BEA WebLogic Workshop™ Tutorial

Building Your First Web Service

Tutorial: Your First Web Service

This tutorial provides a tour of the major features of WebLogic Workshop, a visual development environment for building web services. Through the tutorial, you build a web service called Investigate, which is designed to receive client requests for a credit report, perform the required research and calculation, and return the report. The tutorial takes a little over three hours to complete.

Tutorial Goals

Through this tutorial, you will learn how to create and test a web service with WebLogic Workshop. Along the way, you will also learn how to create methods that expose a service's functionality, become acquainted with WebLogic Workshop's support for asynchronous communication and loose coupling, and learn about its controls to speed the design process.

Tutorial Overview

The Investigate web service you will build with this tutorial is designed to collect credit−related information about the applicant, compute a credit worthiness score and rating, then return the combined information to the client.

There are six actors in this scenario:

- The client of your web service. Clients of Investigate are loan or credit application processing systems. An example might be a credit card application processing system at a department store. These systems will use your Investigate web service to determine the applicant's credit worthiness. They will supply you with the applicant's taxpayer ID, and your service will compute and return a credit score.
- Your Investigate web service. The service will receive a taxpayer ID as part of requests from clients and respond with information about the applicant's credit worthiness.
- An database containing bankruptcy information about the applicant.
- A credit card reporting web service with information about the applicant's credit history.
- A credit scoring application designed to calculate a credit score based on information you've collected.
- An Enterprise Java Bean (EJB) designed to provide a credit rating based on the score.

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

Steps in This Tutorial

Step 1: Begin the Investigate Web Service adov−−> 30 minutes

In this step you build and run your first WebLogic Workshop web service. You start WebLogic Workshop and WebLogic Server, then create a web service that has a single method.

Step 2: Add Support for Asynchronous Communication adov−−> 25 minutes

You will work around a problem inherent in communication across the web: the problem of network latency. Network latency refers to the time delays that often occur when transmitting data across a network.

Step 3: Add a Database Control adov−−> 20 minutes

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

Step 4: Add a Service Control adov−−> 15 minutes

You use a ServiceControl to invoke another web service, collecting credit card data on the applicant.

Step 5: Add a JMS and EJB Control adov−−> 20 minutes

You will take the data you have gathered and request help from an in−house application to retrieve a credit score. You will then send the credit score to an Enterprise Java Bean (EJB) that will use it to measure the applicant's credit risk.

Step 6: Add Script for Mapping adov−−> 15 minutes

You will apply an XML map to translate data stored within the Investigate service into a particular XML shape for delivery to the client.

Step 7: Add Support for Cancellation and Exception Handling adov−−> 15 minutes

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

Step 8: Client Application adov−−> 20 minutes

You will build a Java console client to invoke your web service. You also modify your web service to support a polling interface, an alternative to its current design.

Step 9: Deployment and Security adov−−> 10 minutes

You modify your web service for greater security, exposing it through HTTPS (Secure HTTP) instead of HTTP. You will also package the web service for deployment on a production server.

Summary: Your First Web Service adov−−> 15 minutes

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

To begin the tutorial, see Step 1: Begin the Investigate Web Service.

Click the arrow to navigate to the next step.

➡

Step 1: Begin the Investigate Web Service

In this step, you will learn to start WebLogic Workshop and use it to begin a new web service. WebLogic Workshop is the graphical tool you use to create, implement, test, and debug web services. When working in WebLogic Workshop, you use projects to group files associated with a web service or related web services.

With WebLogic Workshop, you can also start and stop WebLogic Server. WebLogic Server provides functionality that supports web services you build with WebLogic Workshop, and so must be running while you are building a web service.

In this tutorial, you will learn how:

- To start WebLogic Workshop on Microsoft Windows

- To start WebLogic Workshop on Linux
- To open the samples project
- To start WebLogic Server
- To create a new web service
- To add a method
- To add a parameter and return value
- To add code for returning a value
- To launch Test View
- To test the requestCreditReport method

Starting WebLogic Workshop

The first step in any process is to start the program. These sections explain how to do that on Microsoft Windows, Linux, and UNIX platforms.

To Start WebLogic Workshop on Microsoft Windows

- From the Start menu, choose Programs−−>WebLogic Platform 7.0−−>WebLogic Workshop.

To Start WebLogic Workshop on Linux

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

```
$HOME/bea/weblogic700/workshop/Workshop.sh
```

3. In the command line, type the following command:

```
sh Workshop.sh
```

When you start WebLogic Workshop for the first time after installing it, it opens to display the samples project. This project contains sample web services included with WebLogic Workshop. After it has been used for the first time, WebLogic Workshop displays the project last opened.

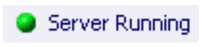
For this tutorial, you will use the samples project.

To Open the Samples Project

1. Choose File−−>Open Project. The Open Project dialog appears.
2. In the Projects on your development server box, select samples.
3. Click Open.

Once you have opened the samples project, you must ensure that WebLogic Server is running while you build your service. All of the web services you create using WebLogic Workshop run on WebLogic Server on your development machine until you deploy them. When your web service is called by other components, WebLogic Server is responsible for invoking it.

You can confirm whether WebLogic Server is running by looking at the indicator in the status bar at the bottom of WebLogic Workshop. If WebLogic Server is running, there is a green ball as pictured here:



If instead you see the a red ball, as in the following picture, then you must start WebLogic Server before proceeding:

To Start WebLogic Server

In WebLogic Workshop, choose **Tools−−>Start WebLogic Server**.

Creating a New Web Service

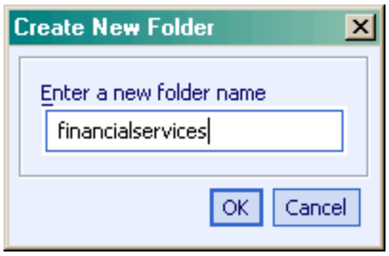
Once WebLogic Workshop and WebLogic Server are running, you can create a new web service. When you create a new web service, you also create a new JWS file. A JWS file is very much like a JAVA file in that it contains code for a Java class. However, because a file with a JWS extension contains the implementation code intended specifically for a web service class, the extension gives it special meaning in the context of WebLogic Server.

To Create a New Web Service

1. In the **Project Tree**, right−click **Project 'samples'** and select New Folder The Create New Folder dialog appears.

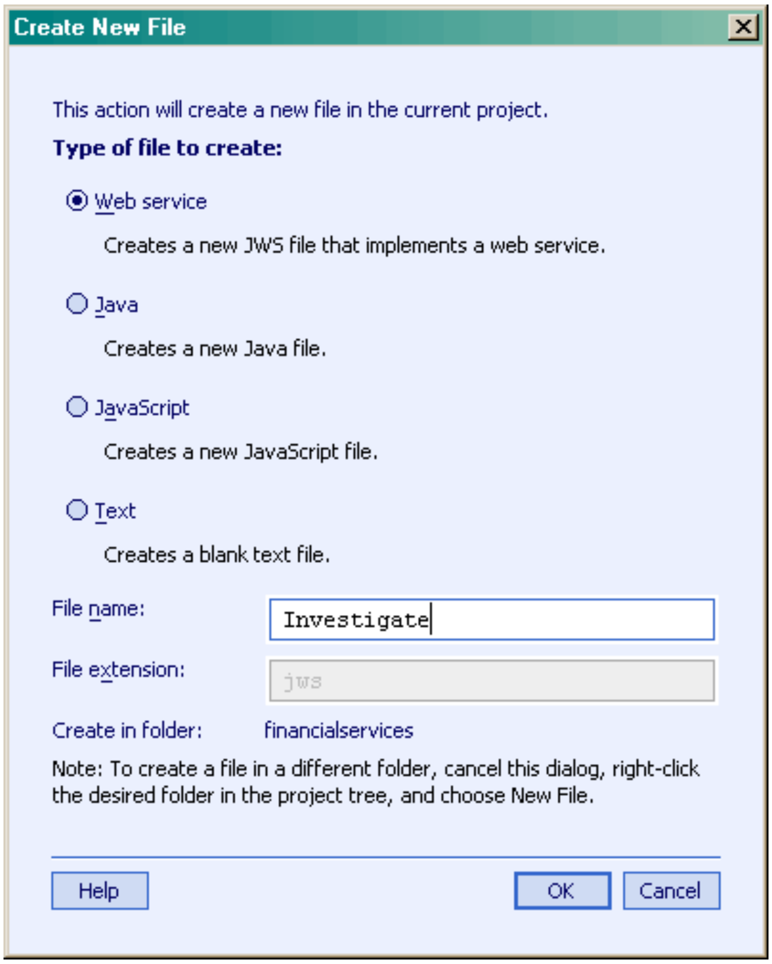
Note:The Project Tree is located at the left side of the screen.

2. In the **Enter a new folder name** box, type financialservices , as shown here:



financialservices

3. Click **OK**. The **financialservices** folder appears in the project tree.
4. Right−click the **financialservices** file and select **New File**. The **Create New File** dialog appears.
5. Confirm that the Web service radio button is selected, as shown in the following picture.


Create New File

This action will create a new file in the current project.

Type of file to create:

● Web service

Creates a new JWS file that implements a web service.

○ Java

Creates a new Java file.

○ JavaScript

Creates a new JavaScript file.

○ Text

Creates a blank text file.

File name: Investigate

File extension: jws

Create in folder: financialservices

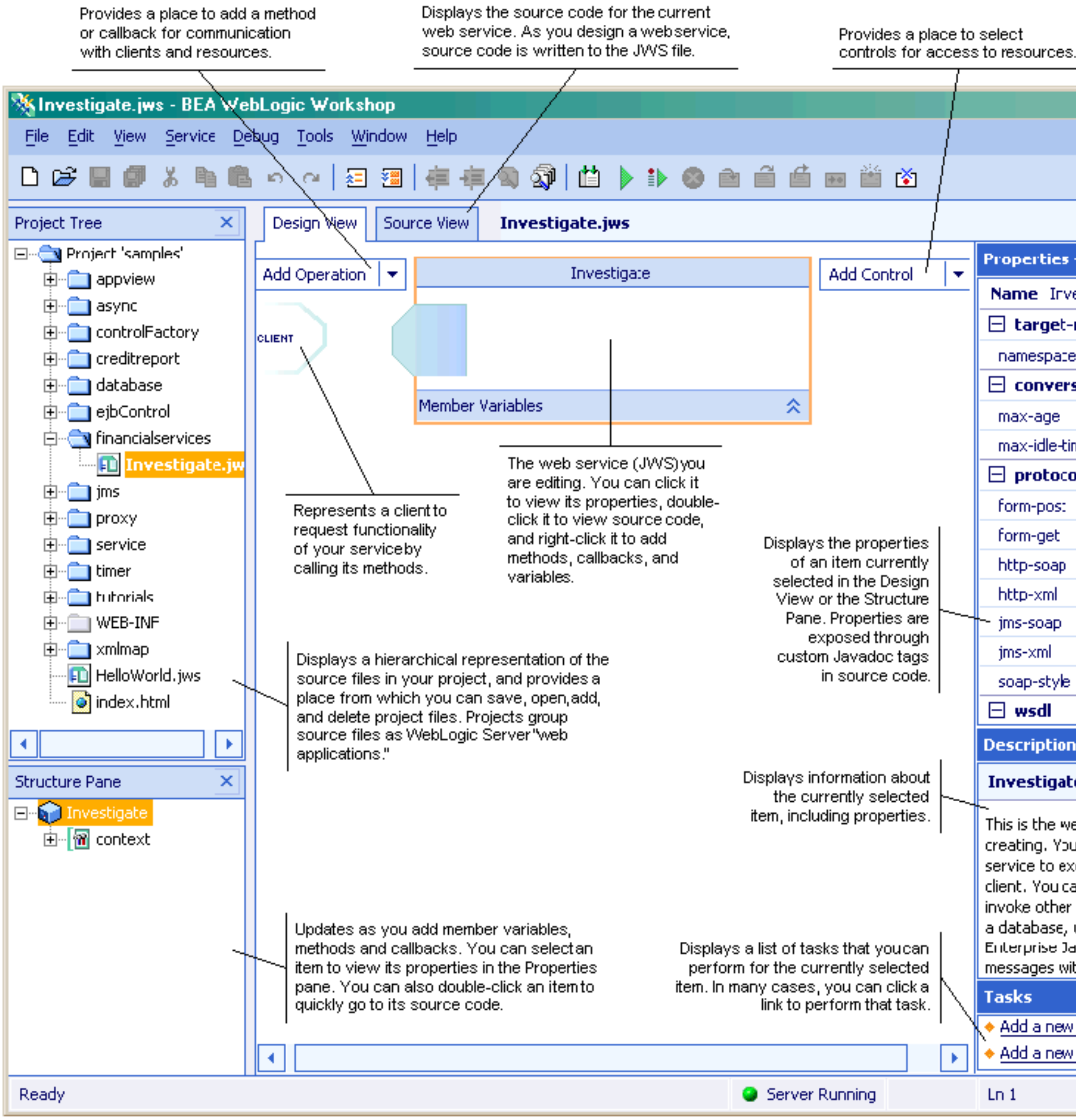
Note: To create a file in a different folder, cancel this dialog, right-click the desired folder in the project tree, and choose New File.

Note: The Java, JavaScript, and Text options provide a way for you to add those kinds of files to your project. For now, however, leave Web service selected.

5. In the File name field, type Investigate. This will be the name of your web service class.

6. Click OK.

Once you create a new service project, WebLogic Workshop displays your new JWS file, entitled Investigate.jws, in Design View. The following screen shot provides a quick tour of your project in Design View.



Provides a place to add a method or callback for communication with clients and resources.

Displays the source code for the current web service. As you design a web service, source code is written to the JWS file.

Provides a place to select controls for access to resources.

Represents a client to request functionality of your service by calling its methods.

The web service (JWS) you are editing. You can click it to view its properties, double-click it to view source code, and right-click it to add methods, callbacks, and variables.

Displays the properties of an item currently selected in the Design View or the Structure Pane. Properties are exposed through custom Javadoc tags in source code.

Displays a hierarchical representation of the source files in your project, and provides a place from which you can save, open, add, and delete project files. Projects group source files as WebLogic Server "web applications."

Displays information about the currently selected item, including properties.

Updates as you add member variables, methods and callbacks. You can select an item to view its properties in the Properties pane. You can also double-click an item to quickly go to its source code.

Displays a list of tasks that you can perform for the currently selected item. In many cases, you can click a link to perform that task.

Graphical Design of Services

In Design View you can design your service in a graphical manner. You create what is effectively a drawing of your service and its interactions with clients and other resources. As you design your service, WebLogic Workshop generates source code that you can edit in the Source View. Through this source code you can add logic behind your design. This enables you to create web services by focusing on application logic rather than on code for infrastructure.

While designing your service, you can add methods and events. You can add controls to represent resources such as other services, databases, and Enterprise Java Beans. You can also specify support for powerful features of the underlying server by setting properties for items in your design.

Note: A good way to get a sense of the design tasks you can perform is to click an item in your design, then view the tasks listed in the **Tasks** pane in the lower right corner of Design View.

One of the key features of WebLogic Workshop is tight integration between tasks you do while designing your web service and the changes you make to source code. For example, if you add a member variable to source code in Source View, your new variable will appear in Design View, and vice versa. This integration is also reflected in the Structure Pane. There you can find a list of items in your service's design, including methods, events, variables, and so on. You do not need to save or refresh the file to cause changes in one view to appear in other views.

For more complete information about projects and the files they contain, see WebLogic Workshop Projects.

Adding the requestCreditReport Method

Web services expose their functionality through methods, which clients invoke to make requests. In this case, clients will use a method you create to request credit reports. The service you are building will eventually collect credit information from other sources. But for now, to keep things simple, you will provide a method that returns a simple report immediately. This web service will provide clients with credit ratings for loan or credit applications. So your web service will need to provide the client with a way to make the initial request. You can do this by adding a requestCreditReport method to the Investigate web service.

To Add a New Method

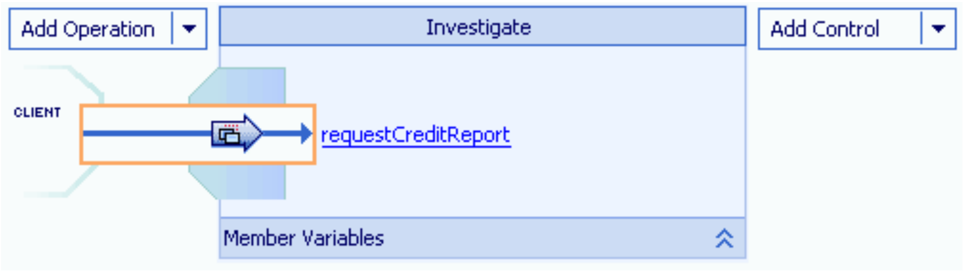
1. If it is not selected already, click the **Design View** tab to ensure that you are viewing the design for the Investigate web service.

Note: It is not necessary to be in Design View while adding methods, but it is helpful to see how changes update the design of your service.

2. From the **Add Operation** drop−down list, select Add Method.
3. In the space provided, replace method1 with requestCreditReport and press Enter.

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

Your design of the Investigate web service should now resemble the following illustration:



Adding a Parameter and Logic

When making a request, the client needs to supply information about the applicant in order for the Investigate file to do its work. For this scenario, the client provides a taxpayer ID. For simplicity, you will enable this method to receive the number as a string. You will also add a return value so that the method can send back to the client the requested results.

To Add a Parameter and Return Value

1. In Design View, double−click the arrow corresponding to the **requestCreditReport** method.

The **Edit Maps and Interface** dialog appears.

2. In the **Java** pane in the lower section of the dialog, edit the text so that it appears as follows:

```
public String requestCreditReport(String taxID)
```

This text is known as the method's declaration. The first occurrence of "String" is the return data type; the text "String taxID" is the parameter data type and parameter name.

3. When you have finished editing the method declaration, click **OK**.

Now you can add a little code to make the method do something. The code you add here will return a string value every time the requestCreditReport method is called. Obviously, this is not very useful, but you will add more intelligent functionality later.

