a band, receive a listing of currently known bands, and, for a given band, add a recording and receive a listing of currently known recordings. In part three of the tutorial, a page flow is used that incorporates all the functionality of the page flow used in part two, plus enables you to view rating information of recordings. The page flows in parts two and three of the tutorial use EJB controls to communicate with the Music bean.
- The test web service of the Band bean. This web service is meant to be used for testing purposes only during development. It allows you to quickly run and test a bean without having to be involved in the (re)design of page flows. The test web service uses an EJB control to communicate with the Band bean.
- A database that will hold the tables used to store information about recordings and bands.
- A JMS provider and queue to send messages to. You will use a provider available in the domain and create the JMS queue.

The entity beans in this tutorial perform operations on database tables that hold information about bands and recordings. The session bean encapsulates the business logic of the music site. It serves as an intermediary between client applications and the entity beans – a common design pattern called Session Façade – and also sends JMS messages. The message-driven bean, which processes these JMS messages, invokes an entity bean to store information generated during the processing of a message. Message-driven beans are frequently used in this manner to enable asynchronous handling of business tasks that require involvement of session and/or entity beans.

### Parts in This Tutorial

#### Part One: Building Your First EJBs

In this part you will create the first set of Enterprise JavaBeans in the tutorial. You will start WebLogic Workshop and WebLogic Server, create a new application and EJB project, import an entity bean that represents band objects, and develop the session bean Music.

#### Part Two: Defining an Entity Relationship

In the second part of the tutorial you will add a second entity bean holding information about recordings, define the relationship between bands and recordings, and update the session bean to enable client applications to add recordings and receive listings of recordings for bands.

### Part Three: Adding a Message-Driven Bean

In part three of the tutorial you will add a message-driven bean that will generate rating information for a recording, update the Recording bean to store this information, and update the session bean to send a message for the message-driven bean and enable client applications to view ratings.

### Summary: Enterprise JavaBean Tutorial

Reviews what you have done in this tutorial.

To begin the tutorial, see Part One: Building Your First EJBs. You can also click the arrow to navigate to this next step.



# Part One: Building Your First EJBs

In part one of the tutorial you will create the first set of Enterprise JavaBeans. You will start WebLogic Workshop and WebLogic Server, create a new application and EJB project, import an entity bean that represents band objects, and develop the session bean Music. Finally, you will run various page flow applications to test the EJBs.

## Steps in Part One

### Step 1: Begin the EJB Tutorial

In this step you set up your application, create an EJB Project, and start WebLogic Server.

### Step 2: Import an EJB from a JAR

In this step you import an entity bean.

### Step 3: Create a Session Bean

In this step you create a stateless session bean that references the entity bean.

### Step 4: Run a Web Application

In this step you will test the EJBs by running a client application. Two client applications are presented that differ in the way they locate and reference the session bean, but are otherwise identical.

### Related Topics

### Developing Enterprise JavaBeans

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



# Step 1: Begin the EJB Tutorial

In this step you will create the application that you will be working with throughout this tutorial. In WebLogic Workshop, an application contains one or more projects. Enterprise JavaBeans are developed in an EJB project.

The tasks in this step are:

- To Start WebLogic Workshop
- To Create a New Application and Select a WebLogic Server Domain
- To Create a New EJB Project
- To Start WebLogic Server

To Start WebLogic Workshop

If you have an instance of WebLogic Workshop already running, you can skip this step.

... on Microsoft Windows

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

- From the *Start* menu, choose *Programs*—>*WebLogic Platform 8.1*—>*WebLogic Workshop 8.1*.

...on Linux

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

- Open a file system browser or shell.
- Locate the *Workshop.sh* file at the following address:

```
$HOME/bea/weblogic81/workshop/Workshop.sh
```

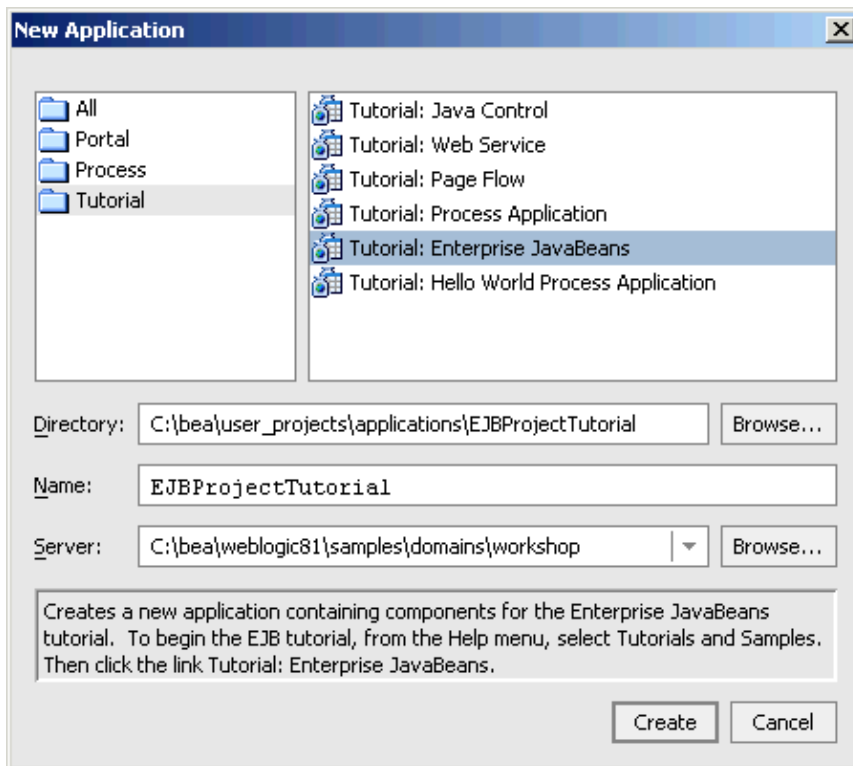
- In the command line, type the following command:

```
sh Workshop.sh
```

To Create a New Application and Select a WebLogic Server Domain

To create the EJB project, you must first create the application to which it will belong:

1. From the *File* menu, select *New*—>*Application*. The *New Application* dialog appears.
2. In the *New Application* dialog, in the upper left-hand pane, select *Tutorial*.  
In the upper right-hand pane, select *Tutorial: Enterprise JavaBeans*.  
In the *Directory* field, use the *Browse* button to select a location to save your source files.  
In the *Name* field, enter EJBProjectTutorial.  
In the *Server* field, from the drop-down list, select  
BEA\_HOME\weblogic81\samples\domains\workshop.

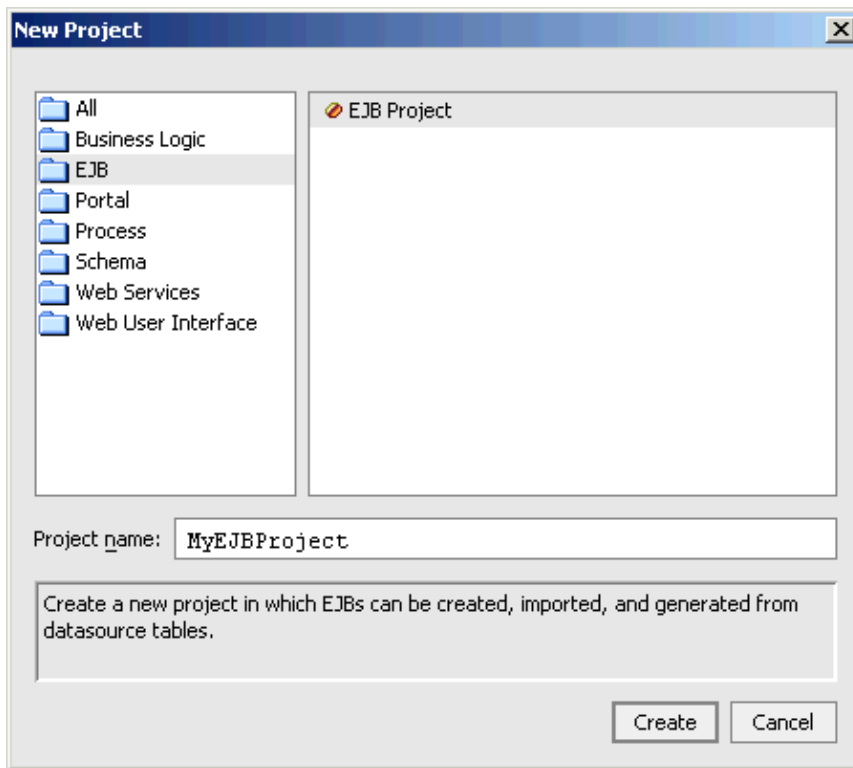


3. Click **Create**.

The new application already contains various web applications that will be used in the tutorial to test your EJBs. Next you need to create an EJB project for the EJBs you are going to develop.

To Create a New EJB Project

1. In the **Application** tab, right-click the EJBProjectTutorial folder and select **New-->Project**.
2. In the **New Project** dialog, in the upper left-hand pane, confirm that **EJB** is selected.  
In the upper right-hand pane, select **EJB Project**.  
In the **Project Name** field, enter MyEJBProject.

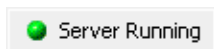


3. Click **Create**.

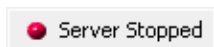
To Start WebLogic Server

Since you will be deploying and running your EJBs on WebLogic Server, it is helpful to have WebLogic Server running during the development process.

You can confirm whether WebLogic Server is running by looking at the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, a green ball is displayed as pictured below.



If WebLogic Server is not running, a red ball is displayed, as pictured below.



If you see the red ball in the status bar, then follow these instructions to start WebLogic Server:

- From the **Tools** menu, choose **WebLogic Server**—>**Start WebLogic Server**.

**Note:** on the **WebLogic Server Progress** dialog, you may click **Hide** and continue to the next task.

Related Topics

Getting Started with EJB Project

How Do I...? EJB Topics

Step 1: Begin the EJB Tutorial

## WebLogic Workshop Tutorials

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



## Step 2: Import an EJB from a JAR

If you have developed Enterprise JavaBeans in another development environment and plan to continue development of these EJBs in WebLogic Workshop, you can import these beans into an EJB project. To do so you must have the EJB JAR and the source code of the EJBs. An EJB Jar typically will contain multiple EJBs that can be imported at the same time. In this step of the tutorial you will import one entity bean.

The tasks in this step are:

- To Import an EJB
- To Inspect the Imported Entity Bean
- To Build the EJBS

**Note.** If you have developed EJBs in another development environment and plan to reuse these EJBs in a WebLogic Workshop application, but are not planning to continue development on these beans, you only have to add the EJB JAR to the application. For more information, see *How Do I: Add an Existing Enterprise JavaBean to an Application?*

To Import an EJB

1. On the **Application** tab, right-click the folder **MyEJBProject** folder and select **Import EJB from jar**. The Import Wizard appears.
2. Click the **Browse** button next to the top field and select the EJB JAR:

```
[BEA_HOME]\weblogic81\samples\platform\tutorial_resources\EJBTutorial\import_jar\ejb20_r
```

3. Click the **Browse** button next to the bottom field and select the source directory:

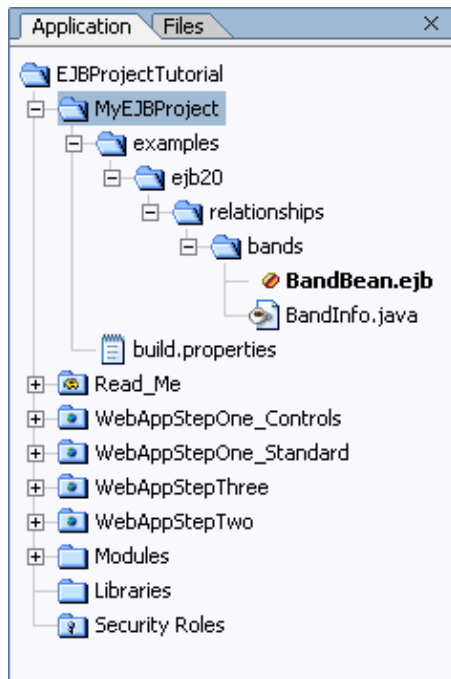
```
[BEA_HOME]\weblogic81\samples\platform\tutorial_resources\EJBTutorial\import_src_dir
```

4. Click **Next**.
5. In **Confirm EJB source file locations**, confirm that the BandBean is checked and click **Finish**.
6. An **Import Progress** window appears. When the import is complete, click **OK**.

