

# Getting Started with Web Services

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. The standard technologies underlying web services are defined by the World Wide Web Consortium.

Web services are a useful way to provide data to an array of consumers over the Internet, like stock quotes and weather reports. But they take on a new power in the enterprise, where they offer a flexible solution for integrating distributed systems, whether legacy systems or new technology. Workshop for WebLogic makes it easy for you to build and deploy applications that provide or access web services.

## Current Release Information:

- [What's New in 9.2](#)
- [Upgrading from 8.1](#)

## Useful Links:

- [Tutorials](#)
- [Tips and Tricks](#)

## Other Resources:

- [Online Docs](#)
- [Dev2Dev](#)
- [Discussion Forums](#)
- [Development Blogs](#)

## Topics Included in This Section

### Tutorial: Web Service

Describes the basic steps for creating a simple web service and testing it.

### Tutorial: Advanced Web Services

Demonstrates additional techniques for working with web services.

### Introduction to Web Service Technologies

Discusses the standard technologies underlying web services.

### Building Web Services with Workshop for WebLogic

Describes the basic components of a web service built with Workshop for WebLogic.

### Using Design View to Create Web Services

Describes how to use the web service Design View.

### Web Service Development Starting Points

Provides an overview of the different design scenarios: (1) starting from a WSDL, (2) starting from an XML Schema, and (3) starting from Java.

## **Testing Web Services with the Test Client**

Provides an overview of testing, debugging, and deploying a web service.

## **WSDL Files: Web Service Descriptions**

Discusses how WSDL files are used to describe web service interfaces.

## **Web Service Dialogs**

These topics explain the web service related UI dialogs and wizards.

## **Related Topics**

### **Designing Asynchronous Interfaces**

---

©2000-2006 BEA Systems, Inc. All Rights Reserved

## Tutorial: Web Service

This tutorial is subdivided into two parts. The first part shows you how to build a web service project containing a simple web service. The second part builds on the project and web service created in the first part by adding a simple custom control that calls methods on existing controls.

You will also learn how to use the Web Service Design View, a graphical editor for creating web services.

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

### Part I : Simple Web Service

In the first part of the tutorial, you will use the IDE to build a web project and a simple web service.

These are the steps you will follow for Part I of the tutorial:

1. Create a web service project.
2. Add a web service to the project.
3. Add an operation (web method) to the web service.
4. Test the web service.

### Part II : Web Service That Calls Methods on Provided Controls

In the second part of the tutorial, you will create a custom control that calls methods on pre-existing controls that are provided to you. You will then add a method to your web service that calls a method on this custom control. The result is to return data from a sample database.

These are the steps you will follow for Part II of the tutorial:

5. Copy existing controls into the web project created in Part I.
6. Create a new custom control called `MailingListControl` that calls methods on the imported controls.
7. Add `MailingListControl` to the web service created in Part I and add an operation (method) to the web service that calls a method on `MailingListControl`.
8. Test the web service.

## Start Workshop for WebLogic

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

- On Microsoft Windows:
  - On the Start Menu, click **All Programs > BEA Products > Workshop for WebLogic Platform 9.2**
- On Linux:
  - Run `BEA_HOME/workshop92/workshop4WP/workshop4WP.sh`

## Create a New Workspace

1. In the **Workspace Launcher** dialog, click the **Browse** button.
2. In the **Select Workspace Directory** dialog, navigate to a directory of your choice and click **Make New Folder**.
3. Name the new folder `webSvcTutorial`, press the **Enter** key and Click **OK**.
4. In the **Workspace Launcher** dialog, click **OK**.

Click the arrow to navigate through the tutorial:

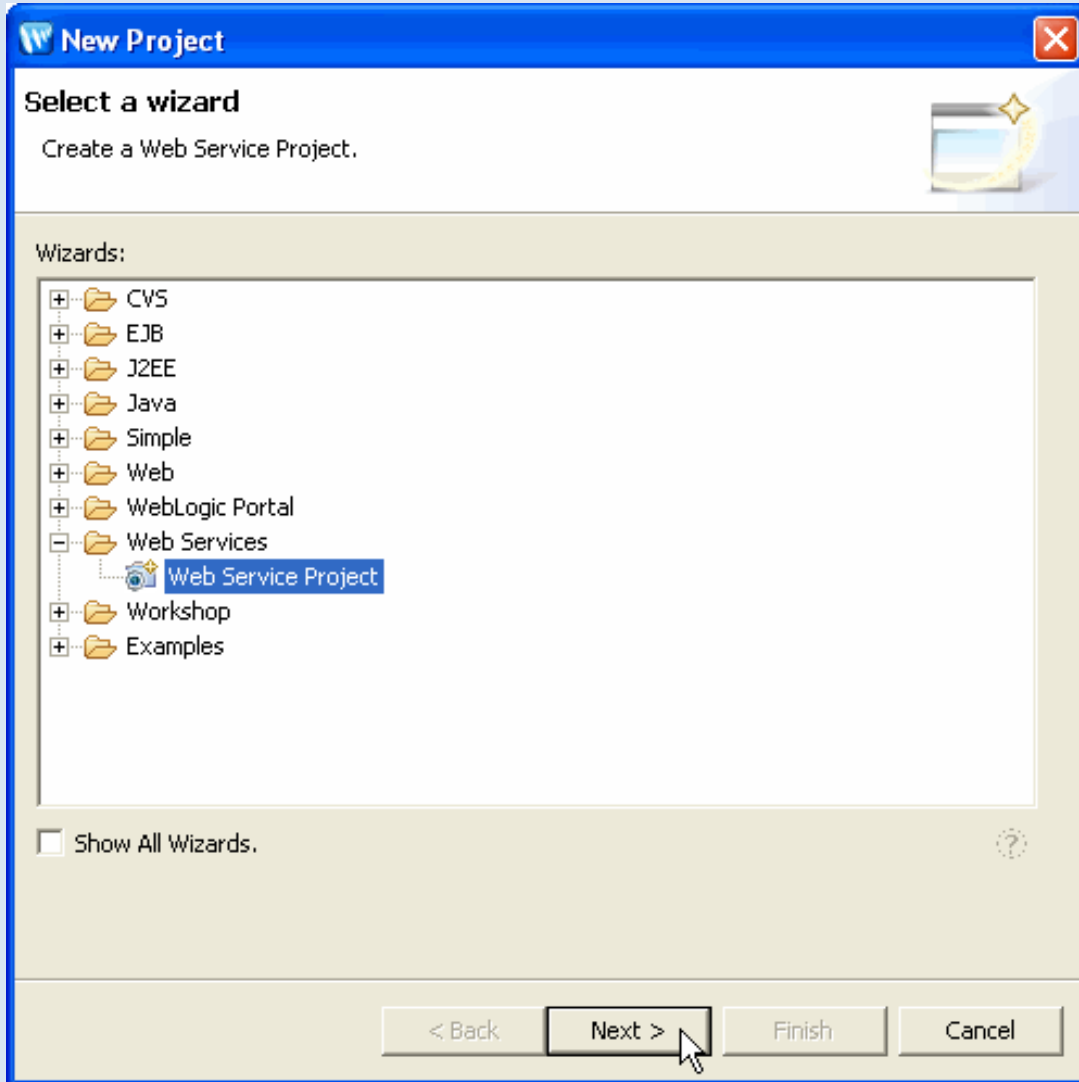


## Part I : Simple Web Service

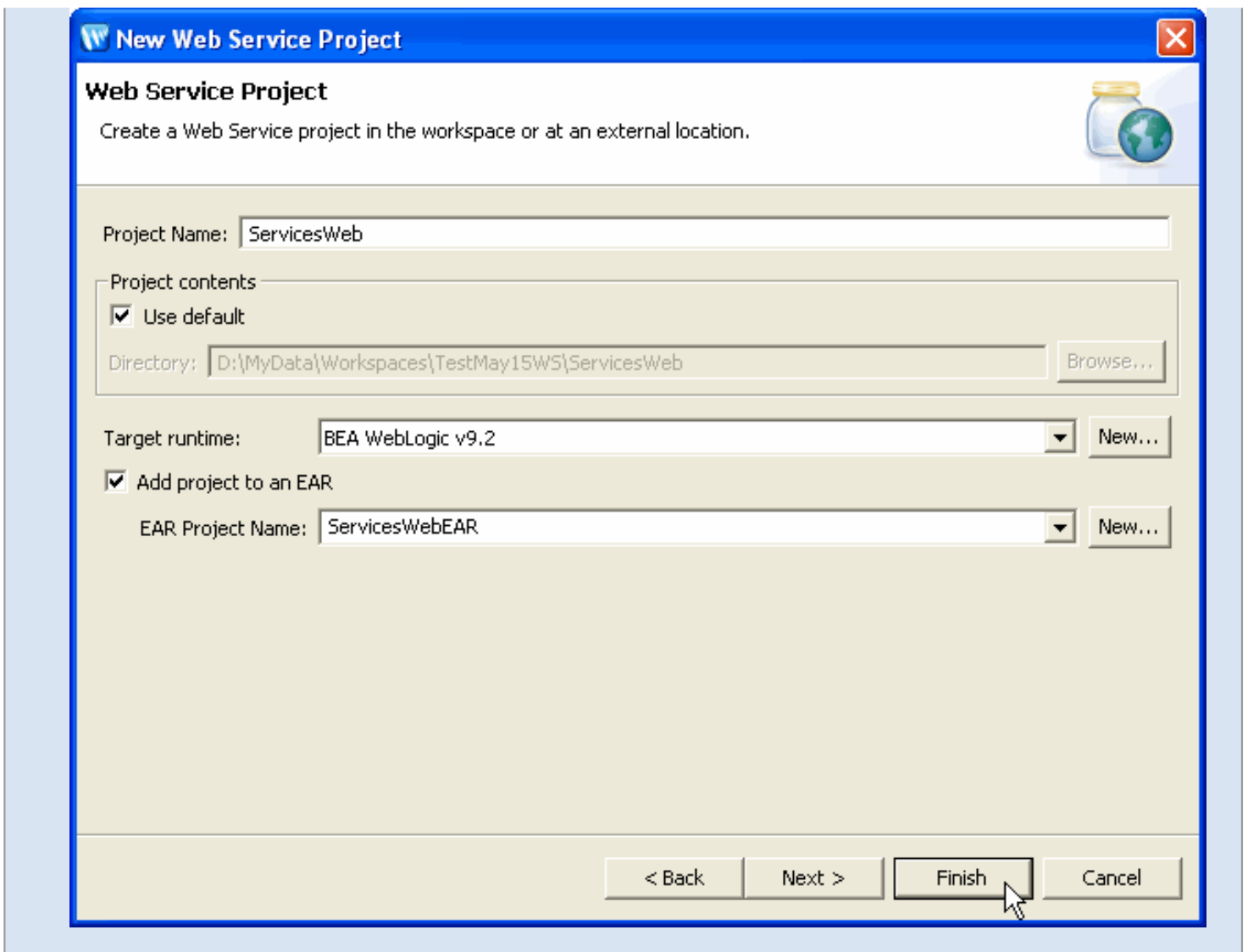
### Step 1: Create a New WebLogic Web Service Project

In this section, you will create the project that will contain your web service. You will also create a related Enterprise Application (EAR) project. The EAR project contains various resources required to run the web service.

1. Click **File > New > Project**.
2. The **New Project - Select a wizard** dialog box appears. Expand **Web Services** and select **Web Service Project**.
3. Click **Next**.



4. The **New Web Project** dialog box appears. Enter **ServicesWeb** in the **Project name** box.
5. Click **Add project to an EAR**. Click **Finish**.



The **Package Explorer** pane in the IDE now displays the two projects you just created - *ServicesWebEAR*, the EAR project, and *ServicesWeb*, the web service project.

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



## Step 2: Add a Web Service to the Project

In this section, you will add a simple web service to the project you created in Step 1 by first creating a Java package and then inserting the web service into the package.

1. In the **Package Explorer**, expand **ServicesWeb** and right-click the **src** folder.
2. Click **New > Package**.
3. The **New Java Package** dialog box appears.

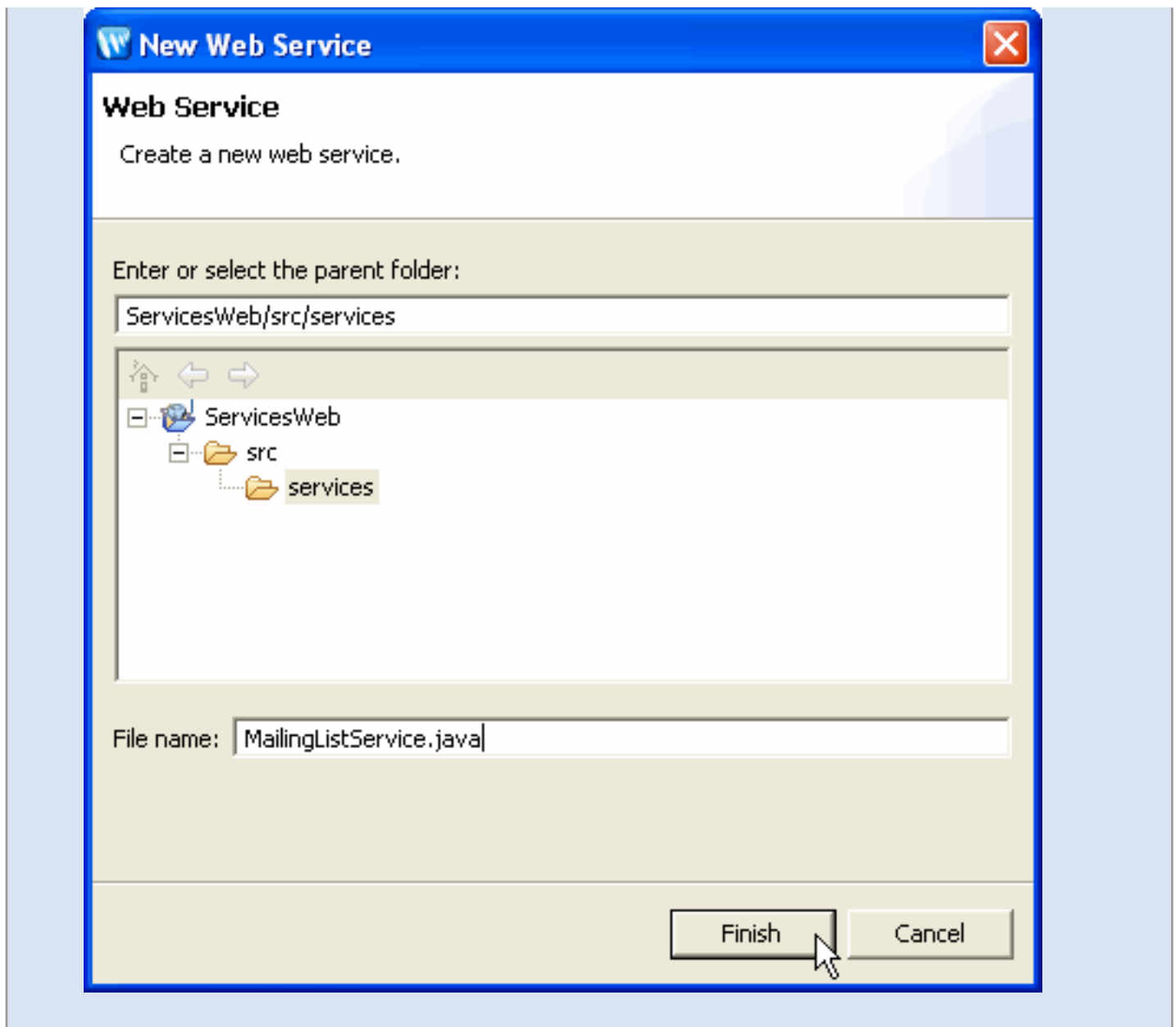
The **Source Folder** text box should be prepopulated with the string **ServicesWeb/src**. If it is not, enter that string.

Enter `services` in the **Name** text box and click **Finish**.

Notice that a package named **services** is now displayed under the **src** directory (that is, **ServicesWeb/src**) in the **ServicesWeb** project in the **Package Explorer** view. Physically, **services** is a directory. In the following steps, you will create a web service within that **services** package.

4. Right-click the **services** package.
5. Click **New > WebLogic Web Service**.
6. The **New Web Service** dialog box appears.

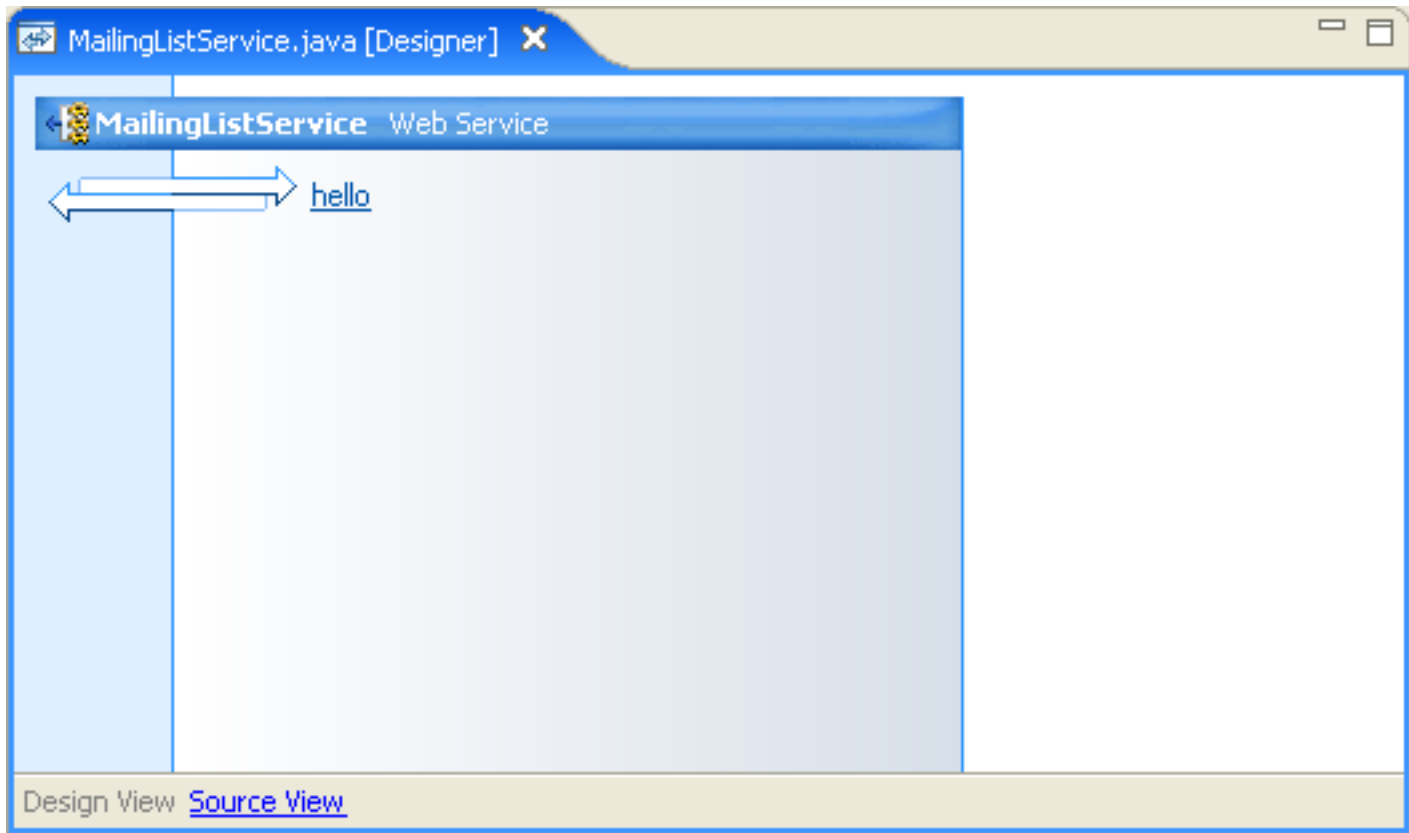
Enter `MailingListService.java` in the **File name** text box and click **Finish**.



The preceding steps created the new Java file `MailingListService.java` in the `services` folder.

You should now see `MailingListService.java` in **Design View**.





Design View gives a graphical representation of your web service, its methods, and any controls it contains. The web service `MailingListService.java` has one method, named **hello**, and no controls. The `hello` method is created by default with each new web service.

To see the underlying source code for the web service, click the link text **Source View** at the bottom of Design View. The source code for the web service appears as follows:

```
package services;

import javax.jws.*;

@WebService
public class MailingListServices {

    @WebMethod
    public void hello() {
    }

}
```

Note the use Java5 annotations in the source code, for example, the `@WebService` annotation, which specifies that the class implements a web service. Java5 annotations are used to set properties on the web service class and its methods.