You change a method's functionality by editing its source code.

To Add Code for Returning a Value

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

The following source code appears:

```
/** * @jws:operation */ public String requestCreditReport(String taxID) { }
```

WebLogic Workshop added this source code while you worked in Design View.

2. Edit the requestCreditReport method code so that it returns a String. The returned code resembles the following:

```
/** * @jws:operation */ public String requestCreditReport(String taxID) {    return "applicant: " + taxID + " is approved."; }
```

3. Press Ctrl+S to save your work.

As it is now written, the code for the **requestCreditReport** method returns a string containing the taxpayer ID the client sent with the method call and a note that the applicant was approved. Remember to press Ctrl+S to save your changes to the web service.

Notice that a comment precedes the method that contains an @jws:operation tag. This is a Javadoc comment (Javadoc comments always begin with /** and end with */). Briefly, Javadoc is a simple system for including program structure−related information in comments, indicated by tags, that may be automatically extracted to produce documentation or other material related to structure of a Java class. Its original use was to specify inline documentation that could be extracted to HTML. WebLogic Workshop uses a variety of Javadoc tags to indicate how web service code should be treated by WebLogic Server.

The @jws:operation tag indicates that the method to which it is attached should be exposed as a web service method that clients can call. The term *operation* refers to the actions web services expose to clients, such as the **requestCreditReport** method you have added here. For more information on Javadoc, please refer to the Javadoc pages at java.sun.com.

Launching a Browser to Test the Service

The best way to test a web service while you're writing and debugging code is to use the Test View, a browser−based tool through which you can call methods of your web service. In this tutorial, you will be using two buttons in WebLogic Workshop to control display of the Test View.

To Launch Test View

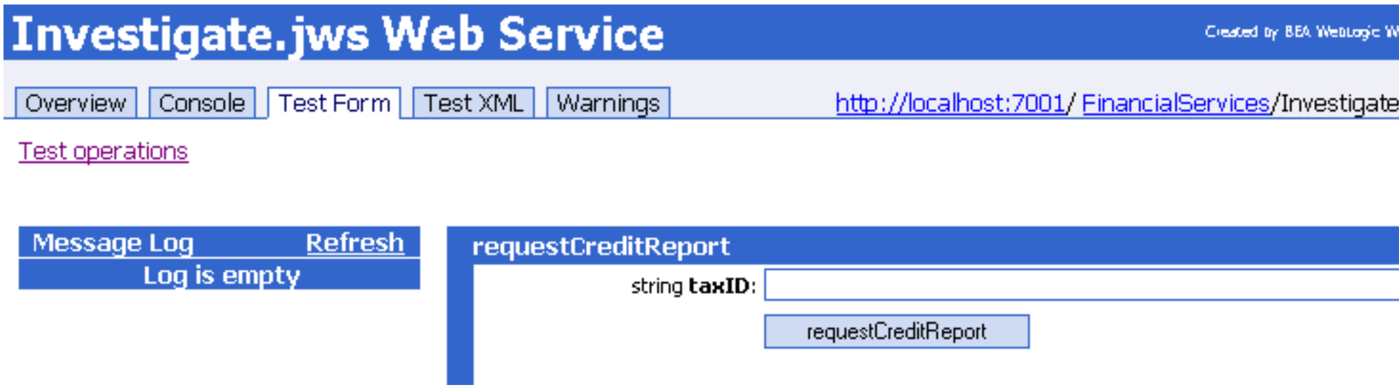
- Click the Start button, which looks like the following illustration:

▶

Note: If you want to stop Test View, click the Stop button shown here:

⊗

Clicking the Start button prompts WebLogic Workshop to build your project, checking for errors along the way. WebLogic Workshop then launches your web browser to display a page through which you can test the service method with possible values. The following illustration shows the Test Form page that appears after you click the Start button.



As you can see, this page provides tabs for access to other pages of the Test View. Test Form and Test XML provide alternative ways to specify the content of the call to your service's methods. Overview and Console include additional information and commands useful for debugging a service. The Warnings tab provides feedback about your service. These tabs are described later in this topic. First, here is a little more information about what you are seeing.

Exploring the Test Form Tab

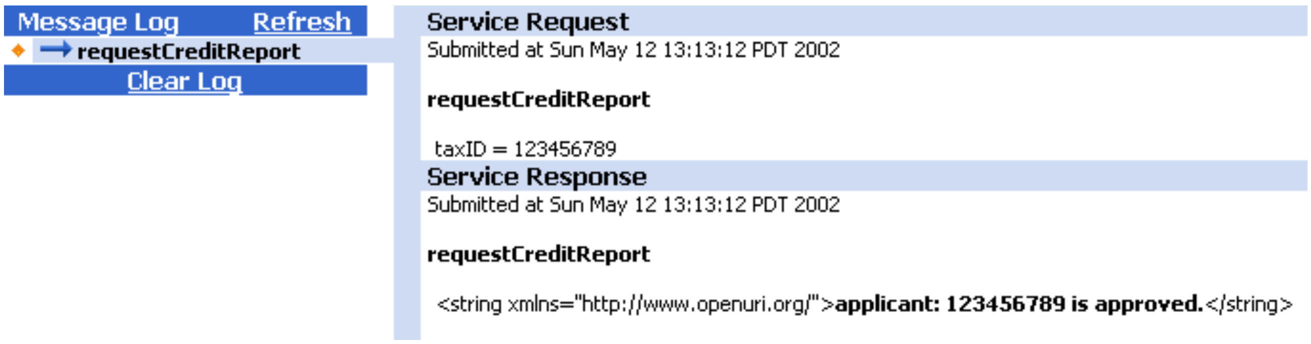
The Test Form tab provides a place for you to test a service with values. You can enter values for the parameters of your methods. For the Investigate service, there is one method with one parameter; as a result, the page provides a box into which you can enter a value for the taxID parameter. When you click the requestCreditReport button, WebLogic Workshop invokes the requestCreditReport method, passing the contents of the text box as a parameter to the method.

To Test the requestCreditReport Method

1. If it is not already selected, click the **Test Form** tab.
2. In the **string taxID** field, enter a value. This value can be any string value, but for the sake of the test it should be something that at least looks like a taxpayer ID, such as 123456789.

3. Click requestCreditReport.

The Test Form page refreshes to display a summary of your request parameters and your service's response, as shown here:



Service Request
Submitted at Sun May 12 13:13:12 PDT 2002

requestCreditReport

taxID = 123456789

Service Response
Submitted at Sun May 12 13:13:12 PDT 2002

requestCreditReport

<string xmlns="http://www.openuri.org/">**applicant: 123456789 is approved.**</string>

Under Service Request, the summary displays the essence of what was sent by the client (you) when the method was called. It shows the parameter values passed with the method call, as in the following example:

taxID = 123456789

Under Service Response, the summary displays what was returned by the Investigate service. The response is formatted as a fragment of Extensible Markup Language (XML), as communications with web services are formatted as XML messages. You can see that the method returned the correct result for the code you provided, as shown here:

```
<string xmlns="http://www.openuri.org/">applicant: 123456789 is approved.</string>
```

To the left of the request and response is the Message Log area. This area lists a separate entry for each test call to the service method. Entries in the Message Log correspond to your service's methods, updating with a new entry for each test you make.

To try this out, click the Test operations link to return to the original Test Form page, then enter a new value in the string taxID box and click requestCreditReport. When the page refreshes, request and response data corresponding to your second test is displayed as a second entry in the Message Log. You can click each log entry to view the data for the corresponding test. Click Clear Log to empty the list of log entries and start fresh.

That's it for the test! With this simple test, you have pretty well exercised this service's abilities adov−−> so far.

But before you move on, take a look at the other information available from Test View. For example, the URL at the upper right corner of the page next to the Warnings tab should appear something like this:

http://localhost:7001/samples/financialservices/Investigate.jws

This is the URL used to call the web service. Each portion of the URL has a specific meaning. The following list explains each portion:

- http://locahost:7001/ adov−−> This means that your browser's requestadov−−>that is, the call to the serviceadov−−>will be intercepted by WebLogic Server, which is listening on port 7001 of your local machine. In your case, the term "localhost" is probably replaced by the name of your computer.
- samples/financialservices/ adov−−> "samples" refers to the web application of which the service is a part. When you create a project in WebLogic Workshop, you are also creating a WebLogic Server "web application." The project name becomes part of the URL for all web services in that project. You will want to keep that in mind when naming new projects so that the resulting web service URLs are meaningful and appropriate.

- Investigate.jws adov−−> This is the name of the web service's JWS file. WebLogic Server is configured to recognize the JWS extension and respond appropriately by serving the request as a web service rather than, for example, an HTML page or a Java servlet.

Exploring the Overview Page

The Overview page deserves special attention because it provides access to all of the files you and your service's clients will need to use your web service. As you update your service's design and source code, you will be able to easily get current versions of these files. As a result, you may find the Overview page to be an invaluable resource as you begin testing and using it in a deployed application.

The following list describes some of the prominent features you will find on the Overview page:

Web Service Description Language files adov−−> Web Service Definition Language (WSDL) files describe your service's interface, giving potential clients what they need to know in order to use your service's functionality.

Web Service Clients adov−−> The following list provides brief descriptions of these links. They will be covered in more detail later in this tutorial.

- The Workshop Control link displays the contents of a CTRL file. A CTRL (or "control") file provides your service's interface in a format specific to WebLogic Workshop.
- The Java Proxy link downloads a JAR file containing code a Java client can use to call the Investigate service.
- The Proxy Support Jar link downloads a JAR file containing supporting classes for clients written in Java.

Service Description adov−−> A list of the methods and callbacks (if any) that are exposed by the web service. This is for verification purposes. If you don't see all of the methods you expected in this list, it indicated a problem with the source code for the web service. Here, you can see that the Investigate web service currently has only the requestCreditReport method and no callbacks.

Useful Links adov−−> A link to a topic on building client proxies, as well as links to the specifications behind the standards for the following languages:

- WSDL, a language (based on the rules of Extensible Markup Language, or XML) for describing a service's interface.
- Simple Object Access Protocol (SOAP), a protocol for formatting the XML−based messages that web services use to communicate.
- XML namespaces, a means to prevent conflicts among XML element and attribute names (but ensuring that names are unique within a given scope of use).

- URIs, strings used to identify a resource, including XML namespaces and schemas.

Exploring the Console Page

The Console page provides information and commands related to the instance of WebLogic Server running on your computer. Through the settings and buttons on this page, you can control aspects of your local instance of WebLogic Server while you are testing your service.

Exploring the Test XML Page

When your web service is deployed and receiving requests from clients, those requests will come in the form of XML messages carrying request−specific information. The Test XML page provides a place for you to test your service with something like the body of an XML message that might be sent to your service. You can try it out now by replacing the placeholder value with the Taxpayer ID you tested with earlier, then clicking requestCreditReport.

```
<requestCreditReport xmlns="http://www.openuri.org/">
  <taxID>
    111111111
  </taxID>
</requestCreditReport>
```

You have successfully added code to a web service’s method and tested the method for correct operation. In the next step, Step 2: Add Support for Asynchronous Communication, you will revise your service's design to solve a problem inherent in communications across the Internet adov−−> latency.

Related Topics

WebLogic Workshop Projects

Design View

Test View

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

Step 2: Add Support for Asynchronous Communication

In this step, you will solve a problem inherent in communication across the web: the problem of network latency. Network latency refers to the time delays that often occur when transmitting data across a network.

This can be a particular problem on the internet, which has highly unpredictable performance characteristics and uncertain reliability. There may be additional latency if it takes a long time for a web service to process a request (for example, if a person "behind" the service must review the applicant's credit before a response can be returned).

So far, the design of the Investigate web service forces the client calling the requestCreditReport method to halt its own processes and wait for the response from the web service. This is called a synchronous relationship, in which the client software invokes a method of the web service and is blocked from continuing its own processes until it receives the return value from the web service. As you might imagine, this won't work well when it is difficult to predict the time it will take for the client to receive a response.

You will solve the latency problem by enabling your web service to communicate with its clients asynchronously. In asynchronous communication, the client communicates with the web service in such a way that it is not forced to halt its processes while the web service produces a response. In particular, you will add a method to the web service that immediately returns a simple acknowledgement to the client, thereby allowing the client to continue its own processes. You will also add a callback that returns the full results to the client at a later time. Finally, you will implement support for conversations that enable your web service to remember which client to send the full response to via the callback.

The tasks in this step are:

- To add a class and a member variable to the web service

- To add a requestCreditReportAsynch method to the web service
- To add an onCreditReportDone callback to the web service
- To add code that sends data back to the client
- To add a buffer and conversation support to the web service
- To test the web service

To Add a Class and Member Variable to the Web Service

The code you add below has two parts: a class, called Applicant, and a member variable, called m_currentApplicant. This variable is an instance of the Applicant class, which serves as a data structure to record the information about credit applicants. Its fields hold information about an applicant's name, currently available credit, and so on. While the member variable m_currentApplicant stores data about a particular applicant, the web service builds a profile about that particular applicant.

1. Click the **Source View** tab. From this tab you can view the web service's Java code.
2. Place the following code within public class Investigate:

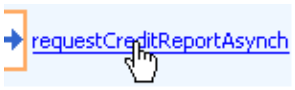
```
public static class Applicant implements java.io.Serializable
{
    public String taxID;
    public String firstName;
    public String lastName;
    public boolean currentlyBankrupt;
    public int availableCCCredit;
    public int creditScore;
    public String approvalLevel;
    public Applicant(String taxID)
    {
        this.taxID = taxID;
    }
    public Applicant() {}
}
Applicant m_currentApplicant = new Applicant();
```

To Add a requestCreditReportAsynch Method to the Web Service

Next you will add a method to be invoked by client software applications that receives a report on an applicant's credit worthiness. The method itself simply returns an acknowledgement to the invoking client without returning any substantial information on the applicant. The finished report on the applicant's credit worthiness is returned in the onCreditReportDone callback, which you will add later.

1. Click the Design View tab.
2. From **Add Operation** drop−down list, select **Add Method**.

3. In the input box that appears, type the method name requestCreditReportAsynch and press Enter.
4. Click the method name, as shown here:



The method code appears in Source View.

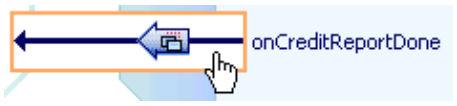
5. Edit the requestCreditReportAsynch method code so that it appears as follows:

```
/** * @jws:operation */ public void requestCreditReportAsynch(String taxID)    {
       m_currentApplicant.taxID = taxID;    }
```

To Add an onCreditReportDone Callback to the Web Service

Once the web service is finished building a credit profile of a credit applicant, you want to add a callback to send a credit report back to a client application. The callback you add below will return the finished report on an applicant's credit worthiness.

1. Click the Design View tab to return to Design View.
2. From the Add Operation drop−down list, select Add Callback.
3. In the input field that appears, type the callback name onCreditReportDone and press Enter.
4. Double−click the arrow associated with the onCreditReportDone callback. The Edit Maps and Interface dialog box appears.



5. In the Java pane, edit the text so that it appears as follows:

```
public void onCreditReportDone(Applicant m_currentApplicant, String responseMessage)
```

6. Click OK. The Edit Maps and Interface dialog closes.

To Add Code that Sends Data Back to the Client

Next, you can edit the requestCreditReportAsynch method so that it initiates the onCreditReportDone callback .

1. If you are not in Source View, click the Source View tab.
2. Modify the requestCreditReportAsynch method to look like the following:

```
/** * @jws:operation */ public void requestCreditReportAsynch(String taxID) {
       m_currentApplicant.taxID = taxID;        callback.onCreditReportDone(m_currentApplicant, null); }
```

To Add a Buffer and Conversation Support to the Web Service

Now you are ready to add a buffer and conversation support to your web service.

The buffer, which you will add to the requestCreditReportAsynch method, serves two purposes. First, it saves client requests in a message queue which prevents requests from being lost during server outages. Second, it immediately returns an acknowledgement to the requesting client, which allows the client to continue processing without waiting for the full response from the web service.

Adding conversation support to your web service lets the client know for sure that the result your service sends back through the onCreditReportDone callback is associated with the request the client originally made. If the same client makes two independent calls with the same taxpayer ID (say, for separate loan applications from the same applicant), how will it know which returned result corresponds to which request? With all the interaction back and forth between the client and your web service, your service needs a way to keep things straight.

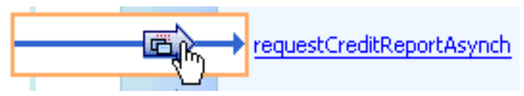
Also, if the server goes down (taking the m_applicantID member variable with it), the data that the client sent as a parameter is lost. The service not only loses track of the client's request, but of who the client was in the first place.

You can solve these problems by associating each exchange of information (request, response, and any interaction needed in between) with a unique identifier—an identifier known to both the client and the service.

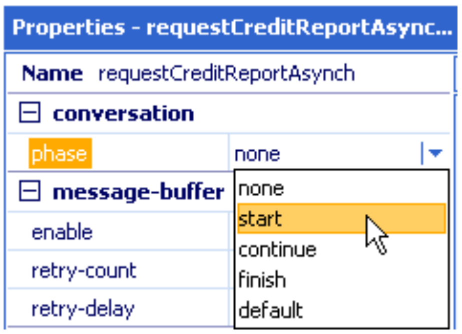
In web services built with WebLogic Workshop, you do this by adding support for a conversation. For services participating in a conversation, WebLogic Server stores state−related data, such as the member variable you added, on the computer's hard drive and generates a unique identifier to keep track of the data and of which responses belong with which requests. This infrastructure remains in place as long as the conversation is ongoing. When the conversation is finished, the resources allocated to the infrastructure are released.

When you add methods to a web service that supports conversations, you can indicate whether each method starts, continues or finishes a conversation. Adding support for a conversation is very easy. With methods that represent both the first step of the transaction (requestCreditReportAsync) and the last step (onCreditReportdone), you want to indicate when the conversation starts and when it finishes. To add a buffer and conversation support to your web service follow the steps below.