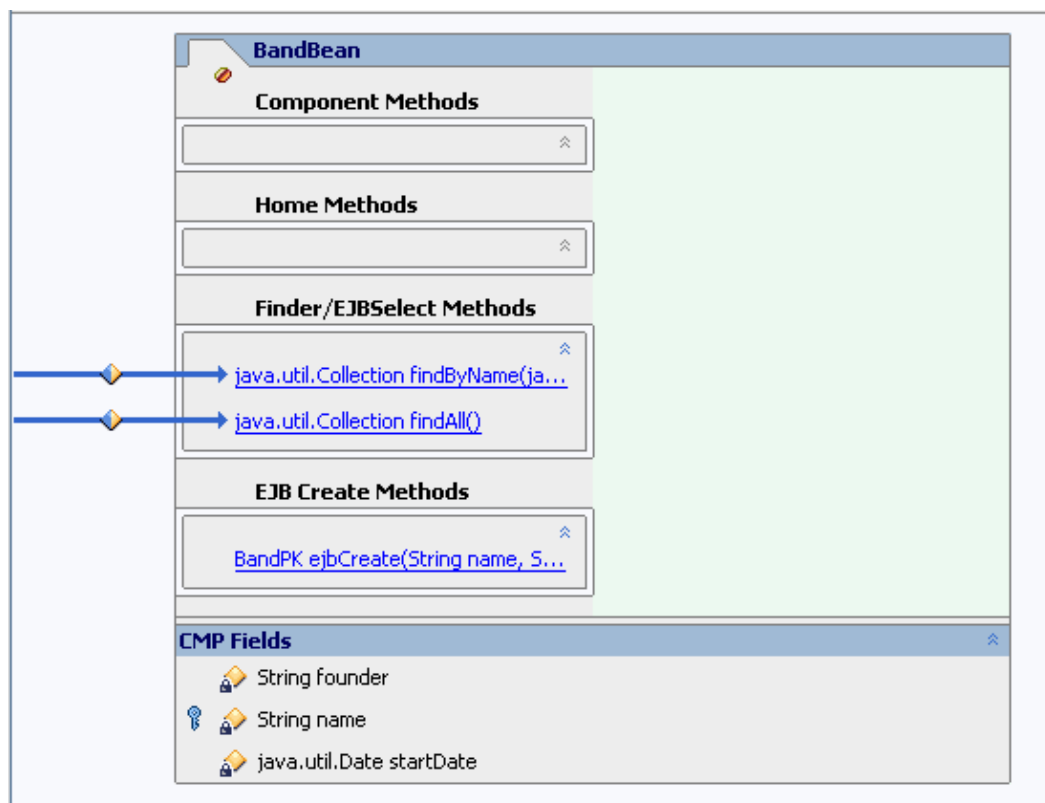
To Inspect the Imported Entity Bean

The **Application** tab displays the BandBean that has been imported. Notice that the package name of the bean is reflected in the directory structure. In addition to the entity bean, a regular java class, which will be used as a dependent value class, has been imported.





The Band EJB in Design View should look like this:



If you have developed Enterprise JavaBeans before, you might notice that the BandBean entity bean is represented by one file with an ejb extension, instead of several java files for the class definition and the definition of the various interfaces. In addition, when you double-click BandBean.ejb and go to Source View, you will notice special tags with the prefix ejbgen. These ejbgen tags are for instance used to mark methods to be exposed in remote and home interfaces, mark relationship fields, and describe EJB QL queries directly in

the bean class. Instead of using separate classes for the various interfaces and instead of directly changing the deployment descriptor, WebLogic Workshop uses these tags to contain all the bean's information in one file. When you build an EJB Project, the WebLogic tool uses these tags to generate the various interfaces and to generate the appropriate deployment descriptor. For more information, see *Getting Started with EJB Project* or the *Getting Started: Enterprise JavaBeans* tutorial.

The Band bean defines three CMP fields to represent bands: (1) the band founder, (2) the name of the band, and (3) the start date when the band was founded. The name field is the primary key field. In other words, the assumption is made that each band has a unique name. All fields are read-only, that is, no setter methods are defined for these fields to make changes. The Band bean has three methods. There is one `ejbCreate` method to create a new band object, and there are two finder methods: the `findByName` methods returns the band object that matches a name argument, and the `findAll` method returns all band objects stored in the database. The *Property Editor* reveals that the *ejb-name* is `BandEJB`, the *data-source-name* is `cgSampleDataSource` and the *table-name* is `bands`. Also notice in the *JAR Settings* section that *create-tables* is set to `CreateOnly`. The latter setting means that when you build and deploy this bean, the table will be automatically created for you. To learn more about the other table creation options set with this property, see `@ejbgen:jar-settings` Annotation. Notice that the *create-tables* setting is part of the JAR settings. In other words, for every entity bean created as part of this project, the same table creation setting applies.

### To Build the EJBs

In this tutorial you will be building the EJBs frequently to verify that changes were made correctly:

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View—>Windows—>Build**.
2. In the **Application** pane, right-click `MyEJBProject`, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR `MyProjectEJB.jar` is created.
4. After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.
5. (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand **Modules**, expand `MyProjectEJB.jar`, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

If you encounter build errors, verify that both `BandBean.ejb` and `BandInfo.java` were imported successfully. If this is not the case, delete the imported files and repeat the import steps above. In particular make sure that you are pointing at the correct folder containing the EJB source files. A build error might be followed by a deployment error, in particular for `WebAppOne_Standard` and related to an unresolved `ejb-link`. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error.

If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step). If you encounter deployment errors that seem to be related to the underlying data source, verify that you are working in the workshop domain by selecting **Tools—>WebLogic Server—>Server Properties**. The **Server Home Directory** should point to the workshop domain.

## WebLogic Workshop Tutorials

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



## Step 3: Create a Session Bean

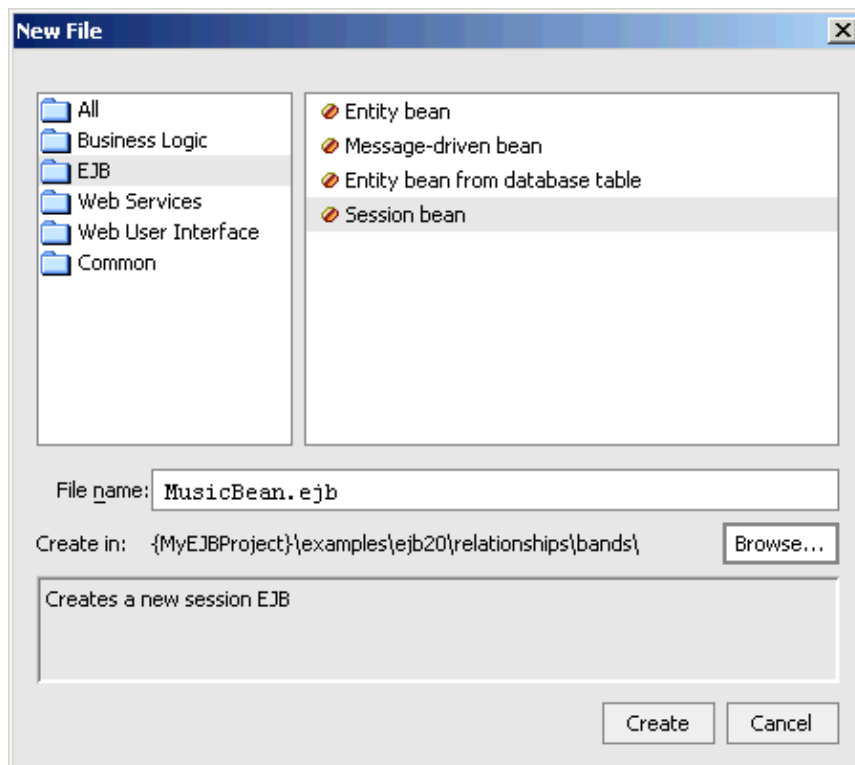
In this step you will create a stateless session bean *Music*, which will be called by client applications. A client application has no direct access to the entity bean *Band* but calls the session bean instead. The *Music* bean implements the business logic to respond to a client request, invoke the *Band* bean to obtain data, and return a result to the client. This use of a session bean as an intermediary is a common design pattern called *Session Façade*.

The tasks in this step are:

- To Create a Session Bean
- To Reference the Entity Bean
- To Add an `ejb-local-ref` Tag
- To Add Component Methods
- To Build the EJBs

To Create a Session Bean

1. Right-click the *bands* folder in the *Application* tab (the same folder that contains *BandBean.ejb* and *BandInfo.java*), and select *New*—>*Session Bean*.
2. Enter the file name *MusicBean.ejb*.



3. Click *Create*.

To Reference the Entity Bean

The Music EJB will be invoking methods of the Band EJB. In order for the Music bean to invoke the Band bean's methods, it must first locate and obtain a reference to the bean's home. In this step, you modify the `ejbCreate` method in `MusicBean.ejb` to locate and obtain a reference to the Band EJB.

**Note.** To learn more about about EJB interfaces, referencing and JNDI naming, see the Getting Started: Enterprise JavaBeans tutorial.

To modify `ejbCreate` to obtain a reference to the Band EJB's local home interface:

1. In the **Application** pane, double-click `MusicBean.ejb` and select the Source View tab.
2. Locate the `ejbCreate` method in `MusicBean.ejb`, and modify its definition as shown in red below. You might notice that the method uses Java JNDI to locate and obtain a reference to `BandHome`:

```
public void ejbCreate() {  
    try {  
        javax.naming.Context ic = new InitialContext();  
        bandHome = (BandHome)ic.lookup("java:comp/env/ejb/BandEJB");  
    }  
    catch (NamingException ne) {  
        throw new EJBException(ne);  
    }  
}
```

3. Locate the import section at the top of the file and add the following import statements, or use Alt+Enter when prompted by the IDE to automatically add these import statements:

```
import javax.naming.InitialContext;  
import javax.naming.NamingException;
```

4. Go to the beginning of the class definition and define `bandHome` as shown in red below:

```
public class MusicBean  
    extends GenericSessionBean  
    implements SessionBean  
{  
    private BandHome bandHome;  
  
    public void ejbCreate() {  
        ...  
    }  
}
```

To Add an `ejb-local-ref` Tag

In the above `ejbCreate` method, in the lookup method, you use `BandEJB` to reference the Band EJB. At deployment time this reference is mapped to the actual location of the Band bean. This mapping is done in the deployment descriptor. In WebLogic Workshop you don't modify the deployment descriptor directly but use `ejbgen` tags inserted in the `ejb` file instead. To map the reference to Band EJB in the Music bean to the actual Band bean location, you need to insert an `ejbgen:ejb-local-ref` tag in `MusicBean.ejb`:

1. Ensure that the Music EJB is displayed in Design View.
2. Right-click the Design View, and select **Insert EJB Gentag**—>**ejb-local-ref**. A new `ejbgen:ejb-local-ref` section appears in the **Property Editor**.
3. Use the **Property Editor** to enter the link property, or go to Source View to add the property:  
\* `@ejbgen:ejb-local-ref link="BandEJB"`

Notice that the attribute specified in the link property corresponds to the `ejb-name` of the Band bean. To learn more about this tag, place your cursor anywhere inside the tag in Source View and press **F1**.

## To Add Component Methods

Now you will add the business methods to the Music bean that will be called by the client application. Before you add the `getBands` and `addBand` methods, you first need to define Music bean's local interfaces:

1. Ensure that the Music EJB is displayed in Design View.
2. In the **Property Editor**, locate the **Naming** section. Clear the **Remote EJB** and check the **Local EJB**. In other words, for the Music EJB you will use its local interfaces.
3. Expand the **Local EJB** section and enter `ejb.Music` in the **JNDI name** field. Verify that the bean class name is `MusicLocal`, and the home class name is `MusicLocalHome`, as is shown in the following picture:

Naming	
<input type="checkbox"/> Remote EJB	
<input checked="" type="checkbox"/> Local EJB	
JNDI name	<code>ejb.Music</code>
Bean class name	<code>MusicLocal</code>
Home class name	<code>MusicLocalHome</code>

4. In Design View, right-click the MusicBean and choose **Add Component Method**. A component method called `void method()` appears in the **Component Methods** section.
5. Rename the method `Collection getBands()`. If you step off the method and need to rename it, right-click the method and select **Rename**.
6. Right-click the arrow to the left of the method and select **Local**. The business method will now be defined in the local interface during build.
7. Click the component method to go to Source View and modify the method as shown in red below:

```
/**
 * @ejbgen:local-method
 */
public Collection getBands() {
    try {
        Iterator bands = bandHome.findAll().iterator();
        Collection result = new ArrayList();
        while (bands.hasNext()) {
            Band band = (Band)bands.next();
            result.add(band.getName());
        }
        return result;
    }
    catch (FinderException fe)
    {
        throw new EJBException(fe);
    }
}
```

Notice that the `getBands` method calls uses the `BandBean` to obtain all the bands currently stored in the database. The method uses the `Band EJB`'s `findAll` method to find all bands, and returns a `Collection` of band names. To learn more about finder methods, see [Query Methods and EJB QL](#).