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

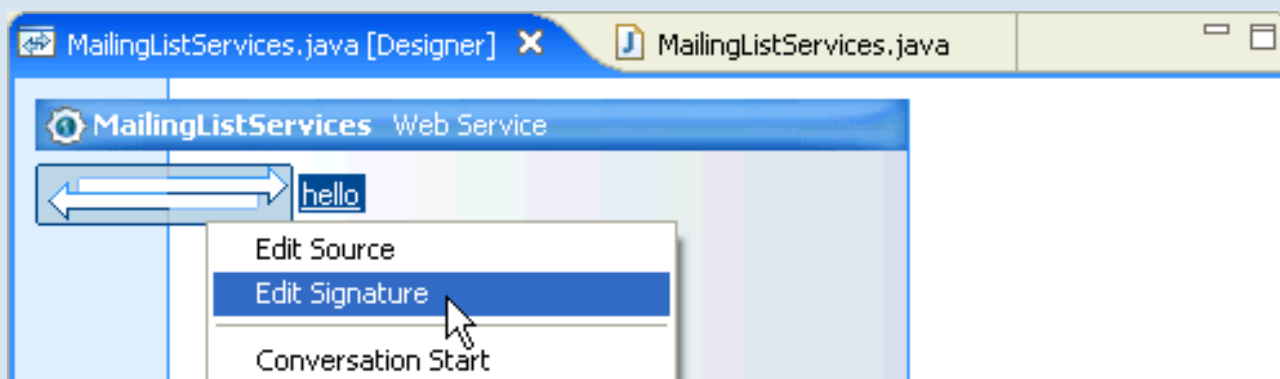


## Step 3: Add a Web Method to the Web Service

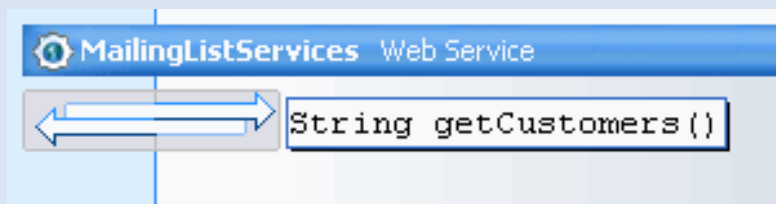
In this section, you will create a simple web method (a method that can be invoked over the web) in the web service. This operation is designed to return customer data. In a "real world" application, this method would probably perform some database lookups, but in this simple example, we will simply return the name "John Smith" to all customer enquiries.

Before you start, be sure that Workshop for WebLogic has `MailingListService.java` open for editing in the **Design View**. To ensure that the file is open for editing, double-click on `MailingListService.java` in the **Package Explorer** view.

1. In **Design View**, right-click the **hello** method icon (either the arrows or the link text will work) and select **Edit Signature**.



2. In the editing area that appears, change the text from **void hello()** to **String getCustomers()** and press **Enter**.



At this point, the method will be marked with red-underlining, indicating a compile error. In the next step, you will correct that error.

3. In **Design View**, right-click the **hello** method icon and select **Edit Source**.



4. In the method body enter the following return statement:

```
return "John Smith";
```

The final method should appear as follows:

```
@WebMethod
public String getCustomers() {
    return "John Smith";
}
```

5. Save the file with the **File > Save** command or by pressing **Ctrl+S**.

In **Source View**, the class should now look like this:

```
package services;

import javax.jws.*;

@WebService
public class MailingListService {

    @WebMethod()
    public String getCustomers() {
        return "John Smith";
    }
}
```

In **Design View**, the class should look like this:



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



## Step 4: Test the Web Service

In this section, you will define and start a server and then use the server's built-in test functionality to test the web service you created in the preceding sections.

Workshop for WebLogic creates a WebLogic Server as part of installation and we will use this local server for testing the application we will create in this tutorial. You must set up the server before you can test any applications.

### To Set Up a Test Server

If you are working through this tutorial for the first time, you must create a server definition for use within the IDE and add the ServicesWebEAR to that server. To set up a server definition, follow the instructions at [Setting up Servers for Use Within the IDE](#). Make sure to [add the ServicesWebEAR project](#) to the server once the server definition has been created.

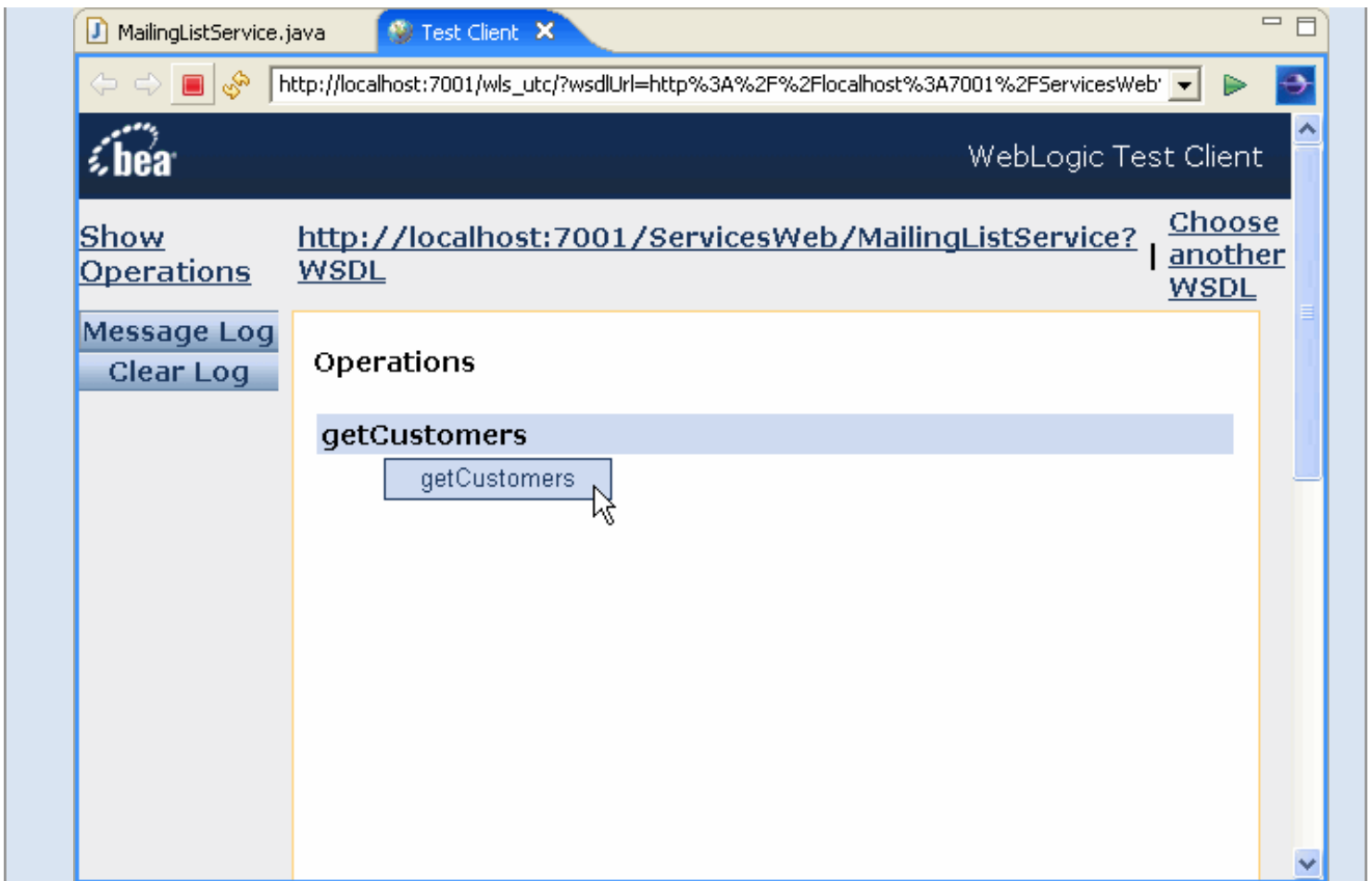
If you are working through this tutorial for a second time, you must [remove previous, duplicate projects \(modules\) from the server](#)

### To Test a Web Service

Unlike a page flow or application, a web service does not do anything unless a request is received from a client. For testing your web service, Workshop for WebLogic provides a **test client** that allows you to send messages to the service and review the response message. The test client runs in the workbench, as an editor window.

To test the web service:

1. In the **Package Explorer** view, if the **services** package is not expanded, expand it now.
2. Right-click `MailingListService.java` and select **Run As > Run On Server**.
3. We recommend that you check the box marked **Set server as project default**. This will reduce the number of steps in the next part of this tutorial, as the IDE will remember your server choice. Click **Finish**.
4. Wait for the server to startup and the application to deploy.  
You will see the test client display:



This web service has only one operation (**getCustomers**). If there were input values required by **getCustomers**, there would be input fields that would allow you to specify values. Clicking on the **getCustomers** button sends a request message to the web service.

5. Click the **getCustomers** button now to invoke the **getCustomers** method.

The test client displays the results of invoking the web service operation (including the returned value) and also the detail of the SOAP-encoded request that was sent to the web service and the response that was received, including the string returned by the **getCustomers** operation/method: "John Smith".

6. You can return to testing your web service operation by clicking on **Show Operations** at the top left corner. You may also examine the automatically generated WSDL (Web Service Description Language) file by clicking on the link to the right of the **Show Operations** link.

Click the arrow to navigate through the tutorial:

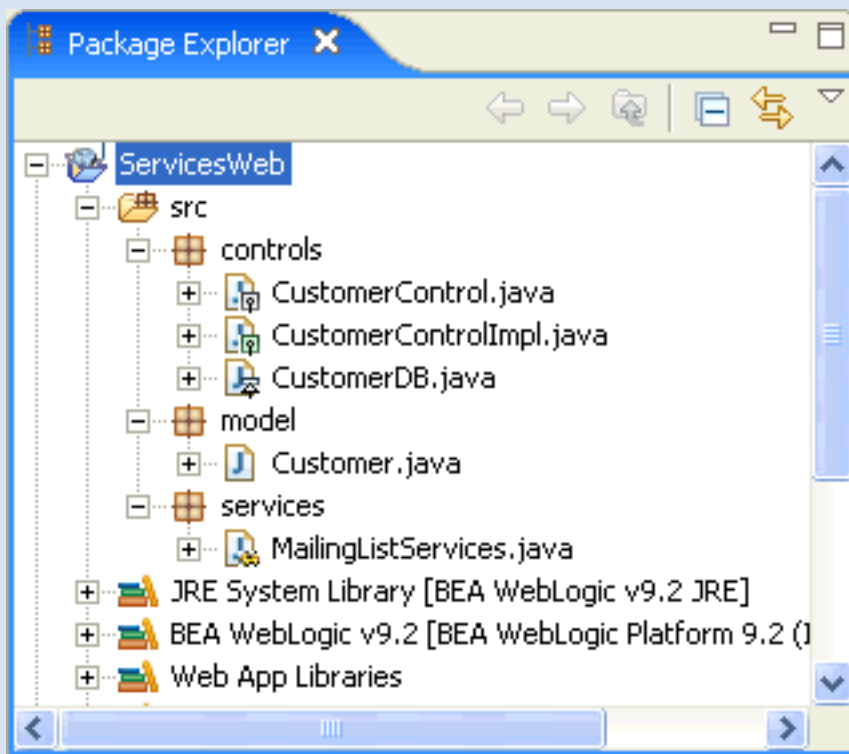


## Part II : Web Service That Calls Methods on Provided Controls

### Step 5: Import Controls into Your Web Services Project

In this section, you will import into your web services project a group of complex control classes that have been created ahead of time.

1. Open Windows Explorer (or your operating system's equivalent) and navigate to the directory **BEA\_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples\_9.2.1/tutorials/resources/webService/**
2. Drag the folders **controls** and **model** (located at **BEA\_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples\_9.2.1/tutorials/resources/webService/**) into the Package Explorer pane directly onto the folder **ServicesWeb/src**.
3. The **src** folder under **ServicesWeb** should now have the **controls** and **model** packages underneath it. If you expand those two packages, you should see a directory tree like this:



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



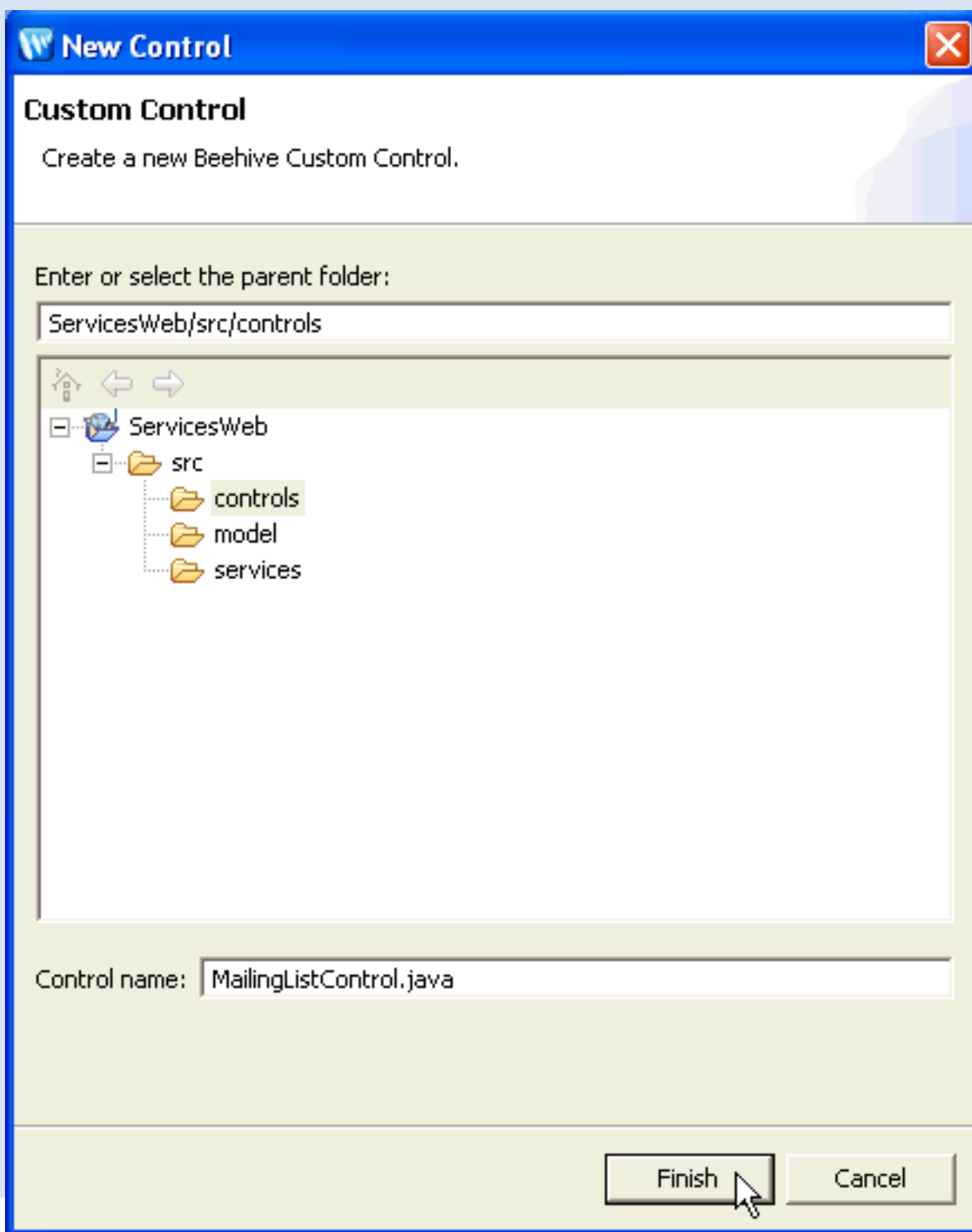


## Step 6: Create a Custom Control

In this step, you will create a new custom control. You will also insert a method into the control that calls a method on one of the controls you imported earlier.

1. In the **Package Explorer** view, right-click the **ServicesWeb/src/controls** folder.
2. Click **New > Custom Control**.
3. The **New Control** dialog box appears.

Enter `MailingListControl.java` in the **Control name** text box and click **Finish**.





4. The preceding steps instructed Workshop to create two new Java files in the controls folder:
  - `MailingListControl.java` - the control interface file
  - `MailingListControlImpl.java` - the control implementation file that implements `MailingListControl.java`.

These files contain only a default framework at this point. You will add a method in the following steps.

5. in the **Package Explorer** view, double-click on **MailingListControlImpl.java**.
6. In the source editor, right-click anywhere within the source code for `MailingListControlImpl.java` and click **Insert > Control**.
7. In the **Select Control** dialog, select **CustomerControl - controls** and click **OK**.
8. Now you are ready to add the new method.

After the variable declaration for `customerControl`, add the method:

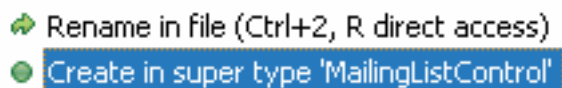
```
public Customer[] getLocalCustomers()
{
    return customerControl.getCustomersByState("CA");
}
```

9. You will see an error because of the undefined type `Customer`. Add this line of code to the imports section of the class:

```
import model.Customer;
```

10. Although you have created a new method in this class, the corresponding abstract method definition does not yet exist in `MailingListControl.java`, the interface class that this class implements.

To correct this situation place the editor's cursor anywhere in the name of the method (`getLocalCustomers`) and press **Ctrl+1**. Select **Create in super type 'MailingListControl'** and press **Enter**.



`MailingListControl.java` opens in the editor, with the new abstract method definition in place.

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

The `getLocalCustomers` method on this control uses the imported controls to query a sample database for all customers in a given state. In this example, we have hard-coded the state to be California. The data returned from the database is returned to the calling method as an array of `Customer` objects.

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



## Step 7: Use the Control from the Web Service

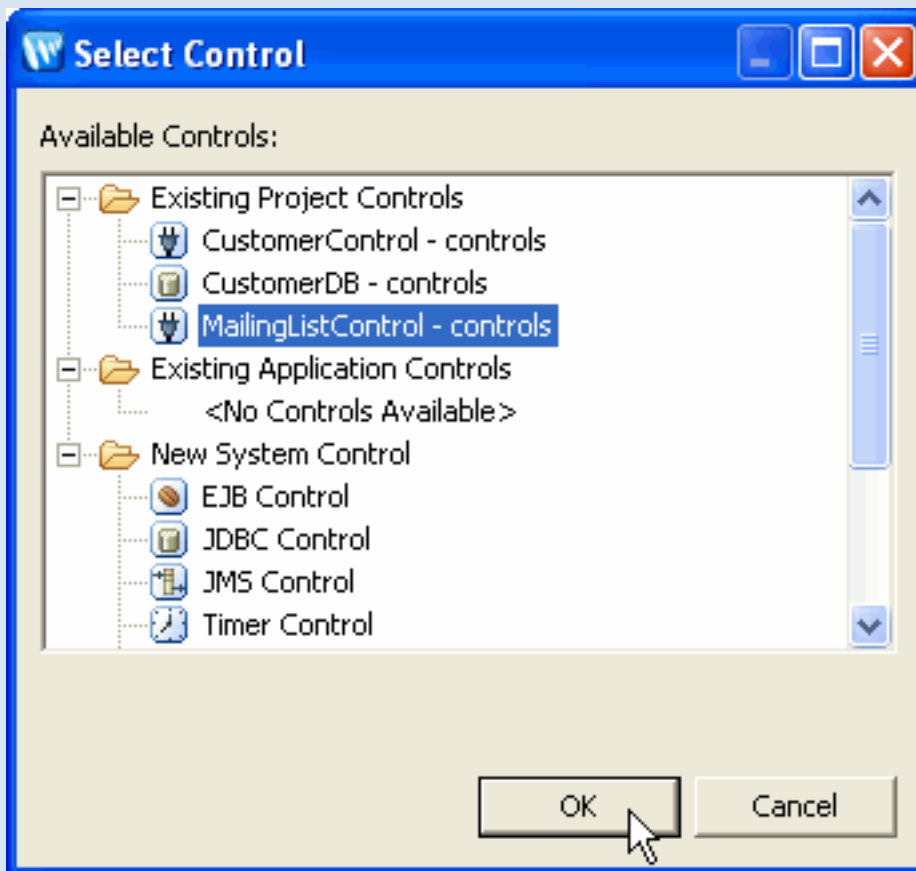
In this section, you will insert a method in the web service to call a method on the custom control.

### Insert a Control

1. In the **Package Explorer**, double-click the web service file **MailingListService.java**.
2. Right-click in the **Design View** editor and select **New Control Reference**.
3. The **Select Control** dialog box appears.

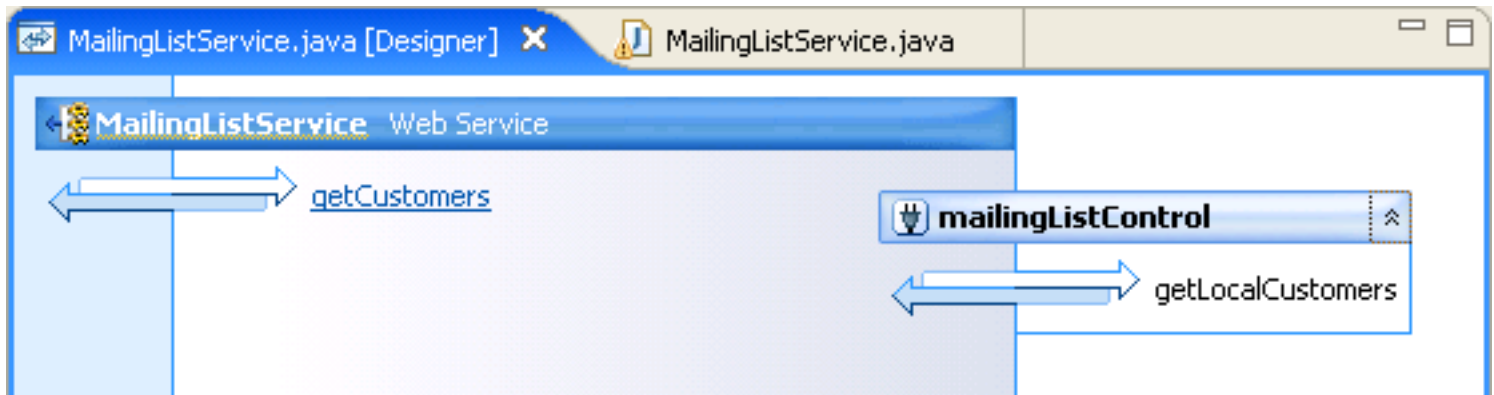
Note that this dialog lets you choose from various existing controls, including **MailingListControl**, the one you created earlier in the **controls** package.