1. Click the Design View tab to return to Design View.
2. Edit the Properties Pane associated with the requestCreditReportAsynch method. To edit the Propeties Pane for this method click the arrow next to the method name requestCreditReportAsynch. Note: do not click the method name requestCreditReportAsynch, instead click its associated arrow, as shown here:

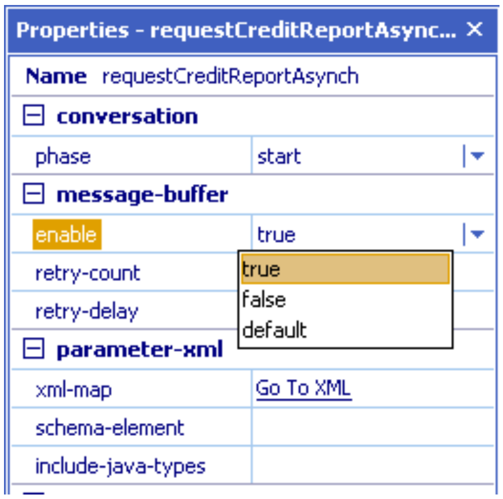


3. In the Properties pane, from the phase drop−down list, select start, as shown here:


Properties - requestCreditReportAsync...

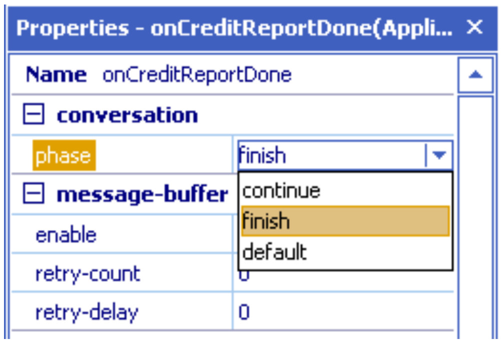
Name	requestCreditReportAsynch
⊟ conversation	
phase	none
⊟ message-buffer	
enable	
retry-count	
retry-delay	

4. In the message−buffer section, from the enable drop−down list, select true, as shown here:


Properties - requestCreditReportAsync...

Name	requestCreditReportAsynch
conversation	
phase	start
message-buffer	
enable	true
retry-count	
retry-delay	
parameter-xml	
xml-map	Go To XML
schema-element	
include-java-types	

5. Click the arrow next to the onCreditReportDone callback.
6. In the Properties pane, from the phase drop−down list, select finish, as shown here:


Properties - onCreditReportDone(Appli...

Name	onCreditReportDone
conversation	
phase	finish
message-buffer	
enable	
retry-count	0
retry-delay	0

7. Press Ctrl+S to save the service.

To Test the Web Service

Now you are ready to compile and test the service.

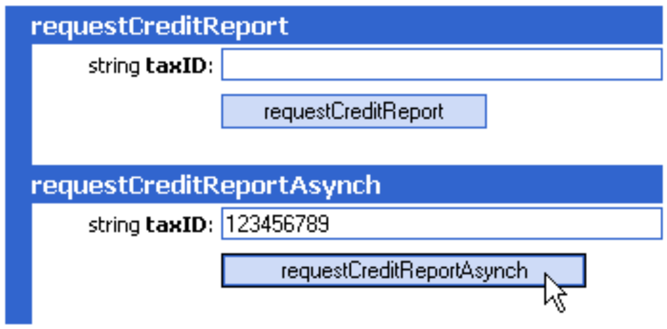
1. Compile and test the web service by clicking the Start button, shown here:

Your web service compiles and a browser window is launched that displays the Test page.

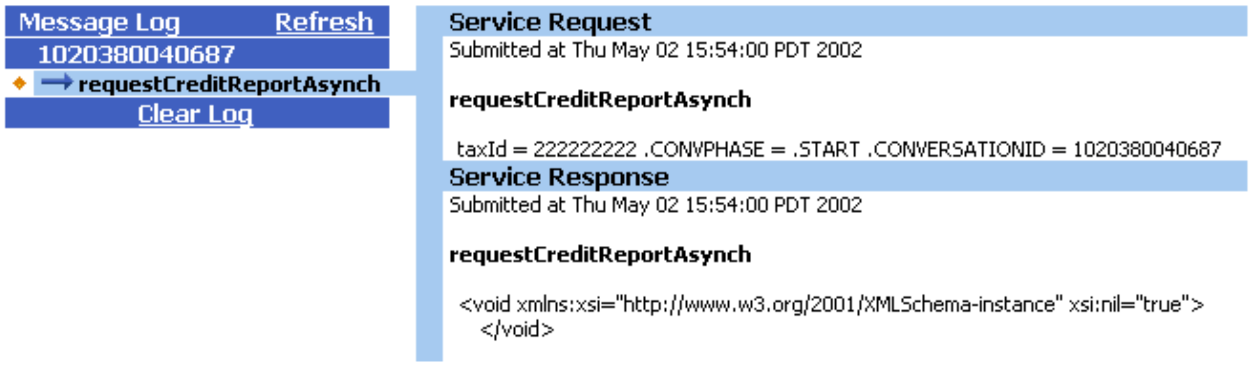
2. Type a number value in the taxID field, as shown below.

Note: Use one of the following (9 digit) taxID's to test your web service throughout the tutorial:

123456789, 111111111, 222222222, 333333333, 444444444, and 555555555.



3. Click requestCreditReportAsynch. The Test page refreshes to display a summary of the request parameters sent by the client and your service's response, as shown here:



Service Request

Submitted at Thu May 02 15:54:00 PDT 2002

requestCreditReportAsynch

```
taxId = 222222222 .CONVPHASE = .START .CONVERSATIONID = 1020380040687
```

Service Response

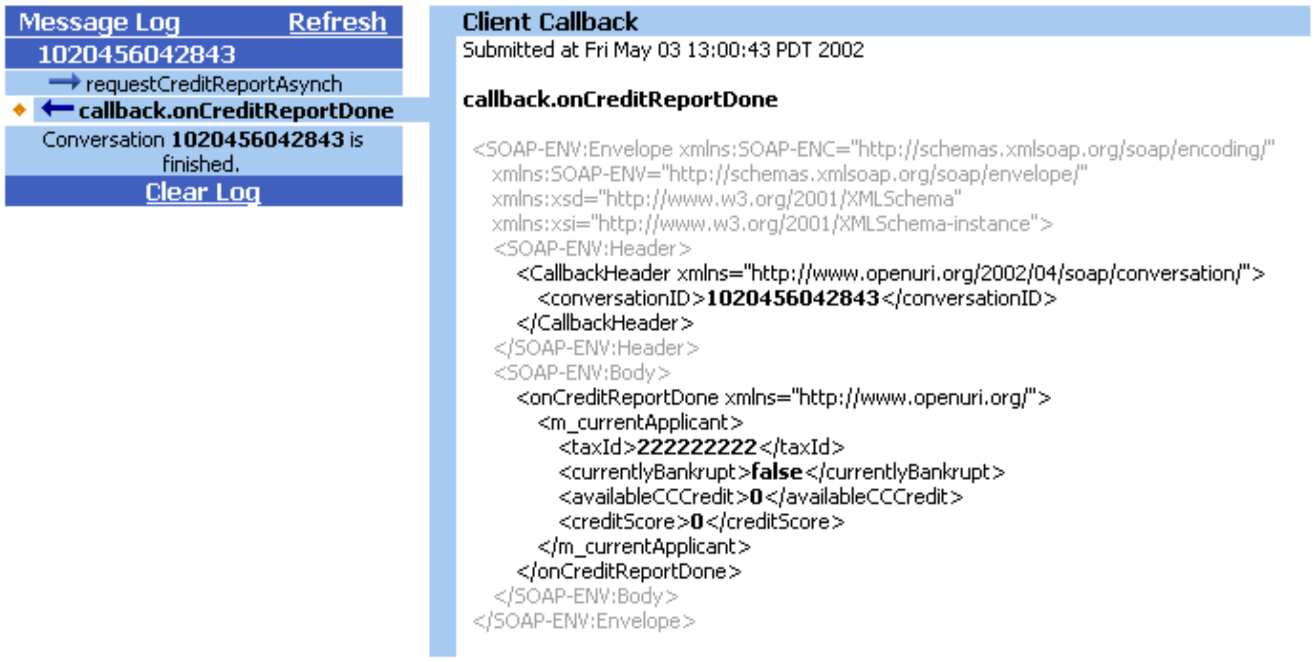
Submitted at Thu May 02 15:54:00 PDT 2002

requestCreditReportAsynch

```
<void xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true">
  </void>
```

Note: Under Service Request, the summary displays what the client sent to the web service to invoke the requestCreditReportAsynch method. Notice that is displays the taxID which you sent to the web service as well as the conversation identification number, which the service uses to associate the initial client request and the callback response it will later send back to the client.

4. Under Service Response, the summary displays what the web service has sent back (so far) to the client. In this case it sends back an XML file serving as an acknowledgement to the client that its request has been received.
5. Refresh the browser. Notice that the Message Log has the new entry callback.onCreditReportDone.
6. Click callback.onCreditReportDone to view the contents of the callback sent from the web service to the client, as shown here:



Conversation 1020456042843 is finished.

Client Callback

Submitted at Fri May 03 13:00:43 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1020456042843</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <taxId>222222222</taxId>
        <currentlyBankrupt>false</currentlyBankrupt>
        <availableCCCredit>0</availableCCCredit>
        <creditScore>0</creditScore>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The contents display the SOAP message that the client receives, including data about the current applicant. Of course, since our web service has no way of learning anything about a credit applicant, the information send back to the client consists of the taxID originally entered, along with default values for the other fields.

In the next step of the tutorial you will add a database control to your web service, which will enable your web service to acquire substantive data about a credit applicant.

Related Topics

User Interface Reference

Overview: Conversations

Step 3: Add a Database Control

In this step you add a database control to your web service. The database control provides your web service access to a database containing bankruptcy information about credit applicants. Controls, i.e., CTRL files, act as interfaces between your web service and other data resources, such as databases, other web services, Java Message Services, etc.

The tasks in this step are:

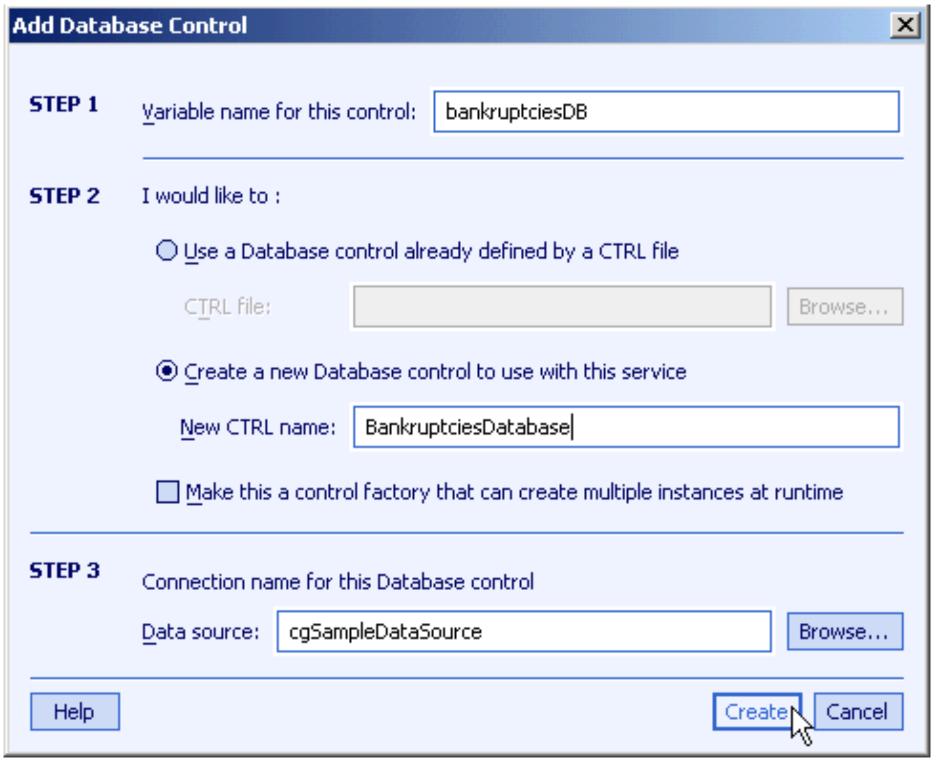
- To create a database CTRL file

- To edit the web service code to incorporate the CTRL file
- To test the web service using the debugger

To Create a Database CTRL file

In this task you will create a database CTRL file and then add a method to this CTRL file. The method you add, called checkForBankruptcies, will query a database using a SQL query.

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list, select Add a Database Control. The Add Database Control dialog appears.
3. Enter values as shown in the following illustration:


Add Database Control

STEP 1 Variable name for this control: bankruptciesDB

STEP 2 I would like to :

○ Use a Database control already defined by a CTRL file

CTRL file:

● Create a new Database control to use with this service

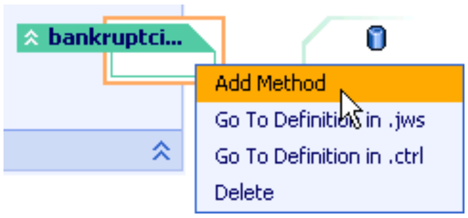
New CTRL name: BankruptciesDatabase

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

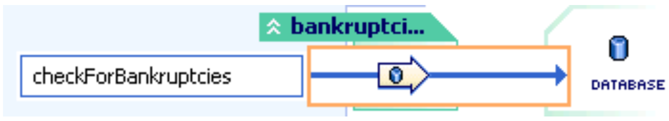
STEP 3 Connection name for this Database control

Data source: cgSampleDataSource

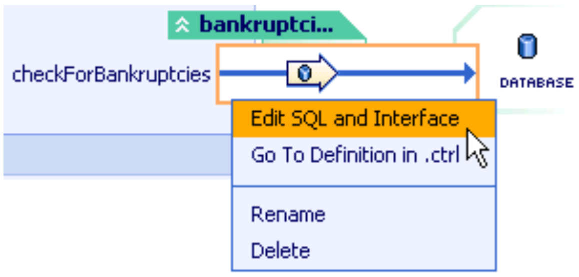
4. Click Create.
5. Right−click the newly created database control and select Add Method.



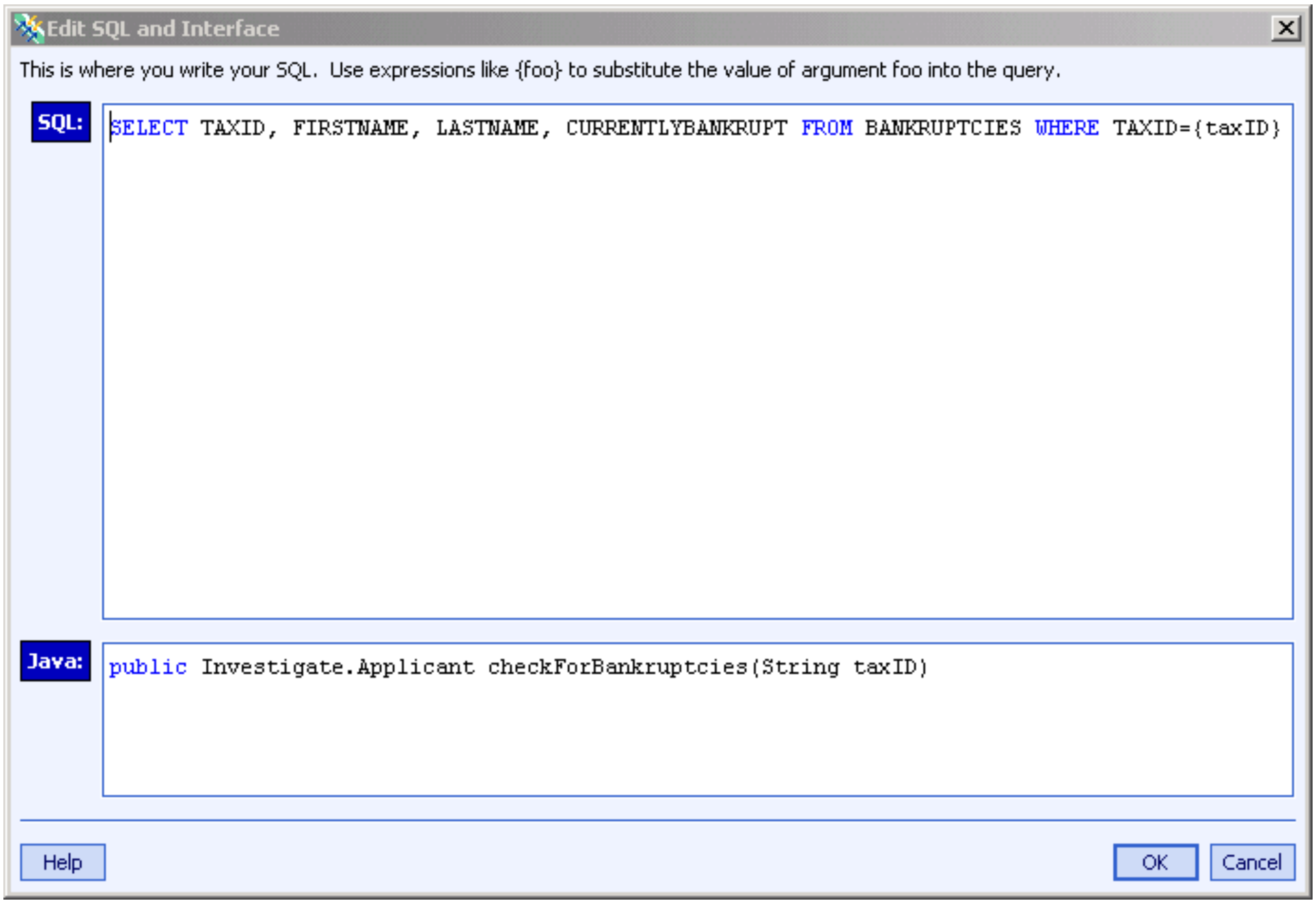
6. In the field that appears, type checkForBankruptcies and press Enter.