8. Locate the import section at the top of the file and add the following import statements, or use **Alt+Enter** when prompted by the IDE to automatically add these import statements:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
```

## WebLogic Workshop Tutorials

- Now repeat steps 4 through 7 for the addBand method. Go to Design View, right-click the MusicBean and choose **Add Component Method**. A component method called void method1() appears in the **Component Methods** section.
- Rename the method void addBand(BandInfo bandInfo).
- Right-click the arrow to the left of the component method and select **Local**.
- Click the component method to go to Source View and modify the method as shown in red below:

```
/**
 * @ejbgen:local-method
 */
public void addBand(BandInfo bandInfo) {
    try {
        bandHome.create(bandInfo.getName(), bandInfo.getFounder(), bandInfo.getStartData
    }
    catch (CreateException ce) {
        throw new EJBException(ce);
    }
}
```

Notice that the method calls the Band EJB's create method to add a new band to the database.

- Save your results.

### To Build the EJBs

- Locate the **Build** window in the IDE. If the **Build** window is not there, select **View—>Windows—>Build**.
- In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
- Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is created.
- After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.
- (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand Modules, expand MyProjectEJB.jar, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

If you encounter a build error related to the ejbgen:ejb-local-ref tag, make sure that you have specified the BandBean's name correctly. If you encounter build errors related to the component methods, verify that the methods have been defined correctly and that you have imported the correct classes. A build error might be followed by a deployment error, in particular for WebAppOne\_Standard and related to an unresolved ejb-link. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error.

If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step).

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



## Step 4: Run a Web Application

Now you will run two page flow applications to test the EJBs. Both page flows allow you to add a band to a database and to view the list of bands. The two applications differ only in how they reference the Music EJB. The first page flow uses EJB Controls in the page flow controller file. The second page flow uses JNDI naming defined directly in the JSP. You can either run both page flow applications or skip the second one.

**Note.** Page flows and EJB controls are not the main topics of this tutorial and are used here primarily to demonstrate how to run EJBs. To learn more about page flows and EJB controls, see the related topics at the bottom of this page.

### The WebAppStepOne\_Controls Page Flow

Before you run this page flow, you first might want to examine the structure of this web application.

To Examine the EJB Control

- In the **Application** pane, double-click WebAppStepOne\_Controls, double-click EJBControls, and double-click the EJB control MusicBeanControl.jcx.
- Go to Source View and verify that the control references the (fully qualified) names of the Music bean's local JNDI name (ejb.Music), its local home interface (examples.ejb20.relationships.bands.MusicLocalHome), and its local business interface (examples.ejb20.relationships.bands.MusicLocal). Remember that you set these Music bean properties using the **Property Editor** in the previous step.

**Note.** If the definition of the EJB control reports errors, please go back to the previous step of the tutorial and make sure that the mentioned names are set correctly in the **Property Editor**. If you make corrections to the Music bean, you must rebuild the EJB project.

To Examine the Page Flow

- In the **Application** pane, locate WebAppStepOne\_Controls and double-click Controller.jpf. The file opens in Flow View.
- Notice that the page flow begins by going to the JSP page addBand.jsp. Also notice the action addABand, which after executing returns to the JSP page.
- Go to Source View. At the top of the class definition, notice the definition of the EJB Control variable library:

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

- Locate the definition of the action addABand, and notice that the EJB's business methods are exposed and invoked via the EJB control's methods with the same names:

```
protected Forward addABand(AddABandForm form)
{
    ...
    library.addBand(info);
    allBands = library.getBands();
    ...
}
```



## WebLogic Workshop Tutorials

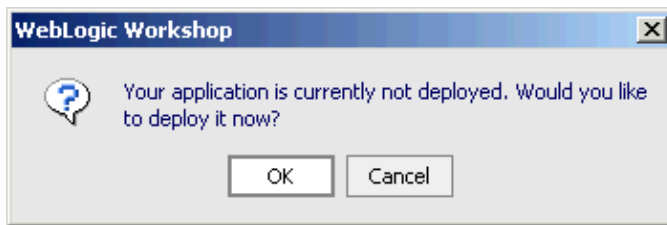
- (Optional.) Double-click the JSP page addBand.jsp to examine its content.

### To Run the Page Flow

- Make sure that Controller.jspf is opened.
- Click the **Start** button.



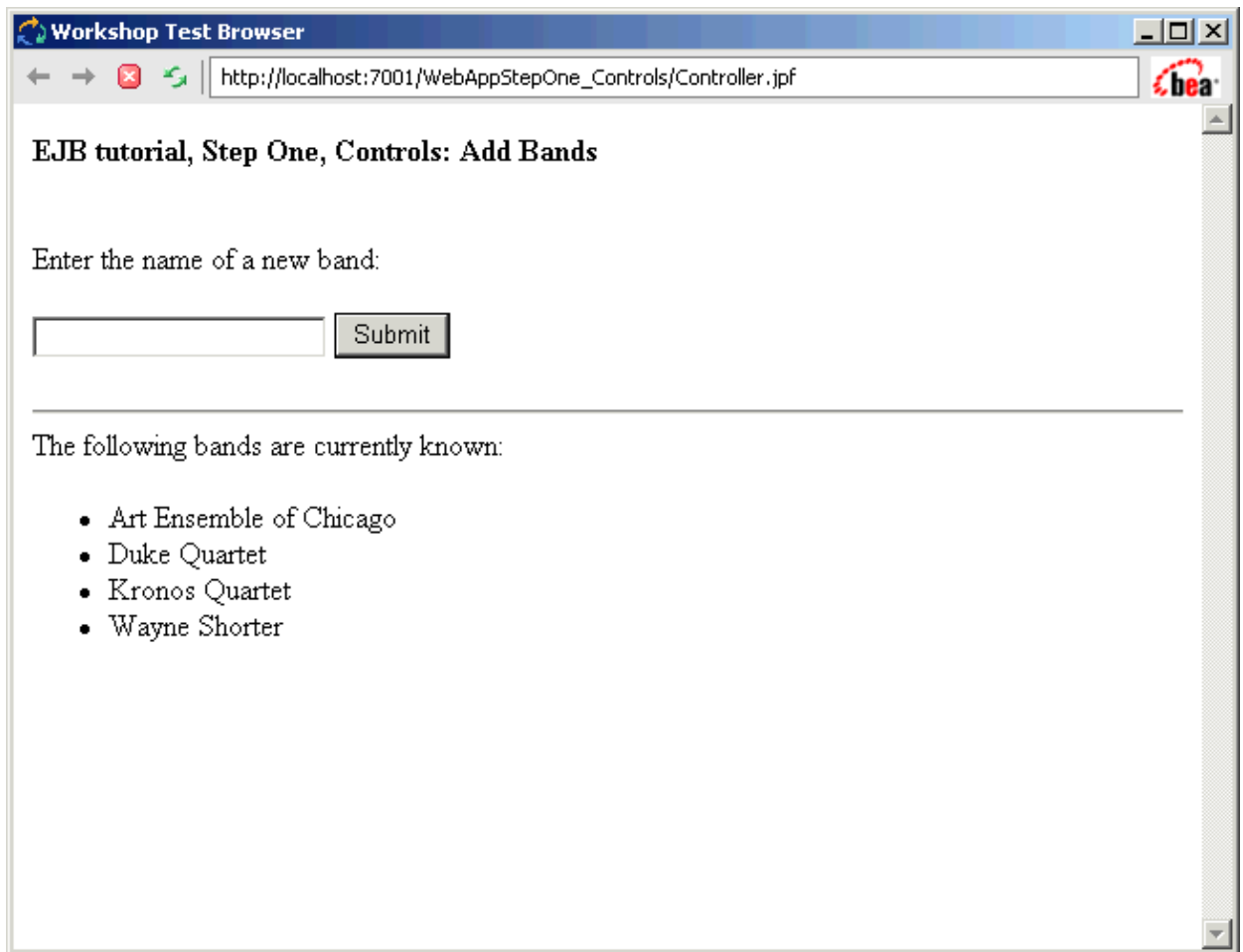
The first time the following dialog might appear:



If this dialog appears, press **OK**.

Workshop launches the Workshop Test Browser.

- Enter a band name and press the **Submit** button. This will call the Music EJB's addBand method, which will add the band to the database via the Band bean. A list containing only the band you just entered appears, which is obtained by calling the getBands method. Now enter as many additional bands as you would like. The names in the list of currently known bands appear in the order that you entered them, and are not sorted by name. You can enter the names as shown in the below picture, or you can add the names of your favorite bands. In either case, after adding a few bands, your browser should be similar to the below picture.



- Return to *WebLogic Workshop* and press the *Stop* button to close the Test Browser.



## The WebAppStepOne\_Standard Page Flow

This page flow is solely intended to demonstrate how EJBs can be called directly from a JSP and without using an EJB control. In general it is recommended that you use an EJB control called from a page flow controller file instead, as shown in the above page flow application.

Before you run this page flow, you first might want to examine the structure of this web application.

To Examine the web.xml Descriptor

- In the *Application* pane, double-click WebAppStepOne\_Standard, double-click WEB-INF, and double-click web.xml.
- Locate the <ejb-local-ref> element and verify that it references the fully qualified names of the Music bean's local home interface and local business interface, and the EJB name qualified by the EJB JAR name. Notice that the definition is similar to the definition used in the EJB control above.

```
<ejb-local-ref>
  <ejb-ref-name>ejb/MusicLink</ejb-ref-name>
```

## WebLogic Workshop Tutorials