Select **MailingListControl - controls**, and click **OK**.



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

In **Design View**, the web service should look like this:



In **Source View**, the web service should now look like this:

```
package services;

import javax.jws.*;
import org.apache.beehive.controls.api.bean.Control;
import controls.MailingListControl;

@WebService
public class MailingListService {

    @Control
    private MailingListControl mailingListControl;

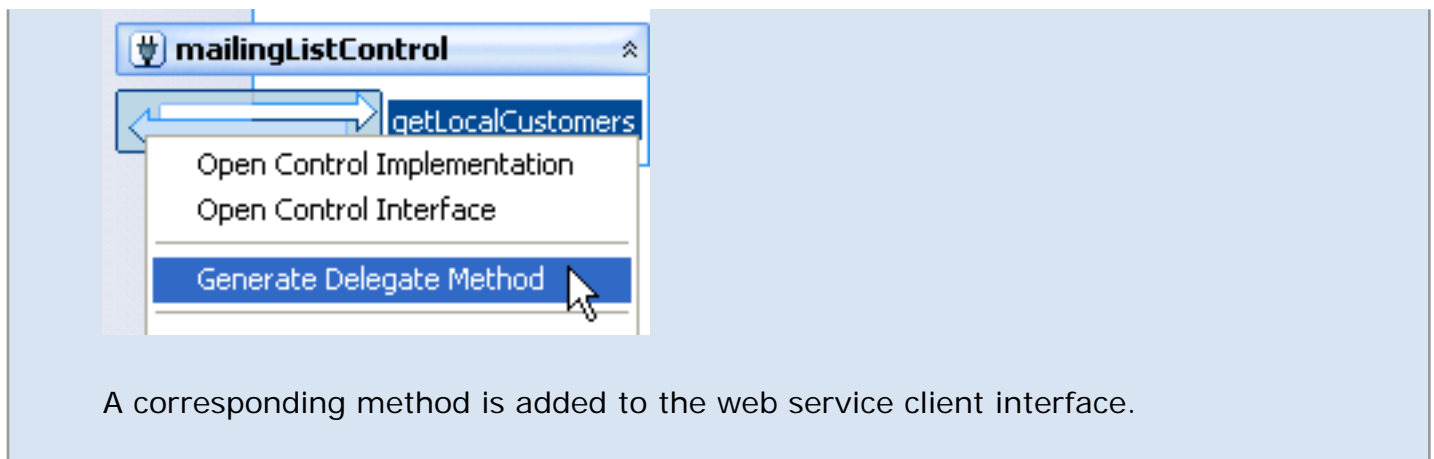
    @WebMethod
    public String getCustomers() {
        return "John Smith";
    }
}
```

Note that Workshop for WebLogic added the required imports for **MailingListControl**, the control you told it to insert. It also added a variable declaration for a control of type **MailingListControl** named **mailingListControl**. Workshop for Weblogic declared **mailingListControl** to be a control by adding the **@Control** annotation.

## Call a Method on the Control

You will now add a method to the service that will call a method on `mailingListControl`, the instance of `MailingListControl` you just created.

1. In **Design View**, right-click the control method **getLocalList** and select **Generate Delegate Method**.



The web service class should now look like this:

```

package services;

import javax.jws.*;
import org.apache.beehive.controls.api.bean.Control;
import controls.MailingListControl;

import model.Customer;

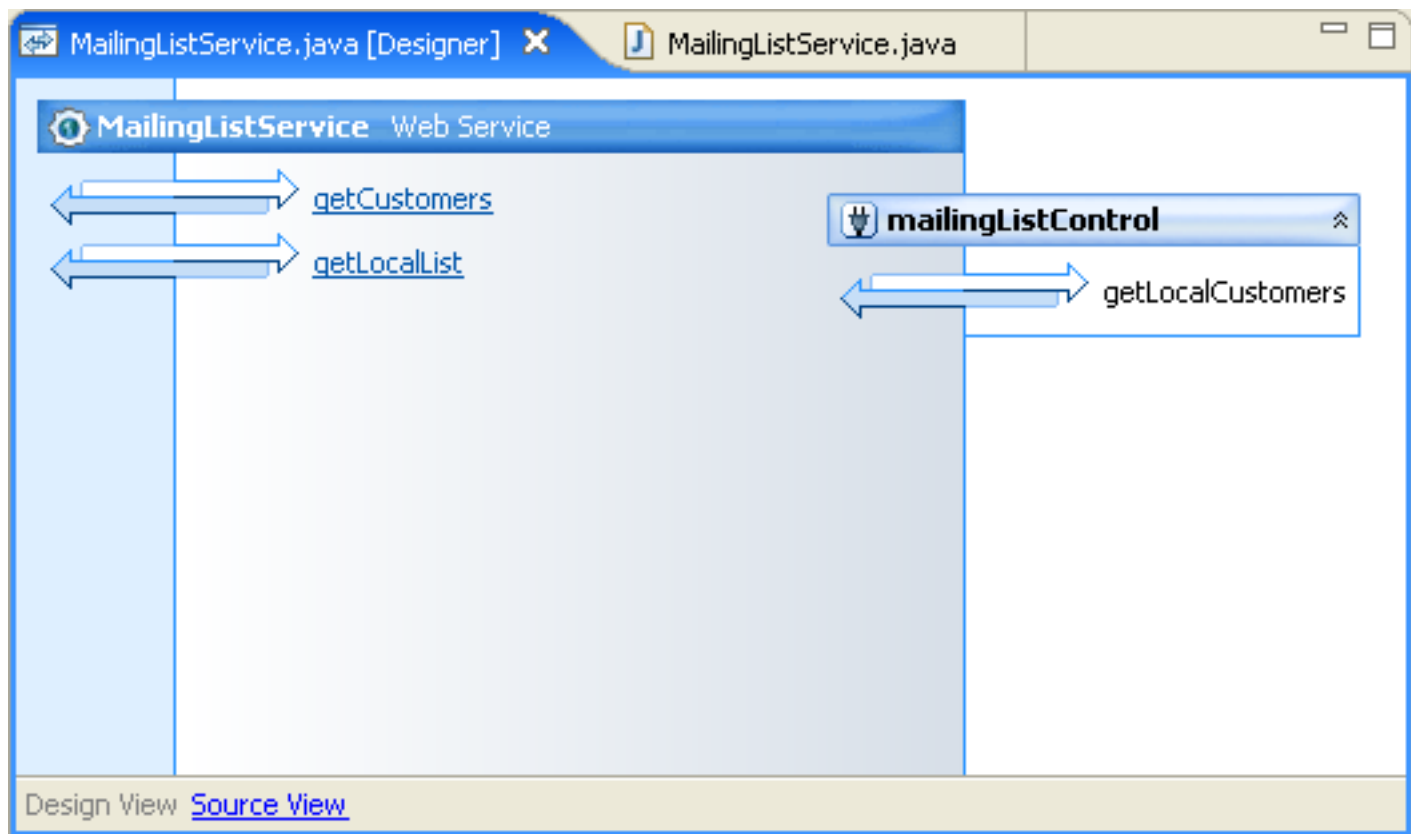
@WebService
public class MailingListService {
    @Control
    private MailingListControl mailingListControl;

    @WebMethod()
    public String getCustomers() {
        return "John Smith";
    }

    @WebMethod()
    public Customer[] getLocalList() {
        return mailingListControl.getLocalCustomers();
    }
}

```

In Design View the web service looks like this:



The new method calls the control method `getLocalCustomers`, which will return an array of Customer objects for all customers in California in the sample database.

In the next step, we will test the new method.

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



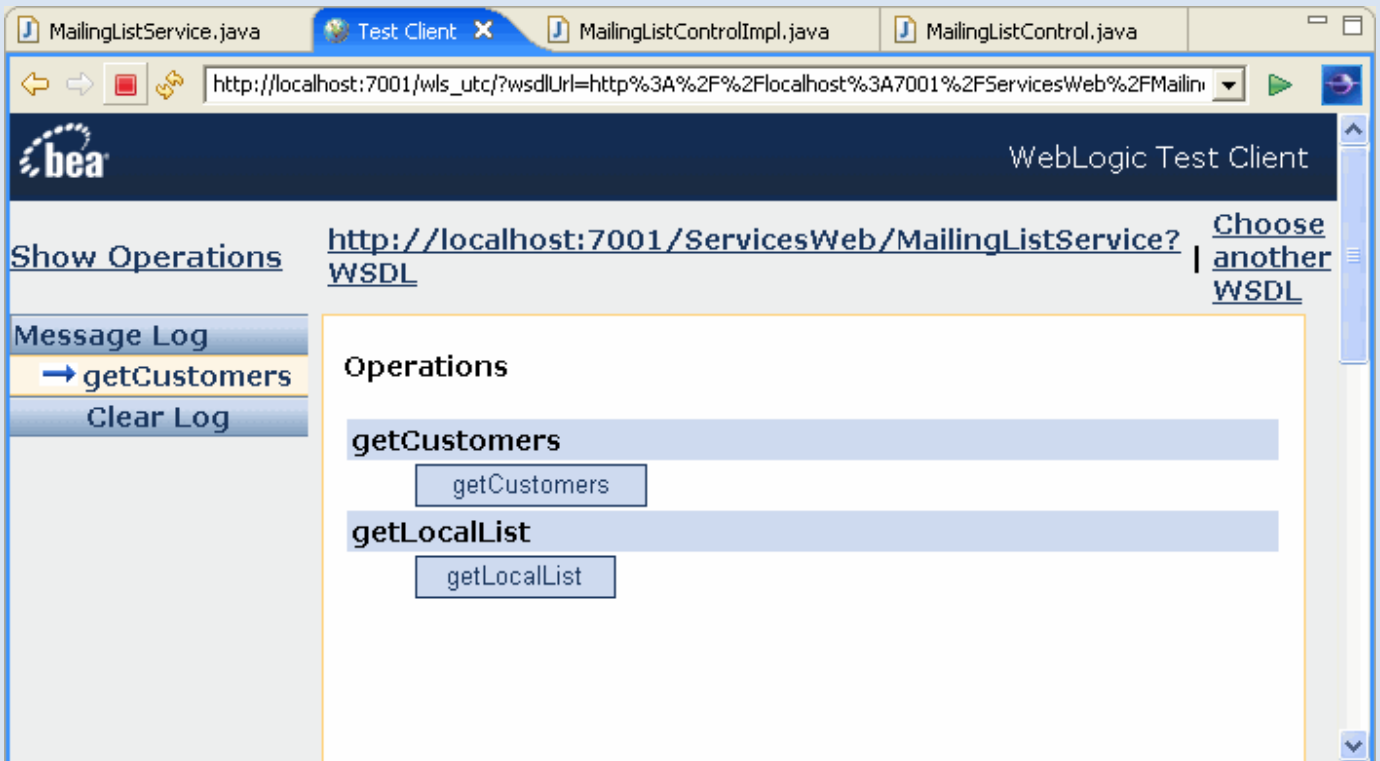
## Step 8: Test the Web Service

In this section, you will start a server (or use one you already have running) and then use the server's built-in test functionality to test the method you added to the web service you created in the preceding section.

### Test the Web Service Methods

Now you are ready to use the test client built into WebLogic Server to test the web service.

1. In the **Package Explorer** view, right-click **MailingListService.java** and select **Run As > Run On Server**.
2. Note: This step applies only if you did not check the **Set server as project default** box when you defined the server. The **Run on Server - Define a New Server** dialog box will appear. Click **Finish**.
3. The test client will be displayed, this time showing two operations:



4. Note that this form includes test buttons for each of the two methods you created in `MailingListService.java`.  
If you click the first one, **getCustomers**, you will see the string "John Smith" that the method returns.  
If you click the second button, **getLocalList**, you will see a SOAP-encoded message returned by the control and containing all the customers the sample database has for the state of California.

Click the arrow to navigate through the tutorial:





## Tutorial: Advanced Web Services

This tutorial demonstrates how to create a web service and insert code to access an existing control. The tutorial then demonstrates how to generate a new web service control from a WSDL and how to access that control from another control.

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

The steps in the tutorial are:

1. Create a LoanApplication web service that calls the existing loan approval control.
2. Create a new credit scoring web service control that accesses the CreditScore web service; then modify the loan approval control to use the new credit scoring control.

### Control Basics

A *control* is simply a Java Bean that provides standardized access to a resource or to encapsulated business logic. Controls use Java 5 metadata annotations for more convenient configuration. Workshop for WebLogic automatically generates code and annotations to create controls and access them. Workshop for WebLogic implements many types of controls. The easiest way to access a web service is to create a web service control, as this tutorial will demonstrate.

A control is implemented as an annotated class definition that looks something like this: (note that the annotation line precedes the class declaration line)

In the interface file:

```
@ControlInterface
public interface LoanApprovalControl
{
    // method declarations
}
```

In the implementation file:

```
@ControlImplementation
public class LoanApprovalControlImpl implements LoanApprovalControl
{
    // object body (methods, etc.)
}
```

To access a control, Workshop for WebLogic declares the control like this:

```
@Control  
private LoanApprovalControl loanApprovalControl;
```

The declaration causes an object to be instantiated before your code runs and the control's methods can then be called like regular object methods:

```
abc = loanApprovalControl.getLoanApproval(ssn, amount);
```

For more detailed information on controls, consult the Beehive documentation at [Working with Beehive Controls](#). For more information on Java 5 annotation consult <http://sun.com>.

## Related Topics

For a discussion of how to build a custom control that accesses other controls and then access that custom control through a web service refer to [Tutorial: Web Service](#).

Click the arrow to navigate through the tutorial:



## Advanced Web Services Tutorial: Step 1: Import the Tutorial Workspace

In this step, you will import an existing set of projects that contains the initial components of your application.

The tasks in this step are:

- [Start Workshop for WebLogic and create the tutorial workspace](#)
- [Import projects into the workspace from an archive](#)
- [Review the existing projects and their contents](#)
- [Test the Web Service](#)

### To Start Workshop and Create the Tutorial Workspace

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

#### ... on Microsoft Windows

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

- From the **Start** menu, click **All Programs > BEA Products > Workshop for WebLogic Platform 9.2**

When prompted for the name of your workspace, click the **Browse** button and create a new (empty) workspace for this tutorial.

- When the Workshop for WebLogic window opens, check that you are in the **Workshop** perspective (indicated just below the toolbar at the top of the window. If you are not in **Workshop** perspective, set that perspective by clicking **Window > Open Perspective > Workshop**.

#### ...on Linux

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

- Run `BEA_HOME/workshop92/workshop4WP/workshop4WP.sh`

When prompted for the name of your workspace, click the **Browse** button and create a new (empty) workspace for this tutorial.

- When the Workshop for WebLogic window opens, check that you are in the **Workshop** perspective (indicated just below the toolbar at the top of the window. If you are not in **Workshop** perspective, set that perspective by clicking **Window > Open Perspective > Workshop**.

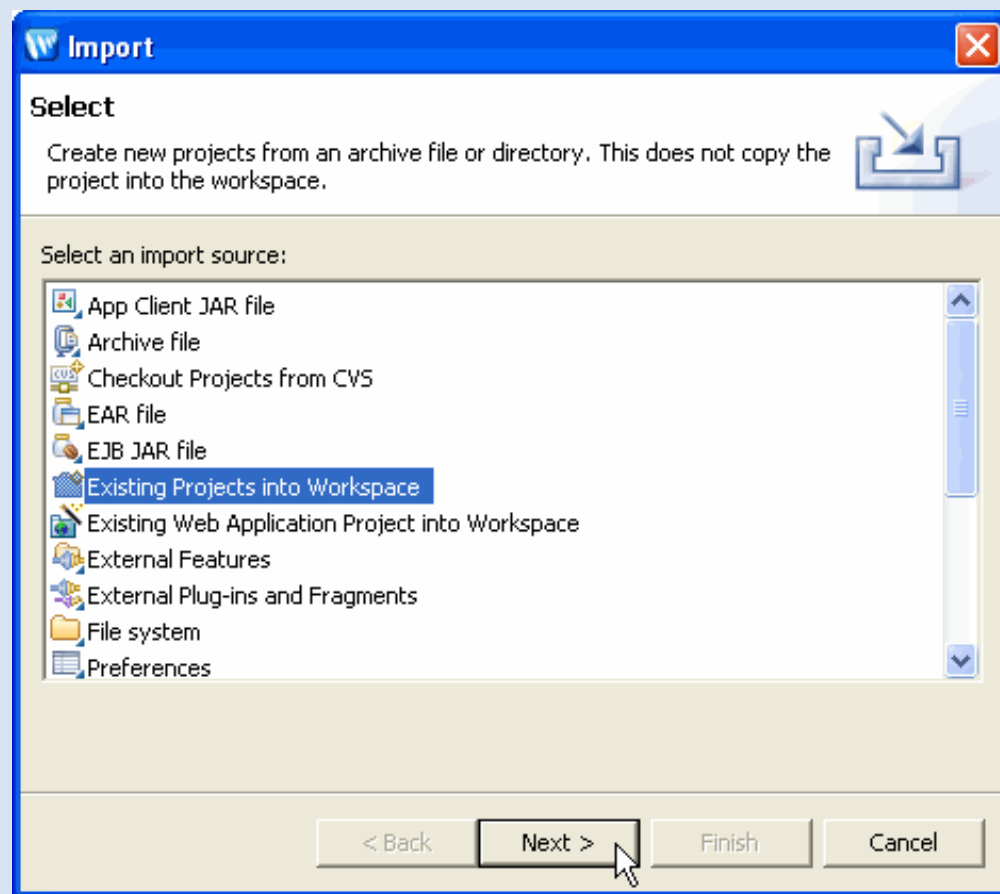
### To Import the Tutorial Projects into Your Workspace

Workshop for WebLogic keeps track internally of the project structure within a workspace. Simply copying folders into the workspace directory does not cause them to appear as projects inside Workshop for WebLogic. The tutorial projects are stored in a .ZIP archive file. There is no need to unzip the files, Workshop for WebLogic will import the .ZIP file directly.

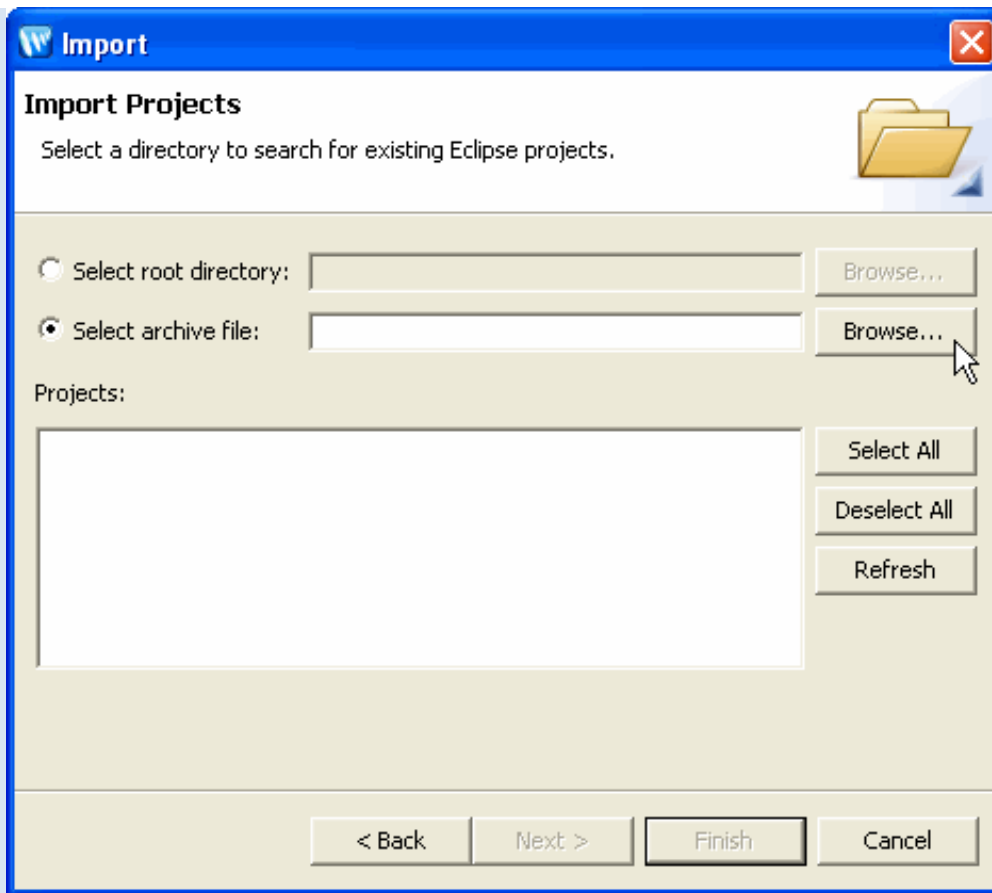
To import these projects and their files:

Click **File > Import**

Choose **Existing Projects into Workspace** from the dialog. Click **Next**.



Click the radio button for **Select archive file**.