7. In Design View, right−click the arrow associated with the checkForBankruptcies method and select Edit SQL and Interface, as shown here. The EditSQL and Interface dialog appears.



8. Enter values as shown in the illustration below and click OK.
   Use the selectable text to cut and paste in the EditSQL and Interface dialog.
   For the SQL widow:
   SELECT TAXID, FIRSTNAME, LASTNAME, CURRENTLYBANKRUPT FROM BANKRUPTCIES WHERE TAXID={taxID}
   For the Java window:
   public Investigate.Applicant checkForBankruptcies(String taxID)



This is where you write your SQL. Use expressions like {foo} to substitute the value of argument foo into the query.

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

Java:
```
public Investigate.Applicant checkForBankruptcies(String taxID)
```

To Edit the Web Service Code to Incorporate the CTRL File

Next you must modify the web service's code to take advantage of the database control that has been added. You will edit the method requestCreditReportAsynch to invoke the control method checkForBankruptcies. Any information found in the database is stored in the member variable m_currentApplicant.

1. Click the Source View tab.
2. Edit the requestCreditReportAsynch method to look like the following:

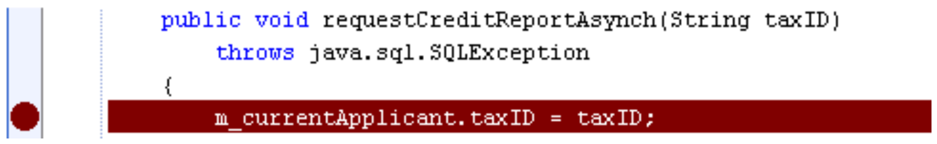
```
public void requestCreditReportAsynch(String taxID)
    throws java.sql.SQLException
{
    m_currentApplicant.taxID = taxID;
    m_currentApplicant = bankruptciesDB.checkForBankruptcies(taxID);
    callback.onCreditReportDone(m_currentApplicant, null);
}
```

To Test the Web Service Using the Debugger

Next you will test your web service using the debugger. The debugger allows you set breakpoints in your code and track how your code is running line−by−line. Setting a breakpoint in your code will cause the Java Virtual Machine to halt execution of your code immediately before the breakpoint, allowing you step through your code beginning at that point.

1. If you are not in Source View, double−click Investigate to view the source code for Investigate.jws.
2. Place the cursor on the first line of code executed within the method requestCreditReportAsynch.
3. Click the Toggle Breakpoint button on the toolbar, as shown here:

The breakpoint appears on the first line of code, as shown here:

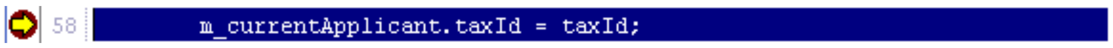


```
    public void requestCreditReportAsynch(String taxID)
        throws java.sql.SQLException
    {
        m_currentApplicant.taxID = taxID;
```

4. Press the Start and Debug button on the toolbar, shown here:

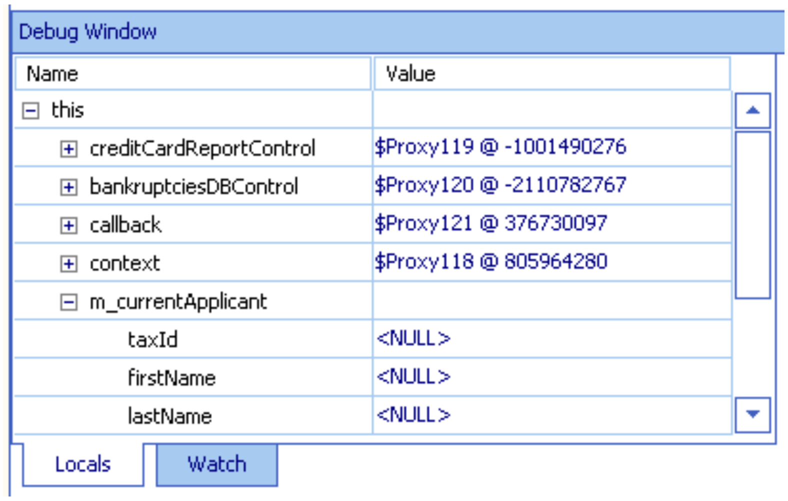
The Test View page appears.

5. In the taxID box, type the nine−digit number 222222222 and click requestCreditReportAsynch. Note: The database you just added to your web service contains data on 6 individuals. The taxIDs of these individuals are 123456789, 111111111, 222222222, 333333333, 444444444, and 555555555. Use these six taxIDs to test your web service throughout the tutorial.
6. The web service does not run through its routine of method calls and callbacks. To verify this, refresh the browser and note that the service has halted at the requestCreditReportAsynch.
7. Return to Workshop and note that the execution of code has halted at the breakpoint shown here:



```
        m_currentApplicant.taxId = taxId;
```

8. On the Locals tab of the Debug Window pane, expand the entries for this and m_currentApplicant, as shown here:



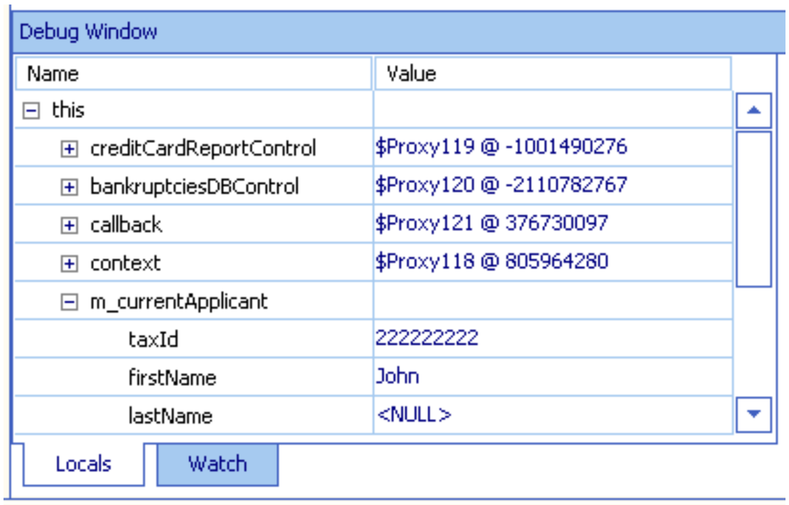
Debug Window

Name	Value
⊟ this	
⊞ creditCardReportControl	$Proxy119 @ -1001490276
⊞ bankruptciesDBControl	$Proxy120 @ -2110782767
⊞ callback	$Proxy121 @ 376730097
⊞ context	$Proxy118 @ 805964280
⊟ m_currentApplicant	
taxId	<NULL>
firstName	<NULL>
lastName	<NULL>

Note: The Debug Window pane gives you information about the current values of the variables within your web service, as well as current position in the call stack.

9. Click the Step Into button on the toolbar, shown here:

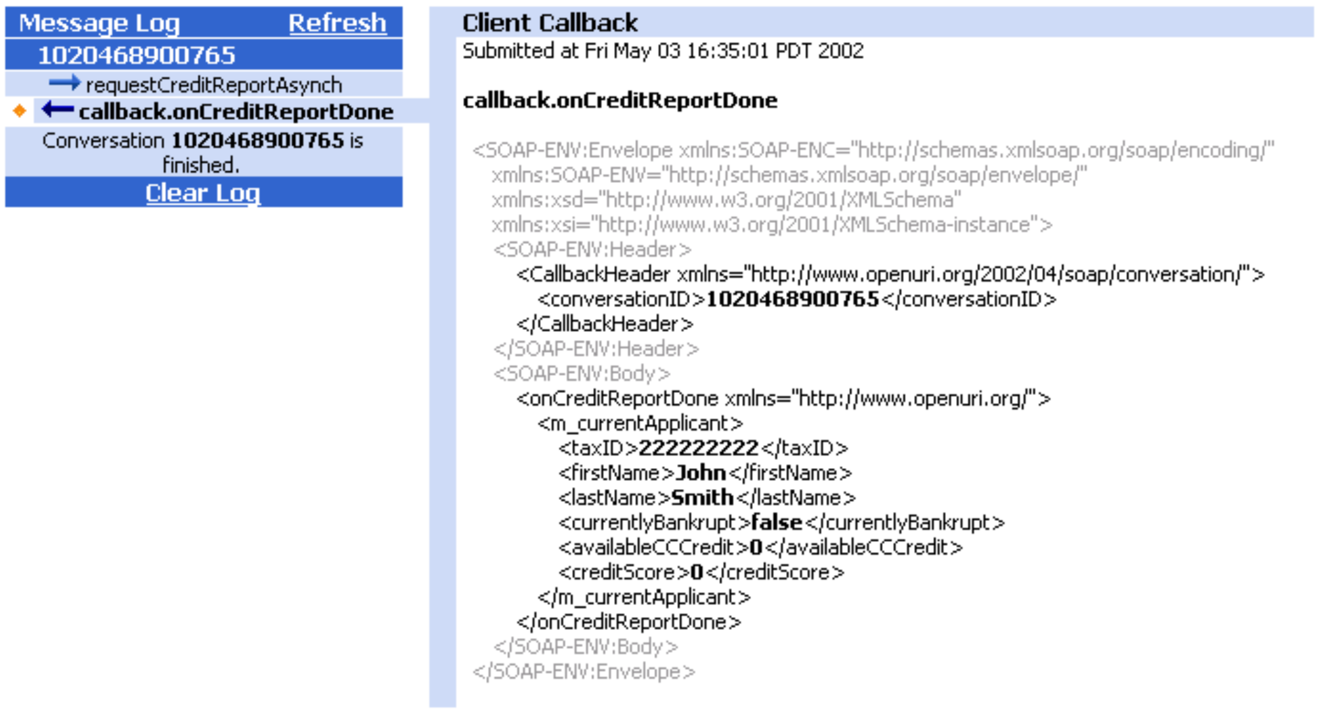
Each time you press the Step Into button, a new line of code within the requestCreditReportAsynch method is executed. The properties of the m_currentApplicant are filled with values as they are retrieved from the database, as shown here:



Debug Window

Name	Value
⊟ this	
⊞ creditCardReportControl	$Proxy119 @ -1001490276
⊞ bankruptciesDBControl	$Proxy120 @ -2110782767
⊞ callback	$Proxy121 @ 376730097
⊞ context	$Proxy118 @ 805964280
⊟ m_currentApplicant	
taxId	222222222
firstName	John
lastName	<NULL>

10. Continue clicking the Step Into button until the web service finishes executing.
11. Return to the browser window that displays the test page and refresh the browser.
12. Under Message Log, click callback.onCreditRepotDone. The response sent back to the client application appears, as shown here:


Message Log
Refresh
1020468900765
requestCreditReportAsynch
callback.onCreditReportDone
Conversation 1020468900765 is finished.
Clear Log

Client Callback

Submitted at Fri May 03 16:35:01 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1020468900765</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <taxID>222222222</taxID>
        <firstName>John</firstName>
        <lastName>Smith</lastName>
        <currentlyBankrupt>false</currentlyBankrupt>
        <availableCCCredit>0</availableCCCredit>
        <creditScore>0</creditScore>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You have completed adding a database control and testing your web service with the debugger. In the next step of the tutorial you will add a service control to your web service.

Related Topics

Debugging Web Services

Database Control: Using a Database from Your Web Service

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

Step 4: Add a Service Control

Below we will add a service control to your web service. In other words, you will add a CTRL file which allows your web service to invoke another external web service. This other web service provides credit card data on a credit applicant.

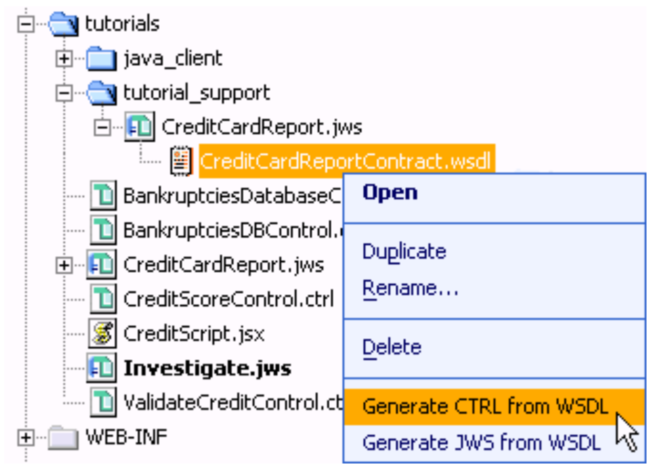
The tasks in this step are:

- To generate a CTRL file from a WSDL file
- To add the CTRL file to your web service
- To add code to invoke an external web service
- To add code to handle callbacks from the external web service
- To test the web service

To Generate a CTRL File from a WSDL File

By definition, each web service provides a WSDL file that explains how the web service is operated. In this task you will automatically generate a CTRL file based the WSDL file of another web service. The resulting CTRL file is incorporated into your web service to form the interface between your web service and the external web service.

1. In the Project tree, expand the tutorials folder, then expand the tutorial_support folder, finally expand CreditCardReport.jws as shown below.
2. Right−click the CreditCardReportContract.wsdl file and select Generate CTRL from WSDL, as shown here:

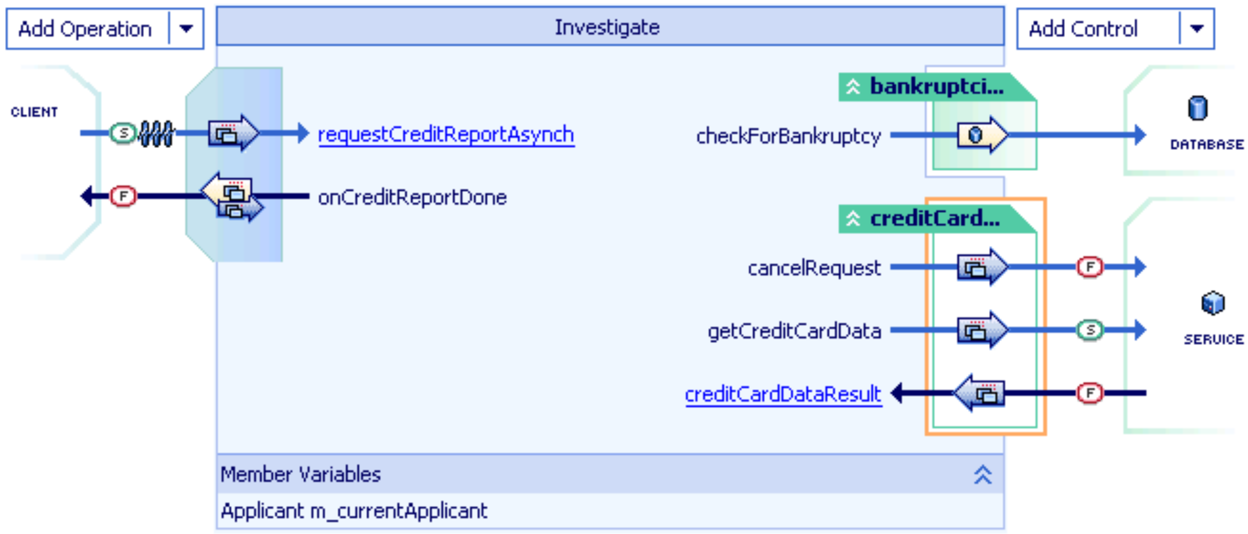


Note that a new CTRL file is generated and placed underneath CreditCardReportContract.wsdl.

To Add the CTRL File to Your Web Service

In this task you will add the new CTRL file to your web service. With the CTRL file added you will be able to invoke the external web service that provides credit card data.

1. If you are not in Design View, click the Design View tab.
2. Drag and drop the newly generated CreditCardReportContract.ctrl from the Project Tree into Design View. Your screen should now look like the one shown here:



A new service control, containing two methods and a callback, is added to your web service.

To Add Code to Invoke the External Web Service

In this task you will add code that invokes the external web service.

1. Click the Source View tab.

2. Edit the requestCreditReportAsynch method to look like the code shown here. Make sure that you delete the line of code shown crossed out below.

```
public void requestCreditReportAsynch(String taxID)
    throws java.sql.SQLException
{
    m_currentApplicant.taxID = taxID;
    m_currentApplicant = bankruptciesDB.checkForBankruptcies(taxID);
    creditCardReportControl.getCreditCardData(taxID);
    callback.onCreditReportDone(m_currentApplicant, null);
}
```

To Add Code to Handle Callbacks from the External Web Service

In this task you will add code to handle the credit card information sent back from the external web service. This code does two things: first, it stores the relevant credit card data in the member variable m_currentApplicant. Second, it sends the applicant profile back to the client using the callback onCreditReportDone.

1. If you are not in Design View, click the Design View tab.
2. Click creditCardDataResult.
3. Edit the callback handler to look like the following:

```
private void creditCardReportControl_creditCardDataResult(CreditCard[] cards) {    for(int i = 0; i <
cards.length; i++)  {        m_currentApplicant.availableCCCredit += cards[i].availableCredit;    }
   callback.onCreditReportDone(m_currentApplicant, null); }
```

4. Press Ctrl+S to save the file.

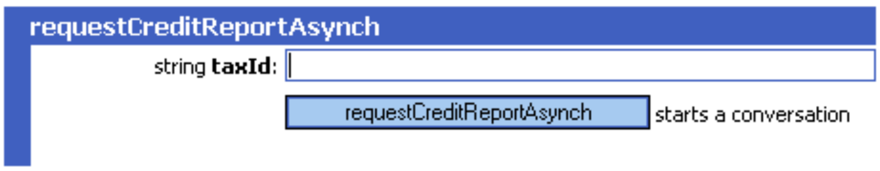
To Test the Web Service

You are now ready to test the new features of your web service.

1. Click the Start button, shown here:

WebLogic Workshop tests and compiles the web service. A browser is launched displaying Test View.