```
<ejb-ref-type>Session</ejb-ref-type>
<local-home>examples.ejb20.relationships.bands.MusicHome</local-home>
<local>examples.ejb20.relationships.bands.Music</local>
<ejb-link>MyEJBProject.jar#Music</ejb-link>
</ejb-local-ref>
```

**Note.** If the definition of the `<ejb-local-ref>` element does not match the Music bean's interfaces, please go back to the previous step of the tutorial and make sure that the mentioned names are set correctly in the **Property Editor**. If the name of the JAR file (in the `ejb-link` descriptor) is incorrect, rename the EJB project as specified in step 1 of the tutorial. After you make corrections, you must rebuild the EJB project.

To Examine the JSP page

- In the **Application** pane, locate WebAppStepOne\_Standard and double-click Controller.jspf. The file opens in Flow View.
- Notice that the page flow simply goes to the JSP page addBand.jsp.
- Double-click the JSP page addBand.jsp, and go to Source View. Notice that the Music EJB is referenced using the `ejb/MusicLink` reference defined in the deployment descriptor, and that JNDI API is used to locate and reference the Music bean.

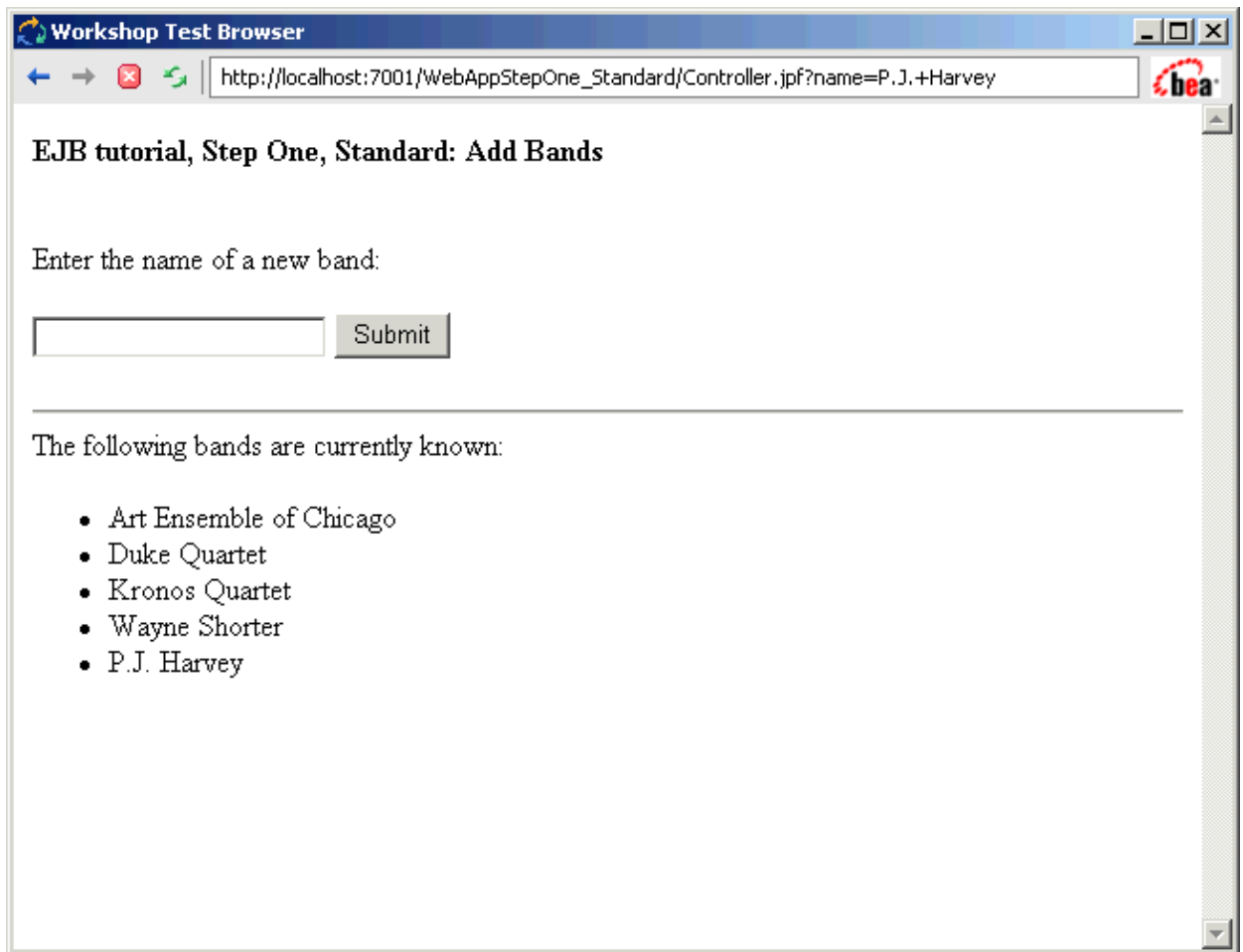
To Run the Page Flow

- Make sure that Controller.jspf is opened.
- Click the **Start** button.



Workshop launches the Workshop Test Browser.

- A listing of the bands you entered with the other page flow appears. Enter a band name and press the **Submit** button. This will call the Music EJB's `addBand` method, which will add the band to the database using the Band bean. Enter the bands shown in the picture below or add your favorite bands.



- Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



#### Related Topics

Getting Started: Java Controls

Getting Started: Web Applications

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



## Part Two: Defining an Entity Relationship

In part one of the tutorial you imported the Band entity bean, created the Music session bean, which referenced the Band bean, and ran two page flows that called the Music bean to add and list bands. In part two of the tutorial you will add a second entity bean called Recording and define the relationship between the Recording and Band beans. After updating the session bean to enable client applications to add a recording and receive a listing of recordings for a given band, you will run a page flow to add and list bands and recordings. Finally, you will create an EJB control and a test web service to test the Band bean directly.

### Steps in Part Two

#### Step 5: Create a New Entity Bean

In this step you create the entity bean Recording and add an entity relation between the Band and Recording beans.

#### Step 6: Update the Band Entity Bean

In this step you update the Band bean and add component methods to add a recording a get a list of recordings.

#### Step 7: Update the Session Bean

In this step you add business methods to the Music bean that invoke the new methods defined for the Band bean.

#### Step 8: Run a Web Application

In this step you will test the EJBs by running a new client application.

#### Step 9: Build and Run a Web Service Application

In this step you will develop an EJB control and generate a test web service to test the Band bean.

#### Related Topics

#### Developing Enterprise JavaBeans

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



## Step 5: Create a New Entity Bean

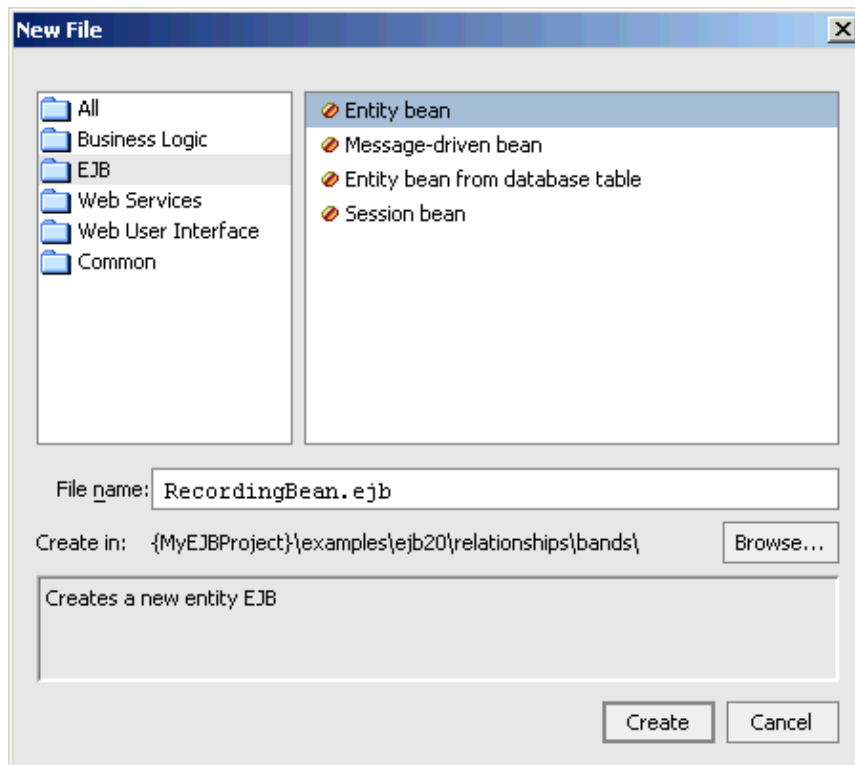
In part one of the tutorial you created the entity bean *Band* by importing it from an EJB JAR file. It is also possible to generate an entity bean from an existing database table. In this step, however, you will develop your own entity bean from scratch.

The tasks in this step are:

- To Create a Entity Bean
- To Change the Data Source
- To Define CMP Fields
- To Add EJBCreate
- To Add a Relation
- To Build the EJBs

To Create an Entity Bean

1. Right-click the *bands* folder in the *Application* tab, and select *New*—>*Entity Bean*.
2. Enter the file name *RecordingBean.ejb*.



3. Click *Create*.

To Change the Data Source

You need to point the Recording bean to the data source that is defined in the workshop domain:

1. Ensure that the Recording EJB is displayed in Design View.
2. In the **Property Editor**, in the **General** section, locate data–source–name and change the entry to cgSampleDataSource.

Notice in same section of the **Property Editor** that the **table–name** is recordings. Also notice in the **Naming** section that the local interfaces and the local JNDI name have been automatically defined, as shown below.

Naming	
Remote EJB	<input type="checkbox"/>
Local EJB	<input checked="" type="checkbox"/>
JNDI name	ejb.RecordingLocalHome
Bean class name	Recording
Home class name	RecordingHome

#### To Define CMP Fields

Now you define the container–managed persistence (CMP) fields for the Recording EJB. Remember that CMP fields are virtual fields that are not defined in the entity bean itself but correspond to columns in the database table. The entity bean implements the accessor (getter and setter) methods to read and write to the table. By default these accessor methods are also added to the bean's local (business) interface. You will define two primary key fields that hold the name of the recording and the recording band:

1. Ensure that the Recording EJB is displayed in Design View.
2. Right–click the RecordingBean and choose **Add CMP field**. A CMP field called String field1 appears in the **CMP Fields** section.
3. Rename the field String bandName. If you step off the field and need to rename it, right–click the field and select **Change declaration**.

**Note.** If you stepped off the field and used **Change declaration** to rename the field, select the bandName declaration in Design View and go to the **Property Editor** to change the ejbgen:cmp–field tag's column property to bandName. (The name of the CMP field and the corresponding column name can be different, but for consistency reasons you should give them the same name here.)

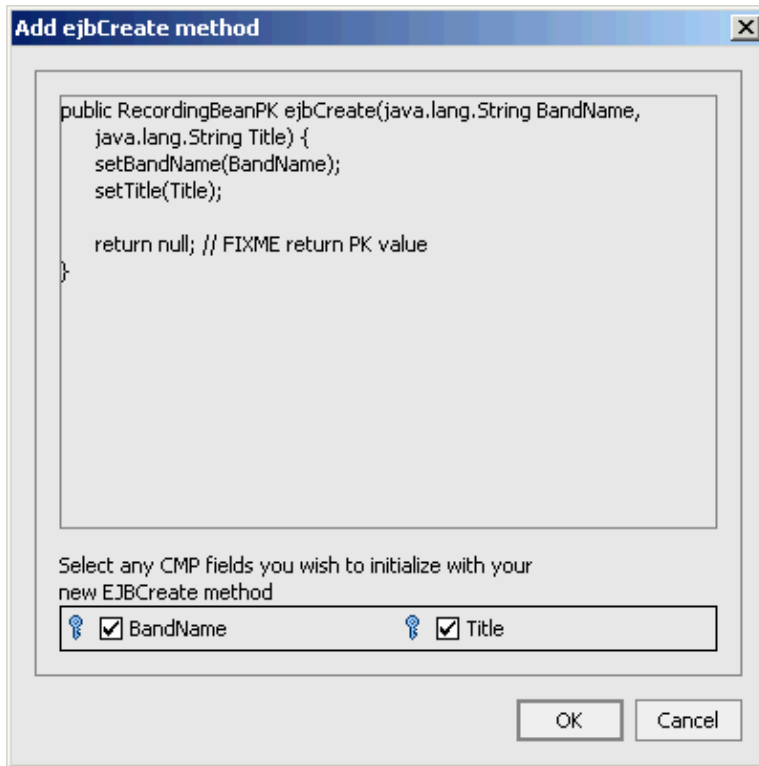
4. Right–click the field String bandName and select **Primary Key**. A key icon appears in front of the field definition.
5. Repeat these steps to create the primary key field String title.
6. Save your results.
7. Go to source view and examine the code. Notice that the accessor methods have been defined, but not the fields, and that ejbgen tags are used to tag the primary key and to define the accessor methods in the bean's local interface.

#### To Add EJBCreate

Now you will add an ejbCreate method, which when invoked creates a new Recording bean and adds the corresponding record to the database table. When you follow the steps below to define ejbCreate, WebLogic Workshop automatically adds the corresponding ejbPostCreate to the bean class definition, and defines the create method in the home interface:

1. Ensure that the Recording EJB is displayed in Design View.
2. Right–click the RecordingBean and choose **Add EJBCreate**. The **Add ejbCreate method** dialog appears.

3. In the top panel the correct definition of the `ejbCreate` method is already displayed. Verify that the method takes a band name and a title of the recording as arguments. Also verify that the order of the arguments is correct, as the code that you will add later assumes this order. Click **OK**.