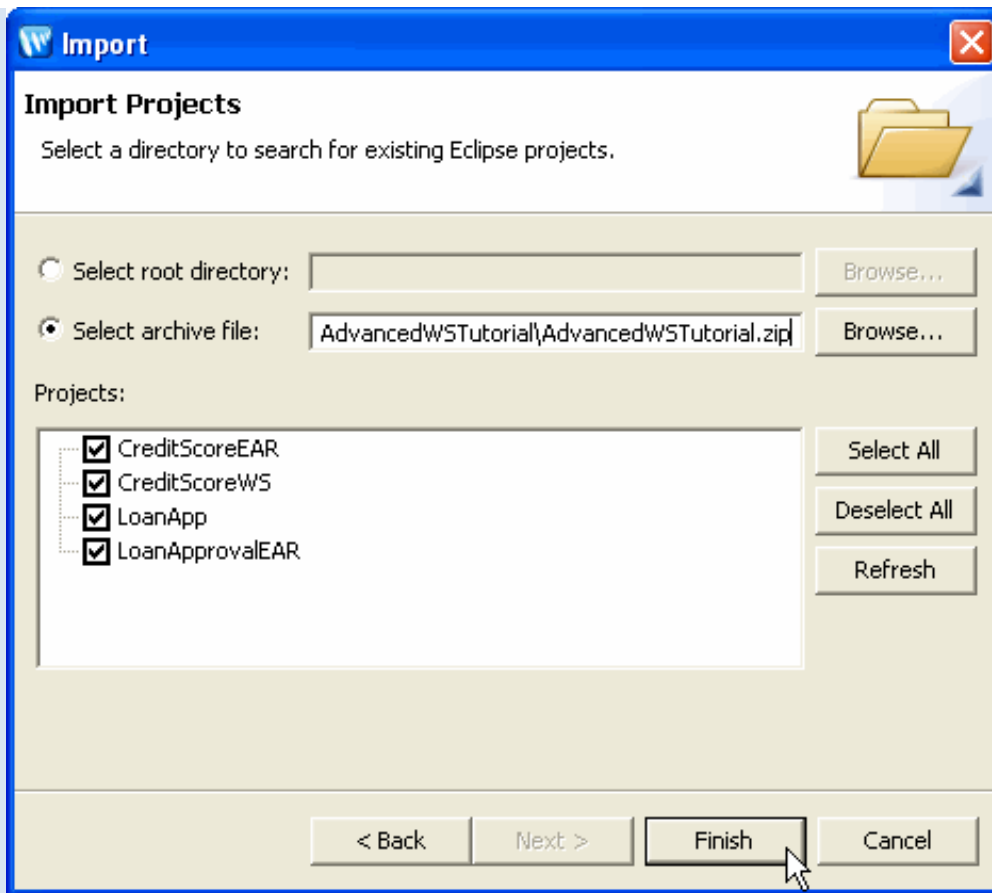


Click **Browse**.

Navigate to the location of the tutorial sample files, packaged as a ZIP file. This should be

BEA\_HOME/workshop92/workshop4WP/eclipse/plugins/com.bea.wlw.samples\_9.2.1/tutorials/resources/AdvancedWSTutorial/AdvancedWSTutorial.zip

Click **Open**.



Click **Finish** to continue. The import process will take few moments, because several projects and their contents must be imported.

## To Review the Contents of Your Workspace

The import process brought two applications into your workspace:

The **LoanApprovalEAR** application contains the project **LoanApp** which contains controls to provide loan approvals. Inside of the **LoanApp** project, there are two controls (inside `LoanApp/src/controls/`):

- **LoansDB.java** is a control that tracks loan requests. Since this is a demo application, the control creates a database the first time that it is called for easier setup. Request information is then stored in the database.
- **LoanApprovalControl.java** defines the method **boolean getLoanApproval(int ssn, float amount)** which returns true if the loan is approved. Note that **LoanApprovalControl** has two files: an interface file that defines the class methods and an implementation file for the actual code.

Inspect the code for these controls by double clicking on their names in the **src** folder of the **LoanApp** project. The source will appear in the editor

window.

Note the simple logic used for loan approval: if the person (identified by SSN) does not have a loan in the database, return true (approval) and store the SSN and loan amount. If the person already has a loan, return false (decline).

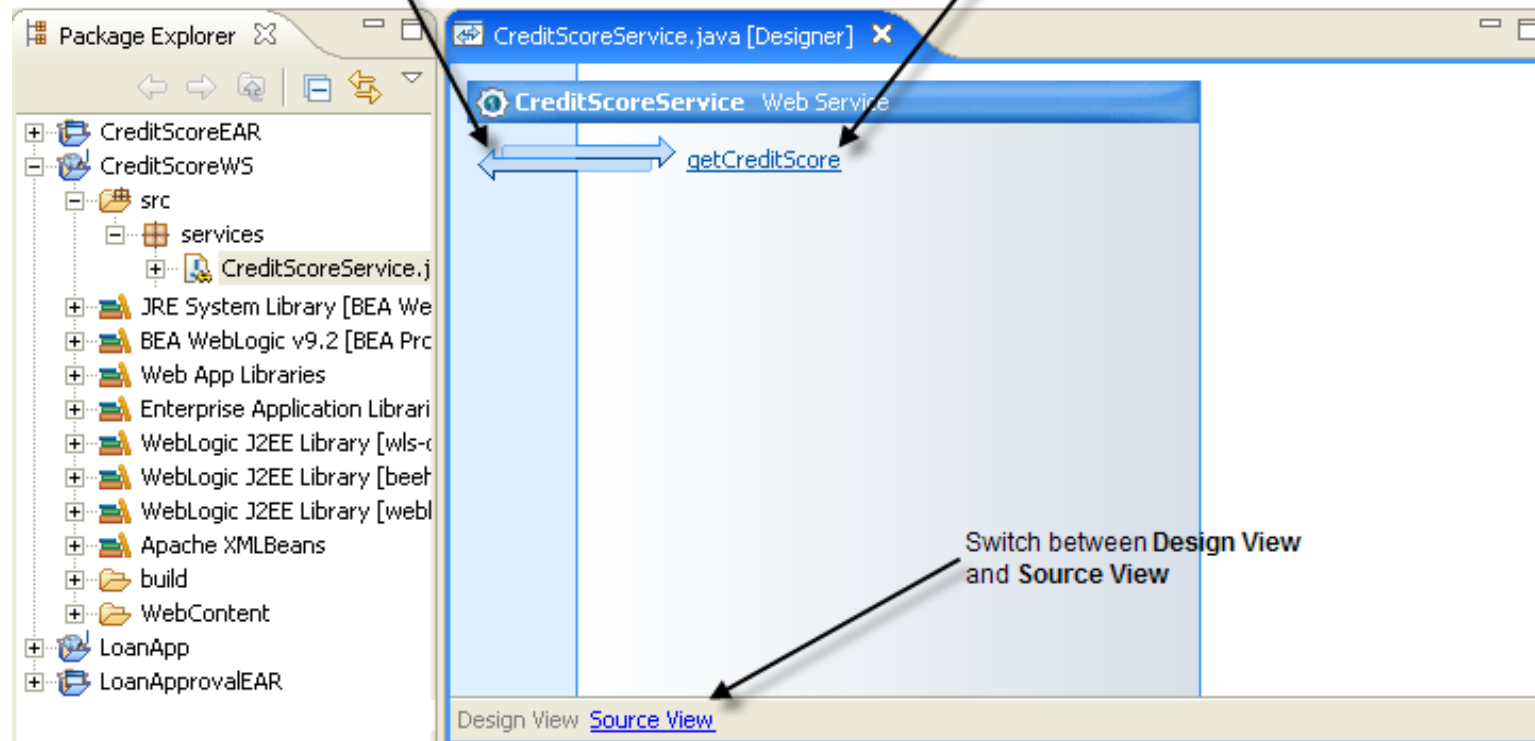
You cannot test the code for the control yet because there is no web service or page flow that instantiates and invokes the control.

The **CreditScoreEAR** application contains the single project **CreditScoreWS**. Open the web service at `CreditScoreWS/src/services/CreditScoreService.java` by double-clicking on the Java file.

When you open a web service in Workshop for WebLogic, it is displayed in **Design View** by default. Design View gives a graphical view of a web service. The Design View for the web service `CreditScoreService.java` shows that it has a single method called **getCreditScore**.

Web method icon. Two blue arrows indicates this method takes parameters and returns a value.

Click this link text to edit the source code for the method **getCreditScore**.



Click the link text **Source View** at the bottom of Design View to view the source code for the web service.

1. Expand the **src** folder and the **services** package. Then double click on the source file to open it in the editor.

2. Note that the file contains a web service (indicated by the **@WebService** annotation) with a single operation/method **getCreditScore** (indicated by the **@WebMethod** annotation).

```

package services;

import javax.jws.*;

@WebService
public class CreditScoreService
{
    static final long serialVersionUID = 1L;

    /**
     * Returns the credit score for a given SSN. Returns -1 if an ir
     */
    @WebMethod
    public int getCreditScore(int ssn)
    {
        if(ssn > 0 && ssn < 300000000)
            return 500;
        ...
    }
}

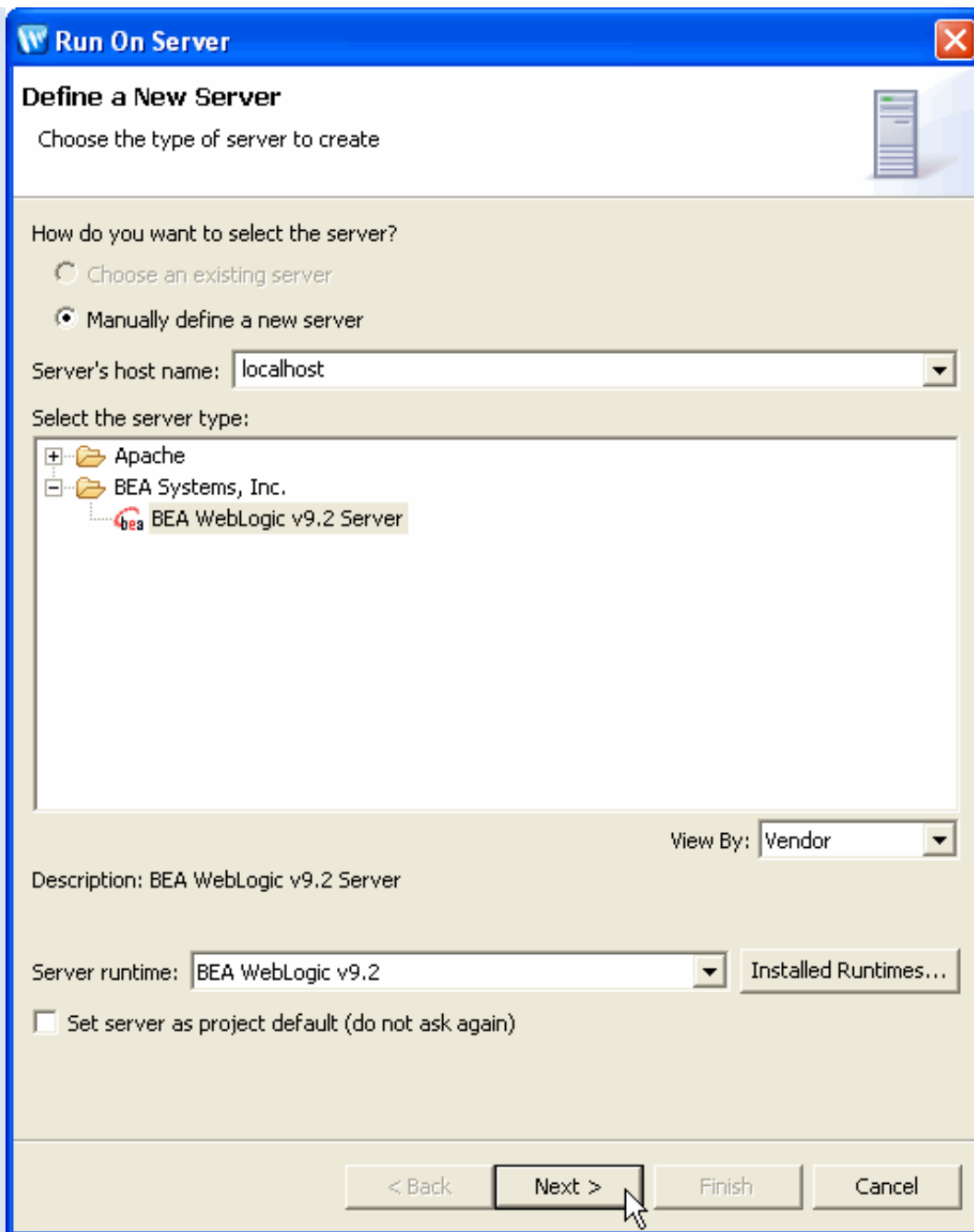
```

The **getCreditScore** method checks credit ratings for the individual, based on their Social Security Number (SSN), the most common identification number used in the United States. The SSN is a 9-digit number. The credit scoring system used in this example assigns a 3-digit valuation to individuals where higher values are better (e.g., 700 is a good credit rating and 500 is not as good).

## Test the Web Service

1. Note if this is your second time through the tutorial, you should remove previous versions of the **CreditScoreEAR** and **LoanApprovalEAR** projects from the server before deploying the current versions. For instructions on removing previous deployed projects see [Adding and Removing Projects from the Server](#).
2. Test the web service by right clicking on the file name **CreditScoreService.java** in the **Package Explorer** view and clicking **Run As > Run on Server**.

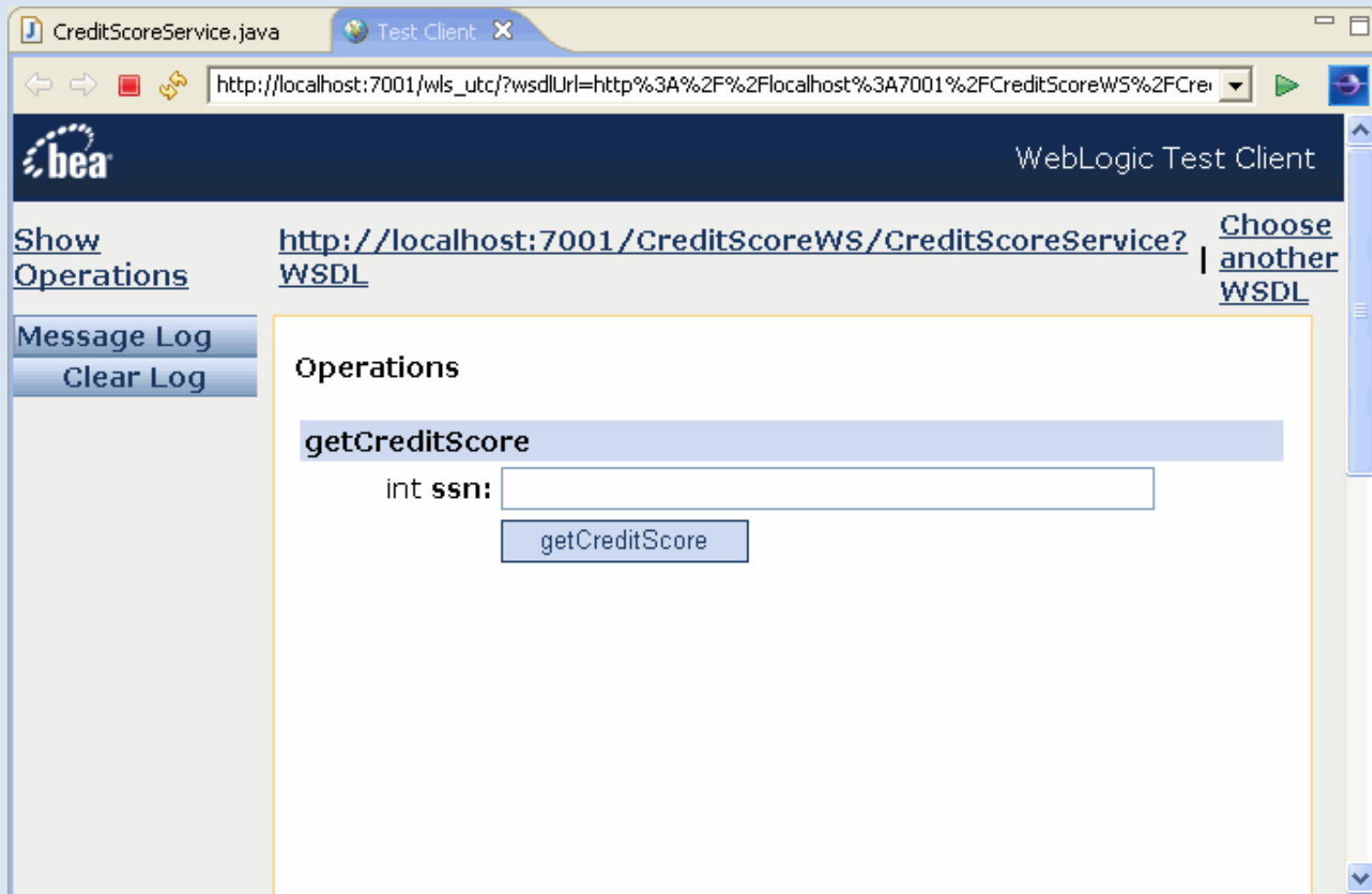




3. Click **Next** to proceed. From the next screen, use the pulldown to choose the default samples domain (BEA\_HOME/weblogic92/samples/domains/workshop). Click **Finish**.

Wait for the server to start and the application to deploy.

A window will open displaying the **Test Client**, a special application that allows you to interact with your web service.



The screenshot shows the WebLogic Test Client interface. The browser window displays the URL `http://localhost:7001/wls_utc/?wsdlUrl=http%3A%2F%2Flocalhost%3A7001%2FCreditScoreWS%2FCre`. The interface includes a 'Show Operations' button, a 'Message Log' section, and a 'Clear Log' button. The 'Operations' section displays the 'getCreditScore' service with an input field for 'int ssn:' and a 'getCreditScore' button.

4. You can enter values into the **ssn** parameter field and click on the **getCreditScore** button to send a value to the web service and get a response. For example, entering the value **123456789** returns a credit score of **500** as shown below.

The screenshot shows a WebLogic Test Client window with the following details:

- Browser Tab:** CreditScoreService.java
- Test Client Tab:** Test Client
- Address Bar:** [http://localhost:7001/wls\\_utc/callOperation.do;jsessionid=JQLKGyYXsC53zvhdz2b34fSqDg6NZv9x1GnTr](http://localhost:7001/wls_utc/callOperation.do;jsessionid=JQLKGyYXsC53zvhdz2b34fSqDg6NZv9x1GnTr)
- Page Header:** bea WebLogic Test Client
- URL:** <http://localhost:7001/CreditScoreWS/CreditScoreService?WSDL>
- Message Log:**
  - getCreditScore (selected)
  - Clear Log
- Request Summary:**
  - Arguments: int ssn: **123456789**
  - Returned: **500**
  - Submitted: Mon May 15 09:56:56 MDT 2006
  - Duration: 370 ms
- Request Detail:**
  - Service Request
  - <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

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



## Advanced Web Services Tutorial: Step 2: Create a New Web Service to Access the LoanApproval Control

You could test the loan application control through a page flow or a web service. In this step, you will create a web service that accesses the control.