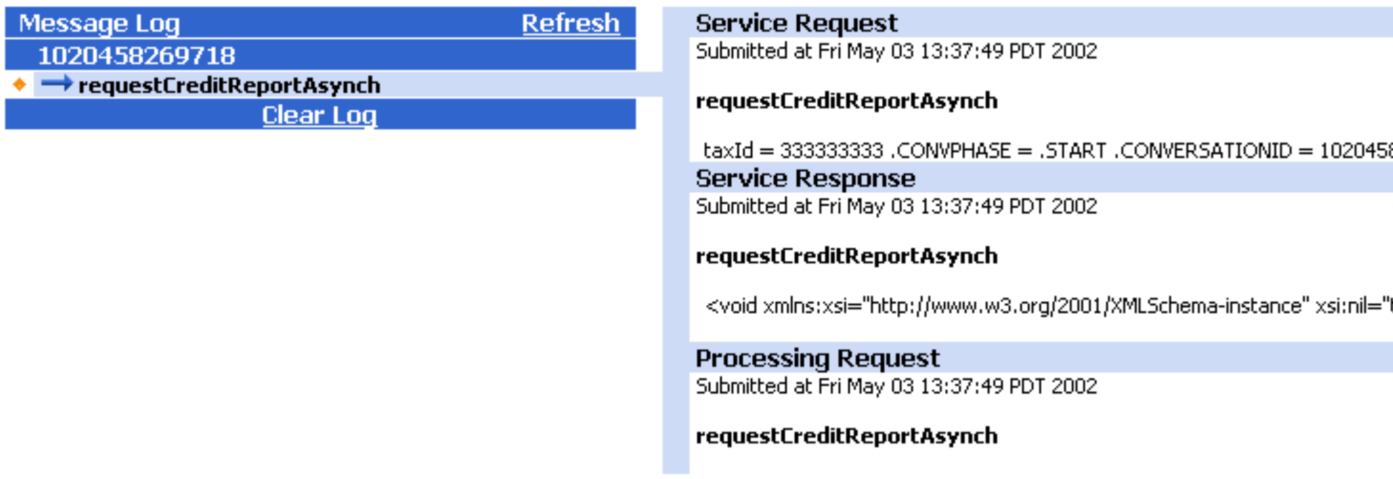
2. When Test View launches, type one of the following nine−digit numbers in the taxID box for requestCreditReportAsynch, as shown below:
   123456789, 111111111, 222222222, 333333333, 444444444, 555555555


requestCreditReportAsynch

string **taxId**:

requestCreditReportAsynch starts a conversation

3. Click requestCreditReportAsynch. The Test View page refreshes to display a summary of the request parameters sent by the client and your service's response, as shown here:



Message Log Refresh
1020458269718
→ requestCreditReportAsynch
Clear Log

Service Request
Submitted at Fri May 03 13:37:49 PDT 2002

requestCreditReportAsynch

taxId = 333333333 .CONVPHASE = .START .CONVERSATIONID = 102045

Service Response
Submitted at Fri May 03 13:37:49 PDT 2002

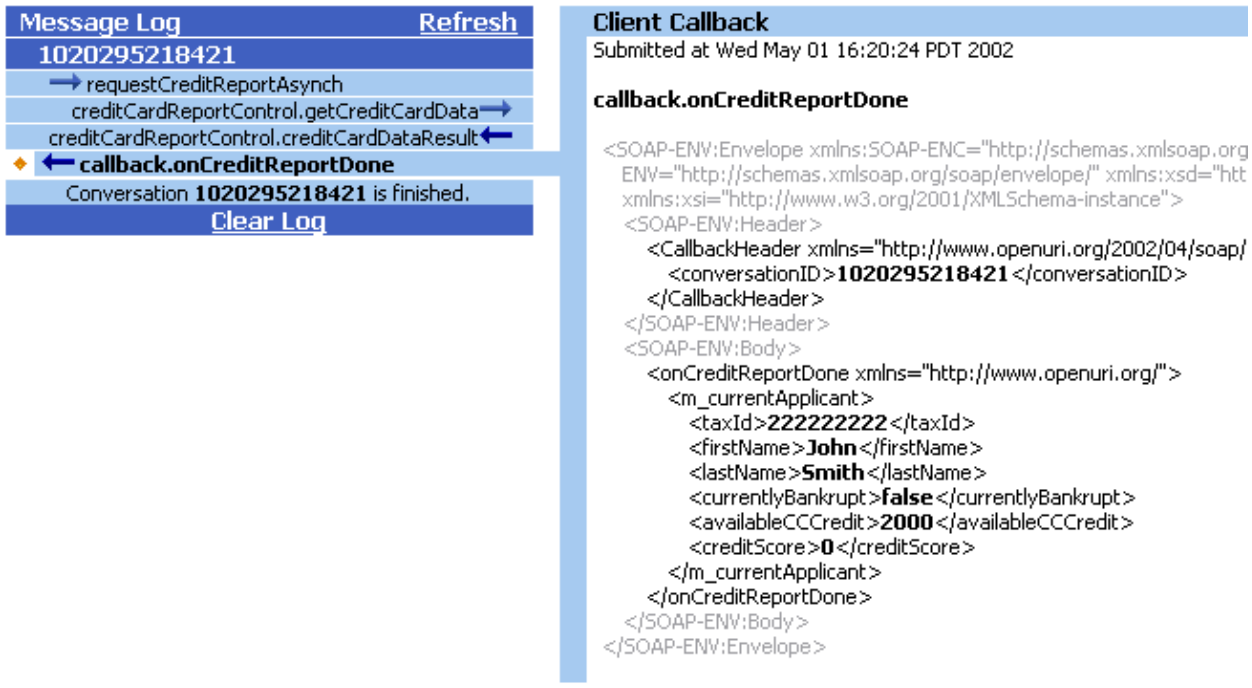
requestCreditReportAsynch

```
<void xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="
```

Processing Request
Submitted at Fri May 03 13:37:49 PDT 2002

requestCreditReportAsynch

4. Refresh the browser. New entries appear in the Message Log, as shown here:



Message Log Refresh
1020295218421
→ requestCreditReportAsynch
creditCardReportControl.getCreditCardData →
creditCardReportControl.creditCardDataResult ←
← **callback.onCreditReportDone**
Conversation **1020295218421** is finished.
Clear Log

Client Callback
Submitted at Wed May 01 16:20:24 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org
  ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="htt
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/
      <conversationID>1020295218421</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <taxId>222222222</taxId>
        <firstName>John</firstName>
        <lastName>Smith</lastName>
        <currentlyBankrupt>false</currentlyBankrupt>
        <availableCCCredit>2000</availableCCCredit>
        <creditScore>0</creditScore>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5. Click callback.onCreditReportDone. The callback sent from the web service to the client appears.

Related Topics

[Service Control: Using Another Web Service]

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

Step 5: Add a JMS and EJB Control

In this step, you will take the data you have gathered and request help from an in−house application to generate a credit score. You will then pass the credit score to an Enterprise Java Bean (EJB) that will measure the applicant's credit risk.

To generate the credit score, you will use an in−house credit scoring application designed to do just this sort of work. To make this legacy application available to other components, it has been exposed to other parts of the company through a messaging system. That's how your web service will get to it. You will send an XML message to the application with applicant data, and receive an XML message with a credit score.

Messaging systems are often used to create bridges between otherwise incompatible software components. Messaging is also useful when asynchronous communication is needed because the requesting component doesn't have to wait for a response in order to continue working. Here, asynchrony is useful because you have no way of knowing how long the credit scoring application will take to respond to your request. (For more information about messaging, see Overview: Messaging Systems and JMS.)

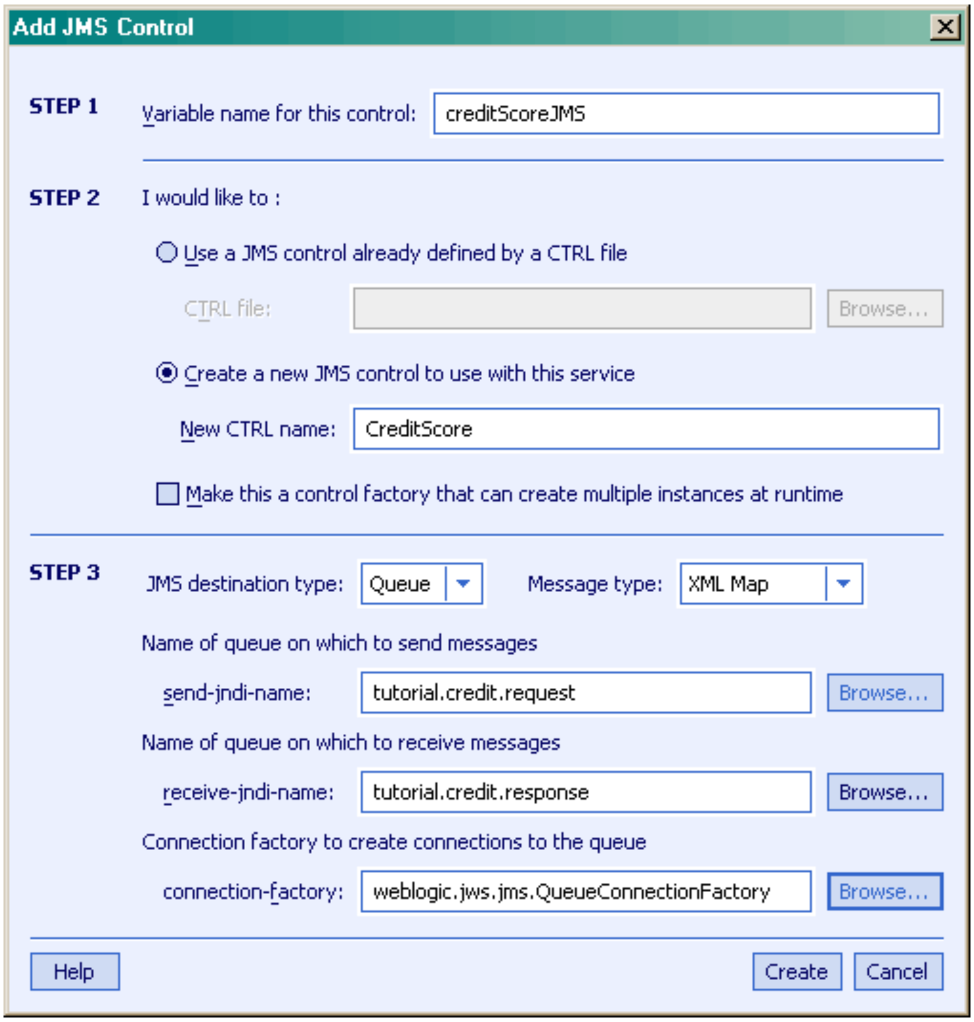
The Java Message Service (JMS), provided with WebLogic Server, supports several ways for applications to use messages. In point−to−point messaging, which you will use here, the message's sender and receiver are both clients of the messaging system (or "JMS provider"). The first client begins the exchange by sending a message to a queue. A queue is a kind of message waypoint to which other clients are listening. When the message arrives at the queue, the receiving client picks it up and reads it. You will use the JMS control to send a message (containing the data you have collected) to a queue that the credit scoring application is listening to. The application will send its response message (containing a credit score) to a queue that your service is listening to.

The tasks in this step are:

- To add a JMS control for requesting a credit score
- To add XML maps for JMS control
- To edit the callback handlers
- To add an EJB control for requesting a credit approval rating
- To add code that requests credit validation
- To launch the Test View

To Add a JMS Control for Requesting a Credit Score

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list, select Add JMS Control. The Add JMS Control dialog appears.
3. Enter values as shown in the following illustration:


Add JMS Control

STEP 1 Variable name for this control: creditScoreJMS

STEP 2 I would like to :

○ Use a JMS control already defined by a CTRL file

CTRL file: Browse...

◉ Create a new JMS control to use with this service

New CTRL name: CreditScore

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

STEP 3 JMS destination type: Queue Message type: XML Map

Name of queue on which to send messages

send-jndi-name: tutorial.credit.request Browse...

Name of queue on which to receive messages

receive-jndi-name: tutorial.credit.response Browse...

Connection factory to create connections to the queue

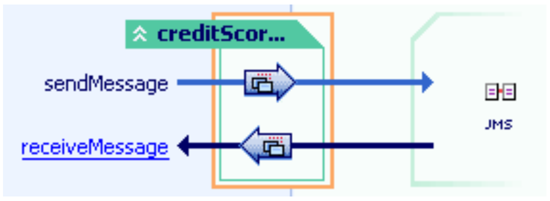
connection-factory: weblogic.jws.jms.QueueConnectionFactory Browse...

Help Create Cancel

Note that the message type is XML Map. XML maps are a powerful tool for turning your Java types into XML. These will be covered in more detail later in this tutorial.

4. Click Create.

Your service design now includes the creditScoreJMS control, as shown in the following illustration.



To Add XML Maps to the JMS Control

1. Double−click the arrow corresponding to the sendMessage method. The Edit Maps and Interface dialog appears.
2. Confirm that the Message XML tab is selected.
3. Edit the contents of the Java field so that it contains the following:

```
public void sendMessage(int availableCredit, boolean bankruptcy)
```

Using this method, you will send data about the applicant to the credit scoring application.

4. Leaving the dialog open, edit the contents of the top box so it contains the following:

```
<score_request>    <credit_remaining>{availableCredit}</credit_remaining>
   <is_bankrupt>{bankruptcy}</is_bankrupt> </score_request>
```

Note: This code represents an XML map. The <score_request>, <credit_remaining>, and <is_bankrupt> elements define the XML expected by the credit scoring application. Notice that the text in curly braces matches parameters of the declaration. This tells WebLogic Server how to insert the values passed to the sendMessage method into the XML. The XML is then sent to the application.

5. Click OK.
6. Double−click the arrow corresponding to the receiveMessage callback handler. The Edit Maps and Interface dialog appears.
7. On the Message XML tab, edit the XML map code as follows:

```
<score_response>    <calculated_score>{score}</calculated_score> </score_response>
```

8. Edit the Java code as follows:

```
public void receiveMessage(int score)
```

This adds an XML map to the callback handler. The map matches the format of the expected XML response. Placing the score parameter in the curly braces tells WebLogic Server to pass the returned data to the callback handler's score parameter.

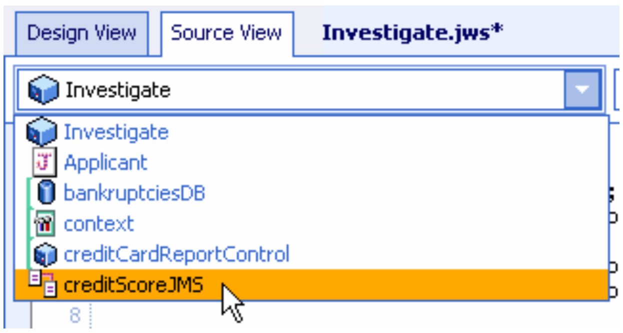
9. Click OK.

To Edit the Callback Handlers

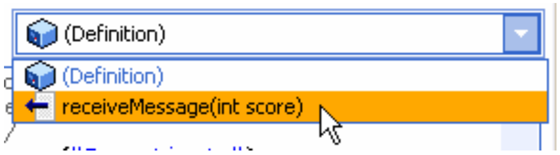
1. In Design View, click the name of the creditCardDataResult callback handler to view its source code.
2. Edit the creditCardDataResult callback handler code so that it appears as follows:

```
private void creditCardReportControl_creditCardDataResult(CreditCard[] cards) {       for(int i = 0; i <
cards.length; i++) {        m_currentApplicant.availableCCCredit += cards[i].availableCredit;    }        /*     *
Use the JMS control to send the available credit and bankruptcy information to the credit     * scoring
application.     */     creditScoreJMS.sendMessage(m_currentApplicant.availableCCCredit,
       m_currentApplicant.currentlyBankrupt); }
```

3. In Source View, immediately beneath the Source View tab, from the class drop−down list, select creditScoreJMS, as shown here:



4. To the right of the class drop−down list, from the member drop−down list, select receiveMessage, as shown here.



The source code for the JMS control's callback handler appears.

5. Edit the receiveMessage callback handler code so that it appears as follows:

```
private void creditScoreJMS_receiveMessage(int score) {    /* Store the score returned with other data about the applicant. */    m_currentApplicant.creditScore = score; }
```

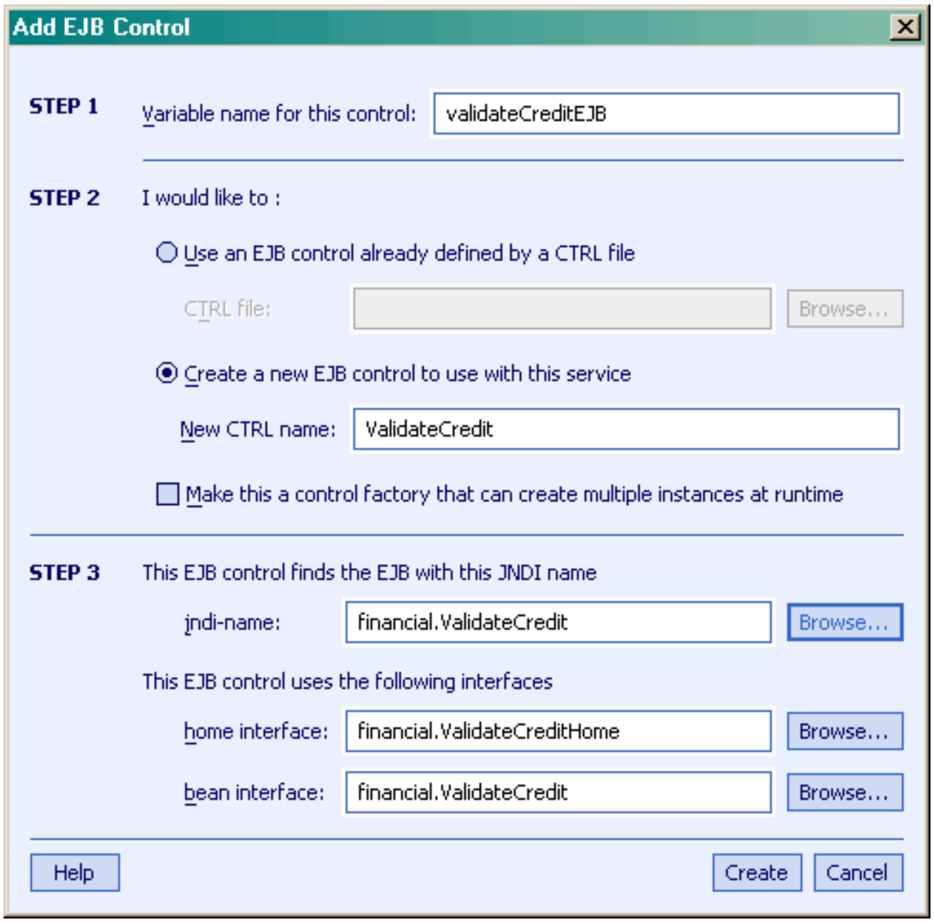
To Add an EJB Control for Requesting a Credit Approval Rating

Now that you have a way to retrieve a credit score, you will use the score to determine whether the applicant deserves credit approval. For this you will use an existing stateless session Enterprise Java Bean (EJB). EJBs are Java software components of enterprise applications. The Java 2 Enterprise Edition (J2EE) Specification defines the types and capabilities of EJBs as well as the environment (or container) in which EJBs are deployed and execute. From a software developer’s point of view, there are two aspects to EJBs: development and deployment of EJBs; and use of EJBs from client software. WebLogic Workshop provides the EJB control as a simplified way to act as a client of an existing EJB from within a web service.

The ValidateCredit EJB you will be accessing is designed to take the credit score and return response about the applicant's credit worthiness. To access the ValidateCredit bean, you will add an EJB control.

Note: In order for you add an EJB control to a web service project, the compiled home and remote interfaces for the EJB must be in the project also. For the sake of the tutorial, the JAR file containing the ValidateCredit bean has already been copied to the WEB−INF\lib folder of the samples project. Having the bean's JAR file in the WEB−INF\lib folder ensures that WebLogic Workshop can find these interfaces. (For more information, see Creating a New EJB Control.)

1. If you are not in Design View, click the Design View tab.
2. From the Add Control drop−down list, select Add EJB Control. The Add EJB Control dialog appears.
3. Enter values as shown in the following illustration. Browse for the jndi−name value required in Step 3 of the dialog. When you select it from the list and click Select, the home interface and bean interface values will be added automatically.



Add EJB Control

STEP 1 Variable name for this control: validateCreditEJB

STEP 2 I would like to :

○ Use an EJB control already defined by a CTRL file

CTRL file: [ ] Browse...

◉ Create a new EJB control to use with this service

New CTRL name: ValidateCredit

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

STEP 3 This EJB control finds the EJB with this JNDI name

jndi-name: financial.ValidateCredit Browse...

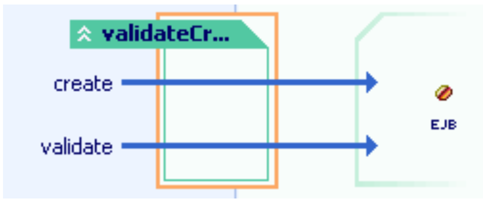
This EJB control uses the following interfaces

home interface: financial.ValidateCreditHome Browse...

bean interface: financial.ValidateCredit Browse...

4. Click Create.

You can see that WebLogic Workshop has added a validateCreditEJB control to your design, as shown below. The control shows that the EJB exposes two methods: create and validate. The validate method is the one you will use (all EJBs expose a create method that is used by WebLogic Server; you will not need to call this).



Now you need to connect the score received through the creditScore JMS control with the validateCredit EJB control. You will do this by updating the JMS control's callback handler. The code you add will send the credit score to the EJB for validation.

To Add Code that Requests Credit Validation

1. If you are not in Design View, click the Design View tab.
2. Click the name of the receiveMessage callback handler. Its source code appears in Source View.
3. Edit the receiveMessage code so that it appears as follows:

```
private void creditScoreJMS_receiveMessage(int score)    throws java.rmi.RemoteException {
   m_currentApplicant.creditScore = score;    /*     * Pass the credit score to the EJB's validate method. Store
the value returned     * with other applicant data.     */    m_currentApplicant.approvalLevel =
validateCreditEJB.validate(m_currentApplicant.creditScore);    /*     * Send the Applicant object, now
```

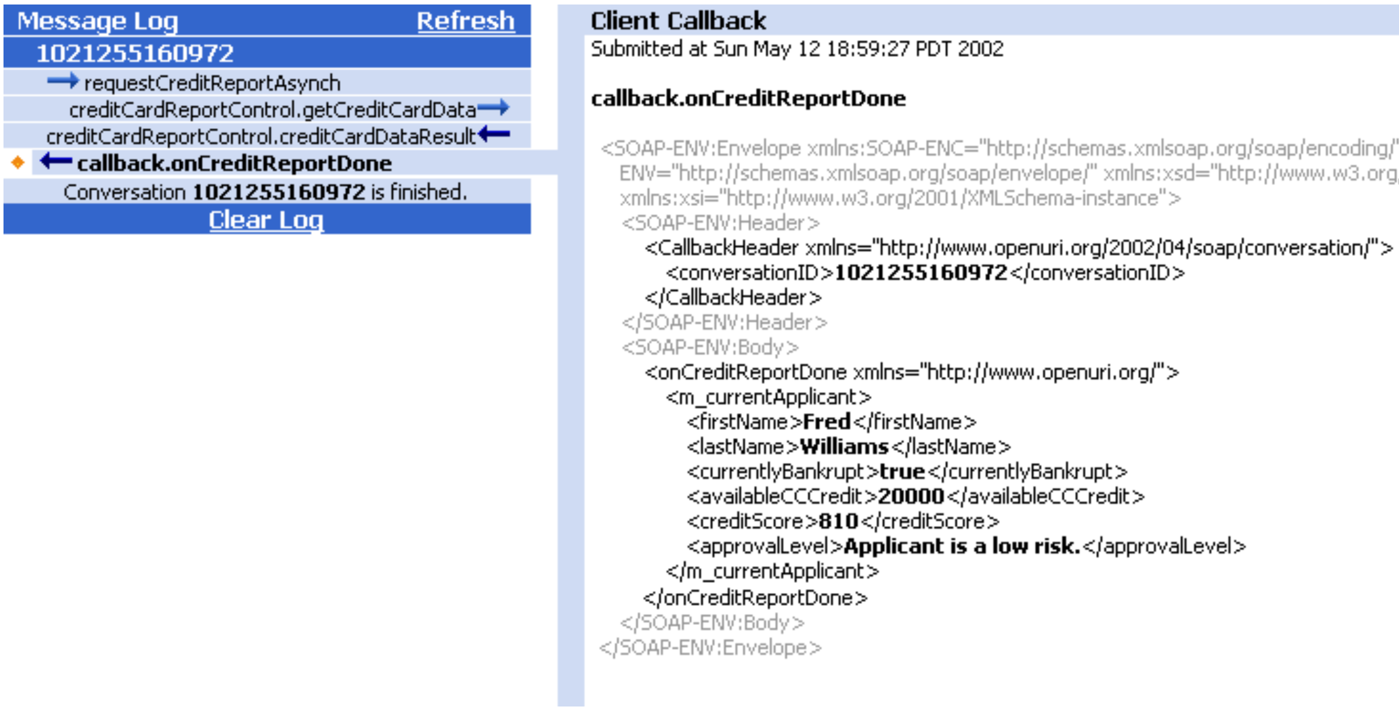
complete with all the data the client requested,   * to the client using the callback.   */
  callback.onCreditReportDone(m_currentApplicant, "Credit score received."); }

You've added a lot of functionality in these last steps. Now it's time to try it out and see if it all works.

To Launch Test View

1. Click the Start button. As before, this launches your browser to display the Test View.

2. Click the Refresh button to reload Test View until callback.onCreditReportDone appears in the Message Log. Your screen should resemble the following:



Message Log
Refresh
1021255160972
requestCreditReportAsynch
creditCardReportControl.getCreditCardData
creditCardReportControl.creditCardDataResult
callback.onCreditReportDone
Conversation 1021255160972 is finished.
Clear Log
Client Callback

Submitted at Sun May 12 18:59:27 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1021255160972</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <m_currentApplicant>
        <firstName>Fred</firstName>
        <lastName>Williams</lastName>
        <currentlyBankrupt>true</currentlyBankrupt>
        <availableCCCredit>20000</availableCCCredit>
        <creditScore>810</creditScore>
        <approvalLevel>Applicant is a low risk.</approvalLevel>
      </m_currentApplicant>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the <creditScore> and <approvalLevel> elements of the response now have values in them. This means that in just the few steps of this topic, you have connected the Investigate web service with a JMS messaging system and an Enterprise Java Bean.

In the next step you will enhance the service's design to return its result in a more generic format, making it more broadly useful.

Related Topics

[Using a Control]

[JMS Control: Using Java Message Service Queues and Topics from Your Web Service]

[EJB Control: Using Enterprise Java Beans from a Web Service]

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

⬅ ➡
Step 6: Add Script for Mapping

The Investigate web service returns an Applicant object to its client. That's fine for clients that have their own version of the Applicant class. When the data returns to them as a SOAP message, their software will translate the message into an Applicant object for their code to use. But what about other clients? In this step you will add an XML map to Investigate's callback to translate the Applicant object into plain XML that can be understood by another client. This client does not have the Applicant class, but does expect to receive XML that looks a particular way.You can shape the XML message the client receives with an XML map.

As you saw in the preceding step with the JMS control, XML maps look like XML documents. WebLogic Server uses XML maps you create to translate Java to XML, or XML to Java. Maps do this by showing where Java values should be inserted as XML element and attribute values (for incoming XML messages, it works the other way around). The direction of substitution depends on whether the XML message is sent or received by your web service.

You will apply an XML map to translate the data in the Applicant object into a particular XML shape. There's a twist in this case, though: The XML shape expected by the client includes an additional element. This element provides a place for the national percentile ranking of the applicant's credit score. But it's a little tedious to go back and retrofit your Java code for this client only. Instead, you can calculate the percentile through the map.

To do this, you will write a function in ECMAScript (also known as JavaScript), then refer to the function from the XML map. It will send the percentile back wrapped in XML that can be inserted into the XML map. This will be easier than you might expect, however. WebLogic Workshop includes support for an extended version of ECMAScript. These extensions turn XML nodes and node lists into native script objects. These objects also expose functions specifically designed for use with XML.

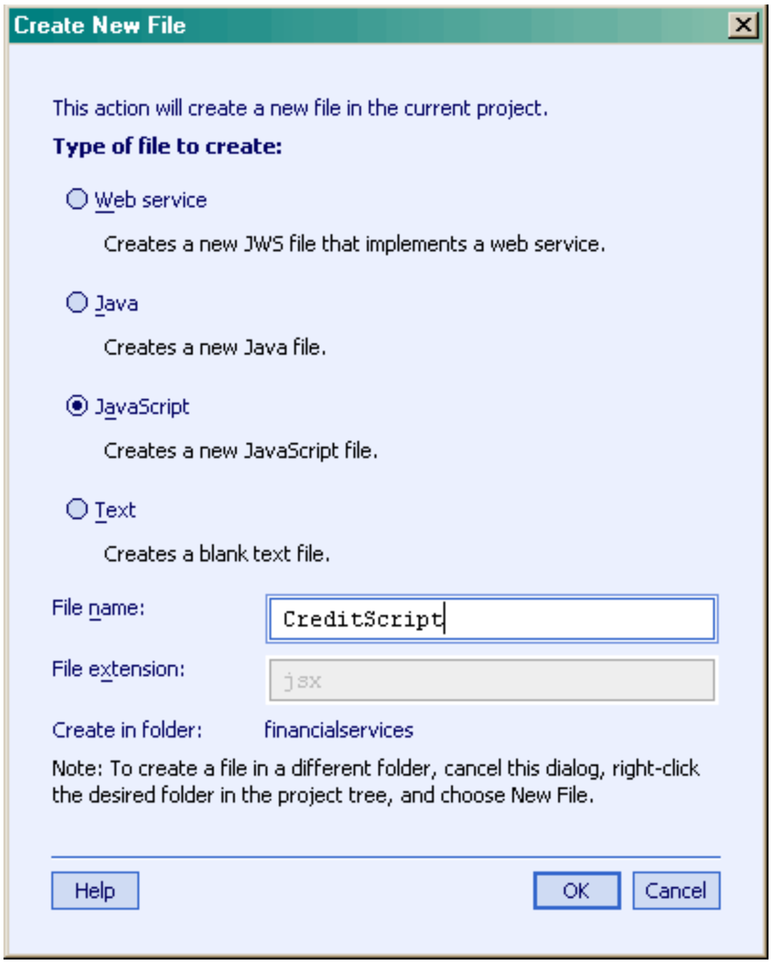
The tasks in this step are:

- To add a script file for use with mapping
- To write script that maps Java types to XML
- To launch Test View

To Add a Script File for Use with Mapping

1. If you are not in Design View, click the Design View tab.
2. Right−click the financialservices folder, then select New File. The Create New File dialog opens.

3. Enter values as shown in the following illustration:


Create New File

This action will create a new file in the current project.

Type of file to create:

( ) Web service
Creates a new JWS file that implements a web service.

( ) Java
Creates a new Java file.

(●) JavaScript
Creates a new JavaScript file.

( ) Text
Creates a blank text file.

File name: CreditScript

File extension: jsx

Create in folder: financialservices

Note: To create a file in a different folder, cancel this dialog, right-click the desired folder in the project tree, and choose New File.

4. Click OK. This adds a JSX file called CreditScript.jsx. As you will see, files with a JSX extension have special meaning. WebLogic Workshop displays the new file in Source View.

Note: The new file includes comments and example code to get you started. As you can see, JSX files support mapping through functions that take a particular form. A function either translates from XML to Java, or Java to XML.

Because you are mapping results to the client, you only need a function that translates Java to XML. You might find that it simplifies your work to delete the other function declaration, along with documentation comments.

5. Edit the file so that only the following code remains:

// import mypackage.MyOuterClass.MyClass; /* Rename the "toXML" function given you by WebLogic Workshop, but keep the toXML suffix. This is required in order for WebLogic Server to know that this function is for translating from Java to XML. A "fromXML" function would be for translation in the other direction. */ function calcScoreInfoToXML(obj) { }

To Write Script that Maps Java Types to XML

Next, you will write script to help generate the XML shape expected by the client. The following XML gives an example of the intended shape. Your script will generate the portion in the <score_info> element; the bold portion will come from an XML map:

```
<applicant id="111111111">
```

```
   <name_last>Walton</name_last>
   <name_first>Bill</name_first>
   <bankrupt>true</bankrupt>
   <balance_remaining>1000</balance_remaining>
   <risk_estimate>Ha! Who are they trying to kid?</risk_estimate>
   <score_info>
      <credit_score>460</credit_score>
      <us_percentile>19.0</us_percentile>
   </score_info>
</applicant>
<notes>Credit score received.</notes>
```

1. At the top of the file, replace the example import statement with the following:

import financialservices.Investigate.Applicant;

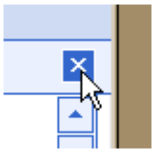
This imports the Applicant class so that you can access it in your script.

Note: The import directive here is not from Java; it is a part of the extended ECMAScript.

2. Edit the function's code so that it appears as follows:

import financialservices.Investigate.Applicant; /* The applicantJava parameter represents the Applicant object * passed from the onCreditReportDone callback through its XML map. */ function calcScoreInfoToXML(applicantJava) {   print(typeof applicantJava.creditScore);   /* Confirm that the applicantJava object passed into the function isn't null.   * If it is, trying to assign values from it will cause a script error.   * If it's null, your script will simply return two empty nodes to add to the   * message created by the XML map.   *   * In the scoreInfoNode variable assignment below, notice too that the two   * new nodes are enclosed in empty elements. These are known as "anonymous"   * elements. They are an ECMAScript extension that enables you to create lists   * of elements such as the two−member list below. Without them, the two elements   * alone would be malformed XML because it lacks a root element.   *   * Simply initializing the variable with a value that begins with < automatically   * makes the scoreInfoNode variable an XML object.   */   if(applicantJava == null) {   var scoreInfoNode = <>   <credit_score/>   <us_percentile/>   </>;   }   /* If the applicantJava value isn't null, use {} to insert a value from it   * into the <credit_score> element. Also, use a function defined in the script   * to calculate the credit score's national percentile ranking.   */   else {   var scoreInfoNode = <>   <credit_score>{applicantJava.creditScore}</credit_score>   <us_percentile>{calcPercentile(applicantJava.creditScore)}</us_percentile>   </>;   }   /* Return the completed XML. */   return scoreInfoNode; } /* A function to figure out the applicant's credit score percentile. */ function calcPercentile (score) {   var percentile = 0;   var scoreLowEnd = 300;   var number = 3162.5;   percentile = Math.ceil((Math.pow(score, 2) − Math.pow(scoreLowEnd, 2))/(2 * number));   return percentile; }

3. Click the close button in the upper right corner of Workshop to return to Design View for the Investigate web service, as shown here:



Note: If you don't see the Investigate service after closing the CreditScript.jsx file, double−click Investigate.jws in the Project Tree.

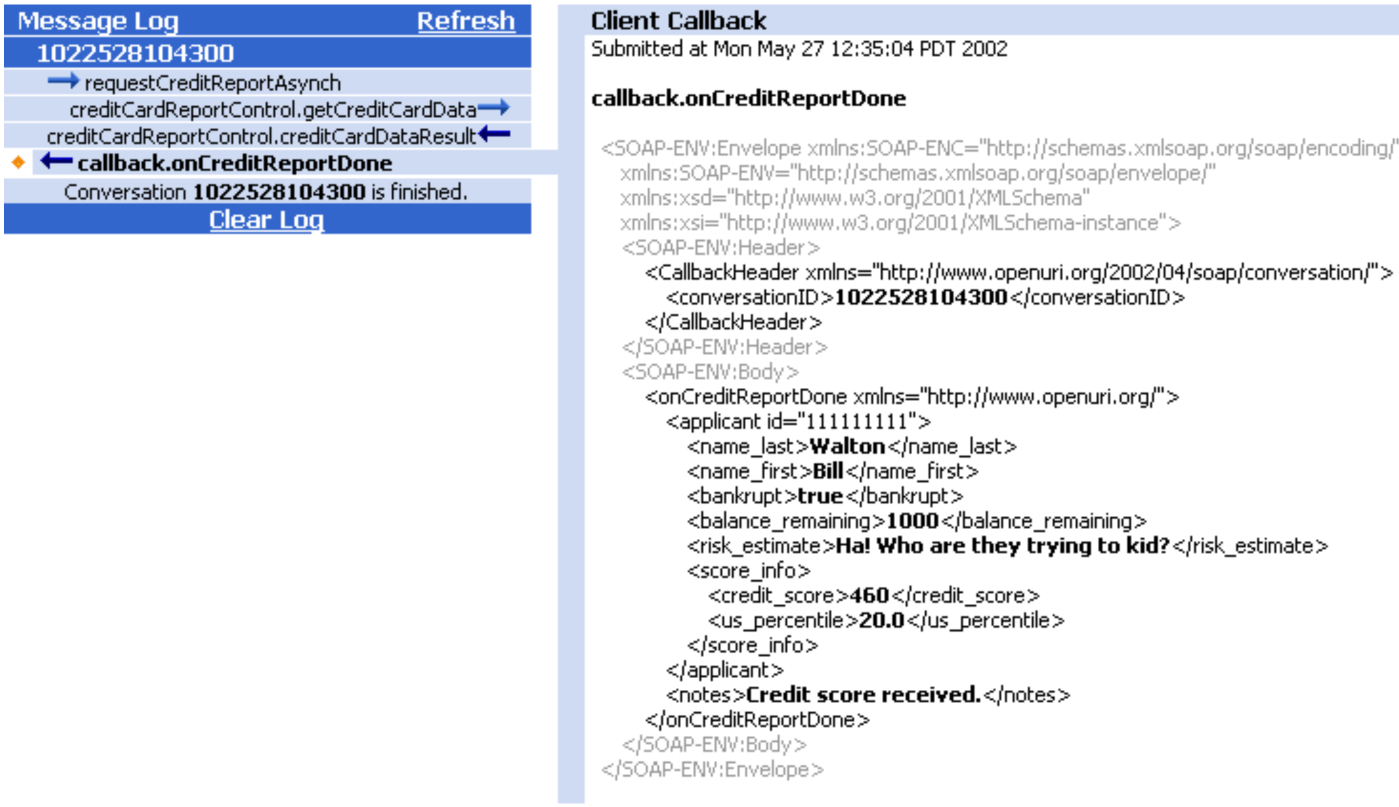
4. Double−click the arrow corresponding to the onCreditReportDone callback. The Edit Maps and Interface dialog appears.
5. On the Parameter XML tab, edit the top pane so that the XML map appears as follows:

```
<onCreditReportDone xmlns="http://www.openuri.org/">    <applicant id="{m_currentApplicant.taxID}">
       <name_last>{m_currentApplicant.lastName}</name_last>
       <name_first>{m_currentApplicant.firstName}</name_first>
       <bankrupt>{m_currentApplicant.currentlyBankrupt}</bankrupt>
       <balance_remaining>{m_currentApplicant.availableCCCredit}</balance_remaining>
       <risk_estimate>{m_currentApplicant.approvalLevel}</risk_estimate>        <score_info>
           {financialservices.CreditScript.calcScoreInfo(m_currentApplicant)}        </score_info>    </applicant>
   <notes>{responseMessage}</notes> </onCreditReportDone>
```

Throughout this XML map, the text between {} tells WebLogic Server to insert values from the callback's parameter, m_currentApplicant. Take a look also at the map's <score_info> element. The code in {} there tells WebLogic Server to use the script you have written to translate the callback's m_currentApplicant parameter.

To Launch Test View

1. Click the Start button. As before, this launches your browser to display the Test View.
2. Enter one of the tax ID numbers in the taxID box (for example, enter 111111111), then click requestCreditReportAsynch.
3. Click the Refresh button to reload Test View until callback.onCreditReportDone appears in the Message Log.
4. Click the callback.onCreditReportDone message. Your screen should resemble the following:



1022528104300
requestCreditReportAsynch
creditCardReportControl.getCreditCardData
creditCardReportControl.creditCardDataResult
callback.onCreditReportDone
Conversation **1022528104300** is finished.

Client Callback
Submitted at Mon May 27 12:35:04 PDT 2002

callback.onCreditReportDone

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <CallbackHeader xmlns="http://www.openuri.org/2002/04/soap/conversation/">
      <conversationID>1022528104300</conversationID>
    </CallbackHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <onCreditReportDone xmlns="http://www.openuri.org/">
      <applicant id="111111111">
        <name_last>Walton</name_last>
        <name_first>Bill</name_first>
        <bankrupt>true</bankrupt>
        <balance_remaining>1000</balance_remaining>
        <risk_estimate>Ha! Who are they trying to kid?</risk_estimate>
        <score_info>
          <credit_score>460</credit_score>
          <us_percentile>20.0</us_percentile>
        </score_info>
      </applicant>
      <notes>Credit score received.</notes>
    </onCreditReportDone>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Compare this image with the onCreditReportDone results in the preceding step of this tutorial. You will see that this one is quite different from that one due to your XML map. Through the map, you have shaped the outgoing XML message so that a particular client can use it.

In the next step you will enhance the service's design to return its result in a more generic format, making it more broadly useful.

Related Topics

Why Use XML Maps?

Getting Started with XML Maps

Getting Started with Script for Mapping

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

Step 7: Add Support for Cancellation and Exception Handling

Working through the first steps of this tutorial has given you a web service that works. In ideal conditions, it does what it is supposed to do. But as it now stands, the service isn't quite ready for prime time. Here are a few possible problems:

- The client may want to cancel the credit report before Investigate has sent back a response.

Providing this ability is especially important in asynchronous exchanges, which may be long−lived.

- Investigate's dependency on the credit card web service (which is accessed asynchronously) may make the overall response time to the client quite long.

You have no way of knowing how long it will take the credit card service to respond to your requests. To better manage this, you can use a timer to set a limit on how the resource may take to respond.

- Operation methods (such as requestCreditReportAsynch) may throw an uncaught exception.

In your current design, this would leave your service, and its client, hanging. The active conversation would continue until it expired. The client would not receive a response to its request and might never know why. You can handle such exceptions to ensure a clean recovery.

Through the following procedures, you will enhance the Investigate web service to better handle these possible problems.

This tasks in this step are:

- To add a method so that clients can cancel operations
- To add a TimerControl to limit the time allowed for response
- To handle exceptions thrown from operation methods

To Add a Method so that Clients Can Cancel Operations

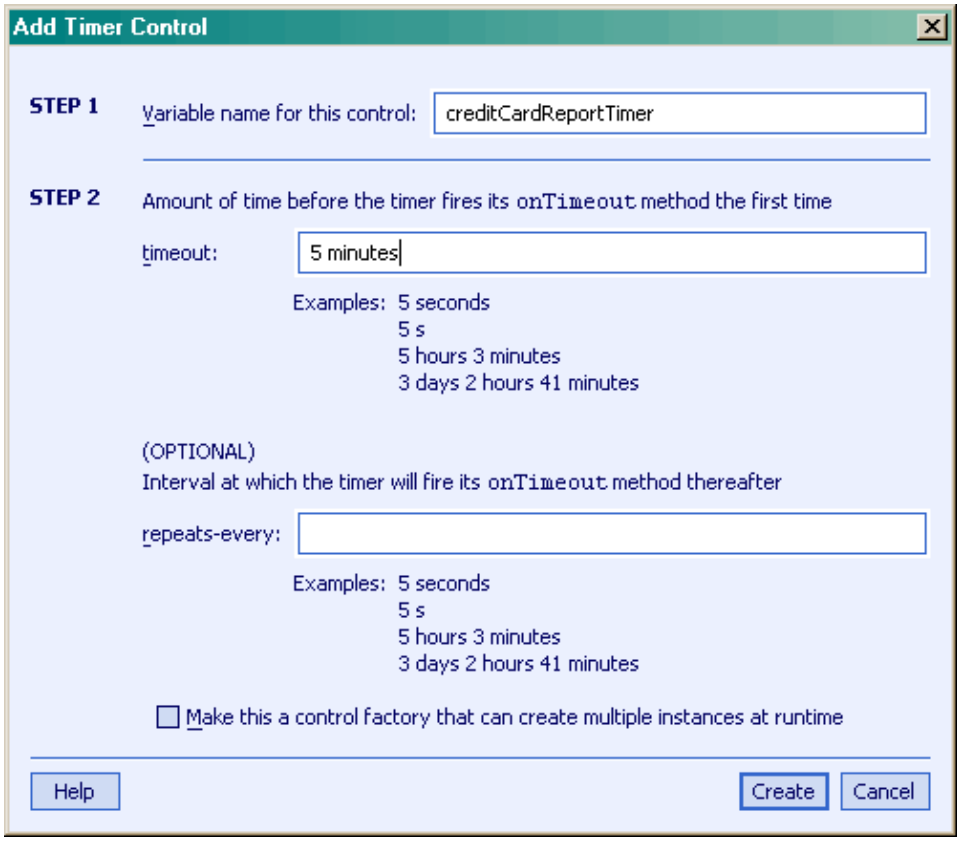
1. If you are not in Design View, click the Design View tab.
2. From the Add Operation drop−down list, select Add Method.
3. In the field that appears, type cancelInvestigation and press Enter. the method appears in Design View.
4. Click cancelInvestigation to view its code in Source View.
5. Edit the cancelInvestigation method code so it appears as follows:

/** * @jws:operation * @jws:conversation phase="finish" */ public void cancelInvestigation() {   /* Cancel the request to the credit card company because it is now unnecessary. */
   creditCardReportControl.cancelRequest();   /* Use the callback to send a message to the client. Note that this also ends   * the conversation because the callback's conversation phase property is set to "finish". */
   callback.onCreditReportDone(null, "Investigation canceled at client's request."); }

To Add a TimerControl to Limit the Time Allowed for Response

It can be difficult to predict how long an asynchronous resource will take to respond to a request. For example, Investigate's call to the credit card service may take hours. Here, you will add a way to limit the amount of time the credit card's web service has to respond. Using a TimerControl, you can specify an amount of time after which the operation should be canceled.

1. Click the Design View tab to return to Design View.
2. From the Add Control drop−down list, select Add Timer Control. The Add Timer Control dialog appears.
3. Enter values as shown in the following illustration:


Add Timer Control

STEP 1 Variable name for this control: creditCardReportTimer

STEP 2 Amount of time before the timer fires its onTimeout method the first time

timeout: 5 minutes

Examples: 5 seconds
5 s
5 hours 3 minutes
3 days 2 hours 41 minutes

(OPTIONAL)
Interval at which the timer will fire its onTimeout method thereafter

repeats-every:

Examples: 5 seconds
5 s
5 hours 3 minutes
3 days 2 hours 41 minutes

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

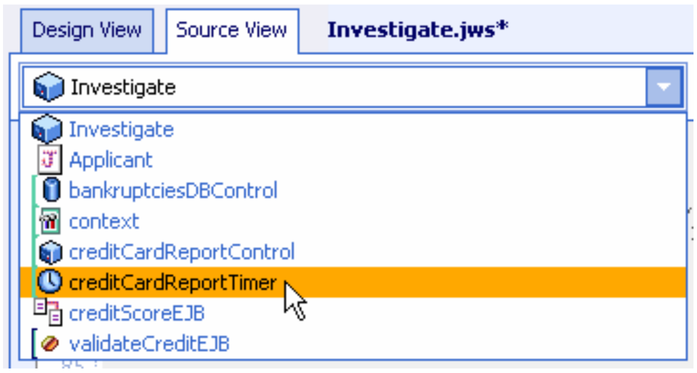
These values specify that the TimerControl will send its onTimeout callback five minutes after the timer starts.

4. Click Create. You are returned to Design View.
5. Click requestCreditReportAsync to view its code in Source View.
6. At the very end of the requestCreditReportAsynch source code, shown below, add the code shown in bold:

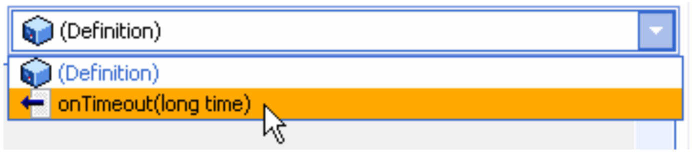
creditCardReportControl.getCreditCardData(taxID); creditCardReportTimer.start();

This start the timer as soon as the credit card service is called. If the service does not respond by the time the timeout duration is reached, then the timer's onTimeout callback handler is invoked.

7. In Source View, immediately beneath the Source View tab, from the class drop−down list, select creditCardReportTimer, as shown here:



8. To the right of the class drop−down list, from the member drop−down list, select onTimeout, as shown here



The source code for the JMS control's callback handler appears.

9. Edit the callback handler source code so that it appears as follows:

private void creditCardReportTimer_onTimeout(long time) {   /* Because the credit card service has not yet returned, cancel the request. */   creditCardReportControl.cancelRequest();   /* Send a response to the client, saying something about what happened.   * Remember that this will also effectively finish the conversation.   */   callback.onCreditReportDone(null, "Unable to retrieve credit card information."); }

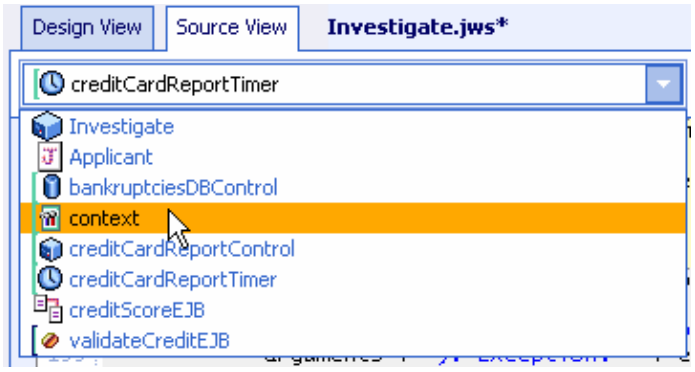
To Handle Exceptions Thrown from Operation Methods

Unhandled exceptions thrown from operations (such as methods exposed to clients) can interrupt your service's work and leave the client hanging. Such exceptions do not automatically end a service's conversation. Instead, the service may simply continue on the system, unnecessarily using resources. As a result, one action your code should take when responding to such exceptions is to finish the conversation.

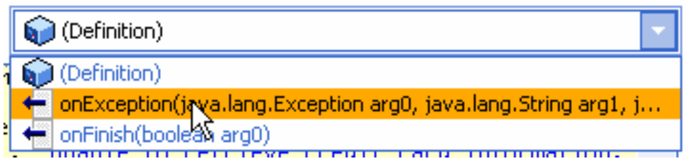
The JwsContext interface that provides the finish method for ending the conversation provides other useful functionality. In this step, you will add code to implement a handler for the JwsContext.onException callback. The callback handler receives the exception object thrown from the operation, the name of the method that threw the exception, and the parameters passed to the method.

WebLogic Workshop provides a way for you to preserve information about exceptions by logging it in a text file. You will add exception handling code to log the exception, then send an appropriate response to the client.

1. In Source View, immediately beneath the Source View tab, from the class drop−down list, select context, as shown here:



2. To the right of the class drop−down list, from the member drop−down list, select onException, as shown here:


(Definition)
(Definition)
onException(java.lang.Exception arg0, java.lang.String arg1, j...
onFinish(boolean arg0)

The source code for the JwsContext onException callback handler appears.

3. Edit the onException callback handler code so that it appears as follows:

```
public void context_onException(Exception e, String methodName, Object[] arguments) {     /* Create a logger variable to use for logging messages. Assigning it the      * "Investigate" category name will make it easier to find messages from this      * service in the log file. */     Logger logger = context.getLogger("Investigate");     /* Log an error message, giving the name of the method that threw the      * exception and a stack trace from the exception object. */     logger.error("Exception in " + methodName + ": " + e);     /* Invoke the callback to send the client a response. */     callback.onCreditReportDone(null, "Unable to respond to request at this time. " +         "Please contact us at 555−5555."); }
```

4. At the top of Investigate's source code, add the following line of code as the last of the imports:

```
import weblogic.jws.util.Logger;
```

Note: This imports the Logger class that makes logging possible.

By default, for the domain in which you develop and debug WebLogic Workshop web services, the log file is located at the following path of your WebLogic Workshop installation:

BEA_HOME/weblogic700/samples/workshop/jws.log

An entry in the log will resemble the following:

```
16:18:11 ERROR Investigate     : Exception in requestCreditReportAsynch: <exceptionStackTrace> [ServiceException]
```

Because you're probably anxious to move on to creating an actual client for the Investigate service, you won't test the code you've added here. But if you want to try it out some time, you might do the following:

- Test the cancellation code by starting the service, then cancelling it in Test View. When Test View launches, enter a tax ID for testing. After the test has begun, click the "Continue this conversation" link. When the page refreshes, click the cancelInvestigation button.
- Test the timer by changing its timeout value to 1 second. Unless your computer is very fast, this should be enough time for the credit card service to return before the timeout. After changing the timeout value, start the service and wait one second, then click the Refresh link.
- Test the exception handler by throwing an exception from somewhere in your code.

Related Topics

Timer Control: Using Timers in Your Web Service

JwsContext.onException Callback

How Do I: Handle Errors in a Web Service?

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

Step 8: Client Application

In this step you will build a Java console client to invoke your web service. Although your web service is currently designed to support a client capable of receiving callbacks, building a client that can receive callbacks is beyond the scope of this tutorial. As an alternative, you will modify your web service to support a polling interface. That is, you will modify your web service so that a client can periodically make requests to, or poll, the web service for the results.

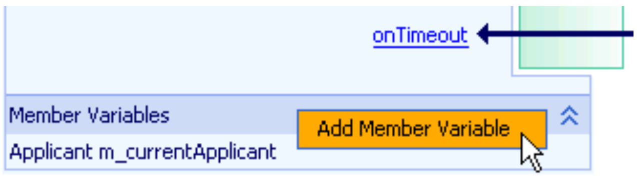
The tasks in this step are:

- To modify your web service to support a polling interface
- To save the web service's proxy classes
- To compile and run the Java client application

To Modify the Web Service to Support a Polling Interface

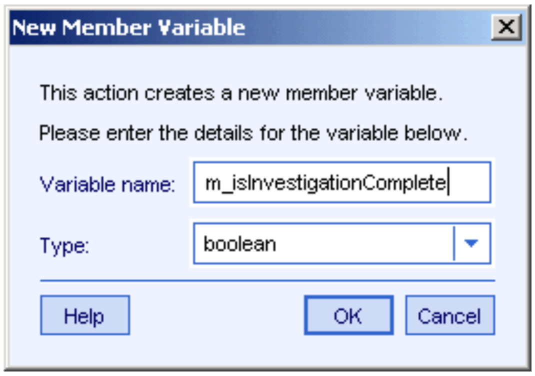
In this section, you will add a new member variable to your web service called m_isInvestigationComplete so that your web service will know when it is time to provide the complete results to the Java client. This member variable records whether the web service has completed its task of building a credit profile of an applicant. When m_isInvestigationComplete is false, the web service returns a null response each time the Java client polls for information. Once m_isInvestigation is true, the web service returns a full profile of the credit applicant to the Java client poll.

1. If you are not in Design View, click the Design View tab.
2. Right−click Member Variables and select Add Member Variable, as shown here:



The New Member Variable dialog appears.

3. In the Variable name field, type m_isInvestigationComplete, as shown below.
4. Confirm that the value in the Type drop−down list is boolean, as shown below.


New Member Variable

This action creates a new member variable.

Please enter the details for the variable below.

Variable name: m_isInvestigationComplete

Type: boolean

5. Click OK.
6. Click receiveJMSMessage. The creditScore_receiveJMSMessage code appears in Source View.
7. Add the code in bold, shown here:

```
private void creditScoreEJB_receiveJMSMessage(Message msg)
    throws JMSException, java.rmi.RemoteException
{
    creditScoreTimer.stop();

    m_currentApplicant.creditScore = ((MapMessage)msg).getInt("credit_score");
    System.out.println(m_currentApplicant.creditScore);
    m_currentApplicant.approvalLevel = validateCreditEJB.validate(m_currentApplicant.credit
    if(context.getCallbackLocation() != "")
        callback.onCreditReportDone(m_currentApplicant, "Credit score received.");
    else
        m_isInvestigationComplete = true;
}
```

8. Add the following method to the web service:

```
/**
 * @jws:operation
 * @jws:conversation phase="continue"
 */
public Applicant checkForResults()
{
    Applicant retval = null;
    if(m_isInvestigationComplete == true){
        retval = m_currentApplicant;
        context.finishConversation();
    }
    return retval;
}
```

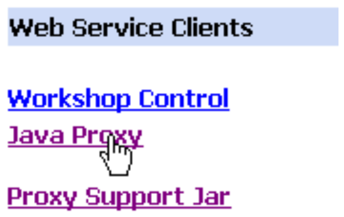
9. Press Ctrl+S to save Investigate.jws.

To Save the Web Service Proxy Classes

In this task, you will save two JAR files (Java Application Archive files) that will form an interface between your web service and the Java console client. These JAR files contain Java classes which form a proxy of your web service, a proxy through which your client can invoke the actual web service.

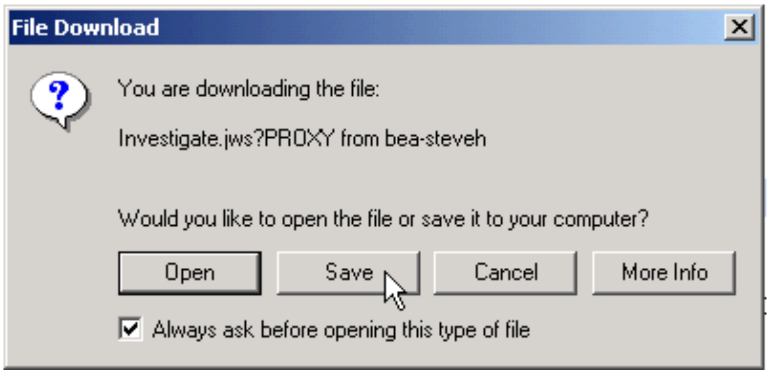
1. Press F5 to compile your web service. Your web browser launches Test View.

2. In the browser window showing Test View there are five tabs at the top of the screen: Overview, Consol, Test Form, Test XML, and Warnings. Click the Overview tab.
3. Under Web Service Clients, click Java Proxy, as shown here:


Web Service Clients

Workshop Control
Java Proxy
Proxy Support Jar

The File Download dialog appears, as shown here:


File Download

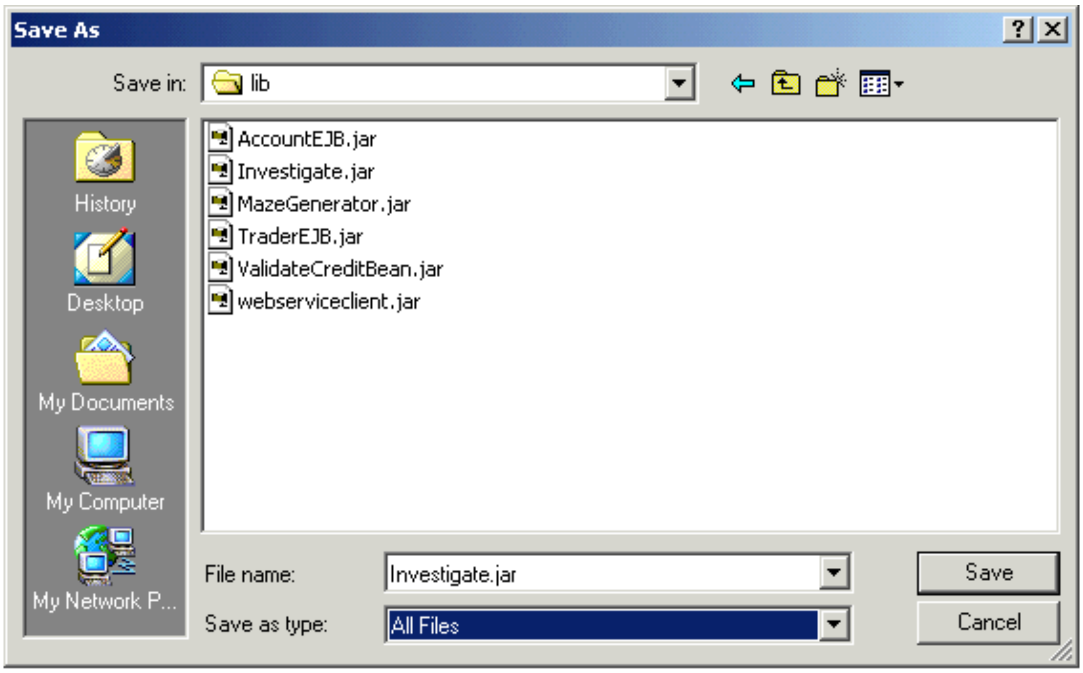
You are downloading the file:

Investigate.jws?PROXY from bea-steveh

Would you like to open the file or save it to your computer?

☑ Always ask before opening this type of file

1. Click Save. The Save As dialog appears.
2. In the Save in drop−down list, navigate to C:\bea\weblogic700\samples\workshop\applications\samples\WEB−INF\lib
3. In the Save as type drop−down list, make sure that All Files is selected, as shown here:


Save As

Save in: lib

AccountEJB.jar
Investigate.jar
MazeGenerator.jar
TraderEJB.jar
ValidateCreditBean.jar
webserviceclient.jar

File name: Investigate.jar
Save as type: All Files

6. Click Save to save Investigate.jar. The Download Complete dialog appears.
7. Click Close.

8. Repeat the entire process for the Proxy Support Jar file.

To Compile and Run the Java Client Application

Now that the JAR files are in place you are ready to compile and run the Java client application.

1. Open a command prompt window.
2. At the command prompt, cd to the following directory:
   C:\bea\weblogic700\samples\workshop\applications\samples\tutorials\java_client
3. To compile the Java console client, type compile and press Enter.
4. To run the Java console client, type run and press Enter. The command prompts you to enter a tax identification number for an applicant.
5. Type one of the following 9 digit numbers:

123456789, 111111111, 222222222, 333333333, 444444444, 555555555

The client now polls the web service every second until the credit report is ready. Once it is ready, the Java console application displays the results.

Note: you may a get warning that your client does not support callbacks. You can ignore this warning because the Java console client does not rely on receiving the web service response as a callback.

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

⬅ ➡

Step 9: Deployment and Security

Your web service is one that sends sensitive information across the internet, so it would be a good idea to insure that your service is a secure one. In this step of the tutorial you will modify your web service so that is it exposed through HTTPS (Secure HTTP) instead of HTTP.

Note that the web services you build with Workshop compiled as EAR files, and then are ultimately deployed to WebLogic Server as web applications. When finally deployed on WebLogic Server, your web service is contained within a web application and exposed through that web application. It is the containing web application that is exposed through an HTTPS−enabled port of WebLogic Server, so a web service is secure because it is contained in a web application that is exposed through a HTTPS−enabled port.

The tasks in this step are:

- To configure the web service to be exposed on HTTPS
- To package the web application as an EAR file
- To deploy the web application on a production server

To Configure the Web Service to Be Exposed on HTTPS

You configure the exposure protocol of your web service by editting the weblogic−jws−config.xml file that resides in the samples project's WEB−INF directory.

1. In the Project pane, open the WEB−INF directory.
2. Open the file weblogic−jws−config.xml.
3. Edit weblogic−jws−config.xml to look like the following. Make sure to remove the comment tags (<!−− and −−>) to activate the XML elements:

```
<config> <protocol>http</protocol> <hostname>localhost</hostname> <http−port>7001</http−port>
<https−port>7002</https−port> <jws>  <class−name>financialservices.Investigate</class−name>
```

```
 <protocol>https</protocol> </jws> ... </config>
```

To Package the Web Application as an EAR File

1. Stop WebLogic Server and close WebLogic Workshop
2. Using Windows Explorer, create a new directory C:\EARS
3. In a new command window, cd to the directory in which Investigate.jws resides. If you installed WebLogic Server on the C drive, the directory is:

```
C:\bea\weblogic700\samples\workshop\applications\samples\financialservices
```

4. Edit the PATH environmental variable by pressing Start−−>Settings−−>Control Panel. Double click System. Click the Advanced Tab. Click the Environmental Variables... button. Under System Variables select the Path variable. Click the Edit... button. Append the following to the Variable Value:
   C:\bea\weblogic700\server\bin
   Make sure that a semi−colon separates the value you append and the values already present.

5. Run the following command.

```
jwsCompile −ear C:\EARS\MyFirstWebService.ear −app MyFirstWebService Investigate.jws
```

Note that you may get a warning about the namespace of your web service. For the sake of the tutorial, you can ignore these warnings. If you are planning to deploy a web service in a real production scenario, you would probably want to change the namespace to something more unique than "openuri".

To Deploy the Web Application on a Production Server

1. In a new command window, start WebLogic Server in production mode using the following:

```
C:\bea\weblogic700\samples\workshop\startWeblogic.cmd production nodebug
```

2. Edit the PATH environmental variable by pressing Start−−>Settings−−>Control Panel. Double click System. Click the Advanced Tab. Click the Environmental Variables... button. Under System Variables select the Path variable. Click the Edit... button. Append the following to the Variable Value:
   C:\bea\jdk131_03\bin
   Make sure that a semi−colon separates the value you append and the values already present.
3. In a new command window, run the following:

```
java −cp C:\bea\weblogic700\server\lib\weblogic.jar weblogic.Deployer −password installadministrator
−source C:\EARS\MyFirstWebService.ear −targets cgServer −name MyFirstWebService −activate
```

3. Open an internet browser, and enter the following address:

```
http://localhost:7001/console
```

4. Logon to WebLogic Server using the password "installadministrator".
5. Navigate to workshop/Deployments/Applications/MyFirstWebService, using the navigation pane on the left−hand side of the screen.
6. Note the two files contained in MyFirstWebService: MyFirstWebService.war and financialservices.InvestigateEJB.jar
7. Open another internet browser, and enter the following address:

https://localhost:7002/MyFirstWebService/financialservices/Investigate.jws

8. The server will present you with a digital certificate, and a summary of the Investigate web service will appear.
9. Modify the address to look like the following:

https://localhost:7002/MyFirstWebService/financialservices/Investigate.jws?WSDL

A WSDL file will appear. This is the WSDL that clients to your web service will use to learn how to operate your service.

You now have a complete and deployed web service ready to take requests from clients.

Summary: Your First Web Service

This tutorial introduced you to the basics of building web services. Along the way, you became acquainted not just with how to use WebLogic Workshop, but with considerations in the design of web services.

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

- For an overview of WebLogic Workshop, see Building Web Services with WebLogic Workshop.
- For a links to information on tasks you can accomplish in WebLogic Workshop, see the How Do I? topics.
- For a list of the samples provided, see Samples.

Concepts and Tasks Introduced in This Tutorial

- You interact with web service source files through a WebLogic Workshop project. You use projects to group files associated with a web service or related web services.

For more information about projects, see WebLogic Workshop Projects.

- As you develop web services, you test and debug code on a running instance of WebLogic Server.

For information on WebLogic Server, see BEA WebLogic Server 7.0 Release Documentation. To learn how to start WebLogic Server from within WebLogic Workshop, see How Do I: Start or Stop WebLogic Server?

- You design a web service as you might make a drawing of it and its relationships with clients and other resources. WebLogic Workshop provides a Design View you can use to generate code to start with as you create your design.

For reference information on Design View, see Design View.

- The source file for a web service is a JWS file, which is automatically recognized as a web service when deployed with WebLogic Server.

For an introduction to JWS files, see JWS Files.

- You expose the functionality of a web service by creating methods within a JWS file. You can add the method in Design View, setting properties to specify its characteristics, including powerful server features.
- To test a service, you use the Test View, a dynamically generated HTML page through which you can invoke service methods with specific parameter values.

For reference information on Test View, see Test View.

- You can easily access databases from your web service with the DatabaseControl.

For an introduction and detailed information about the DatabaseControl, see Database Control: Using a Database from Your Web Service.

- You can use asynchronous communications to handle latency issues inherent on the Internet, and sometimes with web services in general. Through asynchrony, you avoid situations in which client software is blocked from proceeding until it receives the results of its requests.

For an introduction to asynchrony and the WebLogic Workshop tools that support it, see Using Asynchrony to Enable Long−Running Operations.

- Callbacks provide a useful way to return results to clients where asynchronous communication is needed. Callbacks require an agreement that the client will implement a means to handle the callback a service makes.
- For more details, see Using Callbacks to Notify Clients of Events.
- You can use conversations to maintain consistent state even between disparate asynchronous exchanges. Conversations make it possible to correlate these exchanges with the original request and the client that made it.

For more information, see Maintaining State with Conversations and Using Asynchrony to Enable Long−Running Operations.

- You can access other web services using the ServiceControl. By generating a ServiceControl from the WSDL of another web service. When you do, you have a single component to represent the other service in your service's design.

For an introduction and details about the ServiceControl, see Service Control: Using Another Web Service.

- You can access components that are available via the Java Message Service (JMS) by using the JMSControl. Through this control, you can send and receive messages of various types, including XML.

For more on the JMSControl, see JMS Control: Using Java Message Service Queues and Topics from Your Web Service.

- You can access Enterprise Java Beans (EJBs) through the EJBControl. The EJBControl simplifies your use of an EJB by providing a single component representing the EJB's interface in your web service design.

For more detail on the EJBControl, see EJB Control: Using Enterprise Java Beans from a Web Service.

- When you need to handle or control the specific shape of XML messages your service exchanges with other components, you can use XML maps. XML maps customize WebLogic Server's translation of XML to Java and vice versa.

For more on XML maps, see Handling and Shaping XML Messages with XML Maps.

- For calculations outside your Java code, and for more powerful control over the shape of XML messages, you can incorporate ECMAScript into mapping. To use ECMAScript, you write a script function in a JSX file, then refer to the function in your XML map. At run time, WebLogic Server calls the script function when translating between XML and Java.

For an introduction and details about using ECMAScript for mapping, see Using Script Functions From XML Maps.

- With the TimerControl, you can add timer functionality to your service. In this way, you can limit the time specific operations are allowed to execute, cause something to happen and regular intervals, and so on.

For more about the TimerControl, see Timer Control: Using Timers in Your Web Service.

- You can use the onException callback exposed by the JwsContext interface to handle exceptions thrown from your web service's operations. When prompted by this callback, your code can perform any necessary clean−up and send a message to the client.

For reference information about the onException callback, see JwsContext.onException Callback.

- By supporting polling, your web service can offer an alternative to clients that aren't capable of receiving callbacks. With polling, a client can periodically ask your web service if the response to its request is ready yet.

For information about polling, see Using Polling as an Alternative to Callbacks.

- As you build your web service, WebLogic Workshop generates classes that can be used by client software. Through these proxy classes, a client can invoke your service's methods. You can download the proxy classes from Test View.

For more information, see How Do I: Use the Java Proxy for a Web Service?

- You can make your web service secure by ensuring that it is exposed via the https protocol rather than http. To do this, you edit the configuration file associated with your web service's project.

For more detail, see Using HTTPS to Secure a Workshop Web Service.

- Using the JwsCompile command, you can package your web service for deployment to a production server.

For a topic on web service deployment, see Deploying Web Services. For reference information on JwsCompile, see JwsCompile Command.