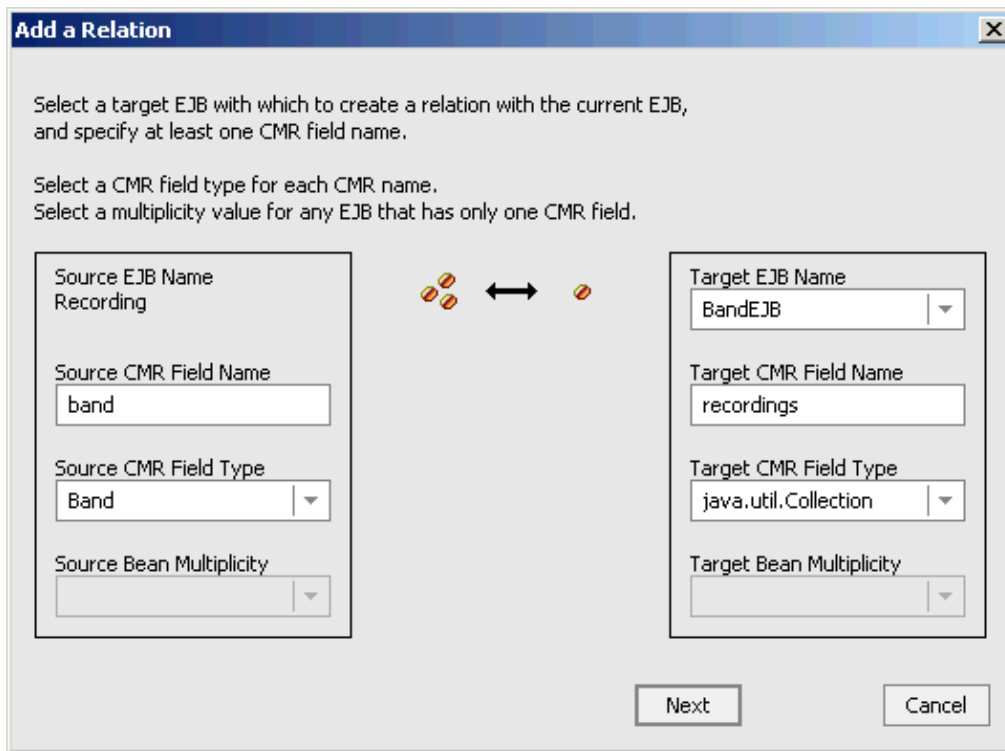
4. The method you created appears in the **EJB Create Methods** section. Click the method to go to Source View and examine the code.
5. Save your results.

### To Add a Relation

In the real world, recordings are made by bands. A band can have one or multiple recordings and, by and large, a recording is made by one band. (For now we will ignore the scenario where multiple bands contribute to a recording.) Here you will define this relationship in your Recording bean:

1. Ensure that the Recording bean is displayed in Design View.
2. Right-click the RecordingBean and choose **Add Relation**. The **Add a Relation** dialog appears.
3. From the drop-down list in the right-hand panel select BandEJB as the **Target EJB Name**.
4. In the **Source CMR Field Name** in the left-hand panel, enter band.
5. In the **Source CMR Field Type**, select Band from the drop-down list.
6. In the **Target CMR Field Name** in the right-hand panel, enter recordings.
7. In the **Target CMR Field Type**, select java-util.Collection from the drop-down list.





**Add a Relation**

Select a target EJB with which to create a relation with the current EJB, and specify at least one CMR field name.

Select a CMR field type for each CMR name.

Select a multiplicity value for any EJB that has only one CMR field.

Source EJB Name: Recording

Source CMR Field Name: band

Source CMR Field Type: Band

Source Bean Multiplicity:

Target EJB Name: BandEJB

Target CMR Field Name: recordings

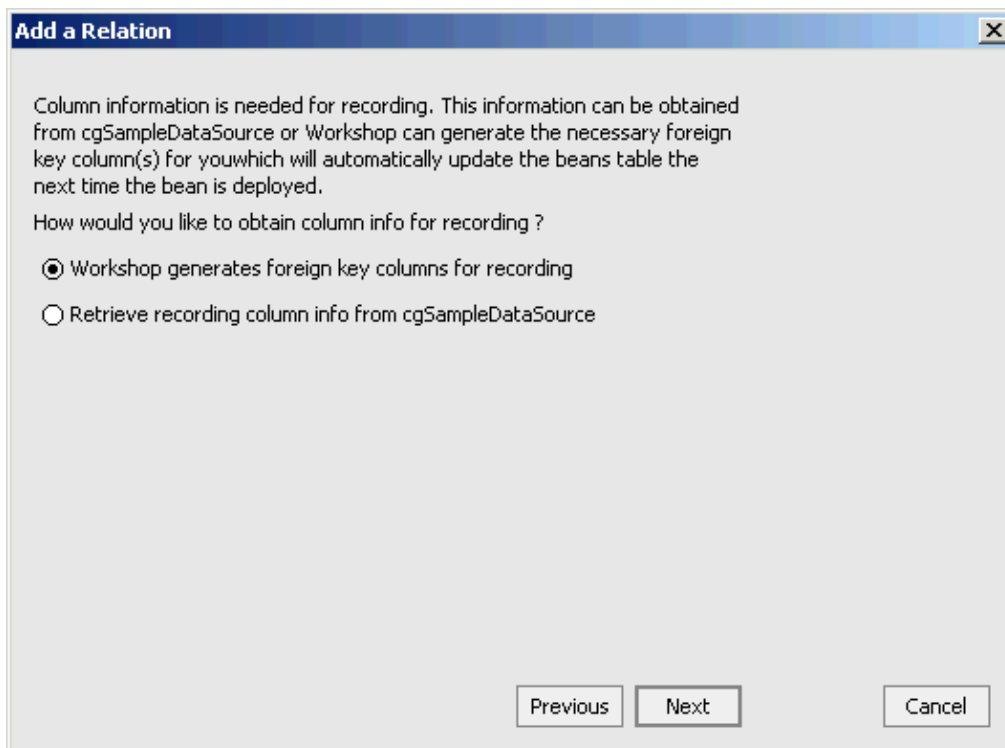
Target CMR Field Type: java.util.Collection

Target Bean Multiplicity:

Next Cancel

8. Click **Next**.

9. You can now decide whether the relationship is modeled using existing columns in the database table or whether you want WebLogic Workshop to create the necessary columns for you. Because Workshop is automatically creating the table for you, that is, the table for the Recording bean does not exist yet, select **Workshop generates foreign key columns for recording**.



**Add a Relation**

Column information is needed for recording. This information can be obtained from cgSampleDataSource or Workshop can generate the necessary foreign key column(s) for you which will automatically update the beans table the next time the bean is deployed.

How would you like to obtain column info for recording ?

☒ Workshop generates foreign key columns for recording

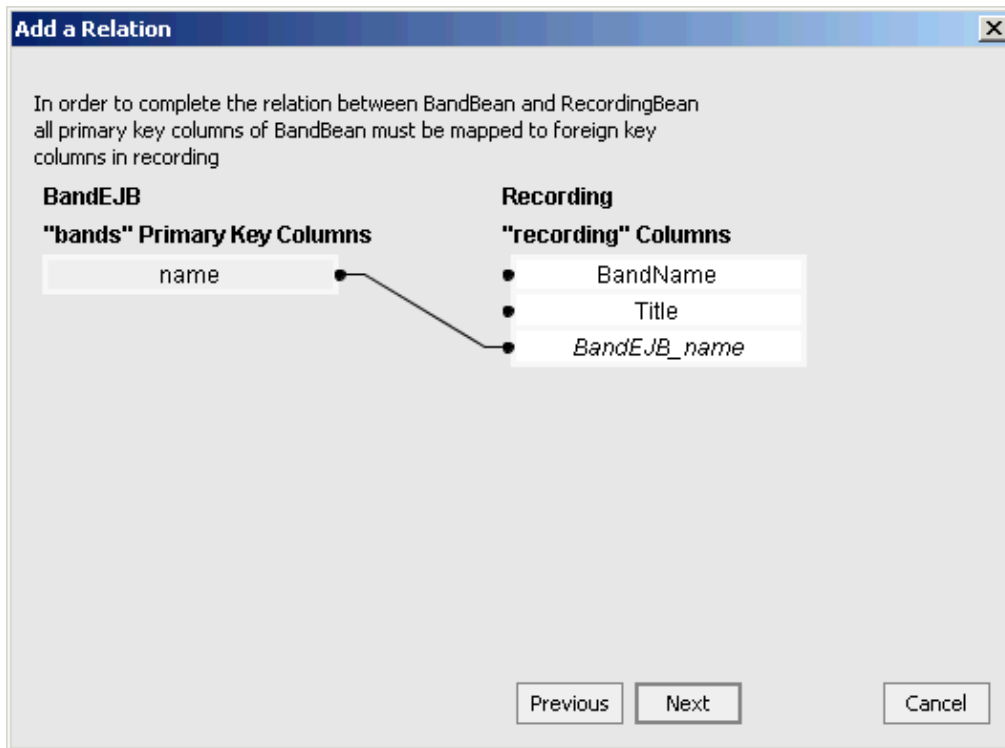
☐ Retrieve recording column info from cgSampleDataSource

Previous Next Cancel

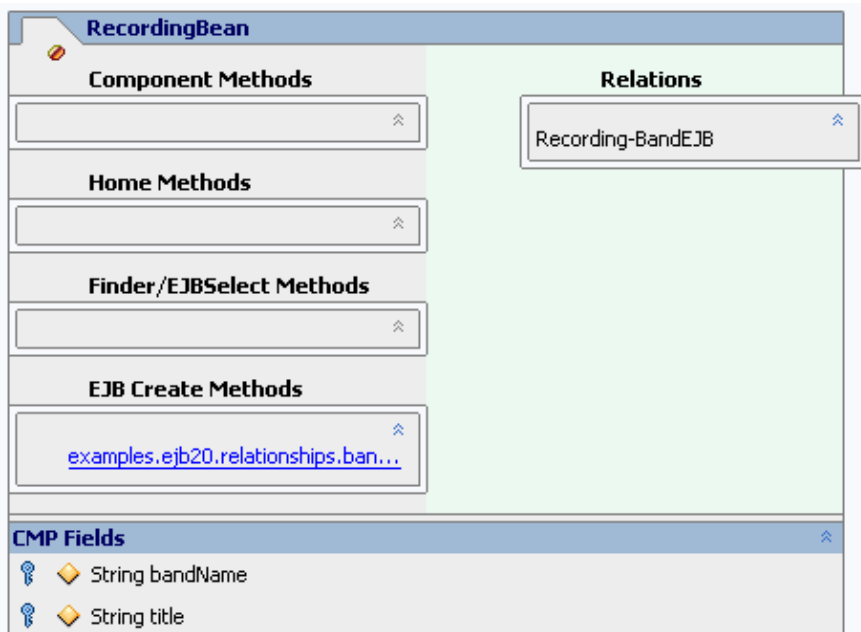
10. Click **Next**.

11. Workshop automatically maps the Band bean's primary key name to a new column BandEJB\_name in

the Recording table. Click *Next*.



12. Workshop provides a name for the relationship. Click *Finish*. The Recording–BandEJB relation appears in Design View in the *Relations* section.



13. Right-click the Recording–BandEJB relation and choose *Edit in source view* to examine the `ejbgen:relation` tag. To learn more about this tag, place the cursor anywhere in the tag and press **F1**. Also scroll down and verify that the relationship accessor methods `getBand` and `setBand` have been automatically created. Invoking the `getBand` method returns a reference to a Band bean object, that is, a reference to the band who made this recording.

14. The Recording–BandEJB relation is now also defined in the Band bean. To verify, go to the Band bean in Design View and observe that the relation is defined. Also, go to Source View to examine the ejbgen tag and the relationship accessor methods getRecordings and setRecordings. Invoking the getRecordings method returns a Java Collection containing references to all the relevant Recording beans, that is, all the recordings made by a particular band.

### To Build the EJBs

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View**—>**Windows**—>**Build**.
2. In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is created.
4. After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.
5. (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand Modules, expand MyProjectEJB.jar, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

If you encounter build errors, please run through the instructions in the current step again and verify that you have entered the correct information. A build error might be followed by a deployment error, in particular for WebAppOne\_Standard and related to an unresolved ejb-link. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error. If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step).

### Related Topics

### Entity Relationships

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



## Step 6: Update the Band Entity Bean

In the previous step you defined an entity relation between the Band and Recording beans. In this step you will use this relation and modify the Band EJB to include business methods that create a new recording for a band and return a list of recordings for a band.

The tasks in this step are:

- To Reference the Entity Bean
- To Add an `ejb-local-ref` Tag
- To Add Component Methods
- To Build the EJBs

### To Reference the Entity Bean

The Band bean will be calling methods of the Recording bean. To do so, the Band bean must first locate and obtain a reference to the Recording bean's home. In part one of the tutorial you created a similar reference in the stateless session bean Music by modifying its `ejbCreate` method, which is always called to obtain a reference to an instance of a Music bean. Unlike with session beans, the `ejbCreate` method is only invoked to create a new entity bean and is not used to obtain a reference to an existing bean; existing bean instances are returned by find methods (such as `findByPrimaryKey`). However, the `setEntityContext` callback method is always invoked for an entity bean during its life cycle, and is used in the Band bean to execute the JNDI API code to obtain a reference to the Recording bean. For more information on the life cycle methods of session and entity beans, see [The Life Cycle of a Session Bean](#) and [The Life Cycle of an Entity Bean](#).

Before you modify Band bean's `setEntityContext`, verify the name of the Recording bean's home interface:

1. In the **Application** pane, double-click `RecordingBean.ejb` and ensure you are in Design View.
2. In the **Property Editor**, locate the **Naming** section and verify that only **Local EJB** is checked.
3. Expand the **Local EJB** section and verify that the local home interface is called `RecordingHome`.

Naming	
<input type="checkbox"/> Remote EJB	<input type="checkbox"/>
<input checked="" type="checkbox"/> Local EJB	<input checked="" type="checkbox"/>
JNDI name	<code>ejb.RecordingLocalHome</code>
Bean class name	<code>Recording</code>
Home class name	<code>RecordingHome</code>

Now modify `setEntityContext` to obtain a reference to the Recording bean's local home interface:

1. In the **Application** pane, double-click `BandBean.ejb` and select the Source View tab.
2. Locate the `setEntityContext`, and modify its definition as shown in red below to obtain a reference to `RecordingHome`:

```
public void setEntityContext(EntityContext c) {  
    ctx = c;  
    try {  
        javax.naming.Context ic = new InitialContext();  
        recordingHome = (RecordingHome)ic.lookup("java:/comp/env/ejb/Recording");  
    }  
}
```

```

        catch(Exception e) {
            System.out.println("Unable to obtain RecordingHome: " + e.getMessage());
        }
    }

```

3. Go to the beginning of the class definition and define recordingHome as shown in red below:

```

abstract public class BandBean implements EntityBean
{
    private EntityContext ctx;
    private RecordingHome recordingHome;
    ...
}

```

### To Add an ejb-local-ref Tag

In the above setEntityContext method you use ejb/Recording to reference the Recording bean. At deployment time this reference is mapped to the actual location of the Recording bean. This mapping is done in the deployment descriptor. As you already saw in part one of the tutorial, in WebLogic Workshop you don't modify the deployment descriptor directly but use ejbgen tags instead. To map the reference in the Band bean to the Recording bean, you need to insert an ejbgen:ejb-local-ref tag in BandBean.ejb:

1. Ensure that the Band EJB is displayed in Design View.
2. Right-click the Design View, and select **Insert EJB Gentag**—>**ejb-local-ref**. A new ejbgen:ejb-local-ref section appears in the **Property Editor**.
3. Use the **Property Editor** to enter the following properties in their respective fields, or go to Source View to add these properties:

```
* @ejbgen:ejb-local-ref link="Recording"
```

### To Add Component Methods

In this step you will add the business methods addRecording and getRecordingValues:

1. Ensure that the Band bean is displayed in Design View.
2. Right-click the BandBean and choose **Add Component Method**. A component method called void method() appears in the **Component Methods** section.
3. Rename the method void addRecording(String recording). If you step off the method and need to rename it, right-click the method and select **Rename**.
4. Click the component method to go to Source View and modify the method as shown in red below:

```

/**
 * @ejbgen:local-method
 */
public void addRecording(String recording) throws CreateException
{
    Recording album = recordingHome.create(getName(), recording);
    Collection recordings = getRecordings();
    if(album != null) {
        recordings.add(album);
    }
}

```

Notice that the method uses the Recording bean's create method, which you defined in an earlier step, to add a new recording record to the database. Also notice that the method subsequently uses the

relationship accessor method `getRecordings` to get a collection of references to all `Recording` beans for this band, and then adds the new recording to this collection. This last step ensures that the `BandEJB_name` column in the `Recording` table is set to the `Band` bean's primary key value. In other words, this ensures that the entity relation between the two beans is now also set for the new `Recording`, and that a subsequent call to the relationship accessor method `getRecordings` will return a list of recordings that includes the new recording.

5. Now repeat these steps for the method `getRecordingValues`. Go to Design View, right-click the `BandBean` and choose **Add Component Method**. A component method called `void method1()` appears in the **Component Methods** section.
6. Rename the method `Collection getRecordingValues()`.

**Note.** Although you might be tempted to call this method `getRecordings()` instead, remember that earlier you already defined a (relationship accessor) method `Collection getRecordings()`.

7. Click the component method to go to Source View and modify the method as shown in red below:

```
/**
 * @ejbgen:local-method
 */
public Collection getRecordingValues()
{
    Collection result = new ArrayList();
    try {
        Collection fields = getRecordings();
        Iterator i_recordings = fields.iterator();
        while (i_recordings.hasNext()) {
            Recording one_recording = (Recording)i_recordings.next();
            result.add(one_recording.getTitle());
        }
    }
    catch(Exception e)
    {
        System.out.println("error getting recording infos: " + e.getMessage());
    }
    return result;
}
```

Notice that the method uses the relationship accessor method `getRecordings` to get a collection of all `Recording` beans related to this band. For each `Recording` bean it uses the method `getTitle` to retrieve the name of the recording. The method returns the collection of recording titles.

**Note.** Instead of returning a collection of recording titles, a modeling alternative would be to return a collection of `Recording` beans instead, and have the calling method determine which specific properties it needs.

8. Save your results.

### To Build the EJBs

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View—>Windows—>Build**.
2. In the **Application** pane, right-click `MyEJBProject`, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR `MyProjectEJB.jar` is created.
4. After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until

## WebLogic Workshop Tutorials

the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.

5. (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand Modules, expand MyProjectEJB.jar, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

If you encounter a build error related to the `ejbgen:ejb-local-ref` tag, make sure that you have specified the Recording bean's name correctly. If you encounter build errors related to the component methods, verify that the methods have been defined correctly. A build error might be followed by a deployment error, in particular for `WebAppOne_Standard` and related to an unresolved `ejb-link`. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error. If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step).

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



## Step 7: Update the Session Bean

The Music session bean will be called by a page flow application to work with band and recording data. Here you will add component methods to work with recording data. The tasks in this step are:

- To Add Component Methods
- To Build the EJBs

### To Add Component Methods

In this step you add the corresponding business methods `addRecording` and `getRecordings` to the Music bean:

1. Ensure that the Music bean is displayed in Design View.
2. Right-click the MusicBean and choose **Add Component Method**. A component method called `void method()` appears in the **Component Methods** section.
3. Rename the method `void addRecording(String band, String recording)`.
4. Right-click the arrow to the left of the component method and select **Local**. The business method is now defined in the local interface.
5. Click the component method to go to Source View and modify the method as shown in red below:

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

Notice that the method uses the Band bean's `findByPrimaryKey` method to locate the band bean and uses the band bean's method `addRecording` to add the recording to the database.

6. Now repeat the above steps for the `getRecordings` method. Go to Design View, right-click the MusicBean and choose **Add Component Method**. A component method called `void method1()` appears in the **Component Methods** section.
7. Rename the method `Collection getRecordings(String band)`.
8. Right-click the arrow to the left of the component method and select **Local**.
9. Click the component method to go to Source View and modify the method as shown in red below:

```
/**
 * @ejbgen:local-method
 */
public Collection getRecordings(String band) {
    Collection result = new ArrayList();
    try {
        Band theBand = bandHome.findByPrimaryKey(new BandPK(band));
```



## WebLogic Workshop Tutorials