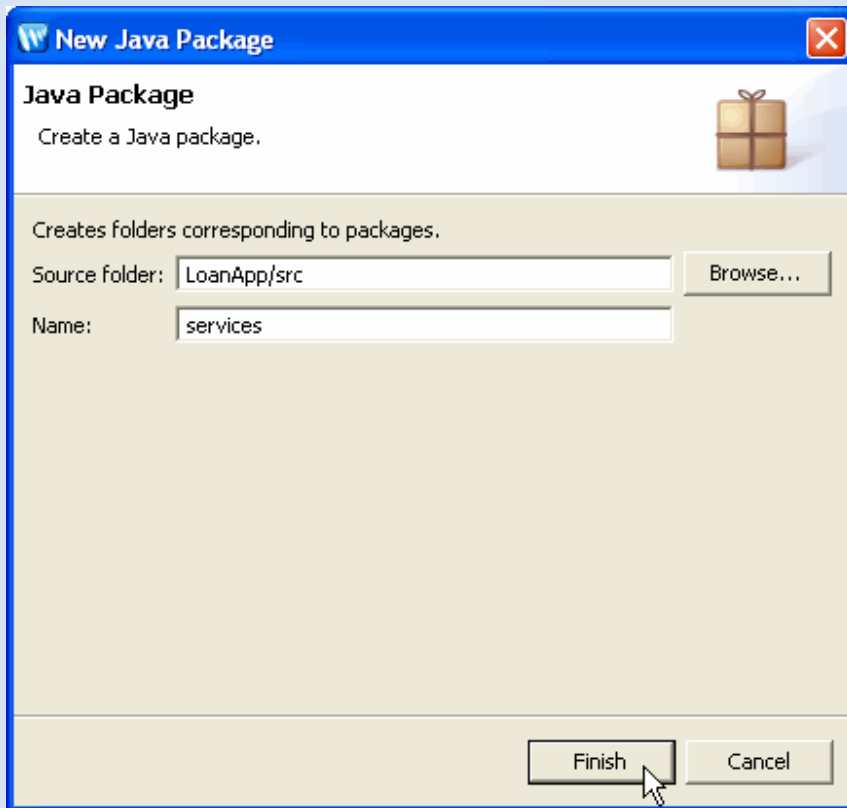
The tasks in this step are:

- [Create the new web service](#)
- [Access the control from the web service](#)
- [Test the web service \(and the control\)](#)

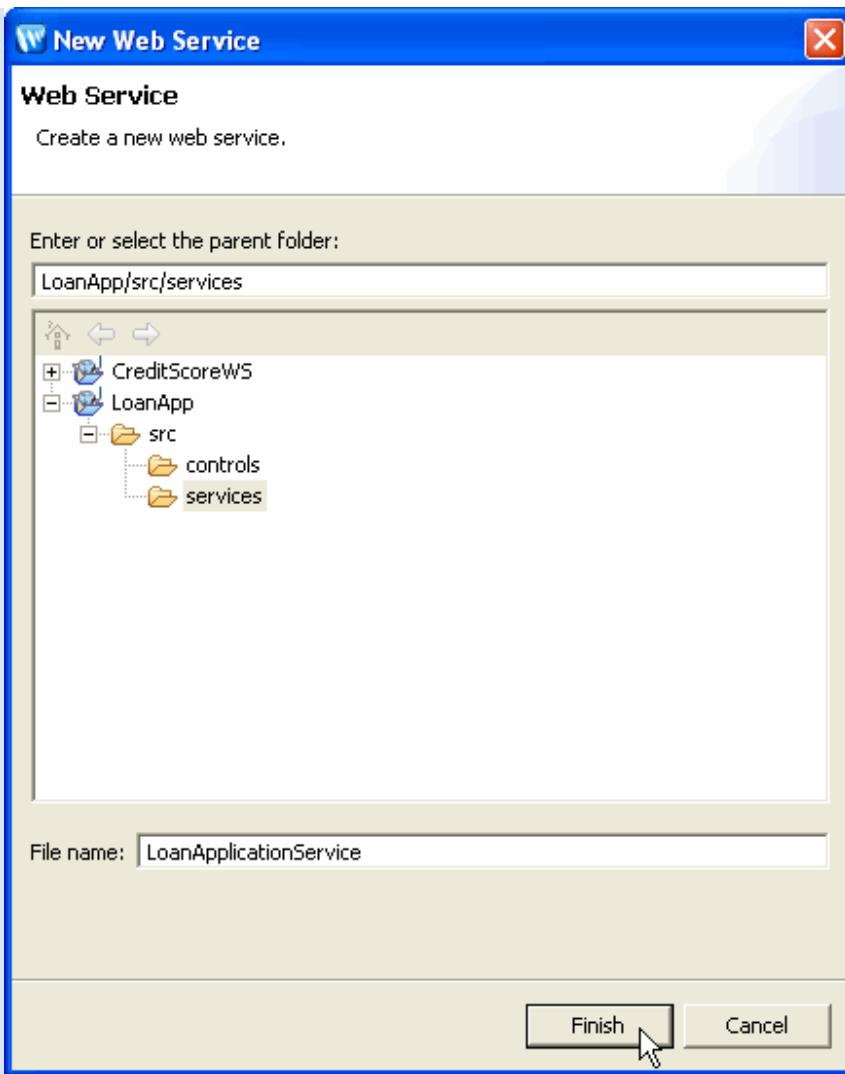
### To Create a New Web Service

To create a new web service:

1. Create a new package for the web service by right clicking on the **LoanApp** project's **src** folder and choosing **New > Package**. Set the package name to be `services` and click **Finish**.



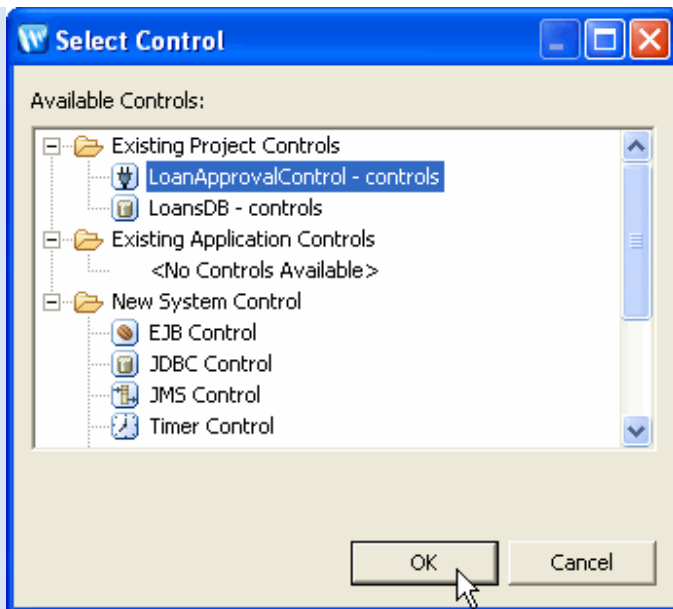
2. Create a new web service by right clicking on the new package and choosing **New > WebLogic Web Service**. Set the name of the web service to be `LoanApplicationService` and click **Finish**.



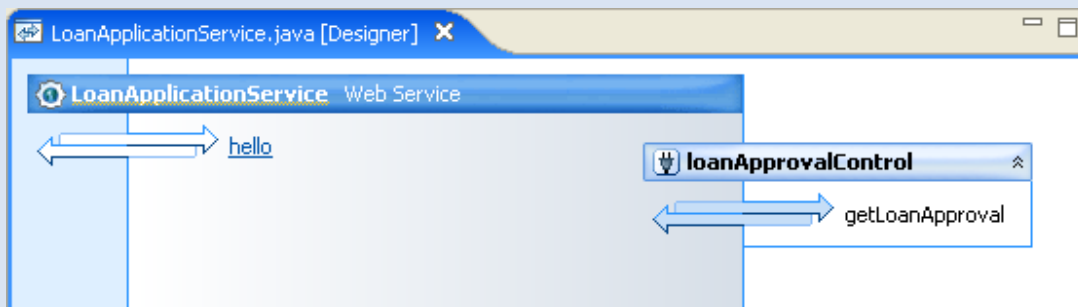
### To Access an Existing Control from a Web Service

After creating the web service, it is automatically displayed in **Design View**. To modify the web service to access the existing loan approval control, do the following steps:

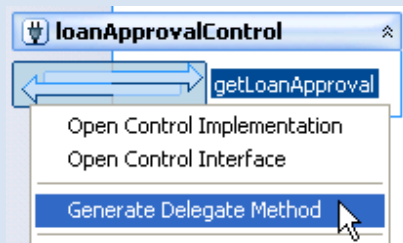
1. Insert the code to instantiate the current loan application control by right-clicking on the **Design View** editor and choosing **New Control Reference**.
2. On the **Select Control** dialog, select **LoanApprovalControl - controls** and click **OK**.



The new control is added to the right-hand side of Design View.



3. Right-click the control method **getLoanApproval** and select **Generate Delegate Method**.



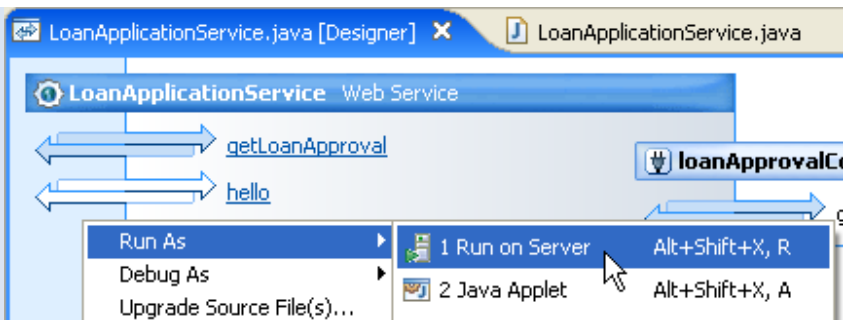
A corresponding web method is added to the web service client interface.

4. We now have a web service that
  - o instantiates a **LoanApproval** control object (with the `@Control` annotation line and the declaration following the `@Control` annotation line) and
  - o defines a single web method (through the `@WebMethod` annotation) that uses the **LoanApproval** control to determine loan approvals.
5. Save the new web service with **File > Save** or by pressing **Ctrl S**.

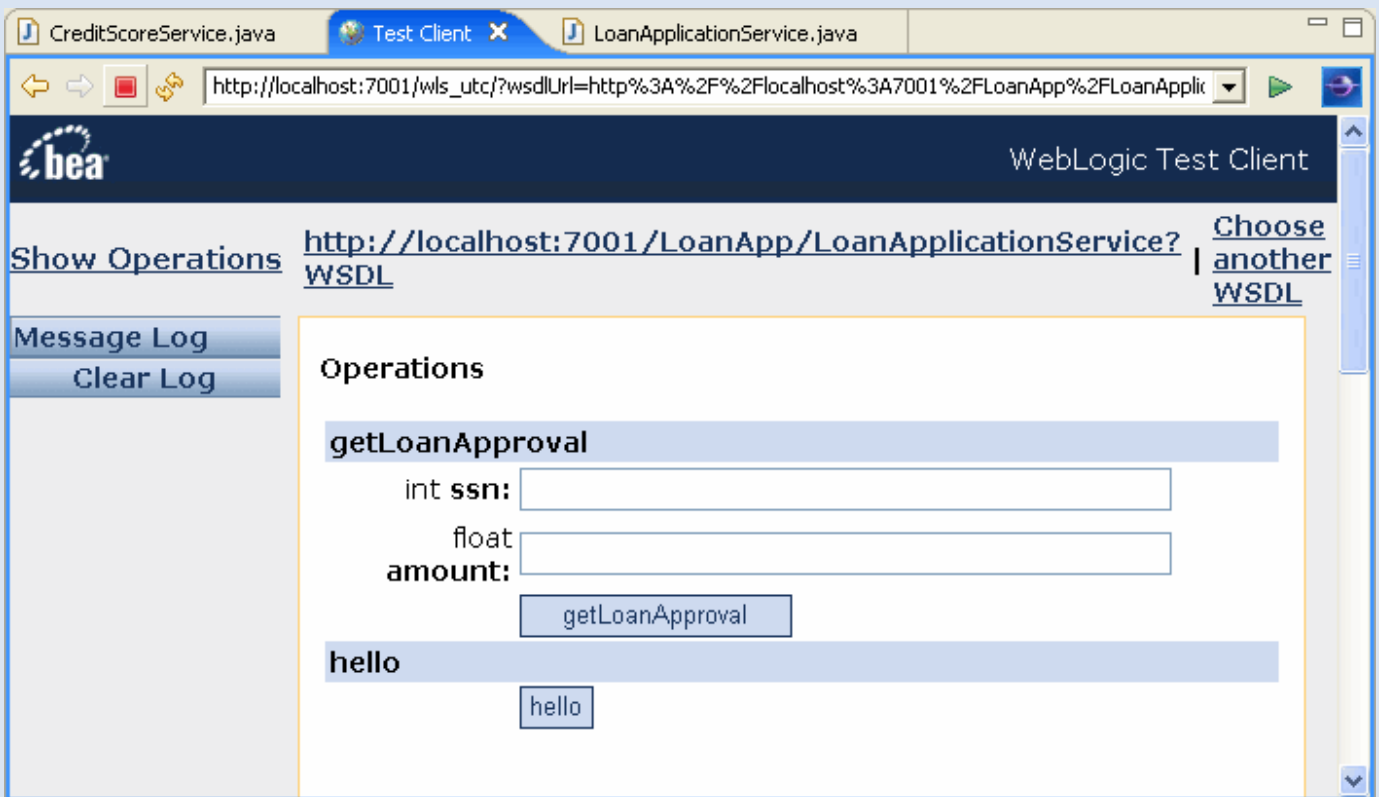
## To Test the Web Service

Now that the web service contains an operation, you can test it. To test the web service:

1. Switch to Design View, right-click anywhere within Design View and select **Run As > Run on Server**.



- In the **Run on Server** dialog click **Next**.
- On the field **Domain home**, click the dropdown and select the default sample domain **BEA\_HOME/weblogic92/samples/domains/workshop**. Click **Finish**.
- The test client window appears in the editors pane, showing the **getLoanApproval** web method.



Enter a 9-digit **ssn** and a loan **amount** and click **getLoanApproval**. When you have reviewed the result of running the operation, click **Show Operations** to return to the main test client page so that you can run another test.

- Enter the same **ssn** value and a loan amount and click **getLoanApproval** again. This time, the operation should return false since the person with this ssn already has a loan.
- Click **Show Operations** to return to the operations page.
- Note that the **Message Log** at the left now has two entries, one for each test. You can click on an entry in the message log and the results of that test will be displayed again.
- You can also click on the link to the right of the **Show Operations** link to display the WSDL file that was generated automatically for your web service.

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



## Advanced Web Services Tutorial: Step 3: Create a Service Control to Access the CreditScore Web Service

In this step, you will enhance the logic of the LoanApproval control to access the **CreditScore** web service. The existing logic of the LoanApproval control is:

```
If a loan exists for this SSN then turn down the application (return false to caller)
    otherwise accept the application (return true to the caller).
```

We are going to expand that logic to:

```
If the person with this ssn already has a loan then turn down their application (return false)
    otherwise check the credit score. If the credit score is < 700, turn down the application (return
false).
If the credit score is 700 or higher, accept the application (return true).
```

To access the external web service, we will create a new web service control ("service control") to access the **CreditScore** web service. We will then modify the **LoanApproval** control to use the new web service control.

### To Create a Control to Access a Web Service

To create a new control we will first generate a WSDL file from a web service and then generate a service control from the WSDL.

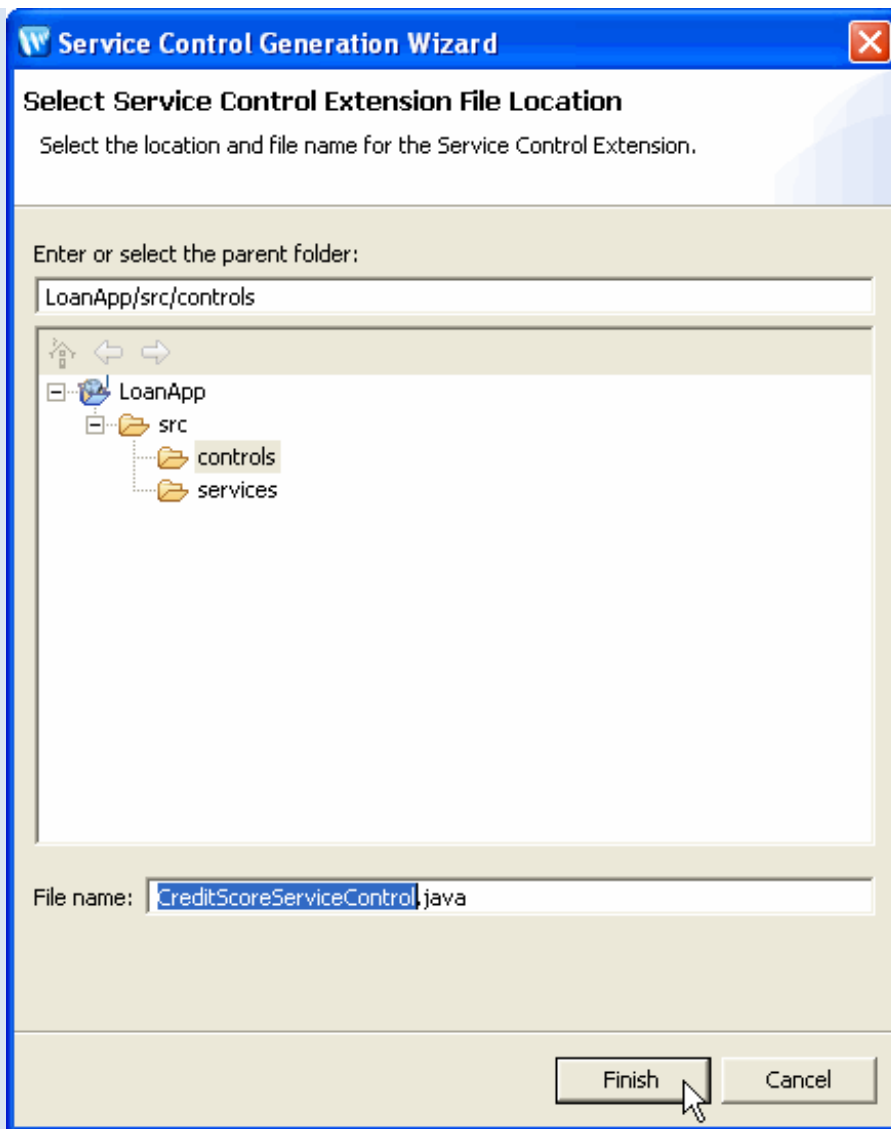
1. On the **Package Explorer** view, open the nodes **CreditScoreWS > src > services**, right-click the web service **CreditScoreService.java**, and select **Web Services > Generate WSDL**.

We will use this WSDL to automatically generate our new control.

2. To copy the WSDL file, right-click on the file **CreditScoreWS/src/services/CreditScoreService.wsdl** and select **Copy**.
3. Right-click on the **LoanApp/src/controls** package and select **Paste**.
4. Generate a web service control by right-clicking on **LoanApp/src/services/CreditScoreService.wsdl** and selecting **Web Services > Generate Service Control**.

Confirm that the name for the service control is **CreditScoreServiceControl.java**. Click **Finish**.





5. In the **Package Explorer** view, double click the file **LoanApprovalControlImpl.java** to open it in the editor.

Right click on the editor window and choose **Insert > Control**. Choose the new **CreditScoreServiceControl** and click **OK**.

Code will appear that declares and instantiates the control

```
@Control
private CreditScoreServiceControl creditScoreServiceControl;
```

6. Now replace the current code for **getLoanApproval** method of the **LoanApproval** control to expand its logic and use the new web service control.

```
public boolean getLoanApproval(int ssn, float amount) throws SQLException
{
    init();

    // if they are already borrowing, don't allow another loan
    if (loansDB.getLoanValue(ssn) > 0)
        return false;

    if(creditScoreServiceControl.getCreditScore(ssn) < 700)
        return false;

    // otherwise, allow the loan.
    loansDB.insertLoan(ssn, amount);
    return true;
}
```

```
}
```

Save your changes with **File > Save**.

7. Test the updated web service by right clicking on **LoanApplicationService.java** in the **services** package of the **LoanApp** project and choosing **Run As > Run on Server**.

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



## Summary: Advanced Web Services Tutorial

This topic lists the ideas this tutorial introduced, along with links to topics for more information. You may also find it useful to look at the following:

- [Tutorial: Getting Started](#) describes the Workshop for WebLogic interface and discusses navigation, common tasks and documentation resources.
- [Tutorial: Web Service](#) contains a simple example of creating a web service and a simple custom control.
- [Tutorial: Accessing a Database from a Web Application](#) describes how to integrate database operations into web applications using controls.

### Concepts and Tasks Introduced in This Tutorial

- A *control* is a Java object that provides standardized access to resources or encapsulated business logic. Controls use Java 5 metadata annotations for more convenient configuration. Controls in Workshop for WebLogic are based on the Beehive open source framework, described in detail at [Working with Beehive Controls](#).
- The easiest way to access a web service from an application is to create a control for the web service. Workshop for WebLogic can use the WSDL file for a web service to automatically create a control.
- Once a control has been created, you can access a web service's operations through simple method calls. The methods exchange SOAP messages with the web service to perform the requested operations.

Click the arrow to navigate back through the tutorial:



## Introduction to Web Service Technologies

A web service makes software application resources available over networks using standard technologies. Because web services are based on standard interfaces, they can communicate even if they are running on different operating systems and are written in different languages. For this reason they are an excellent approach for building distributed applications that must incorporate diverse systems over a network.

The following topic outlines the standard technologies that you use to build web services and the advanced functionality available through asynchronous web services.

### Standard Technologies

Web services are able to expose their resources in this generally accessible way because they adhere to recognized standards. A web service:

- Publicly describes its own functionality through a WSDL file
- Communicates with other applications via XML messages, often formatted with SOAP
- Employs a standard network protocol such as HTTP

### WSDL Files

The Web Service Description Language (WSDL) is a standard XML format for describing web services. A WSDL file describes a particular web service so that other software applications can interface with it.

WSDLs are generally publicly accessible and provide enough detail so that potential clients can figure out how to operate the service solely from reading the WSDL file. If a web service translates English sentences into French, the WSDL file will explain how the English sentences should be sent to the web service, and how the French translation will be returned to the requesting client. For more information on WSDL files see [WSDL Files: Web Service Descriptions](#).

### XML and SOAP

Extensible Markup Language (XML) messages provide a common language by which different applications can talk to one another over a network. Most web services communicate via XML. A client sends an XML message containing a request to the web service, and the web service responds with an XML message containing the results of the operation. In most cases these 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. The types of messages supported by a particular web service are delineated in the service's WSDL file.

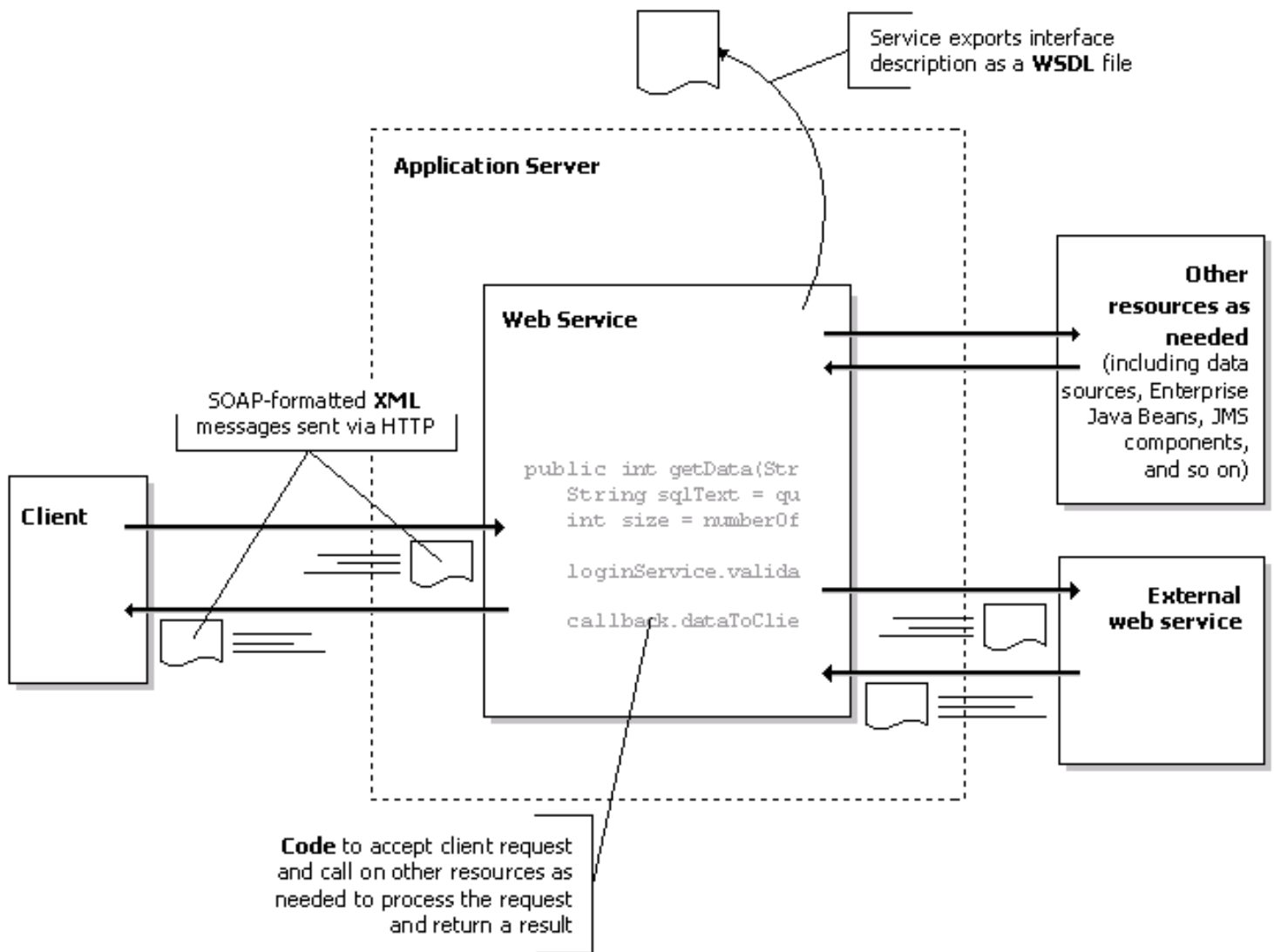
### Network Protocols

Web services receive requests and send responses using widely used protocols such as HyperText Transfer Protocol (HTTP) and Java Message Service (JMS). A web service may support more than one protocol. The protocols that a web service supports are published in the WSDL file.

### Web Service Architecture

The following illustration shows the relationship between a web service (in the center), its client software applications (on the left), and the resources it uses, including databases, other web services, and so on (on the right). A web

service communicates with clients and resources over standard protocols such as HTTP by exchanging XML messages. The WebLogic Server on which the web service is deployed is responsible for routing incoming XML messages to the web service code that you write. The web service exports a WSDL file to describe its interface, which other developers may use to write components to access the service.



## Asynchronous Web Services

Many business processes take more than a few moments to complete, but traditional architectures make it hard to handle long-running tasks efficiently. Workshop for WebLogic helps you architect asynchronous web services easily using conversations and callbacks. Conversations help manage the typical problems in asynchronous messaging, namely correlating messages and managing some information or state between message exchanges. In an ongoing conversation, a web service can notify a client when the results of an operation are ready using a callback.

In addition, WebLogic Server supports the use of Java Message Service (JMS) queues as message buffers to ensure that web service messages are not lost regardless of server load. JMS can also be used to communicate with back end resources. For more information on using buffers see [Creating Buffered Web Services](#) in the WebLogic Server documentation.

## Two Models for Asynchronous Computing with Web Services

BEA WebLogic Platform supports two models for asynchronous web services. One model uses "callbacks"; the other uses "asynchronous request-response". The two models differ in the way that the web service and the client divide up the work of coordinating the asynchronous communication.

## Callbacks

On the callback model of asynchronous web services, both the web service and its client (a web service control) are specially designed for asynchronous communication with one another. On this model, the web service is explicitly designed to be called asynchronously, including specially annotated callback methods that send data back to the client. Similarly the web service control is explicitly designed to listen for and receive callbacks from the web service using specially annotated event set methods.

For more information on the callback model see [Web Service Callbacks](#).

## Asynchronous Request-Response

The asynchronous request-response model places all of the burden of asynchronous coordination on the client. On this model the target web service does not need to be explicitly designed to be asynchronously called. The only requirement of the target web service is that it comply with the [WS-Addressing](#) standard. The client takes on all of the burden of coordinating the asynchronous response or failures that are later returned by the web service.

For more information on the *asynchronous request-response* model, see [Invoking a Web Service Using Asynchronous Request-Response](#).

## Related Topics

[Building Web Services with Workshop for WebLogic](#)

## Building Web Services with Workshop for WebLogic

You can build enterprise-class web services with Workshop for WebLogic. Web services built with Workshop for WebLogic employ standard web service technologies: XML, SOAP, and WSDL. Workshop for WebLogic simplifies web service development by allowing you to focus on application logic, rather than the complex implementation details traditionally required by these technologies.

Workshop for WebLogic also offers the web service Design View, a graphical tool for designing, creating, and edit web services.

The following sections explain the basic concepts that you need to know about to begin building web services with Workshop for WebLogic, and point you to more in-depth information about each.

### Web Service Design View

The web service Design View gives a graphical, intuitive view of a web service and its operations. It also makes it easy to perform complex coding and design tasks. For more information on the Design View see [Using Design View to Create Web Services](#).

### Web Service Projects

You build web services within a [web service project](#). A web service project corresponds to a J2EE web application with the addition of facets to support web services. You may build multiple web services within a single project.

For more information on applications and projects, see [Applications and Projects](#).

### The Web Service Class

The web service class is the core of your web service. It is an ordinary Java class (decorated with the [@WebService](#) annotation) that determines how your web service behaves, often through the use of one or more controls that contain the web service's application logic. You can think of a web service built on Workshop for WebLogic as a Java class which communicates with the outside world through XML messages. This documentation assumes you are familiar with Java programming.

You design a web service in the Workshop for WebLogic integrated development environment.

### Methods and Callbacks

Your web service has a public interface that clients may call over the internet. This interface is made up of methods and callbacks. The methods that your web service exposes are called by clients; the callbacks are methods on the client that your web service calls to send information back to the client. These methods and callbacks are available over the internet because they are

decorated with the [@WebMethod](#) annotation.

Within your web service code, you may also have non-public methods that are not exposed to clients. These methods perform internal functions in your web service. These methods are not decorated with [@WebMethod](#).

The controls that your web service uses also expose methods and events. Your web service functions as a client of the control, calling its methods and implementing its event handlers.

## Custom Controls

You can use custom controls in your web service to implement the application logic of your web service. Custom controls in turn use system controls to access enterprise resources such as databases, legacy applications, and other web services. In other words, your web service interacts with a custom control by calling its control methods and implementing event handlers for its control events, and the custom control calls control methods and implements event handlers for any system controls it uses.

Workshop for WebLogic provides system controls for connecting to common resources. The system controls provided with Workshop for WebLogic are:

- The service control, for calling another web service
- The timer control, which notifies your web service when a specified period of time has elapsed or when a specified absolute time has been reached
- The EJB control, which provides simplified access to Enterprise Java Beans (EJBs).
- The JDBC control, which provides simplified access to a relational database
- The JMS control, which makes it easy to send and receive messages via a Java Message Service (JMS) topic or queue.

For more information about system controls, see [Using System Controls](#). For information on building custom controls, see [Custom Controls](#).

## Properties

Most of the elements that make up your web service-methods, callbacks, controls, and the web service itself-have properties that you can set to specify their characteristics. You can set properties in the **Annotations** view in the Workshop for WebLogic IDE. Each element of your web service has one or more annotations, each with a set of attributes, corresponding to the element's properties in the **Annotations** view. Properties are stored in your code as Java 5 annotations (beginning with @). You can also edit annotations them directly in the code editor if you wish.



## Conversations and Asynchronous Communication

Many processes take time to complete. When a client makes a request from a web service, if the web service doesn't return a response right away, the client may be left waiting for it, unable to continue other operations. Web services that you build with Workshop for WebLogic can address this problem by relying on asynchronous communication.

When a client and web service communicate asynchronously, the web service immediately acknowledges the client's request, then continues processing the request. The client is free to continue performing other work. For more information on building asynchronous web services, see [Designing Asynchronous Interfaces](#).

A web service and its client may also participate in a conversation. The conversation keeps track of state-related data for this exchange between client and service. The conversation correlates the client's requests and the service's response by means of a conversation ID, a unique identifier that is generated when the client initiates a conversation with the service.

For more information on conversations, see [Creating Conversational Web Services](#) in the WebLogic Sever documentation and [Tutorial: Creating a Web Service with Timer Control](#).

## Starting Points for Designing a Web Service

Developers often design web services around preexisting data structures and contracts. Two common starting points are WSDL files and XSD files.

### Starting from a WSDL

When building a web service, it is often easier to build the web service implementation around an already existing web service contract (a WSDL file). This method for creating a web service is sometimes called "WSDL first", "contract first" or "top down" web service design.

### Starting from a Schema File

Another common approach to design web services is to start with an XML schema file (an XSD file), compile XMLBeans from the schema, and then build a web service implementation centered on those XMLBean classes.

For more information on how Workshop for WebLogic supports both these approaches to web service design, see [Web Service Development Starting Points](#).

## Related Topics

[Designing Asynchronous Interfaces](#)

[Working with Controls](#)

## Creating Conversational Web Services

## Using Design View to Create Web Services

The web service Design View gives you a graphical overview and editing environment for web services.

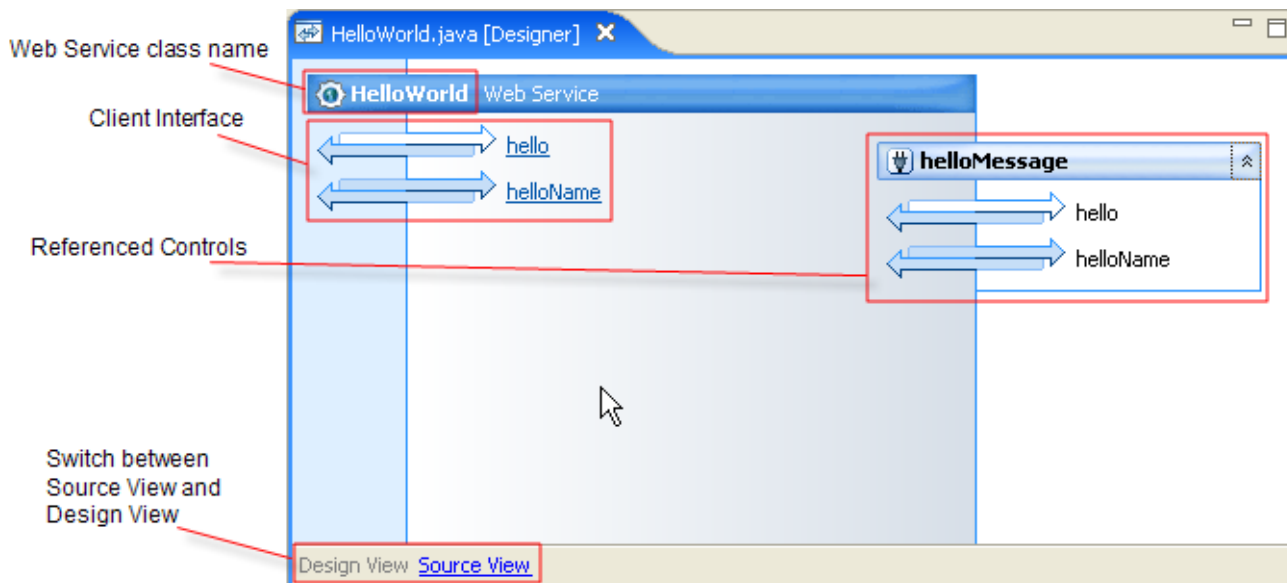
Design View is synchronized with the web service Source View, Annotations View, and all other views of the web service: when you make changes in one view, the changes are reflected in all the others.

This topic describes how to use the graphical elements in Design View to create web services.

### Design View Basics

The main areas of Design View are:

- The Header gives the class name of the web service being shown.
- The Client Interface (left side) represents the web service's methods and callbacks.
- The Referenced Controls area (right side) represents the controls used by the web service.
- The bottom of the view gives links for switching between Source View and Design View.

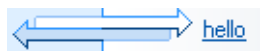


### Client Interface

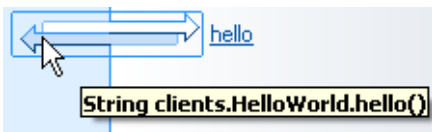
The Client Interface only shows those methods that are publicly accessible over the internet. These include the web service's "web methods" (those methods that are annotated with `@WebMethod`). Other methods are visible only in Source View. (An exception to this rule is when no methods are annotated with `@WebMethod`. In that case, all of the methods are displayed in Design View, because in this case all are assumed to be web methods.)

### Methods and Callbacks

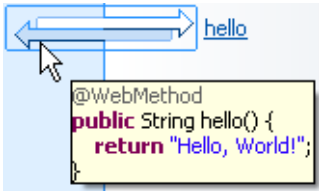
Each web method is represented by one or two arrows and a label.



Hovering over the arrows or label displays the method's return type, package, and class.



Holding down the **Ctrl** key and hovering over an element shows the corresponding source code.



Clicking the label brings you to the method's source code.



Edit the method signature by right-clicking and selecting **Edit Signature**.



## Method and Callback Icons

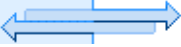



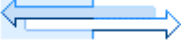

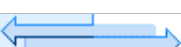
Method and callback icons are constructed according to the following rules:

- Two arrows indicates that the client expects a return value (even if that value is `void`).
- One arrow indicates the `@Oneway` annotation, which means that the client should not expect a return value.
- Methods are depicted by two arrows where the top arrow points to the right and link text that appears to the right of the arrows.
- Callbacks are depicted by two arrows where the top arrow points to the left link text that appears to the left of the arrows.
- A blue-colored top arrow represents a parameter set; a blue-colored bottom arrow represents a return value.

The table below shows some of the methods and callbacks represented on the client interface.

**Client Interface Method Representations**

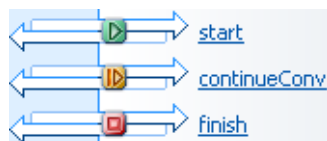
Method Type	Expects Data Parameters	Returns Data	Appearance	Source Code Example
Inbound (client invocable)	no	no		<pre>@WebMethod public void hello() { }</pre>
Inbound (client invocable)	yes	no		<pre>@WebMethod public void hello(String str) { }</pre>
Inbound (client invocable)	no	yes		<pre>@WebMethod public String hello() {     return "Hello, World"; }</pre>

Inbound (client invocable)	yes	yes		<code>@WebMethod public String helloName(String str){     return "Hello, " + str + "!"; }</code>
Inbound (One Way, client expects no return value)	no	no		<code>@Oneway() @WebMethod public void hello() { }</code>
Inbound (One Way, client expects no return value)	yes	no		<code>@Oneway() @WebMethod public void hello(String str) { }</code>
Callback	no	no		<code>@CallbackService public interface CallbackSvc extends CallbackInterface {     @WebMethod     public void callback(); }</code>
Callback	yes	no		<code>@CallbackService public interface CallbackSvc extends CallbackInterface {     @WebMethod     public void callback(String str); }</code>
Callback	no	yes		<code>@CallbackService public interface CallbackSvc extends CallbackInterface {     @WebMethod     public String callback(); }</code>
Callback	yes	yes		<code>@CallbackService public interface CallbackSvc extends CallbackInterface {     @WebMethod     public String callback(String str); }</code>




## Conversation Decorators

Conversation-related annotations are represented by decorator icons.

Conversation starting, continuing, and ending methods are represented by green, yellow, and red decorators, respectively.



The following table summarizes the conversation-related decorator icons.

Conversation Decorator Icons		
Decorator	Description	Source Code
	Indicates the method starts a conversation.	<code>@Conversation(Conversation.Phase.START)</code>
	Indicates the method continues a conversation.	<code>@Conversation(Conversation.Phase.CONTINUE)</code>
	Indicates the method finishes a conversation	<code>@Conversation(Conversation.Phase.FINISH)</code>

You can set conversation properties on a method by right-clicking on the method and choosing **Conversation Start**, **Conversation Continue**, or **Conversation Finish**.

For more information on conversations see [Creating Conversational Web Services](#) in the WebLogic Server documentation.



## Method Buffer Decorators

Method buffers are represented by the following icon:



You can place message buffer on a method by right-clicking on the method and choosing **Message Buffer** or **Message Buffer and Oneway**.

The following table summarizes the buffer-related decorator icons.

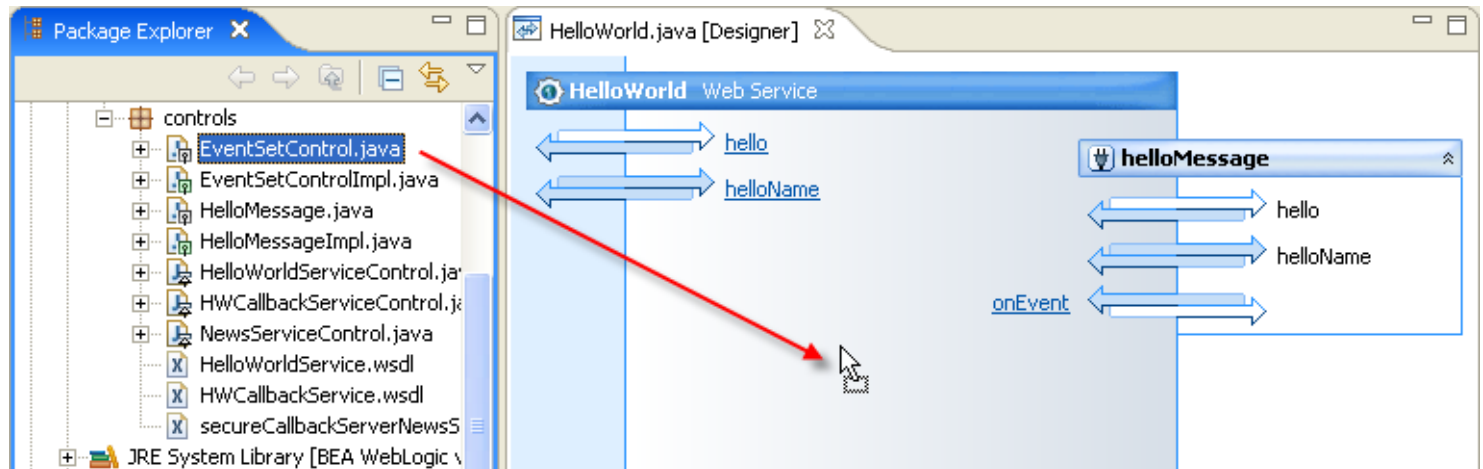
Buffer Decorator Icons		
Decorator	Description	Source Code
	Indicates a buffer is enabled on a method.	@MessageBuffer() is present on the method declaration.
	Indicates a buffer is enabled at the class level.	@MessageBuffer() is present on the class declaration

For more information on message buffers see [Creating Buffered Web Services](#) in the WebLogic Server documentation.

## Referenced Controls

Controls referenced by the web service are represented on the right side of Design View.

You can also add controls to the referenced controls area by dragging and dropping from the **Package Explorer** view.



The following example depicts a control declaration

The following control declaration...

```
@Control
private HelloMessage helloMessage;
```

...is depicted in Design View as shown below.

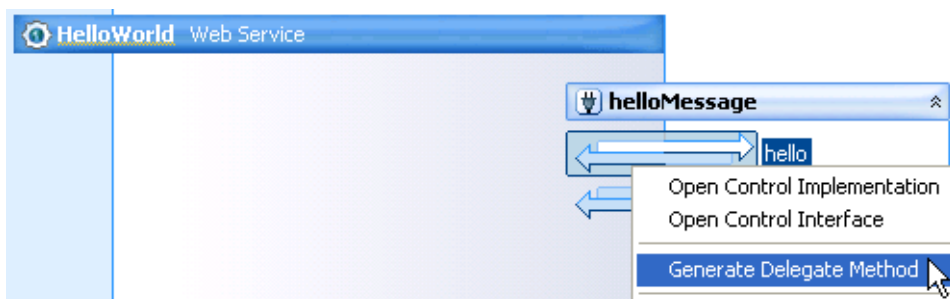


Note that the variable name **helloMessage** is shown in the Design View, not the Control class name.

All methods in the controls interface file are shown in Design View.

### Generating Client Interface Methods

To generate a client interface method that calls a referenced control method, right-click the control method and select **Generate Delegate Method**.



The corresponding method is created in the client interface.

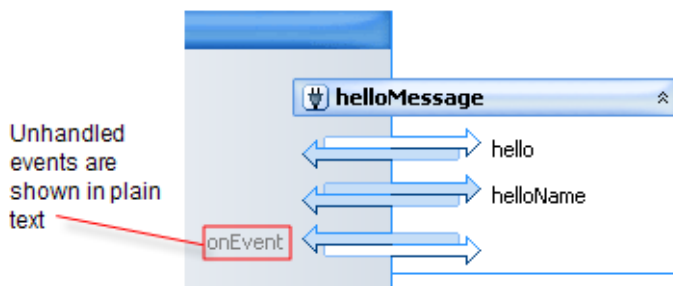


### Event Handlers

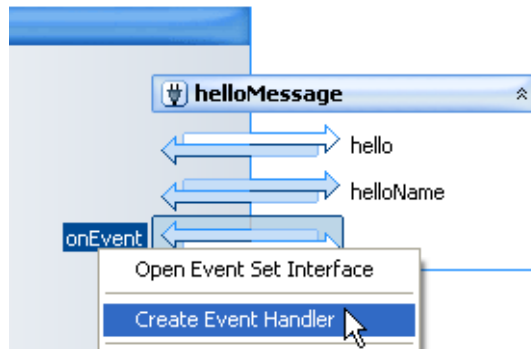
Event handler methods in the web service are shown in Referenced Controls area. Event handlers are displayed with clickable link text. Clicking on the link text will take you to the event handler source code in the web service.



Unhandled events in controls are shown in plain text.



To add an event handler to the web service, right-click on the unhandled event, and select **Create Event Handler**.



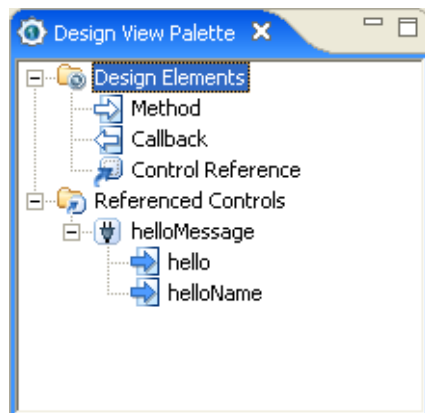
An event handler signature is added to the web service, for example:

```
@EventHandler(field = "helloMessage", eventSet = HelloMessage.NewEventSet.class, eventName = "onEvent")
protected void helloMessage_NewEventSet_onEvent(String msg) {

}
```

## Design View Palette

When the Design View is active it is accompanied by the **Design View Palette**.



You can add items to the **Design View** by dragging and dropping items from the **Design View Palette** (or by double-clicking on those same items).

You can add new control references, methods, and callbacks.

## Common Tasks

### Keyboard Shortcuts



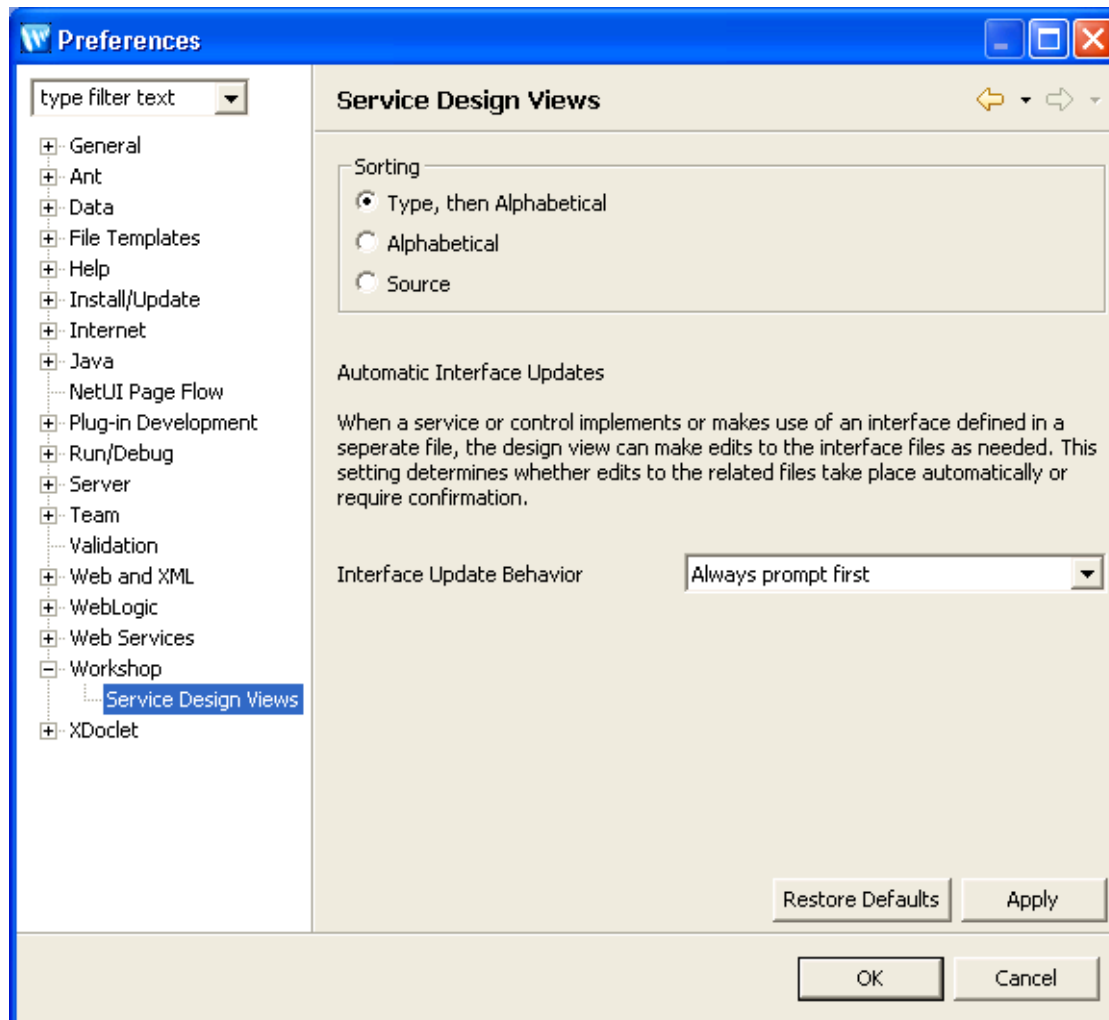
**Keyboard Shortcuts**

Key Stroke	Event
Shift-F10	Shows context menu on selected item
Tab / Shift-Tab	Moves forwards/backwards through referenced controls and the web service class. (Note that pressing Tab will also move focus to the Eclipse toolbar, but focus will eventually return to the Design View elements, provided that the Tab key is pressed a sufficient number of times.)
Up and down arrow keys	Cycles through the selectable elements including: individual methods, callbacks, event handlers, referenced controls, and the entire Design View canvas.
Left and right arrow keys	When a referenced control is selected, the left and right arrows expand and collapse the referenced control.

**Setting Preferences**

You can set Design View preferences using the **Service Design Views** dialog available at **Windows > Prefs > Workshop > Service Design Views**.

For more information on this dialog see [Service Design Views Preferences](#).

**Related Topics**

[Designing Asynchronous Interfaces](#)

[Working with Controls](#)

[Creating Conversational Web Services](#)

## Web Service Development Starting Points

This topic describes three different starting points for developing web services:

- [Starting from a WSDL](#)
- [Starting from an XML Schema](#)
- [Starting from a Java Class](#)

### Starting from a WSDL

In this approach to developing a web service, you begin by defining the WSDL file (or getting a pre-existing one). This is the web service contract that defines how the web service communicates with clients, including the data types conveyed, the available methods, and the protocols and message formats used. Hence, this approach to web service development is sometimes called "contract first" or "top down" development. Note that you can only use JAX-RPC types when using this development approach, XMLBean types are not available.

To generate a web service from a WSDL:

1. Import the WSDL into a [web service project](#).
2. In the Package Explorer or Navigator view, right-click the WSDL and select **Web Services > Generate Web Service**.

Two artifacts will be created: a web service implementation class and a JAR file. The web service class will contain the web methods described by the WSDL (the publicly accessible methods and callbacks) without any method bodies. The developer must fill in the web service's implementation details. The JAR file contains a web service interface class and types referenced in the original WSDL and is located in the project's WEB-INF/lib directory.

For example the generated web service implementation class will resemble the following:

```
@WebService(
    serviceName="MailingListServiceService",
    targetNamespace="http://services",
    endpointInterface="model.MailingListService")
@WLHttpTransport(contextPath="ServicesWeb",serviceUri="MailingListService",
portName="MailingListServiceSoapPort")
public class MailingListServiceImpl implements MailingListService {

    public MailingListServiceImpl() {

    }

    public java.lang.String getCustomers() {
        //replace with your impl here
        return null;
    }
}
```

Notice that this class implements MailingListService, the interface file found in the generated JAR file.

The developer must fill in the method body for the method getCustomers().

### Starting from an XML Schema

In this approach to developing a web service, you begin with an XML schema (XSD file) that defines XML data structures to

be used as parameters and return types in the web service operations. XMLBean Java types are then automatically generated from the schema for use in your web service. This approach gives you the following benefit of a common set of data structures for all of the web services in a project or set of projects.

Once the XMLBean types are available, web service development proceeds according to the "[Start with a Java Class](#)" method described below.

To develop starting from an XML schema:

1. [Enable the XML beans builder facet](#) on your [web service project](#).
2. Import the schema into the project's `schema` directory. (The `schema` directory is automatically created when the builder facet is added to the project.)

Schemas will be automatically compiled into XMLBeans, which you can use in your web service. The XMLBean types will be automatically re-compiled whenever the schemas in the `schema` directory are updated.

**Note:** You do not have to use the XML beans builder facet to create XMLBean types. Alternatively, you can generate a JAR by right-clicking any XSD or WSDL in the project and selecting **Web Services > Generate Types JAR File**. This will open the [Types JAR File Generation Wizard](#), from which you can generate a JAR containing XMLBeans. Or, if you already have a JAR containing the XMLBean types, you can import it into the project and use those types in a web service. Neither of these options provide automatic updating of the XMLBean JAR when the original schema changes.

Available XMLBean types can be seen in the Navigator view (**Window > Show View > Navigator**) in the directory `.xbean_src`. The `.xbean_src` and `.xbean_bin` directories contain generated files that should never be directly edited.

The following example shows one way to incorporate XMLBean types into a web service. Suppose you import the following schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ws="http://openuri.org/bean/samples/workshop"
targetNamespace="http://openuri.org/bean/samples/workshop" elementFormDefault="qualified">
  <xs:element name="applicant">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element name="bankrupt" type="xs:boolean"/>
        <xs:element name="name_first" type="xs:string"/>
        <xs:element name="name_last" type="xs:string"/>
        <xs:element name="risk_estimate" type="xs:string"/>
        <xs:element name="score_info" type="ws:score_infoType"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="score_infoType" mixed="true">
    <xs:choice minOccurs="0" maxOccurs="1">
      <xs:element name="credit_score" type="xs:short"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

The corresponding generated XMLBean types are `ApplicantDocument`, `ScoreInfoType`, etc.

The following web service method uses the `ApplicantDocument` as an input parameter and performs a simple risk assessment calculation.

```
@WebMethod
public String getRiskEstimate(ApplicantDocument appDoc) {

    boolean bankrupt = appDoc.getApplicant().getBankrupt();
    short balanceRemain = appDoc.getApplicant().getBalanceRemaining();
```

```
    if (bankrupt == true && balanceRemain < 200)
        appDoc.getApplicant().setRiskEstimate("high");
    else
        appDoc.getApplicant().setRiskEstimate("low");

    return appDoc.getApplicant().getRiskEstimate();
}
```

For detailed information about using XMLBeans see [Using XMLBeans in the IDE](#).

## Starting from a Java Class

In this approach, you develop a web service as a Java class. Methods become web service operations and method parameters and return types can be simple Java Beans. The Java class is annotated to indicate what methods should be exposed and to set other properties for the service. The following guidelines will help you utilize all of Workshop for WebLogic's web service features.

1. Create a [new web service](#) class (**File > New > Web Service**) in an appropriate package within your [web service project](#). To learn about Workshop for WebLogic projects, see [Applications and Projects](#).
2. All of the following steps can be accomplished using the web service Design View, a graphical editing environment for creating web services. For more information see [Using Design View to Create Web Services](#).
3. Add the methods your web service will expose and configure each method's parameters.
4. Add any callbacks your web service will expose and configure each callback's parameters. To learn more about callbacks see [Web Service Callbacks](#).
5. Implement event handlers for relevant events from controls that the web service utilizes. To learn more about event handlers see [Handling Control Events](#) and [Handling Web Service Callback Messages](#).
6. Determine and configure the conversation phase of each method and callback. To learn more about conversations, see [Designing Conversational Web Services](#).
7. Determine and configure any buffered methods. To learn more about message buffers see [Creating Buffered Web Services](#) in the WebLogic Server documentation.
8. Once your web service is complete, you can generate a WSDL file by right-clicking on the web service in the Package Explorer and selecting **Web Services > Generate WSDL**.

## Related Topics

[WSDL Files: Web Service Descriptions](#)

## Testing Web Services with the Test Client

As you develop a web service, you can typically test it directly by using the Test Client. In some cases, you will need to test indirectly by creating a separate web service that acts as a client for testing.

### Installing the Test Client

The Test Client is included with Workshop for WebLogic Update (in a ZIP file), but not installed. Use the following instructions to install it.

1. Locate the Test Client ZIP file at:

BEA\_HOME\workshop92\workshop4WP\eclipse\features\com.bea.wlw.workshop\_9.2.1.0\archives\wlstestclient.zip

2. Extract the contents of the zip file to a temporary directory such as c:\temp\wlstestclient.
3. If the server is running, shut it down before proceeding to the next step.
4. Locate the existing Test Client EAR file at BEA\_HOME\weblogic92\server\lib and back it up to another location.
5. Copy the new wlstestclient.ear file to the BEA\_HOME\weblogic92\server\lib directory, replacing the old one.
6. Restart the server.

### Testing Web Services with the Test Client

The Test Client provides a user interface through which you can test web service operations with parameter values you choose. With the Test Client you can:

- Test a web service from the project tree.
- Choose which operation you want to test.
- Examine operation and callback results.
- View the WSDL for the web service you're currently testing.
- Choose another web service to test.

**Note:** You can also launch the Test Client without using the IDE.

For an example of using the Test Client, see Web Service Tutorial: Step 4: Test the Web Service.

### Test Client User Interface

Displays the list of operations exposed by this web service.

Displays the operations and callbacks tested so far. Click a log entry to view more information about it.

Displays the WSDL for this web service.

The operations exposed by this web service. Enter test values and click an operation name to test it.

Displays a form through which you can select the WSDL of another web service to test.

The screenshot shows the WebLogic Test Client interface. At the top, the browser address bar displays `http://localhost:7001/wls_UTC/begin.do`. The page title is "WebLogic Test Client". The interface is divided into several sections:

- Show Operations:** A sidebar on the left containing a "Message Log" with entries for `requestMessageSynchronous`, `requestMessageAsynchronous`, and `callback.onMessage`, along with a "Clear Log" button.
- WSDL Section:** A central area displaying the URL `http://localhost:7001/WebServices/HelloWorldSyncAsync?WSDL`.
- Choose another WSDL:** A button on the right side of the interface.
- Operations Section:** A large area containing two forms:
  - requestMessageAsynchronous:** Includes a "string name:" input field and a button labeled "requestMessageAsynchronous".
  - requestMessageSynchronous:** Includes a "string name:" input field and a button labeled "requestMessageSynchronous".

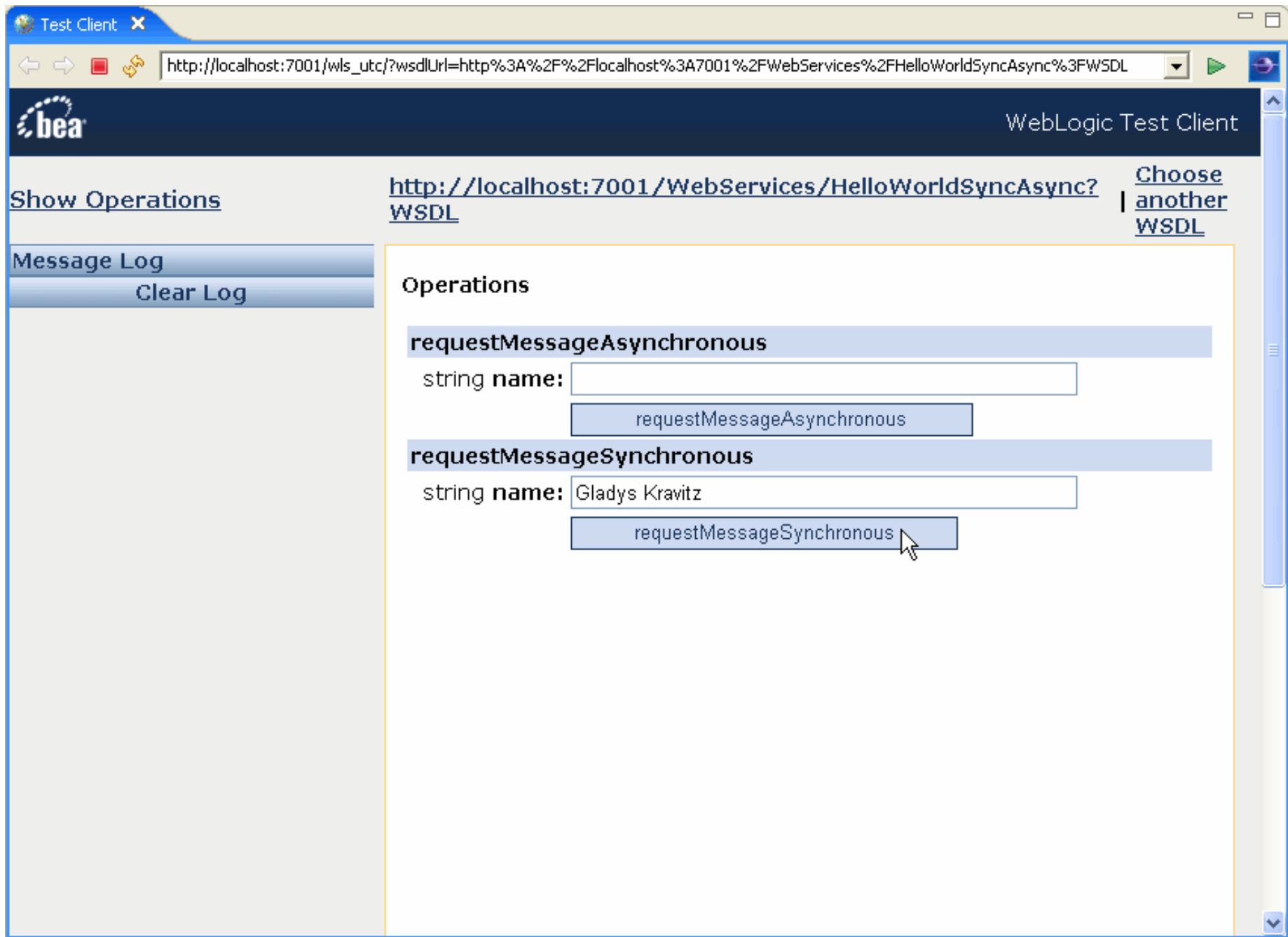
## Basic Testing Steps

When you test web services with Workshop for WebLogic, you follow simple steps that launch the Test Client with a visual interface for invoking the web service's operations. Briefly, these steps are:

1. Start WebLogic Server.
2. Expand the project tree to display the web service source file.
3. Right-click the source file, then click **Run As -> Run on Server**.
4. When the Test Client is displayed, choose the operation you want to test.
5. If the operation has parameters, enter test values in the boxes provided.
6. Click the button labeled with the operation's name.
7. Examine the result of the test.
8. Use the **Message Log** list to view the results of multiple tests.
9. If the web service is designed to receive a callback, click the callback's name in the **Message Log** list to view callback values. (You might need to refresh the Test Client if the callback is not designed to execute right away.)
10. Click **Show Operations** to begin another test.

## Choosing Operations to Test

When the Test Client is displayed, you choose an operation to test by clicking the button labeled with the operation's name. If the operation has parameters, the Test Client provides boxes for you to enter the values to test with.



## Complex Types as Parameters

When an operation includes complex types as parameters, the Test Client will display an XML template with placeholders for your test values. For example, the following illustration shows a template in which "Gladys Kravitz" has been entered for one String placeholder and the other placeholder is about to be replaced with a test value.



## requestMessageSynchronousComplex

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
```

```
    <requestMessageSynchronousComplex xmlns="http://services"
xmlns:java="java:classes">
      <newPeople>
        <java:firstPerson>Gladys Kravitz</java:firstPerson>
        <java:secondPerson>string</java:secondPerson>
      </newPeople>
    </requestMessageSynchronousComplex>
```

```
</soapenv:Body>
</soapenv:Envelope>
```

requestMessageSynchronousComplex

## Navigating Conversational Web Service Tests

The Test Client provides special links through which you can test conversational web services.

When testing a conversational web service, the Test Client will only display the operations that are valid in the current phase of the conversation. In other words, when you begin testing, only START methods show. You click the Continue this conversation (or the conversation's log heading — such as "Conversation 3" in the following illustration) link to display the list of operations after you invoke a START method — then only CONTINUE and FINISH methods are displayed.

Note that the message log groups the operations invoked according to the conversation in which they were tested with each message shown chronologically within the conversation.

Click Start New Conversation to display the list of operations so that you can choose one and start a new conversation.

The screenshot shows the BEA Test Client interface. On the left, there is a 'Message Log' panel with two entries: 'Conversation 2' and '\* Conversation 3'. Each entry has a right-pointing arrow for 'requestMessageAsynchronous' and a left-pointing arrow for 'callback.onMessage'. Below the log is a 'Clear Log' button. On the right, the URL is 'http://localhost:7001/WebServices/HelloWorldSyncAsync?WSDL'. A 'Continue this conversation' button is highlighted. Below that, a 'requestMessageAsynchronous Request Summary' is displayed with the following details:

```

Arguments:
  string name: Darren Stevens
Returned:
  [void]
Submitted:
  Mon Aug 28 20:52:00 PDT 2006
Duration:
  350 ms

```

## Examining Message Contents

When you execute an operation, the Test Client refreshes to display information about the message exchanged by the operation. The user interface provides a summary of message values as well as the message XML itself. This information is provided for both operation messages and callback messages. When an exception occurs, a fault message is displayed.

Notice that in the message XML, all but the most important parts of the message payload are displayed in grey.

## Operation Messages

After you have executed a web service operation, the Test Client displays information about messages related to the operation. The request summary provides a shorthand version of the message's contents. It gives parameter and return values (if any), along with time stamp information.

Each test of a web service operation will have its own entry in the Message Log list. In this way you can compare tests that use different values.

The screenshot shows the Test Client window with the URL `http://localhost:7001/wls_utc/callOperation.do`. The interface displays the following information:

- Show Operations**: <http://localhost:7001/WebServices/HelloWorldSyncAsync?WSDL> | [Ch](#)
- Message Log**: [requestMessageSynchronous](#) (selected), [Clear Log](#)
- requestMessageSynchronous Request Summary**:
  - Arguments: string name: **Gladys Kravitz**
  - Returned: **Hello, Gladys Kravitz!**
  - Submitted: Mon Aug 28 15:24:03 PDT 2006
  - Duration: 80 ms
- requestMessageSynchronous Request Detail**:
  - Service Request**
  - XML payload:
 

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <callback:CallbackTo xmlns:callback="http://www.openuri.org/2006/03/callback"
      <wsa:Address
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://127.0.0.1:7001/
      </wsa:Address>
    </callback:CallbackTo>
  </Header>
  <Body>
    <testclient:callback xmlns:testclient="http://www.bea.com/2006/05/testclient"
      <testclient:wSDL>http://localhost:7001/WebServices/HelloWorldSyncAsync?WSDL
      <testclient:key>8e1dbf8c-7091-42ac-b60b-498dd01dd5dc</testclient:key>
      <testclient:service>{http://services}HelloWorldSyncAsyncService</testclient:service>
      <testclient:port>HelloWorldSyncAsyncSoapPort</testclient:port>
    </testclient:callback>
  </Body>
</soapenv:Envelope>
```

Beneath the request summary the message XML is displayed, as shown in the following image. Messages for both the operation's request and its return value are displayed.

The partial screenshot shows the Test Client window with the URL `http://localhost:7001/wls_utc/callOperation.do`.

Returned:  
**Hello, Gladys Kravitz!**  
 Submitted:  
 Mon Aug 28 15:24:03 PDT 2006  
 Duration:  
 80 ms

### requestMessageSynchronous Request Detail

#### Service Request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns="http://services">
  <Header xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <callback:CallbackTo xmlns:callback="http://www.openuri.org/2006/03/callback">
      <wsa:Address
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://127.0.0.1:7001/wls_utc/CallbackHand
      <wsa:ReferenceParameters xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
        <testclient:callback xmlns:testclient="http://www.bea.com/2006/05/testclient">
          <testclient:wSDL>http://localhost:7001/WebServices/HelloWorldSyncAsync?WSDL</testclient:wSDL>
          <testclient:key>8e1dbf8c-7091-42ac-b60b-498dd01dd5dc</testclient:key>
          <testclient:service>{http://services}HelloWorldSyncAsyncService</testclient:service>
          <testclient:port>HelloWorldSyncAsyncSoapPort</testclient:port>
        </testclient:callback>
      </wsa:ReferenceParameters>
    </callback:CallbackTo>
  </Header>
  <soapenv:Body>
    <requestMessageSynchronous>
      <name>Gladys Kravitz</name>
    </requestMessageSynchronous>
  </soapenv:Body>
</soapenv:Envelope>
```

#### Service Response

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header />
  <soapenv:Body>
    <m:requestMessageSynchronousResponse xmlns:m="http://services">
      <m:return>Hello, Gladys Kravitz!</m:return>
    </m:requestMessageSynchronousResponse>
  </soapenv:Body>
</soapenv:Envelope>
```



## Callback Messages

If your web service sends a callback, you can view the results of the callback's execution by clicking its name in the Message Log list. Note that because the callback log entry won't show up until after the callback executes, you might need to refresh the Test Client after an interval to get the entry (you can click the Test Client's Refresh link).

As with operation messages, the Test Client displays callback message data as a summary as well as the message XML. The callback request message will describe the data sent to your web service by the callback.

 A screenshot of the Test Client interface showing the details of a callback message. The browser address bar shows the URL: http://localhost:7001/wls\_utc/selectResult.do?result=43f7d6ca-88b7-451f-b65c-046a782c4161&conversationId=None. The Message Log on the left shows 'callback.onMessage' selected. The main pane displays the following information:
 

- onMessage Request Summary**
  - Arguments:
    - string aMessage: [Test Client Test Data]
  - Returned: Mon Aug 28 15:36:29 PDT 2006
- onMessage Request Detail**
- Callback Request**

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    <wsa:MessageID>uuid:8702a11dd54ec0f8:72c9f085:10d55f377c8:-7ff0</wsa:MessageID>
    <wsa:Action>http://services/HelloWorldSyncAsyncService_CallbackSvc/
    <n1:callback xmlns:n1="http://www.bea.com/2006/05/testclient">
      <testclient:wSDL
        xmlns:testclient="http://www.bea.com/2006/05/testclient">http://localhost:7001
        WSDL</testclient:wSDL>
        <testclient:key xmlns:testclient="http://www.bea.com/2006/05/testclient">00
        9745d48c9473</testclient:key>
        <testclient:service xmlns:testclient="http://www.bea.com/2006/05/testclient">
        HelloWorldSyncAsyncService</testclient:service>
        <testclient:port
          xmlns:testclient="http://www.bea.com/2006/05/testclient">HelloWorldSyncAsyn
        </n1:callback>
      </testclient:callback>
    </wsa:Header>
  </soapenv:Header>
  <soapenv:Body>
    <wsc:TestResult xmlns:wsc="http://www.bea.com/2006/05/testclient">
      <wsc:TestData>[Test Client Test Data]</wsc:TestData>
    </wsc:TestResult>
  </soapenv:Body>
</soapenv:Envelope>
```

```

<wsa:ReplyTo>
  <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/r
</wsa:ReplyTo>
</soapenv:Header>
<soapenv:Body>
  <m:onMessage xmlns:m="http://services">
    <m:aMessage>Hello, Gladly Kravitz! (This message provided in a callback.
  </m:onMessage>
</soapenv:Body>
</soapenv:Envelope>

```

### Callback Response

Note: Because the Test Client is the "client" for the callback, this portion is just test

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header />
  <soapenv:Body>
    <m:onMessageResponse xmlns:m="onMessage" />
  </soapenv:Body>
</soapenv:Envelope>

```

## Exception Messages

When testing the web service generates a fault or exception, the Test Client displays the resulting message. Note in the following summary example that a fault has been noted. Here, a string was provided for the operation's argument rather than an int.

The message XML below is also displayed.

### guessMyWeight Request Summary

```

Arguments:
  int guessAmount: ninety-two
Fault:
  For input string: "ninety-two"
Submitted:
  Mon Aug 28 21:08:08 PDT 2006
Duration:
  60 ms

```

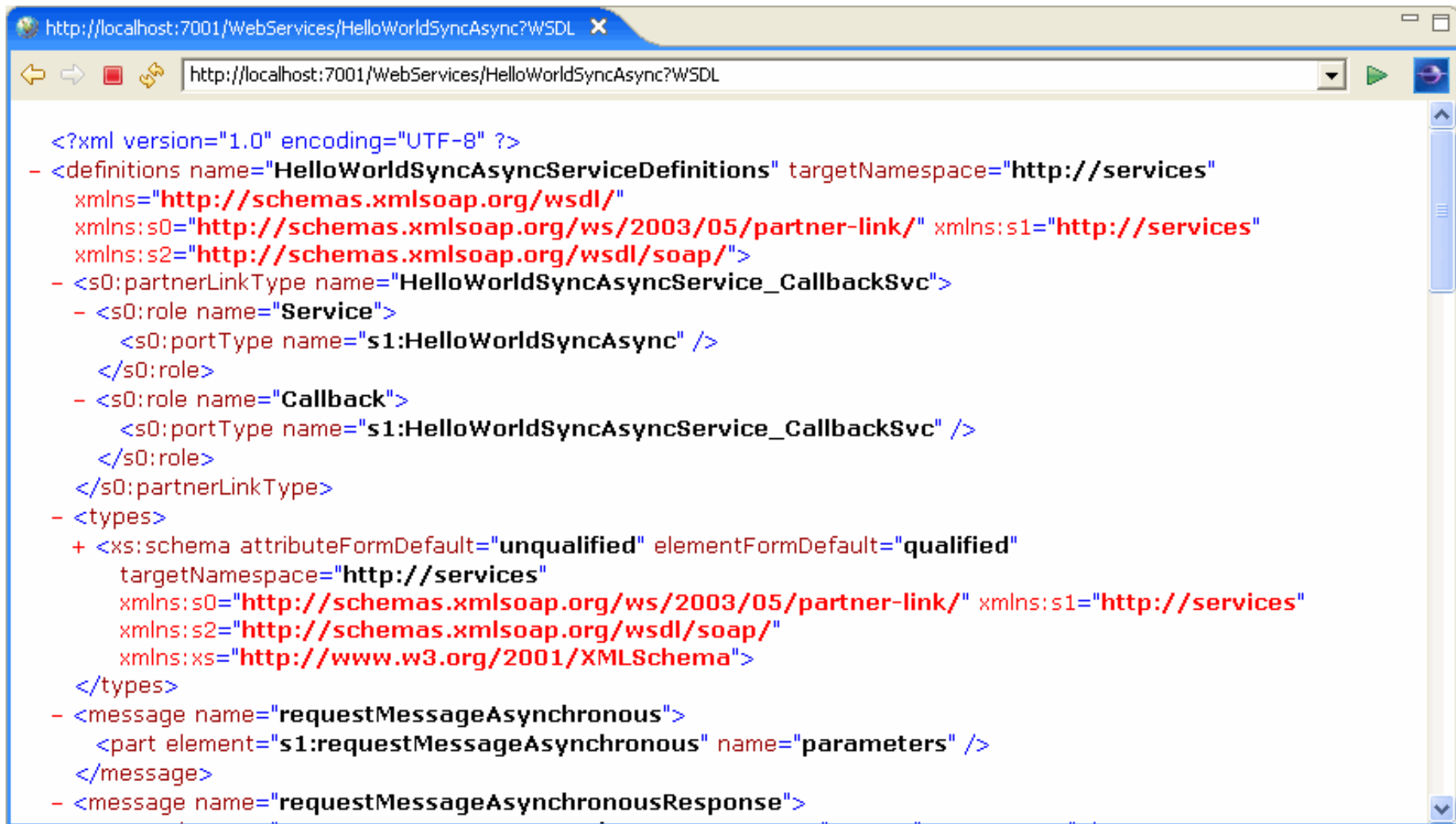
## Service Response

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header />
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>soapenv:Server</faultcode>
      <faultstring>For input string: "ninety-two"</faultstring>
      <detail>
        <bea_fault:stacktrace xmlns:bea_fault="http://www.bea.com/servers/wls70/webservice/fault/1.0.0" />
          java.lang.NumberFormatException: For input string: "ninety-two"
at com.bea.xbean.util.XsTypeConverter.parseIntXsdNumber(XsTypeConverter.java:671)
at com.bea.xbean.util.XsTypeConverter.parseInt(XsTypeConverter.java:624)
at com.bea.xbean.util.XsTypeConverter.lexInt(XsTypeConverter.java:268)
at weblogic.xml.dom.DOMStreamReaderExt.getIntValue(DOMStreamReaderExt.java:98)
at com.bea.staxb.runtime.internal.UnmarshalResult.getIntValue(UnmarshalResult.java:411)
at com.bea.staxb.runtime.internal.IntTypeConverter.getObject(IntTypeConverter.java:27)

```

## Viewing the WSDL File

You can view the WSDL file for the web service you're testing by clicking the WSDL URL provided at the top of the Test Client window.



```

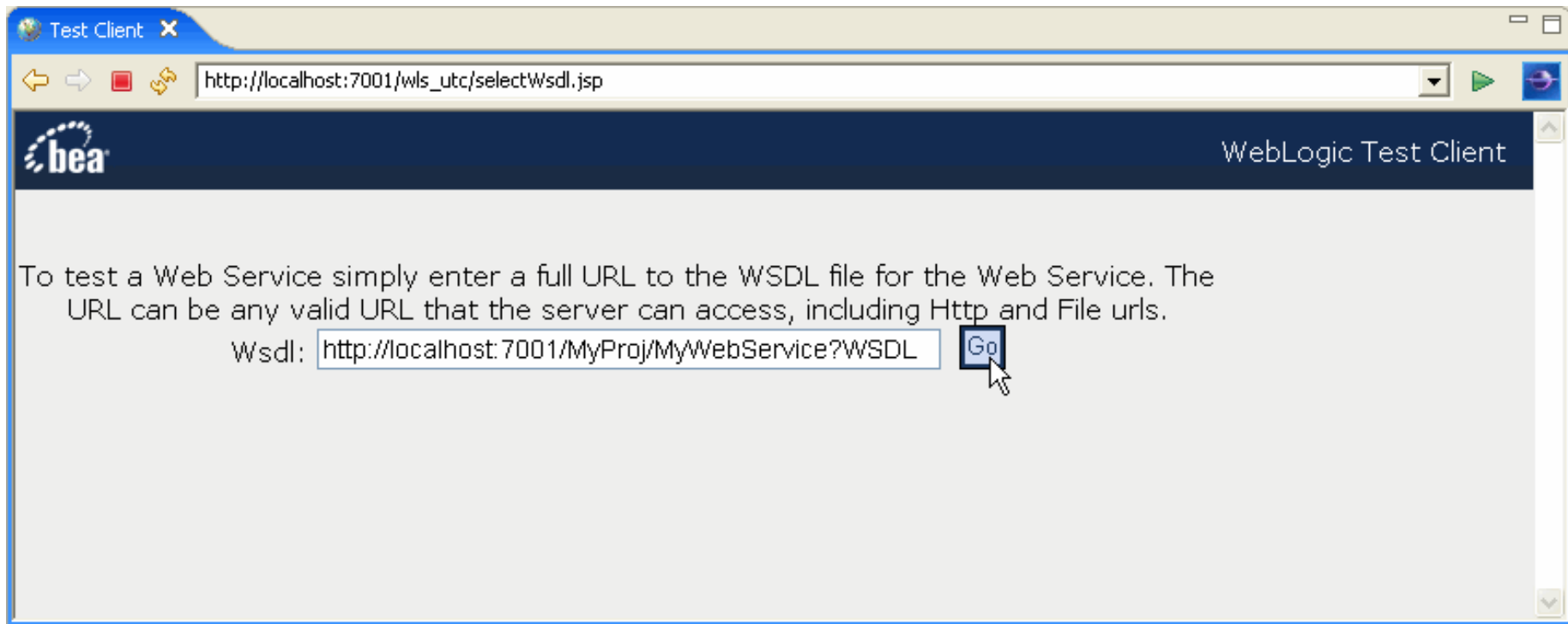
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions name="HelloWorldSyncAsyncServiceDefinitions" targetNamespace="http://services"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:s0="http://schemas.xmlsoap.org/ws/2003/05/partner-link/" xmlns:s1="http://services"
  xmlns:s2="http://schemas.xmlsoap.org/wsdl/soap/">
- <s0:partnerLinkType name="HelloWorldSyncAsyncService_CallbackSvc">
- <s0:role name="Service">
  <s0:portType name="s1:HelloWorldSyncAsync" />
</s0:role>
- <s0:role name="Callback">
  <s0:portType name="s1:HelloWorldSyncAsyncService_CallbackSvc" />
</s0:role>
</s0:partnerLinkType>
- <types>
+ <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  targetNamespace="http://services"
  xmlns:s0="http://schemas.xmlsoap.org/ws/2003/05/partner-link/" xmlns:s1="http://services"
  xmlns:s2="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
</types>
- <message name="requestMessageAsynchronous">
  <part element="s1:requestMessageAsynchronous" name="parameters" />
</message>
- <message name="requestMessageAsynchronousResponse">

```

## Choosing Another Web Service to Test

You can test another web service without closing the Test Client by clicking the Choose Another WSDL link at the top of the Test Client window. The Test Client will display a page with a box where you enter the WSDL URL, then click Go to display the test form for that web service.





## Launching the Test Client Without the IDE

You can use the Test Client outside the IDE by launching the client through a web browser.

1. With the server running, open a browser window and navigate to the following URL to start the Test Client:

[http://localhost:7001/wls\\_utc](http://localhost:7001/wls_utc)

2. In the **WsdL** box, enter the URL for the WSDL of the web service you want to test and click **Go**.

## Setting Up a Web Service Client for Indirect Testing

Some web services can not be tested standalone with the Test Client. In these cases, you will need to create a separate web service to act as a client of the main web service for the purpose of testing. You will need to test in this "indirect" way if the web service you want to test:

- Contains reliable messaging.
- Contains message-level security.

You can test it by setting up a service control and a client web service for that control. The following gives the basic steps for setting up a service control and control client. Note that you do not need to create a separate web service client for every testing scenario.

To create a new web service project, select **File > New > Project > Web Services > Web Service Project**. You should create all of your client-related classes in a different project than the target web services.

2.

To generate a WSDL file, on the **Package Explorer** tab, right-click the target web service file and select **Web Services > Generate WSDL**. When the WSDL file has been generated, drag and drop it into the new web service project. The WSDL should be dropped into an appropriate package under the `src` directory.

3.

To generate a web service control, right-click the WSDL file and select **Web Services > Generate Service Control**.

4.

To generate a client web service, select **File > New > WebLogic Web Service**. Complete the New Web Service dialog to create a web service class.

Add the web service control to the client by right-clicking anywhere in the source view of the client class and selecting **Insert > Control**. Select the service control generated above.

Add any event handler to the client by right-clicking anywhere in the source view of the client class and selecting **Insert > Control Event Handler**. Select the desired event methods from the service control.

Finally add methods to the client that invoke methods on the service control.

Run the client by right-clicking it in the **Package Explorer** and selecting **Run As > Run on Server**. By default, web services are shown in Test Client.

## Debugging Transactional and Conversational Web Services

When debugging a transactional web service, you should consider increasing the transaction timeout period in order to compensate for delays caused by the debugger. The default timeout is 30 seconds, which may be too short in some debugging situations, especially when the web service is conversational.

To increase the timeout period, use the `timeout` attribute on the `@weblogic.jws.Transactional` annotation:

```
@WebService
@Transactional(value=true, timeout=600) // Increase the timeout period to 600 seconds/10 minutes.
public class TransactionalService implements Serializable {
    ...
}
```

## Related Topics

[Service Control](#)

## WSDL Files: Web Service Descriptions

Files with the WSDL extension contain web service interfaces expressed in the Web Service Description Language (WSDL). WSDL is a standard XML document type specified by the World Wide Web Consortium (W3C, see [www.w3.org](http://www.w3.org) for more information).

WSDL files are used to communicate interface information between web service producers and consumers. A WSDL description allows a client to utilize a web service's capabilities without knowledge of the implementation details of the web service.

### Contents of a WSDL File

A WSDL file contains all of the information necessary for a client to invoke the methods of a web service:

- The data types used as method parameters or return values
- The individual methods names and signatures (WSDL refers to methods as *operations*)
- The protocols and message formats allowed for each method
- The URLs used to access the web service

### Imported WSDL Files

When you want to use an external web service from within Workshop for WebLogic, you should first obtain the WSDL file for the service you want to use. For public web services, the WSDL file will typically be available on the web site of the organization that publishes the web service. For private web services, contact the organization that supports the web service to obtain the WSDL file.

WSDL files can also be found through both public and private UDDI registries. To learn more about UDDI, visit <http://www.uddi.org>.

Once you have the WSDL file, you may use Workshop for WebLogic to create a service control. The service control may then be used from your application like any other Workshop for WebLogic control.

Some web service tools produce WSDL files that do not contain an *XML declaration*. Workshop for WebLogic requires that all XML files contain an XML declaration. The XML declaration is just the first line of an XML file of the following form:

```
<?xml version="1.0" encoding="utf-8" ?>
```

If you receive a WSDL file that does not contain an XML declaration, you must add a declaration to the file using a text editor before you can use the WSDL file in Workshop for WebLogic.

Note that the encoding attribute is not required. If an encoding attribute is not present, the default encoding is utf-8.

## Generating a WSDL From a Web Service Class

When you want to make your web service available to others, you do so by producing a WSDL file for your web service and making it available to your service's clients.

To generate the WSDL file for you web service:

1. On the Package Explorer or Navigator tab, right-click the web service class and select **Web Services > Generate WSDL**.

The generate WSDL can then copied to the client's machine.

## Generating a Service control from a WSDL

If the client is a web service or some other Java component built with Workshop for WebLogic, it can use a service control file generated directly from the WSDL file.

To generate a service control from a WSDL:

- On the Package Explorer or Navigator tab, right-click the WSDL and select **Web Services > Generate Service Control**.

## Generating a Web Service from a WSDL

You can also generate a web service class from a WSDL. The resulting web service class will contain the public endpoint interface described by the WSDL (the public methods and callbacks) without the implementation. After the web service has been generated, the developer must fill in the web service implementation details.

To generate a web service from a WSDL:

- On the Package Explorer or Navigator tab, right-click the WSDL and select **Web Services > Generate Web Service**.

## Related Topics

[Service Control](#)

## W3C WSDL Specification