```
        result.addAll(theBand.getRecordingValues());
    }
    catch(Exception e) {
        System.out.println("error getting recordings from music bean: " + e.getMessage());
    }
    return result;
}
```

Notice that the method uses the Band bean's `findByPrimaryKey` to locate the band bean and uses the band bean's method `getRecordingValues` to receive a list of the band's recordings.

10. Save your results.

### To Build the EJBs

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View**—>**Windows**—>**Build**.
2. In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is created.
4. After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.
5. (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand Modules, expand MyProjectEJB.jar, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

If you encounter build errors related to the component methods, verify that the methods have been defined correctly. A build error might be followed by a deployment error, in particular for WebAppOne\_Standard and related to an unresolved ejb-link. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error. If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step).

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



## Step 8: Run a Web Application

Now you will run the predefined web application `WebAppStepTwo`. This page flow uses an EJB control to reference the Music bean, and is based on page flow `WebAppStepOne_Controls`, which you ran in part one of the tutorial. Before you run this page flow, you first might want to examine the structure of this web application.

To Examine the EJB Control

- In the **Application** pane, double-click `WebAppStepTwo`, double-click `EJBControls`, and double-click the EJB control `MusicBeanControl.jcx`.
- Go to Source View and verify that the control references the Music bean. Notice that this definition is exactly the same as the one used in part one of the tutorial in `WebAppStepOne_Controls`.

To Examine the Page Flow

- In the **Application** pane, locate `WebAppStepTwo` and double-click `Controller.jpf`. The file opens in Flow View.
- Notice that the page flow begins by going to the JSP page `addBand.jsp`. Also notice the action `addABand`, which after executing returns to `addBand.jsp`. In addition, a `selectBand` action leads to the JSP page `addRecording.jsp`. This page invokes the actions `addARecording`, which returns to the same page, and `goToBands`, which returns to `addBand.jsp`.
- (Optional.) Examine the two JSPs and/or the source code of the actions.

To Run the Page Flow

- Make sure that `Controller.jpf` is opened.
- Click the **Start** button.



Workshop launches the Workshop Test Browser.

- Enter a band name and press the **Submit** button. This will call the Music EJB's `addBand` method, which will add the band to the database using the Band bean. A listing of known bands appears, which is obtained by calling the `getBands` method.
- Select a band from the list and add a recording. This will call the Music EJB's `addRecording` method, which will add the recording to the database via the Band bean and the Entity bean. A new record will contain the title of the recording as well as the recording band. A listing of known recordings appears for the selected band, which is obtained by calling Music bean's `getRecordings` method.
- Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



Related Topics

Tutorial: Java Control

Tutorial: Page Flow

EJB Control

Developing Web Applications

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



## Step 9: Build and Run a Web Service Application

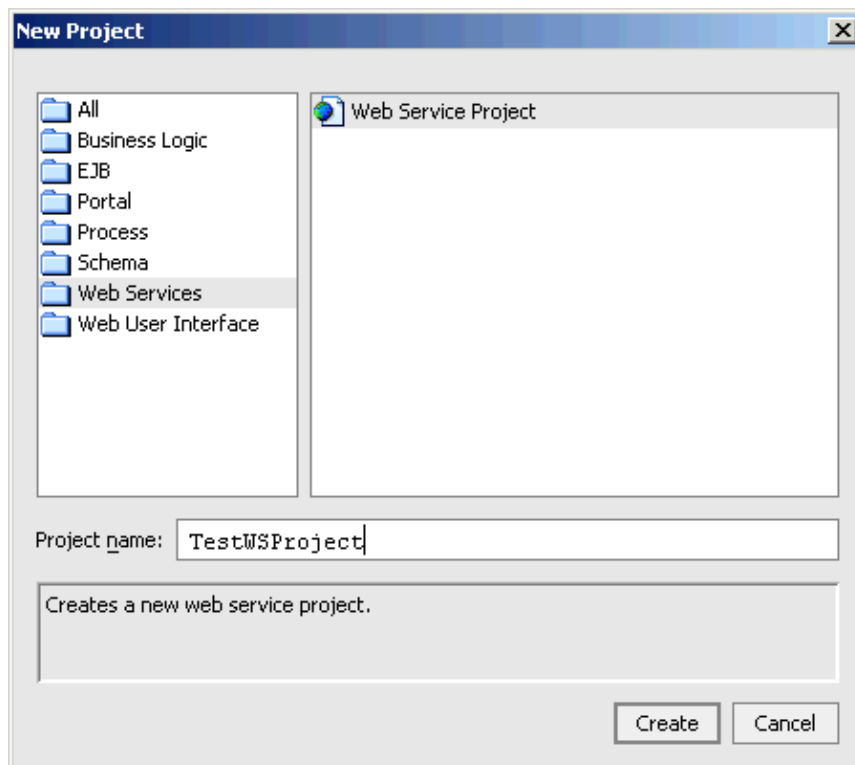
Throughout this tutorial you have used predefined web applications to test the EJBs you develop. However, as you go about developing your own EJBs, you might not be interested in building a page flow in parallel to run and test your EJBs. WebLogic Workshop offers the test web service as a quick alternative to test your beans. In this step you will build your own test web service for the Band bean. This requires generating an EJB control for the EJB, and generating a test web service for the EJB control

The tasks in this step are:

- To Create a Web Service Project
- To Create the EJB Control
- To Generate the Test Web Service
- To Run the Test Web Service

To Create the Web Service Project

1. In the **Application** tab, right-click the **EJBProjectTutorial** folder and select **New-->Project**.
2. In the **New Project** dialog, in the upper left-hand pane, select **Web Services**.  
In the upper right-hand pane, select **Web Service Project**.  
In the **Project Name** field, enter TestWSProject.



3. Click **Create**.

To Create the EJB Control

1. Right-click TestWSProject and select **New**—>**Folder**. The **Create New Folder** dialog appears.
2. Enter EJBControls. Click **OK**.
3. Right-click EJBControls and select **New**—>**Java Control**. The **New Java Control** dialog appears.
4. Select EJB Control. In the **File name** field, enter BandEJBControl. Click **Next**.
5. Click **Browse application EJBs** and select BandEJB (local jndi). Click **Select**.
6. Click **Create**.

### To Generate the Test Web Service

- In the **Application** tab, right-click BandEJBControl.jcx and select **Generate Test JWS File (Conversational)**.

### To Run the Test Web Service

1. In the **Application** tab, double-click the BandEJBControlTest.jws web service to open it in Design View.
2. Click the **Start** button.



Workshop launches the Workshop Test Browser.

3. In **Test View**, you can test the EJB using either the **Test Form** or **Test XML** tab. Click the **Test XML** tab to have access to all methods.
4. Click **startTestDrive** to begin testing. This starts the web service's conversation.
5. After **Test View** refreshes, click the **Continue this conversation** link for access to test forms and buttons through which you can test each of your control's methods.
6. You can now test the methods. After each test, click **Continue this conversation** to test another method.

For instance, locate the `findByName` method, supply the name of a band you entered earlier in the tutorial as an argument (for instance, `<arg0>Art Ensemble of Chicago</arg0>`), and click the **findByName** button. The test web service returns the reference to the entity bean, as seen in the **Returned from findByName** section, and stores this reference as part of the conversation state. In other words, you can now use another method to test this bean. For instance, click **Continue this conversation**, locate the `getName` method, and click the corresponding **getName** button. The name of the entity bean, which you located with `findByName`, is returned, as showing in the **Returned from getName** section.

**Note.** For some methods, such as `findByName` in the example, you will notice that a value is returned correctly, although an `XMLEncodingException` error is thrown in the service response. These types of errors result from the return type not being `Serializable`, which is a prerequisite for a proper XML response. However, these errors do not affect the operations of the EJB or your ability to use the test web service to test the EJB.

7. When you are finished testing, click **finishTestDrive** to finish testing.
8. Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



## WebLogic Workshop Tutorials

When you make changes to the Band bean during testing, you will need to rebuild the bean and, if you added/removed methods or changed method signatures, regenerate the test JWS. However, in those cases it is not necessary to recreate the EJB control.

### Related Topics

[How Do I: Test an Enterprise JavaBean?](#)

[Building Web Services](#)

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



## Part Three: Adding a Message–Driven Bean

In part two of the tutorial you added an entity bean called Recording and modified the Band and Music beans to enable adding and listing recordings for a band. In part three of the tutorial you will add a message–driven bean that will compile rating information for recordings.

To understand what this message–driven bean is doing, let's take a step back to provide a background for the EJBs you have created thus far. Imagine that you are using this EJB application to create an online record of your favorite bands and recordings. You have created the basics of this web site but the completed functionality includes the ability to enter copious notes on bands, including the history of the band and its members, to rate and review recordings, and so forth. In addition, other parts of the EJB application run in the background to search various web sites and collect information on price, average rating, and so forth. The latter functionality is what you will build in this part of the tutorial with a message–driven bean.

It might be clear from the above description that a true and accurate implementation of the business logic of the message–driven bean will require internet search capabilities. At this point you are going to focus on the core functionality of sending messages and the processing of these messages by message–driven beans instead, and implement placeholder business logic for the message–driven bean, which could be fully implemented at a later time.

### Steps in Part Three

#### Step 10: Create a Message–Driven Bean

In this step you create the message–driven bean Statistics.

#### Step 11: Update the Recording Bean

In this step you update the Recording bean to store the statistics generated by the Statistics bean.

#### Step 12: Update the Music Bean

In this step you change the Music Bean to send a message that will be processed by the Statistics bean, and to make rating information stored in the Recording bean available to the client application.

#### Step 13: Run a Web Application

In this step you will test the EJBs by running a page flow.

**Note.** As an alternative to updating the Music bean, WebLogic also offers the JMS control to handle JMS messages. For instance, if for some reason it were impossible to modify the Music bean, you could create a custom control instead to encapsulate the business logic implemented in step 12. This custom control, which would be called by a client application, would call the Music bean to add a recording and subsequently call a JMS control to send a JMS message to be processed by the Statistics bean.

#### Related Topics

[Developing Enterprise JavaBeans](#)

[Working with Java Controls](#)

### JMS Control

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





## Step 10: Create a Message–Driven Bean

In this step you will create the message–driven bean `Statistics`. The tasks in this step are:

- To Create a JMS Queue
- To Create a Message–Driven Bean
- To Listen to the JMS Queue
- To Reference the Recording Bean
- To Add an `ejb–local–ref` Tag
- To Define the `onMessage` Method

### To Create a JMS Queue

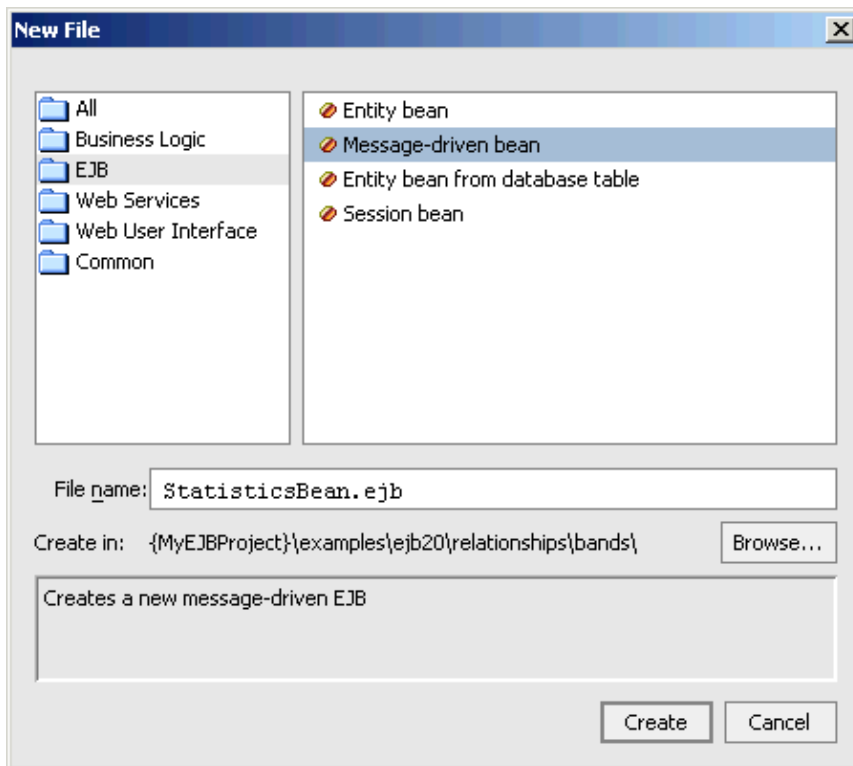
Messages for a message–driven bean are not sent directly to the bean but are delivered via the Java Message Service (JMS). To send and receive a JMS message you need to set up a new JMS Queue on the domain's JMS Server:

1. From the **Tools** menu, select **WebLogic Server—>WebLogic Console**.
2. Sign in using weblogic as *username* and *password*.
3. Under **Services Configurations, JMS**, click **Servers**.
4. Click **cgJMSServer**.
5. Click **Configure Destinations**.
6. Click **Configure a new JMS Queue**.
7. In the **Name** field, enter Recording Ratings Queue.
8. In the JNDI field, enter RecordingRatingsQueue.
9. Click the **Create** button.

**Note.** To learn more about JMS, see your favorite J2EE book or the Java documentation at <http://java.sun.com>.

### To Create a Message–Driven Bean

1. Right–click the **bands** folder in the **Application** tab, and select **New—>Message–driven Bean**.
2. Enter the file name `StatisticsBean.ejb`.



3. Click **Create**. Notice that a message-driven bean only appears in Source View.

Message-driven beans do not have a Design View. Consequently, the StatisticsBean opens in Source View.

#### To Listen to the JMS Queue

Now you will ensure that the message-driven bean listens to the JMS queue you created above:

1. Place the cursor in the @ejbgen:message-driven tag.
2. In the **Property Editor**, locate **destination-jndi-name** and change the entry to RecordingRatingsQueue.
3. Verify that the **destination-type** is javax.jms.Queue.

#### To Reference the Recording Bean

The Statistics EJB will be using the Recording bean to store a recording's statistics. To do so, the Statistics bean must first locate and obtain a reference to the Recording bean's home interface. In this step you modify the Statistics bean's ejbCreate method to locate and obtain that reference:

1. Go to the beginning of the Statistics bean class definition and define recordingHome as shown in red below:

```
public class StatisticsBean
    extends GenericMessageDrivenBean
    implements MessageDrivenBean, MessageListener
{
    private RecordingHome recordingHome;

    public void onMessage(Message msg) {
```

- ...
2. Add the `ejbCreate` method as shown in red below:

```
public class StatisticsBean
    extends GenericMessageDrivenBean
    implements MessageDrivenBean, MessageListener
{
    private RecordingHome recordingHome;

    public void ejbCreate() {
        try {
            javax.naming.Context ic = new InitialContext();
            recordingHome = (RecordingHome)ic.lookup("java:/comp/env/ejb/Recording");
        }
        catch(NamingException ne) {
            System.out.println("Encountered the following naming exception: " + ne.getMessage());
        }
    }
    ...
}
```

3. Locate the import section at the top of the file and add the following import statements, or use Alt+Enter when prompted by the IDE to automatically add these import statements:

```
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

#### To Add an `ejb-local-ref` Tag

In the above `ejbCreate` method you used `ejb/Recording` to reference the Recording EJB. To map the reference in the Statistics bean to the actual location of the Recording bean at deployment time (via the deployment descriptor), you need to insert an `ejbgen:ejb-local-ref` tag as shown in bold below:

```
* ...
* @ejbgen:ejb-local-ref link="Recording"
*/
public class StatisticsBean
...
```

#### To Define the `onMessage` Method

A message-driven bean's `onMessage` method is invoked when a message arrives for processing. In this method you implement the business logic of the bean:

1. Modify the `onMessage` method as shown in red below:

```
public void onMessage(Message msg) {
    try {
        // read the message
        MapMessage recordingMsg = (MapMessage)msg;
        String bandName = recordingMsg.getString("bandName");
        String recordingTitle = recordingMsg.getString("recordingTitle");
        // placeholder logic for the rating
        Random randomGenerator = new Random();
        String rating = new Integer(randomGenerator.nextInt(5)).toString();
        // save the rating with the recording
        Recording album = recordingHome.findByPrimaryKey(new RecordingBeanPK(bandName, r
        album.setRating(rating);
    }
}
```

```
}  
catch(Exception e) {  
    System.out.println("Encountered the following exception: " + e.getMessage());  
}  
}
```

Notice that the method uses the Recording bean's (currently undefined) `setRating` method to store the average rating for this recording. In a later step you will add this accessor method to the Recording bean. Computing the rating information for an album is done by randomly selecting a number between 0 and 4.

2. Locate the import section at the top of the file and add the following import statement, or use Alt+Enter when prompted by the IDE to automatically add the import statement:

```
import java.util.Random;
```

Because the Recording bean's `setRating` method is not defined yet, building the EJBs would fail. You first need to update the Recording bean before you can build without generating errors.

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



# Step 11: Update the Recording Bean

In this step you add a CMP field to the Recording bean to store the rating information collected by the Statistics bean, and change the way tables are created by WebLogic. The tasks in this step are:

- To Define a CMP Field
- To Change the create-tables Setting
- To Build the EJBs

## To Define a CMP Field

1. Ensure that the Recording EJB is displayed in Design View.
2. Right-click the RecordingBean and choose **Add CMP field**. A CMP field called String field1 appears in the **CMP Fields** section.
3. Rename the field String rating. If you step off the field and need to rename it, right-click the field and select **Change declaration**.

**Note.** If you stepped off the field and used **Change declaration** to rename the field, select the rating declaration in Design View and go to the **Property Editor** to change the ejbgen:cmp-field tag's column property to rating. (The name of the CMP field and the corresponding column name can be different, but for consistency reasons you should give them the same name here.)

## To Change the create-tables Setting

When you first used the Recording bean, WebLogic automatically created the corresponding table with the necessary columns to hold the defined CMP fields, as dictated by the CreateOnly setting in the **create-tables** property in the **Property Editor**. This setting does not automatically re-create the table if the definition of CMP fields changes, that is, if you add a new CMP field or change the definition of an existing CMP field. Since you have added an additional CMP field above, you must either manually add the column to the table or tell WebLogic to drop the original table and create a new table. You will do the latter here:

1. Ensure that the Recording EJB is displayed in Design View.
2. In the **Property Editor**, locate the **create-tables** property under **JAR Settings**. From the drop-down list choose **DropAndCreate**. This option will re-create the table whenever the definition of the entity bean no longer matches the table definition. All data stored in the old table will be lost.
3. Save your results.

## To Build the EJBs

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View-->Windows-->Build**.
2. In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is created.
4. After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.

5. (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand Modules, expand MyProjectEJB.jar, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

If you encounter a build error related to the `ejbgen:ejb-local-ref` tag in Statistics bean, make sure that you have specified the Recording bean's name correctly. If you encounter build errors related to the component methods, verify that the methods have been defined correctly. A build error might be followed by a deployment error, in particular for `WebAppOne_Standard` and related to an unresolved `ejb-link`. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error. If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step).

Also, you can compare your code for Statistics Bean and Recording bean with the solution source code located at `[BEA_HOME]\weblogic81\samples\platform\tutorial_resources\EJBTutorial\solutions`.

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



## Step 12: Update the Music Bean

The Music bean will send a message to the JMS queue whenever a new recording is added by the user. In this step you will modify the component method `addRecording` to send the message. Also you will add the component method `getRating` to retrieve the rating information of a recording from the Recording bean.

The tasks in this step are:

- To Reference the Recording Bean
- To Add an `ejb-local-ref` Tag
- To Add Component Method `getRating`
- To Reference the JMS Provider and Destination
- To Add `resource-ref` and `resource-env-ref` Tags
- To Modify Component Method `addRecording`
- To Build the EJBs

### To Reference the Recording Bean

The Music EJB will be invoking a method of the Recording EJB. To do so, it must first locate and obtain a reference to the Recording bean's home. In this step, you modify the `ejbCreate` method in `MusicBean.ejb` to locate and obtain a reference:

1. In the **Application** pane, double-click `MusicBean.ejb` and select the Source View tab.
2. Locate the `ejbCreate` method and modify its definition as shown in red below:

```
public void ejbCreate() {
    try {
        javax.naming.Context ic = new InitialContext();
        bandHome = (BandHome)ic.lookup("java:comp/env/ejb/BandEJB");
        recordingHome = (RecordingHome) ic.lookup("java:comp/env/ejb/Recording");
    }
    catch (NamingException ne) {
        throw new EJBException(ne);
    }
}
```

3. Go to the beginning of the class definition and define `recordingHome` as shown in red below:

```
public class MusicBean extends GenericSessionBean implements SessionBean {
    private BandHome bandHome;
    private RecordingHome recordingHome;
    ...
}
```

### To Add an `ejb-local-ref` Tag

In the above `ejbCreate` method you used `ejb/Recording` to reference the Recording EJB. To map the reference in the Music bean to the actual location of the Recording bean at deployment time (via the deployment descriptor), you need to insert an `ejbgen:ejb-local-ref` tag:

1. Ensure that the Music EJB is displayed in Design View.
2. Right-click the Design View, and select **Insert EJB Gentag**—>**ejb-local-ref**. A new `ejbgen:ejb-local-ref` section appears in the **Property Editor**.

3. Use the **Property Editor** to enter the link property, or go to Source View to add this property:

```
* @ejbgen:ejb-local-ref link="Recording"
```

#### To Add Component Method getRating

1. Ensure that the Music EJB is displayed in Design View.
2. Right-click the MusicBean and choose **Add Component Method**. A component method called void method() appears in the **Component Methods** section.
3. Rename the method String getRating(String band, String recordingTitle).
4. Right-click the arrow to the left of the component method and select **Local**. The business method is now defined in the local interface.
5. Click the component method to go to Source View and modify the method as shown in red below:

```
/**
 * @ejbgen:local-method
 */
public String getRating(String band, String recordingTitle)
{
    try {
        Recording theRecording = recordingHome.findByPrimaryKey(new RecordingBeanPK(band, recordingTitle));
        return theRecording.getRating();
    }
    catch(FinderException fe) {
        throw new EJBException(fe);
    }
}
```

Notice that the method uses the Recording bean's `findByPrimaryKey` method to locate the recording and uses the recording bean's method `getRating` to retrieve the rating.

#### To Reference the JMS Provider and Destination

In order to send a JMS message, the Music bean will need to connect to the JMS provider and it will need to have a destination queue for the message. Next you will modify the `ejbCreate` method to locate and obtain a reference to a `QueueConnectionFactory` that is defined by default in the workshop domain, and the JMS Queue you defined in step 10 of the tutorial:

1. In the **Application** pane, double-click `MusicBean.ejb` and ensure you are in Source View.
2. Locate the `ejbCreate` method in `MusicBean.ejb`, and modify its definition as shown in red below:

```
public void ejbCreate() {
    try {
        javax.naming.Context ic = new InitialContext();
        bandHome = (BandHome)ic.lookup("java:comp/env/ejb/BandEJB");
        recordingHome = (RecordingHome) ic.lookup("java:comp/env/ejb/Recording");
        factory = (QueueConnectionFactory) ic.lookup("java:comp/env/jms/QueueConnectionFactory");
        queue = (Queue) ic.lookup("java:comp/env/jms/RecordingRatingsQueue");
    }
    catch (NamingException ne) {
        throw new EJBException(ne);
    }
}
```

3. Locate the import section at the top of the file and add the following import statements, or use Alt+Enter when prompted by the IDE to automatically add these import statements:



```
import javax.jms.Queue;
import javax.jms.QueueConnectionFactory;
```

4. Go to the beginning of the class definition and define QueueConnectionFactory and Queue as shown in red below:

```
public class MusicBean
    extends GenericSessionBean
    implements SessionBean
{
    private BandHome bandHome;
    private RecordingHome recordingHome;
    private QueueConnectionFactory factory;
    private Queue queue;
    ...
}
```

#### To Add resource-ref and resource-env-ref Tags

In the above step you used `jms/QueueConnectionFactory` to reference the JMS provider and you used `jms/RecordingRatingsQueue` to locate the JMS queue. To map these reference to the actual locations at deployment time (via the deployment descriptor), you need to insert `resource-ref` and `resource-env-ref` tags:

1. Ensure that the Music EJB is displayed in Design View.
2. Right-click the Music bean and choose **Insert EJBGentag**—>**resource-ref**. A new `ejbgen:resource-ref` section appears in the **Property Editor**.
3. Define the reference to the JMS provider in the **Property Editor** by entering the following properties in the respective fields, or go to Source View to add these properties:
 

```
* @ejbgen:resource-ref
*   auth="Container"
*   jndi-name = "weblogic.jws.jms.QueueConnectionFactory"
*   name = "jms/QueueConnectionFactory"
*   type="javax.jms.QueueConnectionFactory"
```
4. In Design View, right-click the Music bean and choose **Insert EJBGentag**—>**resource-env-ref**. A new `ejbgen:resource-env-ref` section appears in the **Property Editor**.
5. Define the local link in the **Property Editor** by entering the following properties in the respective fields, or go to Source View to add these properties:

```
* @ejbgen:resource-env-ref
*   jndi-name="RecordingRatingsQueue"
*   name="jms/RecordingRatingsQueue"
*   type = "javax.jms.Queue"
```

**Note.** You can verify the JNDI name of the QueueConnectionFactory via the WebLogic console. You will find the link *Connection Factories* under **Services Configurations, JMS**. For more information, see step 10 of the tutorial. To learn more about these tags, place your cursor inside the tag in Source View, and press **F1**.

#### To Modify Component Method addRecording

1. Ensure that the Music bean is displayed in Design View.
2. Click the method void addRecording(String band, String recording) to go to Source View and modify the method as shown in red below:

```
/**
 * @ejbgen:local-method
 */
public void addRecording(String band, String recording)
```

```

{
    try {
        Band bandBean = bandHome.findByPrimaryKey(new BandPK(band));
        if(bandBean != null) {
            bandBean.addRecording(recording);
            //send a message
            QueueConnection connect = factory.createQueueConnection();
            QueueSession session = connect.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
            QueueSender sender = session.createSender(queue);
            MapMessage recordingMsg = session.createMapMessage();
            recordingMsg.setString("bandName", band);
            recordingMsg.setString("recordingTitle", recording);
            sender.send(recordingMsg);
            connect.close();
        }
    }
    catch(CreateException ce) {
        throw new EJBException(ce);
    }
    catch(FinderException fe) {
        throw new EJBException(fe);
    }
    catch(JMSEException ne) {
        throw new EJBException(ne);
    }
}

```

Notice that the method creates a message containing the name of the band and the recording title, and sends this to the JMS queue.

3. Locate the import section at the top of the file and add the following import statements, or use Alt+Enter when prompted by the IDE to automatically add these import statements:

```

import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.QueueConnection;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;

```

4. Save your results.

### To Build the EJBs

1. Locate the **Build** window in the IDE. If the **Build** window is not there, select **View—>Windows—>Build**.
2. In the **Application** pane, right-click MyEJBProject, and select **Build MyEJBProject**.
3. Monitor the **Build** window and verify that the build completes correctly, and that the EJB JAR MyProjectEJB.jar is created.
4. After the build completes, the EJBs are deployed: Watch the green ball in the WebLogic Server status bar at the bottom of WebLogic Workshop turn yellow with the message *Updating Server*. Wait until the ball turns green again. The two EJBs are now deployed on the server. In addition, the various web applications that were predefined are deployed to the web container. We will turn to these in the next step.
5. (Optional.) Verify the contents of the EJB JAR. In the **Application** pane, expand Modules, expand MyProjectEJB.jar, and expand the directory structure reflecting the package name. Verify that the bean classes and the various interfaces have been created.

## WebLogic Workshop Tutorials

If you encounter build errors, verify that the above instructions have been followed. A build error might be followed by a deployment error, in particular for WebAppOne\_Standard and related to an unresolved ejb-link. The link cannot be resolved because the EJB JAR was not created. Fixing the build problem should resolve this deployment error. If you encounter deployment errors that seem to be related to one of the web applications, make sure that you build the EJB project and not the entire application (because the application contains page flow applications that have different EJB dependencies from what you have created in this step).

Also, you can compare your code for Music Bean with the solution source code located at [BEA\_HOME]\weblogic81\samples\platform\tutorial\_resources\EJBTutorial\solutions.

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



## Step 13: Run a Web Application

In this step you will run the predefined web application WebAppStepThree. This page flow uses the same EJB control as WebAppStepTwo, and uses the same JSPs and action methods to add bands and recordings and receive listings of bands and recordings. In addition, it is possible to review the rating of a recording.

To Examine the Page Flow

- In the **Application** pane, locate WebAppStepThree and double-click Controller.jspf. The file opens in Flow View.
- Notice that this page flow also contains the JSP page recordingStats.jsp and the action methods goToStats and goToRecordings.
- (Optional.) Examine the JSP and/or the source code of the actions.

To Run the Page Flow

- Make sure that Controller.jspf is opened.
- Click the **Start** button.



Workshop launches the Workshop Test Browser.

- On the first page, locate the list of bands that you created in earlier parts of the tutorial, and click a band to go to the recordings of this band. Because the Recordings table was re-created, the recordings that you entered previously are gone.
- Enter a new recording. This will call the Music EJB's addRecording method, which will add the recording to the database via the Band bean and the Entity bean. The record that is added to the database initially contains the title of the recording, the recording band, and the entity relation between the band and recording stored in the BandEJB\_name column. In addition, the addRecording method sends a message to the JMS queue. When the Statistics bean receives the message, it will create a rating for this recording and add that to the table via the Recording bean. The operations of a message-driven bean are asynchronous; after the addRecording method sends the message to the JMS queue, control immediately returns to the calling page flow action method addARecording.
- Click a recording to view the rating. This will call the Recording bean's getRating method, which will return a rating, or null if the Statistics bean has not processed the JMS message yet.
- Return to **WebLogic Workshop** and press the **Stop** button to close the Test Browser.



**Note.** If you receive a NamingException when the page flow starts, the EJB container is unable to locate and reference the Recording bean, the JMS provider, and/or the JMS queue. Go back to step 12 of the tutorial to verify the references, and use WebLogic console to verify the JNDI names of the JMS provider and queue. When you make changes to EJBs, you must rebuild these before running the page flow again.

Related Topics

Tutorial: Java Control

Tutorial: Page Flow

EJB Control

Developing Web Applications

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



# Summary: Enterprise JavaBean Tutorial

This tutorial introduced you to the basics of building Enterprise JavaBeans.

## Concepts and Tasks Introduced in This Tutorial

- EJBs that weren't originally built in WebLogic Workshop can be imported into EJB project
- EJB project uses wizards to facilitate development of EJBs, for instance when defining `ejbCreate` methods or entity relations. Developing entity relations is abstracted through the Wizard such that you can focus on design, with Workshop taking care of the implementation details.
- The use of message-driven beans for asynchronous processing.
- WebLogic Workshop offers EJB controls for easy interoperability with page flows and web services.
- WebLogic Workshop enables you to build EJBs that are standards compliant and contain no proprietary elements.

Click the arrow to navigate through the tutorial:

