WA2171 Programming Java SOAP and REST Web Services - WebSphere 8.5 - Eclipse

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Configure the Development Environment

Throughout these labs, you will be developing Java EE components using eclipse with WebSphere Application Server V8.5.

Part 1 - Configure Eclipse

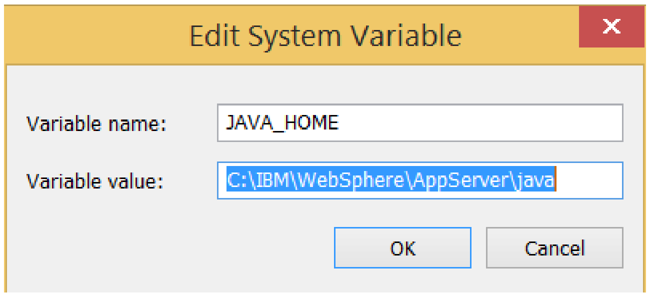
__1. In the Windows Start Menu, right-click on the Computer link in the right-hand side of the Start panel, and then select **Properties**.

__2. Click on **Advanced system settings**.

__3. The system will display the System Properties dialog. Select the **Advanced** tab and click **Environment Variables**.

__4. Double click on **JAVA_HOME**.

__5. Verify it is pointing to the IBM path as below, if not fix it. Click **OK**.



Variable name: JAVA_HOME

Variable value: C:\IBM\WebSphere\AppServer\java

__6. Close all.

__7. Open **C:\Software\eclipse**.

Troubleshooting: If eclipse was not installed at C:\Software\eclipse then open the setup and do Part 8 to install eclipse in the right place, then restart the lab.

__8. Double click **eclipse.exe** to start the software.

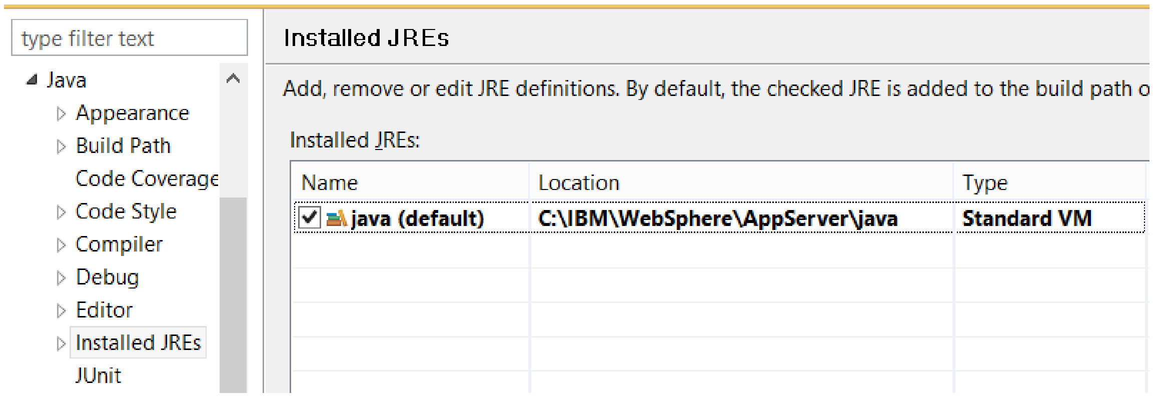
__9. Enter **C:\Workspace** as Workspace and click **Launch**.

__10. Close the Welcome page if open.

__11. From the eclipse menu, click **Windows → Preferences**

__12. Expand **Java → Installed JREs**

__13. Verify your settings are shown as below.



Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path o

Installed JREs:

Name	Location	Type
✓ java (default)	C:\IBM\WebSphere\AppServer\java	Standard VM

__14. Click **Cancel**

Part 2 - Configure WebSphere Application Server

__1. In Eclipse, select **Help** → **Install New Software...**

__2. Click **Add**.

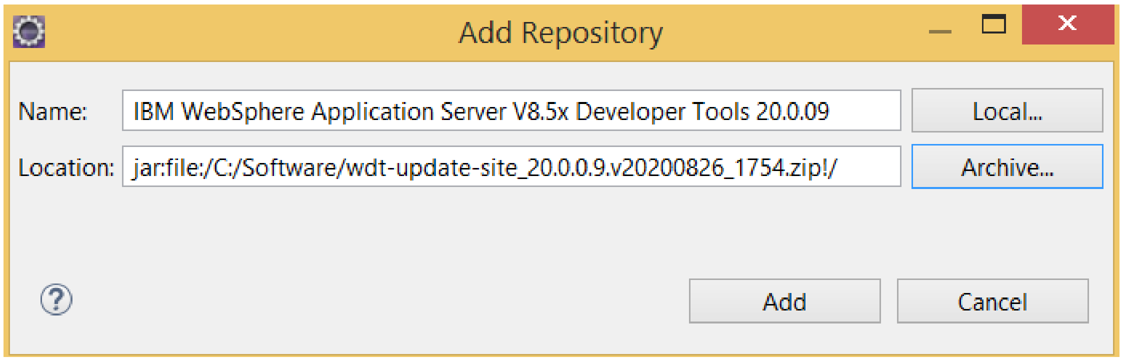
__3. Enter **IBM WebSphere Application Server V8.5x Developer Tools 20.0.09** in the **Name** field.

__4. Click **Archive...**

__5. Browse to and open the following file:

```
C:\Software\wdt-update-site_20.0.0.9.v20200826_1754.zip
```

__6. Make sure your settings are as shown below then click **Add**.



Name: IBM WebSphere Application Server V8.5x Developer Tools 20.0.09

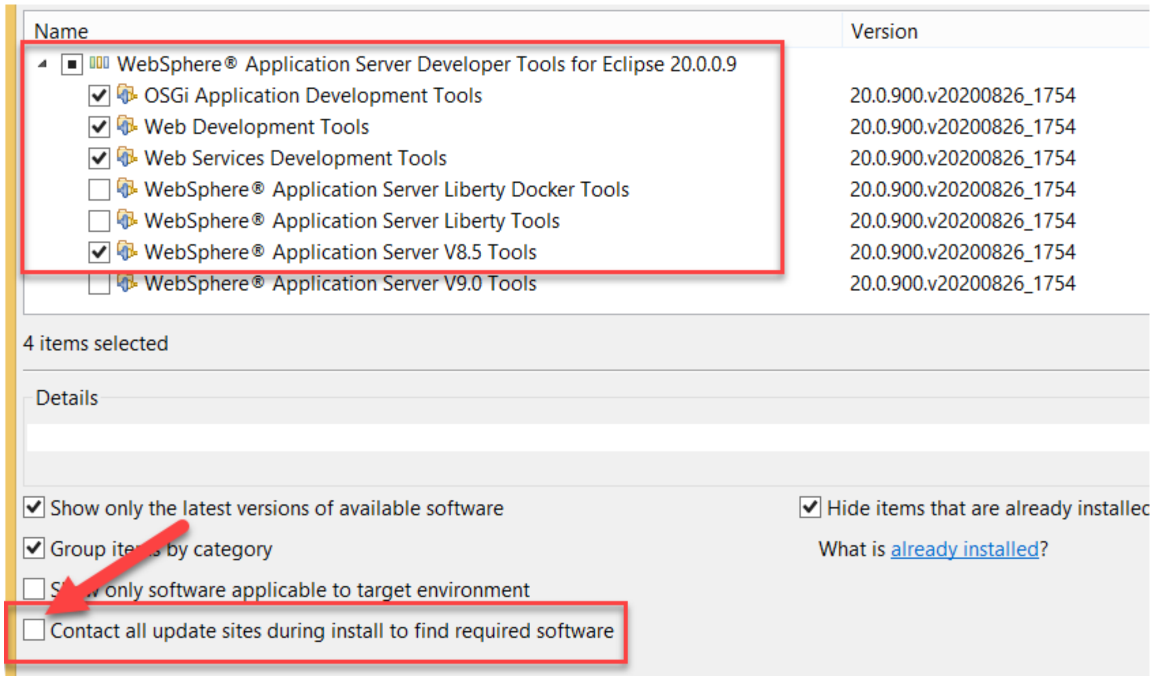
Location: jar:file:/C:/Software/wdt-update-site_20.0.0.9.v20200826_1754.zip!/

__7. Expand the top node and select the following 4 items:

- **OSGi Application Development Tools**
- **Web Development Tools**
- **Web Services Development Tools**
- **WebSphere Application Server V8.5 Tools**

__8. Uncheck **Contact all update sites during install to find required software**

__9. Make sure your settings are selected as shown below then click **Next**.



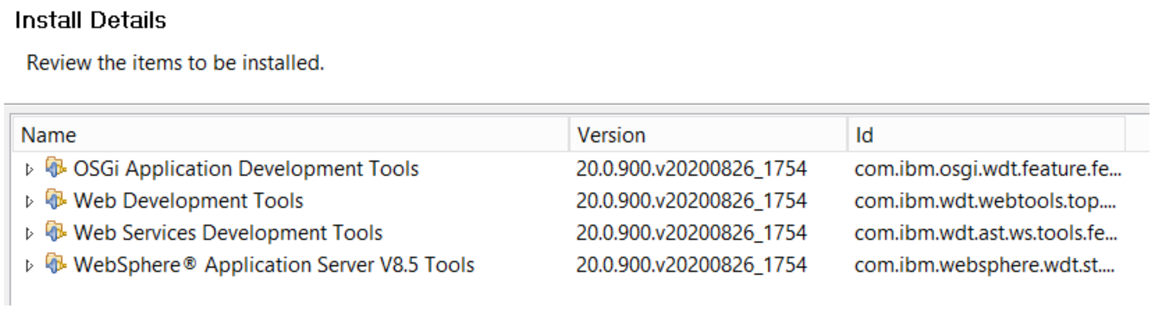
Name	Version
■ WebSphere® Application Server Developer Tools for Eclipse 20.0.0.9	
✓ OSGi Application Development Tools	20.0.900.v20200826_1754
✓ Web Development Tools	20.0.900.v20200826_1754
✓ Web Services Development Tools	20.0.900.v20200826_1754
☐ WebSphere® Application Server Liberty Docker Tools	20.0.900.v20200826_1754
☐ WebSphere® Application Server Liberty Tools	20.0.900.v20200826_1754
✓ WebSphere® Application Server V8.5 Tools	20.0.900.v20200826_1754
☐ WebSphere® Application Server V9.0 Tools	20.0.900.v20200826_1754

4 items selected

Details

✓ Show only the latest versions of available software
✓ Group items by category
☐ Show only software applicable to target environment
☐ Contact all update sites during install to find required software
✓ Hide items that are already installed
What is already installed?

__10. Verify you have the right items and click **Next**.



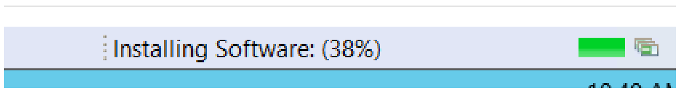
Install Details

Review the items to be installed.

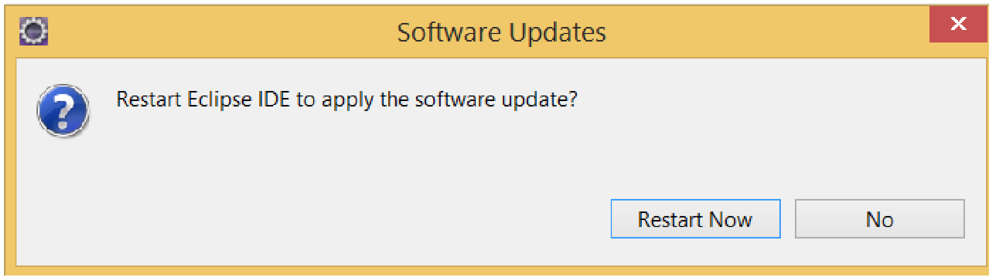
Name	Version	Id
OSGi Application Development Tools	20.0.900.v20200826_1754	com.ibm.osgi.wdt.feature.fe...
Web Development Tools	20.0.900.v20200826_1754	com.ibm.wdt.webtools.top....
Web Services Development Tools	20.0.900.v20200826_1754	com.ibm.wdt.ast.ws.tools.fe...
WebSphere® Application Server V8.5 Tools	20.0.900.v20200826_1754	com.ibm.websphere.wdt.st....

__11. Accept the license agreements and click **Finish**.

Wait until the installation is completed.



__12. You will be request to restart eclipse. Click **Restart Now**.



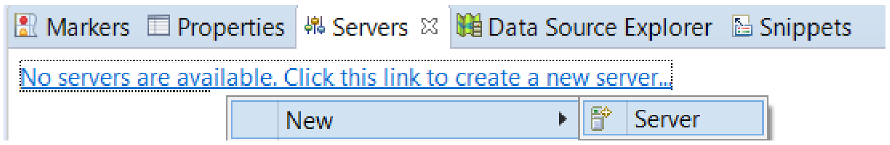
Restart Eclipse IDE to apply the software update?

__13. It may take a while to restart, close the Welcome page if open.

Part 3 - Install WebSphere Application Server

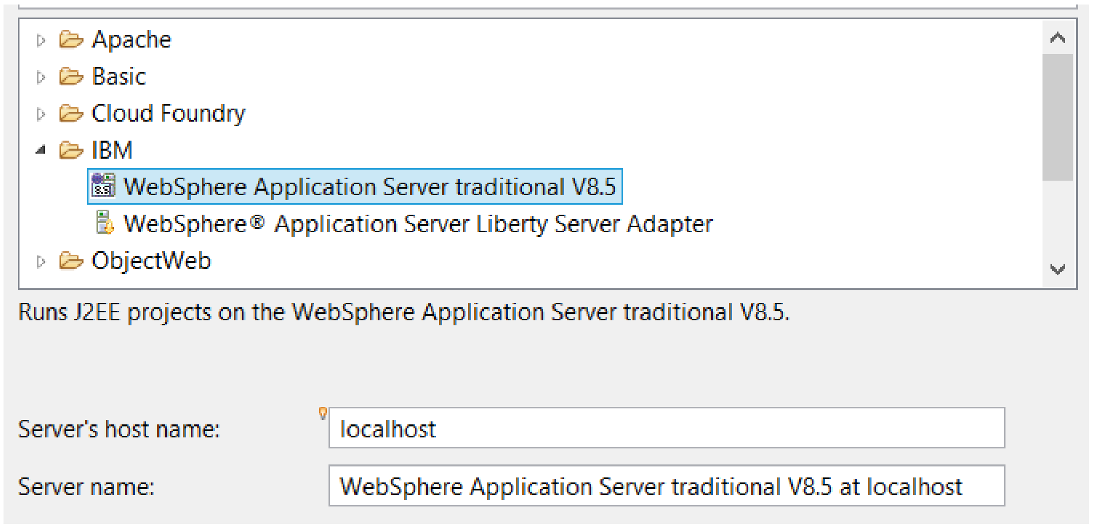
__1. Click on the **Servers** tab.

__2. Right click in the empty area and select **New** → **Server**.



No servers are available. Click this link to create a new server...

__3. Expand **IBM**, select **WebSphere Application Server traditional V8.5** and click **Next**.


Apache
Basic
Cloud Foundry
IBM
WebSphere Application Server traditional V8.5
WebSphere® Application Server Liberty Server Adapter
ObjectWeb

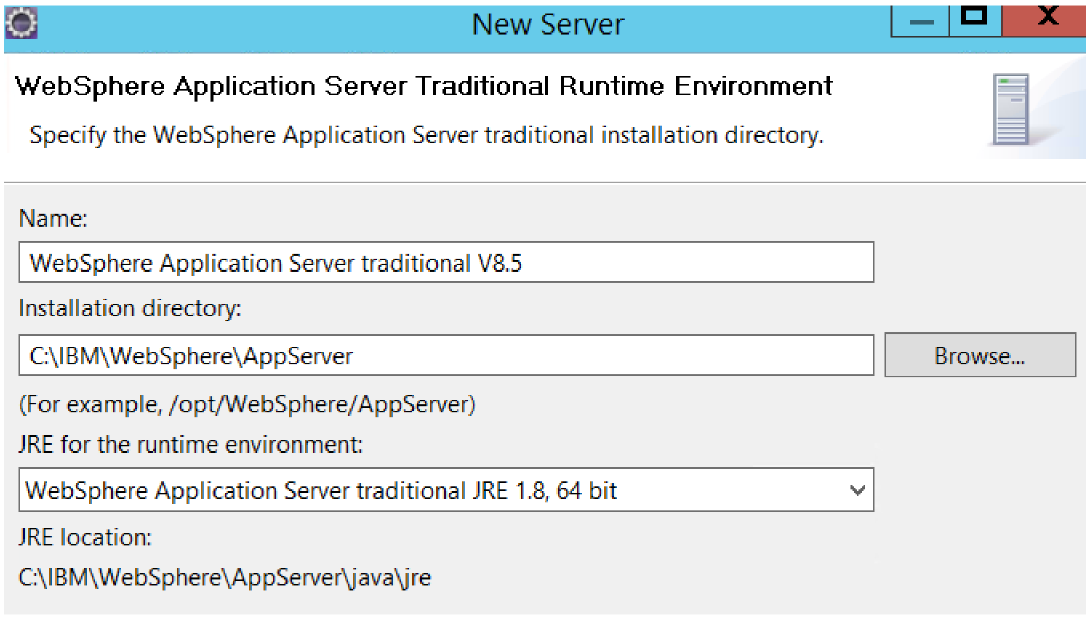
Runs J2EE projects on the WebSphere Application Server traditional V8.5.

Server's host name: localhost

Server name: WebSphere Application Server traditional V8.5 at localhost

__4. Click **Browse** and navigate to **C:\IBM\WebSphere\AppServer** and click **Select Folder**.

__5. The JRE field will be auto-populated, make sure you have the settings like below and click **Next**.



WebSphere Application Server Traditional Runtime Environment

Specify the WebSphere Application Server traditional installation directory.

Name: WebSphere Application Server traditional V8.5

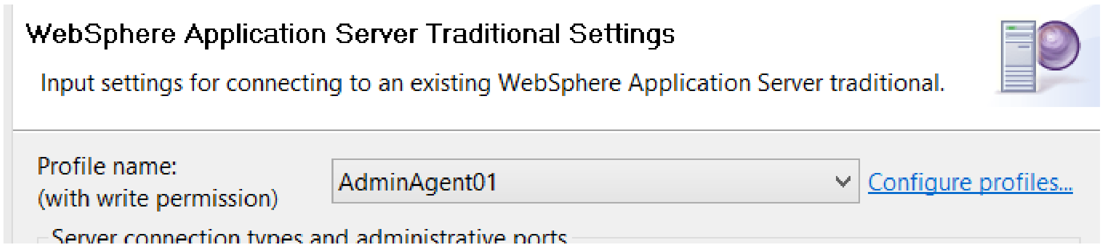
Installation directory: C:\IBM\WebSphere\AppServer

(For example, /opt/WebSphere/AppServer)

JRE for the runtime environment: WebSphere Application Server traditional JRE 1.8, 64 bit

JRE location: C:\IBM\WebSphere\AppServer\java\jre

__6. Click **Configure Profiles**.

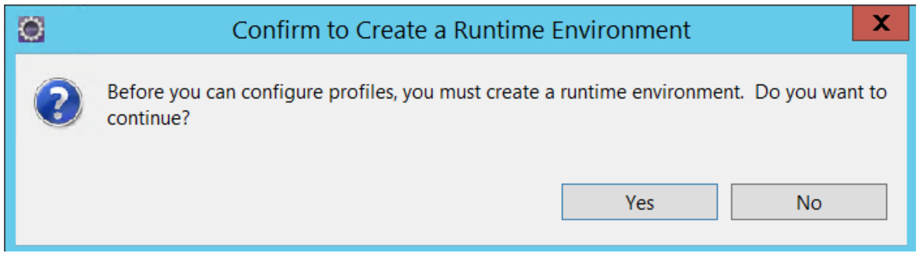

WebSphere Application Server Traditional Settings

Input settings for connecting to an existing WebSphere Application Server traditional.

Profile name: (with write permission) AdminAgent01 Configure profiles...

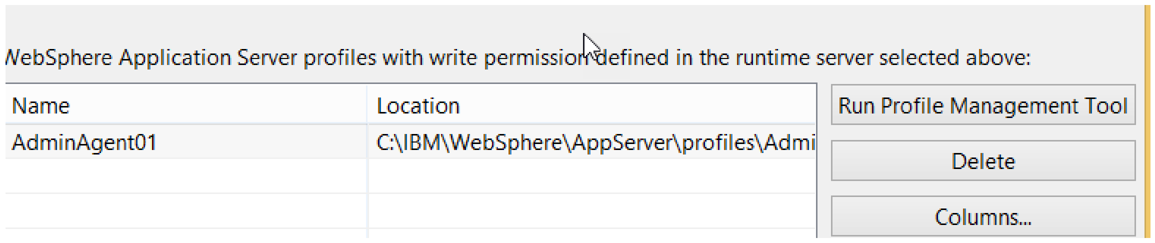
Server connection types and administrative ports

__7. You may see this screen, if so click **Yes**.


Confirm to Create a Runtime Environment

Before you can configure profiles, you must create a runtime environment. Do you want to continue?

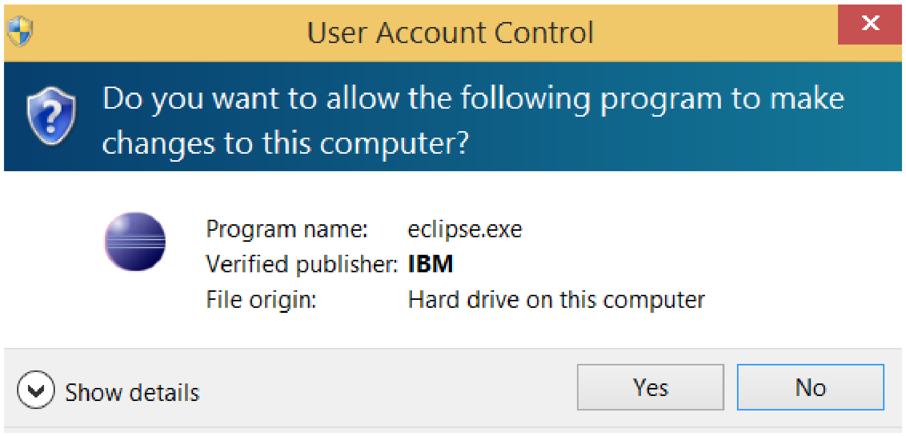
__8. Click **Run Profile Management Tool**.


WebSphere Application Server profiles with write permission defined in the runtime server selected above:

Name	Location
AdminAgent01	C:\IBM\WebSphere\AppServer\profiles\Admi

It may take a few seconds to open the next screen.

__9. Click **Yes** to continue if you see the following message.

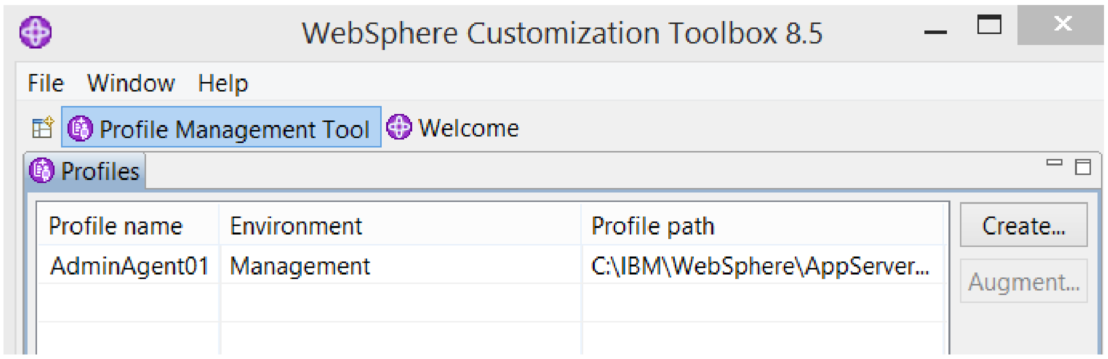


User Account Control

Do you want to allow the following program to make changes to this computer?

Program name:	eclipse.exe
Verified publisher:	**IBM**
File origin:	Hard drive on this computer

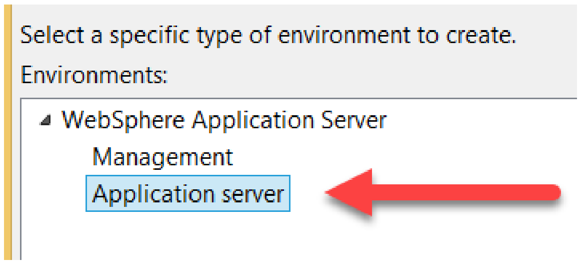
__10. Wait until the following window open, then click **Create**.



WebSphere Customization Toolbox 8.5

Profile name	Environment	Profile path
AdminAgent01	Management	C:\IBM\WebSphere\AppServer...

__11. Select **Application server** and click **Next**.



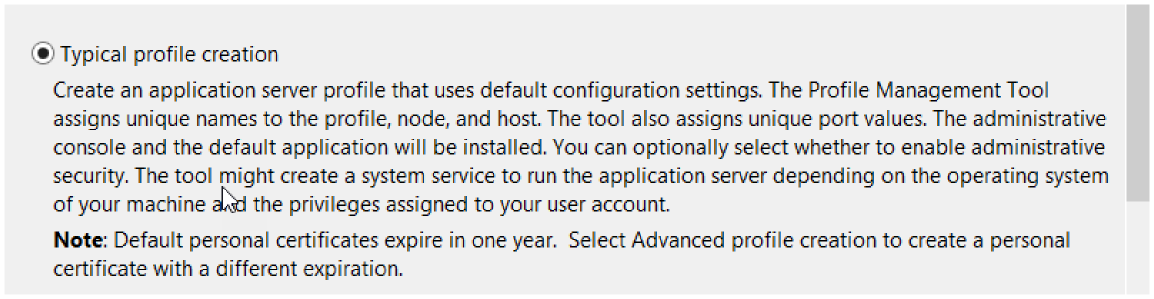
Select a specific type of environment to create.

Environments:

- WebSphere Application Server
  - Management
  - Application server

Important: Make sure to choose **Application server**. The Management profile cannot run your application code.

__12. Select **Typical profile creation** and click **Next**.

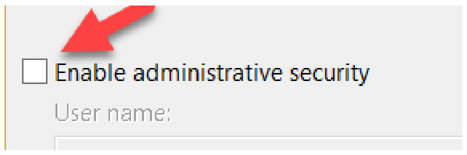


● Typical profile creation

Create an application server profile that uses default configuration settings. The Profile Management Tool assigns unique names to the profile, node, and host. The tool also assigns unique port values. The administrative console and the default application will be installed. You can optionally select whether to enable administrative security. The tool might create a system service to run the application server depending on the operating system of your machine and the privileges assigned to your user account.

Note: Default personal certificates expire in one year. Select Advanced profile creation to create a personal certificate with a different expiration.

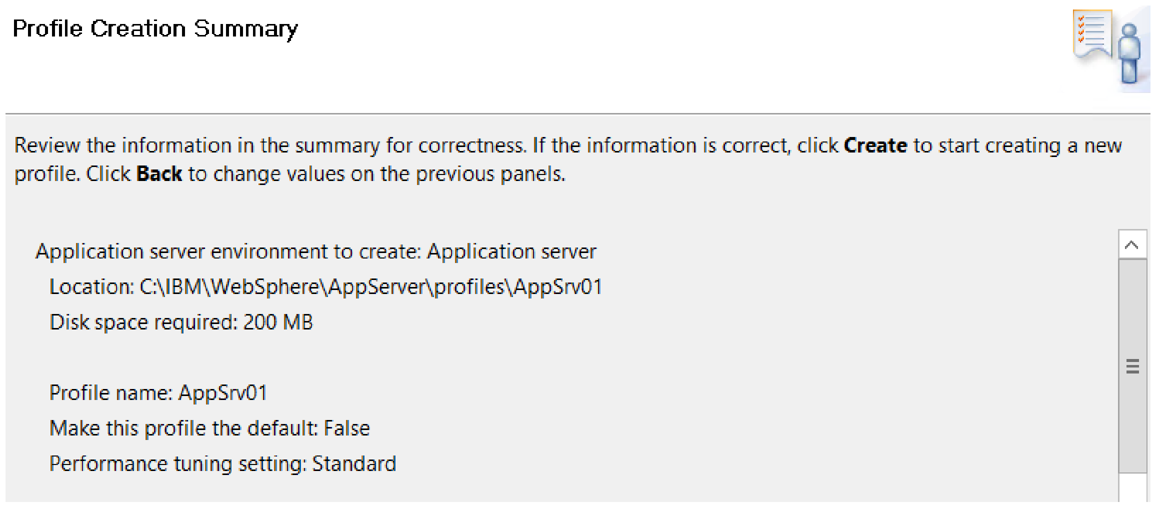
__13. Uncheck **Enable administrative security** and click **Next**.



☐ Enable administrative security

User name:

__14. Click **Create**.



Profile Creation Summary

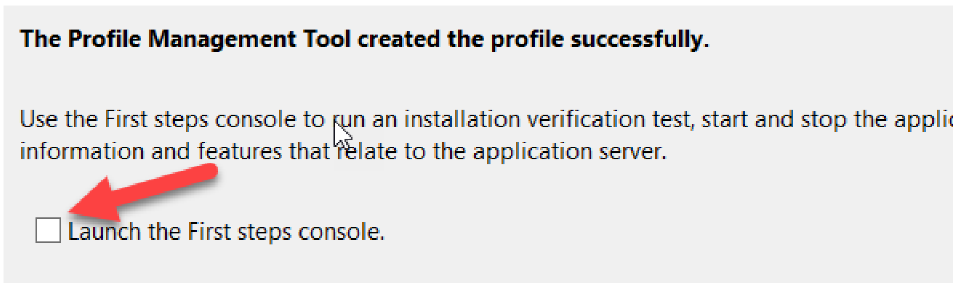
Review the information in the summary for correctness. If the information is correct, click **Create** to start creating a new profile. Click **Back** to change values on the previous panels.

Application server environment to create: Application server
Location: C:\IBM\WebSphere\AppServer\profiles\AppSrv01
Disk space required: 200 MB

Profile name: AppSrv01
Make this profile the default: False
Performance tuning setting: Standard

It will take some time to create the server.

__15. Uncheck **Launch the first steps console** and click **Finish**.

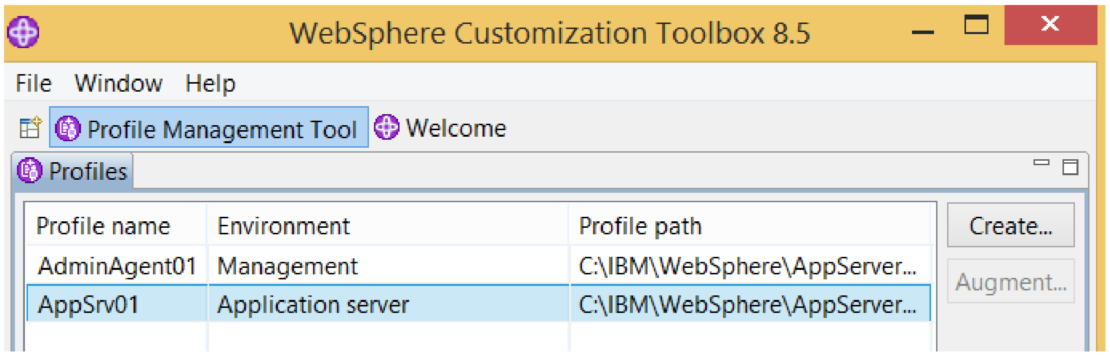


The Profile Management Tool created the profile successfully.

Use the First steps console to run an installation verification test, start and stop the applic
information and features that relate to the application server.

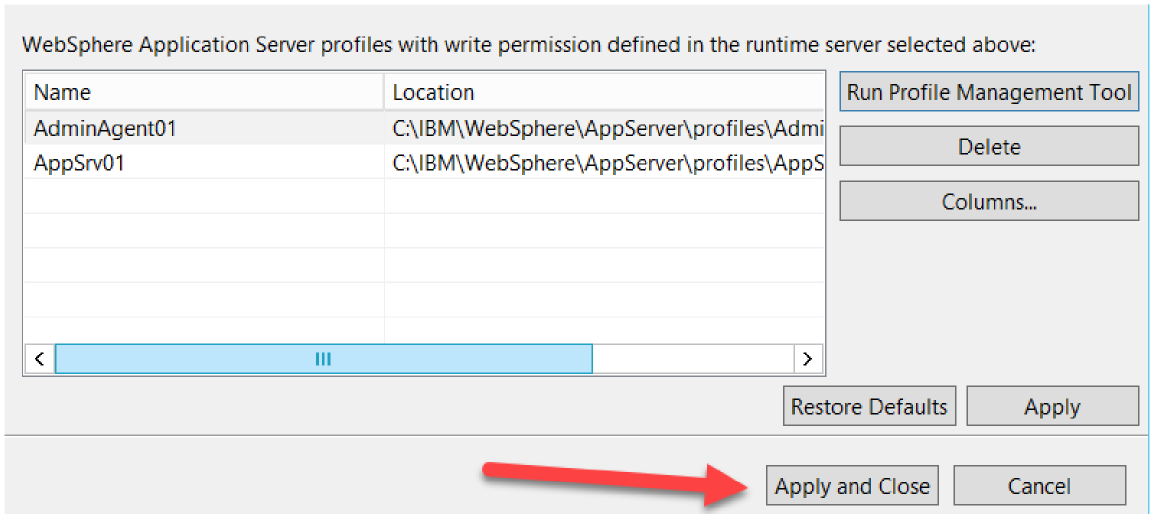
☐ Launch the First steps console.

__16. Close the **Profile Management Tool** window.



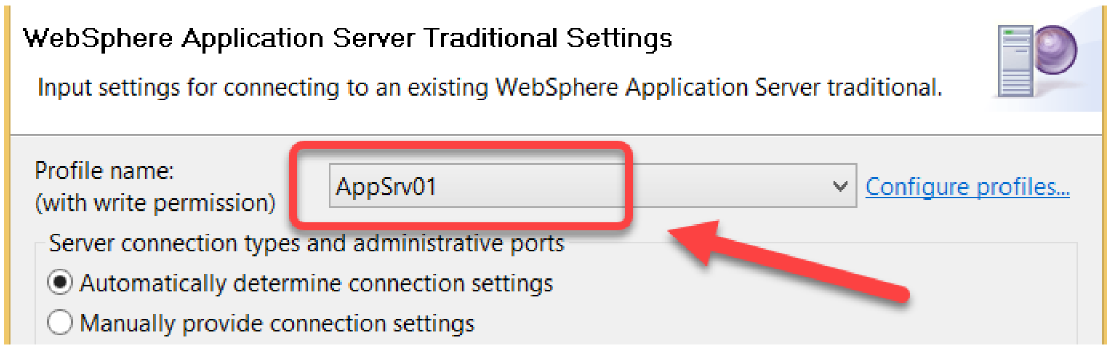
Profile name	Environment	Profile path
AdminAgent01	Management	C:\IBM\WebSphere\AppServer...
AppSrv01	Application server	C:\IBM\WebSphere\AppServer...

__17. Click **Apply and Close**.


WebSphere Application Server profiles with write permission defined in the runtime server selected above:

Name	Location
AdminAgent01	C:\IBM\WebSphere\AppServer\profiles\Admi
AppSrv01	C:\IBM\WebSphere\AppServer\profiles\AppS

__18. Now from the profile name drop down, select the **AppSrv01** that you just created.


WebSphere Application Server Traditional Settings

Input settings for connecting to an existing WebSphere Application Server traditional.

Profile name:
(with write permission) AppSrv01

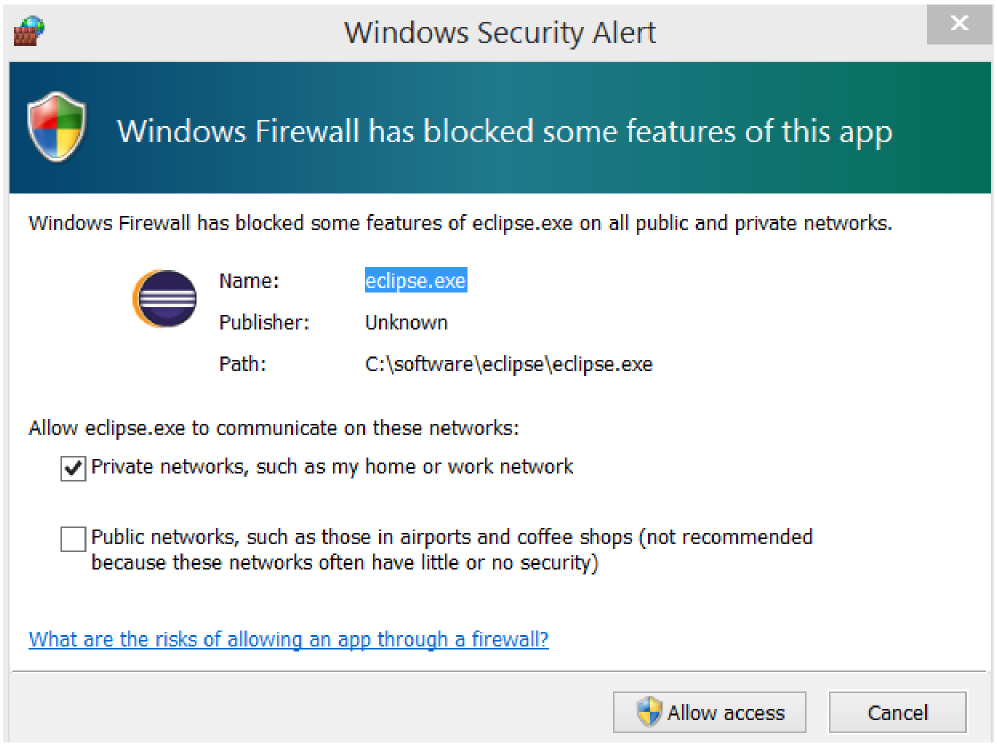
Server connection types and administrative ports

(●) Automatically determine connection settings

( ) Manually provide connection settings

__19. Click **Finish**.

__20. You may see the following window, click **Allow access**


Windows Security Alert

Windows Firewall has blocked some features of this app

Windows Firewall has blocked some features of eclipse.exe on all public and private networks.

Name:	eclipse.exe
Publisher:	Unknown
Path:	C:\software\eclipse\eclipse.exe

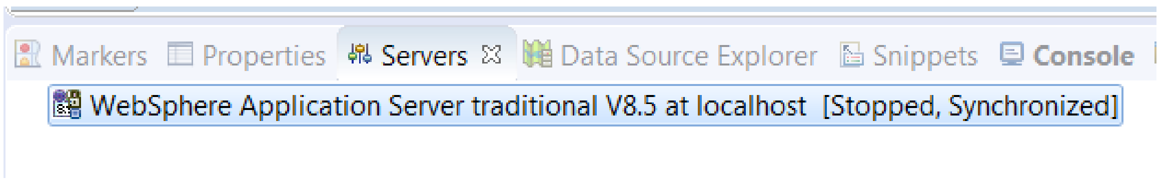
Allow eclipse.exe to communicate on these networks:

[✓] Private networks, such as my home or work network

[ ] Public networks, such as those in airports and coffee shops (not recommended because these networks often have little or no security)

What are the risks of allowing an app through a firewall?

__21. The server will appear in the Server tab.



WebSphere Application Server traditional V8.5 at localhost [Stopped, Synchronized]

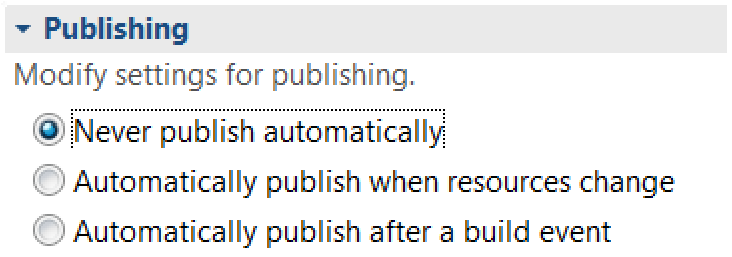
During the Labs we will refer to the "**WebSphere Application Server traditional V8.5 at localhost"** simply as **the Server**.

By default, eclipse will publish your work to the server every time you edit and save a file. Although this sounds like a good idea, in practice, it leads to a continuous stream of specious warnings and errors, not to mention delays. We will shut off this behavior in the server we just created.

__22. Double-click on **WebSphere Application Server traditional V8.5 at localhost**.

__23. Eclipse will display the server configuration window. Look for the **Publishing** section and click on it to expand it.

__24. Click the radio button for **Never publish automatically**.



Publishing

Modify settings for publishing.

[✓ Never publish automatically]
Automatically publish when resources change
Automatically publish after a build event

__25. Save and close the file.

Part 4 - Working with the Server

Throughout this class, we will be using this view to perform 3 basic operations: server *stop*, *start* and *publish*. We will discuss publishing in a later lab, right now let's focus on starting and stopping the server.

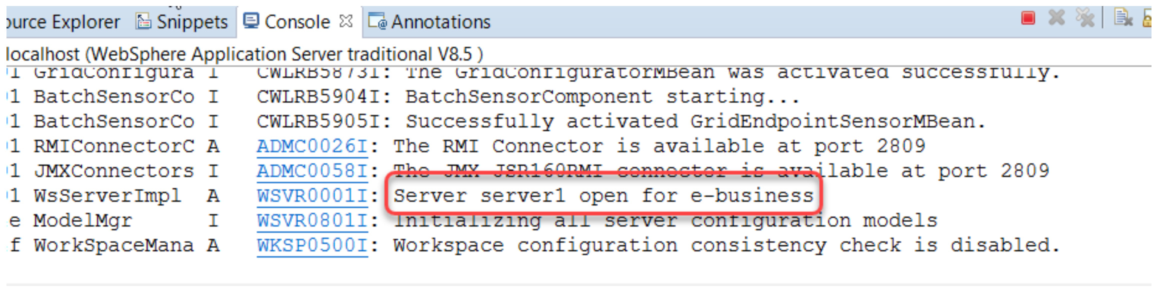
__1. Let us start things off by starting the server. In the *Servers* view, right click on the Server and then select **Start**.

__2. During this process you may be requested to allow access to continue, do it.

It will take some time to start the server for the first time.

The *Console* view should automatically open after a minute. This will allow us to "see" WebSphere booting. Eventually, the server will start.

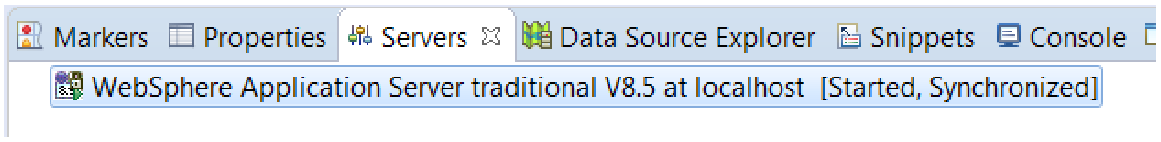
__3. Examine the **Console** window. You should see the following:



```
localhost (WebSphere Application Server traditional V8.5 )
1 GridConfigura I   CWLRB5873I: The GridConfiguratorMBean was activated successfully.
1 BatchSensorCo I   CWLRB5904I: BatchSensorComponent starting...
1 BatchSensorCo I   CWLRB5905I: Successfully activated GridEndpointSensorMBean.
1 RMIConnectorC A   ADMC0026I: The RMI Connector is available at port 2809
1 JMXConnectors I   ADMC0058I: The JMX JSR160RMI connector is available at port 2809
1 WsServerImpl  A   WSVR0001I: Server server1 open for e-business
e ModelMgr      I   WSVR0801I: Initializing all server configuration models
f WorkSpaceMana A   WKSP0500I: Workspace configuration consistency check is disabled.
```

In later lab steps, when we ask you to refer to the *server console*, this is the view we will be referring to.

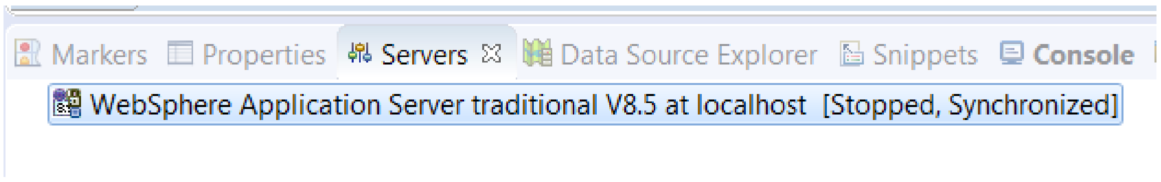
__4. Click the **Servers** tab. The server *State* should be **Started, Synchronized**.



WebSphere Application Server traditional V8.5 at localhost [Started, Synchronized]

__5. Let us stop the server; right click on the server and select **Stop**.

As with starting the server, the console view will open briefly. Then the *Server* view will appear, listing the status as *Stopped*.



WebSphere Application Server traditional V8.5 at localhost [Stopped, Synchronized]

__6. Leave eclipse open.

Part 5 - Review

In this lab, you configured the development environment.

You started things off by configuring environment to recognize the WebSphere Application Server, which will be used to host our web service applications.

You saw how to start and stop a WebSphere Server instance.

Lab 2 - Introduction To Web Services

In this lab, you will review some of the key components of Web Services, namely, WSDL and SOAP. You will also get a chance to see a Web Service in action. The goal of this lab is to get yourself familiar with what a Web service looks like. We will not get into too much detail at this point; just walk away with a basic understanding of the components involved in a web service.

The web service in question is very simple. It has one operation called **sayHello**. It takes as input a person's name (say "John") and returns as output a greeting message (say, "Hello John").

Part 1 - Import the Projects

To save time, the web service is already implemented. You will simply import it into Eclipse for examination.

You should be in the *Java EE* perspective, from which we will do the bulk of our work.

The next step is to import the projects into the workspace.

__1. Right-click anywhere inside the *Enterprise Explorer* view (the pane on the left) and select **Import | Import...**

__2. The *Import* screen will appear. In the list of import sources, expand **General** and select **Existing Projects into Workspace**.

__3. Click **Next**.

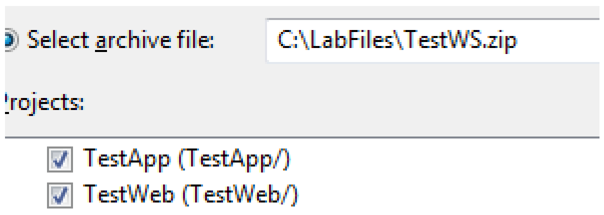
The *Import Projects* screen will appear.

__4. Select the radio button for **Select archive file**.

__5. Next to it, click the **Browse...** button.

__6. Select the **C:\LabFiles\TestWS.zip** file and click **Open**.

The list of *Projects* will automatically be populated, as shown here:



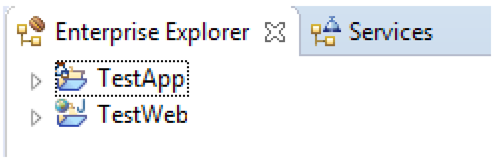
Select archive file: C:\LabFiles\TestWS.zip

'rojects:

- ☑ TestApp (TestApp/)
- ☑ TestWeb (TestWeb/)

__7. Click **Finish**.

The projects will be imported into the workspace and will be listed in the *Enterprise Explorer* view.



The two projects imported in the workspace are:

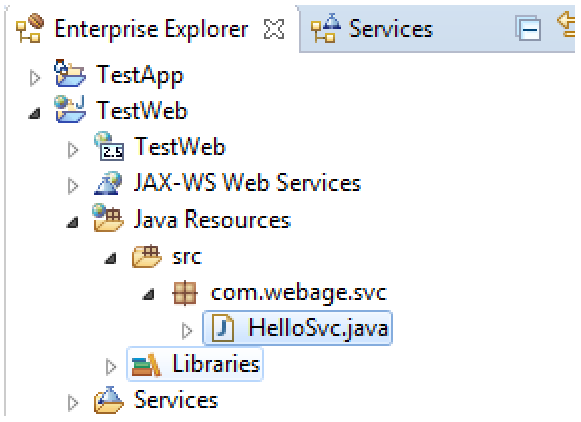
1. **TestApp** – The enterprise application project. This represents the EAR file in Java EE.
2. **TestWeb** – The web module project. This represents a WAR file in Java EE

Our projects are imported. Let us take a look at them.

Part 2 - Inspect the Implementation Class

The business logic for a web service is implemented in a Java class. The Java class may be a plain class or a session EJB. In this case, we used a plain Java class in a web module. We will briefly have a look at it.

__1. In the *Enterprise Explorer* view, expand the **TestWeb** project as shown below.



__2. Double click **HelloSvc.java** to open it in a Java source code editor.

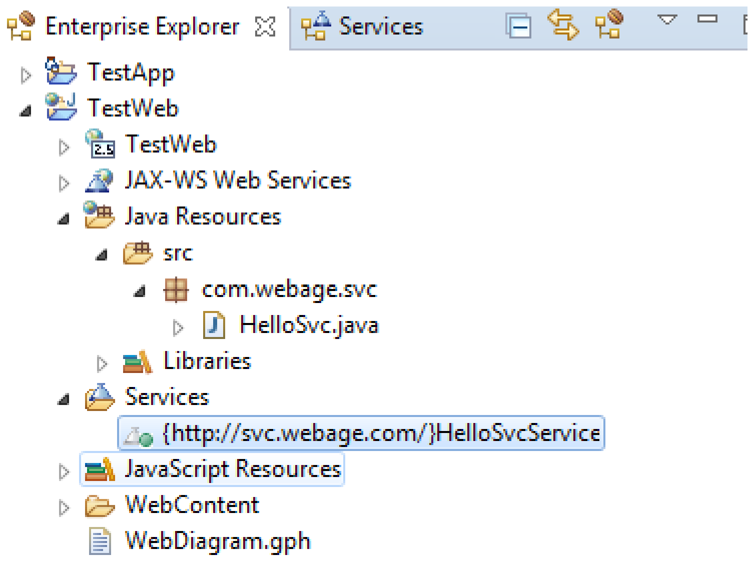
```
package com.webage.svc;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class HelloSvc {

          @WebMethod
          public String sayHello(String name) {
                     return "Hello " + name;
          }
}
```

__3. Examine the class. Note the following aspects of the code:

- By default, the Java method that implements a Web Service operation has the same name as the operation.
- The **@WebService** annotation is used to designate the **HelloSvc** class as a Web Service implementation.
- The **@WebMethod** annotation is used for the **sayHello**() method to designate it as a Web Service operation. These annotations are made available as a part of the JAX-WS specification.

__4. Close the Java editor, by clicking the **X** in its tab.

__5. Back in the *Enterprise Explorer*, notice the ***Services*** node of the **TestWeb** project. Expand it.



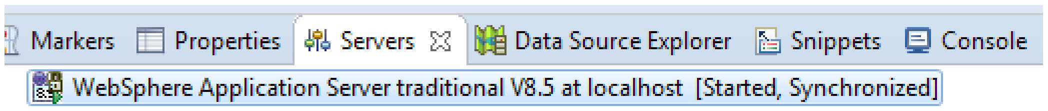
This *Services* node lists all web services located in this web project. Notice that it has located the HelloService service, and is listing it here accordingly, as **{http://svc.webage.com/}HelloSvcService.** This is a good sign that our annotations are correct.

Remember this node; we will be returning to it frequently. We can now test the service!

Part 3 - Deploy the Project to WebSphere Application Server (WAS)

Before we can test the service, we first need to deploy it to WAS. We will do this now.

__1. Locate the *Servers* view. If you server is not started then right click on it and select **Start**.



WebSphere Application Server traditional V8.5 at localhost [Started, Synchronized]

The server listed here is the one we will be using for all our deployments for the duration of this course. We now need to add our **TestApp** project to it. (Recall that the **TestApp** is an EAR file which contains **TestWeb**, which in turn contains the service).

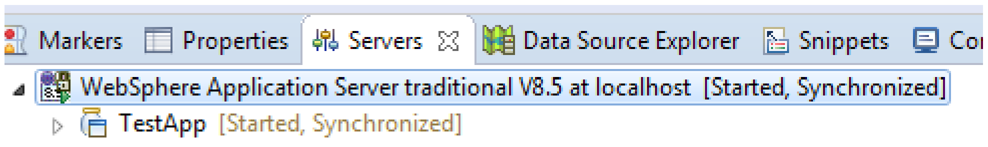
__2. Right-click on the server listed and select **Add and Remove...**

__3. The *Add and Remove Projects* window will appear. In the left pane, select **TestApp** and click the **Add>** button. This will move it to the right pane.

__4. Click **Finish**.

Remember these steps as you will be deploying projects to WAS repeatedly throughout this course.

__5. The project will be added to the server and will start it too, this make take some time and the Console should display some messages, wait until the Console finish and then click on the **Servers** tab and verify your application has been added, Started and Synchronized. If not, then right click on the Server and select **Publish**.



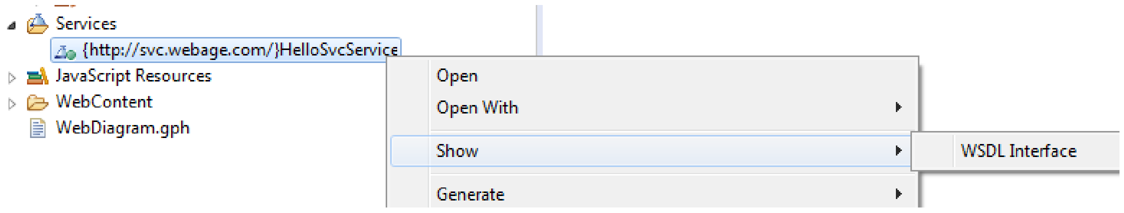
WebSphere Application Server traditional V8.5 at localhost [Started, Synchronized]
TestApp [Started, Synchronized]

A *Synchronized* state means all the latest versions of the code in Eclipse have been deployed to the server. This means we are able to test our code!

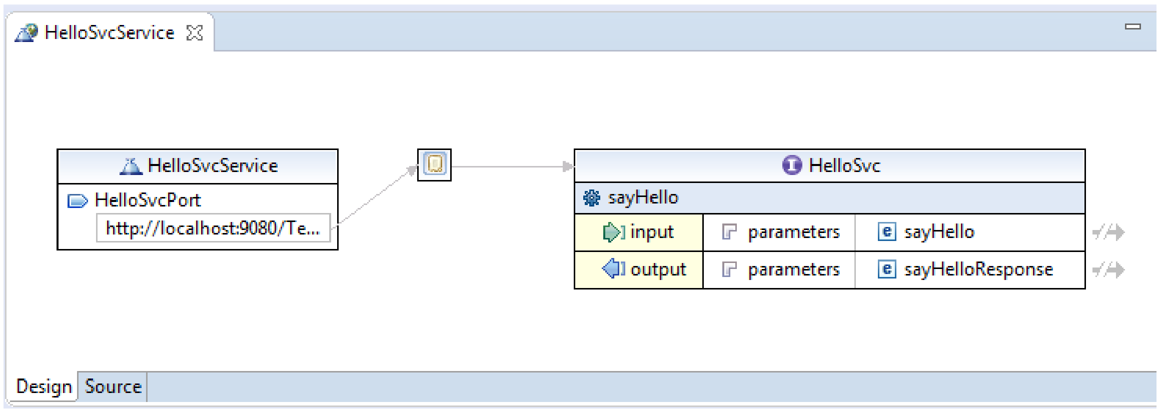
Part 4 - Examine the WSDL File

A WSDL file is used to describe what a web service can do and what type of input and output data it works with. Let us examine the WSDL file for our service.

__1. In the *Enterprise Explorer* view, right click on the service beneath the ***Services*** node (in the **TestWeb** project) and select **Show | WSDL Interface**.



__2. Eclipse will open the *WSDL Editor*, Click the **Design** tab at the bottom which shows a diagram of the service's WSDL file.



__3. This looks daunting, but we will return to it in more detail in a later lab. For now, click the **Source** tab at the bottom of the WSDL editor.

The raw XML source code of the WSDL editor will be visible.

To be honest, this looks even more frightening than the diagram view. Do not worry about it for now, though.

__4. Locate the **<portType>** element of the file. This describe the Web Service. Note that an **<operation>** element exists with the *name* attribute set to "**sayHello**".

```
<portType name="HelloSvc">
    <operation name="sayHello">
      <input message="tns:sayHello" wsam:Action="http://svc.webage.com/
HelloSvc/sayHelloRequest">
    </input>
      <output message="tns:sayHelloResponse"
wsam:Action="http://svc.webage.com/HelloSvc/sayHelloResponse">
    </output>
    </operation>
  </portType>
```

__5. Notice the **<input>** and **<output>** elements within **<operation>.** They describe the format of the input and output data for the operation. We won't go any further to explore the WSDL at this stage, but will return to it in a later lab.

__6. Close the WSDL editor.

Part 5 - Invoke the Service

We should now actually test the service. Testing the service would imply actually invoking the operation.

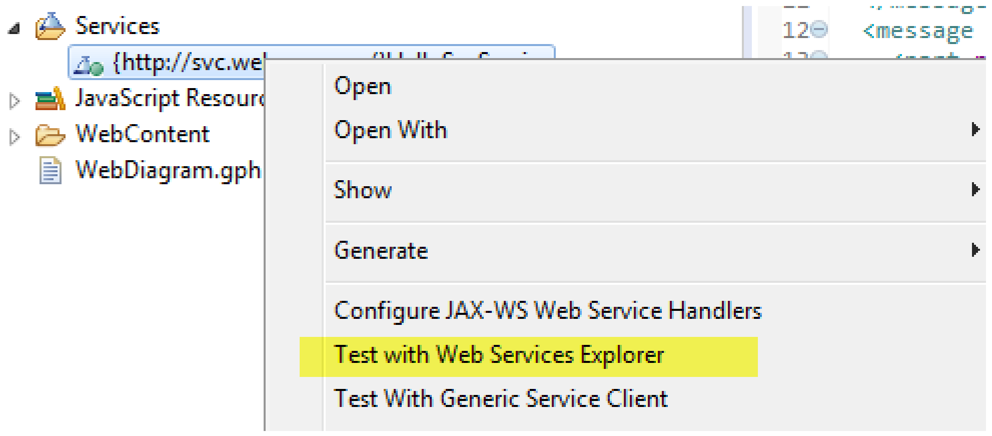
Typically, doing so would require that we create a web service client. A web service client is, as its name implies, any program that makes a call to a web service. The program could be written in Java, .NET, or any other language that understands web service standards.

Writing a client can be time consuming, so we will “cheat” and use a tool built into Eclipse called the *Web Services Explorer* (WSE). The WSE works by reading a WSDL file and then figuring out what services are available and how to invoke them. The WSE will then dynamically generate a form that represents the service's operations that will allow you to invoke it.

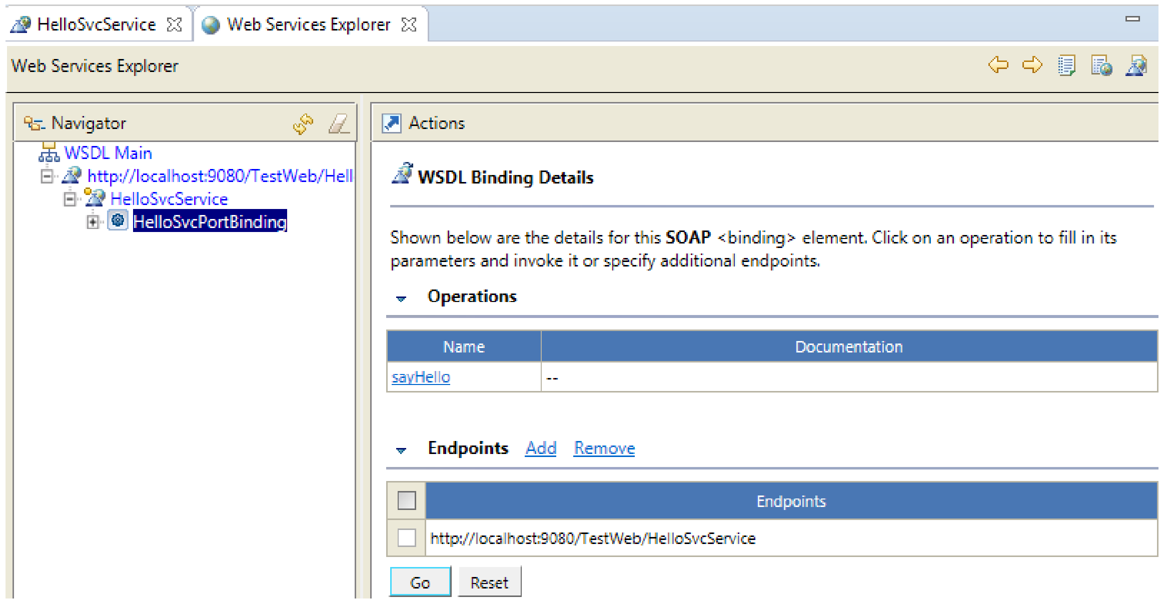
Think about this for a moment: our service, at its core, was a single operation called **sayHello** which took a single string as a parameter. The service would then return a greeting based on that parameter.

Let us try this now.

__1. In the *Enterprise Explorer*, under **TestWeb**, right-click the service (under **Services**) and select **Test with Web Services Explorer**.


Services
{http://svc.we
JavaScript Resour
WebContent
WebDiagram.gph
Open
Open With
Show
Generate
Configure JAX-WS Web Service Handlers
Test with Web Services Explorer
Test With Generic Service Client

The WSE will open.



Shown below are the details for this **SOAP** <binding> element. Click on an operation to fill in its parameters and invoke it or specify additional endpoints.

Operations

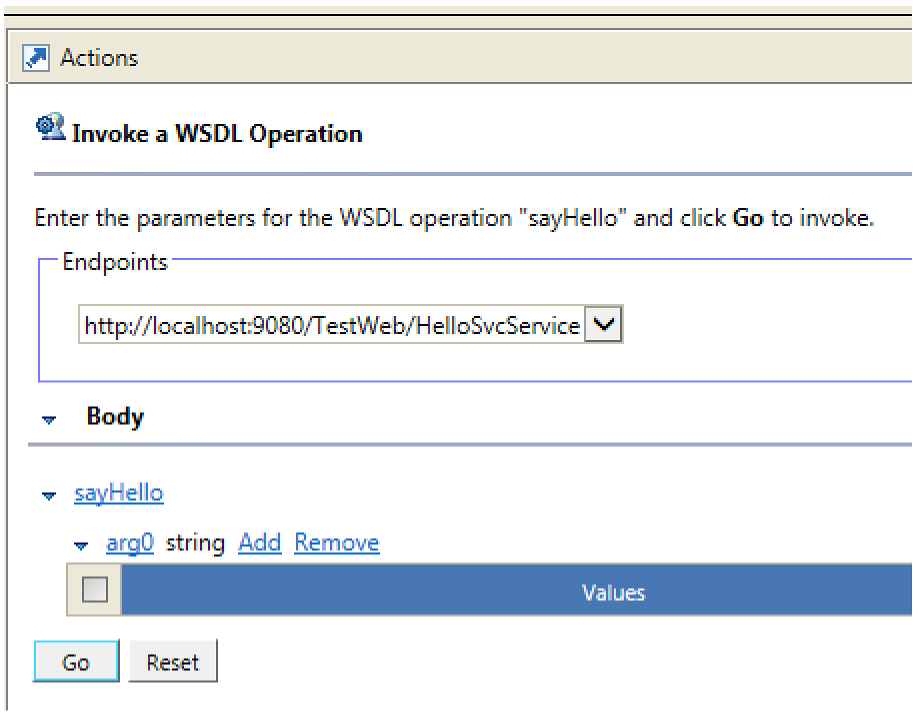
Name	Documentation
sayHello	--

Endpoints Add Remove

	Endpoints
☐	http://localhost:9080/TestWeb/HelloSvcService

Notice that it has located our service and has noticed the **sayHello** operation, which is listed in the right pane.

__2. Click the link for **sayHello**. The WSE will create and display a form to enter the input data for the **sayHello** operation.

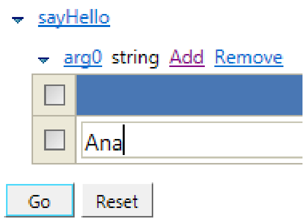


Enter the parameters for the WSDL operation "sayHello" and click **Go** to invoke.

Endpoints: http://localhost:9080/TestWeb/HelloSvcService

The WSE realizes that the operation requires a parameter, and the form presents a location to add one.

__3. Click the **Add** link and enter a name.

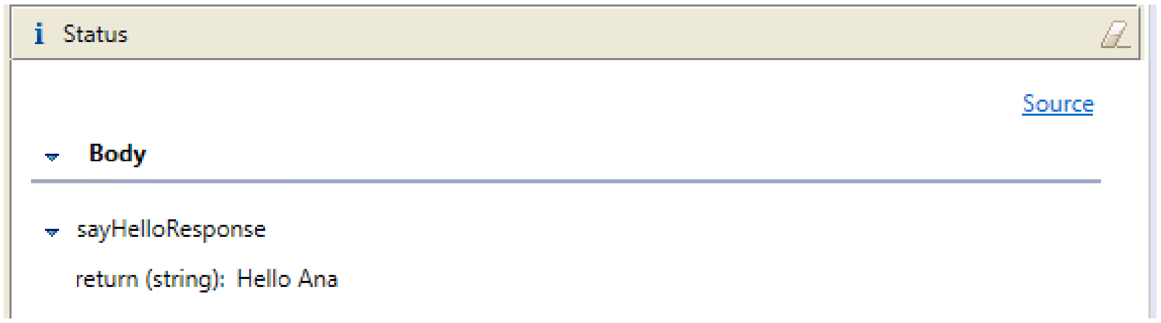


Ana

__4. Click **Go**.

The WSE will take the name you entered and submit it to the service.

The *Status* pane should now appear, immediately below the pane in which you entered the argument.



sayHelloResponse
return (string): Hello Ana

This shows what the call to the web service returned. In this case, it will show **Hello [name]**

Our service works! We have successfully tested our service with the WSE.

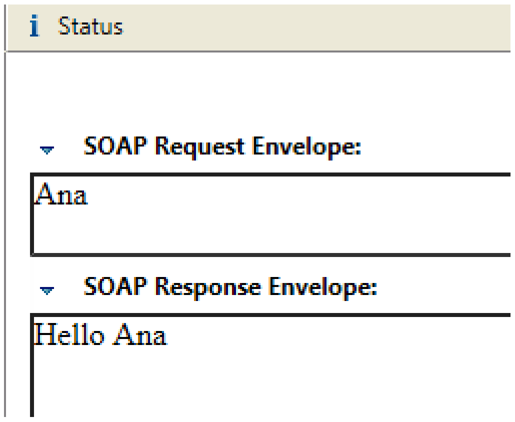
Part 6 - Inspect the SOAP Messages

When we invoked the sayHello operation, WSE actually sent a SOAP formatted XML document to the server using the HTTP protocol. The server also replied back using a SOAP formatted XML document. SOAP is a versatile data format for request response type communication. In its most basic form, as is the case in this lab, it is utterly simple. At the same time, it is capable of taking on more advanced aspects of communication, such as, encryption, message signature, transaction and guaranteed delivery.

__1. In the **Status** pane of WSE, click the **Source** link.

The raw underlying XML SOAP messages will be displayed.

__2. The windows are probably too small to see, however; double-click in the pane header (where it says *Status*) to maximize the window.



Ana

Hello Ana

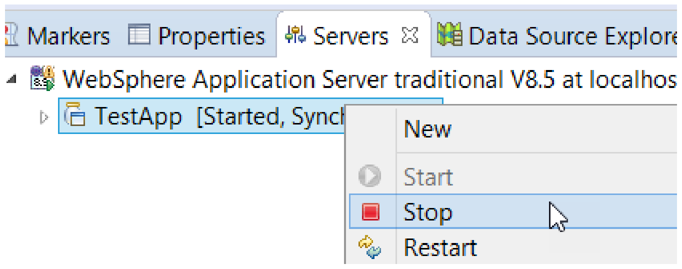
__3. Restore the Status window to the original size.

Congratulations! You have successfully examined the source code for a web service, and then invoked it via the WSE.

__4. Close the WSE by clicking the **X** in its pane.

__5. Close all open windows by selecting CTRL+Shift+W.

__6. Right click on the **TestApp** and select **Stop**.



This will stop the application but leave the server running.

Part 7 - Review

In this lab, you had a brief encounter with a web service. First, we looked at the Java class that implements the business logic of the Web Service. That is the true meat behind the Web Service. When building a Web Service, you will probably spend most of the time writing the business logic. Next, we inspected the WSDL file that describes the nature of the Web Service. After that we tested the Web Service and inspected the SOAP messages.

The lab also served two other purposes. You learned how to deploy an application to the server. We will do that frequently throughout these lab exercises. You also found out how to use the Web Services Explorer to quickly unit test a Web Service.

Lab 3 - Creating An XML Schema

An XML schema is used to verify that the contents of an XML file are valid; essentially, it is a set of “rules” for an XML file. A particular XML file can be checked against a particular schema, and a parser can decide if the XML conforms to the rules set out in the schema.

If, for some reason, the XML does not conform to the schema, the XML parser can reject the XML and raise an immediate error. Essentially, schema adds a layer of validation to any XML-based application. Given that web services use XML as their communication layer, we can see the value of schema. Schema are actually used quite frequently in the various layers of web service communication.

In this lab, we will create an XML schema.

Part 1 - The Movie.xml File

Throughout these labs, we will be developing a system that deals with movies. We have thus decided to model a movie as an XML file. This modeling will come in handy when we start building the web service.

__1. Let us start things off by taking a look at a sample movie.xml file. Using a text editor, open file **C:\LabFiles\movie.xml**

It looks like the following:

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

<m:database xmlns:m="http://www.mycom.com/movie">
          <movie title="Preaching to the Choir" rating="PG13">
                    <description>
                    Two estranged brothers -- one a hip-hop star and
                    the other a Baptist preacher -- find a way
                    to make amends and help one another out as
                    they each face a crisis in their lives.
                    </description>
                    <actor>Billoah Greene</actor>
                    <actor>Novella Nelson</actor>
          </movie>
          <movie title="Kekexili: Mountain Patrol" rating="PG13">
                    <description>
                    A moving true story about volunteers protecting
                    antelope against
                    poachers in the severe mountains of Tibet.
                    </description>
                    <actor>Zhang Lei</actor>
                    <actor>Qi Liang</actor>
          </movie>
</m:database>
<!-- Contains some copyrighted material from imdb.com -->
```

Let us analyze this XML file:

- The outermost element is called **<database>** and it contains multiple **<movie>** elements.
- Each **<movie>** has two attributes; one called **title** and one called **rating**, which reflects a movies title and rating respectively.
- Additionally, each <**movie**> has a single child element called <**description**> (which contains a description of the movie in question) and *multiple* elements called <**actor**>, with each one representing an actor in the associated movie.
- This is a fairly simple modeling of the movie. We capture some important information about each movie (the title, the description, the rating and the actors), and store this information using a mixture of attributes and child elements.
- What we should do now is formalize this model by creating an schema for the **<database>** element (remember that the root element here is **<database>**, not **<movie>)**.
- Once that is done, an application reading/writing one of these movie XML files will be able to use the schema to validate that the structure of the XML is good. (Validating the **content** of the XML, however, is another matter completely).

__2. Close the text editor.

Part 2 - Create the Project

A schema is itself an XML file, and we could create it using a text editor. However, Eclipse provides some nice tools for creating a schema, so we will use those instead.

__1. We will now create a new project to hold our schema file. From the Eclipse menu bar, select **File | New | Project...**

__2. The *New Project* window will appear. In the list, select **Java Project** and click **Next**.

__3. The *New Java Project* screen will appear. Set the *Project name* to be **XMLProject** and click **Finish**.

__4. Eclipse will then prompt you if you would like to switch to the *Java Perspective*. Check **Remember my decision** and click **No**.

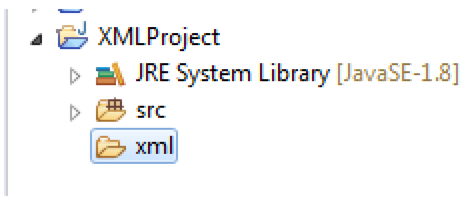
The **XMLProject** project will be created and listed in the *Enterprise Explorer* view. (You may want to minimize all other projects – e.g. **TestWeb** and **TestApp** in the *Enterprise Explorer* to keep things a little easier to read).

We should create a folder to hold our XML resources.

__5. Right-click on the newly created **XMLProject** and select **New | Folder**.

__6. The *New Folder* window will appear. Set the *Folder name* to be **xml** and click **Finish**.

The folder will be created, and should be visible under the project.



Part 3 - Create the Schema

We can now create the schema.

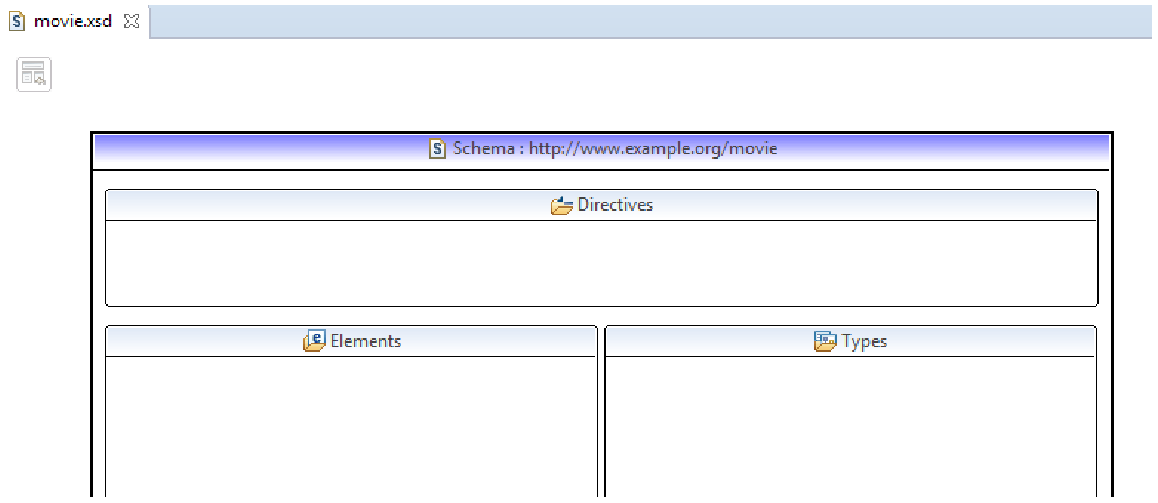
__1. Right click on the newly created **xml** folder and select **New | Other...**

__2. The *New* window will appear. Scroll down and expand **XML**.

__3. Select **XML Schema File** and click **Next**.

__4. The *Create XML Schema* window will appear. Set the *File name* to be **movie.xsd** and click **Finish**. (By convention, schemas always have an XSD extension)

The *XML Schema Editor* will open. This is Eclipse's graphical XSD schema editor. This will save us having to create the schema by hand.



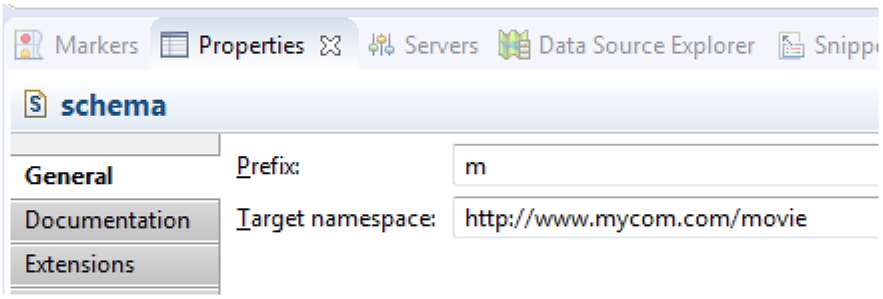
First, we should change the schema's default name space which is currently set to **http://www.example.org/movie** as can be seen at the top of the editor.

__5. Right-click on this area and select **Show properties**.

Now, at the bottom of Eclipse, the *Properties* view should have opened.

__6. Here, change the *Prefix* to **m**

__7. Change the *Target namespace* to **http://www.mycom.com/movie**



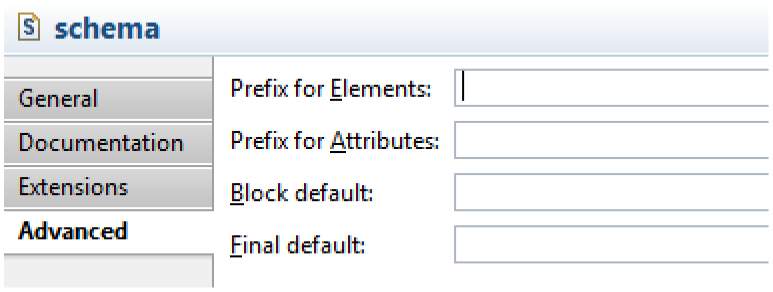
Prefix: m

Target namespace: http://www.mycom.com/movie

We should also specify that by default, the namespaces are not qualified.

__8. Click the **Advanced** tab in the left.

__9. Change the *Prefix for Elements* drop down to a blank.



Prefix for Elements:

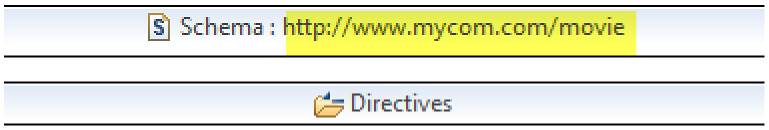
Prefix for Attributes:

Block default:

Final default:

__10. Save changes by hitting **Ctrl-S**.

The editor should update with the new namespace.



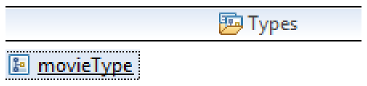
Schema : http://www.mycom.com/movie

We can now go about defining the *types* that make up this schema. The core type represents the movie data, which we will call the **movieType**. (The other type is the <database>, which is composed of multiple **movieType** elements – we will get to that later).

We will get started defining the **movieType**. Recall that we want the **movieType** to model rating, description, title and multiple actors as a combination of attributes and elements.

__11. Right click inside the *Types* pane and select **Add Complex Type**.

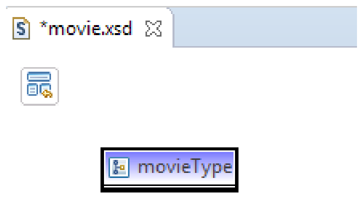
__12. A new type will appear. Change its name to **movieType**



The type has been defined. We now need to specify what data it contains.

__13. Double click on the newly created **movieType**.

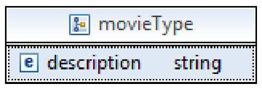
The view will change to a closeup of the **movieType**.



It is from here that we can add elements/attributes. Let us add the *description* element first.

__14. Right-click on the **movieType** and select **Add Element**.

__15. A new element box will appear. Rename it to **description**



Notice that the type for *description* was automatically set to **string**. This is fine because we want our description modeled as a string.

We have now just declared that our schema defines a type called **movieType** which itself has an element called **description** of type **string**.

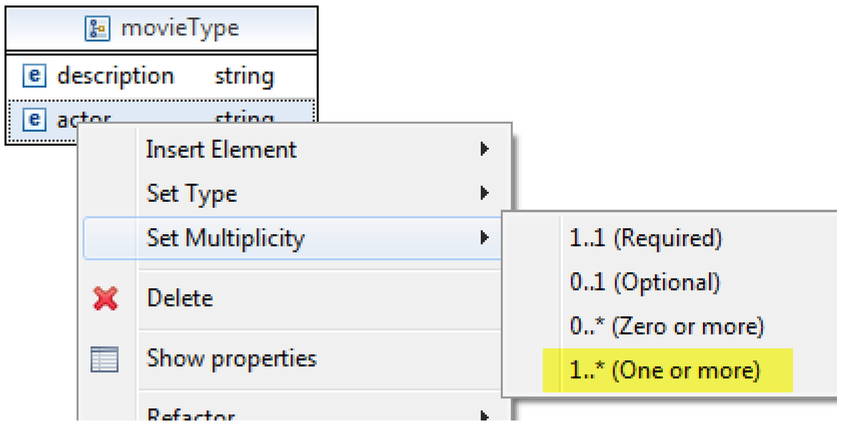
Now, we can add the **actor** element.

__16. Right click on **movieType** and again select **Add Element**.

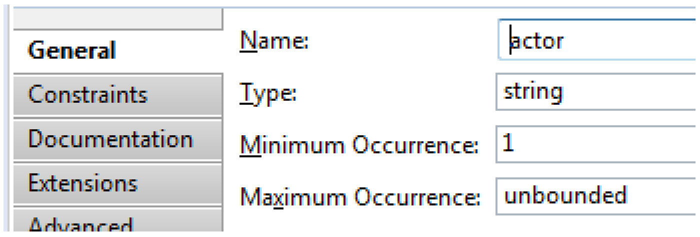
__17. This time, name it **actor**

Recall however, that our movie should have one or more actor elements associated with it. We can specify that here in the schema by using *multiplicity*

__18. Right click on the **actor** element and select **Set Multiplicity | 1..* (One or more)**



__19. In the **Properties** tab, click **General** at the left, for actor the minimum and maximum occurrence will change.



Name: actor
Type: string
Minimum Occurrence: 1
Maximum Occurrence: unbounded

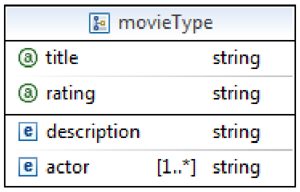
Now we can add the **rating** and **title** attributes.

__20. Right click on **movieType** and select **Add Attribute**.

__21. Name the attribute **title**

__22. Repeat this step and add another attribute called **rating**

The **movieType** element should now look like this.

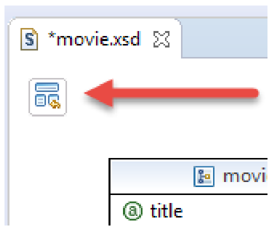


movieType		
title		string
rating		string
description		string
actor	[1..*]	string

As discussed earlier, a **movieType** has attributes for **title** and **rating** and elements for **description** and multiple **actors**.   (All of these are of type **string**)

We now need to define a **database** element that contains multiple instances of this **movieType**.

__23. Go back to the main schema design page by clicking the icon in the top left corner.



__24. Back at the main schema page, right click in the *Elements* pane (on the left) and select **Add Element**.

__25. Name it **database**

Let us now define that a **database** is composed of multiple **movie** elements. We will do so using an *anonymous type.*

__26. Right click on **database** and select **Set Type | New...**

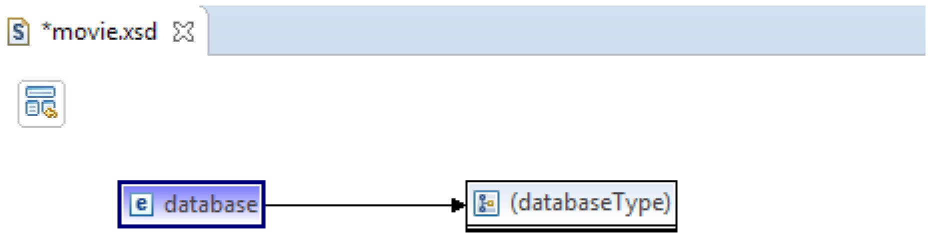
The *New Type* window will appear.

__27. Check **Create as local anonymous type**.   This will grey out the *Name* field.

__28. Click **OK**.

Seemingly, nothing will change.

__29. Double click on **database**.



We see that the **database** element points to something called a **(database type)**. This is an anonymous field, which means is basically just a placeholder for another element type – like our **movie**! (Using anonymous types can simplify schema).

We should now specify that the (**databaseType)** points to multiple **movie** elements.

__30. Right click on **(databaseType)** and select **Add Element**.

__31. A new element will be created. Name the element **movie**

__32. We know there are multiple movies, so we should set the multiplicity. Right click on **movie** and select **Set Multiplicity | 1..* (One or more)**

__33. Click the database to refresh the diagram.

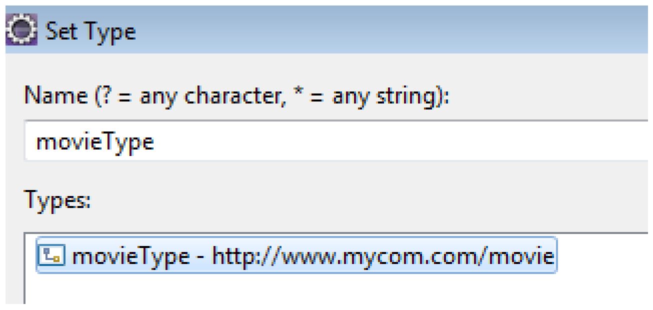
One more thing. We have to set the type of each **movie** element to be **movieType**.

__34. Click on **movie**, and then under **Properties** tab click the drop-down next to **Type**.

__35. Select **Browse...** from the drop-down box.

The *Set Type* window will appear.

__36. In the *Name* field, enter **movieType** and then click on it under **Types**.



Set Type

Name (? = any character, * = any string):

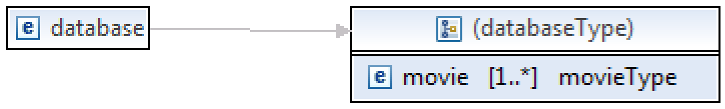
movieType

Types:

movieType - http://www.mycom.com/movie

__37. Click **OK**.

The editor will now show the association between the **<movie>** element and the **movieType**.



We are done!

__38. Hit **Ctrl-S** to save changes.

We have completed creating the schema.

Part 4 - Examine the Schema

We have now successfully created a schema using the Eclipse schema editor tool. The tool generated all the necessary code for us; let us take a look at it now.

__1. Click the **Source** tab at the bottom of the schema editor. The raw source code forming the schema will be displayed. It should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.mycom.com/movie"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:m="http://www.mycom.com/movie">

<complexType name="movieType">
          <sequence>
          <element name="description" type="string"></element>
          <element name="actor" type="string" maxOccurs="unbounded"
                              minOccurs="1">
          </element>
          </sequence>
          <attribute name="title" type="string"></attribute>
          <attribute name="rating" type="string"></attribute>
</complexType>

<element name="database">
          <complexType>
          <sequence>
                    <element name="movie" type="m:movieType"
                              maxOccurs="unbounded" minOccurs="1">
                    </element>
          </sequence>
          </complexType>
</element>
```

```
</schema>
```

Notice that the schema itself is written in XML. Go through it and mentally understand what is being described here:

- We state that a **database** is composed of 1 or more **movie** elements. Each **movie** element is of type **movieType**.
- A **movieType** is composed of elements representing **description** and 1 or more **actor,** and attributes representing the **title** and **rating.**
- If you compare this with the **movie.xml** we examined at the start of this lab, you should realize that **movie.xml** indeed conforms to the rules set out in the schema.
- Now, when an XML parser reads **movie.xml**, it can automatically check with this **xsd** to ensure it is valid. We will use this schema in later labs in exactly this manner.

__2. Close the schema editor.

Part 5 - Review

In this lab, you examined XML schema.

We started off by examining an XML file that gave us an idea of how to represent a movie.

We then created a schema that to formalize these rules of representation. We used the Eclipse schema editor to create the XML file and in there defined the various elements and attributes that should make up a movie database.

Finally, we examined the raw XML source behind the schema to get a feel for what it describes, and saw that it indeed validates the **movie.xml** that we started off with.

Lab 4 - JAXB Programming

In the previous lab, we created a schema for an XML file which we claimed would make parsing an XML file less error-prone. Thus far, however, we have dodged the question of actually parsing XML code; that is to say, we have not actually seen how to read/write an XML file.

Reading an XML file typically involves the use of a *parser*. A *parser* is a set of Java classes (typically provided as a part of the JDK distribution) that performs the low level operations of opening XML files and reading the contents (elements/attributes) into a format that your Java programs can use. Two common parsers are the SAX parser and the DOM parser.

While these parsers work, they are rather primitive; for example, SAX essentially treats an XML file as a file stream which forces the developer to write an event handling mechanism in order to detect content in elements or attributes. This is very low-level code and not object oriented at all.

Ideally, we would like to be able to treat XML as an 'object'. Imagine being able to read an XML file and automatically have the contents of that file being parsed into Java objects; this would allow us to use regular Java API instead of having to deal with issues like characters and white spacing.

This is precisely what JAXB offers. Given a schema file, JAXB can parse through it and create *bindings*. A binding is a Java class that contains business logic to interact with underlying XML, performing all the parsing for us. If a schema file describes a Person that has Name and Age elements, JAXB could generate *binding classes* that represent a Person class with name and age fields. Code wanting to use the XML data would now use the *binding classes* as regular Java classes, which would effectively hide all the XML code from us.

Creating the binding classes can be done via an Eclipse wizard. Once the binding classes are created, our Java classes can then use them to *marshal* (write the classes out to XML) or *unmarshal* (read classes in from an XML file) objects.

In this lab, we will use JAXB to unmarshal the contents of an XML file.

Part 1 - Generating the Bindings

JAXB starts by using a schema to generate binding classes. Basically, JAXB first examines the schema model to figure out how to build a representative Java object. JAXB then generates source code for new Java classes that will contain the corresponding object model (as derived from the schema) as well as the code needed to read/write the equivalent XML. So, the first thing we need to do is generate a binding for a schema.

We will use the **movie.xsd** schema we created in the previous lab.

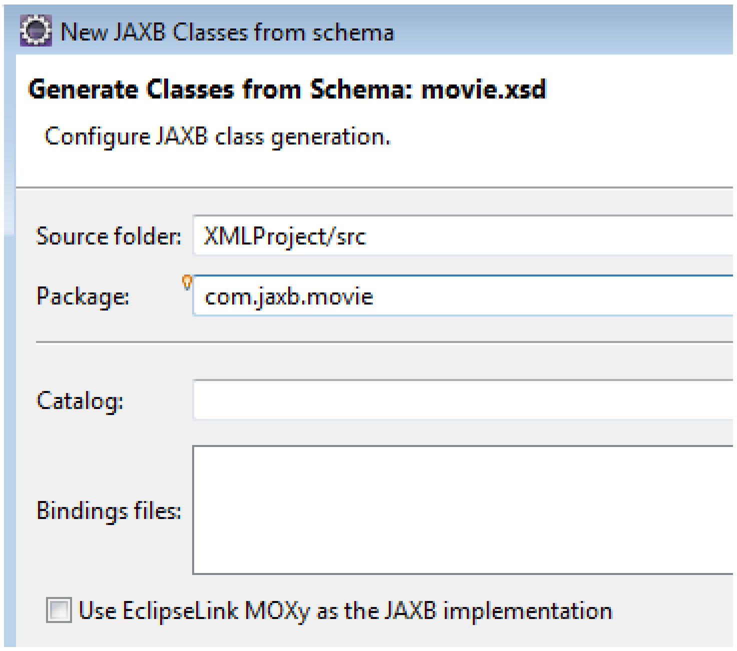
The most recent version Eclipse includes a wizard that can generate Java classes from an XML schema. This is much easier than trying to run a command from a command line.

__1. Expand the **xml** folder of the **XMLProject**.

__2. Right click the **movie.xsd** file and select **Generate** → **JAXB Classes**.

__3. Make sure **XMLProject** is selected and click **Next**.

__4. Change the **Package** to '**com.jaxb.movie**' as shown below and click the **Finish** button.



New JAXB Classes from schema

Generate Classes from Schema: movie.xsd

Configure JAXB class generation.

Source folder: XMLProject/src

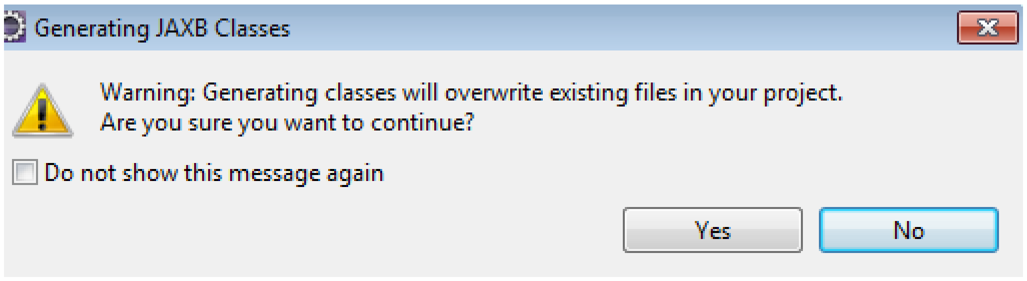
Package: com.jaxb.movie

Catalog:

Bindings files:

☐ Use EclipseLink MOXy as the JAXB implementation

__5. A Warning message will come up, click **Yes**.



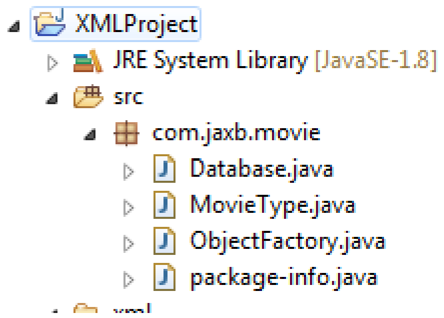
Generating JAXB Classes

Warning: Generating classes will overwrite existing files in your project.
Are you sure you want to continue?

☐ Do not show this message again

__6. Expand the **src** folder of project **XMLProject**.

__7. A newly created package **com.jaxb.movie** should appear. Expand it and you should see the generated binding classes.



Our binding classes have been generated, and we can now use them to read in from the **movie.xml** file! The **Database** class (**Database.java**) represents the <**database**> root level element. The **MovieType** class represents the <**movie**> element. Elements such as <**actor**> and <**description**> are of simple data types and no class was generated for them.

Part 2 - Use the Binding Classes To Read XML

We can now use the binding classes to write a Java class that reads **movie.xml**, but without having to use any parsers. The new code will be easier to read.

We will take advantage of the JAXB framework that provides an unmarshalling feature. (Recall that to *unmarshal* means to read from an XML file and convert content to Java objects)

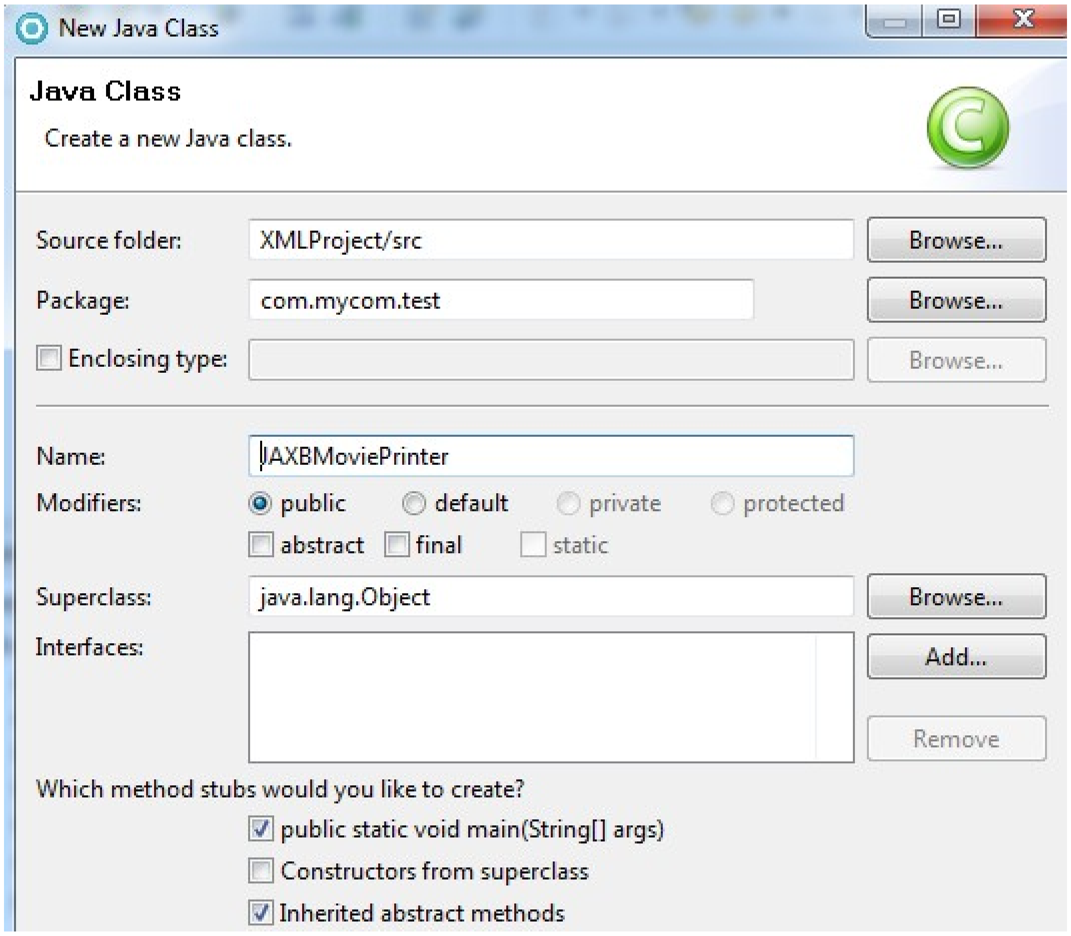
__1. Right click **XMLProject** and select **New > Package**.

__2. Enter **com.mycom.test** as the Name and click **Finish**.

__3. Right click the **com.mycom.test** package and select **New > Class**.

__4. Enter **JAXBMoviePrinter** as Name.

__5. Check the **Public static void main** check box.


New Java Class

Java Class

Create a new Java class.

Source folder: XMLProject/src

Package: com.mycom.test

[ ] Enclosing type:

Name: JAXBMoviePrinter

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

[✓] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

__6. Click **Finish**.

(If you have trouble typing in the following code, we have provided the completed class source code at **C:\LabFiles\JAXBMoviePrinter.java.solution** Feel free to copy and paste if necessary).

__7. In the **main** method, we first need to setup the unmarshaler. Add the following code to the **main** method.

```
JAXBContext jc = JAXBContext.newInstance("com.jaxb.movie");
Unmarshaller unmarshaller = jc.createUnmarshaller();
```

Note that here, the package name for the binding classes (**com.jaxb.movie**) is specified as input to the newInstance() method. If there are multiple packages, the names should be separated by ":". For example: JAXBContext.newInstance(**"com.foo.movie:com.bar.movie"**).

__8. There are some errors in the code. To fix a few of them, perform an *Organize imports*. (**Ctrl + Shift + O**), selecting **javax.xml.bind.Unmarshaller** when prompted. After you do this, you will still see some errors, but ignore them for now.

__9. Recall that in our schema, that "top-most" element was **database**. If you look at the generated binding classes, you will notice that a **Database** class was created. We can use the unmarshaler to create an instance of this object. Add the following code:

```
Database db = (Database) unmarshaller.unmarshal(
    new File("C:\\LabFiles\\movie.xml"));
```

Note here that the call to the **unmarshal** method returns an instance of the **Database**. This Java object represents the XML equivalent of the **<database>** element, populated with the data that was in the XML file. All the XML parsing has been performed for us!

There are several errors appearing in the code. This is because the appropriate packages have not been imported.

__10. Fix this by performing an *Organize Imports* which is done by typing **Ctrl-Shift-O**. Select **java.io.File** when prompted. There will still be some errors. Do not worry about them for now.

__11. Remember that a **<database>** element contained several <**movie**> elements. Similarly, the **database** object will have a field that represents a list of **MovieType** objects. We can *get* this list. Add the following code:

```
List<MovieType> movies = db.getMovie();
```

__12. We now have a hold of the list of movies that the database was keeping track of. We can iterate through it, and operate on it! Let us simply go through the list and print the title of each movie. Add the following code:

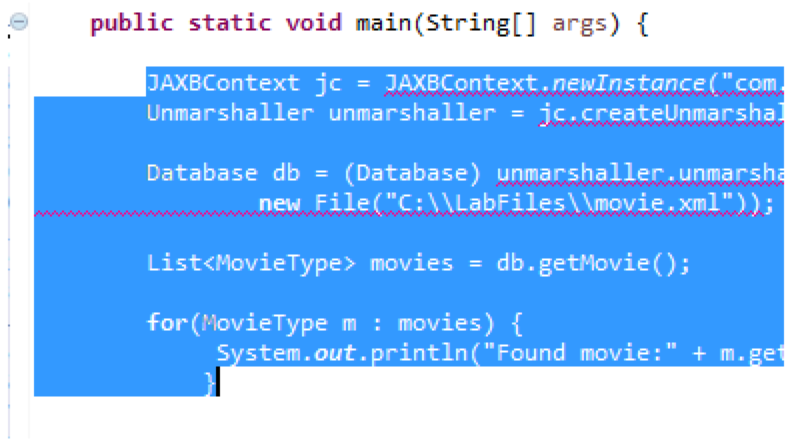
```
for(MovieType m : movies) {
 System.out.println("Found movie:" + m.getTitle());
}
```

Notice that we are using the **MovieType** object here. What exactly is this object? It is one of the generated binding classes. If you examine the code for it, you will notice that it has fields (and accessors for those fields) for each of the elements specified in the schema. Essentially, it is the Java version of the **<movie**> element. The code we entered makes use of the **getTitle**() method which again was auto generated for us!

__13. Organize imports again, by hitting **Ctrl-Shift-O**. Select **java.util.List** when prompted.

We are almost ready to run the code. There are, however, some errors left and these are because we are not handling the various exceptions that the marshaling process can throw. We will fix this now.

__14. Select all the code in the **main** method (but not the method declaration).



```
public static void main(String[] args) {

        JAXBContext jc = JAXBContext.newInstance("com.
        Unmarshaller unmarshaller = jc.createUnmarshal

        Database db = (Database) unmarshaller.unmarsha
                new File("C:\\LabFiles\\movie.xml"));

        List<MovieType> movies = db.getMovie();

        for(MovieType m : movies) {
            System.out.println("Found movie:" + m.get
        }
```

__15. Right click on it and select **Surround With → Try/catch block**.

A **try/catch** clause will be added to the code. All the errors should disappear.

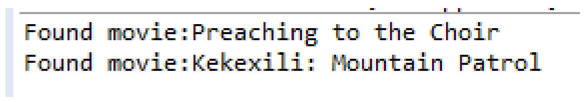
```
public static void main(String[] args) {
    try {
        JAXBContext jc = JAXBContext.newInstance("com.jaxb.movie");
        Unmarshaller unmarshaller = jc.createUnmarshaller();
        Database db = (Database) unmarshaller.unmarshal(
                new File("C:\\LabFiles\\movie.xml"));
        List<MovieType> movies = db.getMovie();

        for(MovieType m : movies) {
            System.out.println("Found movie:" + m.getTitle());
          }
    } catch (JAXBException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

__16. Save the code.

Part 3 - Test JAXB

__1. Right click on the code and select **Run As > Java Application**.

The console will open and you should see the following.



```
Found movie:Preaching to the Choir
Found movie:Kekexili: Mountain Patrol
```

The unmarshalling process worked!

Note: If you do not see any movies it is likely because you have the wrong setting in the schema for qualified/unqualified prefix (elementFormDefault). If you get this behavior you can add a 'm:' prefix to every <movie> and </movie> tag in the XML or switch the schema and recreate the JAXB code by running the build again.

Using JAXB, we were able to read in data from an XML file without having to write any XML parsing code! Even better, we were able to use the data in the XML file as if they were pure Java objects. All this was done using very little code.

The binding classes could also be used to write out an XML file, with a similarly simple API.

__2. Close all open editors.

Part 4 - Review

In this lab, you used JAXB to parse an XML file. You used JAXB to generate binding classes from the schema. You then used the binding classes in a Java program allowing with the marshaling features of JAXB to read in an XML file. The read-in data was then translated to objects of the generated binding class type, allowing us to use a more OO model of coding.

Lab 5 - Create A Bottom Up Web Service

In this lab, we will build a bottom-up web service. This involves creating the Web service implementation Java class first and then generating the other supporting artifacts that complete the Web service implementation.

To keep things simple we will implement the same HelloSvc that we have used in a previous lab. But, this time, you will get to do all the work from scratch. The goal of the lab is to learn what it takes to implement a bottom up Web Service using JAX-WS.

We will *annotate* the Java bean class to declare that it indeed is a web service. We will then generate a Web Service. Fortunately, in Eclipse there is very little to do. In other platforms, like Weblogic, you may have to run an ANT task or some other command to generate the supporting artifact files that complete the Web Service implementation.

The generated service will then be deployed and tested.

Finally, we will see how to make changes to a deployed web service.

Part 1 - Create the Project

We need to create a *dynamic web project* to host our web service code. The web project will be nested inside an EAR project to simplify packaging. We will create both these projects now.

__1. From the menu, select **File | New | Dynamic Web Project**.

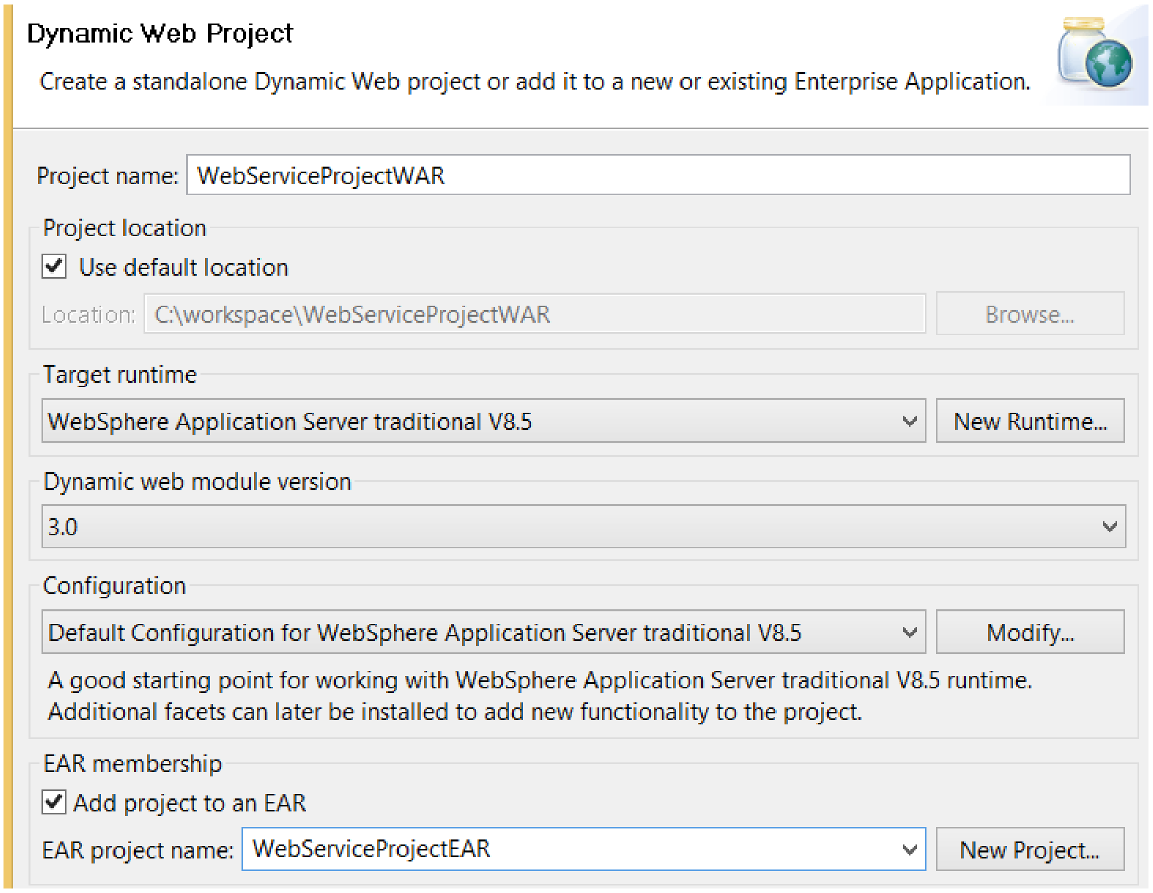
The *Dynamic Web Project* screen will appear.

__2. Enter **WebServiceProjectWAR** for the *Project name*.

__3. At the bottom of the window, in the *EAR Membership* section, make sure **Add project to an EAR** is selected. (We will want to add this WAR file into an EAR file)

The *New EAR Application Project* screen will appear.

__4. Set the *Project name* to **WebServiceProjectEAR**


Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: WebServiceProjectWAR

Project location

☑ Use default location

Location: C:\workspace\WebServiceProjectWAR

Target runtime: WebSphere Application Server traditional V8.5

Dynamic web module version: 3.0

Configuration: Default Configuration for WebSphere Application Server traditional V8.5

A good starting point for working with WebSphere Application Server traditional V8.5 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☑ Add project to an EAR

EAR project name: WebServiceProjectEAR

Your screen should now look above:

__5. Click **Finish**. Both the **EAR** and the **WAR** will be created.

Eclipse will displayed them in the *Enterprise Explorer*.

__6. Start the Server if it is not running.

__7. Right click on the server and select **Add and Remove..**

__8. Click **Add All**.

__9. Click **Finish**. Wait until the server is synchronized.

Our project(s) have been created and deployed in the server! We will perform all our work in the *WebServiceProjectWAR* dynamic web project.

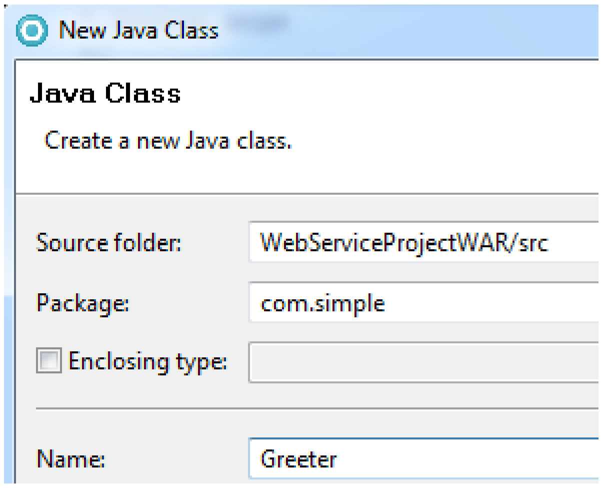
Part 2 - Create the Implementation Class

Since this is a *bottom-up* web service, we will first create the implementation class. Recall that this is a plain Java bean class that has a simple business method. We will eventually want this method to be exposed as a web service operation.

__1. Right click on **WebServiceProjectWAR** and select **New** → **Class**.

__2. Set the *Package* to be **com.simple**

__3. Set the *Name* to be **Greeter** (we will re-create the class we created in Lab 1).



New Java Class

Java Class

Create a new Java class.

Source folder: WebServiceProjectWAR/src

Package: com.simple

Enclosing type:

Name: Greeter

__4. Click **Finish**.

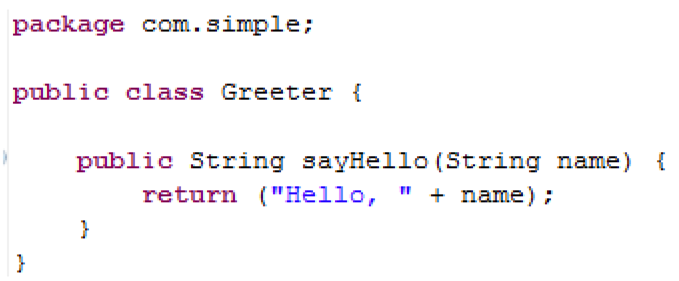
An editor will open on the newly created source file.

__5. Add the following method to the class:

```
public String sayHello(String name) {
          return ("Hello, " + name);
}
```

__6. Save the code. There should be no errors.

Your code should now look like the following:



```
package com.simple;

public class Greeter {

    public String sayHello(String name) {
        return ("Hello, " + name);
    }
}
```

We have completed the implementation class. We now need to turn this class into a web service.

Part 3 - Annotate the Class

JAX-WS requires that a Java class be annotated to make it a Web service implementation. You will do that now.

__1. We should first mark the class as being a web service. To do this, add the following annotation in bold to the class, right before the class definition:

```
@WebService
public class Greeter {
          ...
```

Adding this annotation marks the class as a web service.

We should now specify which methods we want exposed as operations.

__2. Annotate the **sayHello** method as follows in bold:

```
@WebMethod
public String sayHello(String name) {
…
```

Note: A method designated as @WebMethod must be public.

__3. There will be two errors on the code, complaining about the annotations. Fix that now by performing an Organize Imports. (**Ctrl-Shift-O**).

Note: Eclipse supports code completion for annotation names and their attributes. Feel free to use that feature any time the lab guide asks you to type in an annotation.

__4. Save the code. There should be no errors.

We have completed annotating the source code. The class can now be executed as a web service!

Part 4 - Deploy and Test the Service

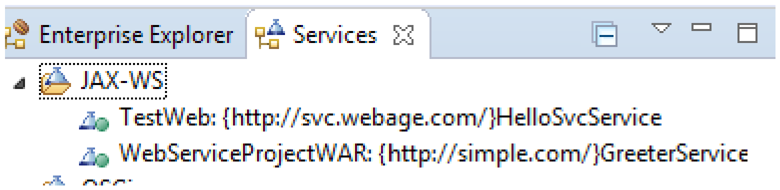
__1. Open the *Servers* view.

__2. Right click the server and select **Publish**. Wait until the server is synchronized.

The service should now be deployed. How can we tell? A good indication is checking for the WSDL file. If the service is deployed, its WSDL file should be available for consumption. We will test this now.

__3. In the *Enterprise Explorer*, click the *Services* tab (top right). This will open the *Services* view.

__4. Expand the **JAX-WS** tree.



```
JAX-WS
  TestWeb: {http://svc.webage.com/}HelloSvcService
  WebServiceProjectWAR: {http://simple.com/}GreeterService
```

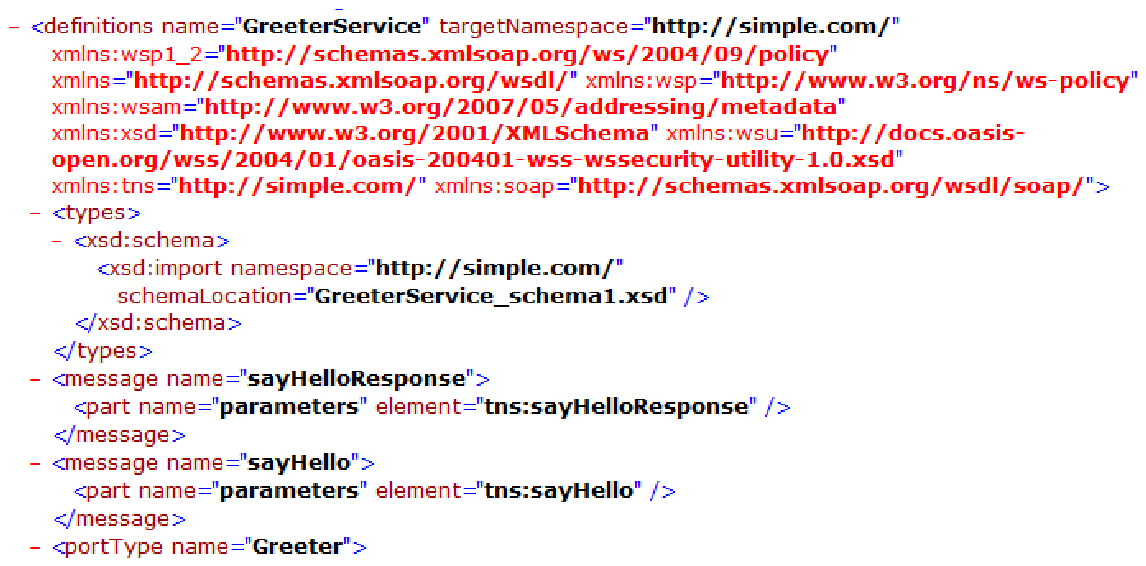
Notice that two services are listed. One is the **HelloSvcService** service from the **TestWeb** project (which we examined in previous Labs), and the other is **GreeterService** we have just created, in the **WebServiceProjectWAR** project. Eclipse has automatically read our annotated code and is aware they are services.

Additionally, this *Services* view makes testing easy. We will use it in a moment.

__5. Open a web browser and navigate to: (remember that there is a web browser inside eclipse and it's located in the bar)

```
http://localhost:9080/WebServiceProjectWAR/GreeterService/GreeterService.wsdl
```

The browser should show the WSDL file for the service.



```
- <definitions name="GreeterService" targetNamespace="http://simple.com/"
    xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsu="http://docs.oasis-
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:tns="http://simple.com/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  - <types>
    - <xsd:schema>
        <xsd:import namespace="http://simple.com/"
          schemaLocation="GreeterService_schema1.xsd" />
      </xsd:schema>
    </types>
  - <message name="sayHelloResponse">
      <part name="parameters" element="tns:sayHelloResponse" />
    </message>
  - <message name="sayHello">
      <part name="parameters" element="tns:sayHello" />
    </message>
  - <portType name="Greeter">
```

A web service client could now download and examine that WSDL file to figure out how to invoke the service.

We will now test the service using the Web Services Explorer (WSE) which we used before.

__6. In the *Services* view, right-click on the **WebServiceProjectWAR** service and select **Show > WSDL Interface**.

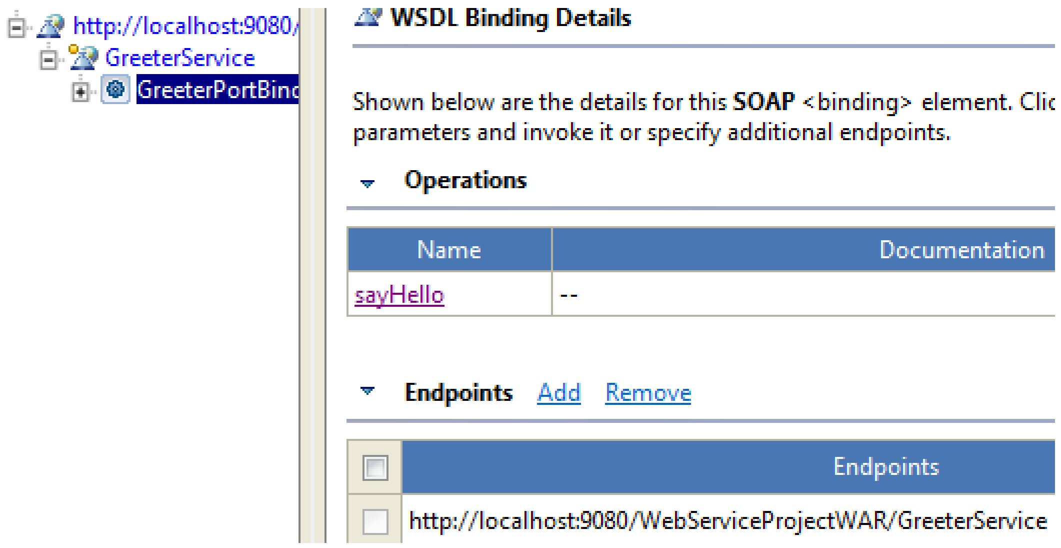
__7. Select the **Source** tab to view the file as code.

__8. Scroll down to the bottom of the code and verify the soap address has been updated to the service URL. If it is not showing the above URL continue with the steps, there is a note about this later (Step 10).

```
<service name="GreeterService">
  <port name="GreeterPort" binding="tns:GreeterPortBinding">
    <soap:address location="http://localhost:9080/WebServiceProjectWAR/GreeterService"/>
  </port>
</service>
```

__9. In the *Services* view, right-click on the **WebServiceProjectWAR** service and select **Test with Web Services Explorer**.

The WSE will open.



Shown below are the details for this **SOAP** <binding> element. Clic
parameters and invoke it or specify additional endpoints.

Operations

Name	Documentation
sayHello	--

Endpoints Add Remove

☐	Endpoints
☐	http://localhost:9080/WebServiceProjectWAR/GreeterService

Verify the Endpoint is configured as shown above. If not then close all open files, stop the server, start the server and run the service again. If the endpoint is not updated then you will need to create a new workspace and perform this Lab again. Ask your instructor for help, there is the solution for previous Lab in C:\LabFiles\Solutions that you can import and start again this lab again.

The WSE has located the WSDL file of the service, and from there has figured out what operations are available.

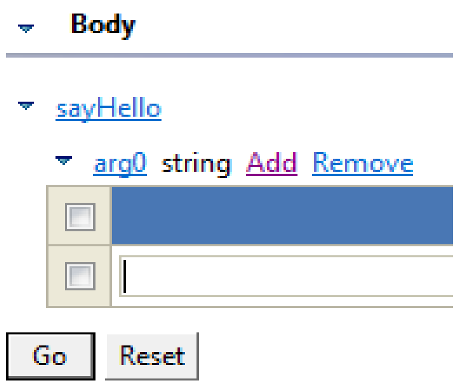
__10. Look in the *Actions* pane, on the right, click in the **sayHello** operation.

The *Actions* pane changes to display possible inputs to the service.

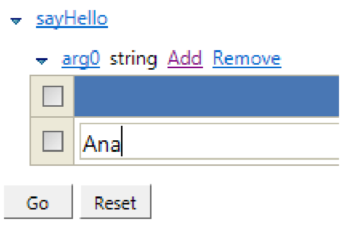
The WSE realizes that the *sayHello* service expects an argument – however, it does not appear to have rendered a field for us to enter an argument.

Why is this happening? This is because the WSDL file that was automatically generated did not specify that a parameter *must* be there.

__11. For now, however, we can still test our service. Click the **Add** link, and a field will appear.

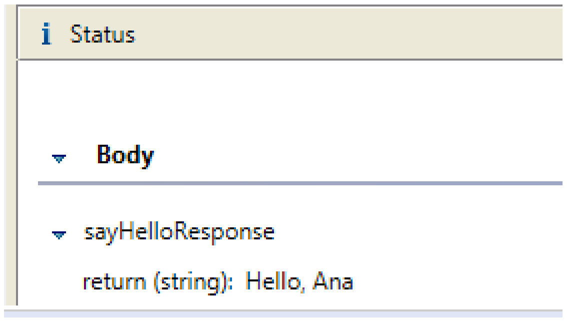


__12. Enter a name into the field.



__13. Click **Go**.

An appropriate response should come back, visible in the *Status* pane below.



return (string): Hello, Ana

Note: you may have to resize the Status pane to make it more visible; simply click and drag the area between the *Actions* and *Status* panes to resize.

Our web service has been invoked successfully!

Part 5 - Change the Implementation

At this point the system has used the default JAXB binding rules to generate the schema used by the Web Service. The default settings work but may not be ideal. First, we will look at the generated schema behavior and then modify it.

Let us examine some of the conventions that the generated service is using.

__1. In the WSE, examine the *Action* pane.

Notice that the name of the parameter is **arg0**. What exactly is the impact of this? To see, we have to look at the underlying SOAP messages. To be precise, we want to take a look at the SOAP request message that the WSE generated and was submitted to the service for processing.

__2. Click **Go**.

__3. In the lower *Status* pane of the WSE, click the ***Source*** link.

__4. Maximize the *Status* pane by double clicking on its title bar.

__5. Right click in the **SOAP Request Envelope** section and select **View source**.

An XML file will open.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:q0="http://simple.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:sayHello>
      <arg0>Ana</arg0>
    </q0:sayHello>
  </soapenv:Body>
</soapenv:Envelope>
```

- You are now looking at the raw XML-based SOAP messages that were submitted to – and returned from the server. The *SOAP Request Envelope* shows what the WSE submitted to the web service, running on WebSphere. The *SOAP Response Envelope* shows what the service returned to the WSE.
- First of all, notice that the default **q0** namespace prefix is for the namespace **http://simple.com**. This was derived from the package name of the implementation class. This works, but sometimes simply using the package name as the namespace is not desirable.
- Secondly, we see that the main body of the SOAP message contains the element **<q0:sayHello**> which is the name of the operation. That is fine; what is not ideal, however, is the fact that the argument is simply called **arg0**. While there is technically nothing wrong with this, it does make the SOAP a little more abstract; **arg0** is not a very meaningful name to any human reading the SOAP message.
- The question is now this: how did these two conventions (the default namespace and the argument name) receive these default values? An even better question is this: how do we *control* these default values? The answer is simple; via *annotations*.
- We can annotate the implementation class and specify what we want the namespace and the argument names to be.

Let us fix these now.

__6. Close the xml file.

__7. Return the *Status* pane to its normal size by double clicking on it again.

__8. Switch back to the *Enterprise Explorer* view and open **Greeter.java** in the editor again (under **com.simple** package).

__9. Update the **@WebService** annotation of the class to look like the following:

```
import javax.jws.soap.SOAPBinding;

@WebService(targetNamespace="www.example.org")
public class Greeter {
...
```

Here, we update the @WebService annotation to specify what we want the target namespace to be; in this case we will use **www.example.org**

__10. Now, let us fix the **arg0** problem. Add the following annotation to the *parameter* of the **sayHello** method, as follows in bold:

```
@WebMethod
public String sayHello(@WebParam(name="greet_name") String name) {
...
```

We are annotating the parameter to have a more meaningful name.

__11. Organize imports. There should be no errors.

__12. Save the file.

We have updated our implementation class by changing (and adding) annotations.

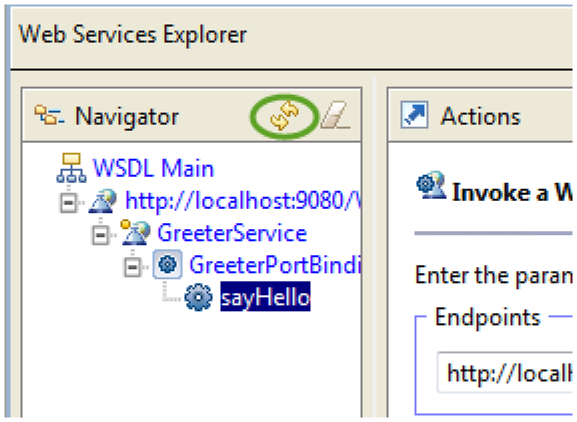
__13. We now need to push these latest code changes out to the server. Locate the *Servers* view and *Publish* the server by right-clicking on it and selecting **Publish**. Wait until finish synchronizing.

Part 6 - Examine the Updated Service

We should now see if the annotations worked.

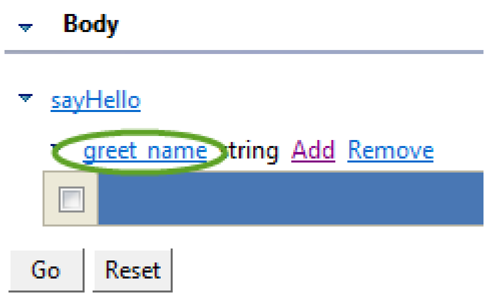
__1. Go back to the WSE which should still be pointing at the GreeterService.

__2. Click the "Refresh" button, which is to the right of the title bar in the *Navigator* pane of the WSE.



__3. Click the **sayHello** link again.

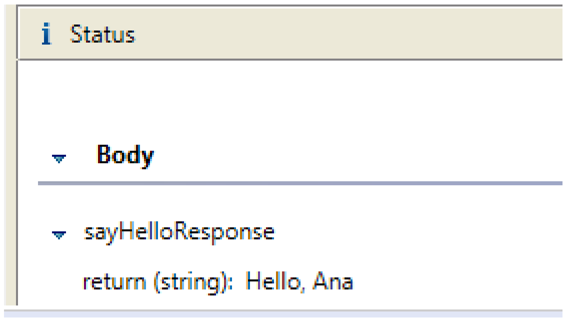
The WSE updates:



__4. Notice the parameter is now called **greet_name** as opposed to **arg0** as it was before.

__5. Click **Add**, enter a name and click **Go**.

__6. In Status, click **Form** link at the right. The response will come back as before.



sayHelloResponse
return (string): Hello, Ana

This is a good sign. Let us look at the underlying SOAP messages.

__7. Click the **Source** link in the *Status* pane, and then maximize the *Status* pane.

__8. Right click in the **SOAP Request Envelope** section and select **View source**.

An xml file will open.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:q0="www.example.org" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:sayHello>
      <greet_name>Ana</greet_name>
    </q0:sayHello>
  </soapenv:Body>
</soapenv:Envelope>
```

- Notice that the default namespace has been updated to www.example.org
- Also notice that the argument is, as expected, now called the more meaningful **greet_name** as opposed to just **arg0**.

__9. Close the XML file.

Congratulations! You have successfully updated the web service implementation by making use of the annotations.

__10. Close all open editor windows, including the WSE and the web browser.

Part 7 - Review

In this lab, you built a bottom up web service.

You started off by creating an implementation class, which is plain Java code. Your class contained a single method which we wanted to be exposed as an operation.

You then annotated the source code using JAX-WS annotations that marked the class as a web service.

You deployed the generated web service to WAS.

You then looked at the resulting SOAP request/response messages that resulted from the web service and noticed how certain defaults were being chosen for the namespace and the argument names.

After this, you went back to the implementation class and updated the annotations to change the default namespace and argument names.

Finally, you tested the service again and saw the impact of the updated annotations.

Lab 6 - Creating A WSDL File

A web service is often accompanied by a WSDL file. A WSDL file is essentially a description of the service, containing information such as the service operations' names, the operation parameters, etc.

A client wishing to invoke a web service often does so by first reading the WSDL file. The knowledge gleaned from examining the WSDL file allows the client to invoke the service.

For a developer, it is common to develop a service using a *top-down* approach. This means that the WSDL file is created *first*, and the a tool is used to generate service implementation code from the WSDL.

As you can see, a WSDL file is an integral part of any web service. In this lab, we will create a WSDL file using Eclipse.

Part 1 - Create A WSDL File

While it is possible to create the WSDL file by hand (and manually scribing all the XML code), this would be rather error-prone and time consuming. Instead, we will use the graphical capabilities to build the WSDL in a more intuitive way.

The service we will describe makes use of our developing movie application. It will describe one simple operation: *addMovie*. This operation will have one input parameter (of type *movie*) and return one output parameter, a String which represents a return code.

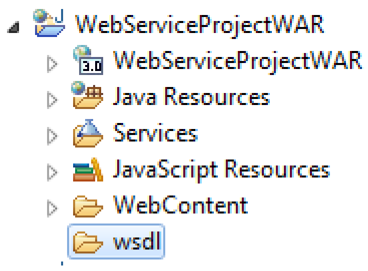
Note that we will only worry about *describing* the service here. We will examine *implementation* of the service in the next lab.

We will create the WSDL file in our **WebServiceProjectWAR**. First, though, we will create a directory for it.

__1. In the *Enterprise Explorer* view, right click on **WebServiceProjectWAR** and select **New | Folder**.

__2. The *New Folder* window will appear. Set the *Folder name* to be **wsdl** and click **Finish**.

The folder will be created.



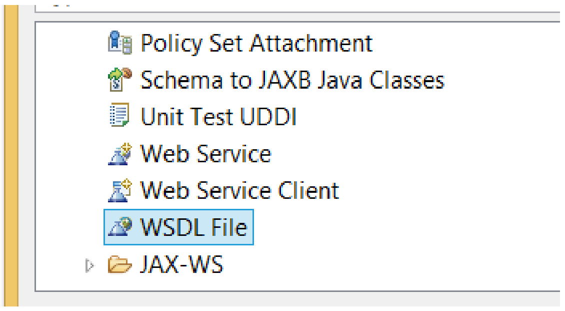
__3. In the *Enterprise Explorer* view, expand the **XMLProject** → **xml** folder.

__4. Copy the **movie.xsd** file.

__5. Paste the **movie.xsd** file into the **wsdl** folder of the **WebServiceProjectWAR** project.

__6. Right click on the **wsdl** folder and select **New > Other...**

__7. In the *New* window that appears, expand **Web Services** and select **WSDL File**.



__8. Click **Next**.

The *New WSDL File* window will appear.

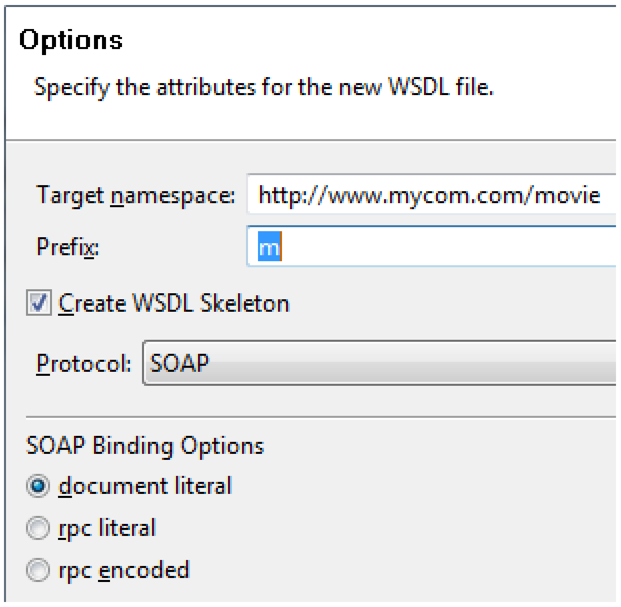
__9. Set the *File name* to be **movie_service.wsdl**

__10. Click **Next**.

The *Options* window will appear.

__11. Set the *Target namespace* to be **http://www.mycom.com/movie**

__12. Set the *Prefix* to be **m**



Options

Specify the attributes for the new WSDL file.

Target namespace: http://www.mycom.com/movie

Prefix: m

[✓] Create WSDL Skeleton

Protocol: SOAP

SOAP Binding Options

[● document literal]

○ rpc literal

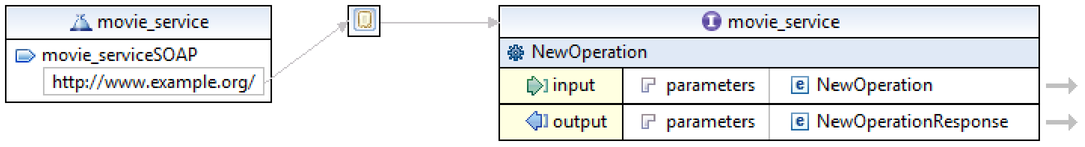
○ rpc encoded

__13. Click **Finish**.

The WSDL file will be created, and the WSDL editor will open on it. It will be showing the raw source code which is a rather frightening stack of XML.

Take a moment to examine the source code of the underlying **WSDL** file. It may seem a little daunting, but we will examine in a little more detail later on.

__14. Click the **Design** tab at the bottom of the editor. This shows a graphical representation of the WSDL file and is much easier to deal with.

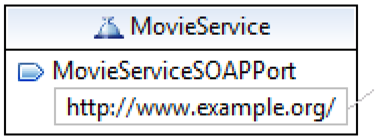


- Note the two main parts of the diagram; the box on the left represents the *port* definition of the WSDL file.
- The box on the right represents the various *operations* that the service offers. At the moment, the values for all this data are samples, populated by the editor. We will change these in a moment.
- We are currently viewing the WSDL in its *Simplified* form. While this may appear to make working with WSDL files easier, it does in fact remove some of the features we will use.

We are now ready to start editing our WSDL!

__15. First of all, let us rename the service to something a little more proper. In the *port* box (the box on the left), click the title area where **movie_service** is displayed. Click the text **movie_service** and rename it to **MovieService**

__16. Beneath it, in the same manner (i.e. clicking on the text and then renaming it), change **movie_serviceSOAP** to **MovieServiceSOAPPort**

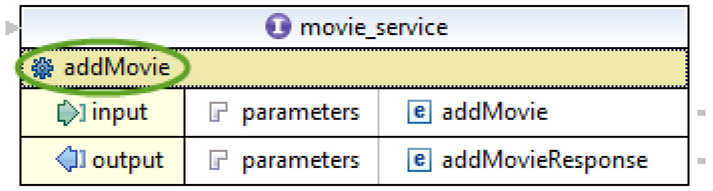


We can now update the operation. Our operation will be described as follows:

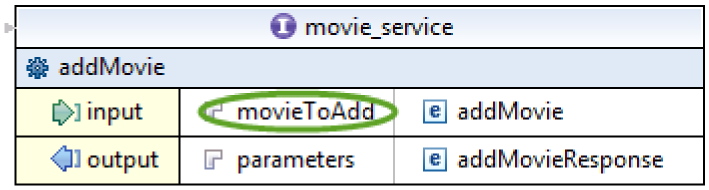
- The operation will be called *addMovie*
- It will take an input parameter called **movieToAdd** which will be of type **addMovie**.
- An **addMovie** type will be composed of a **movieType**, and a **movieType** is composed of the elements and attributes as we defined in the movie.xsd schema. This extra layer of 'wrapping' wherein we nest a **movieType** within an **addMovie** is due to the requirements of the SOAP encoding style (wrapped document literal) we are using.
- The *addMovie* operation will return one output parameter called **opStatus** which will be of type **addMovieResponse**. An **addMovieResponse** is merely a wrapper for a simple String. (Again, this is due to the SOAP encoding we are using)

We will specify all this now.

__17. Let us rename the service first. In the operation box (the box on the right) click **NewOperation** and rename it to **addMovie**

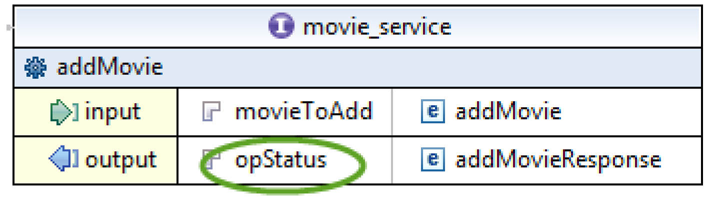


__18. Click on **parameters** next to *input* and rename it to **movieToAdd**



__19. Click on **parameters** next to *output* and rename it to **opStatus**

The diagram should now look like this:



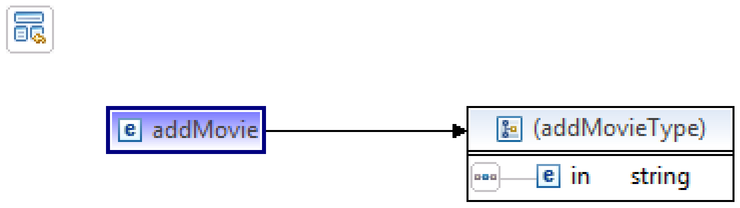
OK, we have fleshed out the specifics of the message. We now need to clarify the types of the message input/output.

__20. Click on the right-facing arrow on the far right of the *input* row.

The editor view will change to the *Inline Schema of movie_service.wsdl*, may be displaying the source.

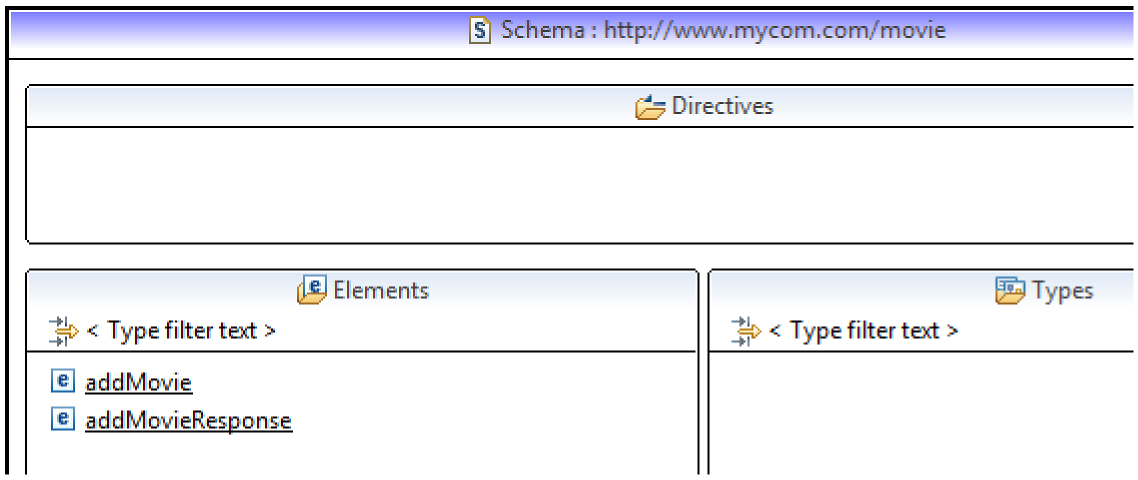
__21. Click the *Design* tab at the bottom of the editor if you are in source view.

The view will change to show a diagram.



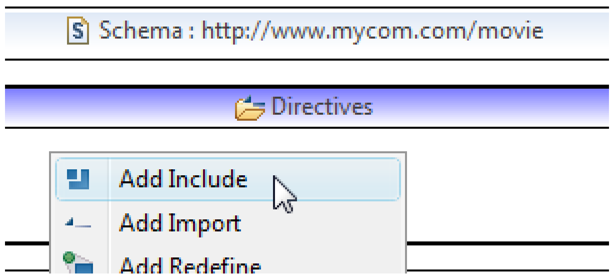
__22. Click on the icon in the top left of the editor.

This will bring us to the schema editor. This gives us a high level overview of the schema that this WSDL file is maintaining.



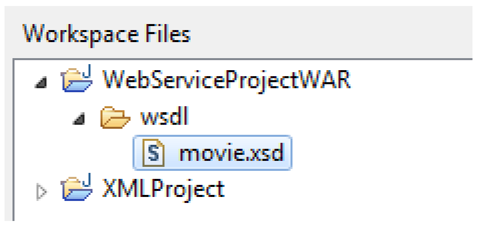
This should look familiar; we used this in a previous lab. Remember that in the Schema Creation Lab, you used this tool to define a schema for the movie type. We will be using a similar schema here in the WSDL file, and so we will be using the same tool.

__23. In the ***Directives*** box, right-click and select **Add Include**.



__24. Leave the default option of **Workbench projects** and click the **Next** button.

__25. Expand **WebServiceProjectWAR** → **wsdl** and select the **movie.xsd** file.

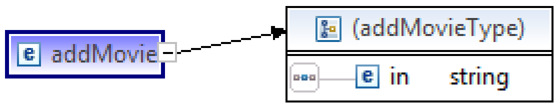


__26. Click the **Finish** button to create the link to the file.

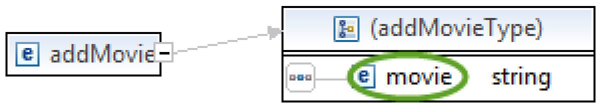
Note: Although it would be possible to select the file from the **XMLProject** directly this would not have included the file in the web project which would lead to problems when deploying to the server.

__27. In the *Elements* area, double click on **addMovie**.

The editor screen will change to show details of the **addMovie** type.



__28. In the box on the right, rename **in** to **movie**



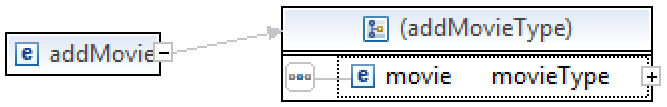
__29. We now need to set the type. Click **movie** and then under *Properties* tab at the bottom expand the dropdown that appears next to **Type**.

__30. In the list, click on **Browse...**

The *Set Type* window appears.

__31. In the list, type m in the name and locate **movieType** and double-click on it.

The editor will update to show the type has been set.

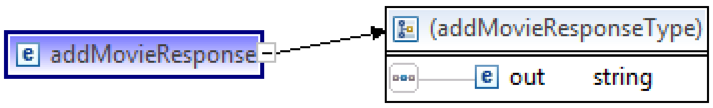


__32. Go back to the main schema view by clicking on the icon in the top left of the editor.

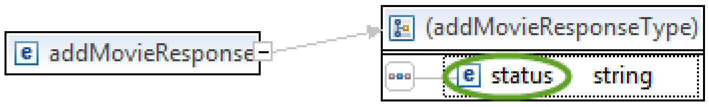
We now need to specify the **addMovieResponse** element is actually of type String and its body is called **status**.

__33. In the *Elements* box, double-click on **addMovieResponse**.

The editor will change to show details of **addMovieResponse**



__34. Rename the *out* to **status**



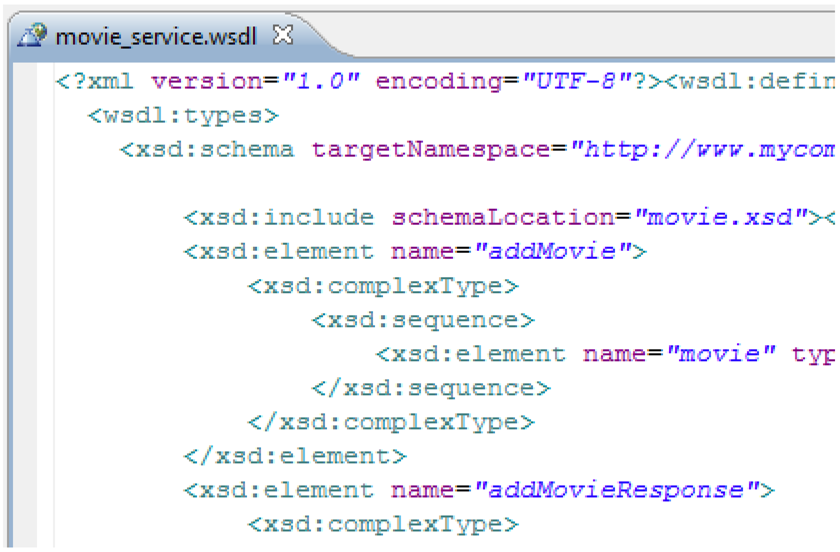
__35. We are done! Save changes by hitting **Ctrl-S**.

__36. Close the *Inline Schema of movie_service.wsdl* view. (Leave the other editor, showing **movie_service.wsdl** open)

Part 2 - Examine the Generated WSDL Source

Let us take a look under the covers and examine the underlying XML code that was generated for us.

__1. Click the **Source** tab at the bottom of the WSDL editor.



```
<?xml version="1.0" encoding="UTF-8"?><wsdl:defin
  <wsdl:types>
    <xsd:schema targetNamespace="http://www.mycom

        <xsd:include schemaLocation="movie.xsd"><
        <xsd:element name="addMovie">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="movie" typ
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="addMovieResponse">
            <xsd:complexType>
```

__2. The indentation may be a little off; right click anywhere on the source and select **Source** | **Format**.

__3. Save all the files by selecting from the menu, **File > Save All**.

That will make it a little more readable.

Let us take a look at the first part of the source WSDL file; the definition of the types.

- This is the content contained within the **<wsdl:types>** body. This section looks like regular schema XML which, in fact, it is.
- Notice that have been defined **addMovie** (which is the wrapper for the **movieType**) and the **addMovieResponse**.

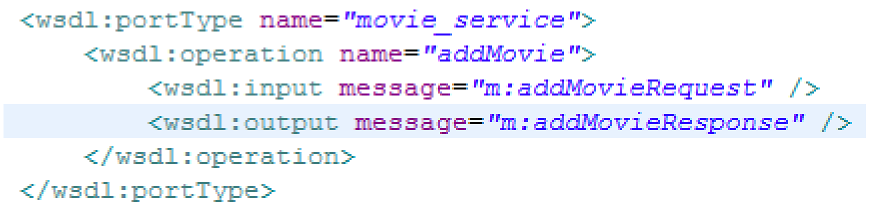
```
<wsdl:types>
    <xsd:schema targetNamespace="http://www.mycom.com/movie">
        <xsd:include schemaLocation="movie.xsd"></xsd:include>
        <xsd:element name="addMovie">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="movie" type="m:movieType" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="addMovieResponse">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="status" type="xsd:string" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
</wsdl:types>
```

Now, let us examine the second part of the WSDL file; the definition of *messages*:

```
<wsdl:message name="addMovieRequest">
    <wsdl:part element="m:addMovie" name="movieToAdd" />
</wsdl:message>
<wsdl:message name="addMovieResponse">
    <wsdl:part element="m:addMovieResponse" name="opStatus" />
</wsdl:message>
```

Two messages have been defined, representing the incoming request message (**addMovieRequest**) and outgoing response message (**addMovieResponse**) appropriately. Notice that the **element** attribute specifies a type that was defined in the previous types section. These messages will be used elsewhere in the file.

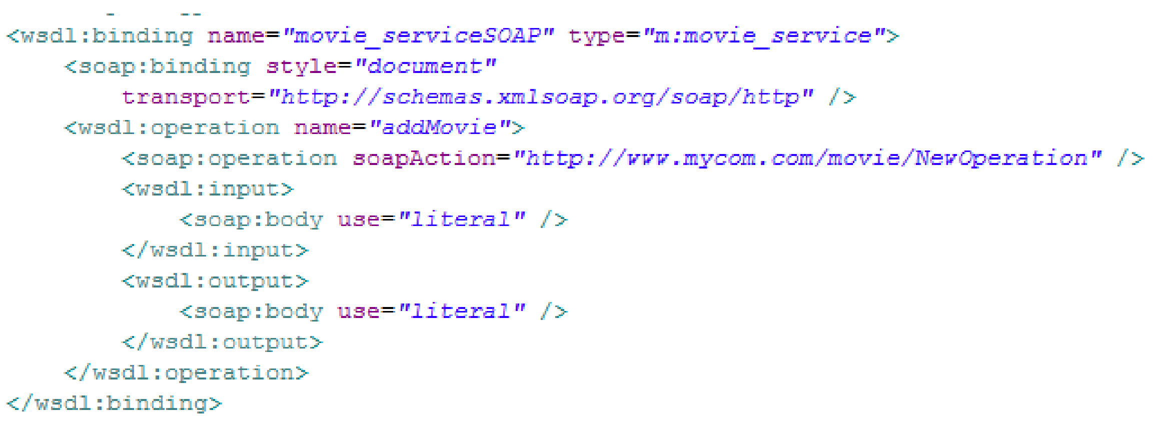
Next let us examine the *portType* definition.



```
<wsdl:portType name="movie_service">
    <wsdl:operation name="addMovie">
        <wsdl:input message="m:addMovieRequest" />
        <wsdl:output message="m:addMovieResponse" />
    </wsdl:operation>
</wsdl:portType>
```

Here, we specify the actual operation and dictate what it uses as request/response messages. Notice that these request/response messages were defined in the previous *messages* section.

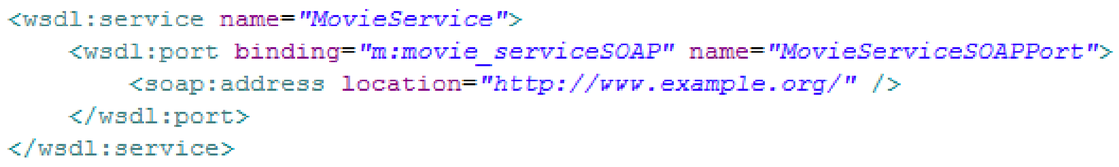
Now, we can look at the *binding* components.



```
<wsdl:binding name="movie_serviceSOAP" type="m:movie_service">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="addMovie">
        <soap:operation soapAction="http://www.mycom.com/movie/NewOperation" />
        <wsdl:input>
            <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

The **<wsdl:binding>** element defines how operations should be invoked using a specific protocol. In our case, the service implementation will be accessible using SOAP over HTTP. The binding section deals with SOAP. We have specified that we will use the wrapped literal style.

Finally, we examine the *endpoint*.



```
<wsdl:service name="MovieService">
    <wsdl:port binding="m:movie_serviceSOAP" name="MovieServiceSOAPPort">
        <soap:address location="http://www.example.org/" />
    </wsdl:port>
</wsdl:service>
```

The **<wsdl:service>** element specifies various protocol specific parameters. These are useful to the consumer of the service. These parameters will define the address location of the service. For SOAP over HTTP, that will be the URL where the HTTP request should be sent. For SOAP over JMS, the endpoint address will be a JMS queue name.

At the moment, it is pointing at some nonsensical address (http://www.example.org/).

When the service implementation corresponding to this WSDL file is deployed, the server runtime will update this part of the WSDL file to point at the actual URL where the service is running.

That's it! You have created and analyzed a WSDL file.

__4. Close the WSDL editor.

Part 3 - Review

In this lab, you used WTP's GUI WSDL editor to create a simple WSDL file. You then examined the WSDL file and saw that it is broken down to several parts:

- Types: defines the various schema of objects that the service uses as input/output parameters.
- Messages: defines the format of incoming/outgoing messages.
- Port type: defines the actual operation(s) that the service provides.
- Binding: describes how the service can be invoked over which protocol.
- Service: Defines the actual instructions for actually invoking the service.

Lab 7 - Creating A Top-Down Web Service

So far, we have examined the concept of a bottom-up web services; this was where we developed an implementation class first. This was called the bottom-up approach.

A different approach to writing a web service is via a *top-down* approach. Here, we start things off with a WSDL file that describes the service. We then use a tool to generate a skeleton implementation class from the WSDL file. At that point, all the developer has to do is add the implementation code to the generated skeleton.

This approach is handy because this essentially allows the specification of a contract before any coding is done. Top-down development has two advantages:

1. You have precise control over the XML schema and namespaces. You don't need to worry about using JAX-WS or JAXB annotations to control the schema.
2. In many cases, the WSDL will be designed by another organization and your company will have to implement a Web Service that conforms to it. In such a situation bottom-up approach will not work very well.

In this lab exercise, we will develop an implementation for the **movie_service**.**wsdl** file that we created in an earlier lab.

Part 1 - Generate the Implementation Classes

Remember that we are now going "backwards"; we will use the tool to generate skeleton implementation classes based on the WSDL. We will then add the actual business logic to the generated implementation classes. The entire package can then be deployed and the service will be made available.

- We will use the WSDL for the **movie_service** that we created in an earlier lab. Recall that this service exposed a single operation – **addMovie**
- As an input, the service required an instance of an **addMovie** element, which contained a **movieType** element (which in turn, defined fields and attributes for actual movie data).
- As an output, the service returned an **addMovieResponse** which was simply a wrapper for a string.
- From this, you should get an idea of what needs to be generated. We will need one class to contain the implementation business logic of the **addMovie** operation. On top of that, we will need various classes for the input and output parameters.
- The generated implementation class should contain a method called **addMovie(AddMovieType aMovieType**) returning an **AddMovieResponse**. In theory, then, all we should have to do is add some business logic to that implementation class.

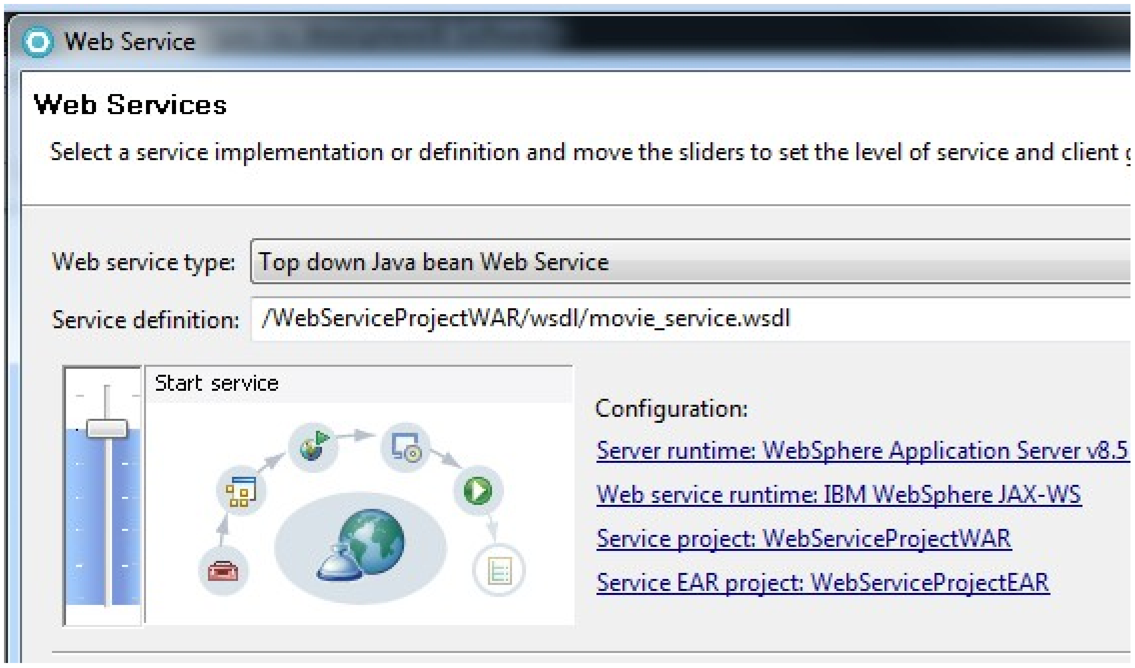
Let us test this theory.

__1. Generating a web service implementation from a WSDL file is quite simple in Eclipse. In the *Enterprise Explorer*, expand **WebServiceProjectWAR > wsdl**.

__2. Right-click on **movie_service.wsdl** and select **Web Services | Generate Java bean skeleton**.

The *Web Service* window will appear.

This wizard will generate the required components, including a Java bean skeleton class to which we can add our required business logic.



Web Service

Web Services

Select a service implementation or definition and move the sliders to set the level of service and client

Web service type: Top down Java bean Web Service

Service definition: /WebServiceProjectWAR/wsdl/movie_service.wsdl

Start service

Configuration:

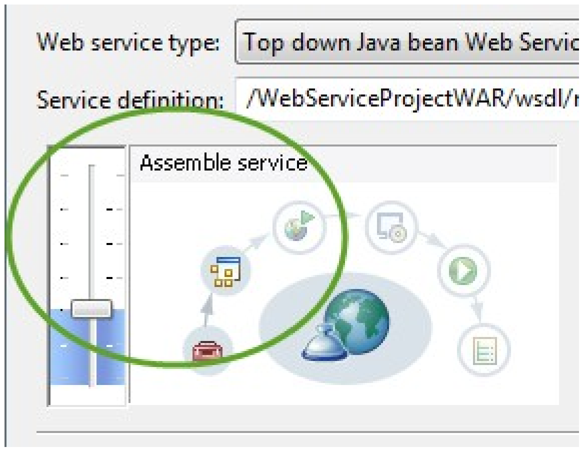
Server runtime: WebSphere Application Server v8.5

Web service runtime: IBM WebSphere JAX-WS

Service project: WebServiceProjectWAR

Service EAR project: WebServiceProjectEAR

__3. Drag the top slider down to **Assemble**.



__4. Click **Finish**.

__5. If you get a Results message just click **OK**.

Eclipse will go about generating the code.

Note. If fails means that you did something wrong in the previous Lab, delete the **wsdl** folder, publish the server, stop the server and then do the previous Lab again.

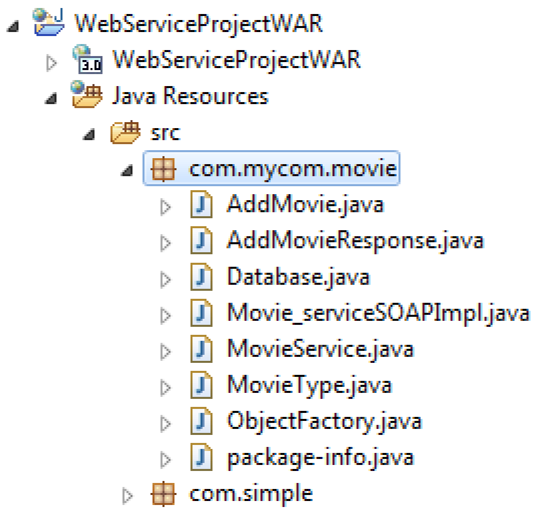
Eventually, a code editor window will open. The code generation is complete! Ignore the code in the editor window for now. Let us take a look at the generated files.

Part 2 - Examine the Generated Artifacts

__1. First, the project needs to be "refreshed" so the generated components will be picked up by Eclipse. In the *Enterprise Explorer,* right click **WebServiceProjectWAR** and select **Refresh**.

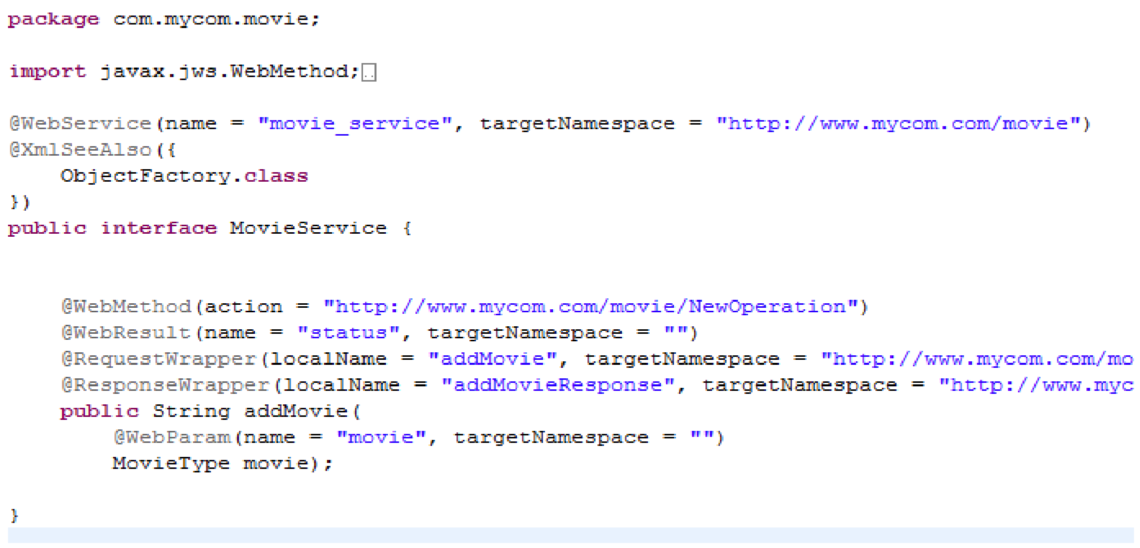
__2. Expand **WebServiceProjectWAR | Java Resources | src | com.mycom.movie**

Notice that package **com.mycom.movie** has been created. This **com.mycom.movie** package contains the service interface – **MovieService.java**. The service interface directly corresponds with the <portType> in the WSDL. They are both abstract representation of a service without any reference to any specific implementation. There may be several different implementations that follow the same abstract contract.



```
WebServiceProjectWAR
    WebServiceProjectWAR
    Java Resources
        src
            com.mycom.movie
                AddMovie.java
                AddMovieResponse.java
                Database.java
                Movie_serviceSOAPImpl.java
                MovieService.java
                MovieType.java
                ObjectFactory.java
                package-info.java
            com.simple
```

__3. Open **MovieService.java**

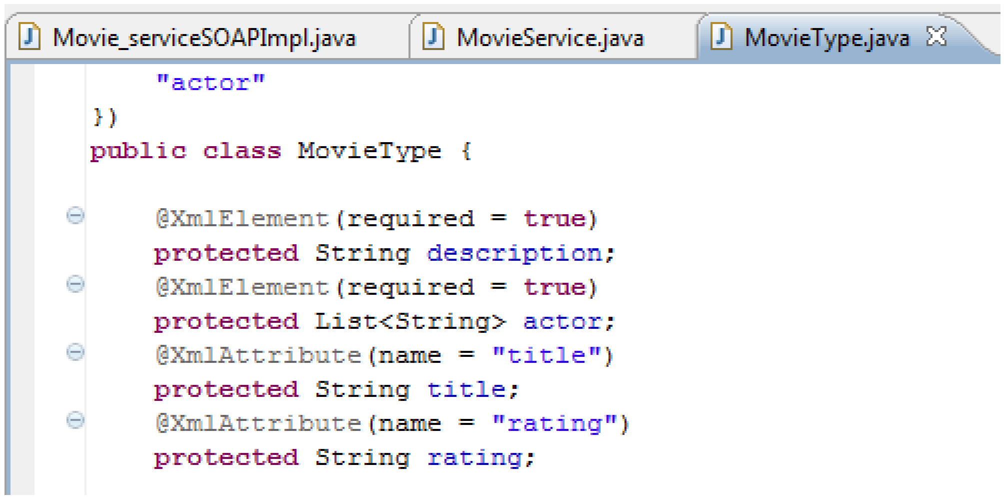


```
package com.mycom.movie;

import javax.jws.WebMethod;

@WebService(name = "movie_service", targetNamespace = "http://www.mycom.com/movie")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface MovieService {


    @WebMethod(action = "http://www.mycom.com/movie/NewOperation")
    @WebResult(name = "status", targetNamespace = "")
    @RequestWrapper(localName = "addMovie", targetNamespace = "http://www.mycom.com/mo
    @ResponseWrapper(localName = "addMovieResponse", targetNamespace = "http://www.myc
    public String addMovie(
        @WebParam(name = "movie", targetNamespace = "")
        MovieType movie);

}
```

Notice the method **addMovie** matches the same signature as defined in the WSDL file; makes sense really, because this class actually implements the WSDL!

Note that this interface has been annotated to control the namespace, method parameters and return types. This way, the generated service will accurately conform to the WSDL.

__4. Next, open **MovieType.java**



```
        "actor"
    })
    public class MovieType {

        @XmlElement(required = true)
        protected String description;
        @XmlElement(required = true)
        protected List<String> actor;
        @XmlAttribute(name = "title")
        protected String title;
        @XmlAttribute(name = "rating")
        protected String rating;
```

The **MovieType** class represents the <movieType> element defined in the WSDL. Notice that various JAXB annotations have been used to accurately map the Java class with the XML schema. If we had chosen the bottom-up development approach, we would have had to type all of them in manually to meet the requirements of the WSDL.

Finally, look at the **AddMovie.java** and **AddMovieResponse.java**. They represent the input and output data for the **addMovie** operation. They are wrapper types only. They are created to conform to the document literal wrapped style of SOAP. As a result, the actual **addMovie**() method in the **MovieService** interface does not use them and uses the inner types only (such as **MovieType**).

The use of the inner types in our implementation code allows us to deal with object level data instead of having to deal with raw XML/SOAP. If the names of these classes look somewhat familiar, they should; this is the result of running the JAXB binding!

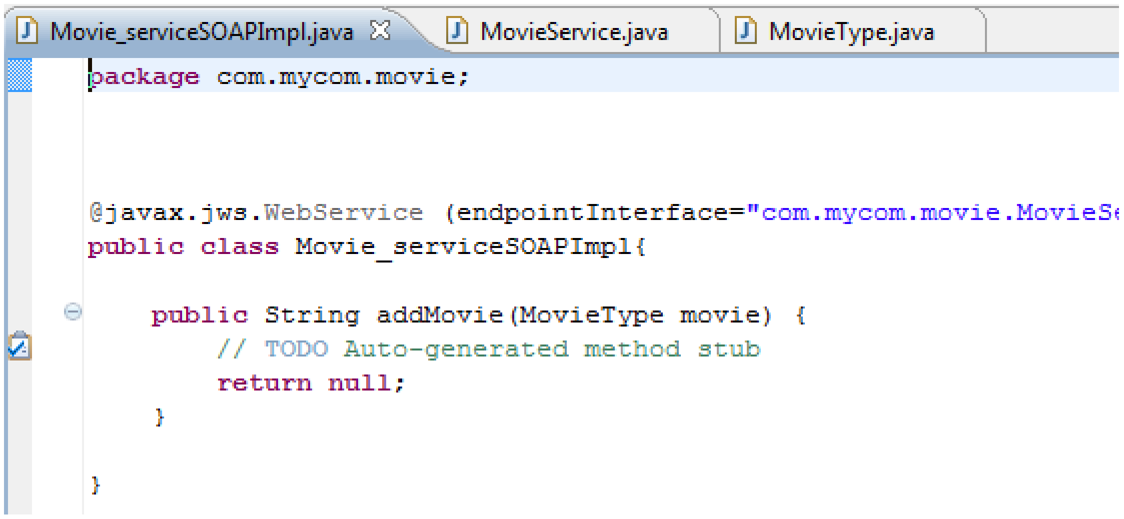
Remember that these objects will be marshaled/unmarshaled from XML SOAP messages.

Part 3 - Create the Implementation Class

Now that we've seen the generated classes, we can get down to coding the implementation.

__1. Open **Movie_serviceSOAPImpl.java**.

This was the first class that was opened in the editor automatically by Eclipse when we generated the service.



```
package com.mycom.movie;



@javax.jws.WebService (endpointInterface="com.mycom.movie.MovieSe
public class Movie_serviceSOAPImpl{

    public String addMovie(MovieType movie) {
        // TODO Auto-generated method stub
        return null;
    }

}
```

This is the implementation class. It is here that we will place our actual web service business logic implementation code.

Notice the appearance of the **@WebService** annotation at the head of the class. Examine the code and check out the various arguments to the annotations, such as the *targetNamespace*, the *serviceName*, and the *portName*.

Also notice the one method present in the class: **addMovie(MovieType movie)**. Note that this corresponds to the operation in the service of the WSDL file. In short, this method should contain the actual code for the addMovie operation.

__2. Set the code for **addMovie** method as shown in boldface below:

```
public String addMovie(MovieType movie) {
          System.out.println("Recieved a movie:");
          System.out.println("Title:  " + movie.getTitle());
          System.out.println("Rating:  " + movie.getRating());

          List<String> actors = movie.getActor();
                   for(String actor : actors) {
                            System.out.println("Actor:  " + actor);
                   }

          System.out.println("Description:  " + movie.getDescription());

          return ("Successfully added movie " + movie.getTitle());
}
```

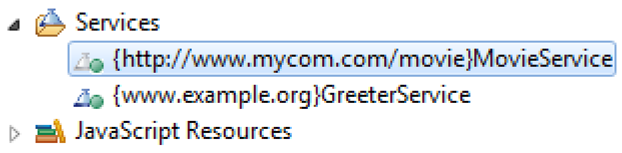
Notice the use of the **MovieType** class, which is one of the generated binding classes.

__3. Organize imports and select **java.util.List** when prompted.

__4. Save the code. There should be no errors.

Our web service implementation code has been completed!

__5. In the *Enterprise Explorer*, expand the *Services* node in the **WebServiceProjectWAR**


Services
{http://www.mycom.com/movie}MovieService
{www.example.org}GreeterService
JavaScript Resources

Notice that our newly created service is listed.

We have now completed the implementation!

Part 4 - Test the Service

We can now test our newly written service. As usual, we need to push our code changes out to the server before we can test.

__1. Close all open editor windows.

__2. Start the server if it's not running.

__3. Publish the server. Once the server is synchronized, you will start the WSE on our newly created MovieService.

__4. Right-click on the service under the *Services* node in the *Enterprise Explorer* and select **Test with Web Services Explorer**.

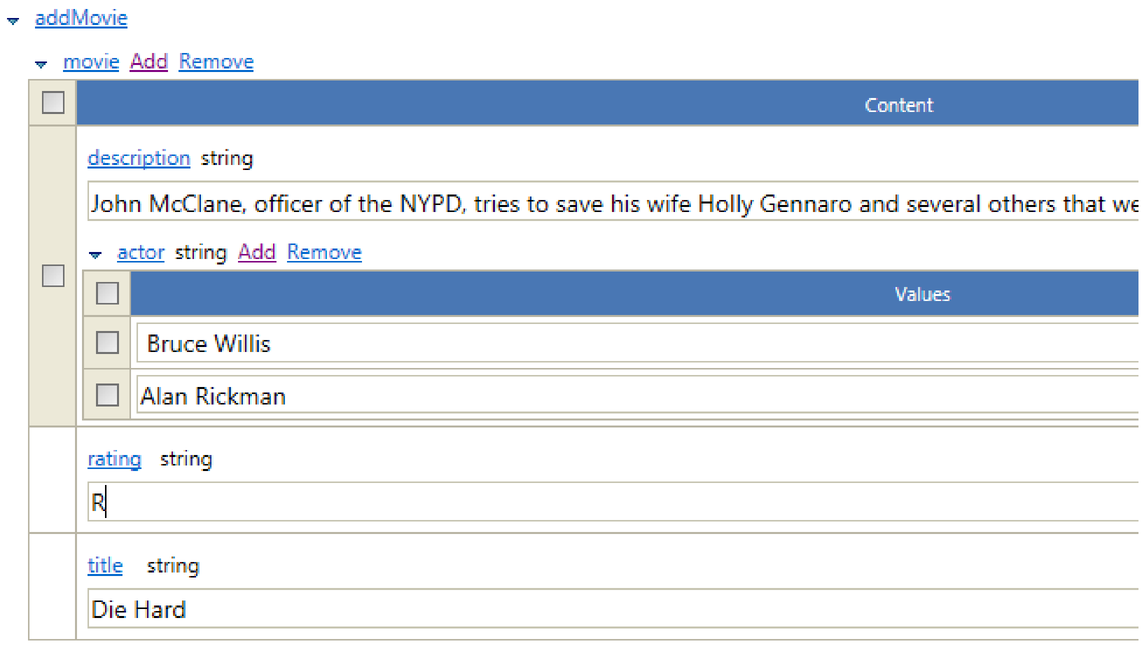
The WSE will open.

__5. Click the **addMovie** link.

__6. Click **Add**.

The WSE now builds a form that displays appropriate fields for the service. Notice that we can enter the description, title, rating and actors.

__7. Enter some sample data. (To add multiple actors, click the **Add** button next to **actor**).



John McClane, officer of the NYPD, tries to save his wife Holly Gennaro and several others that we

Bruce Willis

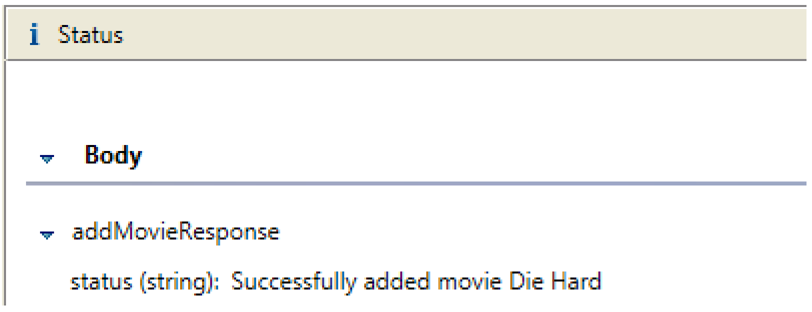
Alan Rickman

R

Die Hard

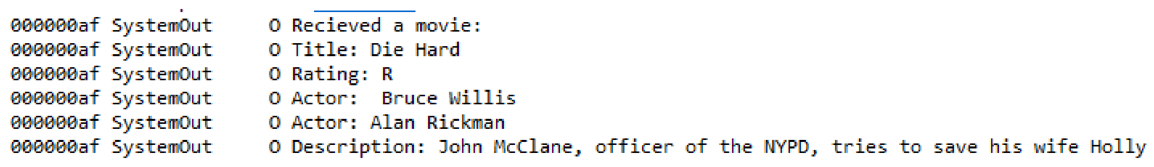
__8. Click **Go**. The service will be invoked.

First of all, notice the *Status* pane. This displays what the service returns to the invoker (in our case, the WSE). [You may need to click Form at the top right of Status]



status (string): Successfully added movie Die Hard

Now, let's take a look at the server console.



```
000000af SystemOut     O Recieved a movie:
000000af SystemOut     O Title: Die Hard
000000af SystemOut     O Rating: R
000000af SystemOut     O Actor:  Bruce Willis
000000af SystemOut     O Actor: Alan Rickman
000000af SystemOut     O Description: John McClane, officer of the NYPD, tries to save his wife Holly
```

These statements are the result of the implementation code executing. (Note that if we were very industrious we could change the implementation code to do something like write to a database instead of just printing to the console)

Our web service works!

__9. Try adding a few more movies and make sure everything works.

Congratulations! You have created a top-down web service.

__10. Close all.

Part 5 - Review

In this lab, you created a top-down web service.   Compare this to the bottom-up approach we took in an earlier lab wherein we started with the implementation class and generated the WSDL from it.

You started off by using Eclipse to generate implementation classes from the **movie_service.wsdl** file.

You then went and added business logic code to the implementation class. The implementation needed to use the generated JAXB binding classes to access the input and output data.

You then deployed and tested the web service and saw the implementation working and that the implementation is compatible with the original WSDL file.

Lab 8 - Developing A Web Service Client

So far, we have developed several web services; however, the client for each one has been the WSE – Web Services Explorer. While that has served as a good test platform, it is not indicative of an actual run time environment.

A web service is invoked by a web service client. A web service client is any component that has read the WSDL file and understands how to invoke the service. An example: a standalone Java client. We could write a standalone Java client that invokes the movie service we wrote in the previous lab. The Java client could then be run from anywhere, with the service logic remaining on the server side.

The client might not even be a Java component. It might be a .NET component. As long as the client knows *how* to invoke the service properly, invocation should work without any problems. (Realize that WSE is itself a web service client)

In this lab, we will create a web service client for our movie service.

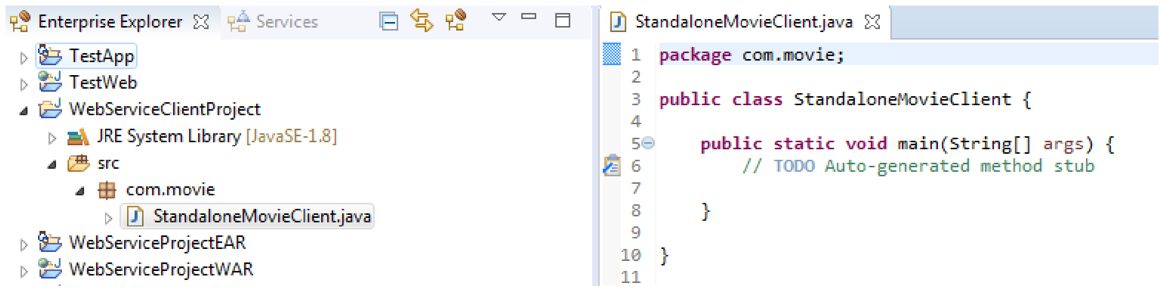
Part 1 - Generate the Client Stub

We will create a stand alone regular Java class to act as a client, called the **StandaloneMovieClient**. The **MovieClient** will be quite simple; it will create an instance of a movie, and it will invoke the **addService** method with the movie as an argument. We should see that on the server side, the usual **System.out** print statements appear.

__1. Create a new **Java Project** called **WebServiceClientProject**

__2. Do *not* switch over to the Java perspective.

__3. In the **WebServiceClientProject**, create a new Java class called **StandaloneMovieClient** having **com.movie** as package with a **main** method.



```
package com.movie;

public class StandaloneMovieClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

So now, we can start coding our **main** method. Remember, the first thing we need to do is to create an instance of a movie. Then, we need to invoke the service.

But ... how do we create a movie? We know that the service requires a **movieType**, so what are we supposed to do now? Create a new POJO object that represent a movie type and use that here?

On top of that, we do not actually know how to invoke the service. We know the service is running on a server at an address/port and uses SOAP messages to communicate. Does this mean our client has to write socket code and SOAP messages?

The answer to both these questions is **No**.

While it would be possible to write the type class yourself, it would potentially be error prone; what if the type we create is not compliant with what the service is expecting? Also, writing socket code exposing SOAP would require a huge amount of development time. Neither of these approaches is practical.

Fortunately, the web service spec comes to the rescue. What we need to do is generate the *client stub***.**

The client stub is the series of Java classes that a client needs to invoke the service. Given a WSDL file, the generation process will create two main artifacts; all the type classes that the service uses (e.g. the movie type class) as well as a proxy class for the service.

The generated proxy class contains all the networking and SOAP code, thus hiding it from us. All our client has to do is call the business methods on the proxy class, and the proxy class will handle all the network details.

So, once we run the client stub generation process, our client code can just used the generated type classes and the proxy to invoke the service. Simple stuff.

Let us create the client stub now. How do we go about this task?

Eclipse, as usual, provides the answer. Eclipse has tooling built into it to generate the stub for us. We can point Eclipse at a deployed WSDL file, and have it examine the contents. From this examination, Eclipse can build the client stub classes required to invoke the service.

We will have Eclipse to this; we will point it at the WSDL file for our movie service and have it generate the client in our newly created **WebServiceClientProject**.

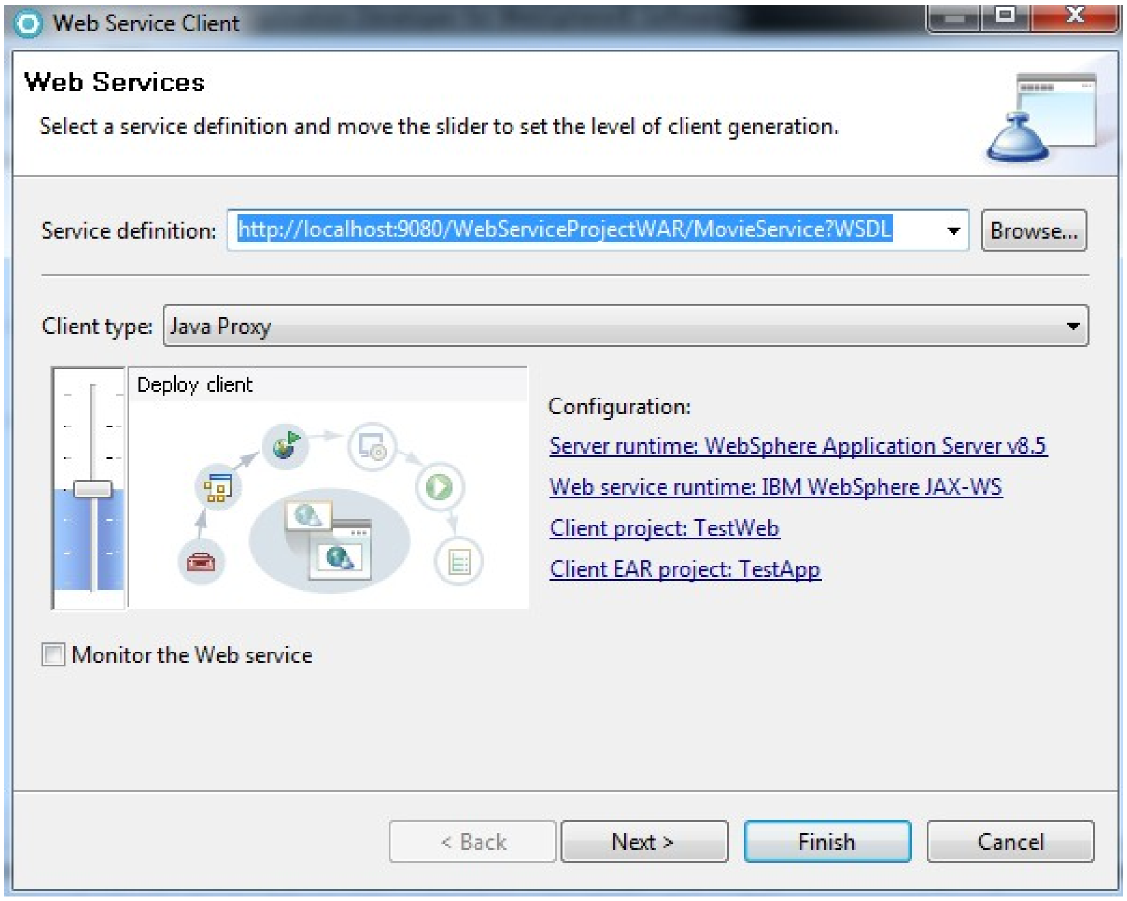
Note that in WebSphere 8.5 version doesn't support creating in the standalone project, for that reason in this part we will use web project to create the client stubs and we will copy over to standalone project.

We need the deployed (i.e. served up) WSDL file for this.

__4. In the *Enterprise Explorer*, expand the ***Services*** node of the **WebServiceProjectWAR**

__5. Right-click on **{http://www.mycom.com/movie}MovieService** and select **Generate > Client**.

The Web Service Client window will appear.


Web Service Client
Web Services
Select a service definition and move the slider to set the level of client generation.
Browse...
Deploy client
Back
Next >
Finish
Cancel

Service definition: http://localhost:9080/WebServiceProjectWAR/MovieService?WSDL

Client type: Java Proxy

Configuration:
Server runtime: WebSphere Application Server v8.5
Web service runtime: IBM WebSphere JAX-WS
Client project: TestWeb
Client EAR project: TestApp

Monitor the Web service

This is a wizard that will generate a client for us.

Notice that the *Service definition* points to a URL. Open a web browser and copy and paste that URL in. What do you see?

__6. On the left, drag the slider down to **Assemble Client**.

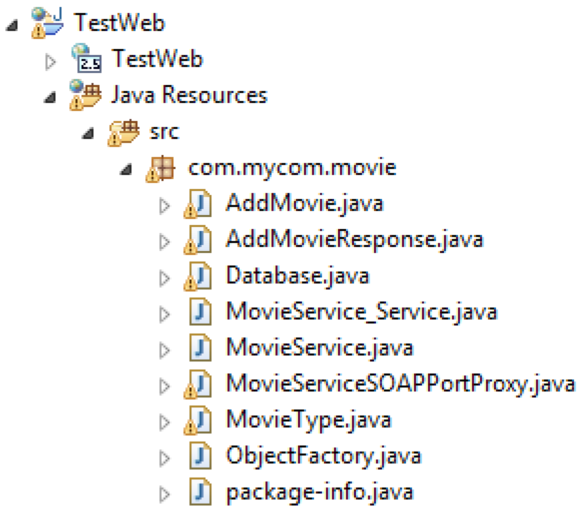
Currently, the wizard is set to generate the client stub in the **TestWeb** project, We want it generated in our newly created **WebServiceClientProject**. Because WebSphere 8.5 doesn't support generating on the standalone project. We will use **TestWeb** and copy over necessary generated stub classes and generated wsdl file to **WebServiceClientProject**.

__7. Leave the client project as TestWeb. Change it to this if it is not set to this value.

__8. Click **Finish** and then click OK. The stub will be generated.

__9. In the *Enterprise Explorer*, right-click on **TestWeb** and select **Refresh**.

__10. Expand the **src** folder in **TestWeb**. Notice a new package **com.mycom.movie** has been created. This package contains our client stub classes! Expand it.



The generated classes will be nearly same as what we have seen in the previous lab. For example, notice the appearance of the JAXB binding classes, like **MovieType**

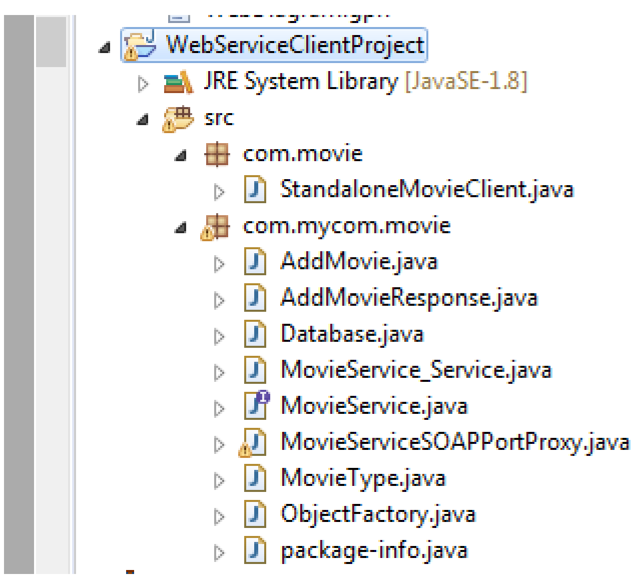
Our **StandaloneMovieClient** Java class will have to make use of these classes to invoke the service.

Let's now copy those stub class to **StandaloneMovieClient**.

__11. In the Enterprise Explorer view, copy **com.mycom.movie** package folder from the src folder of **TestWeb (TestWeb → Java Resources → src)**

__12. Paste the copied content to **WebServiceClientProject** → **src** folder.

__13. Once you copied it should look as shown below:

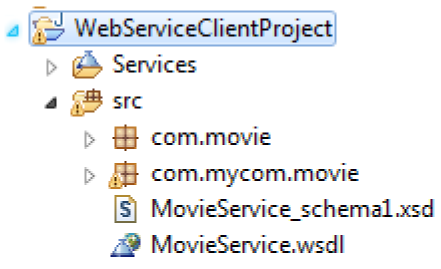

WebServiceClientProject
JRE System Library [JavaSE-1.8]
src
com.movie
StandaloneMovieClient.java
com.mycom.movie
AddMovie.java
AddMovieResponse.java
Database.java
MovieService_Service.java
MovieService.java
MovieServiceSOAPPortProxy.java
MovieType.java
ObjectFactory.java
package-info.java

Now we should copy wsdl file that generated in the above process which point to correct end point of the service.

__14. Expand **TestWeb** project, copy **MovieService.wsdl** and **MovieService_schema1.xsd** from **WebContent** → **WEB-INF** → **wsdl**.

__15. Paste the content to **WebServiceClientProject** → **src** folder. Make sure to copy to the 'src' folder and not in one of the packages. Collapse the packages to hide the code in those packages and verify you can still see the WSDL and XML schema file.

Your project should look as shown below:


WebServiceClientProject
Services
src
com.movie
com.mycom.movie
MovieService_schema1.xsd
MovieService.wsdl

__16. Open the **MovieService_Service.java** file. This is the *client side proxy class*.

Notice how the @**WebServiceClient** annotation is used. The proxy class is entirely JAX-WS standards based. As a result, it should work in most Java EE vendor's platforms.

Currently though the generated Java code refers to the location of the WSDL as being in the 'WEB-INF/wsdl' folder. When we copied the WSDL and Schema files they were copied to the root of the project source folder. We need to update the references so the code will work. There are 3 places this should be done.

__17. Find the following code in the @WebServiceClient annotation:

```
, wsdlLocation = "WEB-INF/wsdl/MovieService.wsdl")
```

__18. Modify it to be:

```
, wsdlLocation = "MovieService.wsdl")
```

__19. Find the following code in a static initialization block:

```
MOVIESERVICE_WSDL_LOCATION =
com.mycom.movie.MovieService_Service.class.getResource(
"/WEB-INF/wsdl/MovieService.wsdl");
```

__20. Modify it to be the following, making sure to leave the leading '/':

```
MOVIESERVICE_WSDL_LOCATION =
com.mycom.movie.MovieService_Service.class.getResource(
"/MovieService.wsdl");
```

__21. Find the following code:

```
"Cannot find 'WEB-INF/wsdl/MovieService.wsdl' wsdl. Place the resource
```

__22. Modify it to be:

```
"Cannot find 'MovieService.wsdl' wsdl. Place the resource
```

Note: Technically this last location is in an Exception message so not critical but good to be consistent.

__23. Save the code and make sure there are no errors.

__24. Close all open files.

__25. Publish the server.

Part 2 - Code the Client

We can now write the client code, making use of the generated client stub classes.

__1. Open the source for **StandaloneMovieClient.java**. (which was in the **com.movie** package in the **WebServiceClientProject** project)

__2. We should first create an instance of a movie and populate it with some data. Add the following code to the **main** method.

```
public static void main(String[] args) {
          MovieType movie = new MovieType();
          movie.setTitle("The Big Lebowski");
          movie.setRating("PG-13");
          movie.setDescription("Let's Go Bowling");
          movie.getActor().add("Jeff Bridges");
          movie.getActor().add("John Goodman");
}
```

Not much special here; we merely instantiate and populate the fields of a movie type instance. One issue is that there is not an **addActor** method in the generated class; we have to **get** the actor List, and manually add the actor to it via the List API. This is, unfortunately, a side effect of the JAXB protocol. While it works very well, it isn't terrific.

__3. Organize imports, to import the **MovieType** class. There should be no errors.

Now that we have populated the movieType, we should be able to invoke the service. To do so, we first need to get a handle to the service class.

__4. Add the following code to the main method, <u>after</u> the **movieType** code:

```
          movie.getActor().add("Jeff Bridges");
          movie.getActor().add("John Goodman");

          MovieService_Service mss = new MovieService_Service();
          MovieService service = mss.getMovieServiceSOAPPort();
```

The **service** variable can now be used as a handle to the actual service. (Note that we are making use of the generated client proxy classes here)

__5. Organize imports. There should be no errors.

__6. We can now go ahead and invoke the service. Add the following code at the end of the main method.

```
            MovieService_Service mss = new MovieService_Service();
            MovieService service = mss.getMovieServiceSOAPPort();

            System.out.println(service.addMovie(movie));
}
```

We invoke the service (by calling **service.addMovie)** and print the result to the console. So, in the console window for this Java program, we should see the service's return. In the console for the server, however, we should see the movie details printed out.

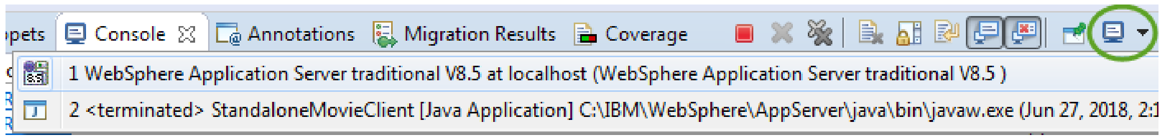
__7. Save the code. We have completed coding our client.

Part 3 - Run the Client

Let us now run the client and see if it worked.

__1. Run the **StandaloneMovieClient** class as a Java application. To do this, right click anywhere on the source code and select **Run As → Java Application**.

__2. Switch to the **Console** for the class by clicking in the icon to display all consoles running and selecting the **<terminated> StandaloneMovieClient** console.



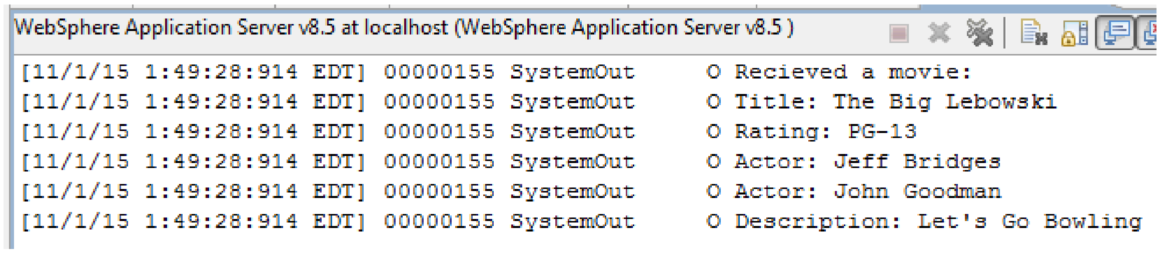
1 WebSphere Application Server traditional V8.5 at localhost (WebSphere Application Server traditional V8.5)
2 <terminated> StandaloneMovieClient [Java Application] C:\IBM\WebSphere\AppServer\java\bin\javaw.exe (Jun 27, 2018, 2:1

The console window for this class will appear.

```
Successfully added movie The Big Lebowski
```

Looks like it worked!

__3. Check the console for the server. (To view it, click the button in the console tool bar. It will swap between the console for the server and for our standalone program)


WebSphere Application Server v8.5 at localhost (WebSphere Application Server v8.5)

```
[11/1/15 1:49:28:914 EDT] 00000155 SystemOut     O Recieved a movie:
[11/1/15 1:49:28:914 EDT] 00000155 SystemOut     O Title: The Big Lebowski
[11/1/15 1:49:28:914 EDT] 00000155 SystemOut     O Rating: PG-13
[11/1/15 1:49:28:914 EDT] 00000155 SystemOut     O Actor: Jeff Bridges
[11/1/15 1:49:28:914 EDT] 00000155 SystemOut     O Actor: John Goodman
[11/1/15 1:49:28:914 EDT] 00000155 SystemOut     O Description: Let's Go Bowling
```

Success! So, our standalone Java client has successfully invoked the service.

How much actual "web service" code did our client have to write? Well, none really. Once the generated classes were created for us, the web service layer was essentially invisible. The client Java code just looks like regular Java invocations. The fact that the code we are invoking is a web service is essentially abstracted away from us.

Part 4 - Adding A Web Layer

Using a standalone Java client may work, but is not very interactive. In order to add another movie, we would have to edit the Java source code and run it again. Fine for developers, but not so good for presentation.

To spruce things up a bit, a better presentation layer could be used. Ideally, in order to add a movie, we should have the user operate a web browser. The browser would display a form with fields for the movie data. The user could then click submit on the form, and that data would be passed to the service for invocation.

We will actually go ahead and code this presentation layer now. The architecture will be very simple. As described, we will provide an HTML form that contains fields for the movie info. A user can browse to it and enter appropriate data. When the user clicks submit, the form will POST to a servlet. Here, the servlet will invoke the web service via a generated client proxy.

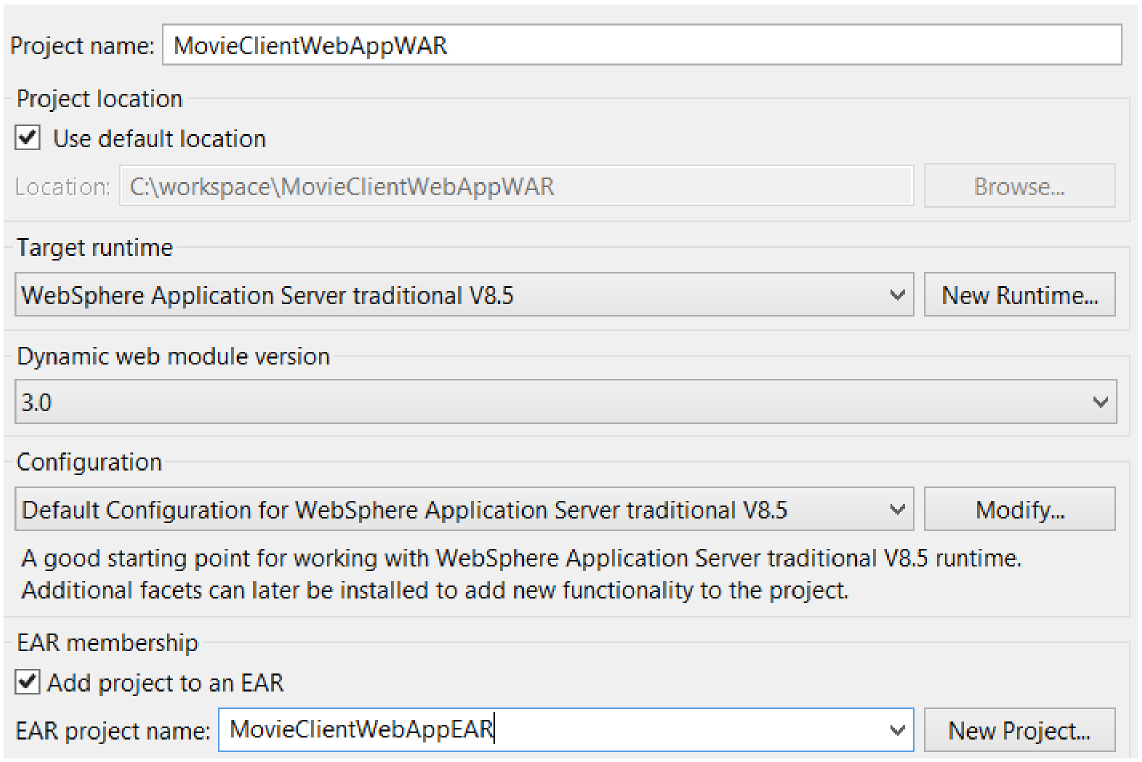
This may sound complex but is actually quite simple; the HTML form serves as front end to the servlet, which acts as the actual web service client. The servlet code will make use of code very similar to the StandaloneMovieClient class we just wrote; the main difference will be that instead of populating the movieType's fields from scratch, it will use the data submitted to the form.

We will create a fresh project to keep it separate from our other projects.

__1. Close all open editor windows. (**Ctrl-Shift-W**)

__2. Create a new **Dynamic Web Project** called **MovieClientWebAppWAR**

__3. Enter **MovieClientWebAppEAR** as EAR project name



Project name: MovieClientWebAppWAR

Project location

☑ Use default location

Location: C:\workspace\MovieClientWebAppWAR

Target runtime

WebSphere Application Server traditional V8.5

Dynamic web module version

3.0

Configuration

Default Configuration for WebSphere Application Server traditional V8.5

A good starting point for working with WebSphere Application Server traditional V8.5 runtime.
Additional facets can later be installed to add new functionality to the project.

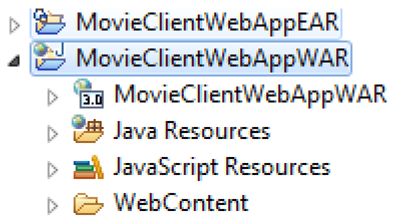
EAR membership

☑ Add project to an EAR

EAR project name: MovieClientWebAppEAR

__4. Click **Finish**.

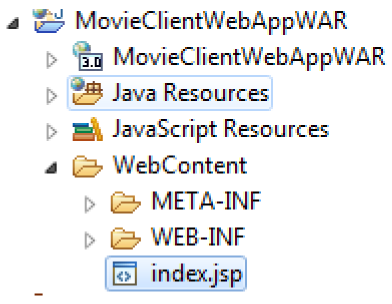
Make sure the projects are visible in the *Enterprise Explorer.*



```
▷ MovieClientWebAppEAR
◢ MovieClientWebAppWAR
    ▷ MovieClientWebAppWAR
    ▷ Java Resources
    ▷ JavaScript Resources
    ▷ WebContent
```

__5. Navigate to **C:\LabFiles** and locate file **index.jsp**.

__6. Copy this file into the **WebContent** folder of the **MovieClientWebAppWAR** project.



__7. Open up and take look at this **index.jsp** in the editor. It is a very simple form that posts to a servlet called **AddMovieServlet**. It is the servlet that will take the data input from the form, and invoke the service with it. We will code this servlet in a moment.

One thing to note in the JSP, however, is the handling of **actors**. Remember that there can be multiple actors; this makes it difficult to decide how many actor fields to render. So, to keep things simple, we provide only one field and ask the user to enter all the names separated by comma in this one field.

We can now turn attention to the servlet that will invoke the web service. Before we code the servlet, we need to do something else. Our servlet will act as the client for the web service – and that client will need the client stub. So, as before, we need to generate the client stub, but this time in our new web project.

__8. Close the file.

__9. Right click the server and select **Publish**. Remember, we need the deployed WSDL file to generate a client.

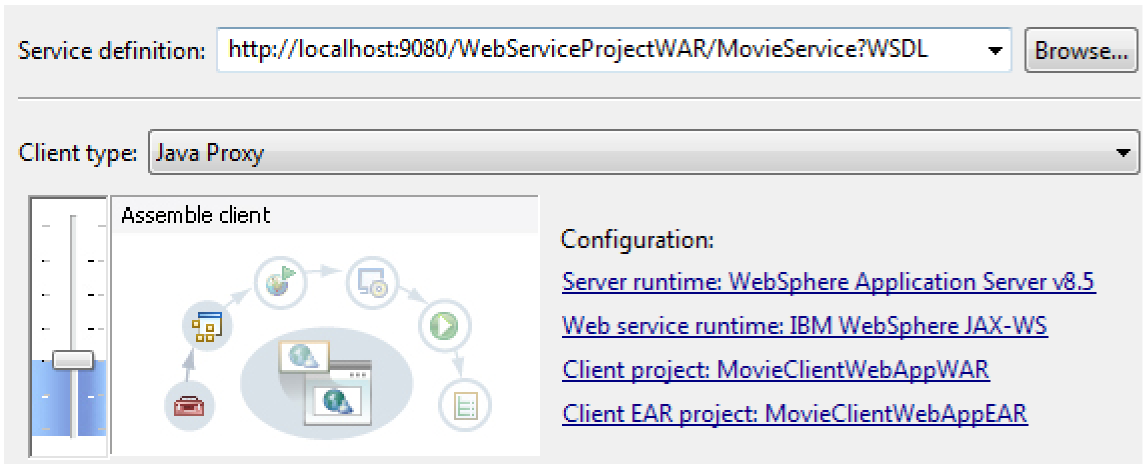
__10. In the *Enterprise Explorer*, expand **WebServiceProjectWAR | Services**.

__11. Right-click on **{http://www.mycom.com/movie}MovieService** and select **Generate | Client**.

The *Web Service Client* window will appear again.

__12. Drag the slider down to **Assemble Client**.

__13. The client project should already be set to *MovieClientWebAppWAR*. If it is not, change it.



Service definition: http://localhost:9080/WebServiceProjectWAR/MovieService?WSDL

Client type: Java Proxy

Assemble client

Configuration:

Server runtime: WebSphere Application Server v8.5

Web service runtime: IBM WebSphere JAX-WS

Client project: MovieClientWebAppWAR

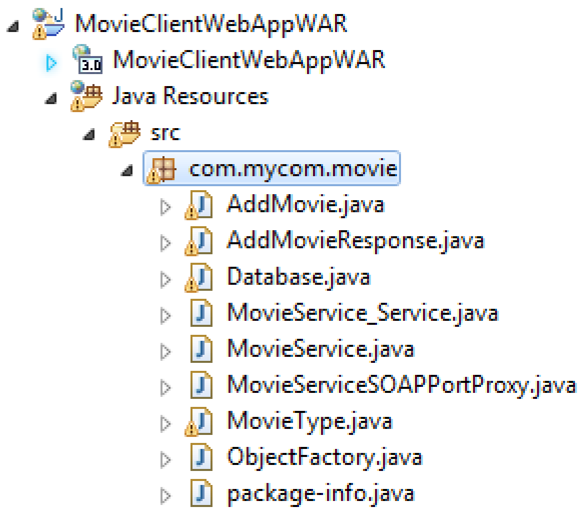
Client EAR project: MovieClientWebAppEAR

__14. Click **Finish** and click **OK**.

The client stub will be generated in the **MovieClientWebAppWAR** project.

__15. Refresh the **MovieClientWebAppWAR** project.

__16. Expand the project and make sure the client stub classes were generated.



```
MovieClientWebAppWAR
  MovieClientWebAppWAR
  Java Resources
    src
      com.mycom.movie
        AddMovie.java
        AddMovieResponse.java
        Database.java
        MovieService_Service.java
        MovieService.java
        MovieServiceSOAPPortProxy.java
        MovieType.java
        ObjectFactory.java
        package-info.java
```

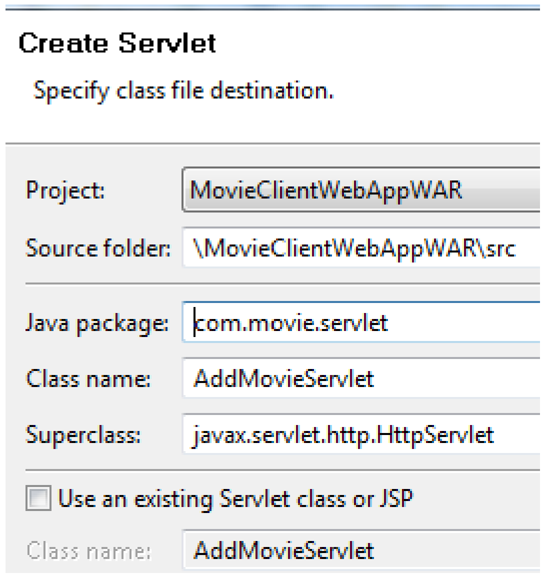
We can now go ahead and code the servlet.

__17. Right click **MovieClientWebAppWAR** and select **New > Other**.

__18. Expand **Web** and select **Servlet**.

__19. Click **Next**.

__20. Enter **AddMovieServlet** as class name and **com.movie.servlet** package.



Create Servlet

Specify class file destination.

Project: MovieClientWebAppWAR

Source folder: \MovieClientWebAppWAR\src

Java package: com.movie.servlet

Class name: AddMovieServlet

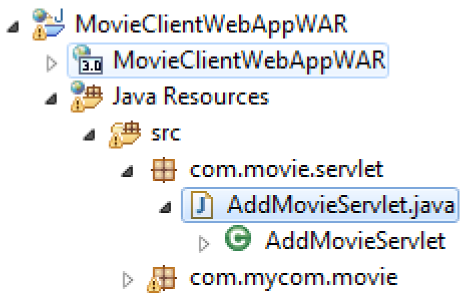
Superclass: javax.servlet.http.HttpServlet

☐ Use an existing Servlet class or JSP

Class name: AddMovieServlet

__21. Click **Finish**.

Note. If you get an error creating the servlet, close the Create Servlet window, delete the created package and try again.



```
MovieClientWebAppWAR
  MovieClientWebAppWAR
  Java Resources
    src
      com.movie.servlet
        AddMovieServlet.java
          AddMovieServlet
      com.mycom.movie
```

__22. An editor will open on the servlet. In the servlet code, locate the **doPost** method. Here, we will now add the code to get the data submitted from the form and create a new movie based on that data.

__23. Add the following code in the **doPost** method replacing the existing code.

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();

MovieType movie = new MovieType();
movie.setTitle(request.getParameter("title"));
movie.setDescription(request.getParameter("description"));
movie.setRating(request.getParameter("rating"));
String[] actors = request.getParameter("actors").split(",");
for(int x = 0; x != actors.length; x++) {
   movie.getActor().add(actors[x]);
}
```

This code is fairly straightforward. We create a new instance of a **MovieType** and then use the data from the form (via the call to **request.getParameter)** to populate it. The only tricky part is tokenizing the actors from the form, using the comma as a separator.

__24. Organize imports.

__25. Now that we have a populated movie, we can make a call to the service! Add the following code in the **doPost** method:

```
MovieService_Service mss = new MovieService_Service();
MovieService service = mss.getMovieServiceSOAPPort();
service.addMovie(movie);
```

This invokes the actual service, in an identical fashion to our standalone client.

__26. Organize imports.

__27. Add the following code in the **doPost** method:

```
out.println("Successfully added a movie: " + movie.getTitle());
```

This is what will be displayed back to the user in the browser.

__28. We are done! Save the code. There should be no errors.

We have completed our web layer. It can be deployed and tested.

Part 5 - Deploy and Test

We can now deploy and test our web client.

__1. Right click on the server and select **Add and Remove...**

__2. Select **MovieClientWebAppEAR** and select **Add**.

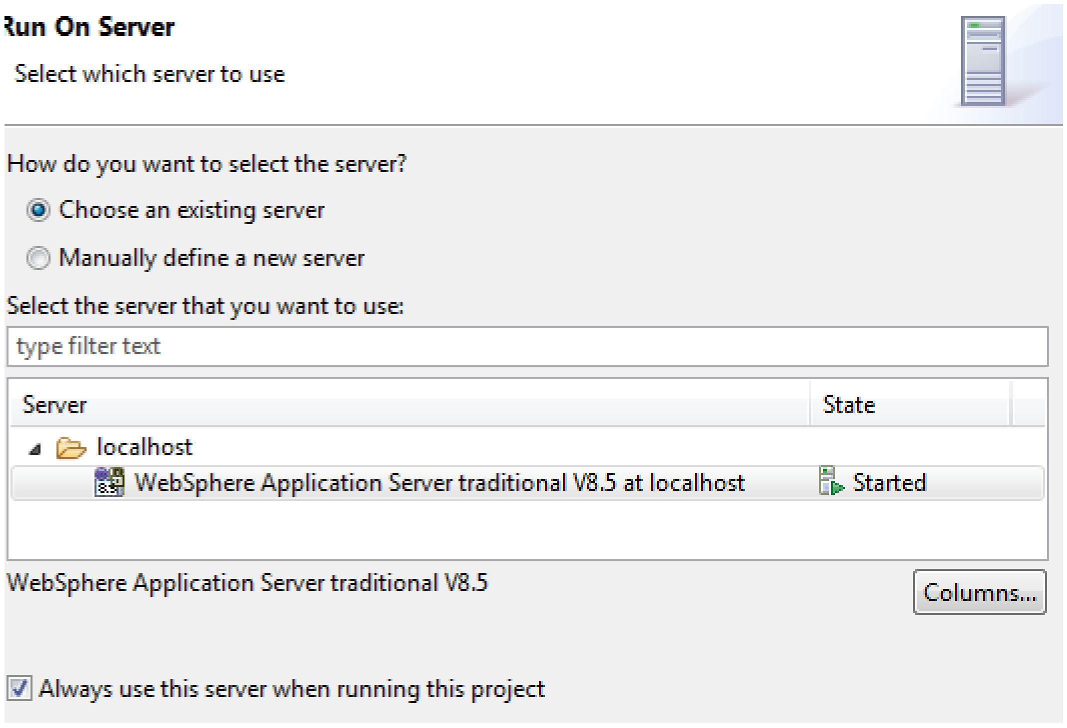
__3. Click **Finish**.

__4. Publish the server.

__5. Right-click on **index.jsp** under **WebContent** in the **MovieClientWebAppWAR** project and select **Run As → Run On Server**.

__6. Make sure your WAS server is selected.

__7. Check the box to always use this server.



ᴚun On Server

Select which server to use

How do you want to select the server?

(●) Choose an existing server

( ) Manually define a new server

Select the server that you want to use:

type filter text

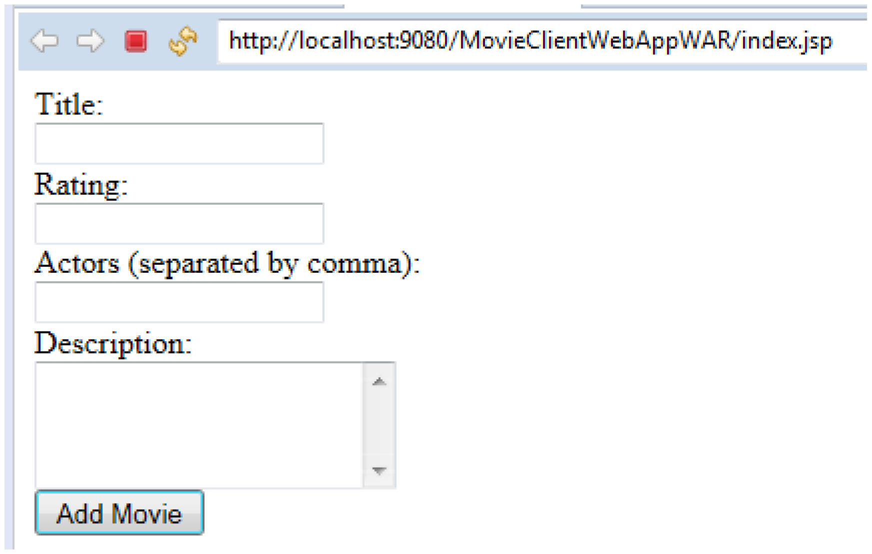
Server	State
localhost	
WebSphere Application Server traditional V8.5 at localhost	Started

WebSphere Application Server traditional V8.5

[✓] Always use this server when running this project

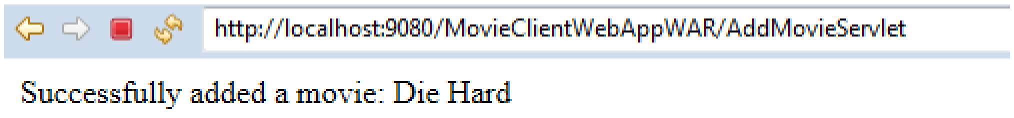
__8. Click **Finish**.

Eclipse will open a browser, displaying our JSP form.


http://localhost:9080/MovieClientWebAppWAR/index.jsp
Title:
Rating:
Actors (separated by comma):
Description:
Add Movie

__9. Enter some sample data and click **Add Movie**.

You should see an appropriate response from the browser.



Successfully added a movie: Die Hard

__10. Check the server console to make sure the appropriate print statements are generated.

```
000000af SystemOut     O Recieved a movie:
000000af SystemOut     O Title: Die Hard
000000af SystemOut     O Rating: R
000000af SystemOut     O Actor:  Bruce Willis
000000af SystemOut     O Actor: Alan Rickman
000000af SystemOut     O Description: John McClane, officer of the NYPD, tries to save his wife Holly
```

Our web client works!

We have successfully created a web front end to a web service!

__11. Close all open editor windows.

Part 6 - Review

In this lab, you examined the concept of web service clients. You:

- Used a WSDL file to form the basis of a client stub. This included generated classes for types as well as a proxy class. You used the Eclipse tools to perform this generation.
- Used the generated type and proxy class from a standalone client. You used the generated type to act as a parameter for the service, and use the proxy class to invoke the service. You did not have to deal with any low level networking or SOAP code.
- Ran the client to see that it indeed invoked the service.
- Create a web-based front end to act as a client for the service. This allowed for more interactive use of the service. There was nothing really web-service specific here; this was mostly regular J2EE servlet coding, but you did have to make use of the WSDL-generated client stub.

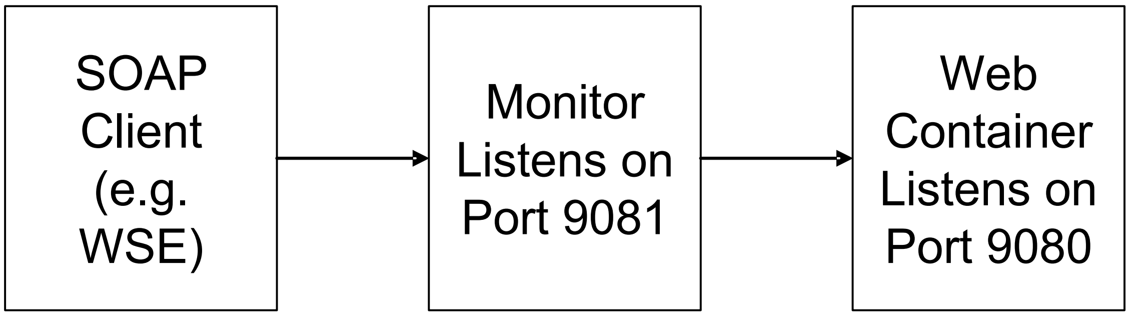
Lab 9 - Monitoring SOAP Messages

We have seen how to create a web service and how to invoke it using a web service client (such as the WSE). However, the details of the actual communication layer (the SOAP messages) have been hidden from us. Aside from a few brief examinations in an earlier lab, we have not been concerned with the underlying SOAP exchanges. This is normal; the whole point of using a web services implementation is to avoid having to deal with low level SOAP messages.

However, it may occasionally be desirable to actually see the messages that are being transmitted back and forward between web service and client. This can assist greatly in situations such as debugging.

In this lab exercise, we will use the TCP Monitor feature of Eclipse to observe the SOAP traffic between our movie service client and the movie service we created in the previous lab.

The basic architecture of the monitor is as simple: it acts as a intermediary (or a proxy server) between a HTTP client and the web container. The monitor listens on a port (say 9081) and forwards all request data to the web container.


SOAP Client (e.g. WSE)
Monitor Listens on Port 9081
Web Container Listens on Port 9080

The monitor captures and displays all traffic before forwarding it onto the actual web container where the service is running. This allows us to view the actual SOAP request being submitted by the client.

Additionally, the monitor also displays all traffic coming back from the web service, allowing us to see the low level SOAP response.

In order to make this happen, we have two main tasks to perform:

- Activate the monitor (a Eclipse tool) and set it to forward requests from 9081 to 9080.
- Change the client to invoke the service on 9081 instead of 9080.

Part 1 - Activate the Monitor

We will first turn on the monitor. Remember, we want it to listen on port 9081, and have it forward all requests to port 9080 (which is where WebSphere is listening).

__1. Start the server if it is not running. The server must be running to activate the monitor.

__2. Open the *Servers* view.

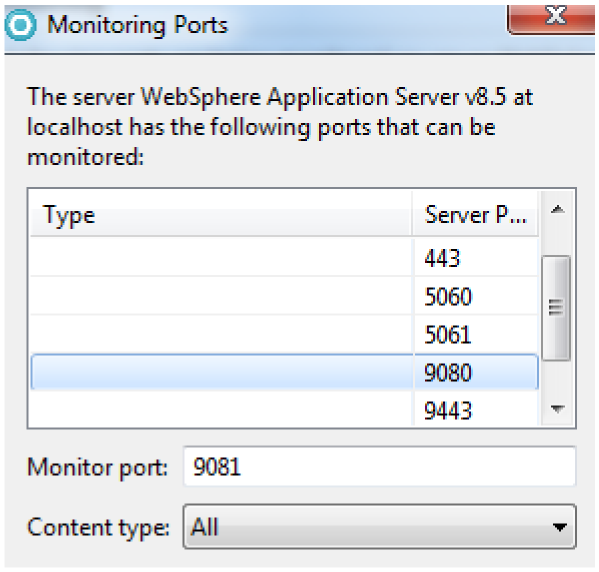
__3. Right-click on the server and select **Monitoring | Properties**.

The *Properties* window will appear, opening to the *Monitoring* screen.

__4. Click the **Add...** button.

The *Monitoring Ports* window will appear.

__5. Select **9080** in the list. This is the web container port on which our web services are running.



The server WebSphere Application Server v8.5 at localhost has the following ports that can be monitored:

Type	Server P...
	443
	5060
	5061
	9080
	9443

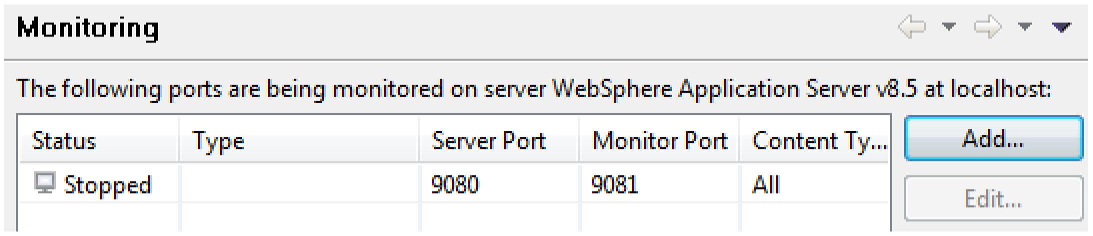
Monitor port: 9081

Content type: All

Note that the *Monitor port* changes to **9081**. Our client will have to attempt to invoke the service at this port; the monitor which is listening on this port, will then forward the request onto 9080, but not before showing us the back-and-forth traffic.

__6. Click **OK**.

Back at the *Properties* window, the newly added monitor should be listed.



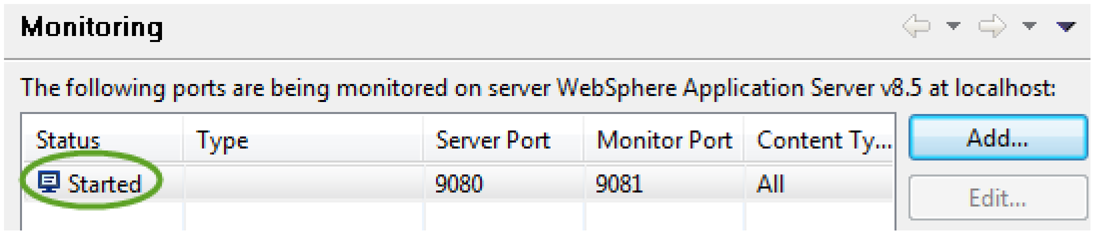
The following ports are being monitored on server WebSphere Application Server v8.5 at localhost:

Status	Type	Server Port	Monitor Port	Content Ty...
Stopped		9080	9081	All

It is not currently running.

__7. Select the monitor and click **Start**.

The monitor's status should become *Started*.



The following ports are being monitored on server WebSphere Application Server v8.5 at localhost:

Status	Type	Server Port	Monitor Port	Content Ty...
Started		9080	9081	All

__8. Click **Apply and Close**.

The monitor has been created and started. Any traffic submitted to 9081 will now be intercepted for display by the monitor.

Part 2 - Update the Client

The next thing to do is to get the client to submit its requests to the service via port 9081 instead of port 9080.

How do we do this? Well, first of all, let us figure out how the web service client knows to use port 9080 in the first place. The answer is simple: the **WSDL** file.

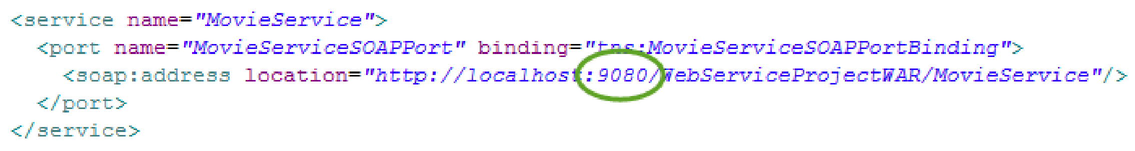
Recall that before the web service client could invoke the service, we had to generate the client stub for the service, based on the WSDL file for the service. Eclipse examined the WSDL file and figured out what operations were there, as well as what parameters those operations used. In addition to that, the Eclipse also looked at the **<wsdlsoap:address**> element of the WSDL file to figure out the address of the service.

Let us take a look at this WSDL again.

__1. In the *Enterprise Explorer*, expand **WebServiceProjectWAR | Services**.

__2. Right-click on **{http://www.mycom.com/movie}MovieService** and select **Show | WSDL Interface**.

__3. Scroll to the bottom of the WSDL file and locate the **<soap:address>** tag.



```
<service name="MovieService">
  <port name="MovieServiceSOAPPort" binding="tns:MovieServiceSOAPPortBinding">
    <soap:address location="http://localhost:9080/WebServiceProjectWAR/MovieService"/>
  </port>
</service>
```

Notice that the location there is specified; the service "lives" at the URL

```
http://localhost:9080/WebServiceProjectWAR/MovieService
```

Any client wishing to invoke this service will send a SOAP over HTTP request to that address. (Try punching that URL into a browser. What do you see?)

So, this is how the web service client figures out that address to connect to. The client stub that was generated for us by Eclipse (by examining this WSDL file) has been hard-coded to connect to this address. This is *not* what we want. We want our client stub to use port **9081**.

So what can we do?

There are two options:

- The first option would be to regenerate the client. We would have to save a local copy of the WSDL file, and edit the <soap:address location> URL to point to 9081, and then run the Eclipse client generation tool on the modified WSDL file. While this would work in theory, it is very cumbersome and error prone.
- A second (and much better) option is to use the JAX-WS API at client invocation time to over-ride the port address. This means we would not need to change our client stub; our web service client, however, would need to add a few lines of code before invoking the **addMovie** operation.

Let us take the second approach now.

__4. Close the WSDL file.

__5. Open **AddMovieServlet.java** located in the **MovieClientWebAppWAR** project.

__6. Add the following lines to the **doPost** method. Make sure this code is added right before the call to **service.addMovie(movie)** line.

```
BindingProvider bp = (BindingProvider) service;
Map<String, Object> context = bp.getRequestContext();
context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"http://localhost:9081/WebServiceProjectWAR/MovieService");
```

Essentially, we are type casting the **MovieService** object into **javax.xml.ws.BindingProvider**. This interface gives us access to the SOAP endpoint properties. We are setting the endpoint URL property in the request context.

__7. Organize imports. Select **java.util.Map** and **javax.xml.ws.BindingProvider** when prompted.

__8. Save the file. There should be no errors. (If a *Publishing Error* notification appears, merely ignore it)

We are done updating our client. Now, when the servlet attempts to invoke the service, it will go through port **9081**, which is exactly what we want.

Part 3 - Test

Let us now invoke the service. If all goes well, any traffic the web service encounters will be displayed in the TCP/IP monitor.

__1. Publish the server.

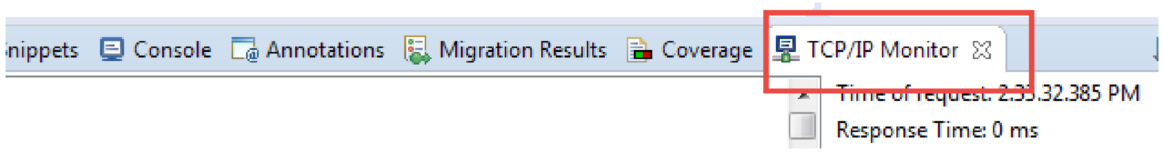
__2. Run the JSP. To do this, right click the MovieService under Services and select **Test with Web Services Explorer**.

__3. Click **Add movie**.

__4. Click **Add** and fill the page.

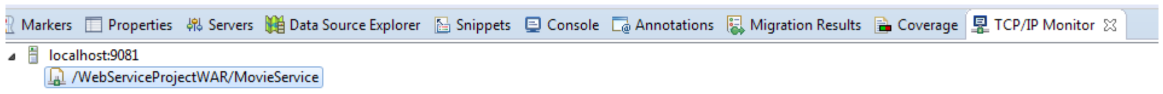
__5. Click **Go**.

__6. In Eclipse, down by the *Console* view, a new ***TCP/IP Monitor*** tab should have appeared. Click on its tab.



Time of request: 2:33.32.385 PM
Response Time: 0 ms

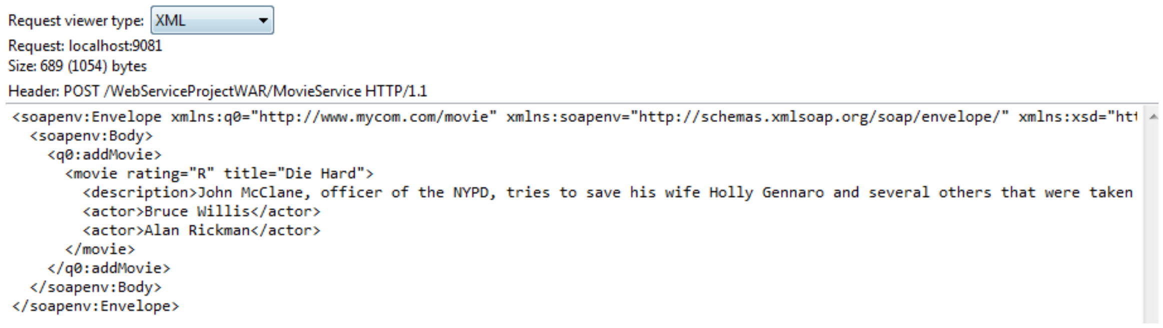
__7. Maximize the ***TCP/IP Monitor*** view by double-clicking on its tab.



This is the TCP/IP Monitor, and it has indeed intercepted our traffic to/from the service. The bottom left pane represents what the client sent to the service, and the bottom right pane shows what the service returned.

__8. In both of these panes, change the **byte** drop-down to say **XML**. This will make the formatting a little easier to read.

__9. Examine the **request** message.



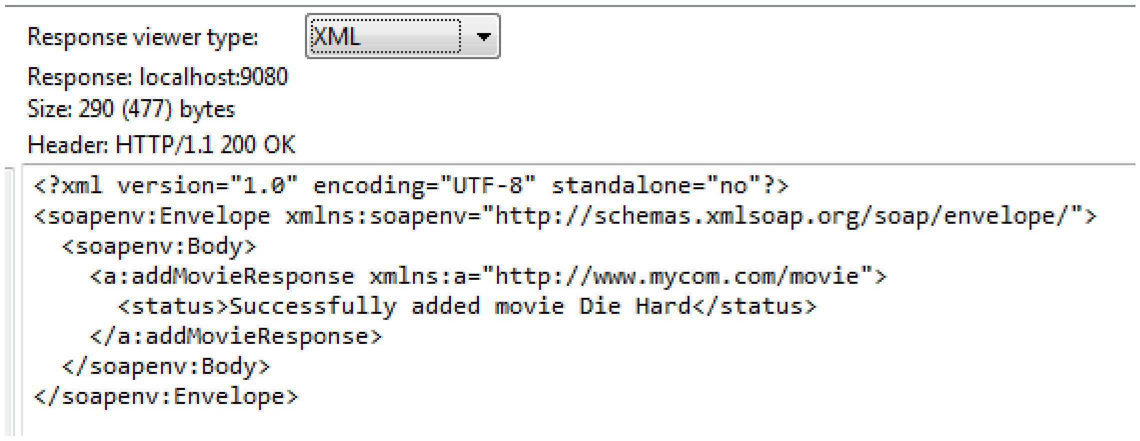
Request viewer type: XML
Request: localhost:9081
Size: 689 (1054) bytes
Header: POST /WebServiceProjectWAR/MovieService HTTP/1.1

```
<soapenv:Envelope xmlns:q0="http://www.mycom.com/movie" xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="htt
  <soapenv:Body>
    <q0:addMovie>
      <movie rating="R" title="Die Hard">
        <description>John McClane, officer of the NYPD, tries to save his wife Holly Gennaro and several others that were taken
        <actor>Bruce Willis</actor>
        <actor>Alan Rickman</actor>
      </movie>
    </q0:addMovie>
  </soapenv:Body>
</soapenv:Envelope>
```

Look at the SOAP syntax, and notice how the movie parameters are being submitted. This is JAXB in action.

__10. Now, look at the response message.

Notice how SOAP is encoding the actual service response.



Response viewer type: XML
Response: localhost:9080
Size: 290 (477) bytes
Header: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <a:addMovieResponse xmlns:a="http://www.mycom.com/movie">
      <status>Successfully added movie Die Hard</status>
    </a:addMovieResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Congratulations! You have successfully setup up monitoring to view low level SOAP communication.

__11. Return the *TCP/IP Monitor* view to its original size by double-clicking on its tab.

__12. Select the **Server** tab.

__13. Right click on the server and select **Monitoring → Properties**.

__14. Click **Stop**.

__15. Click **Apply and Close**.

__16. Close all.

Part 4 - Review

In this lab, you used the TCP/IP Monitor to examine SOAP messages. You saw:

- How to start the TCP/IP Monitor. This tool can be useful in debugging problems with SOAP communication.
- How to dynamically change the SOAP endpoint URL from the client. The default URL is taken from the WSDL file.
- How to validate the SOAP communication for WS-I compatibility.

Lab 10 - Controlling JAXB Mappings

So far, you have seen how to create a simple bottom-up web service. You have seen that an annotated implementation class with some methods on it can be turned into a web service.

The methods we have written so far, however, have been quite simple in the sense that they have only been using simple types for parameters and returns. Consider the **sayHello** method you wrote earlier; it took a string as a parameter and returned a string.

A string is a very simple type; but what happens if the method returns a complex type, in the form of a JavaBean, with fields? How will the JavaBean be mapped to XML schema?

Let us say that we create a new Java bean called **Person** which has a field called *name*. When we invoke a method on the **Greeter** class, we want it to return an instance of that Person object, with a populated name field. From a Java perspective, we want to be dealing with objects. But what happens at the web service level? Web services deal with SOAP messages in XML. What happens to our Java objects?

This is where JAXB comes to the rescue. Remember that JAXB allows for the creation of *binding classes* which represent XML forms of Java objects. Under the hood, the web service engine will be using JAXB to handle XML marshaling/unmarshaling for us. We do not have to know anything about JAXB, however – the web service engine will do all the JAXB work for us.

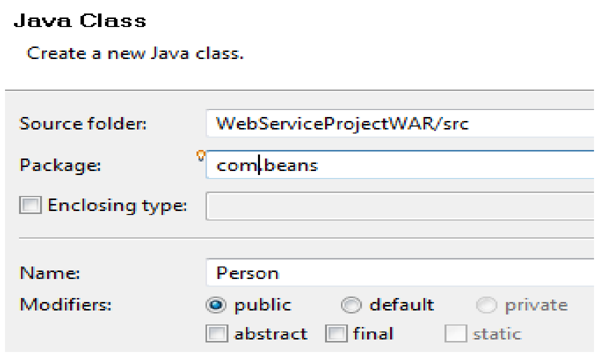
By default, the web service engine uses JAXB as its mapping. This is a part of the JAX-WS spec that we are coding our web service to conform to.

In this lab, we will take a look at how this all works together.

Part 1 - Update the Implementation

We will change our implementation class to use a Java bean as a method return parameter. First of all, let us create the Java bean.

__1. In the **WebServiceProjectWAR**, create a new Java class called **Person** belonging to package **com.beans**



Java Class
Create a new Java class.

Source folder:	WebServiceProjectWAR/src
Package:	com.beans
☐ Enclosing type:	
Name:	Person
Modifiers:	● public ○ default ○ private
	☐ abstract ☐ final ☐ static

An editor will open on the class.

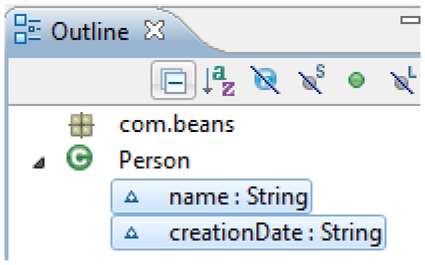
__2. Add the following two fields to the class.

```
String name;
String creationDate;
```

__3. Save the class.

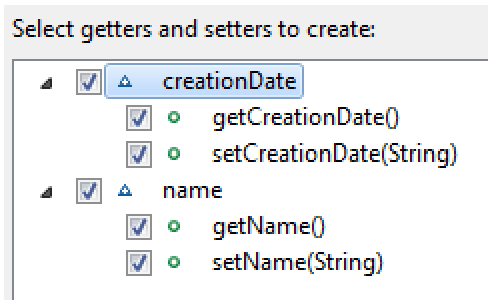
We now need to create accessors (get and set methods) for these fields.

__4. In the *Outline* view (to the right), select both fields.



__5. Right click on these selected fields and select **Source | Generate Getters and Setters...**

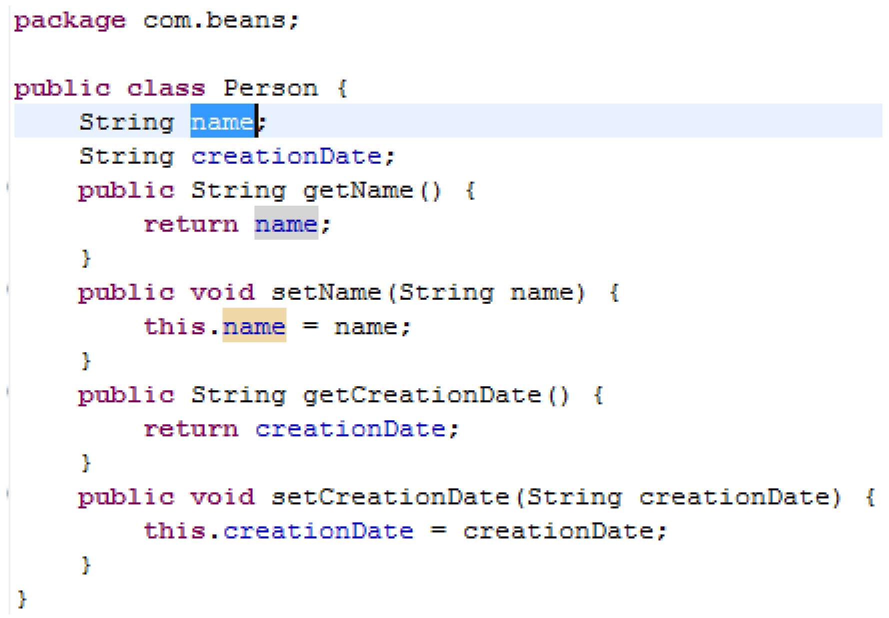
The *Generate Getters and Setters* window will appear.


Select getters and setters to create:

- [✓] creationDate
  - [✓] getCreationDate()
  - [✓] setCreationDate(String)
- [✓] name
  - [✓] getName()
  - [✓] setName(String)

__6. Click the **Generate** button.

The source code will be updated and should look like the following:



```
package com.beans;

public class Person {
    String name;
    String creationDate;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCreationDate() {
        return creationDate;
    }
    public void setCreationDate(String creationDate) {
        this.creationDate = creationDate;
    }
}
```

__7. Save the file.

__8. Close it.

__9. We have greeted our Java bean. Now, let us create an operation that uses it. Open **Greeter.java** (in the **com.simple** package, in the **WebServiceProjectWAR**)

__10. Add the following method to the class:

```
@WebMethod
public Person createPerson(String name) {
          Person p = new Person();
          p.setName(name);
          String time = "" + new java.util.Date();
          p.setCreationDate(time);
          return p;
}
```

Nothing special here; we pass a name to the method; the method then figures out the current time and creates a new Person object with fields being the passed in parameter and the current time.

__11. Now, add the following method:

```
@WebMethod
public String getPersonName(Person person) {
          return person.getName();
}
```

__12. Organize imports. When prompted, select **com.beans.Person** and click **Finish**.

__13. Save changes. There should be no errors.

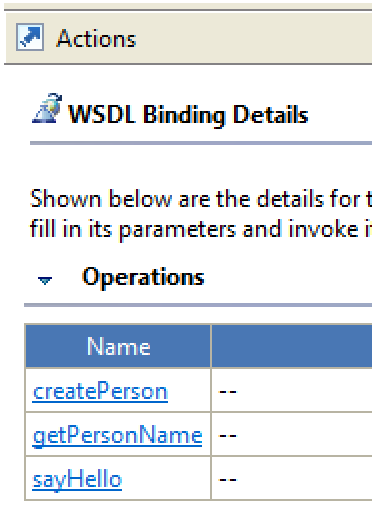
These are both fairly simple (and silly) methods that don't do very much. What is important to note is that in one method, we return a complex Java object. In the other, we pass in a complex Java object. We will see that JAX-WS (the web service implementation in WebSphere) will be able to handle these cases.

Part 2 - Test the Web service

__1. Publish the server. Wait until finish synchronizing.

__2. Start the WSE on the **GreeterService**. To do this, expand **Services** in the **WebServiceProjectWAR**, right-click on **{www.example.org}GreeterService** and select **Test with Web Services Explorer**.

The WSE will open. The three operations of our service will be listed, including the two we just added.



Shown below are the details for t
fill in its parameters and invoke i

Operations

Name	
createPerson	--
getPersonName	--
sayHello	--

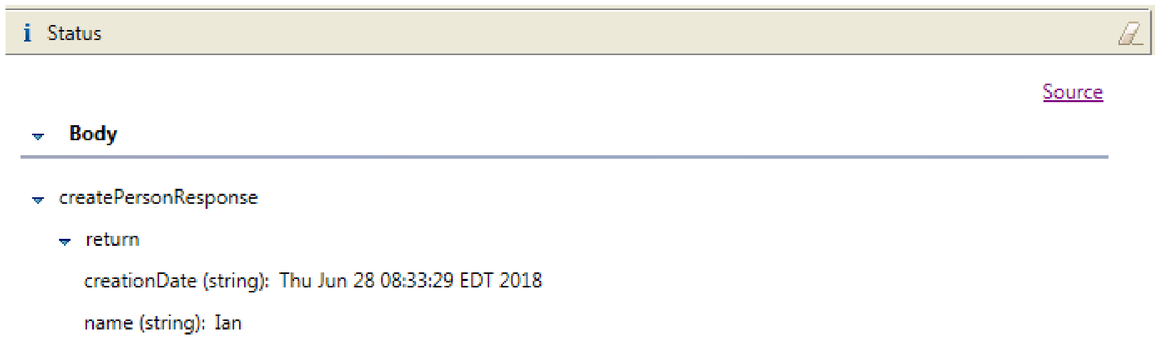
Let us try the **createPerson** operation first. Remember that this method takes a single **String** parameter – the name of the person to be created. It should return a Person object. Let us see how the WSE handles this return.

__3. Click the **createPerson** link.

__4. Click the **Add** link and then enter a name.

__5. Click **Go**.

Look what is returned.



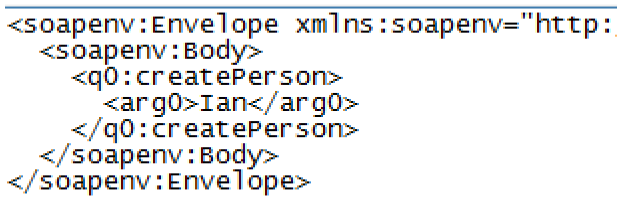
```
createPersonResponse
  return
    creationDate (string): Thu Jun 28 08:33:29 EDT 2018
    name (string): Ian
```

The WSE displays the two fields of the returned Person as just two pieces of data in two different group. Let us take a look at the source.

__6. Click the **Source** link and then maximize the pane.

__7. Right click the **SOAP Request Envelope** and select **View source**.

Examine the *SOAP Response Envelope*.



```
<soapenv:Envelope xmlns:soapenv="http:
  <soapenv:Body>
    <q0:createPerson>
      <arg0>Ian</arg0>
    </q0:createPerson>
  </soapenv:Body>
</soapenv:Envelope>
```

Remember that an XML-field instance of a Person object is what should be returned. Looks like this is more-or-less what we are getting back.

The main element being returned is the **<return>** element, but underneath that we see sub-elements called **<creationDate>** and **<name>,** which were the names of the fields. This is JAXB doing its thing. Aside from the fact that the main element is called **<return>,** the XML marshaling seems to be working well.

So how do we get the JAXB mappings to use a better name than **<return>**? Simple. We can annotate the bean class! We will do this in a moment.

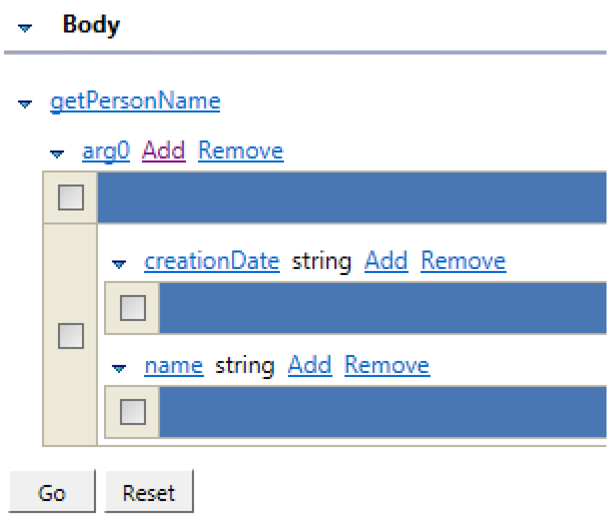
__8. Minimize the *Status* pane.

__9. Click the **Form** link the *Status* pane to show the original return message.

__10. Execute the **getPersonName** method. (This can be done by clicking the **getPersonName** in the left pane but if this doesn't work just run again the Service).

Recall that this method expects a Person object as an input.

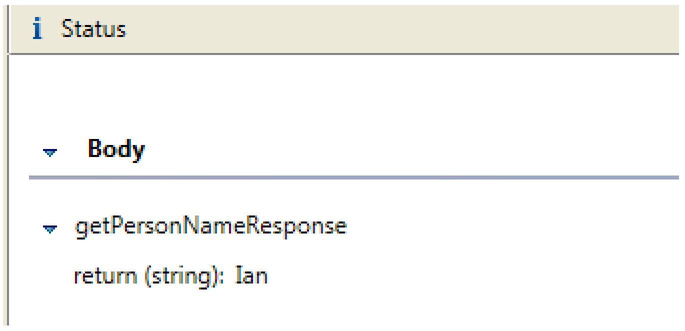
__11. Click the **Add** button, next to **arg0**. What happens?



The WSE automatically shows fields for a **creationDate** and a **name**. These are the two fields that form a Person object.

__12. Click **Add** for both these fields.

__13. Enter some sample data for both fields (the date can be any String) and click **Go**.



```
return (string): Ian
```

__14. The expected response is returned. Maximize the **Status** pane.

__15. Click the **Source** link and then maximize the pane.

__16. *Right click the **SOAP Request Envelope** and select **View source**.*

__17. Look at the *SOAP Request Envelope.*

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:q0="www.example.org" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:getPersonName>
      <arg0>
        <creationDate>2018-06-28</creationDate>
        <name>Ian</name>
      </arg0>
    </q0:getPersonName>
  </soapenv:Body>
</soapenv:Envelope>
```

We see that the argument name is the not-very-meaningful **arg0**. We know how to fix this – by annotating the method with the **@WebParam** annotation.

However, notice that the two child elements are **creationDate** and **name** – again, the fields of the Java **Person** class. Again, this is JAXB doing work under the covers for us.

We have not had to write any JAXB code – but we can be confident in knowing that JAXB is being used to good effect.

__18. Click the **Form** link the *Status* pane to show the original return message.

__19. Minimize the *Status* pane to its normal size in the WSE.

__20. Close the WSE.

Note
JAXB is being used here because it is the default binding for the JAX-WS web service standard. Recall that we are writing JAX-WS code, and WebSphere is complying by using JAXB

Part 3 - Change the JAXB Mappings

What if we do not like the fact that the field names **creationDate** and **name** are being used as element names in the JAXB mappings? What we wanted is for them to be something else (like **creation_date** and **person_name**). How would we change the service to make this happen?

Well, we know that JAXB will automatically use the field names for the element names, so the obvious answer would be to go and change the source code. In the **Person** class, we would rename the fields (from **creationDate** to **creation_date** and **name** to **person_name**), and then re-write the accessor methods.

This, however, is clunky – and could result in broken code if other Java classes depend on the old field names.

A better solution would be to tell JAXB to use different names at generation time. Somehow, we would inform JAX that the names of the XML elements in the generated binding classes should be mapped differently instead of using the default field names.

We can do this by *annotating the Java bean class***.** By annotating the **Person** class, we can control how JAXB operates.

We will now annotate the Person class.

__1. Open **Person.java**

__2. Add the following annotation in bold to the class.

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Person {
```

That means that JAXB will directly access the member variables (without going through the getter and setter methods).

__3. Now, annotate the two fields as follows:

```
@XmlElement(namespace="", name="person_name")
String name;
@XmlElement(namespace="", name="creation_date")
String creationDate;
```

Here, we specify what the names of the field mapped elements should be.

__4. A few errors will appear. Organize imports (**Ctrl-Shift-O**). When prompted, select **javax.xml.bind.annotation.XmlElement** and click **Finish**.

__5. Save the class. There should be no errors.

__6. Close **Person.java**

We have updated the JAXB mappings!

Part 4 - Fix the Return Element

One more thing to do; fix the name of the **<return>** element for the **createPerson** method.

__1. Open **Greeter.java**

__2. Locate the **createPerson** method, and add the following bolded annotation :

```
@WebMethod
@WebResult(name="newperson")
public Person createPerson(String name) {
```

Here, we specify that the element returned should be called **<newperson>**

__3. Organize imports.

__4. Save changes. There should be no errors.

Part 5 - Test JAXB Binding Changes

Let us now see the effects of our annotation.

__1. Publish the server.

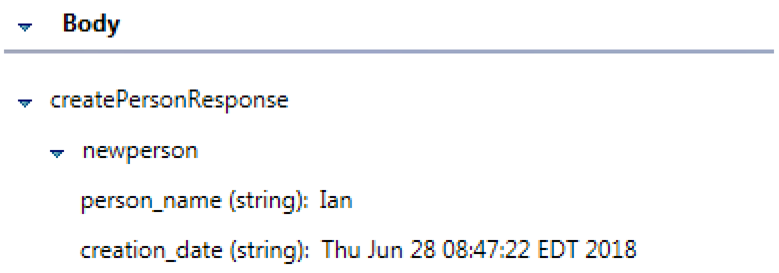
__2. Open the WSE on the service, as you did before.

__3. Run the **createPerson** operation again.

__4. Click **Add**.

__5. Enter a name and click **Go**.

Examine the response.



```
createPersonResponse
  newperson
    person_name (string): Ian
    creation_date (string): Thu Jun 28 08:47:22 EDT 2018
```

Note that the names **person_name** and **creation_date** are being used, as expected.

__6. Maximize the **Status** window.

__7. Click **Source**.

__8. Right click the **Soap Response Envelope** and click **View source**.

__9. Examine the source for the response.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns2:createPersonResponse xmlns:ns2="www.example.org">
      <newperson>
        <person_name>Ian</person_name>
        <creation_date>Thu Jun 28 08:47:22 EDT 2018</creation_date>
      </newperson>
    </ns2:createPersonResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the element being returned is now a **<newperson>** instead of the generic <**return**> that was coming in earlier. This is a result of the **@WebResult** annotation we added to the bean implementation class.

Also note that as we observed earlier, the field names are now **creation_date** and **person_name**. We specified this by annotating the bean class, and the JAXB mapping took note of these annotations.

Congratulations! You have successfully examined JAXB usage.

__10. Close all open editor windows.

Part 6 - Review

In this lab, you examined the use of JAXB by JAX-WS.

We saw that an implementation class can make use of complex Java bean objects as input and return parameter types; by coding the Java beans first, we saw that the web service generation process used JAXB as the marshaling/unmarshaling layer.

In an earlier lab, we saw that a JAXB binding class uses the field names of the bean as element names of the XML by default, and we saw this held up in our web service.

We then examined how we could annotate the bean class to control how JAXB generated its bindings.

Lab 11 - Develop EJB Based Service

There are two ways of developing a web service in JAX-WS. Thus far, we have been taking the approach of annotating a POJO Java implementation class in a web module.

Alternatively, you can use a *stateless session Enterprise Java Bean* (EJB) as your implementation, exposing its logic as web service. An EJB has some advantages over using a POJO implementation class; the main advantage is transaction management. An EJB container (hosting the stateless session bean) automatically manages a global transaction for the EJB methods.

In this lab, we will develop a EJB-based web service. We will develop a simple stateless session EJB. We will then expose the session EJB as a web service, and then create a servlet client that uses the service.

The EJB will be a simple stateless session bean, with a single method called **orderSoup**. This method will take as argument a single String.

Note: EJB coding is an entire topic in itself, and a thorough discussion of this is beyond the scope of this course. This lab will take you through the basics of creating a stateless session EJB and will assume you are already familiar with the technical details. If you would like to know more about EJB programming, please consult your instructor.

Part 1 - Create the Projects

We will start things off by creating the projects. We will have 3 to deal with:

EJBService – this will contain the EJB code, which will be wrapped as a service

EJBServiceClient – this will contain the EJB client stubs

EJBServiceEAR – the EAR file that will contain the above 2 projects.

Let us get started!

__1. From the menu, select **File | New | EJBProject**.

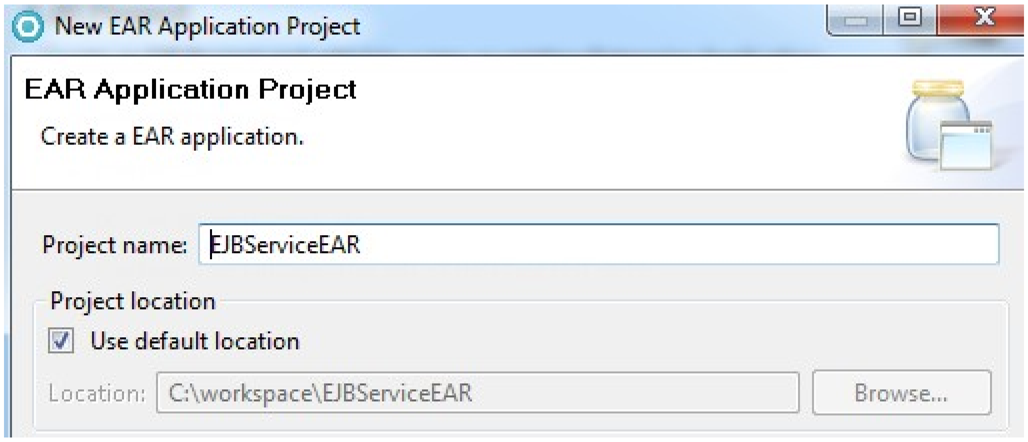
The *New EJB Project* window will appear.

__2. Set the *Project name* to **EJBService**

__3. In the *EAR Membership* area of the window, click **New Project...**

The *New EAR Application Project* window will appear.

__4. Set the *Project name* to **EJBServiceEAR**



EAR Application Project

Create a EAR application.

Project name: EJBServiceEAR

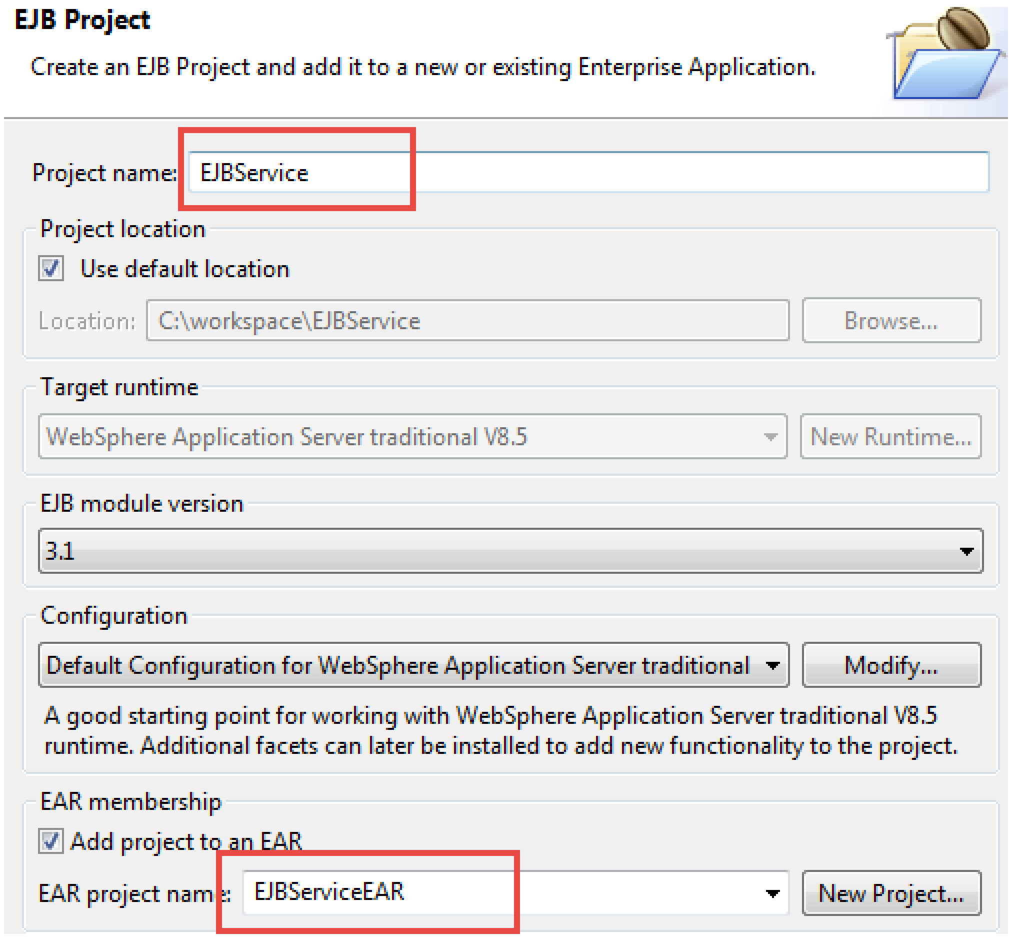
Project location

☑ Use default location

Location: C:\workspace\EJBServiceEAR

__5. Click **Finish**.

You will be returned to the *New EJB Project* screen.



EJB Project

Create an EJB Project and add it to a new or existing Enterprise Application.

Project name: EJBService

Project location

☑ Use default location

Location: C:\workspace\EJBService

Target runtime: WebSphere Application Server traditional V8.5

EJB module version: 3.1

Configuration: Default Configuration for WebSphere Application Server traditional

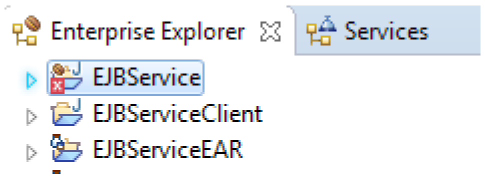
A good starting point for working with WebSphere Application Server traditional V8.5 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☑ Add project to an EAR

EAR project name: EJBServiceEAR

__6. Check that the new EAR project is listed and click the **Finish** button. Three projects will be created: **EJBService**, **EJBServiceClient** and **EJBServiceEAR**. Make sure you see them in the *Enterprise Explorer*.



Our projects have been created! We can start coding the session bean.

You might see an error on EJBService project, because it did not have any EJB component. This error will go away one we define a component later.

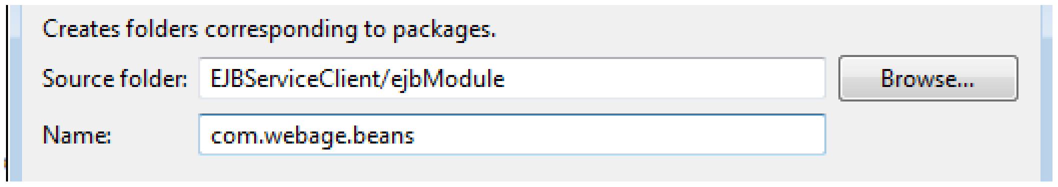
__7. Close all files.

Part 2 - Create the EJB

Our next step is to code the actual EJB. We will start by creating a JavaBean that will be used in the service input and then create the EJB.

__1. In the *Enterprise Explorer*, right click on the **EJBServiceClient** project and select **New → Package**.

__2. Enter **com.webage.beans** for the new package name and click the **Finish** button.



Creates folders corresponding to packages.

Source folder: EJBServiceClient/ejbModule

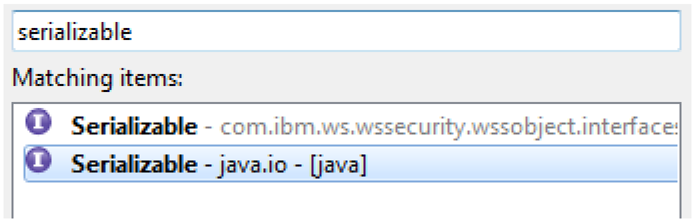
Name: com.webage.beans

__3. Right click the new **com.webage.beans** package and select **New → Class**.

__4. Enter a class name of **Soup** but DON'T click the Finish button yet.

__5. Click the **Add...** button next to the empty list of interfaces.

__6. In the dialog that appears enter 'serializable' in the box and then pick the **java.io.Serializable** interface from the list and click the **OK** button.

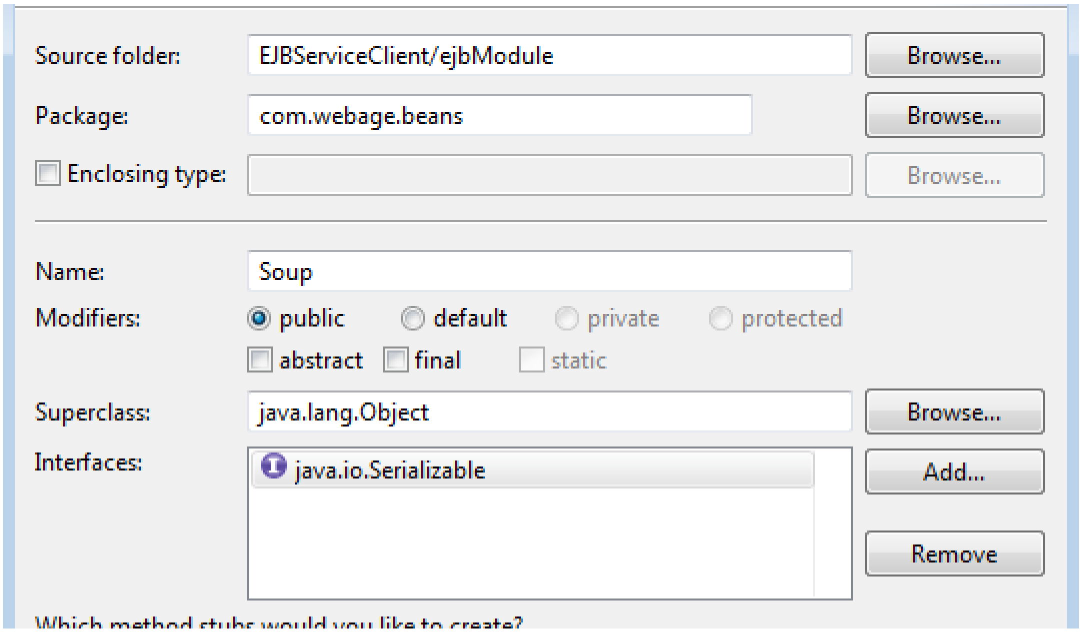


serializable

Matching items:

- Serializable - com.ibm.ws.wssecurity.wssobject.interface
- Serializable - java.io - [java]

__7. Check that your dialog appears as below, including the 'Serializable' interface and click the **Finish** button.



Source folder: EJBServiceClient/ejbModule

Package: com.webage.beans

[ ] Enclosing type:

Name: Soup

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

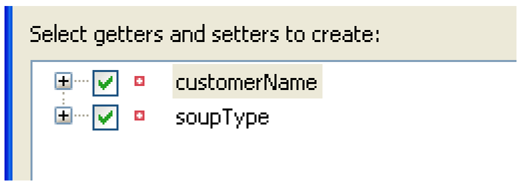
Interfaces: java.io.Serializable

__8. In the editor that opens add the following two fields.

```
public class Soup {
          private String customerName = "";
          private String soupType = "";
```

__9. From the menu, select **Source → Generate Getters and Setters**.

__10. In the dialog click the **Select All** button and click the **Generate** button to generate the code.



Select getters and setters to create:

- [✓] customerName
- [✓] soupType

__11. Save and close the Soup class.

__12. In the *Enterprise Explorer*, right click on the **EJBService** project and select **New | Session Bean**.

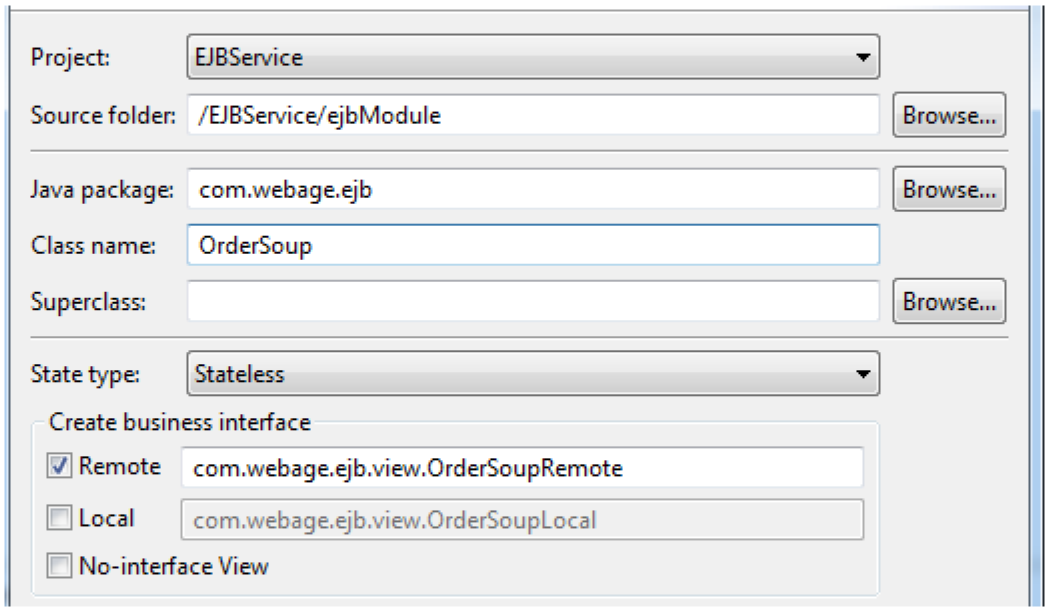
The *Create EJB 3.x Session Bean* window will appear.

__13. Set the *Java package* to **com.webage.ejb**

__14. Set the *Class name* to **OrderSoup**

__15. In the section *Create Business Interface*, uncheck **No-interface View** and check **Remote**.

__16. Check that your dialog appears like that below and click **Finish**.



Project: EJBService

Source folder: /EJBService/ejbModule

Java package: com.webage.ejb

Class name: OrderSoup

Superclass:

State type: Stateless

Create business interface

- [✓] Remote: com.webage.ejb.view.OrderSoupRemote
- [ ] Local: com.webage.ejb.view.OrderSoupLocal
- [ ] No-interface View

The bean (and remote interface) will be created.

The source for **OrderSoup.java** will be open.

This is the session bean class where we will add our business logic and is what will eventually be wrapped as a web service. (This is analogous to the POJO implementation class we coded earlier)

__17. Add the following method to the class:

```
public String orderSoup(Soup soup) {
          String message = soup.getSoupType() + " served to "
                    + soup.getCustomerName();

          System.out.println("EJB said: " + message);

          return message;
}
```

__18. Organize imports (Source → Organize Imports).

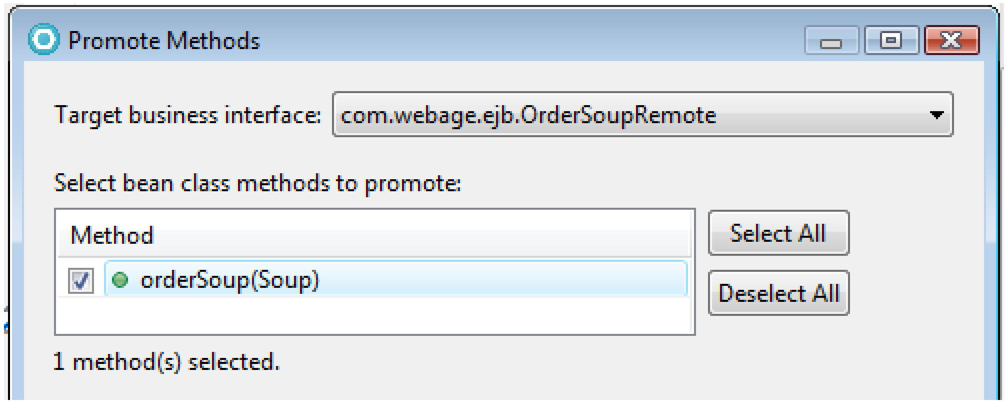
__19. Save the class. There should be no errors.

__20. We now need to promote the **orderSoup** method to the remote interface. In the *Enterprise Explorer*, expand **EJBService | ejbModule | com.webage.ejb**.

__21. Right-click on **OrderSoup.java** and select **Java EE Tools | Promote Methods...**

The *Promote Methods* window will appear.

__22. Check **orderSoup(Soup)** and click **OK**.



Target business interface: com.webage.ejb.OrderSoupRemote

Select bean class methods to promote:

Method
✓ orderSoup(Soup)

1 method(s) selected.

We have completed coding our session bean!

Part 3 - Create the Web Service

We have created our implementation, in the form of a session bean. If we were using a POJO implementation class, we would simply annotate the class. So how do we turn a session bean into a web service?

Doing so is a two stage process. The first stage is the same as with the POJO implementation; we will need to annotate our bean class.

The second stage involves the creation of a *router module*. Let us start off with annotating the EJB class.

__1. Open **OrderSoup.java** from the **com.webage.ejb** package of the **EJBService** project in the editor.

__2. Add the **@WebService** annotation to the class, immediately after the **@Stateless** annotation.

```
@Stateless
@WebService
@Remote(OrderSoupRemote.class)
public class OrderSoup implements OrderSoupRemote {
```

__3. Organize imports and save the code. There should be no errors although there will be a warning about the lack of a router module.

__4. Close the file.

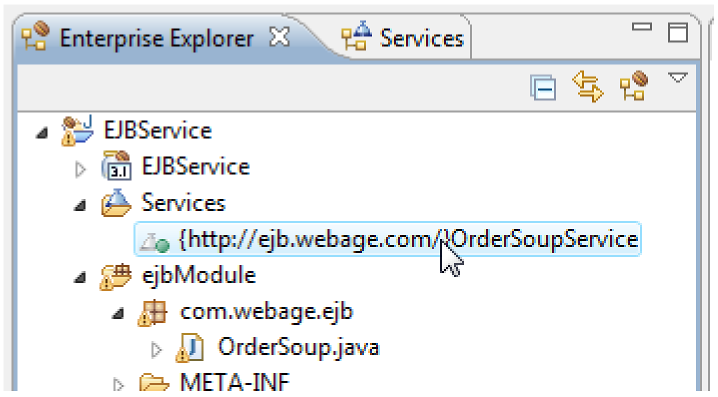
Now, we can create the *router module*.

The router module is essentially a servlet that listens for incoming requests to our service; when it receives a SOAP request for the service, it automatically invokes the session bean.

From a client perspective, the router module *is* the service, when in reality it is acting as a delegate to the session bean implementation.

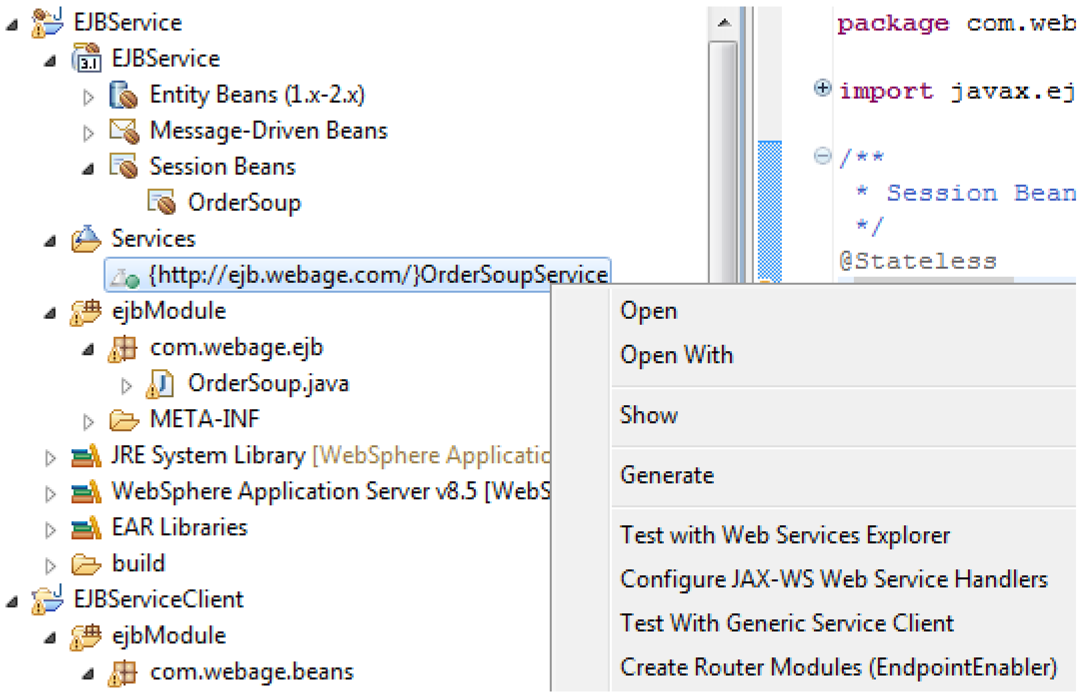
Creating a router module is very simple. We will do so now.

__5. In the *Enterprise Explorer*, expand **EJBService | Services**.



Notice that the Eclipse has detected our @WebService annotation in the bean class and is recognizing it as a service.

__6. Right-click on **{http://ejb.webage.com/}OrderSoupService** and select **Create Router Modules (Endpoint Enabler)**

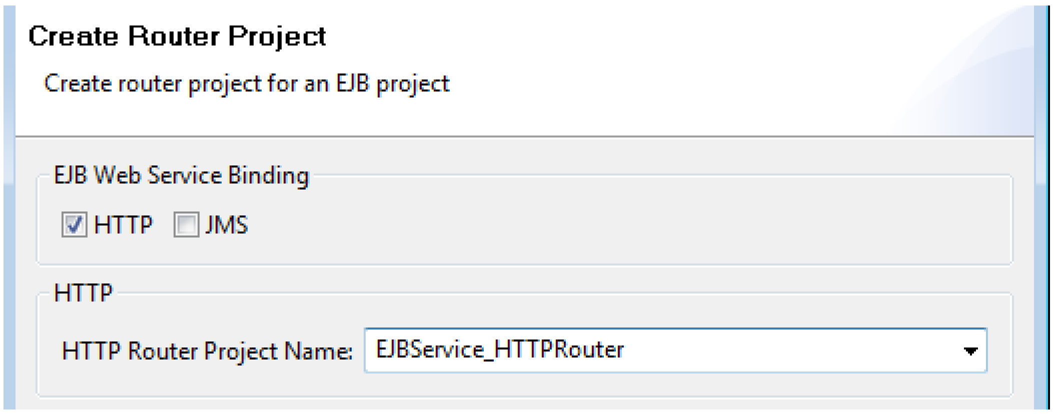

EJBService
EJBService
Entity Beans (1.x-2.x)
Message-Driven Beans
Session Beans
OrderSoup
Services
{http://ejb.webage.com/}OrderSoupService
ejbModule
com.webage.ejb
OrderSoup.java
META-INF
JRE System Library [WebSphere Applicatic
WebSphere Application Server v8.5 [WebS
EAR Libraries
build
EJBServiceClient
ejbModule
com.webage.beans
Open
Open With
Show
Generate
Test with Web Services Explorer
Configure JAX-WS Web Service Handlers
Test With Generic Service Client
Create Router Modules (EndpointEnabler)

```
package com.web

import javax.ej

/**
 * Session Bean
 */
@Stateless
```

The *Create Router Modules (EndpointEnabler)* screen will appear. This wizard will create the router module for us.

__7. Make sure **HTTP** is selected and click **Finish**.

Note that it will create a new project called **EJBService_HTTPRouter**. This is where the router module will be created. (The router module will be placed in the same EAR file as the EJB project this is being generated from)



Create Router Project

Create router project for an EJB project

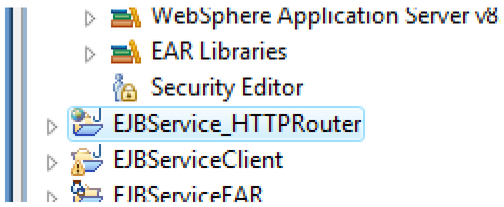
EJB Web Service Binding

[✓ HTTP] JMS

HTTP

HTTP Router Project Name: EJBService_HTTPRouter

Note: The Create Router Modules wizard gives you the option to create a router module for the HTTP transport, the JMS transport, or both transports. A JMS router module would only be necessary if you needed the web service to be able to receive requests from a JMS queue or topic. Eclipse will create the router project. It will be visible in the *Enterprise Explorer*.



__8. Expand **EJBService_HTTPRouter** and open the project's web deployment descriptor (in **WebContent | WEB-INF | web.xml**)

__9. Click on the **Source** tab. Notice the following:

- A single servlet is registered which has the same name as our bean class.
- The servlet corresponds with the **com.ibm.ws.websvcs.transport.http.WASAxis2Servlet** class. This is an IBM provided servlet used to route requests to the web service.
- The mapping for the servlet is **/OrderSoupService**

__10. Close all open files.

Part 4 - Test the Web Service

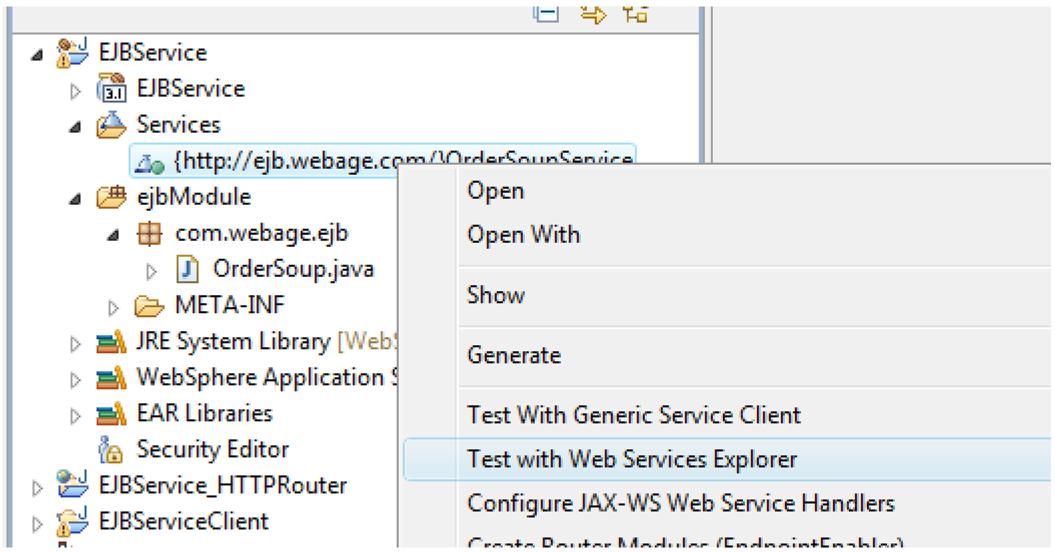
We can now test our web service wrapped session EJB. We will do so using the WSE.

__1. Start the server if it's not running.

__2. Add **EJBServiceEAR** to the server.

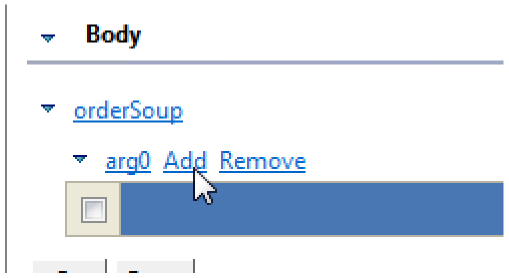
__3. Publish the server.

__4. Right-click on **{http://ejb.webage.com/}OrderSoupService** and select **Test with Web Services Explorer**.

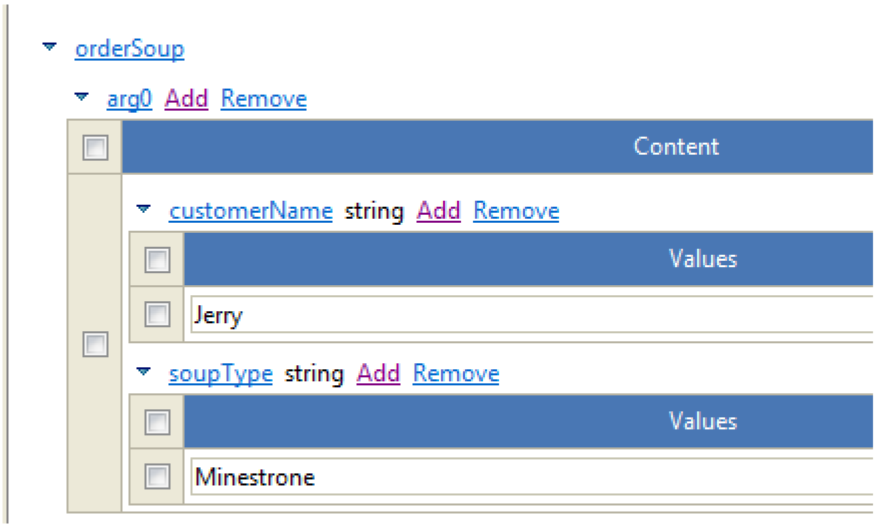


__5. Click the link for the '**orderSoup**' operation on the right.

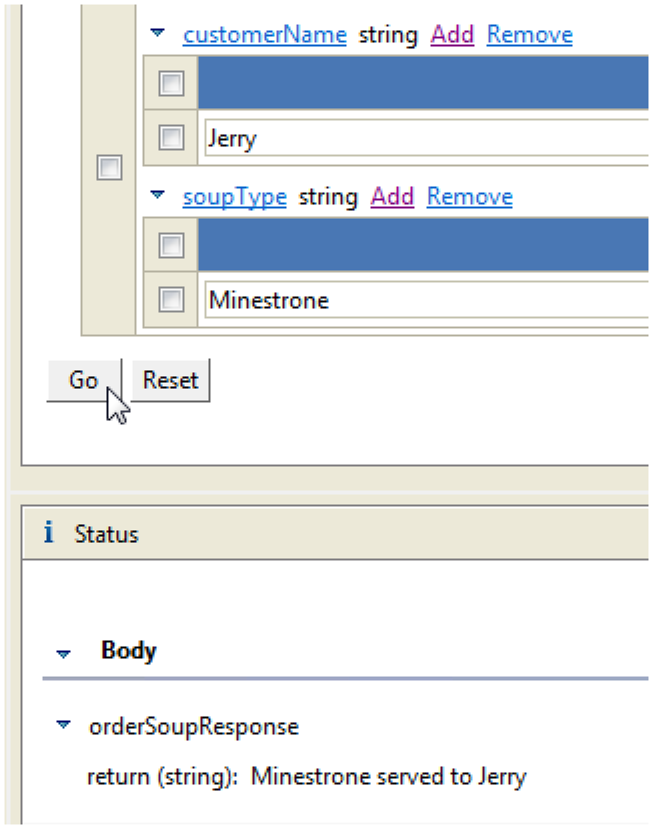
__6. Click the **Add** link to add a parameter to the request body.



__7. Click the **Add** links again to be able to enter values for the parameters.



__8. Click the **Go** button and verify you get the expected response.



orderSoupResponse

return (string): Minestrone served to Jerry

Our web service works! We have successfully implemented a web service as a session bean.

__9. Close the Web Services Explorer.

Part 5 - Create A Web Client

We will now create a proper Java client for our web service. We will create a web application, wherein the user enters data into an HTML form that posts to a servlet; the servlet will then invoke the service via a client stub.

We should create a web project to hold our client.

__1. Create a new **Dynamic Web Project** called **EJBServiceWebClient** belonging to an EAR called **EJBServiceWebClientEAR**. Make sure the web project is NOT added to an existing EAR project but a new EAR project.

Now we should generate the client stub.

__2. Make sure you are in the *Java EE* perspective.

__3. Make sure the server is running.

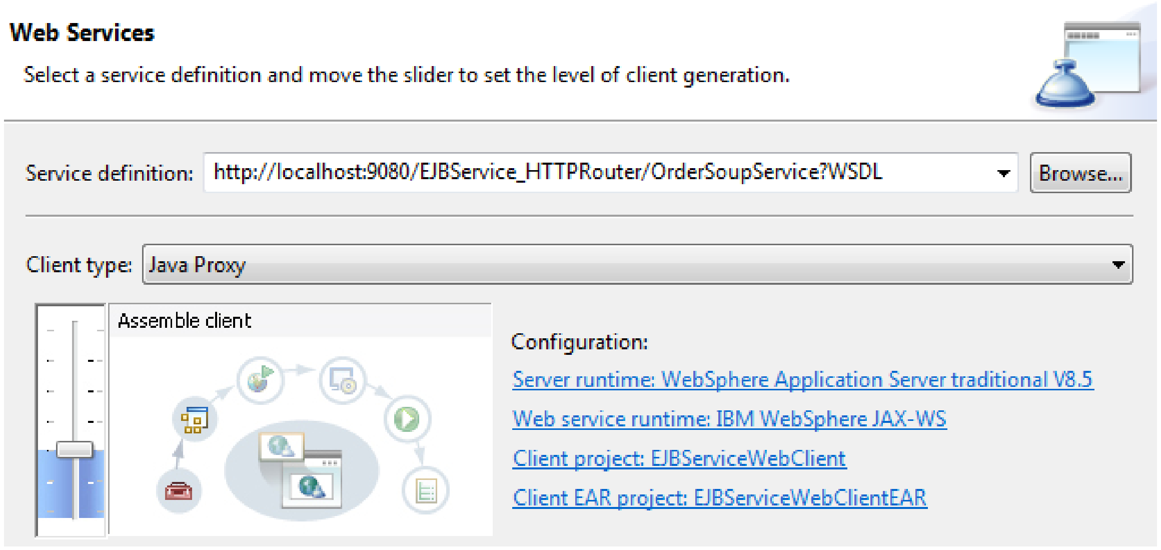
__4. In the *Enterprise Explorer*, expand **EJBService | Services**.

__5. Right-click on **{http://ejb.webage.com/}OrderSoupService** and select **Generate → Client**.

The *Web Service Client* screen will appear.

__6. Drag the slider down to **Assemble client**.

__7. Change the *Client project* to **EJBServiceWebClient**.


Web Services

Select a service definition and move the slider to set the level of client generation.

Service definition: http://localhost:9080/EJBService_HTTPRouter/OrderSoupService?WSDL

Client type: Java Proxy

Assemble client

Configuration:

Server runtime: WebSphere Application Server traditional V8.5

Web service runtime: IBM WebSphere JAX-WS

Client project: EJBServiceWebClient

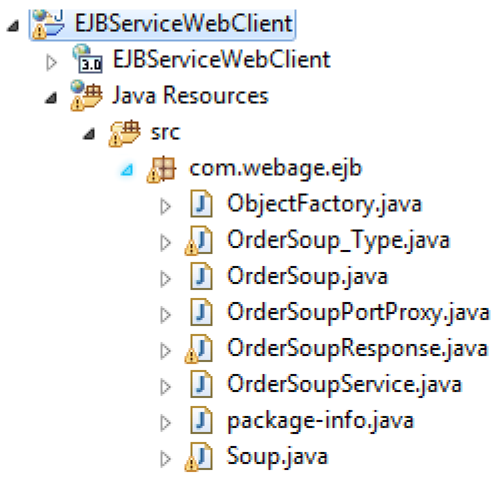
Client EAR project: EJBServiceWebClientEAR

__8. Click **Finish** once your dialog appears as shown above.

The client will be generated in our **EJBServiceWebClient** project.

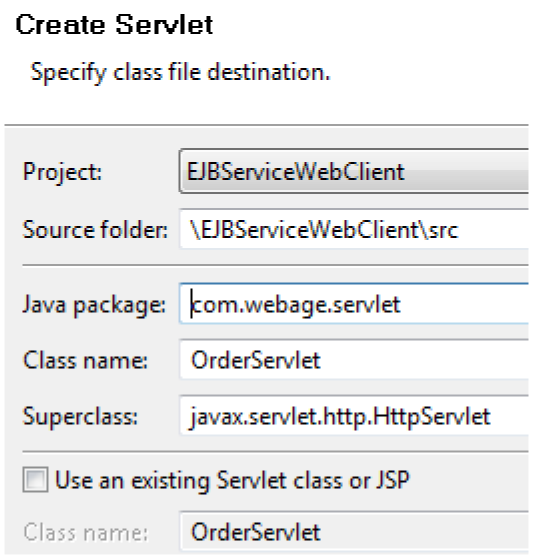
__9. Click **OK** to dismiss the message.

__10. Refresh and then expand **EJBServiceWebClient**. Make sure you see the client stub!



```
EJBServiceWebClient
  EJBServiceWebClient
  Java Resources
    src
      com.webage.ejb
        ObjectFactory.java
        OrderSoup_Type.java
        OrderSoup.java
        OrderSoupPortProxy.java
        OrderSoupResponse.java
        OrderSoupService.java
        package-info.java
        Soup.java
```

__11. Now, in the **EJBServiceWebClient** project, create a servlet called **OrderServlet** in the **com.webage.servlet** package. You can do this by selecting the project, selecting **File → New → Servlet** from the list, filling in the dialog as shown below and clicking **Finish**.



Create Servlet

Specify class file destination.

Project:	EJBServiceWebClient
Source folder:	\EJBServiceWebClient\src
Java package:	com.webage.servlet
Class name:	OrderServlet
Superclass:	javax.servlet.http.HttpServlet

☐ Use an existing Servlet class or JSP

Class name:	OrderServlet

An editor will open on the servlet.

__12. Set the text of the **doGet(..)** method to the following:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
            String customerName = request.getParameter("customerName");
            String soupType = request.getParameter("soupType");
            Soup soupToOrder = new Soup();
            soupToOrder.setCustomerName(customerName);
            soupToOrder.setSoupType(soupType);

            OrderSoupService os = new OrderSoupService();
            OrderSoup o = os.getOrderSoupPort();

            PrintWriter out = response.getWriter();
            String message = o.orderSoup(soupToOrder);
            out.println(message);
}
```

__13. Organize imports.

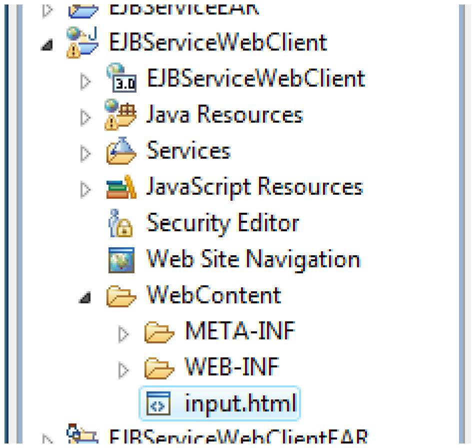
__14. Save changes. There should be no errors.

Our code is done.

__15. Close all open files.

Our last task is to create a nice HTML form that posts to this servlet. Fortunately, we have provided this form for you.

__16. Copy and paste file **C:\LabFiles\input.html** into the **WebContent** folder of **EJBServiceWebClient**



We are done creating our web front end to the web service!

Part 6 - Test

Let us now try our web front end.

__1. Add **EJBServiceWebClientEAR** to the server.

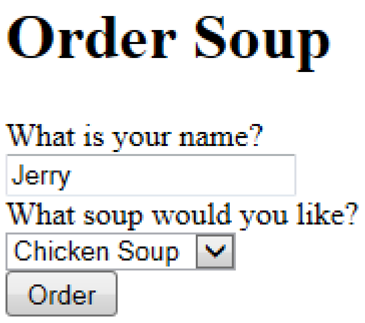
__2. Publish the server.

__3. Right-click on **input.html** (under **EJBServiceWebClient | WebContent**) and select **Run As | Run on Server**.

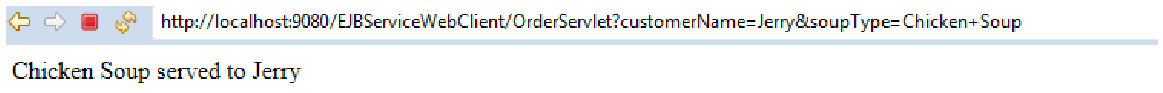
__4. The *Run On Server* screen may appear. Check the box at the bottom marked *Always use this server when running this project* and click **Finish**.

A browser will open with a pre-populated information.

__5. Click the **Order** button.



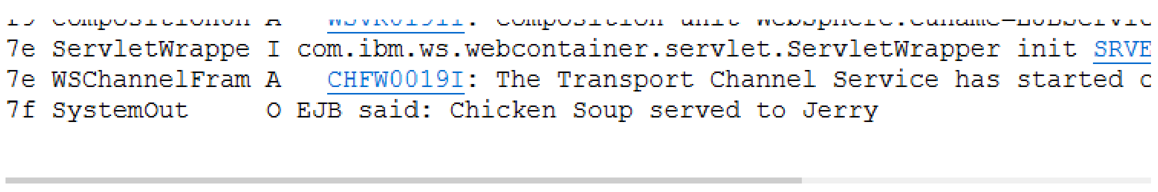
__6. Make sure that the response shows the appropriate response.



```
http://localhost:9080/EJBServiceWebClient/OrderServlet?customerName=Jerry&soupType=Chicken+Soup
```

Chicken Soup served to Jerry

__7. Back in the *Console* view you should see a message that the web service request was indeed processed by the EJB code.



```
7e ServletWrappe I com.ibm.ws.webcontainer.servlet.ServletWrapper init SRVE
7e WSChannelFram A  CHFW0019I: The Transport Channel Service has started c
7f SystemOut     O EJB said: Chicken Soup served to Jerry
```

Our web service is being successfully invoked by the servlet.

__8. Go back and test using other soup options and other names.

__9. Close all open files and web browsers.

__10. Stop the server.

Part 7 - Review

In this lab, you created a session bean implementation for a web service. You saw:

- How to create and test a simple session EJB
- How to annotate a simple session EJB to expose it as a web service
- How to generate a *router module*, which is required by WebSphere to allow the service to be invoked

Lab 12 - Handle Web Service Faults

The goal of this lab is to learn how fault handling works in JAX-WS. We will convert a web service method to throw a Java exception which will be converted into SOAP fault by JAX-WS. The Exception will be modeled as a "fault" in the web service. You will then see how an exception will be part of the client generated code to model the fault on the client side.

Even though this lab will add a fault to a service deployed as an EJB it is important to note that the same behavior would be true for web services that use a POJO JavaBean implementation.

Part 1 - Create the Exception Class

Our service method will potentially throw an exception; specifically, if a particular customer name ("Elaine") is contained in the 'Soup' order parameter, a **BadCustomerException** will be thrown. This **BadCustomerException** is an exception class that we will write.

The question to ask is this: what will happen when someone invokes the service with "Elaine" as a customer name? What will the web service layer do with the **BadCustomerException**? We shall see.

The exception class should be created in the project with EJB service client classes.

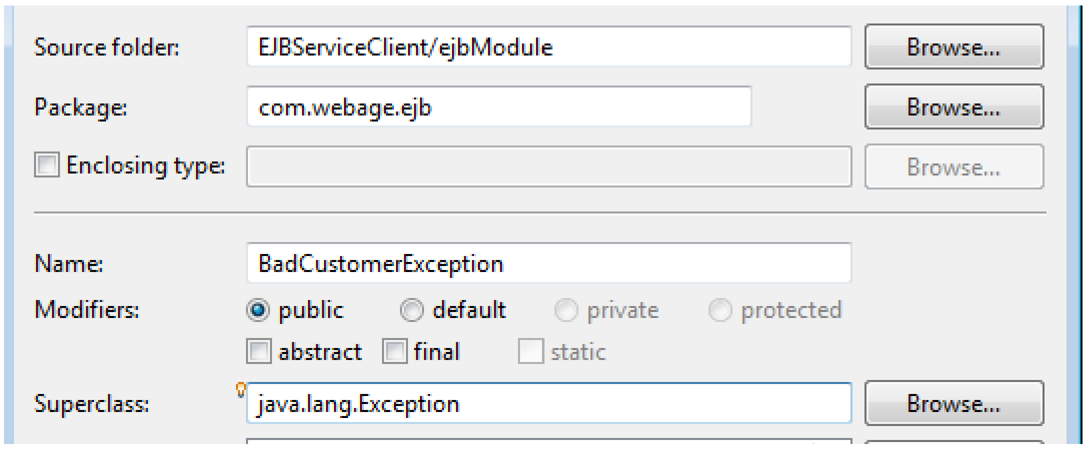
__1. In the *Enterprise Explorer,* right-click on the **EJBServiceClient** project and select **New → Class**.

The *New Java Class* window will appear.

__2. Set the *Package* to be **com.webage.ejb**

__3. Set the *Name* to be **BadCustomerException**

__4. Set the *Superclass* to be **java.lang.Exception**



Source folder: EJBServiceClient/ejbModule

Package: com.webage.ejb

[ ] Enclosing type:

Name: BadCustomerException

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Exception

__5. Click **Finish** when your options match those above.

The class will be created, and an editor will open on it.

```
package com.webage.ejb;

public class BadCustomerException extends Exception {

}
```

__6. Add the following annotations to the class declaration:

```
@WebFault(name="BadCustomer")
@XmlAccessorType(XmlAccessType.FIELD)
public class BadCustomerException extends Exception {
```

Note: We are using the @WebFault annotation to customize the name of the XML element that will be present in the generated WSDL file. Without the 'name' attribute the fault element name would be 'BadCustomerException', the same as the class name. Although this does not sound bad, if the name ended in '..Exception', when the client code were generated it would add '_Exception' to the end of the generated Java exception class name. Having a class name in the client of 'BadCustomerException_Exception' might be confusing so we are customizing the name of the XML element for the fault.

__7. Add the following code to the class:

```
private Soup soupOrdered;

public BadCustomerException(Soup soupOrdered) {
          super("No soup for you!");
          this.soupOrdered = soupOrdered;
}

public Soup getSoupOrdered() {
          return soupOrdered;
}
```

Note: We are also using the @XmlAccessorType annotation because we will add a field to the exception class to send back custom data with the exception. JAX-WS faults follow the same JAXB binding rules and by default the only elements that would show up in the XML are properties that have a get/set method. We only are going to provide a 'get' method since the data contained in the exception shouldn't be modified after the exception is created. The XmlAccessType.FIELD value will turn every field into an element in the XML of the fault.

__8. Organize imports.

__9. Save the file. There should no errors.

__10. Close all open files.

Our exception class is coded.

Part 2 - Modify Service Implementation Using Exception

The next step is to modify the service implementation, in this case an EJB, to throw the Exception under the correct conditions. You will see that the coding in the implementation is just regular Java Exception code and not unique to web services.

__1. Open the **OrderSoupRemote.java** file in the **com.webage.ejb.view** package in the **EJBServiceClient** project.

__2. Modify the method on the interface to throw the exception:

```
public interface OrderSoupRemote {
   String orderSoup(Soup soup) throws BadCustomerException;
}
```

__3. Organize imports.

__4. Save the interface and make sure there are no errors in the interface although you may have errors in other classes.

__5. Close the **OrderSoupRemote.java** file.

__6. Open the **OrderSoup.java** file in the **com.webage.ejb** package in the **EJBService** project.

__7. Modify the method signature to throw the exception class and the method implementation to check if the 'customerName' is "Elaine" and throw the exception if it is.

```
public String orderSoup(Soup soup) throws BadCustomerException {
          if (soup.getCustomerName().equalsIgnoreCase("Elaine")) {
                    throw new BadCustomerException(soup);
          }
...
```

__8. Save and close all open files.

Part 3 - Test Web Service Fault

Before modifying the client to handle the web service fault it will be good to test it with the Web Services Explorer.

__1. Stop the server.

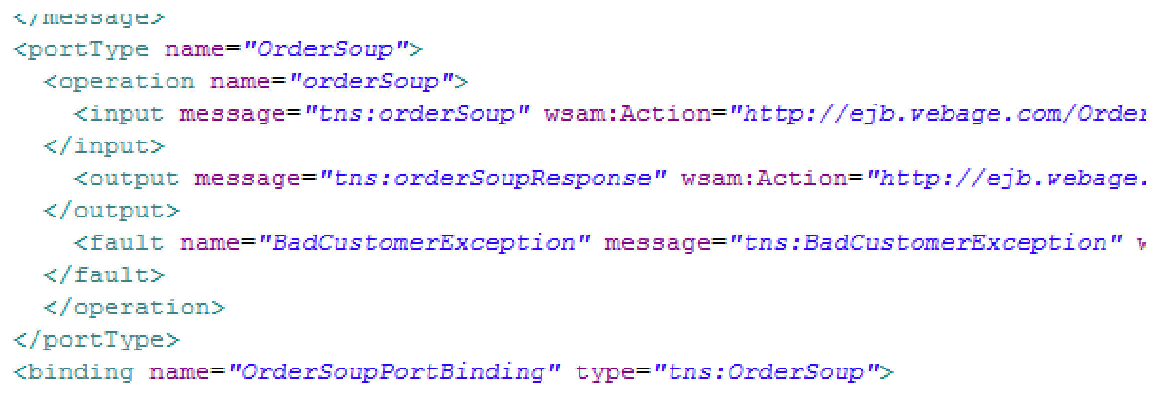
__2. Start the server.

__3. In the *Enterprise Explorer*, expand **EJBService | Services**.

__4. Right-click on **{http://ejb.webage.com/}OrderSoupService** and select **Show → WSDL Interface**.

__5. Click on the **Source** tab of the editor.

__6. There is a **<fault ..>** element in the definition of the '**orderSoup**' operation of the service port.



```
</message>
<portType name="OrderSoup">
  <operation name="orderSoup">
    <input message="tns:orderSoup" wsam:Action="http://ejb.webage.com/Orde
  </input>
    <output message="tns:orderSoupResponse" wsam:Action="http://ejb.webage.
  </output>
    <fault name="BadCustomerException" message="tns:BadCustomerException" w
  </fault>
  </operation>
</portType>
<binding name="OrderSoupPortBinding" type="tns:OrderSoup">
```

__7. Close the WSDL file.

__8. In the *Enterprise Explorer*, expand **EJBService | Services**.

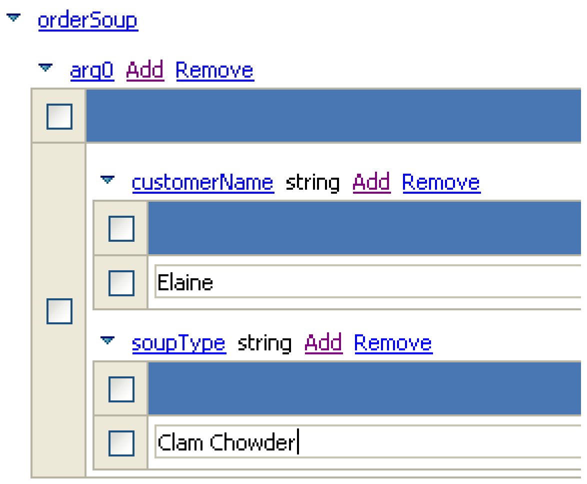
__9. Right-click on **{http://ejb.webage.com/}OrderSoupService** and select **Test with Web Services Explorer**.

__10. Click the **orderSoup** operation.

__11. Click the **Add** link next to the main operation parameter.

__12. Click the **Add** link next to the '**customerName**' and '**soupType**' elements of the form to create empty text boxes for these parameters.

__13. Fill in a '**customerName**' of "Elaine" and a type of soup.

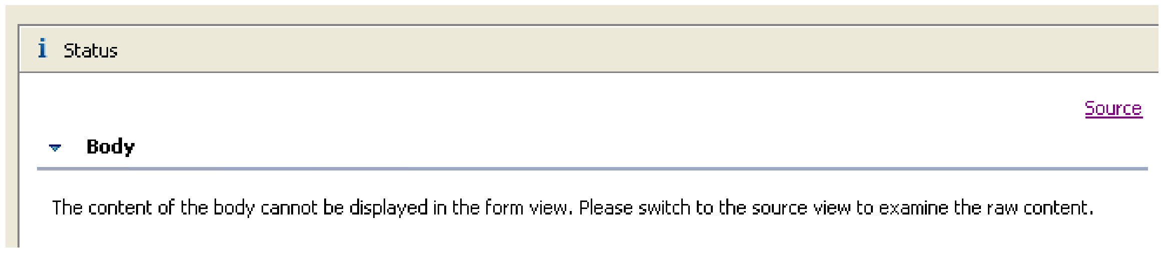


Elaine

Clam Chowder

__14. Click **Go**.

__15. You will get a message that the content can't be displayed in this view.

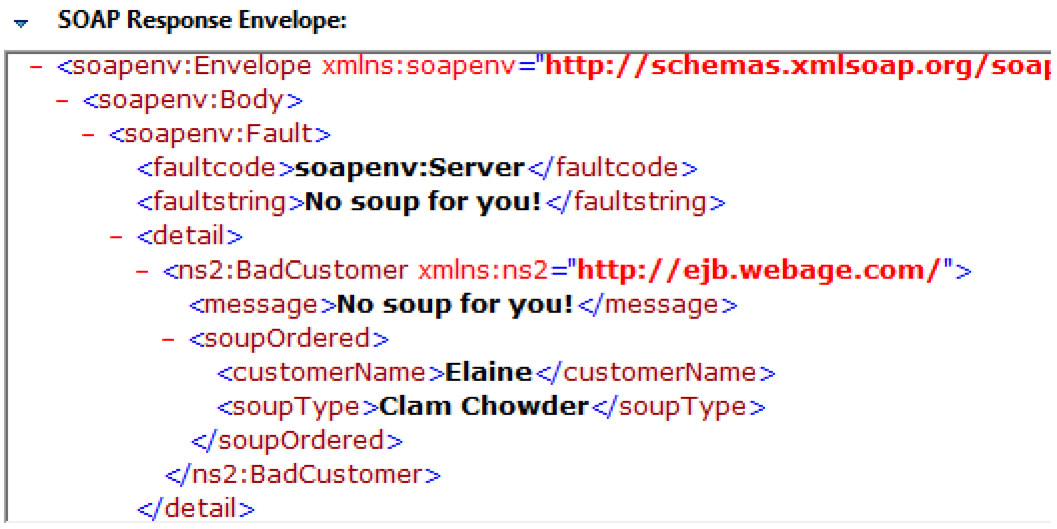


The content of the body cannot be displayed in the form view. Please switch to the source view to examine the raw content.

__16. Click the **Source** link and maximize the *Status* pane.

__17. Right click on **SOAP Response Envelope** and select **View Source**.

__18. Examine the *SOAP Response Envelope*.



```
- <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soa
  - <soapenv:Body>
    - <soapenv:Fault>
        <faultcode>soapenv:Server</faultcode>
        <faultstring>No soup for you!</faultstring>
      - <detail>
        - <ns2:BadCustomer xmlns:ns2="http://ejb.webage.com/">
            <message>No soup for you!</message>
          - <soupOrdered>
              <customerName>Elaine</customerName>
              <soupType>Clam Chowder</soupType>
            </soupOrdered>
          </ns2:BadCustomer>
        </detail>
```

Note that a **<soapenv:Fault>** element has been returned, and that its **faultString** is set to the exception message i.e **No soup for you!**

Also note that the **<detail>** of the fault is basically a SOAP-ified version of the exception (**BadCustomerException**)

What we are seeing here is that a web service will wrap any thrown Java exceptions as a SOAP message for transmission. What we do not see here is the fact that a client invoking this operation through code generated for the client would simply catch a regular Java Exception. Just like the conversion from Java exception to web service fault was done automatically, so will the conversion back when using generated code.

__19. Close the XML file.

__20. Minimize the Status pane.

__21. Close the Web Services Explorer.

__22. Leave the server running.

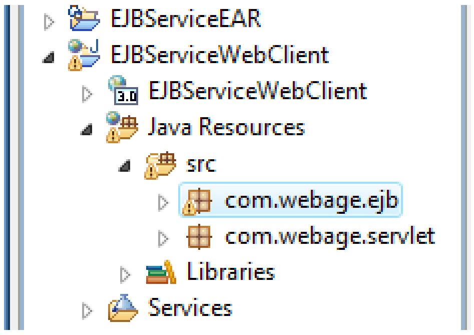
Part 4 - Modify Web Client to Handle Fault

Getting the service to return the fault is only half the story. We must also modify the client so it will be able to expect the fault and handle it properly.

The best way to do this is to regenerate the client stub code for the service so it includes code to handle the fault. This regenerated code will be able to convert the web service fault into a Java exception so the client can simply catch the exception like normal Java code.

__1. Expand the **EJBServiceWebClient → Java Resources → src** folders.

__2. Select the **com.webage.ejb** package.



__3. Right click the selected package and select **Delete** and then confirm you want to delete the package. You will get errors in other code.

Note: The reason we are deleting the packages are because they contain the generated client code but do not reflect the addition of the fault. Deleting the packages first will ensure that all of the generated code will be current with the updated WSDL of the service.

It is suggested to not alter the generated code or place other classes in the same packages. This way you can always delete and regenerate the client code when the WSDL of a service changes. After regenerating the new code you can fix any other compilation errors that remain.

__4. Make sure your server is running.

__5. In the *Enterprise Explorer*, expand **EJBService | Services**.

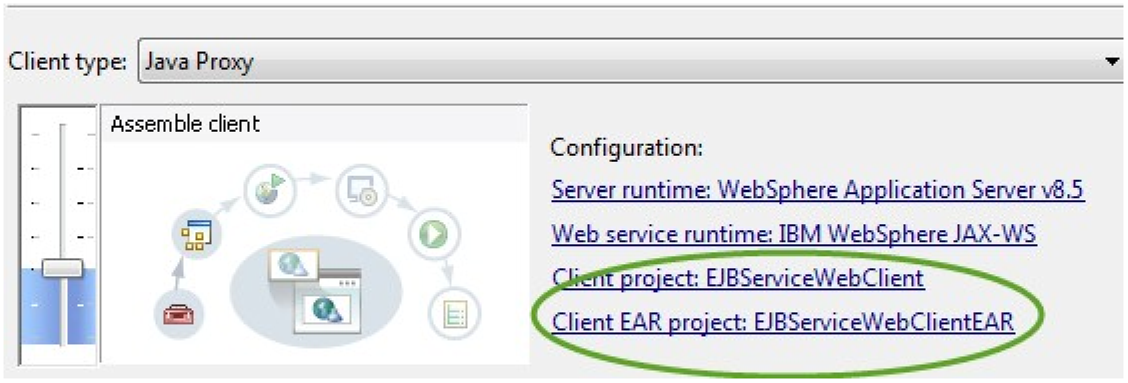
__6. Right-click on **{http://ejb.webage.com/}OrderSoupService** and select **Generate → Client**.

The *Web Service Client* screen will appear.

__7. Drag the slider down to **Assemble client**.

__8. Make sure the *Client project* is **EJBServiceWebClient**.

__9. Click **Finish** once your dialog appears as shown below and then click OK.


Client type: Java Proxy

Assemble client

Configuration:

Server runtime: WebSphere Application Server v8.5

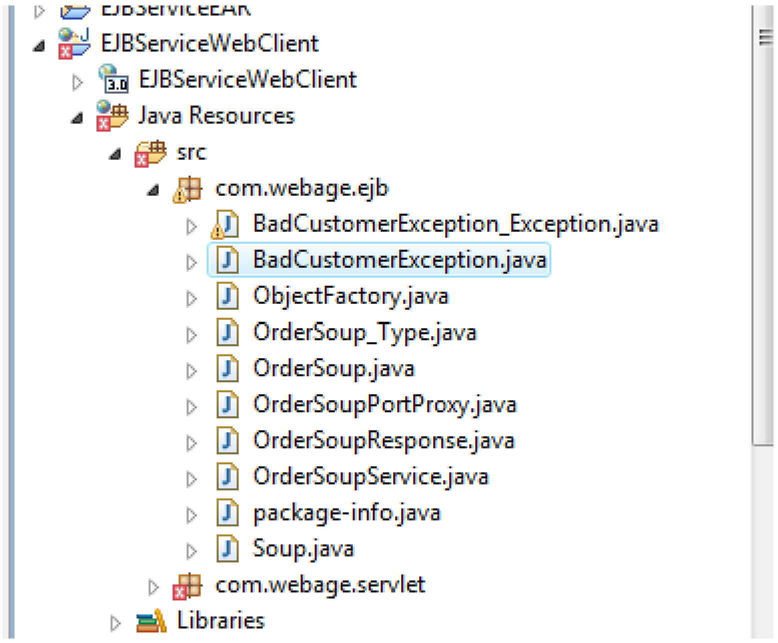
Web service runtime: IBM WebSphere JAX-WS

Client project: EJBServiceWebClient

Client EAR project: EJBServiceWebClientEAR

The client will be generated in our **EJBServiceWebClient** project.

__10. Refresh and then expand **EJBServiceWebClient**. You will still have errors in the servlet. Notice there are some classes generated for the Exception.


```
EJBServiceWebClient
  EJBServiceWebClient
  Java Resources
    src
      com.webage.ejb
        BadCustomerException_Exception.java
        BadCustomerException.java
        ObjectFactory.java
        OrderSoup_Type.java
        OrderSoup.java
        OrderSoupPortProxy.java
        OrderSoupResponse.java
        OrderSoupService.java
        package-info.java
        Soup.java
      com.webage.servlet
    Libraries
```

__11. Open the **OrderServlet.java** file in the **com.webage.servlet** package.

__12. Find the **doGet** method and modify the code at the end of the method to match the following:

```
    PrintWriter out = response.getWriter();

    try {
        String message = o.orderSoup(soupToOrder);
        out.println(message);
    } catch (BadCustomerException_Exception e) {
        out.println("<h2>A bad customer tried to order!</h2>");
        Soup soupOrdered = e.getFaultInfo().getSoupOrdered();
        out.println(soupOrdered.getCustomerName()
            + " is not allowed to order " + soupOrdered.getSoupType());
    }
}
```

Note: This code surrounds the call to '**orderSoup**' on the service with a try/catch block that will handle the exception. Notice it is regular Java exception handling. The only thing that is a little different is the call to the '**getFaultInfo**' method.

__13. Organize imports.

__14. Save the file. There should no longer be any errors in the project.

Part 5 - Test Updated Web Client

Now we can test to make sure the web client can properly handle the situation where a web service fault will be thrown. We will first check to be sure the "normal" case behaves correctly and then test the condition that causes the fault to be thrown.

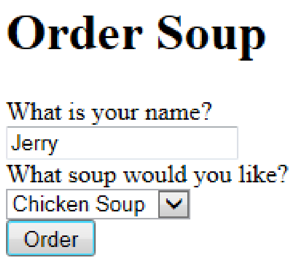
__1. Return to the *Servers* view and publish the server by right clicking the server and selecting **Publish** after making sure the server is running first.

__2. Right-click on **input.html** under **WebContent** in the **EJBServiceWebClient** project and select **Run As → Run On Server**.

__3. If prompted, leave all deployments selected and click **Finish**.

Eclipse will open a browser, displaying our input form.

__4. Leave the default information and click the **Order** button.



Order Soup

What is your name?

Jerry

What soup would you like?

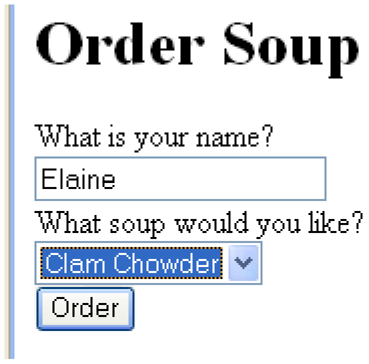
Chicken Soup

__5. Make sure that the response shows the appropriate response.

```
Chicken Soup served to Jerry
```

__6. Click the back arrow button in the upper left of the embedded browser to return to the form.

__7. Fill in data using the name "Elaine" and click the **Order** button.



Order Soup

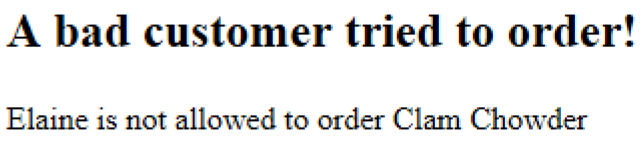
What is your name?

Elaine

What soup would you like?

Clam Chowder

__8. Check that you get the correct output that is triggered by the web service fault being returned from the service back to the client.



A bad customer tried to order!

Elaine is not allowed to order Clam Chowder

__9. Close all open files and web browsers.

Part 6 - Review

In this lab you saw how exception handling from a web service is done. You saw that an exception thrown by a web service is marshalled into a SOAP fault for transport, and then un-marshalled back to a Java exception class on the client side.

Lab 13 - Managed Service Client

Currently the generated code of the client that communicates with the service is "hard wired" to a specific location, or endpoint, of the service. This presents a problem as the client gets deployed to different environments, like testing, staging, and production, because the client needs to use a service that is appropriate for the type of environment the client is in. Using the testing version of the service in a production client would be a big error because the service would not be using production data, etc.

Although there are many ways to approach changing the endpoint "binding" of the service client we will investigate one option that will allow this change to be made administratively without changing the code of the client directly. Allowing the server to "manage" this client endpoint binding will certainly give us more flexibility.

This is not the default way we have been using the generated code so far so we will need to change our client code somewhat to get this ability.

Part 1 - Change Deployment Method for Test Server

In order to have the ability to change the endpoint binding of the client we must modify the way the application is deployed to our test server. The default of deploying applications to a test server from Eclipse is to run the application with resources in the workspace. For whatever reason this does not enable the administrative capabilities we will need. In order for these capabilities to be available we need to have Eclipse copy the applications to the test server.

__1. Start the server.

__2. Right-click on the server and select **Add and Remove...**

__3. Select the **Remove All** button to remove all projects from the server.

__4. Click the **Finish** button in the dialog.

__5. Click OK.

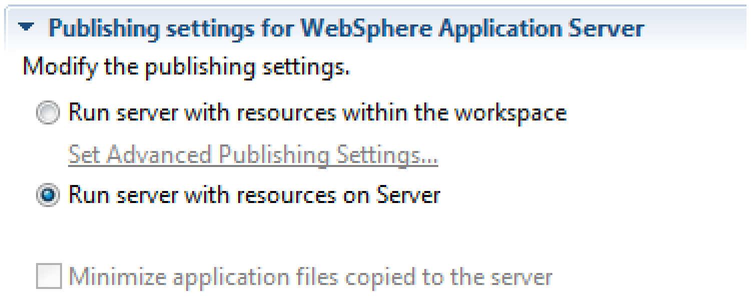
__6. Right click the server and select **Clean** and click **OK**.

__7. Stop the server.

__8. Double click the server to open the server configuration editor. You can also right click and select **Open** if double clicking does not open it.

__9. On the right side of the editor, expand the section for **Publishing settings for WebSphere Application Server traditional**.

__10. Select the '**Run server with resources on Server**' option as shown below if it's not already selected by default.


Publishing settings for WebSphere Application Server

Modify the publishing settings.

( ) Run server with resources within the workspace

Set Advanced Publishing Settings...

(●) Run server with resources on Server

[ ] Minimize application files copied to the server

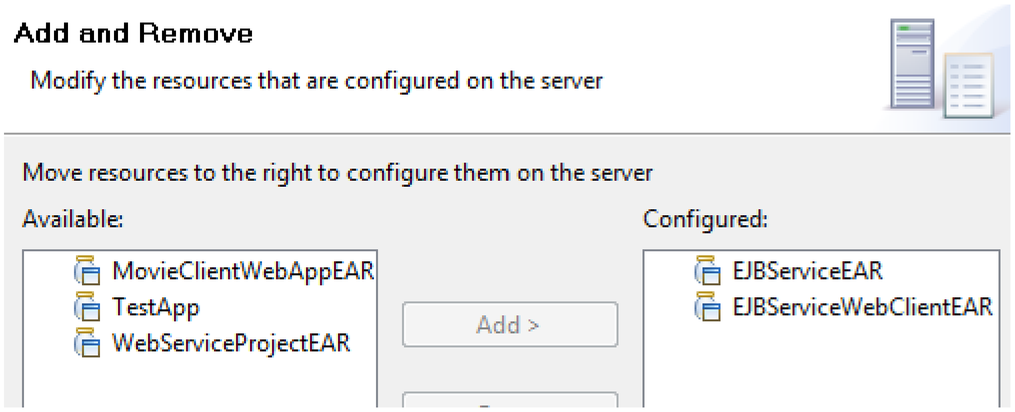
__11. Save and close the server configuration editor.

Note: We removed all of the projects because if you don't when you change the publishing settings you will get errors. We also stopped the server because the only way the server configuration can be edited is if the server is stopped.

__12. Start the server and be sure you don't get any errors in the **Console** view when starting.

__13. Right-click on the server and select **Add and Remove...**

__14. Add only the **EJBServiceEAR** and **EJBServiceWebClientEAR** projects.


Add and Remove

Modify the resources that are configured on the server

Move resources to the right to configure them on the server

Available:	Configured:
MovieClientWebAppEAR	EJBServiceEAR
TestApp	EJBServiceWebClientEAR
WebServiceProjectEAR	

__15. Click the **Finish** button in the dialog.

__16. Publish the server.

__17. Make sure you do not get any errors in the **Console** view when publishing.

Part 2 - Test Client

__1. Right-click on **input.html** (under **EJBServiceWebClient | WebContent**) and select **Run As | Run on Server**.

A browser will open. A simple form should appear.

__2. Fill in some sample input (without using "Elaine" as the name) and click the **Order** button.

__3. Make sure that the response shows the appropriate response.

__4. Click the back arrow button in the upper left of the embedded browser to return to the form.

__5. Fill in data using the name "Elaine" and click the **Order** button.

__6. Check that you get the correct output that is triggered by the web service fault being returned from the service back to the client.

As expected, the exception was caught and successfully handled by the servlet.

__7. Close the test browser but leave the server running.

Part 3 - Simulate Deploying the Client to a Different Environment

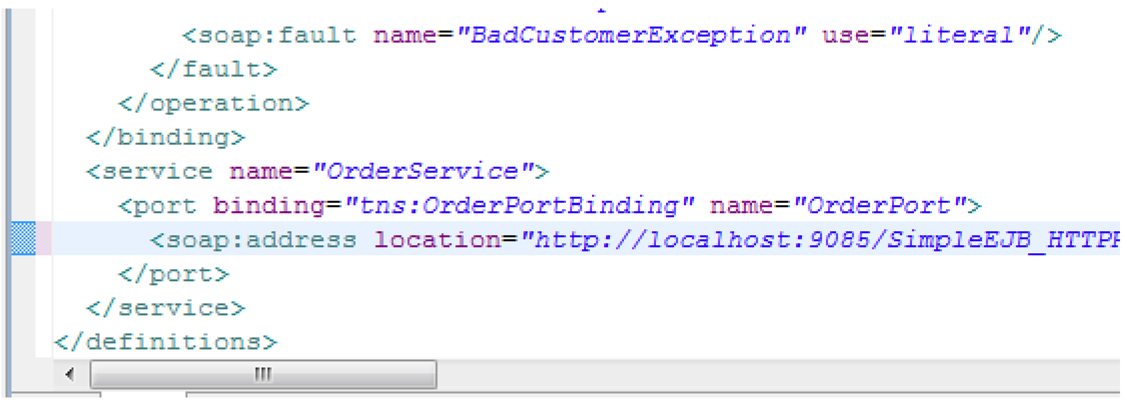
The goal of this lab is to show a way to change the endpoint the service client is bound to administratively at runtime. To simulate the idea that the endpoint in the code of the application is not appropriate for a different environment we will "break" the client code as it currently is implemented. This way when we make changes to the code and "fix" the endpoint administratively we know it is not the values in the code making things work but the ability we have from the Admin Console.

__1. Expand the **EJBServiceWebClient → WebContent → WEB-INF → wsdl** folder.

__2. Double click the **OrderSoupService.wsdl** file to open it. This file defines the endpoint location currently used by the client code.

__3. Switch to the **Source** tab.

__4. At the bottom of the file <u>change</u> the SOAP address location property to 'localhost:908**5**'. This port is not available on the server and will represent the client not being able to communicate with the version of the service used to develop the code.



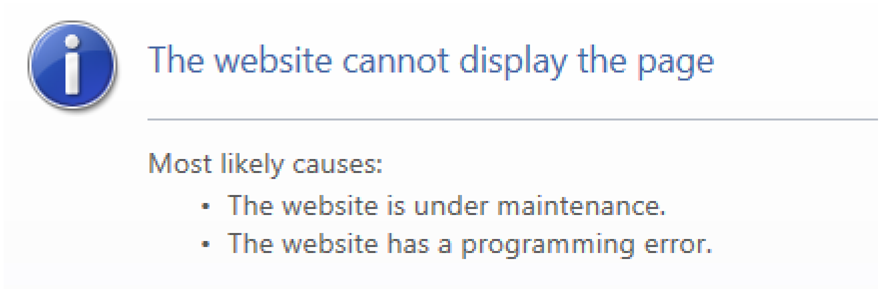
```
        <soap:fault name="BadCustomerException" use="literal"/>
      </fault>
    </operation>
  </binding>
  <service name="OrderService">
    <port binding="tns:OrderPortBinding" name="OrderPort">
      <soap:address location="http://localhost:9085/SimpleEJB_HTTPR
    </port>
  </service>
</definitions>
```

__5. Save and close the file.

__6. Publish the server.

__7. Right-click on **input.html** (under **EJBServiceWebClient | WebContent**) and select **Run As | Run on Server**.

__8. Fill in any name on the input form and click **Order**. You should get an error shown.

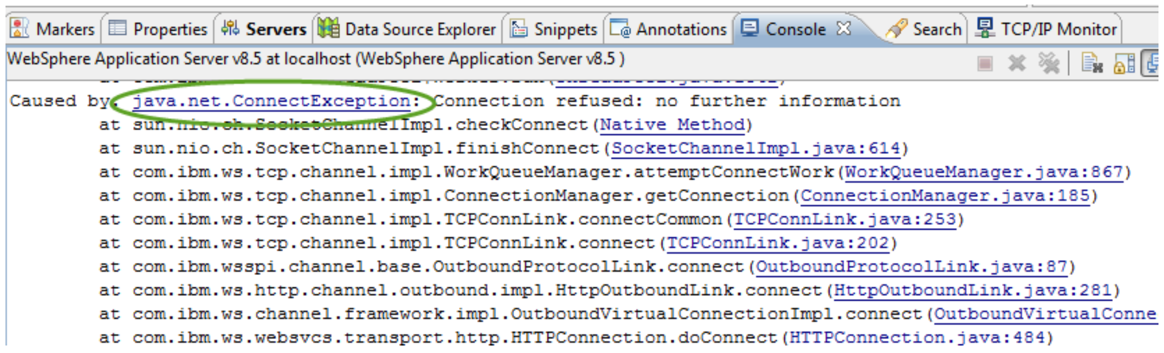


The website cannot display the page

Most likely causes:

- The website is under maintenance.
- The website has a programming error.

__9. If you look in the **Console** view you should be able to find an error that mentions a '**ConnectException**'. You will not get many details besides the type of error but it is clear this is related to a network communication problem.



```
Caused by: java.net.ConnectException: Connection refused: no further information
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:614)
        at com.ibm.ws.tcp.channel.impl.WorkQueueManager.attemptConnectWork(WorkQueueManager.java:867)
        at com.ibm.ws.tcp.channel.impl.ConnectionManager.getConnection(ConnectionManager.java:185)
        at com.ibm.ws.tcp.channel.impl.TCPConnLink.connectCommon(TCPConnLink.java:253)
        at com.ibm.ws.tcp.channel.impl.TCPConnLink.connect(TCPConnLink.java:202)
        at com.ibm.wsspi.channel.base.OutboundProtocolLink.connect(OutboundProtocolLink.java:87)
        at com.ibm.ws.http.channel.outbound.impl.HttpOutboundLink.connect(HttpOutboundLink.java:281)
        at com.ibm.ws.channel.framework.impl.OutboundVirtualConnectionImpl.connect(OutboundVirtualConne
        at com.ibm.ws.websvcs.transport.http.HTTPConnection.doConnect(HTTPConnection.java:484)
```

__10. Close the browser you were using to test the application but leave the server running.

Part 4 - Modify Code to Provide a Managed Endpoint Capability

At this point we have three ways of updating the client code to allow for linking the client application to the service in the "new" environment:

1. We could provide an updated version of the WSDL file in the application which is used by the client code.
2. We could add code in the client to set the endpoint based on a properties file or something else outside of the Java code.
3. We could use the @WebServiceRef annotation to allow the WebSphere environment to manage the endpoint and allow administrative configuration.

Of all three options the third is likely the best because we would not have to change ANYTHING in the EAR as it is moved between environments. It is tough to convince administrators to install and EAR file that is not the same exact version tested in development or staging no matter how small the change (the URL in a WSDL file) is.

__1. Expand the **EJBServiceWebClient → Java Resources → src → com.webage.ejb** folder.

__2. Open the **OrderSoupService.java** file by double clicking it.

__3. Notice the @WebServiceClient annotation. This declares a "name" for the service client and a default WSDL location which is the file you edited previously. The generated code also extends the javax.xml.ws.Service class.

```
@WebServiceClient(name = "OrderSoupService",
          targetNamespace = "http://ejb.webage.com/",
          wsdlLocation = "WEB-INF/wsdl/OrderSoupService.wsdl")
public class OrderSoupService
    extends Service
{
```

__4. Look for the constructors in the class a little further down. Notice that the constructor that doesn't take parameters uses the default WSDL location to construct a reference to the service.

```
    public OrderSoupService() {
        super(__getWsdlLocation(), ORDERSOUPSERVICE_QNAME);
    }
```

Note: Although it might be possible to write code that used the constructor that takes the WSDL location as a parameter, perhaps to a WSDL available over the network, this is still not a good idea. We would still need to modify the code in the application to provide a different WSDL location for each environment and retrieving a WSDL file over the network for every request from the client would not be good for performance either.

__5. Close the **OrderSoupService** class.

__6. Expand the **com.webage.servlet** package in the **EJBServiceWebClient** project.

__7. Open the **OrderServlet.java** file.

__8. In the **doGet** method remove the line of code that is crossed out below. You will get errors after doing so but they will be fixed later. You may also get errors if the server tries to republish but ignore them for now.

```
...
soupToOrder.setCustomerName(customerName);
soupToOrder.setSoupType(soupType);

~~OrderSoupService os = new OrderSoupService();~~
OrderSoup o = os.getOrderSoupPort();
...
```

Note: This will remove the call to the constructor of the OrderSoupService class. This constructor we have seen is not able to use any other endpoint than the one in the WSDL file packaged with the application.

__9. Add the following code in bold as a new field in the class. Make sure to add it inside the brackets for the class but outside of any other method.

```
public class OrderServlet extends HttpServlet {
          private static final long serialVersionUID = 1L;

          @WebServiceRef(name="OrderSoupService")
          private OrderSoupService os;
```

__10. Organize imports (CTRL-SHIFT-O) to include the import for the WebServiceRef annotation.

__11. Save the code and be sure there are no errors.

Part 5 - Change Endpoint Binding Administratively

The @WebServiceRef annotation on a field in the Servlet creates a setting that we can change administratively. In this section you will see how that is done. These steps are similar to steps that administrators might take to link the service client to the service implementation appropriate for the environment.

__1. Publish the server.

__2. Right-click on the server and select **Add and Remove...**

__3. Remove the **EJBServiceWebClientEAR** project by selecting it and clicking the **Remove** button.

__4. Click **Finish** in the dialog and click OK. Wait for the server to finish publishing.

__5. Publish the server.

__6. Right-click on the server and select **Add and Remove...**

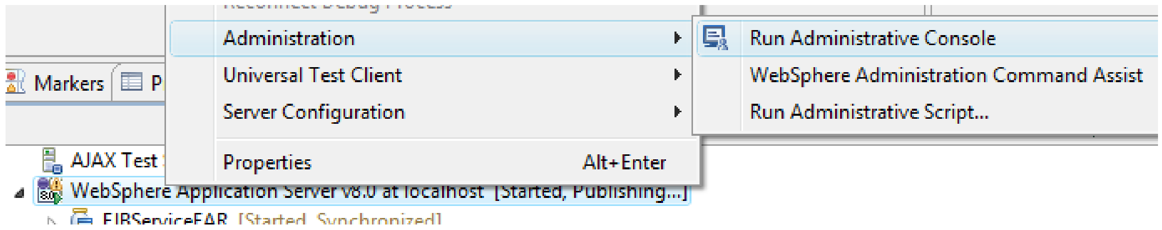
__7. Add the **EJBServiceWebClientEAR** project by selecting it and clicking the **Add** button.

__8. Click **Finish** in the dialog.

__9. Publish the server. Wait for the server to finish publishing.

Note: We are removing and adding the project to trigger the full deployment process. For whatever reason the server does not seem to pick up the addition of the @WebServiceRef annotation if the application is just "updated" so we have to do this to redeploy from scratch.

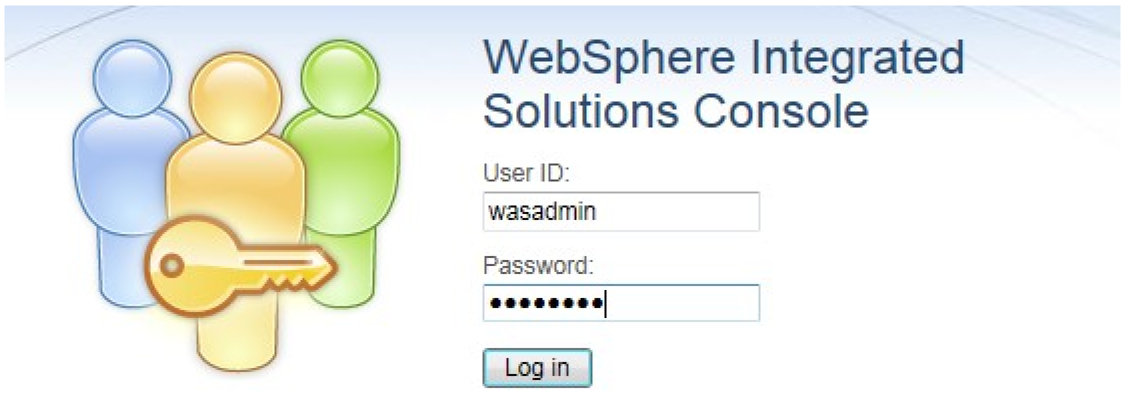
__10. Right click the server in the **Servers** view and select **Administration → Run Administrative Console**.


Administration
Universal Test Client
Server Configuration
Properties Alt+Enter
Run Administrative Console
WebSphere Administration Command Assist
Run Administrative Script...
Markers
AJAX Test
WebSphere Application Server v8.0 at localhost [Started, Publishing...]
EJBServiceEAR [Started, Synchronized]

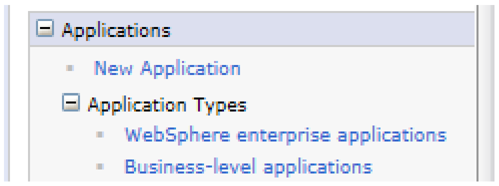
__11. Click Yes in the *Security Alert* window that may pop up.

__12. Click Yes in the second *Security Alert* window that may pop up.

__13. Enter wasadmin for **User ID** and **password** and click **Login** if prompt.

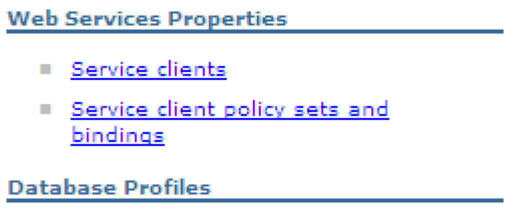


__14. Expand **Applications → Application Types → WebSphere enterprise applications** on the left.


Applications
New Application
Application Types
WebSphere enterprise applications
Business-level applications

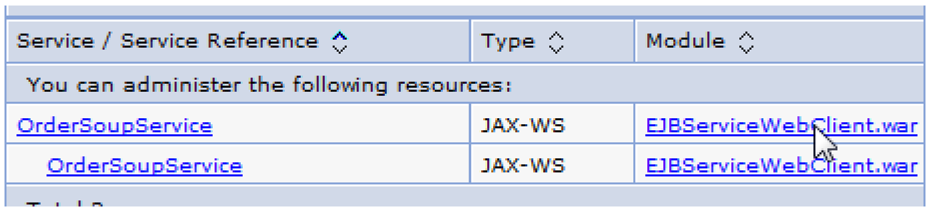
__15. Click the link for the **EJBServiceWebClientEAR** application.

__16. Scroll down and in the list of 'Web Services Properties' click the link for **Service clients**.



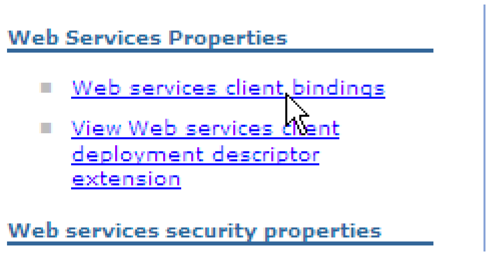
Note: This will list only the service clients in this application. You can list all service clients in the WebSphere environment by selecting '**Services → Service clients**'.

__17. Click the link for one of the **Modules** on the right. If you click the service name on the left it will take you to web service policies which are not being used for this web service.

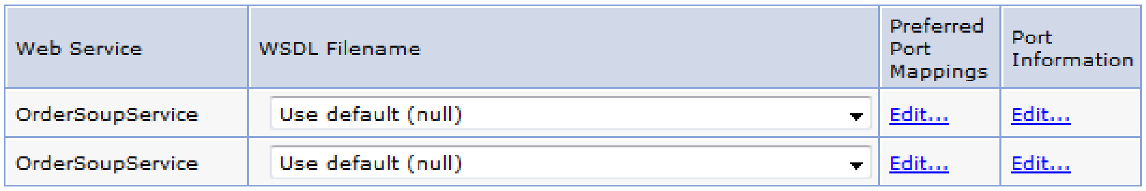


Service / Service Reference	Type	Module
You can administer the following resources:		
OrderSoupService	JAX-WS	EJBServiceWebClient.war
OrderSoupService	JAX-WS	EJBServiceWebClient.war

__18. Under 'Web Services Properties' click the link for **Web services client bindings**.


Web Services Properties
Web services client bindings
View Web services client deployment descriptor extension
Web services security properties

In the list you will see two entries for "services" that have the same name. This is because the code generated by the development tool contains a reference to the service as well as the Servlet used to send the service request. Since the "name" property of the annotations you saw is the same ("OrderSoupService") we only need to update the settings on one to modify the binding for both.

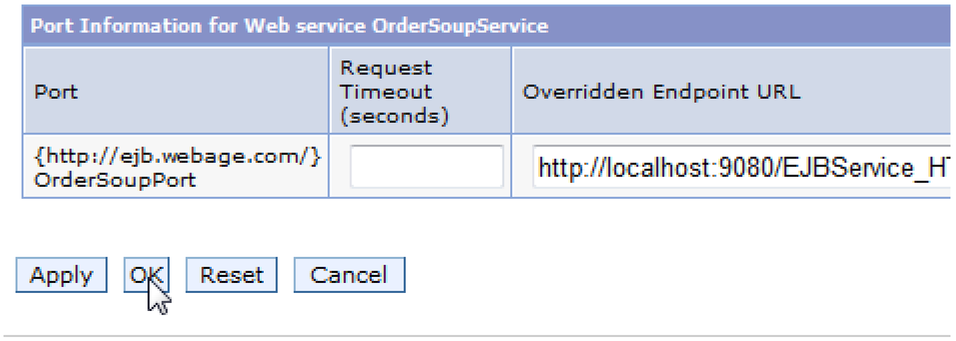


Web Service	WSDL Filename	Preferred Port Mappings	Port Information
OrderSoupService	Use default (null)	Edit...	Edit...
OrderSoupService	Use default (null)	Edit...	Edit...

__19. Click the first '**Edit**' link in the '**Port Information**' column. Make sure you do not click the link in the 'mappings' column.

For this client this setting might be a little confusing. Even though it looks like the correct URL is listed (with port 9080), this setting is not yet actually applied to the endpoint binding.

__20. Accept the default URL option as shown below and press the **OK** button. This will apply the URL with the correct port to the endpoint binding.

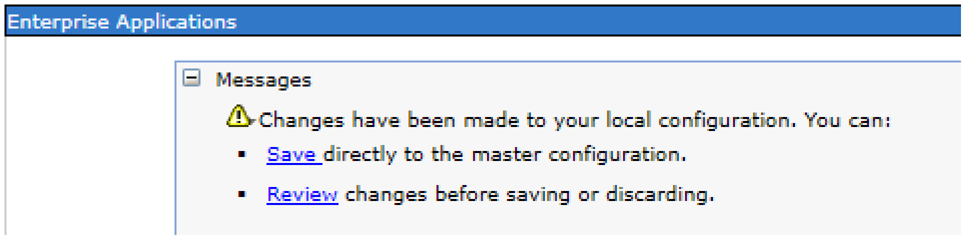


Port Information for Web service OrderSoupService

Port	Request Timeout (seconds)	Overridden Endpoint URL
{http://ejb.webage.com/}OrderSoupPort		http://localhost:9080/EJBService_H

Note: In a "real" environment where the service is likely not running on the same machine as the client you would obviously use the hostname and port of the service implementation here.

__21. Click the **Save** link in the messages at the top of the Admin Console.


Enterprise Applications

Messages

Changes have been made to your local configuration. You can:

- Save directly to the master configuration.
- Review changes before saving or discarding.

__22. Click the **Logout** link at the top of the Admin Console and close it.

Part 6 - Verify Client Works with the New Binding

We have already proven that if the client code uses the address in the WSDL in the application the client would not work. Now we need to prove that the @WebServiceRef annotation and being able to update the client binding in the Admin Console has "fixed" the client application.

__1. Right-click on **input.html** (under **EJBServiceWebClient | WebContent**) and select **Run As | Run on Server**.

__2. Fill in some sample input and click the **Order** button.

__3. You should get the "normal" output of the client which will be the "Served" or "Bad Customer" message depending on the name. You will not get the generic error page you saw before because of the connection error when the client used the wrong port.

Part 7 - Get WSDL Files of Published Service (Optional)

One thing we did not take a look at is how the service provider location in the WSDL was rewritten when it was deployed to the server. The steps in this section will show you how to get this information from the deployed server.

__1. Right click the server in the **Servers** view and select **Administration → Run Administrative Console**.

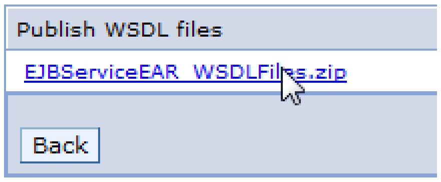
__2. Login again as wasadmin if prompt.

__3. In the Admin Console, navigate to **Applications → Application Types → WebSphere enterprise applications**.

__4. Click the link for the **EJBServiceEAR** application.

__5. Scroll down and in the list on the right of 'Web Services Properties' click the link for **Publish WSDL files**.

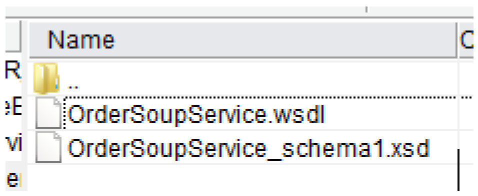
__6. On the page that appears click the link for the zip file of WSDL files.



Note: It is common to have a "service registry" in an architecture that includes web services. This is a central repository of information on the services available. This zip file contains the information that would be needed to register the service.

__7. If prompted to open or save the file, open it with zip file software if possible. If you can't open it directly save it first and then open the zip file.

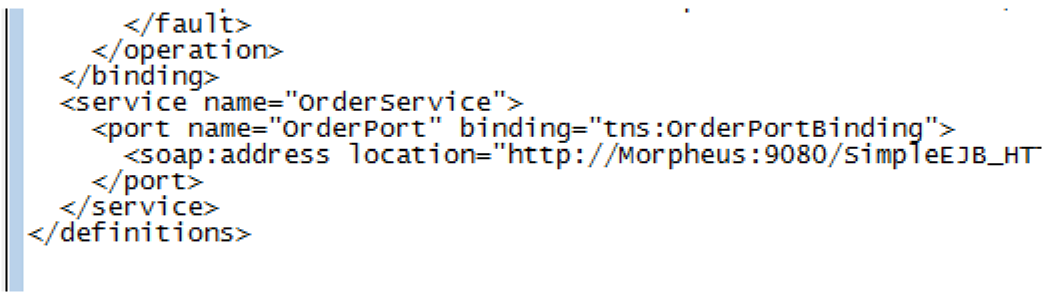
__8. Expand the folders of the zip file if needed until you see the '**OrderSoupService.wsdl**' file.



Note: The WSDL file references the XML schema file which is why they are packaged together in the zip file. Your own services might have more files but they will likely be mainly WSDL and XML schema files.

__9. Open the '**OrderSoupService.wsdl**' file with a text editor like WordPad. If you need to, extract the file first and then open it.

__10. Scroll down in the WSDL file and notice at the end that the SOAP address uses the actual hostname of your machine (which will be different than shown below). Since this address uses the actual hostname and not 'localhost' this address could be used by a client even on another machine to link to this service implementation.



```
      </fault>
    </operation>
  </binding>
  <service name="OrderService">
    <port name="OrderPort" binding="tns:OrderPortBinding">
      <soap:address location="http://Morpheus:9080/SimpleEJB_HT
    </port>
  </service>
</definitions>
```

Note: This address could have also been used in the properties of the client endpoint binding to link the client to the service in the Admin Console.

__11. Close the WSDL and the zip file.

__12. Click Logout and close the Admin Console.

__13. Close all.

Part 8 - Review

In a development environment it may be unlikely that you think much about how the client will be bound to the service in other environments. Although it would "work" to have the address of the service in the WSDL included in the client application, relying on this alone would not be very flexible.

In this lab we used the @WebServiceRef annotation to let WebSphere manage the client endpoint binding so we could also change it at runtime from the Admin Console. If we need to change this binding as the application moves between different environments (testing, staging, production, etc) we would not have to change anything in the EAR file itself.

One issue you have to deal with is that during the installation of the EAR file in the Admin Console you WILL NOT SEE options for the client endpoint binding. You have to follow the steps you took AFTER the application is deployed to the server to update this.

Remember, if the address of the web service running in testing or staging environments is available from the network of the production web service client environment this could be VERY BAD. You would not have seen the errors we saw earlier because the production client would still be able to communicate with the testing or staging service implementation. This is definitely not the behavior we want as we want to link the production web service client with the production service implementation. Using network firewalls to prevent this communication between different environments would be one way to avoid this.

Lab 14 - Getting Setup for WS-Security

Throughout the next few labs, we will apply various WS-Security protections to a web service provider and consumer. In this lab, we will install and test these applications before any security is enabled.

Tip: Always do function testing for a provider and consumer before security is enabled. This will help you debug problems when security related problems begin to appear.

The business logic for these applications is very simple. The provider is called BillingManagerService. It has only one operation called addAccount. A consumer calls this operation to create a new customer account.

The consumer is a web based application. When a new account form is submitted, it calls the addAccount operation of the web service.

Part 1 - Review Code

We will briefly review the code of the provider and consumer. They have been developed using the JAX-WS API. There is nothing special about them. In other words, there is nothing done to them specific to WS-Security. All security settings will later be applied through policies.

__1. Right-click on the Server in the list of servers and select **Add and Remove...**

The *Add and Remove Projects* window will appear.

__2. Click **Remove All**.

__3. Click **Finish**.

__4. Click OK.

__5. Stop the server.

We will now import the given projects and study before apply any security setting.

__6. From the menu select **File > Import**.

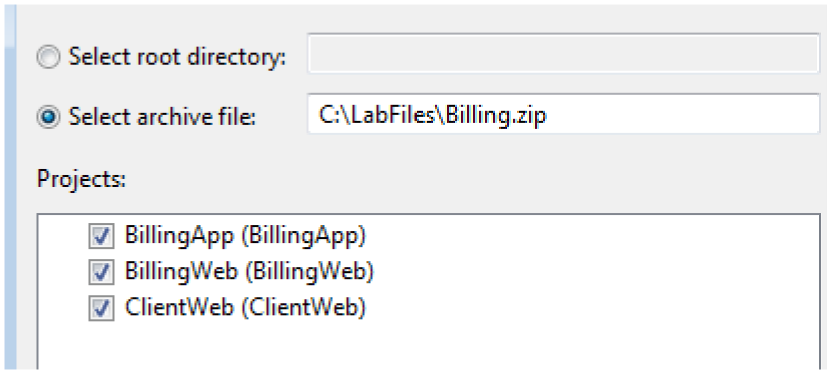
__7. Expand **General** and select **Existing Projects into Workspace**.

__8. Click **Next**.

__9. Chose the '**Select archive file**' option and click the second **Browse** button next to the archive option.

__10. Select the ZIP file **C:\LabFiles\Billing.zip** and click **Open**.

__11. Click **Select All** to select all projects.



( ) Select root directory:

(●) Select archive file: C:\LabFiles\Billing.zip

Projects:

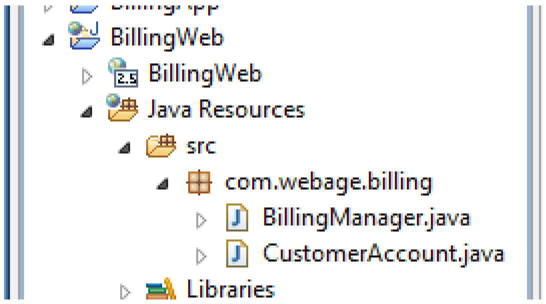
☑ BillingApp (BillingApp)
☑ BillingWeb (BillingWeb)
☑ ClientWeb (ClientWeb)

__12. Click **Finish** to import the projects.

First, we will review the service provider.

__13. Expand the **BillingWeb** project.

__14. Then expand **Java Resources > src > com.webage.billing**.



__15. Double click **BillingManager.java** to open the provider implementation class.

__16. Notice that the @WebService annotation is used for the class.

```
@WebService
public class BillingManager {
```

__17. The class has only one public method called addAccount. Various JAX-WS annotations are used with that method to customize the XML schema for the request and response.

```
@WebMethod
@WebResult(name="status")
public String addAccount(
          @WebParam(name="account")
          CustomerAccount account) {
```

This is pretty basic JAX-WS API. If you have any questions about them ask the instructor.

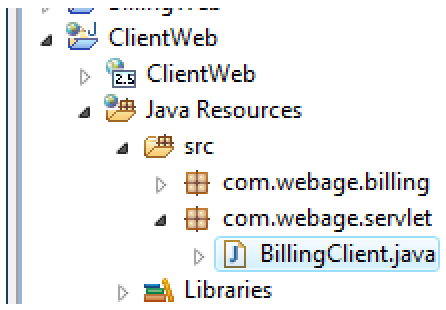
__18. Note that the addAccount operation simply prints out a few lines in the log file and finally returns "OK".

__19. Close the editor.

Now, we will review the consumer.

__20. Expand **ClientWeb** project.

__21. Then expand **Java Resources > src > com.webage.servlet**.



__22. Double click **BillingClient.java** to open the servlet implementation class.

__23. Observe the following:

- The Servlet has a field for a reference to a BillingManagerService with a @WebServiceRef annotation
- The Servlet processes the submission of a form, calls the service, and then forwards to the 'index.jsp' page for results.

__24. Close the editor.

Part 2 - Deploy the Applications

Normally, as a developer, you would deploy the projects to the server using Eclipse. But, in the following labs, we will do things like the administrators. That means, we will export an EAR file and deploy the EAR using WebSphere admin console. This will give us a feel for how policies are actually configured and attached to service providers and consumers in real life. That approach will also drive home the point that no code change is required to enable security.

__1. Open the **Servers** view.

__2. Start the server.

__3. Right click the server and select **Administration** → **Run Administrative Console**.

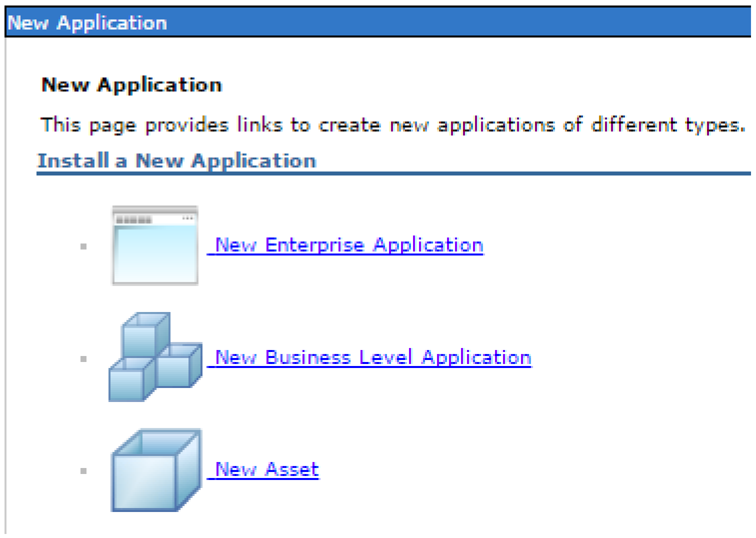
__4. If prompt, enter wasadmin as user ID/password and log in.

The admin console will open.

__5. Expand **Applications**.

__6. Click **New Application**.

__7. Click **New Enterprise Application**.


New Application

New Application

This page provides links to create new applications of different types.

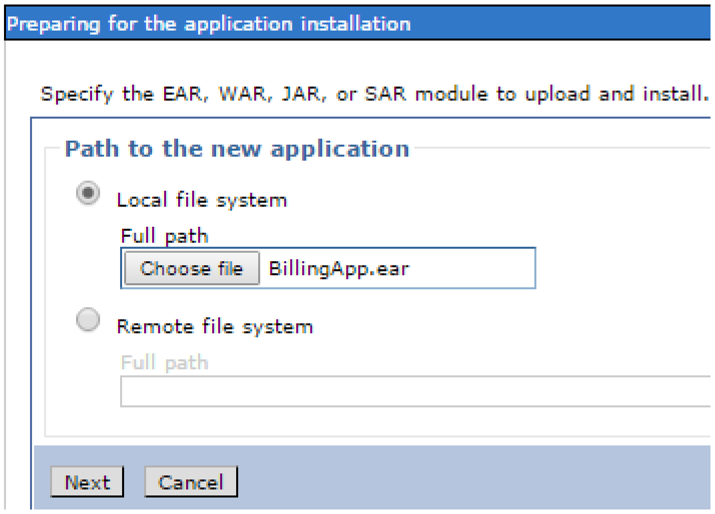
Install a New Application

- New Enterprise Application
- New Business Level Application
- New Asset

__8. Make sure that **Local file system** is selected.

__9. Click **Browse**.

__10. Select **C:\LabFiles\BillingApp.ear** and click **Open**.


Preparing for the application installation
Specify the EAR, WAR, JAR, or SAR module to upload and install.
Path to the new application
Local file system
Full path
Choose file
BillingApp.ear
Remote file system
Full path
Next
Cancel

__11. Click **Next**.

__12. Leave the default in the **How do you want to install the application?** page and click **Next**.

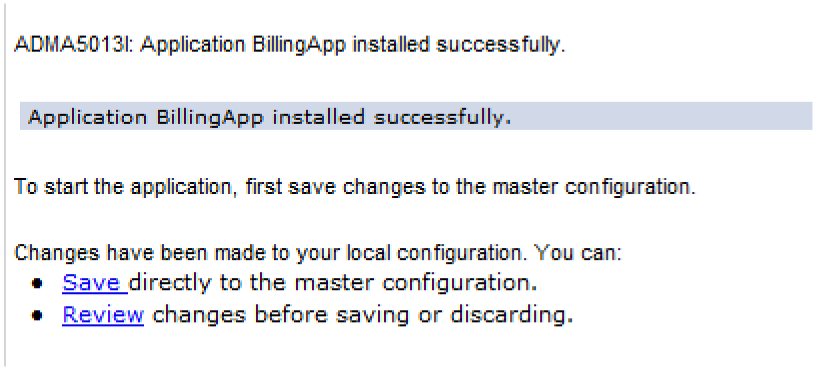
__13. In the **Step1: Select installation options** page just click **Next**.

__14. In the **Step 2: Map modules to servers** page, again just click **Next**.

__15. In the **Step 3: Metadata for modules** page, again just click **Next**.

__16. Click **Finish**.

The application will be installed, wait until you see the following message:



ADMA5013I: Application BillingApp installed successfully.

Application BillingApp installed successfully.

To start the application, first save changes to the master configuration.

Changes have been made to your local configuration. You can:
- Save directly to the master configuration.
- Review changes before saving or discarding.

__17. Click **Save**.

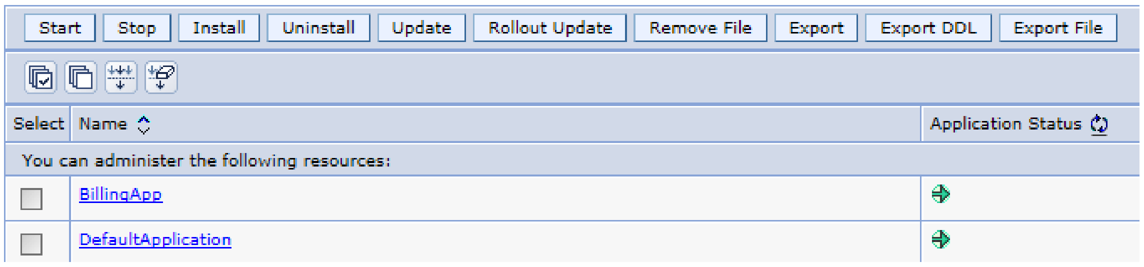
Part 3 - Test the Application

__1. On the left hand side, expand **Applications > Application Types**.

__2. Click **WebSphere enterprise applications**.

__3. Select the check box next to **BillingApp**.

__4. Click the **Start** button and wait until the status shows started.



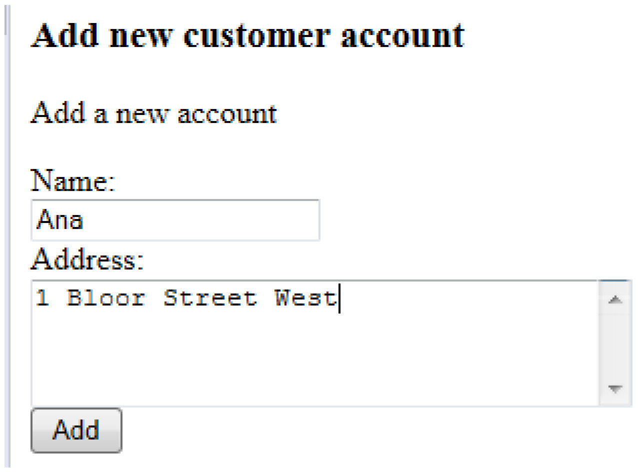
Select	Name	Application Status
You can administer the following resources:		
☐	BillingApp	➜
☐	DefaultApplication	➜

__5. Launch a web browser window.

__6. Enter the URL:

```
http://localhost:9080/ClientWeb/index.jsp
```

__7. The page will be displayed. Add some information and click **Add**.



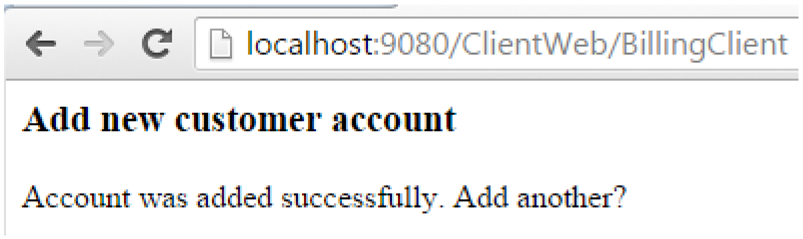
Add new customer account

Add a new account

Name: Ana

Address: 1 Bloor Street West

__8. The JSP will call the web service. Make sure that the page shows the success message:

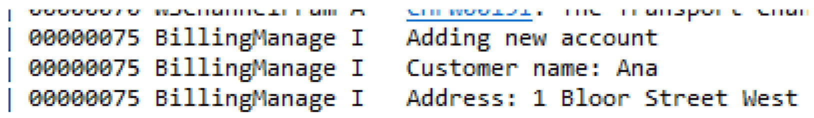


localhost:9080/ClientWeb/BillingClient

Add new customer account

Account was added successfully. Add another?

__9. Back in Eclipse, the **Console** view will show the log output from the web service.



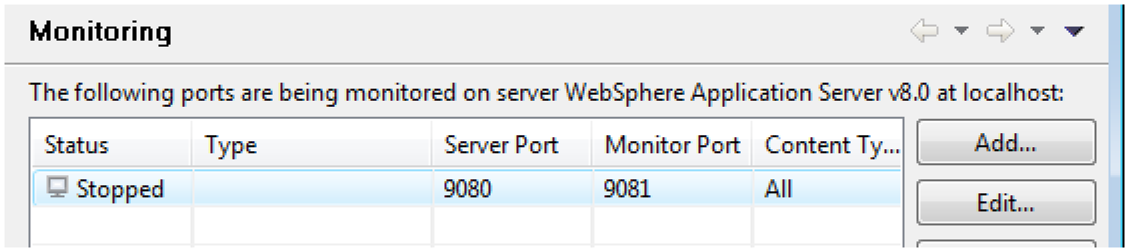
```
| 00000075 BillingManage I   Adding new account
| 00000075 BillingManage I   Customer name: Ana
| 00000075 BillingManage I   Address: 1 Bloor Street West
```

Part 4 - Enable SOAP Monitoring

We will use the TCP monitor tool to inspect SOAP messages.

__1. In the **Servers** view, right click the server and select **Monitoring > Properties**.

__2. If you already have a monitor for port 9080 as shown below, skip to step 6. If you do not have this port continue with the next few steps to create it.



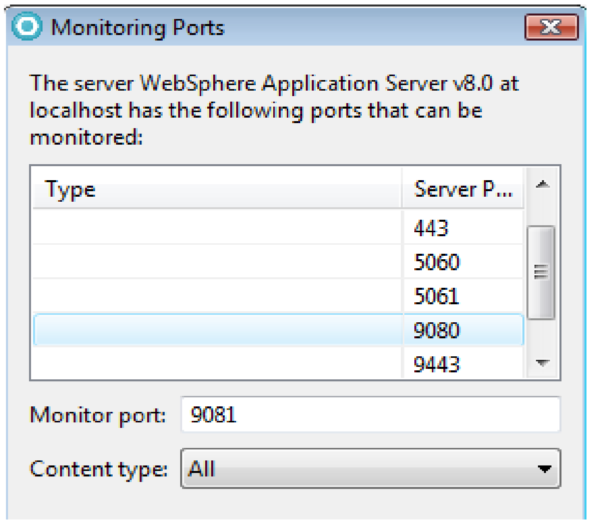
Monitoring

The following ports are being monitored on server WebSphere Application Server v8.0 at localhost:

Status	Type	Server Port	Monitor Port	Content Ty...
Stopped		9080	9081	All

__3. Click **Add**.

__4. Select the port **9080**



The server WebSphere Application Server v8.0 at localhost has the following ports that can be monitored:

Type	Server P...
	443
	5060
	5061
	9080
	9443

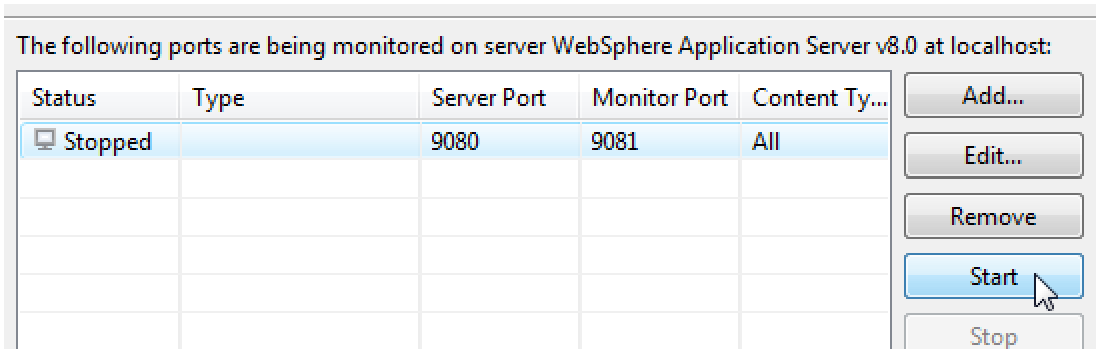
Monitor port: 9081

Content type: All

The **Monitor port** will be set to 9081 by default. That means, all HTTP requests submitted to localhost:9801 will be forwarded to 9080.

__5. Click **OK**.

__6. Select the monitor port for 9080 and click **Start**.



The following ports are being monitored on server WebSphere Application Server v8.0 at localhost:

Status	Type	Server Port	Monitor Port	Content Ty...
Stopped		9080	9081	All

__7. Click **Apply and Close**.

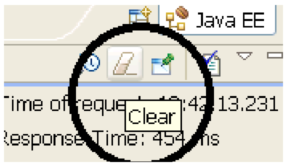
__8. Select **Window > Show View > Other**.

__9. Expand **Debug** and select **TCP/IP Monitor**.

__10. Click **Open**.

__11. Select the **TCP/IP Monitor** view.

__12. Click the **Clear** button.

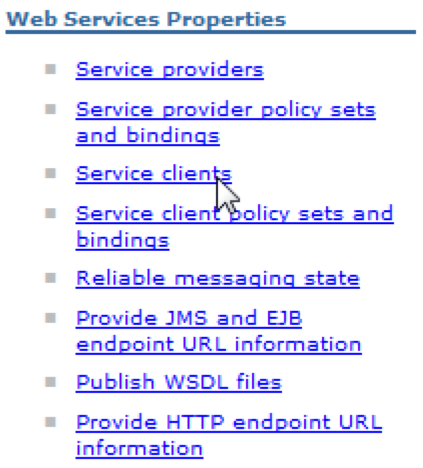


By default, the consumer submits SOAP request to port 9080. We need to change it to 9081.

__13. In the Admin Console, navigate to **Applications** → **Application Types** → **WebSphere enterprise applications**.

__14. Click the link for the **BillingApp** application.

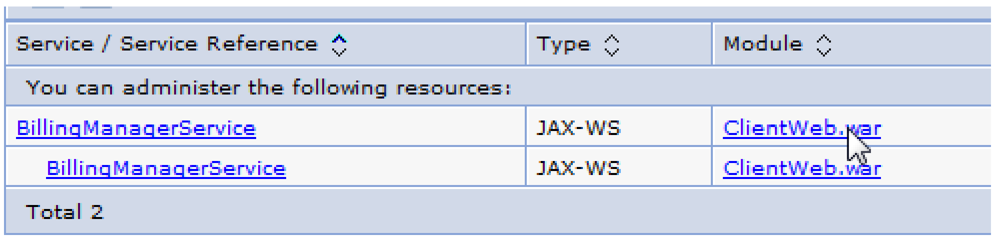
__15. Scroll down and in the list of 'Web Services Properties' click the link for **Service clients**.



Web Services Properties

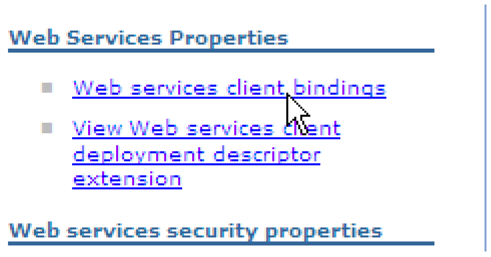
- Service providers
- Service provider policy sets and bindings
- Service clients
- Service client policy sets and bindings
- Reliable messaging state
- Provide JMS and EJB endpoint URL information
- Publish WSDL files
- Provide HTTP endpoint URL information

__16. Click the first link for the **Module** on the right. If you click the service name on the left it will take you to web service policies which are not being used for this web service.

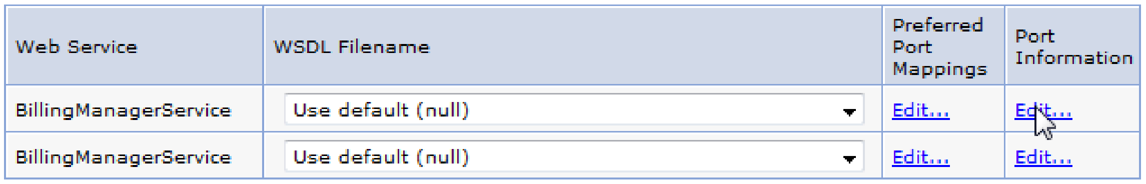


Service / Service Reference	Type	Module
You can administer the following resources:		
BillingManagerService	JAX-WS	ClientWeb.war
BillingManagerService	JAX-WS	ClientWeb.war
Total 2		

__17. Under 'Web Services Properties' click the link for **Web services client bindings**.

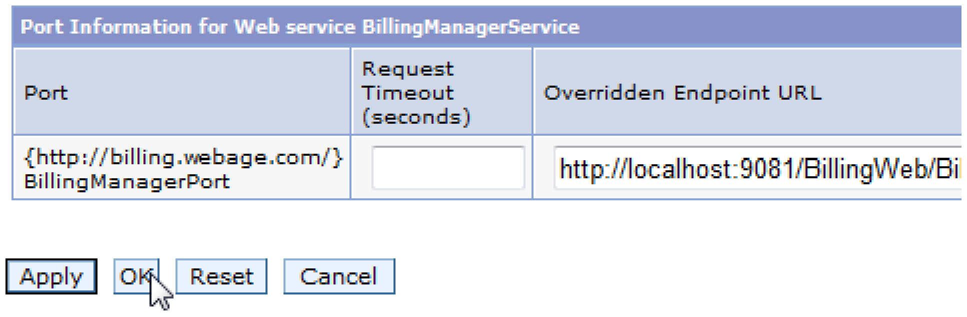

Web Services Properties
Web services client bindings
View Web services client deployment descriptor extension
Web services security properties

__18. Click the '**Edit**' link in the '**Port Information**' column. Make sure you do not click the link in the 'mappings' column.



Web Service	WSDL Filename	Preferred Port Mappings	Port Information
BillingManagerService	Use default (null)	Edit...	Edit...
BillingManagerService	Use default (null)	Edit...	Edit...

__19. Change the port to **9081** in the URL option as shown below and press the **OK** button. This will apply the URL with the correct port to the endpoint binding.



Port Information for Web service BillingManagerService

Port	Request Timeout (seconds)	Overridden Endpoint URL
{http://billing.webage.com/} BillingManagerPort		http://localhost:9081/BillingWeb/Bi

__20. Click the **Save** link in the messages at the top of the Admin Console.

__21. Click the **Logout** link at the top of the Admin Console and close it.

__22. Go back to the consumer web page and refresh the page:

```
http://localhost:9080/ClientWeb/index.jsp
```

__23. Enter some input data and click **Add**.

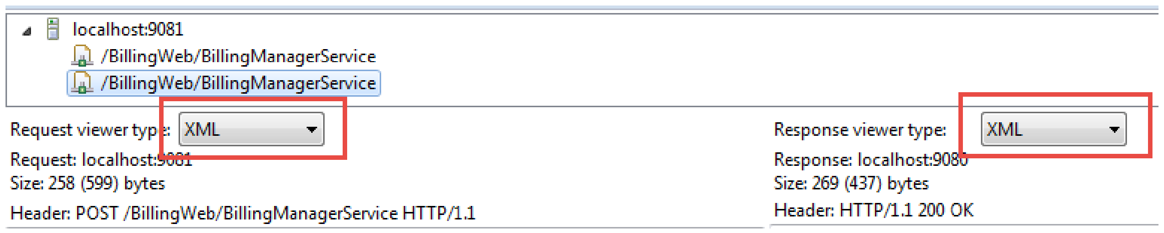
__24. Make sure that the success message is shown.

We will now inspect the SOAP messages.

__25. Select **TCP/IP Monitor** view.

__26. Maximize the **TCP/IP Monitor** view.

__27. You should see a request already captured. Select it and set the display type to XML.

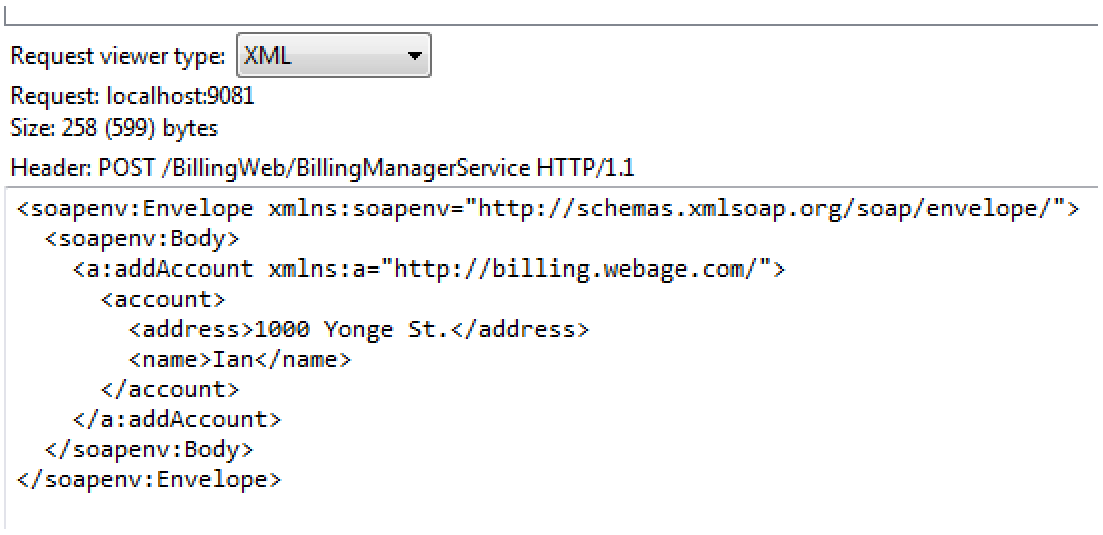


```
localhost:9081
    /BillingWeb/BillingManagerService
    /BillingWeb/BillingManagerService
```

Request viewer type: XML
Request: localhost:9081
Size: 258 (599) bytes
Header: POST /BillingWeb/BillingManagerService HTTP/1.1

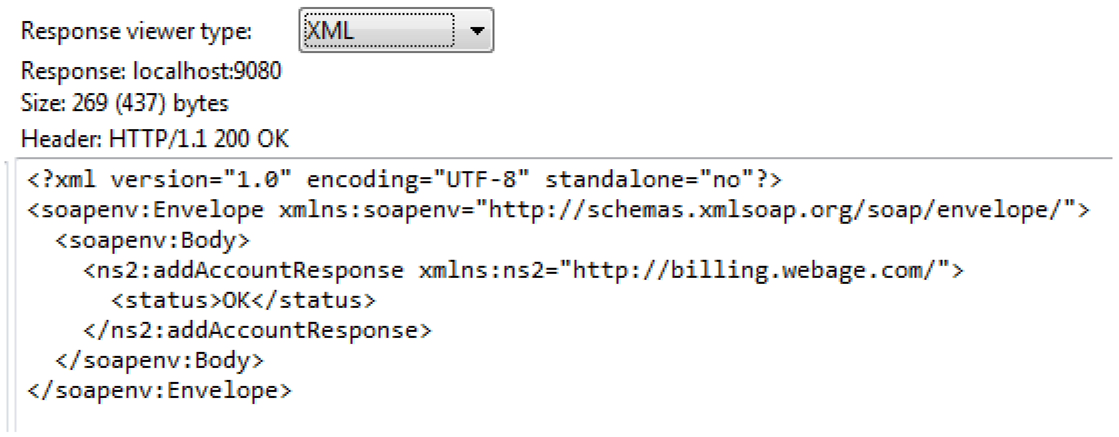
Response viewer type: XML
Response: localhost:9080
Size: 269 (437) bytes
Header: HTTP/1.1 200 OK

__28. Study the request and response messages. They are in plain text since we have not enabled any message security.



Request viewer type: XML
Request: localhost:9081
Size: 258 (599) bytes
Header: POST /BillingWeb/BillingManagerService HTTP/1.1

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <a:addAccount xmlns:a="http://billing.webage.com/">
      <account>
        <address>1000 Yonge St.</address>
        <name>Ian</name>
      </account>
    </a:addAccount>
  </soapenv:Body>
</soapenv:Envelope>
```



Response: localhost:9080
Size: 269 (437) bytes
Header: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns2:addAccountResponse xmlns:ns2="http://billing.webage.com/">
      <status>OK</status>
    </ns2:addAccountResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

__29. Click the eraser button in the TCP monitor view toolbar to remove all captured messages.

__30. Restore the size of the **TCP/IP Monitor** view.

__31. Close the web browser.

Part 5 - Review

In this lab, we installed a basic web service provider and consumer. They have been developed using the JAX-WS API.

We tested the applications to make sure that they are functionally correct.

We have enabled TCP monitoring so that we can inspect the SOAP messages.

In the subsequent labs, we will enable various message protection features.

Lab 15 - Message Integrity using WS-Security

In this lab, we will implement message integrity for the BillingManagerService provider and consumer. But, before that we will review the basics of WS-Security.

The BillingManagerService web service so far has been operating without any form of security. Our clients have been sending SOAP messages to the web service in clear plain text, with no regard for security at all.

This is clearly not practical in many production environments. For a web service, the concept of security has three concerns:

Integrity: Integrity ensures that a message came from a unique party, and that the message has not been tampered with in transmission. This is achieved by *signing* the outgoing SOAP message. In this lab exercise, we will implement message integrity.

Authority: Ensures that an invocation of a web service is only allowed for authenticated clients. This is achieved by adding a username/password token to the message.

Confidentiality: Ensures that a message being transmitted can only be understood by the receiving party. This is achieved by encrypting the outgoing SOAP message.

Ideally, we would like to add some combination of these 3 factors into our web service layer. Invoking a service should be done with integrity, with authority and with confidentiality.

But how can we do this? The hard way would be to add code to our SOAP layer. Remember that SOAP (XML) is the underlying communication mechanism for a client and service. For confidentiality, we could go about adding code to our service to return encrypted responses; for integrity, we could add code to our client to attach a digital signature. This would involve an extremely long and complex series of coding additions to our client.

Fortunately for us, the JAX-WS spec can help us avoid having to do just that. JAX-WS, instead makes use of *policies*, which will provide all the security layers that we need.

Security in web services is handled by using the concept of *policy set*. A policy set is a set of *policies*; policies are rules (defined by a specification) specifying what features should be enabled. For example, a policy can be created stating that *integrity* should be enabled; or *integrity* and *authority*; or even integrity and authority and encryption. (Multiple policies can be combined into a single policy set). These policies are defined by the WS-Security specification, and any JAX-WS compliant run-time must support these policies.

Once a policy has been created (in the form of an XML file), it is attached to a service, using vendor tools. No code changes have to be made. Now, when a client attempts to invoke the service, the container (WebSphere) will insist that the client encode the request message appropriately (e.g. encrypting the message, attach a signature, etc). If the client does not have the appropriate security measures in place, the service will reject the client.

So, we know how to get the service to require security; but how do we get the client to attach the required security? Again, the answer is the use of a policy. A policy can also be attached to a client. The client stub will read the attached policy and at invocation time, the client stub will generate the required SOAP – encrypting, attaching signatures, etc- as necessary.

This means our client code does not have to change; we just have to make sure we attach the policy set to the client stub, and the client stub will handle all the rest.

In this lab, we will examine a simple policy set to enable security; we will then attach it to a service (and client) and invoke it.

Part 1 - Understanding Policy Set

A policy set is a ZIP file consisting of a file called **policySet.xml** as well as one or more **policy.xml** files, each in its own sub-directory. Each **policy.xml** file contains a list of features to be enabled. The **policySet.xml** file then contains a list of which **policy.xml** files to use.

Once a policy set ZIP file has been created, it needs to be *imported* into the run-time (i.e. Eclipse and WebSphere). When imported, the run-time will examine set the policy set's **policySet.xml** and, in turn, read each **policy.xml** that is referenced.

Out of the box, several policy sets are available. For example, WebSphere and Eclipse come bundled with a policy set called **"WS-ReliableMessaging"** - which enables reliable messaging. We saw this feature in the earlier labs.

In this lab, however, we will use our own policy set. We will now briefly inspect a policy set ZIP file. We will inspect the policies in more details from within WebSphere admin console.

__1. Using a Windows file explorer, navigate to **C:\LabFiles**. Notice that it contains a file called **IntegrityOnly.zip**. This is the policy set has a policy that enables message integrity.

__2. Open the ZIP file using Winzip or any other archiving program.

__3. In the **PolicySets\IntegrityOnly** folder, locate the **policySet.xml** file.

__4. Open the **policySet.xml** file. Note that the set includes a single policy as follows:

```
<ps:PolicyType type="WSSecurity" provides="" enabled="true">
</ps:PolicyType>
```

__5. Close the file.

__6. Open the **PolicyTypes** sub-folder. In here, there is a sub-directory called **WSSecurity** – which is exactly what the **PolicyType** element in **policyTypes.xml** referred to. Basically, the run-time will read the **policyTypes.xml** file and then search for an appropriate sub-directory with the same name.

But what goes into the policy type sub-directory?

__7. Open the **WSSecurity** folder.

It contains a single file – **policy.xml.** This file specifies what features to activate.

__8. Open **policy.xml** with a text editor.

Ouch. This is a fairly complex XML file. For now, focus on these elements:

```
    <wsp:Policy wsu:Id="request:app_signparts">
        <sp:SignedParts>
            <sp:Body/>
...
        </sp:SignedParts>
```

This enables signing of the SOAP <Body> element.

__9. Close the file and close the ZIP file.

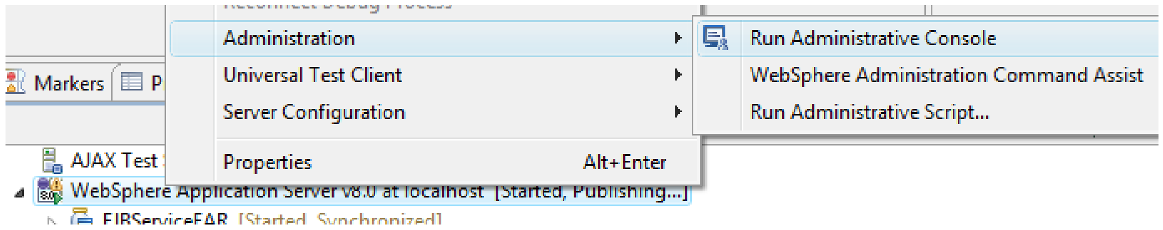
Part 2 - Import the Policy Set

Before a policy set can be attached to a service provider or consumer, we must import it in the WebSphere profile. We will do so now.

__1. In Eclipse, locate the **Servers** view.

__2. Start the server if it is not running.

__3. Right click the server in the **Servers** view and select **Administration** → **Run administrative console**.

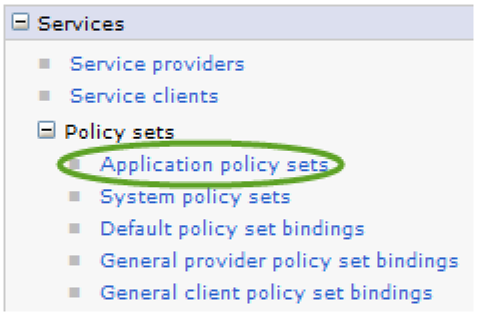

Administration
Universal Test Client
Server Configuration
Properties
Alt+Enter
Run Administrative Console
WebSphere Administration Command Assist
Run Administrative Script...
Markers
AJAX Test
WebSphere Application Server v8.0 at localhost [Started, Publishing...]
EJBServiceEAR [Started, Synchronized]

__4. Login as wasadmin if prompt.

The admin console will open inside Eclipse.

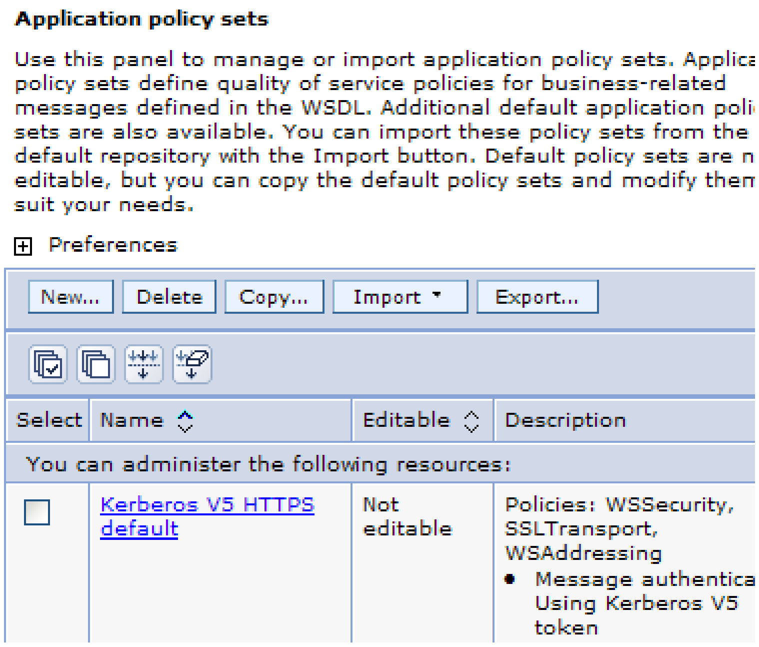
This is the administrative front end to WebSphere that you saw in the previous Lab, and is where a WebSphere administrator would perform the majority of administration tasks.

__5. In the left pane, expand **Services | Policy Sets**.


Services
Service providers
Service clients
Policy sets
Application policy sets
System policy sets
Default policy set bindings
General provider policy set bindings
General client policy set bindings

__6. Click **Application policy sets**.

The right pane will display the *Application policy sets* screen.

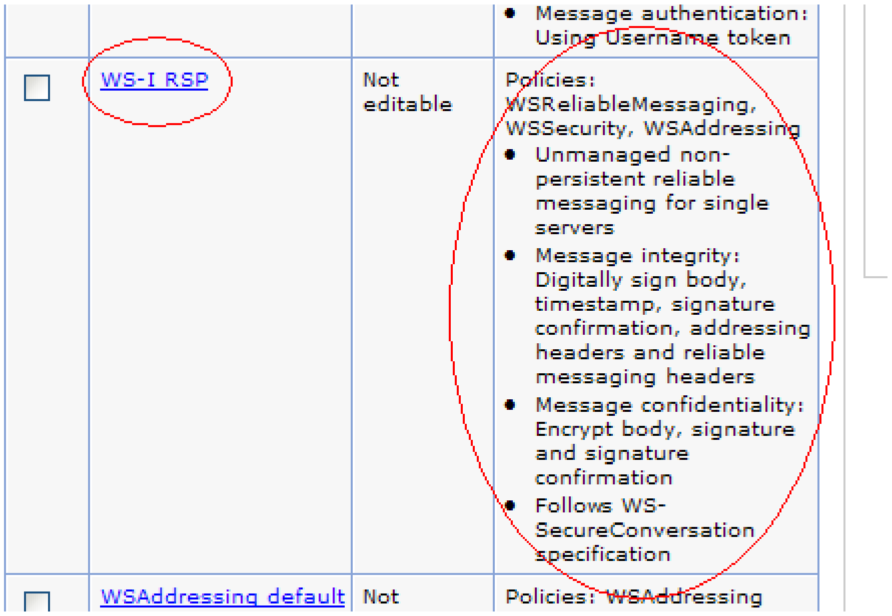

Application policy sets

Use this panel to manage or import application policy sets. Applica
policy sets define quality of service policies for business-related
messages defined in the WSDL. Additional default application poli
sets are also available. You can import these policy sets from the
default repository with the Import button. Default policy sets are n
editable, but you can copy the default policy sets and modify them
suit your needs.

Select	Name	Editable	Description
You can administer the following resources:			
☐	Kerberos V5 HTTPS default	Not editable	Policies: WSSecurity, SSLTransport, WSAddressing ● Message authentica Using Kerberos V5 token

This lists all the policy sets that are currently deployed to WebSphere. As mentioned above, there are a few “out of the box” policy sets available.

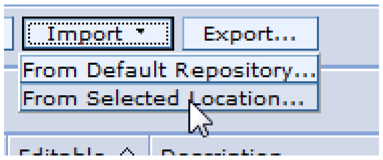
__7. Scroll down the list to examine some of them and what they offer. For example, examine the **WS-I RSP** profile, and look at what it offers.



			• Message authentication: Using Username token
☐	WS-I RSP	Not editable	Policies: WSReliableMessaging, WSSecurity, WSAddressing • Unmanaged non-persistent reliable messaging for single servers • Message integrity: Digitally sign body, timestamp, signature confirmation, addressing headers and reliable messaging headers • Message confidentiality: Encrypt body, signature and signature confirmation • Follows WS-SecureConversation specification
☐	WSAddressing default	Not	Policies: WSAddressing

Instead of using one of these existing profiles, however, we will use the one we examined previously. To do so, we need to import it here.

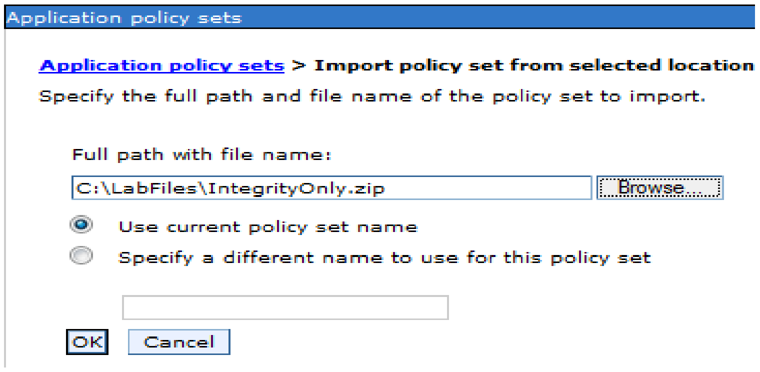
__8. Scroll up. Click the **Import** button, and select **From Selected Location...** from the drop-down.



The screen will change to allow you to specify a location for the policy set.

__9. Click the **Browse...** button.

__10. Select **C:\LabFiles\IntegrityOnly.zip** and click **Open**.


Application policy sets

Application policy sets > **Import policy set from selected location**

Specify the full path and file name of the policy set to import.

Full path with file name:

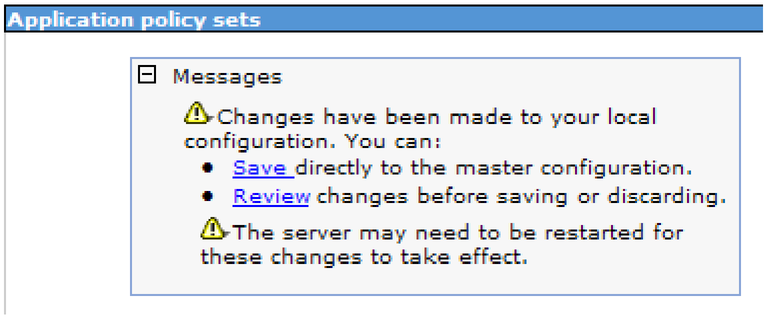
C:\LabFiles\IntegrityOnly.zip

(●) Use current policy set name

( ) Specify a different name to use for this policy set

__11. Click **OK**.

A *Messages* pane will appear.


Application policy sets

Messages

Changes have been made to your local configuration. You can:

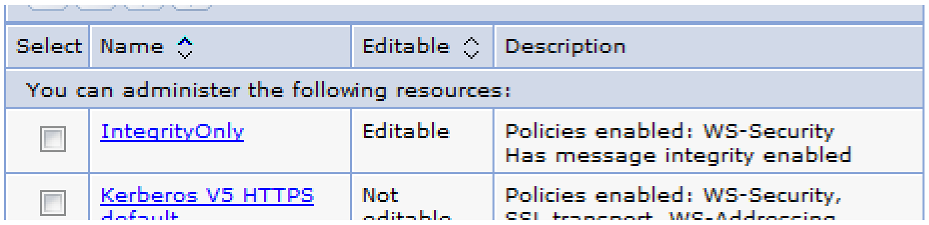
- Save directly to the master configuration.
- Review changes before saving or discarding.

The server may need to be restarted for these changes to take effect.

__12. Click the **Save** link.

The policy set will be imported.

__13. In the list of policies, scroll down. You should see the newly imported policy set.


Select	Name	Editable	Description
You can administer the following resources:			
☐	IntegrityOnly	Editable	Policies enabled: WS-Security Has message integrity enabled
☐	Kerberos V5 HTTPS	Not	Policies enabled: WS-Security,

Part 3 - Attach Policy Set

To enable message integrity, we need to attach the policy set to the provider. And since the consumer is also running in WebSphere, we need to attach the policy set to it as well. If the consumer was running in a different platform, say .NET, you will have to use vendor specific ways to enable integrity. If you do not enable integrity at the client level, the server will reject a SOAP request that does not contain a digital signature of the message.

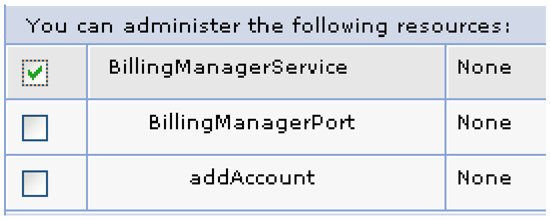
First, we will attach the policy set to the provider.

Note: A policy set can be attached to a provider or consumer using either Eclipse 7.5 or WebSphere adminconsole. The former is perhaps easier during development. Here, we will do things like an administrator would in a production environment. That means, we will attach policy sets using the adminconsole.

__1. On the left hand side of admin console, expand **Services** and click **Service providers**.

__2. Click **BillingManagerService**.

__3. Select the checkbox for **BillingManagerService**.

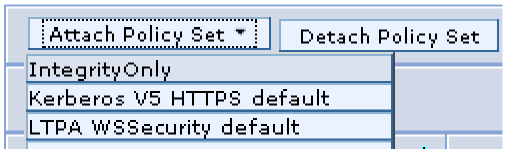


You can administer the following resources:

✓	BillingManagerService	None
	BillingManagerPort	None
	addAccount	None

This will select the entire service so that we can attach the policy set to it. Alternatively, you can select a specific operation, such as addAccount, and attach a policy set to it. That way, different operations can have different policies. In this lab, we will keep things simple and attach the policy set for the whole service.

__4. Click the **Attach Policy Set** button and click **IntegrityOnly**.



Now, we need to attach a policy set binding to the provider.

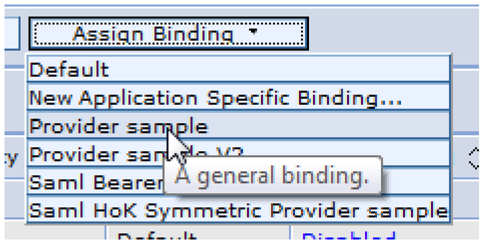
What is a **policy set binding**? A policy simply states what communication behavior should be enabled. For example, integrity and encryption. Various encryption and digest algorithms are also specified in the policy. A policy *does not* specify machine or installation specific details. For example, it does not describe the location of the private key and the password to open the key store file. Policy set binding is used to describe those items.

A policy set is generic and not specific to any business or installation of application server. For example, the same policy set ZIP file can be copied from development machine to production without any modification. A policy set binding ZIP file, on the other hand, contains installation specific details.

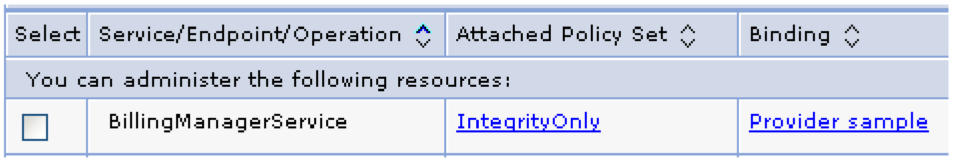
In this and the next few labs, we will use a predefined policy set binding. This binding uses pre-generated keys for the provider and consumer. This is good enough for development. In a production environment, you must generate your own keys. Preferably a key should be signed by a well known certificate authority. We will learn about key management and custom binding design in a latter lab.

__5. Select the checkbox for **BillingManagerService** again.

__6. Click **Assign Binding > Provider sample**.



__7. Verify that the correct policy set and binding is displayed for the service.



Select	Service/Endpoint/Operation	Attached Policy Set	Binding
You can administer the following resources:			
☐	BillingManagerService	IntegrityOnly	Provider sample

__8. Click the **Save** link at the top of the page to save changes.

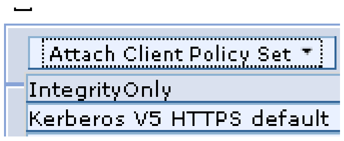
Now, we will configure the consumer.

__9. On the left hand side, click **Service clients** under **Services**.

__10. Click the first **BillingManagerService**.

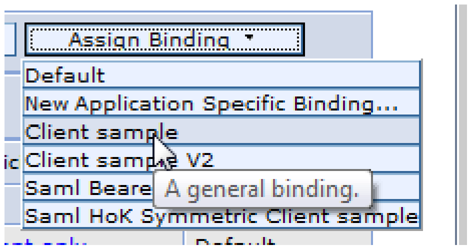
__11. Select the checkbox for **BillingManagerService**.

__12. Click **Attach Client Policy Set > IntegrityOnly**.

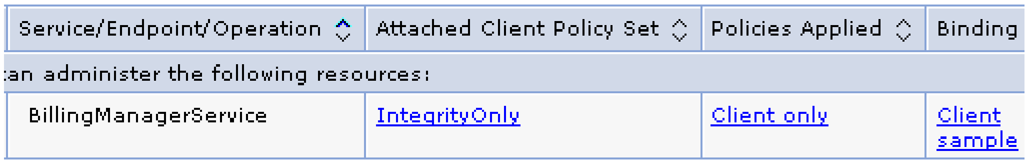


__13. Select the checkbox for **BillingManagerService** again.

__14. Click **Assign Binding > Client sample**.



__15. Verify that the correct policy set and binding is displayed for the consumer.



Service/Endpoint/Operation	Attached Client Policy Set	Policies Applied	Binding
:an administer the following resources:			
BillingManagerService	IntegrityOnly	Client only	Client sample

__16. Click the **Save** link at the top of the page to save changes.

Now, integrity should enabled. Before we can test things, we need to restart the server. **Note:** In WebSphere 8, the server must be restarted after policy set and binding attachments have been modified for a service provider or client.

__17. Log out of admin console by clicking the **Logout** link at the top right corner.

__18. Close the browser.

__19. Select the server in **Servers** view.

__20. Stop the server.

__21. Start the server.

__22. Wait for the server status to change to **Started**.

Part 4 - Test Message Integrity

__1. First make sure that the TCP monitoring proxy service is still running. To do that, right click the server and select **Monitoring > Properties**. Make sure the status is **Started**. Otherwise, start it.

__2. Click **OK** to close the monitor settings dialog.

__3. Open a browser window and enter the URL for the consumer application:

```
http://localhost:9080/ClientWeb/index.jsp
```

__4. Enter some input value. Then click **Add**.

__5. Make sure that the success message is displayed.

__6. Back in Eclipse, open the **TCP/IP Monitor** view and maximize it.

__7. Change the display mode from **Byte** to **XML** for both request and response.

__8. For the request message, observe the SOAP <Body> element of the **Request** is still in plain text. This is expected since we have not enabled encryption.

__9. In the header of the **Request**, locate the **<ds:SignatureValue>** element.

```
<ds:SignatureValue>r43pj30kuKqKy ... Mp10m994z8Rk=</ds:SignatureValue>
```

This contains the digital signature of the message.

The digest of the message is also available in the <DigestValue> element as shown below.

```
<ds:DigestValue>EVFKYcBANxvWQavJm3spuddWNrk=</ds:DigestValue>
```

But, that is rather redundant. The signature is the digest encrypted by the consumer's private key.

The consumer sends its public key in the form of its certificate. You can see that in the BinarySecurityToken element.

A BinarySecurityToken can carry any token that uniquely identifies the consumer. The ValueType attribute clearly describes the nature of the token. In our case, it is ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#**X509v3**". This value type indicates that the token is a X509 certificate.

Why does the client send its certificate? The provider uses that to decrypt the signature and obtain the digest. If encryption is enabled, the provider uses the public key from the certificate to encrypt the SOAP response message.

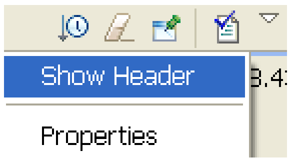
__10. Similarly, locate and observe the SignatureValue, DigestValue and BinarySecurityToken elements in the SOAP response message.

Part 5 - Test for Integrity Failure

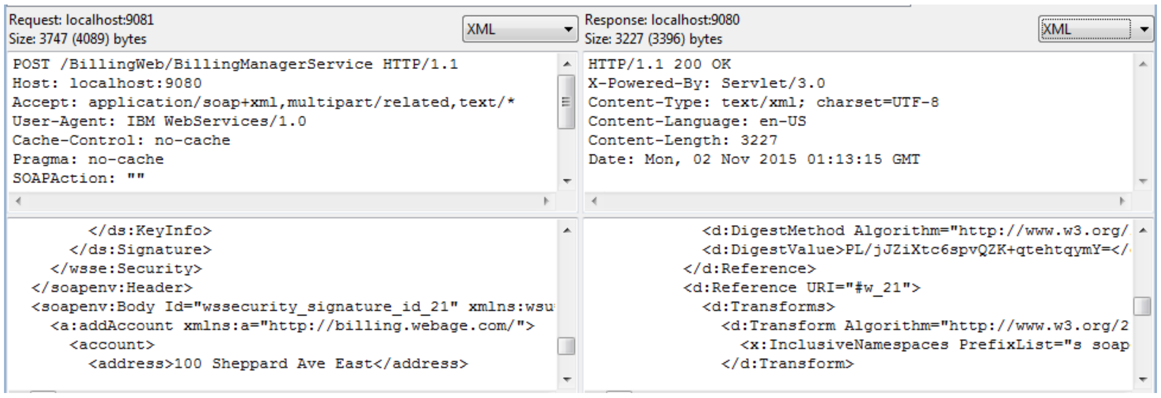
Now, we will mount a man in the middle attack. We will alter the request message and resend it. The TCP/IP Monitor view allows us to modify an existing request and resend it.

First, we need to view the HTTP header of the existing request. If we do not do that, headers will be omitted from the modified request. Which appears to be a bug in Eclipse.

__1. In the **TCP/IP Monitor** view, click the 3 dots at the top-right of the menu bar and select **Show Header**.



__2. The **TCP/IP Monitor** view now will show the header for the Request and Response.



Request: localhost:9081
Size: 3747 (4089) bytes

```
POST /BillingWeb/BillingManagerService HTTP/1.1
Host: localhost:9080
Accept: application/soap+xml,multipart/related,text/*
User-Agent: IBM WebServices/1.0
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
```

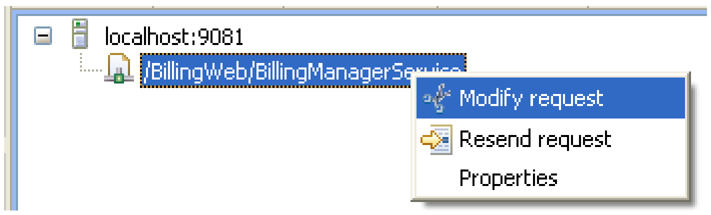
```
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body Id="wssecurity_signature_id_21" xmlns:wsu
    <a:addAccount xmlns:a="http://billing.webage.com/">
      <account>
        <address>100 Sheppard Ave East</address>
```

Response: localhost:9080
Size: 3227 (3396) bytes

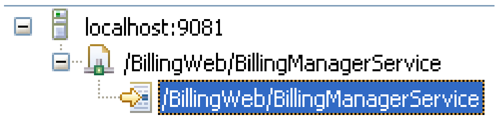
```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0
Content-Type: text/xml; charset=UTF-8
Content-Language: en-US
Content-Length: 3227
Date: Mon, 02 Nov 2015 01:13:15 GMT
```

```
            <d:DigestMethod Algorithm="http://www.w3.org/
            <d:DigestValue>PL/jJZiXtc6spvQZK+qtehtqymY=</
          </d:Reference>
          <d:Reference URI="#w_21">
            <d:Transforms>
              <d:Transform Algorithm="http://www.w3.org/2
                <x:InclusiveNamespaces PrefixList="s soap
              </d:Transform>
```

__3. In the top pane of the **TCP/IP Monitor** view, right click the request and select **Modify request**.

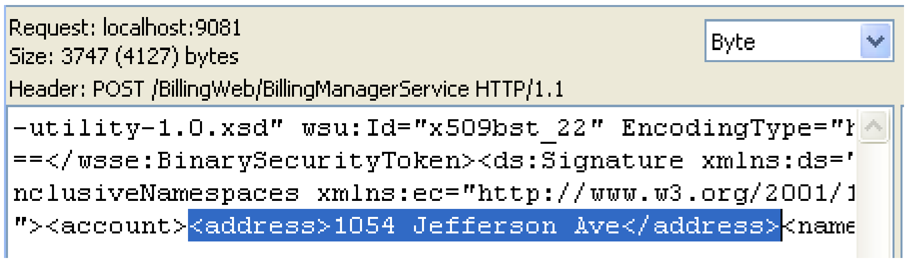


The tool will create a modified request.


localhost:9081
/BillingWeb/BillingManagerService
/BillingWeb/BillingManagerService

Note: The request can be edited in Byte mode only. If you change the display mode to, say, XML, the editor will become read only.

__4. In the **Request** editor, scroll carefully to the right and locate the <**address**> element in the bottom line.



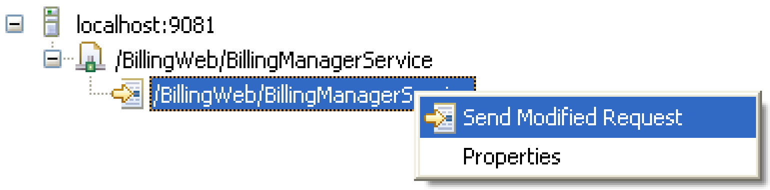
Request: localhost:9081
Size: 3747 (4127) bytes
Header: POST /BillingWeb/BillingManagerService HTTP/1.1

Byte

```
-utility-1.0.xsd" wsu:Id="x509bst_22" EncodingType="h
==</wsse:BinarySecurityToken><ds:Signature xmlns:ds="
nclusiveNamespaces xmlns:ec="http://www.w3.org/2001/1
"><account><address>1054 Jefferson Ave</address><name
```

__5. Replace some of the characters in the body of the <address> element. For example, change 1054 to 2054. Or, change an upper case letter to lower case. **Important:** Make sure that the total number of characters remain the same.

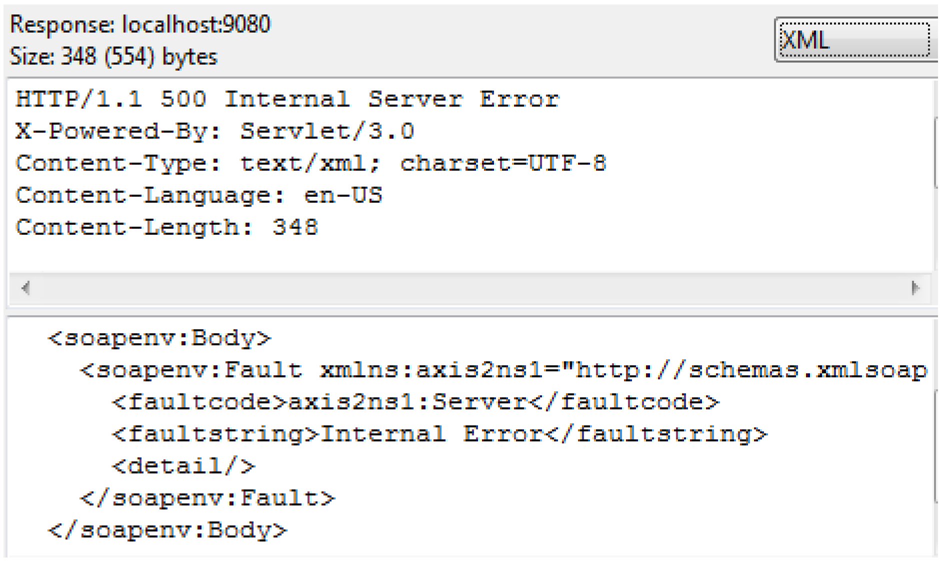
__6. Right click the modified request and select **Send Modified Request**.


localhost:9081
/BillingWeb/BillingManagerService
/BillingWeb/BillingManagerS
Send Modified Request
Properties

__7. The **Console** view will show an exception from the service.

__8. Back in the TCP/IP Monitor view, for the **Response** pane, change the display mode to **XML**.

__9. Notice that a fault has been returned by the server.



```
HTTP/1.1 500 Internal Server Error
X-Powered-By: Servlet/3.0
Content-Type: text/xml; charset=UTF-8
Content-Language: en-US
Content-Length: 348
```

```
  <soapenv:Body>
    <soapenv:Fault xmlns:axis2ns1="http://schemas.xmlsoap
      <faultcode>axis2ns1:Server</faultcode>
      <faultstring>Internal Error</faultstring>
      <detail/>
    </soapenv:Fault>
  </soapenv:Body>
```

__10. Clear all requests in TCP/IP Monitor by clicking the eraser toolbar button.

__11. Hide the Header since you don't need it anymore, remember that you opened the header to modify a request.

__12. Close any open web browser.

Part 6 - Review

Message integrity is used to implement the non-repudiation feature. That is a legal term which means, a party sending a message can not deny that it had sent that message. In addition, it can not claim that a third party had altered the message.

Enabling message integrity is the first step in setting up non-repudiation. The provider will validate the request and the consumer will validate the response. Any man in the middle attack will be immediately detected. However, to resolve any disputes between business at a later date, you need to also save the message, its signature and sender's certificate in an audit log. In case of a dispute, the message should be validated again.

In this lab, we enabled message integrity. We also attempted a man in the middle attack which was successfully thwarted.

We also learned a lot of policy set and policy binding. They allow us to change the client server communication without changing any code.

Lab 16 - Message Confidentiality

In this lab, we will enable message encryption for the BillingManagerService. Message integrity is already enabled. We will add confidentiality on top of that.

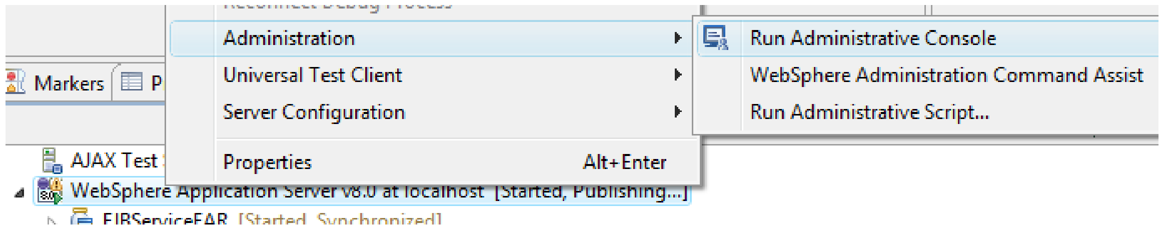
Part 1 - Import Policy Set

A policy set containing confidentiality and integrity has been already designed for you. You will import it and then attach it to the provider and consumer.

__1. In Eclipse, locate the **Servers** view.

__2. Start the server if it is not running.

__3. Right click the server in the **Servers** view and select **Administration** → **Run administrative console**.


Administration
Universal Test Client
Server Configuration
Properties
Alt+Enter
Run Administrative Console
WebSphere Administration Command Assist
Run Administrative Script...
Markers
AJAX Test
WebSphere Application Server v8.0 at localhost [Started, Publishing...]
FIBServiceEAR [Started, Synchronized]

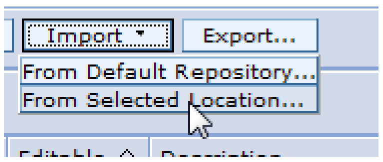
__4. Login as wasadmin if prompt.

The admin console will open.

__5. In the left pane, expand **Services > Policy Sets**.

__6. Click **Application policy sets**.

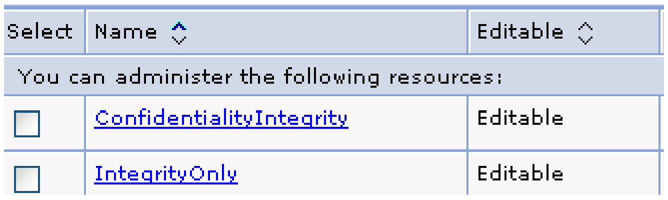
__7. Click **Import** and select **From Selected Location**.



__8. Browse, select **C:\LabFiles\ConfidentialityIntegrity.zip** and click Open.

__9. Click **OK**.

__10. Make sure that the policy set called ConfidentialityIntegrity has been imported.



Select	Name	Editable
You can administer the following resources:		
☐	ConfidentialityIntegrity	Editable
☐	IntegrityOnly	Editable

__11. Click **Save** at the top of the page to save changes.

Part 2 - Review the Confidentiality Policy

We will now inspect the policy.

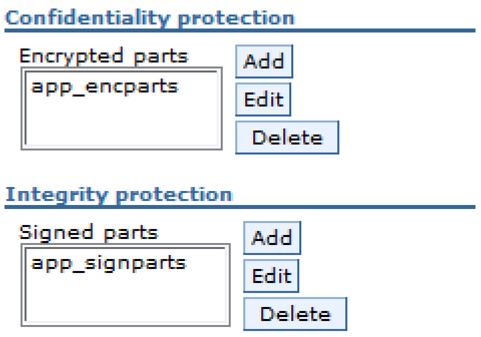
__1. Click **ConfidentialityIntegrity**.

__2. Click **WS-Security**.

__3. Click **Main policy**.

__4. Click **Request message part protection**.

__5. Note that parts of the request message are protected for both encryption and integrity.



Confidentiality protection

Encrypted parts: app_encparts

Integrity protection

Signed parts: app_signparts

Part 3 - Attach Policy Set

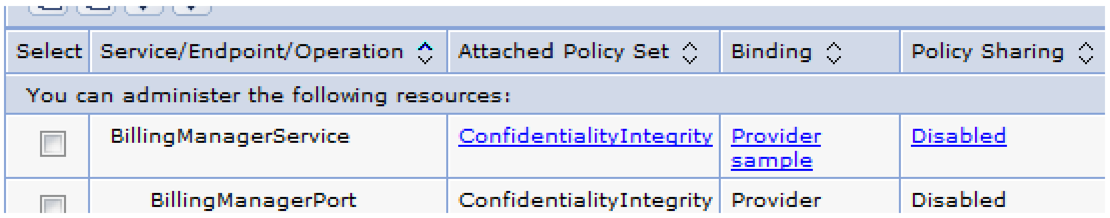
We will now attach the new policy set to our provider and consumer. The steps are same as before. If you need, refer back to the detailed steps in page 168.

__1. On the left hand side click **Service providers**.

__2. Click **BillingManagerService**.

__3. Attach the **ConfidentialityIntegrity** policy set to the **BillingManagerService**.

__4. Assign the **Provider sample** binding to the service.



Select	Service/Endpoint/Operation	Attached Policy Set	Binding	Policy Sharing
You can administer the following resources:				
☐	BillingManagerService	ConfidentialityIntegrity	Provider sample	Disabled
☐	BillingManagerPort	ConfidentialityIntegrity	Provider	Disabled

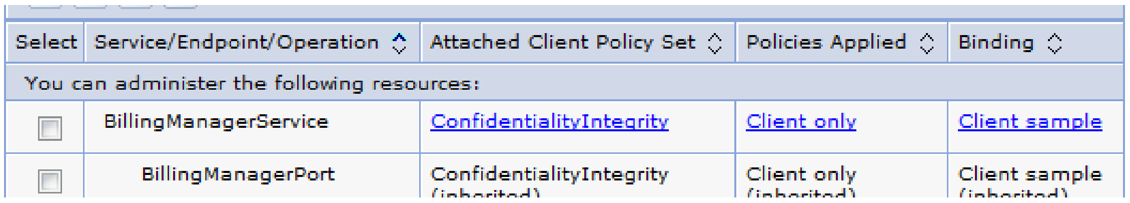
__5. Save changes.

__6. On the left hand side click **Service clients**.

__7. Click **BillingManagerService**.

__8. Attach the **ConfidentialityIntegrity** policy set to the **BillingManagerService**.

__9. Assign the **Client sample** binding to the consumer.



Select	Service/Endpoint/Operation	Attached Client Policy Set	Policies Applied	Binding
You can administer the following resources:				
☐	BillingManagerService	ConfidentialityIntegrity	Client only	Client sample
☐	BillingManagerPort	ConfidentialityIntegrity (inherited)	Client only (inherited)	Client sample (inherited)

__10. Save changes.

__11. Log out of admin console and close the browser.

__12. Stop the server. If takes too long to stop it it may show you a message to Terminate it.

__13. Start the server. Wait for the server to start.

Part 4 - Test for Confidentiality

__1. Open a browser window and enter the URL for the consumer application:

```
http://localhost:9080/ClientWeb/index.jsp
```

__2. Enter some input value. Then click **Add**.

__3. Make sure that the success message is displayed.

__4. Back in Eclipse, review that the **Console** shows the output from the service provider.

__5. Open the **TCP/IP Monitor** view and maximize.

__6. Change the display mode from **Byte** to **XML** for both request and response.

__7. For the request message, observe the following.

The <Body> element contains only encrypted data.

```
   <soapenv:Body ...>
     ...
     <enc:EncryptedData ...>
       <enc:EncryptionMethod .../>
       <enc:CipherData>

<enc:CipherValue>V1Oz7mr9UgEnMQuUgkklwpmlEGQTSjckMk0ZuOHXaclE8X+ZOZf8IZ
dXN4MUU+wYWYgzsYAi7+AhnZXxOy8OnixrWt5GUJX3NhW8LahAOe7cUrovl1Lm3D8Sff5rH
nZnY0F0mfyD7xkFFykQma8vwgrQ8qRbTn9Lg29NPx4HyemYIIPIx4Ksp40yeJFXTeXVZ+E5
PS+5beL0IEf/Nrjwz5T1SFtDbPHe45PtBOkjGmE=</enc:CipherValue>
       </enc:CipherData>
     </enc:EncryptedData>
     ...
   </soapenv:Body>
```

__8. Similarly, verify that the Body of the response message is also encrypted.

Note: By the time the provider implementation class is invoked, the JAX-WS runtime has already decrypted the message. As a result, the log output in the **Console** view shows the plain text values of the data input in the web browser.

__9. Clear all requests and restore the size in the TCP/IP Monitor view.

__10. Close the web browser.

Part 5 - Review

In this lab, we enabled message encryption.

Lab 17 - Develop a Simple RESTful Service

Over the next few labs, our goal will be to learn the fundamentals of the JAX-RS API.

In this lab, we will develop a very simple RESTful web service. This will let us focus on the development process using Eclipse and the WebSphere server. We will also learn about the basic JAX-RS annotations.

The main goal of this lab is to understand how to create a web service project and develop a few basic REST services.

Part 1 - Create the Web Service Project

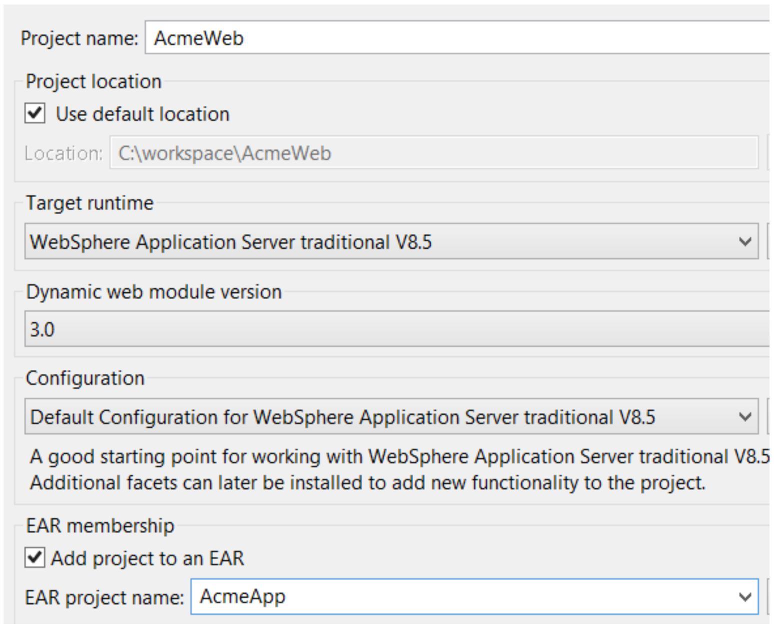
In Eclipse, the project to use to develop JAX-RS services is just a regular "Dynamic Web" project. When you target a WebSphere 8.5 server, the JAR files for the JAX-RS API are already on the classpath.

__1. From the menubar, select **File** → **New** → **Dynamic Web Project**.

__2. Enter **AcmeWeb** for the *Project name*.

__3. At the bottom of the window, in the *EAR Membership* section, make sure **Add project to an EAR** is selected. (We will want to add this WAR file into an EAR file)

__4. Set the *Project name* to **AcmeApp**



Project name: AcmeWeb

Project location

☑ Use default location

Location: C:\workspace\AcmeWeb

Target runtime

WebSphere Application Server traditional V8.5

Dynamic web module version

3.0

Configuration

Default Configuration for WebSphere Application Server traditional V8.5

A good starting point for working with WebSphere Application Server traditional V8.5
Additional facets can later be installed to add new functionality to the project.

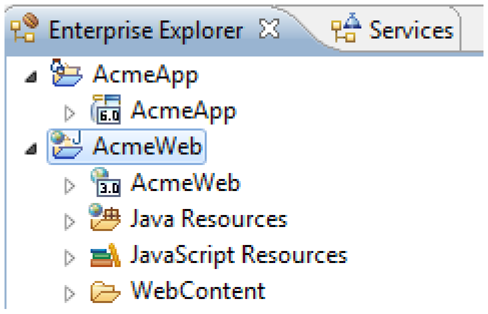
EAR membership

☑ Add project to an EAR

EAR project name: AcmeApp

__5. Click **Finish**. Both the **EAR** and the **WAR** will be created. If prompted, don't switch to the web perspective by clicking **No**.

Eclipse will displayed them in the *Enterprise Explorer*.

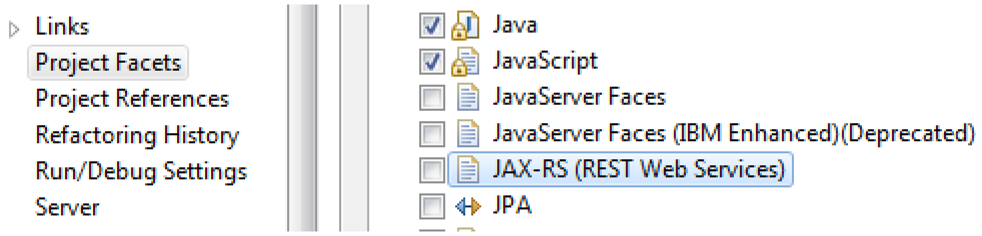


We will briefly review the project.

__6. Right click the newly created **AcmeWeb** project and select **Properties**.

__7. Now, select the **Project Facets** property.

__8. Note that the JAX-RS facet is not selected. This facet plays no role in developing JAX-RS services for WebSphere 8.5.



__9. Click **Cancel** to close the dialog.

Part 2 - Register the REST Application

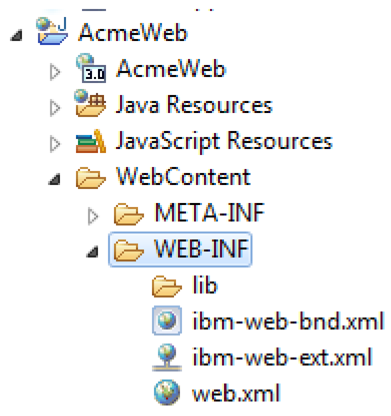
There are a few different ways to configure a REST application. This can depend on if you want a common URL prefix for all REST services. This might be useful if you have REST services within a project with other web components. If you want to register a common URL prefix you can do so in a web.xml file. To save time, this file is given to you. You will simply import it.

__1. Open Windows file explorer.

__2. Go to **C:\LabFiles\**

__3. Copy **web.xml**

__4. In Eclipse, paste the file inside the **WebContent > WEB-INF** folder of **AcmeWeb**.



__5. Open **web.xml** and study how the REST application is configured. Switch to the Source view and specifically, note the servlet mapping:

```
<servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/svc/*</url-pattern>
</servlet-mapping>
```

This means, the URL for every REST request will start with http://host:port/AcmeWeb/svc/. You can choose any other path for the REST application. But, we will stick to the short and sweet "svc".

__6. Close the file.

Part 3 - Create the Resource Class

The Java class that implements a RESTful service is called a resource. We will now develop a Java class for a simple service.

__1. Right click **AcmeWeb** project and select **New > Class**.

__2. Enter **com.acme.services** as the package name and **SimpleService** as the class name.

__3. Click **Finish** to create the new class.

__4. Add a member variable that will help us do logging.

```
public class SimpleService {
          Logger logger = Logger.getLogger("SimpleService");
```

__5. Organize imports (Control+Shift+O) and select **java.util.logging.Logger**. Make sure you select the correct class as there are several 'Logger' classes on the classpath.

We will now add the testGET() method. It will not have any business logic. Later, we will map this method to a GET request.

__6. Add the method as follows.

```
public String testGET() {
          logger.info("Got a GET request");

          return "OK";
}
```

__7. Save changes.

Part 4 - Configure the Resource

We will now configure the URI and HTTP method for the service resource and its methods. We will use the "/simple" root URI for the service.

__1. Define the URI of the root resource, by adding the @Path annotation above the class.

```
@Path("/simple")
public class SimpleService {
```

The testGET() method does not need any path extension. All we have to do is set GET as the HTTP method.

We will also set "text/plain" as the content MIME type of the reply. That is good enough for this method. For XML data type, the MIME will be "text/xml".

__2. Set the HTTP method and reply MIME type for the testGET() method sub-resource as follows.

```
@GET
@Produces("text/plain")
public String testGET() {
```

__3. Organize imports. Select **javax.ws.rs.Produces**.

__4. Save changes.

Part 5 - Unit Test

We will now exercise the web service from a browser.

__1. Start the server.

__2. Right click the server and select **Add and Remove**.

__3. Add the **AcmeApp** project to the server.

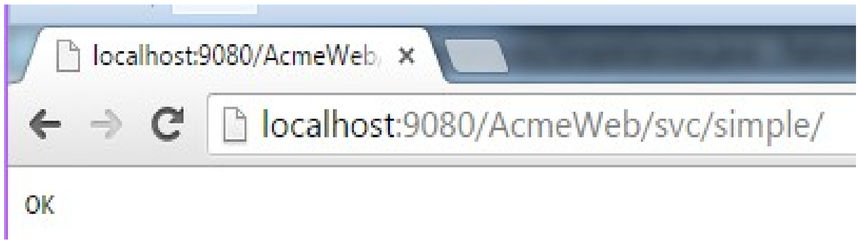
__4. Click **Finish**.

__5. Right click the server and select **Publish**. Do this every time you are requested to Publish the server.

__6. Open a web browser and enter the URL:

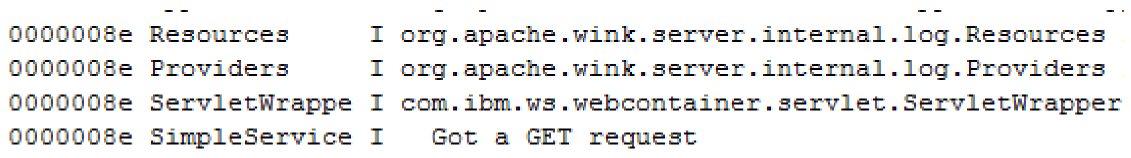
```
http://localhost:9080/AcmeWeb/svc/simple/
```

__7. You should see OK in the browser.



OK

__8. The Console view in Eclipse will show the log output.



```
0000008e Resources     I org.apache.wink.server.internal.log.Resources
0000008e Providers     I org.apache.wink.server.internal.log.Providers
0000008e ServletWrappe I com.ibm.ws.webcontainer.servlet.ServletWrapper
0000008e SimpleService I   Got a GET request
```

This proves that our project has been setup correctly and JAX-RS is working fine.

Part 6 - Use Path Extension for Sub-resource

A sub-resource – Java method – can be mapped to an URI extension path. Any HTTP request for that path will be handled by that method. We will now create a method that will respond to the /simple/**mypath** URI.

__1. First, add this method to the SimpleService class.

```
public String testGETWithPath() {
          logger.info("Got a GET request with path extension.");

          return "OK";
}
```

__2. Annotate the method as follows.

```
@GET
@Produces("text/plain")
@Path("/mypath")
public String testGETWithPath() {
```

__3. Save changes.

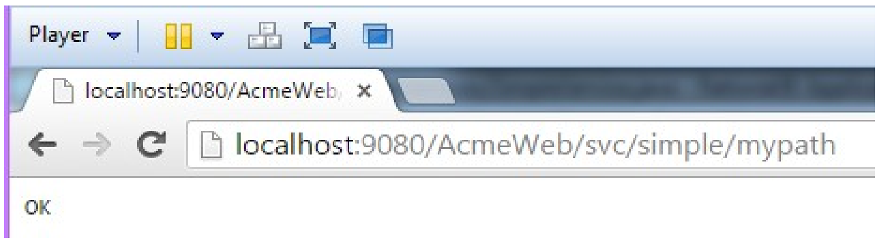
Part 7 - Unit Test

__1. Right click the server and select **Publish**. Wait until publishing is done.

__2. In a web browser, enter the URL:

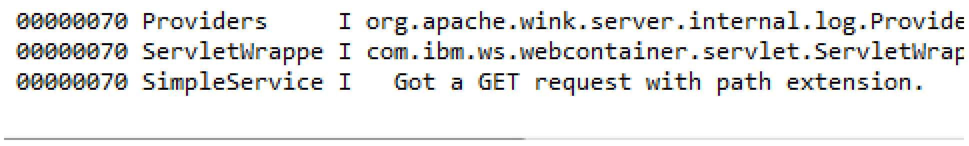
```
http://localhost:9080/AcmeWeb/svc/simple/mypath
```

__3. You should see OK in the browser.



```
OK
```

__4. Make sure that the Console shows the log output.



```
00000070 Providers     I org.apache.wink.server.internal.log.Provide
00000070 ServletWrappe I com.ibm.ws.webcontainer.servlet.ServletWrap
00000070 SimpleService I   Got a GET request with path extension.
```

__5. Open a new browser and re-test the testGET method for regression using the URL:

```
http://localhost:9080/AcmeWeb/svc/simple/
```

__6. Close all open files.

__7. Close all browsers.

Part 8 - Review

In this lab, we created and configured a web service project. We created a very simple JAX-RS web service. At this point, you should know how to configure the URI of the root resource – the Java class - using the @Path annotation. Also, we configured the HTTP method of a sub-resource – a method – using the @GET annotation.

Lab 18 - Extracting Information from a HTTP Request

RESTful services are executed by sending HTTP requests. The request contains input data in various areas:

1. In the URI path. For example: /orders/**1051**.
2. As URL parameters. For example: /orders?**status=P**
3. Input data from a form submission.
4. Less commonly, from HTTP header and cookie.

We will now learn how to extract data from common locations.

Part 1 - Get Root Resource Path Parameters

Input data can be added to the path of the root resource as well as the path of a method (sub-resource). In REST, data is added to the URI to form an unique identifier.

First, we will add an input parameter in the root path of the SimpleService resource. The root URI will now look like /simple/**somedata**. The root URI will continue to execute the testGET() method since this method defines no path extension. To execute the testGETWithPath(), the URI will need to be /simple/**somedata**/mypath.

__1. Open **SimpleService.java** from the **AcmeWeb** project.

__2. Change the @Path annotation for the class and add a parameter there.

```
@Path("/simple/{myRootPathData}")
public class SimpleService {
```

Here, {myRootPathData} is a placeholder for a parameter. You can use anything as a name of the parameter. The name will play a role to obtain the value of the parameter.

The best place to capture input from a root path is a member variable of the resource. We will do that now.

__3. Add a member variable as follows.

```
public class SimpleService {
          String rootPathData;
```

__4. Annotate the member variable to save the value of the myRootPathData parameter.

```
@PathParam("myRootPathData")
String rootPathData;
```

That's it. Now, JAX-RS will extract the value of the parameter from the URI and set it to the member variable right after the resource object is created. By default, a POJO resource instance is created for every HTTP request. Hence, every request can have a different parameter value in the path.

__5. Organize imports.

__6. Change the log statement of the testGET() method as follows.

```
logger.info("Got a GET request with root path data: " + rootPathData);
```

__7. Make a similar change to the testGETWithPath() method.

```
logger.info(
    "Got a GET request with path extension with root path data: " +
    rootPathData);
```

__8. Save changes.

Part 2 - Unit Test

__1. Publish the server.

__2. First, test the testGET() method by entering the URL:

```
http://localhost:9080/AcmeWeb/svc/simple/somedata/
```

__3. The log output will like this:

```
Got a GET request with root path data: somedata
```

__4. Now, test the testGETWithPath() method using the URL:

```
http://localhost:9080/AcmeWeb/svc/simple/somedata/mypath
```

__5. The log output will be:

```
Got a GET request with path extension with root path data: somedata
```

Part 3 - Get Sub-resource Path Parameters

Methods can also let us add parameters to its path. For example to get the billing address of an order #1051, we can use the URI: /orders/**1051**/address/**billing**. Never lose sight of the fact that a URI uniquely identifies an entity or a collection of entities. In this example, we are specifically pointing to the billing address used with an order. Here, the orderId as well as the type of address requested can be added as parameters to the path of the sub-resource. Let's try out this example.

__1. Add a basic method as follows.

```
public String getAddress(
          int oId,
          String type) {
          logger.info("Order Id: " + oId);
          logger.info("Address type: " + type);

          return "OK";
}
```

__2. Annotate method with path and method.

```
@GET
@Produces("text/plain")
@Path("/{orderId}/address/{addressType}")
public String getAddress(
```

__3. Now, extract the path parameters and save them in the two input argument variables of the method. The syntax for the @PathParam annotation on a method parameter is tricky so be careful.

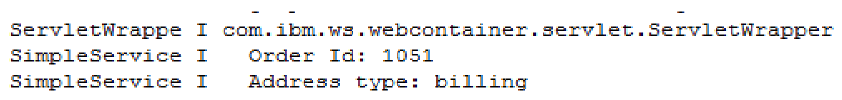
```
public String getAddress(
          @PathParam("orderId")
          int oId,
          @PathParam("addressType")
          String type) {
```

__4. Save changes.

__5. Publish the server.

__6. Test the method by entering the URL:

```
http://localhost:9080/AcmeWeb/svc/simple/somedata/1051/address/billing
```



```
ServletWrappe I com.ibm.ws.webcontainer.servlet.ServletWrapper
SimpleService I   Order Id: 1051
SimpleService I   Address type: billing
```

Part 4 - Extract Query Parameters

Query parameters are available from the requested URI as:

```
GET /path?param1=value&param2=value2 HTTP/1.1
```

You can capture them either using resource class member variable or method argument variable. The latter is generally recommended for better code readability. We will try that out now.

__1. Add the giveRaise() method as shown below.

```
@GET
@Produces("text/plain")
@Path("raise")
public String giveRaise(
    @QueryParam("name")
    String employeeName,
    @QueryParam("amount")
    double amount
    ) {
    logger.info("Giving raise to " + employeeName + " by " + amount);

    return "OK";
}
```

__2. Organize imports.

__3. Save changes.

__4. Publish the server.

__5. Test the change by entering the URL:

```
http://localhost:9080/AcmeWeb/svc/simple/somedata/raise?name=Daffy&amount=10000
```

__6. Make sure that the Console shows:

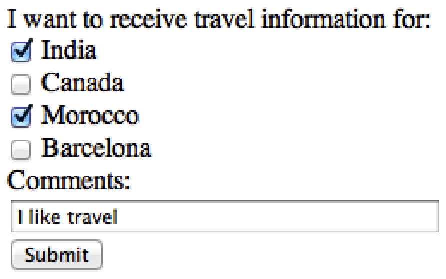
```
Giving raise to Daffy by 10000.0
```

Note, how JAX-RS converted the amount URL parameter from text to double.

Part 5 - Extracting Form Post Data

Form data is available in the HTTP request body. The data is encoded using MIME type application/x-www-form-urlencoded. Usually, such data is submitted using the POST method.

Note: The Java EE Servlet specification does not distinguish between URL query parameter and POST data in HTTP request body. They are both accessed using request.getParameter() method. JAX-RS, however, makes a distinction. Query parameter can be accessed using @QueryParam. Form POST data has to be accessed using @FormParam.


I want to receive travel information for:
✓ India
☐ Canada
✓ Morocco
☐ Barcelona
Comments:
I like travel

We will now build a service method that will accept input from the form above.

__1. Add this method.

```
@POST
@Produces("text/plain")
@Path("feedback")
public String submitFeed(
        @FormParam("interest")
        List<String> interestList,
        @FormParam("comments")
        String comments
    ) {
    for (String interest : interestList) {
        logger.info("Interest: " + interest);
    }
    logger.info("Comments: " + comments);

    return "OK";
}
```

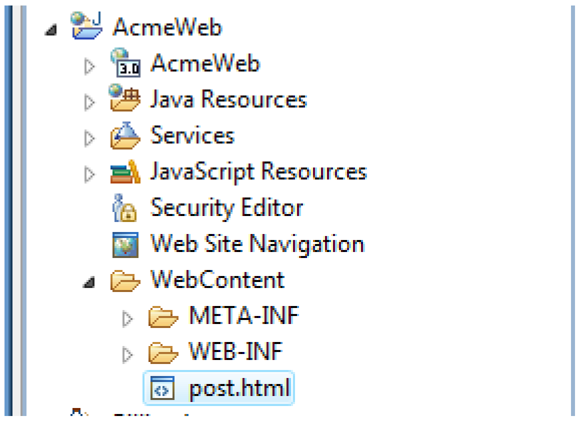
Note that the HTTP method is set to POST. Also, we expect multiple values for the "interest" parameter. That is why we have set the data type of the variable to java.util.List<String>.

__2. Organize imports. Select **java.util.List**.

__3. Save changes.

Part 6 - Unit Test

__1. We will import the form HTML page. Copy **C:\LabFiles\post.html** and paste it inside the **WebContent** folder of the **AcmeWeb** project.

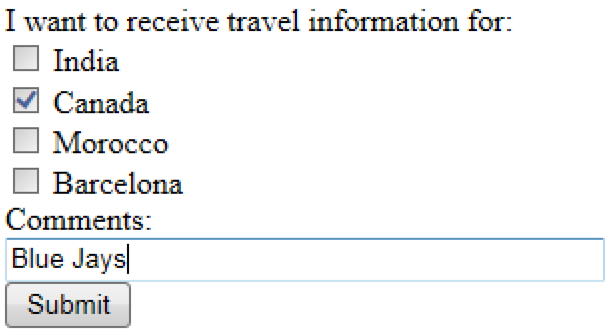


__2. In the *Servers* view, right click the server and select **Publish**.

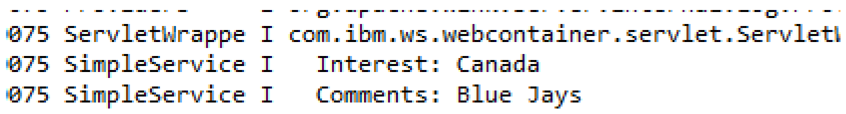
__3. Open a browser and enter the URL:

`http://localhost:9080/AcmeWeb/post.html`

__4. Fill out the form and submit it.



__5. Make sure that the console log shows the input values correctly.



```
075 ServletWrappe I com.ibm.ws.webcontainer.servlet.Servlet
075 SimpleService I   Interest: Canada
075 SimpleService I   Comments: Blue Jays
```

__6. Close all open files.

__7. Close all open browsers.

Part 7 - Review

In this lab, we learned how to gather input data from a couple of common sources:

1. URI path of the root resource
2. URI path of the method sub-resource.
3. URL query parameter.
4. Form POST data.

Lab 19 - Designing a RESTful Service

Before we get into actual coding of RESTful services, we need to learn about how to design or architect one. This skill will be independent of the programming model and will apply equally if you choose to use PHP, ASP.NET or JAX-RS as the programming model.

We will begin by gathering business requirements. We will then analyze the requirements and then take various architectural decisions.

You will only need a pen and a notepad for this lab.

Part 1 - The Business Requirements

Acme Inc. is a machine parts manufacturing company. It manufactures components that other companies buy to build machines. Most of the parts are standard and are purchased by many companies. Acme also offers custom parts manufacturing services. A customer can submit the design for a part which Acme manufactures just for that client.

Currently, Acme offers a web site where companies can place orders and check order status. This has worked well for a few years. Now, customers are increasingly asking for a Web Service interface. This will help the clients integrate parts ordering with their own internal software systems and allow them to place orders without human intervention.

Short Theory of Requirements

Usually, requirements are first written using brief unstructured statements. They are captured as feature requests, with unique number and status. In this manner, features are very similar to defects. Features that are within the scope of a project are approved and then further elaborated using use cases. Use cases also have unique identification numbers. Use cases follow a more structured format and are immensely helpful in any software design.

To keep things simple in this lab, we will just write down the basic requirements. In real life, you are encouraged to develop use cases before proceeding with the subsequent stages of service design.

R1. Get price quote and availability – A client organization should be able to get a quote for a part and a confirmation if it can be available within a given date. The quote includes unit price and total applicable price taking into account any discounts.

R2. Place an order for standard parts – A client organization should be able to place an order. An order is a list of part numbers and quantities. After the order is placed successfully, an identifier for the order is returned.

R3. Get order details – Client should be able to get the status of an order by the order ID. The information includes the total cost of the order. If the order is pending shipment then an expected delivery date is also returned.

R4. Get order history – Client should able to get a list of orders by order status.

R5. Cancel an order – Client should able to cancel an order.

R6. Place order for a custom part – Client uploads a CAD (computer aided design) document that captures the design of the part.

Part 2 - Define Project Scope

In Phase I of the project, requirements R1-R5 will be addressed. Ordering of custom parts (R6) will be addressed in Phase II.

Part 3 - Designing the Object Model

This is the first stage in designing a RESTful web service. At this stage, we define the structure of the data that is exchanged between the web service and its consumers. The process is very similar to class design in Object Oriented Programming.

First step in object model definition is the identification of entities. Entities are nouns in business requirements.

__1. Can you go through the business requirements and identify the entities?

____

____

____

____

____

____

__2. Now that you have identified the entities, can you list the attributes or fields for each entity? Once again, the answer is below. But do your best to do the design yourself.

Entity types:

- Price quote request
- Price quote response
- Order
- Order line item
- Order status

Entity attributes:

Price quote request

- Part number
- Quantity
- Client organization ID
- Date the part needs to be available by

Price quote response

- Part number
- Quantity
- Client organization ID
- Date the part needs to be available by
- Unit price
- Total price
- If order can be shipped within the required date.

Order line item

- Part ID
- Quantity requested

Order

- Order ID
- Client organization ID.
- Order total
- Status
- Expected shipment date
- List of order line items

Part 4 - Identify the Service

Service identification is an important stage in Service Oriented Architecture (SOA). At this point we study the requirements and look for services that need to be implemented.

First, we look for the verbs in the requirements. They signify the actions or operations that the services need to be performed. Then, we group the logically related operations into services.

__1. Can you go through the requirements and list the tasks or operations that various yet to be identified services need to perform? Then compare your findings with the answer below.

____

____

____

____

____

____

Note: In our case, each requirement (or use case) yields one operation. In more complex cases, there may be multiple operations identified from a use case.

Next, we need to group the operations in a logical grouping called a service. In REST, they key factor that affects grouping is entity. That means, operations that work on any given entity (like Order, Customer), should be grouped in a single service.

__2. Can you group the operations based on the entity they work with? Name each group as a service accordingly. The answer is below.

Operations:

- Get price quote and availability (R1)
- Place an order for standard parts (R2)
- Get order details (R3)
- Get order history for a client organization (R4)
- Cancel an order (R5)

Operation groupings:

Quote Service:

- Get price quote and availability (R1)

Order Service:

- Place an order for standard parts (R2)
- Get order details (R3)
- Get order history for a client organization (R4)
- Cancel an order (R5)

Part 5 - Design the Service Interface

For SOAP based services, designing service interface involves writing the WSDL document and the associated XML schema files. For REST, the process involves assigning a URI structure and HTTP method for each identified operation.

In RESTful web service, a URI uniquely points to an entity. The HTTP method determines what action needs to be performed on that entity. For example, each order is identified by the /orders/{orderId} URI. Hence, to get the status of an order, we do a GET on that URI. To place an order, we will do a POST on the /orders URI with the server being responsible for generating the (unique) value for orderId (that later is referenced in the GET request).

Each operation identified from the business requirement needs to be mapped to a URI and HTTP method pair.

__1. First, let us decide on the URI for each entity. This is a pretty straightforward exercise. **Note:** As a matter convention, plural forms of the entity names are used in URI.

1. Price quote - /quotes.
2. Order - /orders.

There is no need for URIs for the other entities since we do not directly perform any action on them.

These URIs will also form the root URI for the identified services. Individual operations within a service may add extra path elements. For example, to get the status of order, the URI will be /orders/{orderId}. Some operations may need to add URL parameters. This is discussed in more details later.

Now, we are ready to design the service interfaces by assigning a URI/HTTP method pair to each operation. Since you might be new to this, the work has been already done for you. Go through the design below and try to understand the rationale for various decisions taken.

Service: Quote Service
Root URI: /quotes

Operation	URI extension	HTTP Method	Remarks
Get price quote and availability (R1)		POST	Normally for such a read-only request, we should use GET. But, we need to send the quote request XML document to the service. This will be difficult to do for a GET request. That is why we choose POST.
Place an order for standard parts (R2)		POST	No resource id is supplied with the client request; the task of id generation for the new resource is delegated to the server.
Get order details (R3)	/{orderId} e.g.: /orders/1051	GET	The order ID is now added to the root URI to point to a specific order.
Get order history for a client organization (R4)	/clients/{clientId} e.g.: /orders/clients/18	GET	The path /clients/{clientId} is now added to the root URI to point to a specific client organization that we want to retrieve the order history for.
Cancel an order (R5)	/{orderId}	DELETE	We choose to use the DELETE method to signify cancellation. This is bit of a tricky decision because the entity is not actually deleted.

Design Tip

According to REST methodology, the URI syntax is heavily entity centric, in the sense that a URI points to an entity or a set of entities that match certain criteria. Actual operations performed on the entity are not reflected in the URI. The HTTP methods go a long way to define the actions. But, even that may not be enough.

Consider the situation if we had two separate requirements to cancel and delete an order. In that case, the DELETE method will apply to the deletion case. How will we model cancellation? In situations like this, where HTTP method is not sufficient to model an operation, you need to resort to URL parameters. For example, /orders/{orderId}?action=cancel can be used.

In summary, HTTP methods are sufficient to model CRUD (create, retrieve, update and delete) operations. Anything beyond that, use URL parameters.

Part 6 - Decide on the Data Format

The two prevalent formats in the REST world are JSON and XML. XML is better for applications written in Java, .NET and PHP since they all come with XML parsers. JSON is the most convenient format to work with from JavaScript. JavaScript includes the DOM API. So, in the worst case, you can consume XML from JavaScript.

As stated in our business requirements, the web service will be consumed mainly by software applications of the client organizations. This will steer our decision to XML. In the future, if the need arises, we can support JSON.

Part 7 - Unaddressed Architectural Issues

Our web service has to be secured. Otherwise, anyone can place an order or check the status. To keep things simple at this stage, we will not deal with the security requirements. We will get to that in a later lab.

We do not deal with hypermedia controls either. We will learn about hypermedia later.

Part 8 - Review

Designing a RESTful web service requires a series of steps. We saw some of these steps in play here:

1. Design the object model.
2. Identify services
3. Design the service interface (URI structure and HTTP method for the operations).
4. Decide on the data format (XML or JSON).

Lab 20 - Add Support for XML Format

We know enough about JAX-RS at this stage to start implementing the Acme Inc. web services. First, we will create the entity classes that model items like price quote request, response, order etc. Next, to work with XML documents in the HTTP request and response, we will need to annotate the entity classes with JAXB annotations.

Part 1 - Create the Entity Classes

We had designed the object model for the Acme web services in a previous lab. We will now create Java classes corresponding to that.

__1. In the **AcmeWeb** project, create a package called **com.acme.services.entities**

__2. Within that package, create a Java class called **QuoteRequest**

__3. In the **QuoteRequest** class, add these member variables. Feel free to refer back to the object model to make sure we are doing this correctly.

```
public class QuoteRequest {
          String partNumber;
          int quantity;
          String clientId;
          String dateRequiredBy;
}
```

__4. Generate getters and setters for the variables. To do this select any variable, then from the menu select **Source > Generate Getters and Setters**. Click **Select All** and click **Generate**.

__5. Add the @XmlRootElement JAXB annotation to the class as shown in boldface below.

```
@XmlRootElement(name="quote_request")
public class QuoteRequest {
```

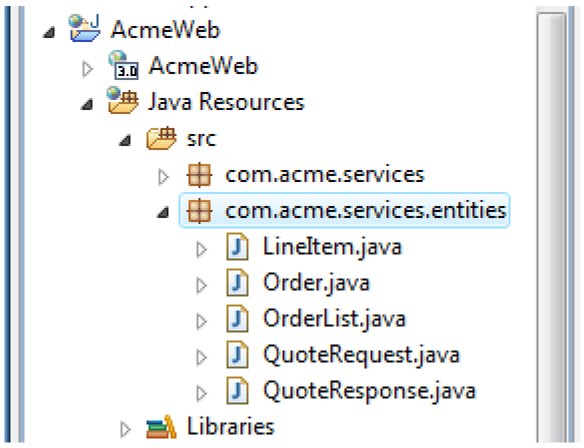
__6. Organize imports.

__7. Save changes.

The other entity classes can be created the same way. To save time, these classes have been created for you. You will import them and review them.

__8. Open a file browser and go to **C:\LabFiles\entities**.

__9. Copy all the files within the folder and then paste them into the **com.acme.services.entities** package.



__10. Open each entity class file and review them. Specifically, look at the Order class. Working with list of child objects can be tricky in JAXB. Look at how we are dealing with line items:

```
@XmlElement(name="item")
public List<LineItem> getLineItemList() {
          return lineItemList;
}
```

For each LineItem object in lineItemList, JAXB will create a child element within <order>. By default, the child element's name will be <lineItemList>. That is not good. To fix that, we set the element name to "item". Now an Order object will look like this in XML:

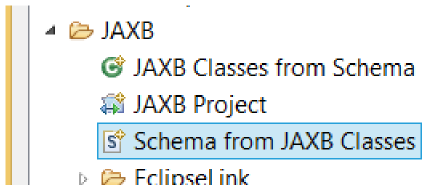
```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <item>
        <partNumber>P01010</partNumber>
        <quantity>10</quantity>
    </item>
    <item>
        <partNumber>P18210</partNumber>
        <quantity>5</quantity>
    </item>
    <clientId>C001</clientId>
    <orderId>0</orderId>
    <orderTotal>0.0</orderTotal>
</order>
```

Part 2 - Generate the XML Schema

XML schema generation is strictly not necessary for REST. (It is, however, required for SOAP based services). We will now generate the schema anyway. This will help us better understand the structure of the XML documents used in HTTP requests and responses. Basically, the schema serves as a great documentation for anyone developing a client.

__1. In the *Enterprise Explorer* view, right click on the **com.acme.services.entities** package and select **New** → **Other**

__2. Select **JAXB** → **Schema from JAXB Classes** and click the **Next** button.

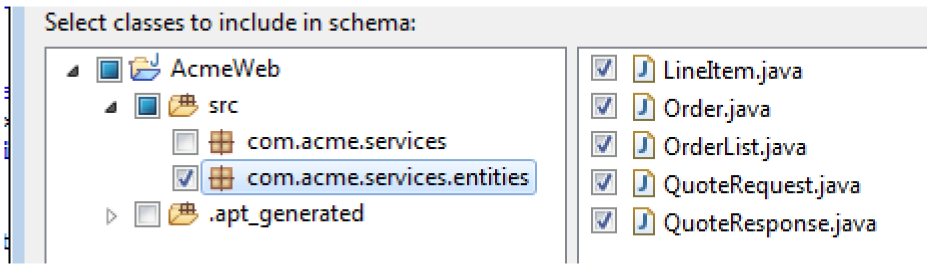


__3. Expand the **AcmeWeb** → **src** folder and select the **com.acme.services.entities** package to make sure all Java files in the package are selected to be included.

__4. Enter **AcmeWeb.xsd** as File name.

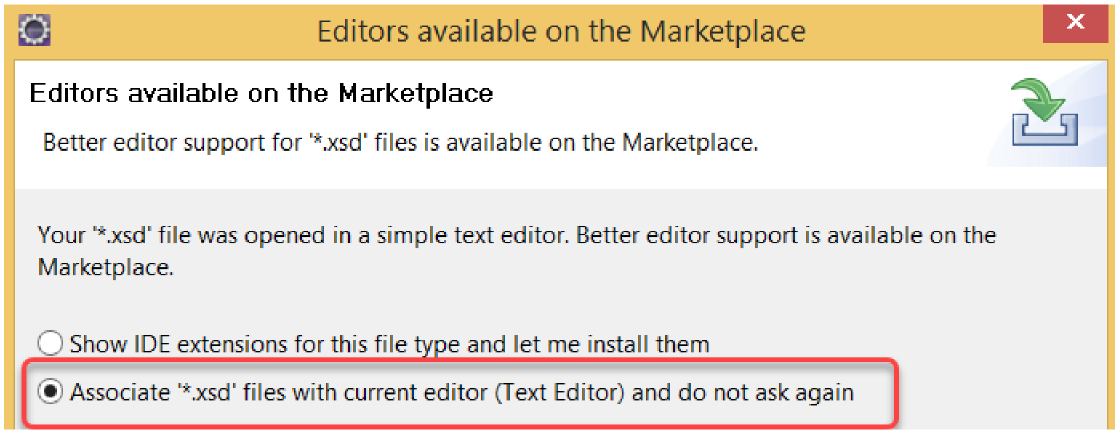
__5. Click the **Next**.

__6. Expand **AcmeWeb** → **src** and check the box for **com.acme.services.entities**. Make sure all classes are selected.


Select classes to include in schema:

__7. Click the **Finish** button to generate the schema.

__8. If you see the below, select the second option and click OK.



Editors available on the Marketplace

Better editor support for '*.xsd' files is available on the Marketplace.

Your '*.xsd' file was opened in a simple text editor. Better editor support is available on the Marketplace.

( ) Show IDE extensions for this file type and let me install them

(●) Associate '*.xsd' files with current editor (Text Editor) and do not ask again

__9. After a few seconds check that the project has the schema, the schema name may change to AcmeWeb1.xsd

__10. Open the generated schema file and study it if you wish.

Part 3 - Create the Quote Service Class

__1. In the **com.acme.services** package, create a class called **QuoteService**

__2. Set the root URI of the service as shown in boldface below.

```
@Path("/quotes")
public class QuoteService {
```

__3. Add a member variable for logging.

```
public class QuoteService {
          Logger logger = Logger.getLogger("QuoteService");
```

__4. Add the getQuote() method as shown below. The business logic is very simple. We hard code the price and availability.

```
public QuoteResponse getQuote(QuoteRequest req) {
          QuoteResponse res = new QuoteResponse();

          res.setClientId(req.getClientId());
          res.setDateRequiredBy(req.getDateRequiredBy());
          res.setPartNumber(req.getPartNumber());
          res.setQuantity(req.getQuantity());
          res.setTotalPrice(10.0 * req.getQuantity());
          res.setUnitPrice(10.00);
          res.setProductAvailable(true);

          return res;
}
```

__5. Annotate the method as shown below.

```
@POST
@Produces("text/xml")
public QuoteResponse getQuote(QuoteRequest req) {
```

Basically, this says that the method is called when a POST request is made. The MIME type of the response document is set to text/xml.

__6. Organize imports. Select **java.util.logging.Logger** and **javax.ws.rs.Produces**.

__7. Save changes.

Part 4 - Better Content Handling

Our getQuote() method expects that the input data should be in XML format. As a best practice, you should clearly state that using the @Consumes annotation. This might not be strictly necessary as it might work just fine with XML without this annotation. However, if we are to also add support for JSON, this annotation becomes mandatory.

In addition, XML has two different but equally applicable MIME types – application/xml and text/xml. We should designate both as acceptable.

__1. Add the @Consumes annotation to getQuote as follows in bold.

```
@POST
@Produces("text/xml")
@Consumes({MediaType.APPLICATION_XML, MediaType.TEXT_XML})
public QuoteResponse getQuote(QuoteRequest req) {
```

Note, as a best practice, we used the constant MediaType.APPLICATION_XML in place of "application/xml" and so on. This avoids the possibility of any typo.

__2. Similarly, change the @Produces annotation to support both MIME types for XML.

```
@POST
@Produces({MediaType.APPLICATION_XML, MediaType.TEXT_XML})
@Consumes({MediaType.APPLICATION_XML, MediaType.TEXT_XML})
public QuoteResponse getQuote(QuoteRequest req) {
```

__3. Organize imports. Select **javax.ws.rs.core.MediaType**.

__4. Save changes.

Part 5 - Unit Testing

We need to post an XML document to the server. This can not be done easily using a browser. We will use an open source test client (http://rest-client.googlecode.com/).

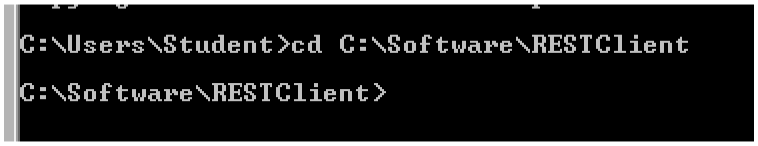
__1. Start the server if it is not running.

__2. Publish the server.

__3. Open a Windows Explorer file browser and verify the following directory exists. If it doesn't you might need to find out where WebSphere was installed and alter the next several steps accordingly.

```
C:\IBM\WebSphere\AppServer\java
```

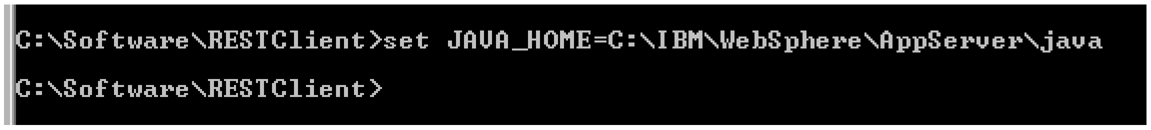
__4. Open a command window and change directories with the 'cd' command to **C:\Software\RESTClient**.



```
C:\Users\Student>cd C:\Software\RESTClient

C:\Software\RESTClient>
```

__5. Issue the following command assuming the WebSphere java directory was at the location previously mentioned:

```
set JAVA_HOME=C:\IBM\WebSphere\AppServer\java
```



```
C:\Software\RESTClient>set JAVA_HOME=C:\IBM\WebSphere\AppServer\java

C:\Software\RESTClient>
```

__6. Run the command:

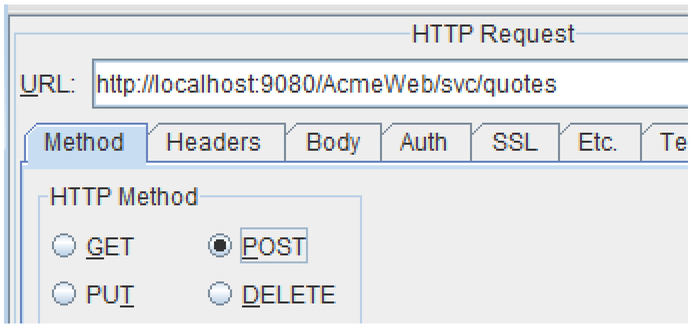
```
restclient
```

Note: If you close the RESTClient tool you will need to use the steps above to start it again. Since the lab setup doesn't install Java separately you must use the 'JAVA_HOME' variable to point to the WebSphere Java install.

__7. In the URL text box, enter:

```
http://localhost:9080/AcmeWeb/svc/quotes
```

__8. Select POST method.



URL: http://localhost:9080/AcmeWeb/svc/quotes

GET [✓ POST]
PUT DELETE

Now, we will configure the request body.

__9. Click the **Body** tab.

First, we will set the MIME type of the request. This has to be text/xml or application/xml for our service to work. Without that, JAX-RS will not use JAXB to convert the XML document into Java object.

__10. Click the icon.

__11. From the drop-down select the **application/xml** content type.

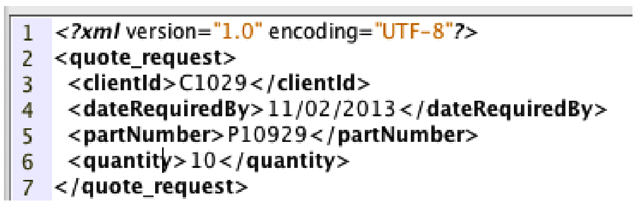
__12. Click **OK**.

Now, we will set the XML document for the body. To save time, a file has been given to you. You will simply import the contents.

__13. Click the icon to open the file.

__14. Select **C:\LabFiles\quote_request.xml** and open it.

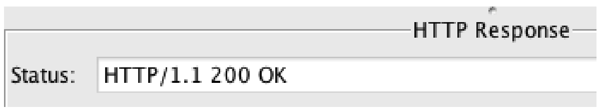
The tool may offer you an option to change the MIME type to application/xml. This is not necessary (either text/xml or application/xml will work fine), study the request document briefly.



```
<?xml version="1.0" encoding="UTF-8"?>
<quote_request>
  <clientId>C1029</clientId>
  <dateRequiredBy>11/02/2013</dateRequiredBy>
  <partNumber>P10929</partNumber>
  <quantity>10</quantity>
</quote_request>
```

__15. We are now ready start testing. Click the icon next to the URL.

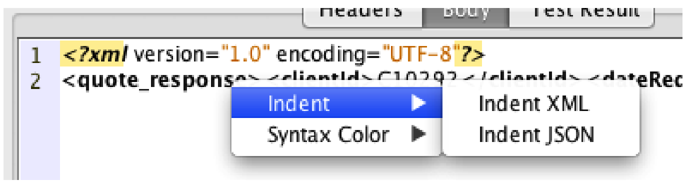
__16. Make sure that the HTTP reply status code is 200.



Status: HTTP/1.1 200 OK

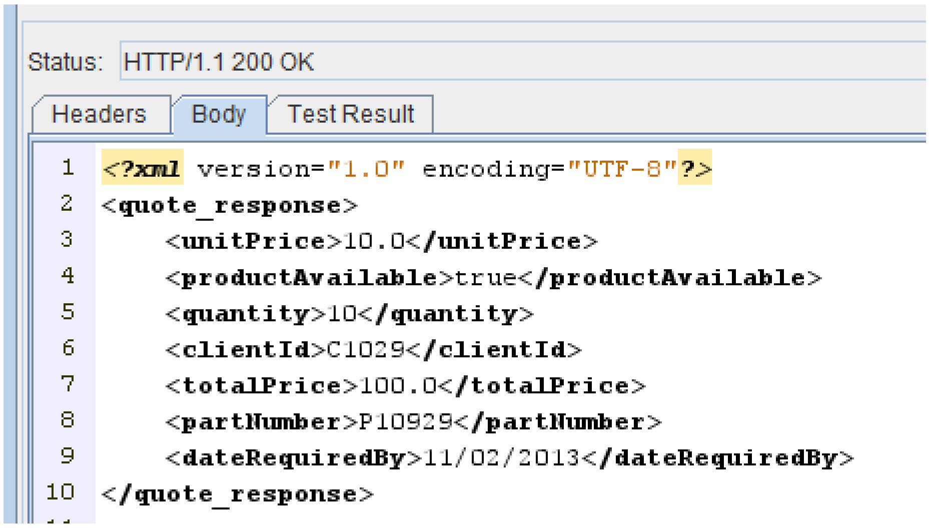
__17. Click the **Body** tab for the response.

__18. Right click the response body text and select **Indent > Indent XML**.



```
<?xml version="1.0" encoding="UTF-8"?>
<quote_response><clientId>C10292</clientId><dateRec
```

Study the response document.



```
<?xml version="1.0" encoding="UTF-8"?>
<quote_response>
    <unitPrice>10.0</unitPrice>
    <productAvailable>true</productAvailable>
    <quantity>10</quantity>
    <clientId>C1029</clientId>
    <totalPrice>100.0</totalPrice>
    <partNumber>P10929</partNumber>
    <dateRequiredBy>11/02/2013</dateRequiredBy>
</quote_response>
```

Note: The ordering of the elements may not match the ordering above.

__19. Back in Eclipse close all open files.

__20. Do not close the REST Client.

Part 6 - Review

In this lab, we implemented the quote service. The focus was to find out how to enable XML data in request and response. Basic steps are as follows:

1. Annotate the entity classes with JAXB annotations like @XmlRootElement.
2. Use the @POST annotation for the method since that is what the service design calls far. Without POST, it is difficult to send an XML document in the request.
3. Use the @Produces("text/xml") for the method to set the response MIME type. This will help the client identify the data type.

Lab 21 - Add Support for JSON

In the previous lab, we implemented the get quote operation using the XML data format. Browser based clients can work with XML since JavaScript includes a DOM API. However, JSON is much easier to work with from JavaScript. We will now add support for JSON. This will make life much easier for anyone developing a client using JavaScript.

This lab will also show you how content negotiation works.

Part 1 - Do the Design First

Before we go ahead and code support for JSON, we need to make a decision. Should we create separate methods for XML and JSON? Or, should a single method support both formats? Using the same method will naturally save work. In this case, however, we will choose to create a separate method to deal with JSON. Why? Consider the signature of the getQuote() method:

```
public QuoteResponse getQuote(QuoteRequest req)
```

If we handle JSON from this method, the JavaScript code in the browser will need to construct a JSON document and send it in the request body as follows:

```
{"clientId":"10101","dateRequiredBy":"01/23/2012","partNumber":"P1827",
"quantity":"15"}
```

Since, most web applications work with forms, it will take more work to construct the JSON document than to just submit a form.

So, we will now develop a method, that will take as input each quote request field as a URL parameter. But, it will return the quote response as a JSON object.

Part 2 - Add the JSON Method

__1. Open **QuoteService** class.

__2. First, add the shell of the method.

```
public QuoteResponse getQuoteJSON(
          String clientId,
          String dateRequiredBy,
          String partNumber,
          int quantity) {

}
```

__3. Then, annotate the method and the input parameters as shown in boldface below.

```
@POST
@Produces(MediaType.APPLICATION_JSON)
public QuoteResponse getQuoteJSON(
          @FormParam("clientId")
          String clientId,
          @FormParam("dateRequiredBy")
          String dateRequiredBy,
          @FormParam("partNumber")
          String partNumber,
          @FormParam("quantity")
          int quantity) {

}
```

__4. Now, fill in the body of the method. It basically calls the getQuote() method to get the job done.

```
logger.info("Client wants JSON response");
QuoteRequest req = new QuoteRequest();

req.setClientId(clientId);
req.setDateRequiredBy(dateRequiredBy);
req.setPartNumber(partNumber);
req.setQuantity(quantity);

return getQuote(req);
```

__5. Organize imports.

__6. Save changes.

Part 3 - Understanding Content Negotiation

Now, we have two methods – getQuote() and getQuoteJSON(). How do we determine which method gets called and when? We can assign separate paths for each method, such as /quotes/xml and /quotes/json. But, there is a better way of doing this. We can inspect the capability of the client through the HTTP Accept header. If the client prefers to receive XML, we will send back XML. if the client prefers JSON, we will send back JSON.

The MIME types in the Accept header are matched to a resource method using the @Produces annotation. We have already annotated the two methods accordingly. Now, if the Accept header shows preference for application/xml, then getQuote() will be called. If the preference is application/json then getQuoteJSON() will be called.

Part 4 - Test

First, we will do regression testing and make sure that support for XML is not broken.

__1. Publish the server by right-clicking and selecting **Publish**.

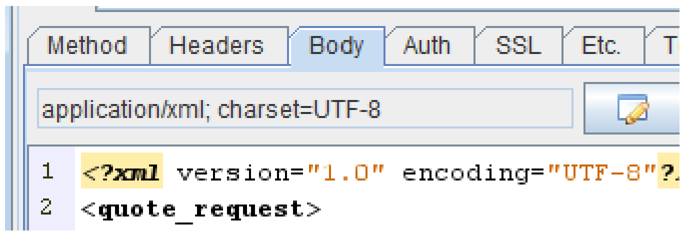
__2. Launch the REST client tool if it is not already running. Use steps from previous labs to set the 'JAVA_HOME' variable in the command prompt to launch the tool.

__3. Set the URL to:

```
http://localhost:9080/AcmeWeb/svc/quotes
```

__4. Choose **POST** method.

__5. Set the body using the XML code in **C:\LabFiles\quote_request.xml** and set content type to **application/xml; chartset=UTF-8**.



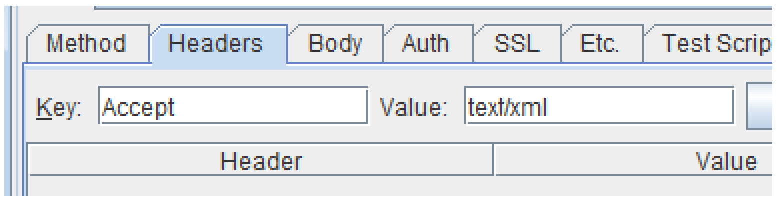
application/xml; charset=UTF-8

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

Now, we will configure the Accept header. Without this content negotiation will not work.

__6. Click the **Headers** tab.

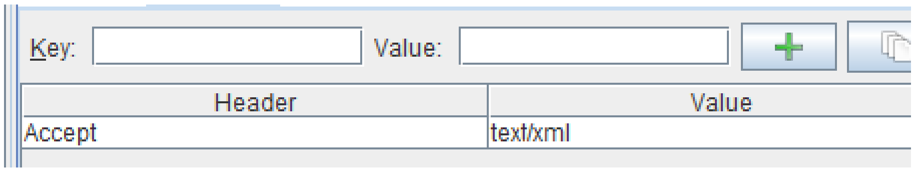
__7. Enter **Accept** as Key and **text/xml** as Value.



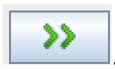
Key: Accept Value: text/xml

Note: The MIME type in Accept header must match what was stated in the @Produces annotation.

__8. Click the + icon to add the header.



Header	Value
Accept	text/xml

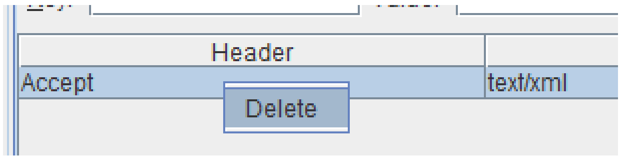
__9. Run the test by clicking .

__10. Right click the response body area in the **Body** tab and select **Indent > Indent XML**.

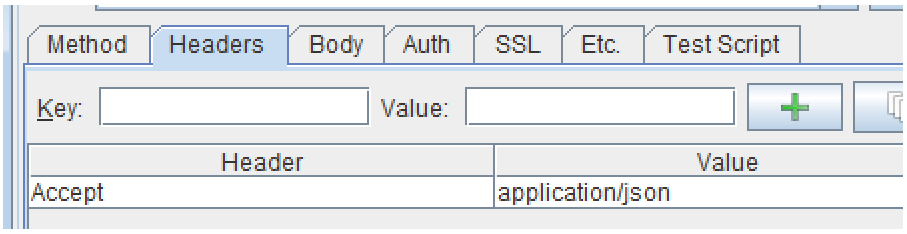
__11. Verify that the reply body contains the response.

Now, we will test for JSON.

__12. Select the **Accept** header, then right click on it and select **Delete**.

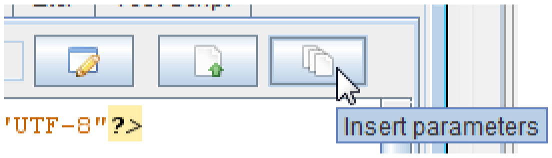


__13. Enter **Accept** as Key and **application/json** as Value, then click the + icon to add the header.

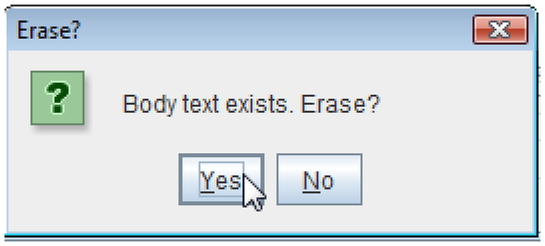


Header	Value
Accept	application/json

__14. Select the request **Body** tab and click on the button for '**Insert parameters**'

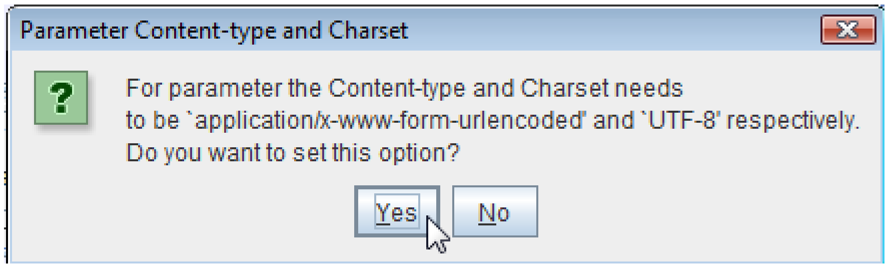


__15. Confirm that you want to erase the current body text by clicking the **Yes** button.


Erase?

Body text exists. Erase?

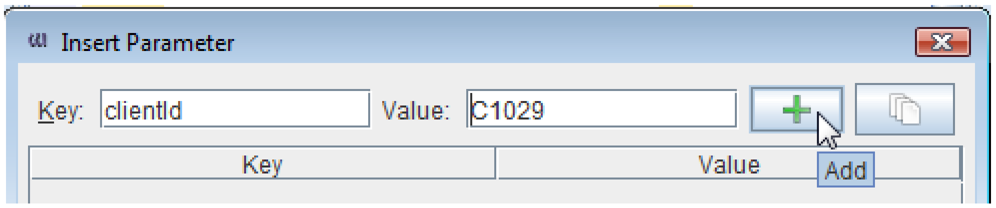
__16. Confirm that you want to switch the content type to the value needed for a form, 'application/x-www-form-urlencoded', by clicking the **Yes** button.


Parameter Content-type and Charset

For parameter the Content-type and Charset needs
to be `application/x-www-form-urlencoded' and `UTF-8' respectively.
Do you want to set this option?

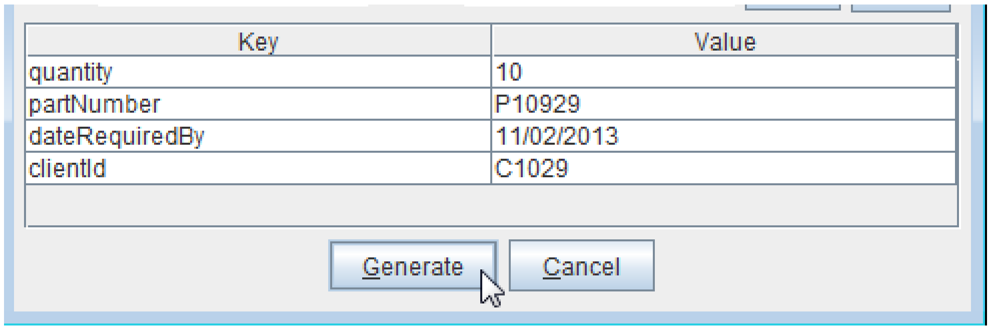
__17. Use the tool to add each of the following parameters by filling in the key and value and clicking the button to add the parameter.

clientId	C1029
dateRequiredBy	11/02/2013
partNumber	P10929
quantity	10


Insert Parameter

Key: clientId Value: C1029

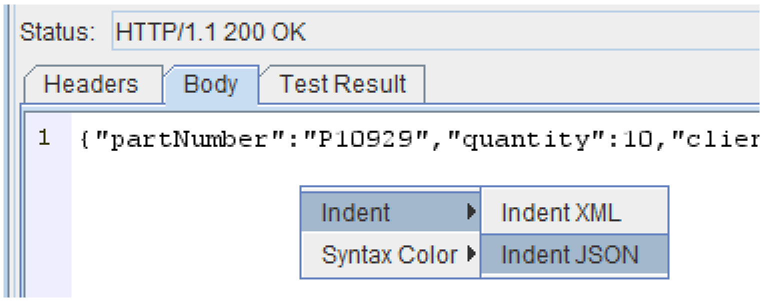
__18. Once you have all the parameters filled in as shown below, click the **Generate** button.



Key	Value
quantity	10
partNumber	P10929
dateRequiredBy	11/02/2013
clientId	C1029

__19. Run the test.

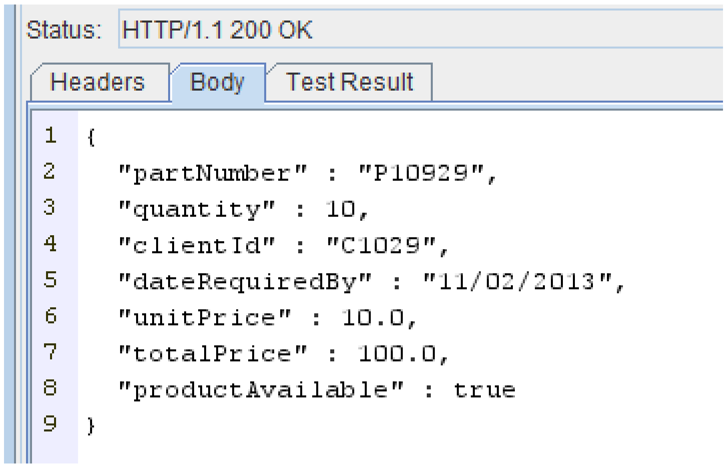
__20. Right click the response body area and select **Indent > Indent JSON**.



Status: HTTP/1.1 200 OK

```
{"partNumber":"P10929","quantity":10,"clier
```

__21. Verify the result.



Status: HTTP/1.1 200 OK

```
{
  "partNumber" : "P10929",
  "quantity" : 10,
  "clientId" : "C1029",
  "dateRequiredBy" : "11/02/2013",
  "unitPrice" : 10.0,
  "totalPrice" : 100.0,
  "productAvailable" : true
}
```

__22. Back in Eclipse close all open files.

__23. Do not close the REST Client.

Part 5 - Review

Our service can now talk JSON as well as XML. We designed the JSON interface to be very friendly to browser based applications. A request is sent using the familiar paradigm of submitting a form or clicking a hyperlink. The response contains a JSON document which can be easily parsed into a JavaScript object. We will get into client development later.

Lab 22 - Build the Order Web Service

In this lab, we will start to build the order web service. Specifically, the goal of this lab is to learn about ResponseBuilder to control the HTTP response. Sometimes, we have to go beyond just using annotations to better take advantage of the HTTP protocol. ResponseBuilder is a part of that strategy. Example use cases for ResponseBuilder are:

- By default the HTTP status code is set to 200 or 204 (when the reply body is empty). In some cases a different status code may be more appropriate. For example, 404 when a queried entity can not be found or 304 when a queried entity has not changed since it was last retrieved by the client. The latter example can optimize network bandwidth by not returning the entity.
- By default the @Produces annotation fixes the reply content type. You can dynamically change the content type using ResponseBuilder.
- Work with cookies.
- Throw an exception from a resource method and let JAX-RS convert that to an appropriate reply.

Part 1 - Create the Resource Class

__1. In the **com.acme.services** package, create a class called **OrderService**

__2. Annotate the class to set the path as follows.

```
@Path("/orders")
public class OrderService {
```

__3. Add a member variable for logging.

```
public class OrderService {
          Logger logger = Logger.getLogger("OrderService");
```

__4. Organize imports. Select **java.util.logging.Logger**.

__5. Save changes.

Part 2 - Implement Order Placement

We will now develop a method that will let clients place a new order. First, we will work on the design of the method. The design considerations are very typical of a method that creates an entity. Pay attention to each factor carefully:

- The method will be invoked when a POST request is sent to the /orders path. The method doesn't need any extra path of its own.
- The method will take as input a single Order object.
- The method should return an HTTP status code of 201 (created).
- The method will return a String indicating the URL that points to the new order. This URL is normally stored in the Location HTTP reply header. Other than this URL, no other information is returned and the reply body is empty.

__1. Write the shell of the method as follows. We will not deal with the returned URL just yet.

```
public Response placeOrder(Order o) {

    return null;
}
```

Here, we return a javax.ws.rs.core.Response from the method. This is necessary when you want to set a specific status (201 in our case) and work with the HTTP headers (Location in this case).

__2. Annotate the method with @POST and @Consumes.

```
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.TEXT_XML})
public Response placeOrder(Order o) {
```

__3. Organize imports. Select these classes when prompted (they may come up in a different order):

- javax.ws.rs.core.Response
- com.acme.services.entities.Order
- javax.ws.rs.core.MediaType

__4. Save changes.

We are now ready to develop the business logic. As a matter of architecture, you should never directly code the business logic in a resource class. Think of a resource class as a Facade or a front door to the application. Actual business logic is developed in service classes or Data Access Object (DAO) classes. A Facade is very much tied down to a specific communication protocol, XML over HTTP in our case. In contrast, the service and DAO classes are generic and usable from any protocol (such as a Servlet, Struts action, SOAP web service and so on).

To save time, a DAO class is already given to you. In real life, this class will probably use a database to store the orders. To keep things simple, it stores the orders in a Hashtable (not a HashMap for thread safety). This internal implementation can be changed easily without affecting the web service.

__5. Copy the file **C:\LabFiles\OrderDAO.java** into the **com.acme.services** package.

__6. Open the class and briefly study it. The code should be very simple.

Now, we will use the OrderDAO class from the resource class.

__7. In the **placeOrder** method of the **OrderService** class, use OrderDAO as shown in boldface below.

```
public Response placeOrder(Order o) {
    try {
        OrderDAO dao = new OrderDAO();
        long orderId = dao.createOrder(o);
    } catch (Exception e) {
        logger.log(Level.SEVERE, "Error creating order", e);
    }

    return null;
}
```

__8. Organize imports. Select this class:

- java.util.logging.Level

__9. Save changes.

We will not deal with the URL for the newly created order just yet. We will get to that soon.

Part 3 - Implement Retrieval of Order

We will now implement the method that returns an order given an order ID. The design considerations are typical of a method that returns an entity:

- The path for the method will be /orders/{orderId}
- Method is GET.
- The reply body will contain an Order object in XML format.

__1. First write the method shell along with the required annotations.

```
@GET
@Path("/{orderId}")
@Produces({MediaType.APPLICATION_XML, MediaType.TEXT_XML})
public Order getOrder(@PathParam("orderId") long orderId) {

}
```

__2. Next, fill in the method body.

```
public Order getOrder(@PathParam("orderId") long orderId) {
          Order o = null;

          try {
              OrderDAO dao = new OrderDAO();
              o = dao.findOrder(orderId);
          } catch (Exception e) {
              logger.log(Level.SEVERE, "Error finding order", e);
          }

          return o;
}
```

__3. Organize imports. Select these classes:

- javax.ws.rs.Produces

__4. Save changes.

Part 4 - Construct Entity URL

As a general convention, a method that creates a new entity should return the URL to the newly created entity. This is saved in the Location reply header. HTTP status code is set to 201.

In our case, the placeOrder() method should return a URL like "http://localhost:9080/AcmeWeb/svc/orders/1000". There are a couple of challenges to this:

- We should not hard code the host name, port number and web application context root. This can change from development to production.
- We should not assume the position of the order ID in the URI. In this particular case, the order ID can only go at the end. For more complex cases, when multiple identifying parameters need to be stored in the URI, the order becomes important.

JAX-RS provides us with a couple of classes like UriInfo and UriBuilder that helps us solve these problems. The UriInfo class can extract the URL used to send the original HTTP request. This way, we don't have to hard code the host name, port number, context root etc. The UriBuilder can help us add the order ID in the correct place by querying the path configuration of the getOrder() method. Let's get started.

__1. First, add a member variable to the class as follows.

```
public class OrderService {
          @Context
          UriInfo ui;
```

__2. In the placeOrder() method, add the bold code:

```
try {
          OrderDAO dao = new OrderDAO();
          long orderId = dao.createOrder(o);

          UriBuilder ub = ui.getRequestUriBuilder();
} catch (Exception e) {
```

This gives us a UriBuilder object that has already been initialized with the URL used in the original request. That will be http://localhost:9080/AcmeWeb/svc/orders.

__3. Add this line in bold below:

```
                    UriBuilder ub = ui.getRequestUriBuilder();
                    ub = ub.path(OrderService.class, "getOrder");
          } catch (Exception e) {
```

This adds to the URL the path structure required by the getOrder() method. That means, "/{orderId}" is added to the URL.

__4. Next, add the following line. This plugs in the actual value of the order ID to the path and returns a java.net.URI object.

```
                    UriBuilder ub = ui.getRequestUriBuilder();
                    ub = ub.path(OrderService.class, "getOrder");
                    URI orderURI = ub.build(orderId);
          } catch (Exception e) {
```

__5. Finally, add this line.

```
                    ub = ub.path(OrderService.class, "getOrder");
                    URI orderURI = ub.build(orderId);

                    return Response.created(orderURI).build();
          } catch (Exception e) {
```

The Response.created() does a number of things:

- It saves the URL in the Location header.
- It sets the HTTP status code to 201.

__6. Organize imports. Choose these classes:

- javax.ws.rs.core.Context
- java.net.URI

__7. Save the file and make sure you don't have any compilation errors.

Part 5 - Test

We can now test the order creation and retrieval use cases. To save time, a pre-configured set of requests are already given to you. You will open them from the REST test client and run them.

First, we will test for order creation.

__1. Go to the 'Servers' view and publish the changes to the server by right clicking the server and selecting '**Publish**'.

__2. Launch the REST client tool if it is not already running. Use steps from previous labs to set the 'JAVA_HOME' variable in the command prompt to launch the tool.

__3. From the REST client's menu bar, select **File Open Request (Control+O)**.

__4. Open **C:\LabFiles\order_create.rcq**

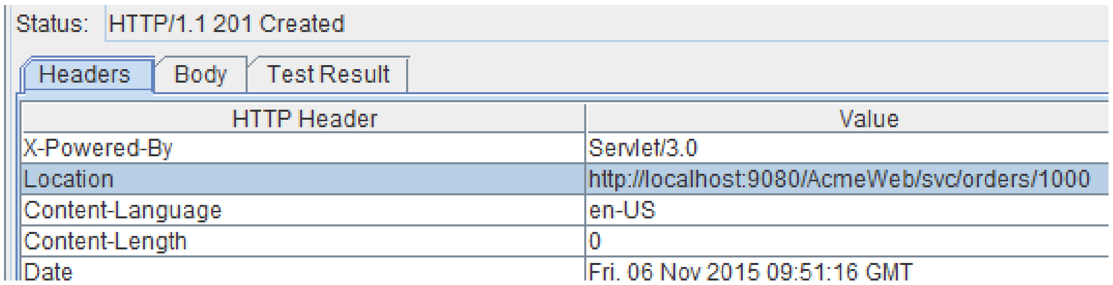
__5. Review these items about the request:

- The URL is http://localhost:9080/AcmeWeb/svc/orders
- The method is POST.
- The body contains an order XML document.
- The request MIME type is text/xml.

__6. Click the button to send the request.

__7. Verify that:

- The status code is 201.
- The Location header is http://localhost:9080/AcmeWeb/svc/orders/1000



Status: HTTP/1.1 201 Created

HTTP Header	Value
X-Powered-By	Servlet/3.0
Location	http://localhost:9080/AcmeWeb/svc/orders/1000
Content-Language	en-US
Content-Length	0
Date	Fri. 06 Nov 2015 09:51:16 GMT

__8. In the Eclipse Console, verify the log output from the OrderDAO class.

```
[11/6/15 4:51:16:307 EST] 0000007b OrderService  I   New order ID: 1000
[11/6/15 4:51:16:307 EST] 0000007b OrderService  I   Item: P01010 Qty: 10
[11/6/15 4:51:16:307 EST] 0000007b OrderService  I   Item: P18210 Qty: 5
```

Now, we will test for order retrieval.

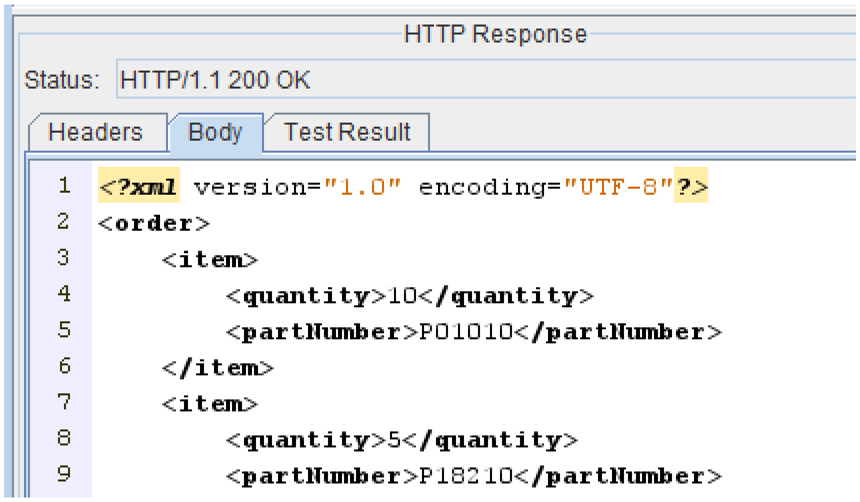
__9. In the REST client, open the request file **C:\LabFiles\order_retrieve.rcq**.

__10. Inspect these items about the request:

- The URL is http://localhost:9080/AcmeWeb/svc/orders/1000.
- The method is GET.

__11. Send the request.

__12. Verify that the reply body has the correct order information.



HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <item>
        <quantity>10</quantity>
        <partNumber>P01010</partNumber>
    </item>
    <item>
        <quantity>5</quantity>
        <partNumber>P18210</partNumber>
```

Part 6 - Enable Error Handling

Right now, the OrderDAO class has decent error handling built in. The createOrder() method throws an exception if there is no line item in the order. The findOrder() method returns null if the order ID is invalid. However, the service resource class is doing nothing to deal with these errors. Let us first decide how we will handle the error situations from the OrderService class:

- If createOrder() fails with an exception, we should set the reply status code to 500.
- If findOrder() returns null, we should return a status code of 404.

We can throw the WebApplicationException unchecked exception to achieve both objectives.

__1. In the **placeOrder**() method of **OrderService** class, add this line shown in bold face.

```
} catch (Exception e) {
          logger.log(Level.SEVERE, "Error creating order", e);
          throw new WebApplicationException(e);
}
```

__2. Comment out the following unreachable line:

```
// return null;
```

__3. Organize imports.

__4. In the **getOrder**() method, above the line:

```
return o;
```

__5. Add the following bold code:

```
if (o == null) {
    throw new WebApplicationException(Status.NOT_FOUND);
}

return o;
```

__6. Organize imports. Choose this class:

- javax.ws.rs.core.Response.Status

__7. Save changes.

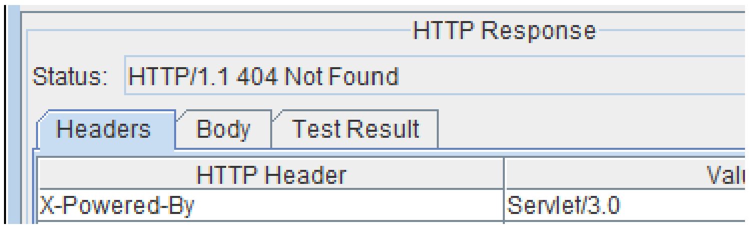
Part 7 - Test

First, we will test for order retrieval failure. After you saved the code changes, the web application was restarted. All previously created orders have been destroyed in the process. Any request to obtain order information will fail.

__1. Publish the server in the **Servers** view.

__2. In the REST client, the order retrieval request file should be open already. Just run the test.

__3. Verify that the status code is 404. This is because the previous order was lost when the application was updated and restarted.



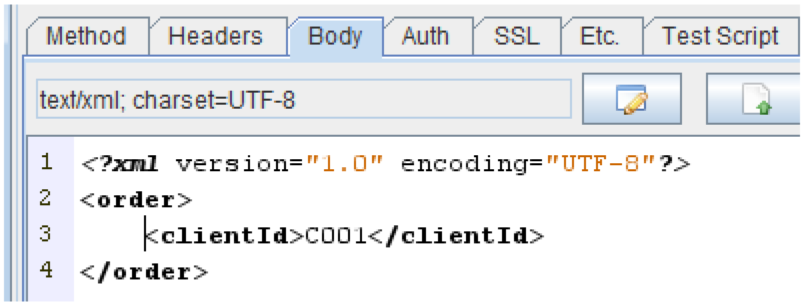
Status: HTTP/1.1 404 Not Found

HTTP Header	Valı
X-Powered-By	Servlet/3.0

Now, we will test for order creation failure.

__4. Open the request file **C:\LabFiles\order_create.rcq**.

__5. In the **Body** tab, delete all the <item> elements.

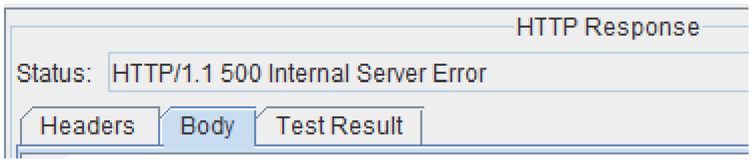


text/xml; charset=UTF-8

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <clientId>C001</clientId>
</order>
```

__6. Run the test.

__7. Verify that status code 500 is returned.



Status: HTTP/1.1 500 Internal Server Error

Troubleshooting: If you get a 405 (Method Not Found) error, try closing RESTClient and reopening it. In all likelihood, the error will go away.

__8. Do regression tests to make sure that success conditions are working fine.

__9. In Eclipse close all open files.

__10. Do not close the REST Client.

Part 8 - Review

In this lab, we learned how to implement classic create and retrieve use cases in a RESTful web service. You can use this lab as a template for most real life web services. A couple of new concepts were introduced:

1. We used the UriBuilder class to return the URL of an entity.
2. We used the WebApplicationException class to indicate errors to the JAX-RS engine.

Lab 23 - Complete the Order Service

So far, we have implemented the create and retrieve operations for orders. In this lab, we will add support for getting order history for a client organization and cancellation of order. In addition, although, the requirements do not call for an update function, we will add that feature just to learn how to do so.

Part 1 - Implement Order Cancellation

We can think of cancellation in two different ways:

1. After an order is canceled, it is practically removed from the system. In this case, we should model it using the DELETE request method.
2. Cancellation of an order is nothing other than updating the status to canceled. Updates are done using POST request.

In this lab, we will choose to use the DELETE request.

The URIs that update or delete an entity should have the ID in the path. In our case, the path will be /orders/{orderId}.

In summary, a client will send a DELETE request to the URI /orders/{orderId} to cancel an order.

__1. In the **AcmeWeb** project, open **OrderService.java**

__2. Add this method.

```
@DELETE
@Path("/{orderId}")
@Produces({ MediaType.APPLICATION_XML, MediaType.TEXT_XML })
public void cancelOrder(@PathParam("orderId") long orderId) {


}
```

Note that the path for this method is same as that for getOrder(). The only thing that differentiates them is the HTTP method.

__3. Add these lines to the cancelOrder() method.

```
public void cancelOrder(@PathParam("orderId") long orderId) {
          OrderDAO dao = new OrderDAO();

          try {
                    logger.info("Canceling order: " + orderId);
                    dao.cancelOrder(orderId);
          } catch (Exception e) {
                    throw new WebApplicationException(e);
          }
}
```

__4. Organize imports.

__5. Save changes.

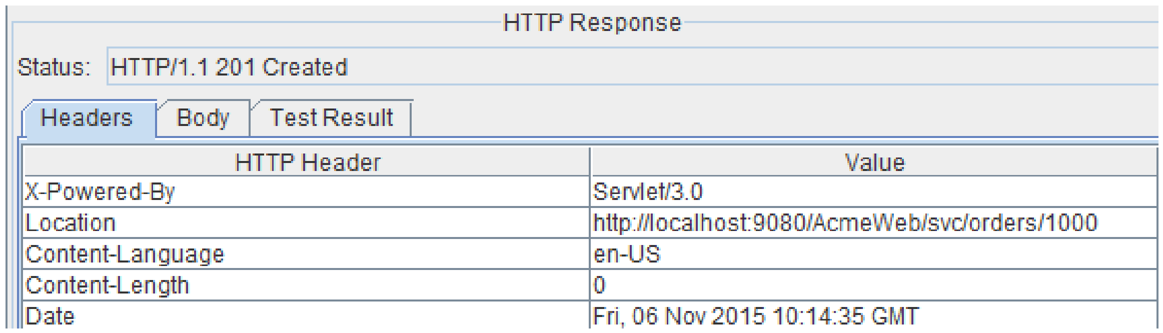
Part 2 - Test

__1. Publish the server.

__2. Launch the REST client tool if it is not already running. Use steps from previous labs to set the 'JAVA_HOME' variable in the command prompt to launch the tool.

__3. From the client, open the request file **C:\LabFiles\order_create.rcq**

__4. Run the test. You should get a status code of 201.



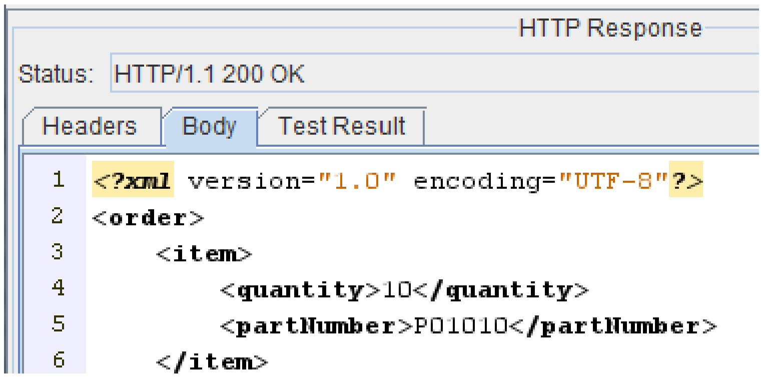
Status: HTTP/1.1 201 Created

HTTP Header	Value
X-Powered-By	Servlet/3.0
Location	http://localhost:9080/AcmeWeb/svc/orders/1000
Content-Language	en-US
Content-Length	0
Date	Fri, 06 Nov 2015 10:14:35 GMT

Now, we will test to make sure that we can get the order status by its ID. We have already run this test before.

__5. Open the request file **C:\LabFiles\order_retrieve.rcq**

__6. Run the test. You should get a status code of 200.

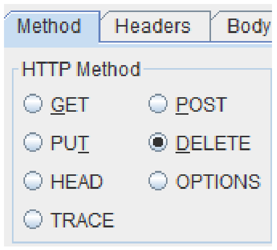


Status: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <item>
        <quantity>10</quantity>
        <partNumber>P01010</partNumber>
    </item>
```

We will now cancel the order. The URL will be same as for retrieving the order – http://localhost:9080/AcmeWeb/svc/orders/1000. Only thing we have to do differently is to use the DELETE method.

__7. From the **Method** tab, select **DELETE**.

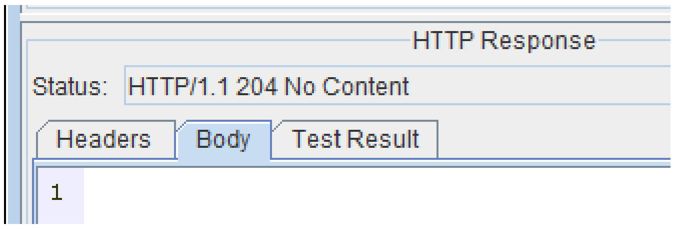


HTTP Method

GET POST PUT [✓ DELETE] HEAD OPTIONS TRACE

__8. Run the test.

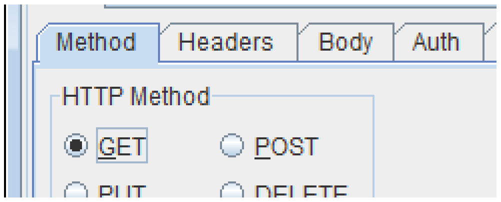
__9. Since the reply body is empty, you should see HTTP status code 204. This is equivalent to status 200 and indicates success.



Status: HTTP/1.1 204 No Content

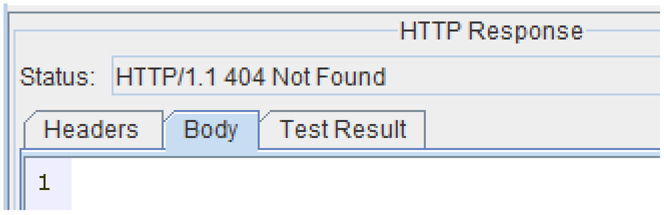
Now, if we try to retrieve the order, we should get a 404 not found status.

__10. In the **Method** tab, select **GET** method.



__11. Run the test.

__12. Verify that 404 is returned.



Status: HTTP/1.1 404 Not Found

```
1
```

Part 3 - Implement Order History

We will now add a method to the web service that will return all past orders placed by anyone from a given organization. The reply XML will look something like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<order_list>
  <order>
    <item>
        <partNumber>P01010</partNumber>
        <quantity>10</quantity>
    </item>
    <item>
        <partNumber>P18210</partNumber>
        <quantity>5</quantity>
    </item>
    <clientId>C001</clientId>
    <orderId>1000</orderId>
    <orderTotal>0.0</orderTotal>
  </order>
  <order>
    <item>
        <partNumber>X1923</partNumber>
        <quantity>3</quantity>
    </item>
    <clientId>C001</clientId>
    <orderId>1030</orderId>
    <orderTotal>0.0</orderTotal>
  </order>
</order_list>
```

The OrderList entity class represents the <order_list> root element.

__1. From the **com.acme.services.entities** package, open **OrderList.java** and inspect it.

Now, we will have to design the interface (URI, method etc.) for the order history feature. The question you will have to ask is, what will uniquely identify all orders from a client organization? A possible syntax could be /orders/clients/{clientId}. The HTTP method should be GET.

__2. Close the class.

__3. In the **OrderService** class, add the method shell as follows.

```
@GET
@Path("/clients/{clientId}")
@Produces({ MediaType.APPLICATION_XML, MediaType.TEXT_XML })
public OrderList getOrderHistory(
          @PathParam("clientId") String clientId) {

}
```

__4. Add the method body as follows.

```
public OrderList getOrderHistory(
          @PathParam("clientId") String clientId) {
          OrderList list = null;

          try {
                    OrderDAO dao = new OrderDAO();
                    list = dao.findOrderForOrganization(clientId);
          } catch (Exception e) {
                    throw new WebApplicationException(e);
          }

          return list;
}
```

__5. Organize imports.

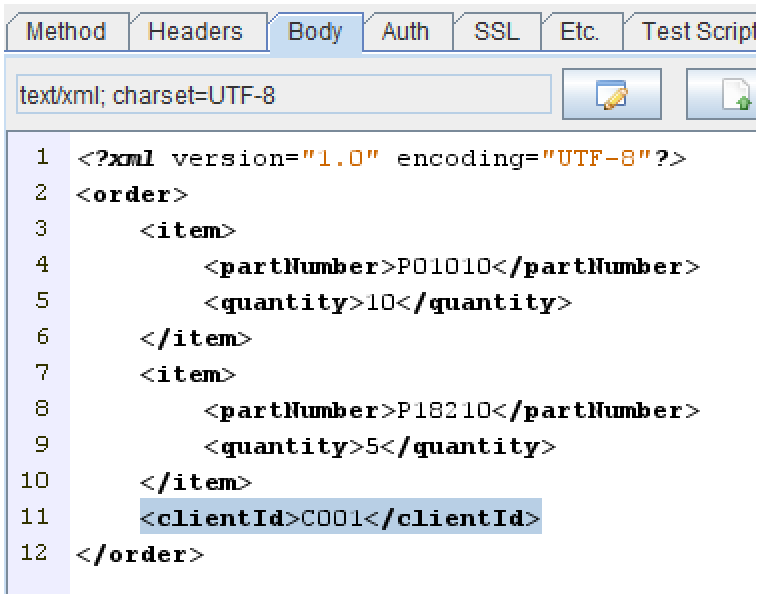
__6. Save changes.

Part 4 - Test

__1. Publish the server.

__2. From the REST test client, open the request file **C:\LabFiles\order_create.rcq**

__3. From the **Body** tab, note the client ID – C001.



text/xml; charset=UTF-8

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <item>
        <partNumber>P01010</partNumber>
        <quantity>10</quantity>
    </item>
    <item>
        <partNumber>P18210</partNumber>
        <quantity>5</quantity>
    </item>
    <clientId>C001</clientId>
</order>
```

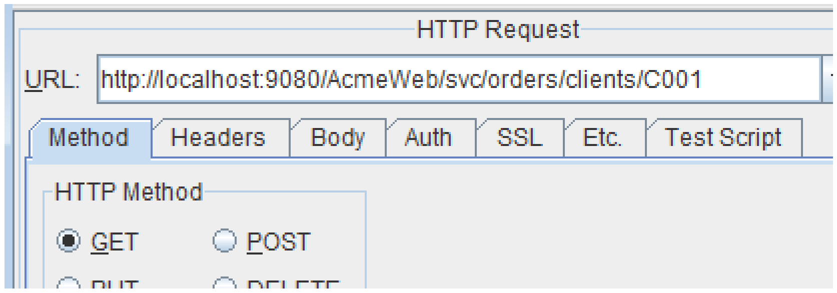
__4. Run the test to create an order.

__5. From the **Body** tab of the request, change the **part number** and **quantity ordered** and place another order.

Now, we will test for the order history use case.

__6. From the test client, open the request file **C:\LabFiles\order_history.rcq**

Note the URL and HTTP method for the request.



URL: http://localhost:9080/AcmeWeb/svc/orders/clients/C001

HTTP Method: [✓ GET] POST PUT DELETE

__7. Run the test.

__8. Verify the response.

Part 5 - Implement Order Update Function

Although, we did not identify need for an order update function, we will develop it anyway to complete the CRUD implementation.

As usual, first, we need to design the interface. Since we are going to update an existing entity by replacing it (not by a partial update), we will use the PUT method. The unique identifier of the entity we are going to update is placed in the URI. In our case, the URI will be in the /orders/{orderId} form.

__1. In the **OrderService** class, add the method shell.

```
@PUT
@Path("/{orderId}")
@Consumes({ MediaType.APPLICATION_XML, MediaType.TEXT_XML })
public Response updateOrder(
          @PathParam("orderId") long orderId, Order newOrder) {


}
```

__2. Fill in the method body as follows.

```
public Response updateOrder(
          @PathParam("orderId") long orderId, Order newOrder) {
          Order o = null;

          try {
                    OrderDAO dao = new OrderDAO();
                    o = dao.updateOrder(orderId, newOrder);
          } catch (Exception e) {
                    logger.log(Level.SEVERE,
                              "Error updating order", e);
          }

          if (o == null) {
                    throw new WebApplicationException(
                              Status.NOT_FOUND);
          } else {
                    return Response.ok().build();
          }
}
```

__3. Organize imports.

__4. Save changes.

Part 6 - Test

First, we will place a new order.

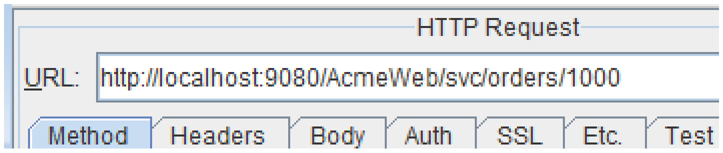
__1. Publish the server.

__2. From the REST test client, open the request file **C:\LabFiles\order_create.rcq**

__3. Run the test.

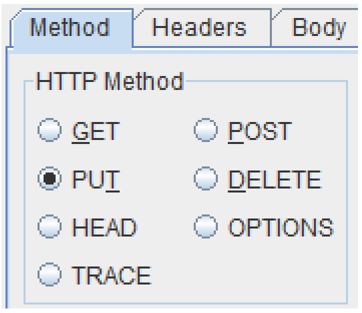
Now, we will test for order update. We will modify the request that created the order.

__4. Add the order ID 1000 to the URL (add **/1000** at the end of the URL).



URL: http://localhost:9080/AcmeWeb/svc/orders/1000

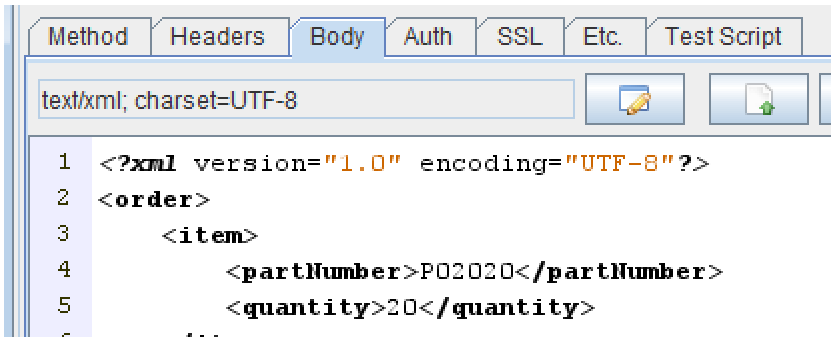
__5. From the **Method** tab, choose **PUT**.



○ GET ○ POST
[✓ PUT] ○ DELETE
○ HEAD ○ OPTIONS
○ TRACE

Now, we make some changes to the order data.

__6. From the **Body** tab, change quantity and/or part number.

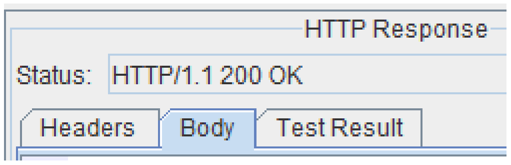


text/xml; charset=UTF-8

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <item>
        <partNumber>P02020</partNumber>
        <quantity>20</quantity>
```

__7. Run the test.

__8. Verify that status code 200 is returned.



Status: HTTP/1.1 200 OK

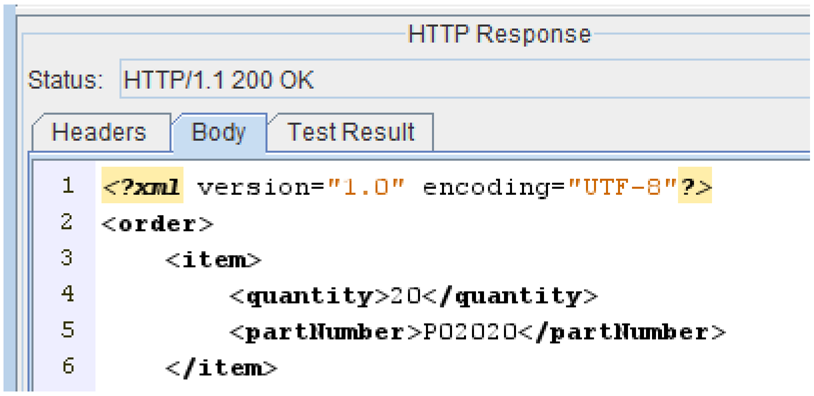
__9. Now lets check the retrieve request. Open the request file **C:\LabFiles\order_retrieve.rcq**

__10. Run the test.

__11. Verify that status code 200 is returned.

Note: There is no HTTP response code for the "Resource updated / patched" status and REST re-uses code 200 (OK) in this context for that.

__12. Format the result (Indent > Indent XML) and verify that the body contains the information that you changed.



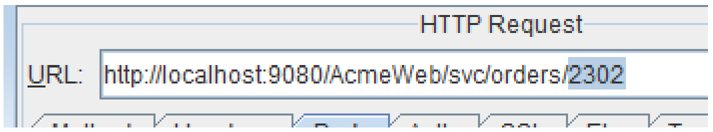
Status: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <item>
        <quantity>20</quantity>
        <partNumber>P02020</partNumber>
    </item>
```

Now, we will test the failure condition.

__13. Open the request file **C:\LabFiles\order_create.rcq**

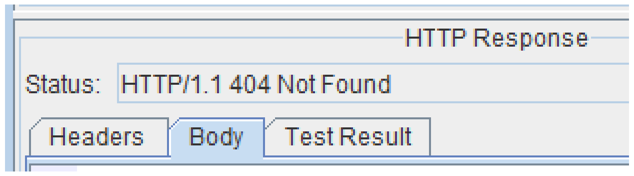
__14. Change the order ID in the URL to an invalid value as shown below.



URL: http://localhost:9080/AcmeWeb/svc/orders/2302

__15. From the **Method** tab, choose **PUT**.

__16. Run the test and make sure 404 status code is returned.



Status: HTTP/1.1 404 Not Found

__17. Close all the open files.

Part 7 - Review

In this lab, we completed the full CRUD implementation for the order entity. The focus has always been the design of the interface. You should be able to use these exercises as templates for most RESTful web services.

Lab 24 - Developing a JAX-RS Client

There are mainly two ways to consume a RESTful web service:

1. **From browser:** A service URL can be requested by a web page using JavaScript and AJAX. The web page and the service must have the same origin policy. This limits the client to calling in house services only.
2. **From application:** A service URL can be requested from an application developed in Java, C#, PHP. There is no same origin policy restriction and you can invoke web service from any location. Complex application integration across organizational boundaries should be done this way.

In this lab, we will explore option #2.

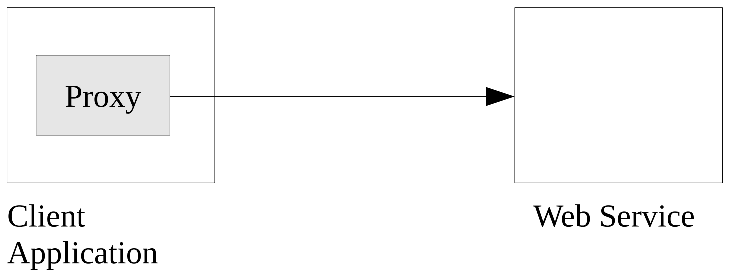
JAX-RS 2.0 will have a client API. For now, a Java based client has access to several other APIs to develop a client:

1. The java.net.* package has classes like URL and HttpURLConnection that can be used to send HTTP requests.
2. Apache foundation provides a library with classes like DefaultHttpClient. They provide a richer set of functions than the java.net.* package. Also, this library is available from Android mobile OS.
3. Apache Wink implementation of JAX-RS comes with its own set of client API.

In this lab, we will choose to use the Wink client API. The advantage of using this API is that we do not need to package any extra JAR files with the application. Wink JAR files are already deployed to WebSphere Application Server.

We will build a simple web based application that will act as a client of the price quote service.

Part 1 - Client Design Considerations



The client application uses a proxy class to invoke a remote service. This proxy class hides the details of network communication, data marshaling and security. That way, it follows the Data Access Object (DAO) pattern.

Acme is responsible for implementing the web service. Its customers, on the other hand, will implement the client applications. Acme should make life easier for its customers by providing a client API JAR file containing the proxy classes.

In other words, the job of developing client gets divided as follows:

1. Acme produces a JAR file containing all the classes necessary to invoke its web services.
2. A customer organization uses the JAR file and invokes web services.

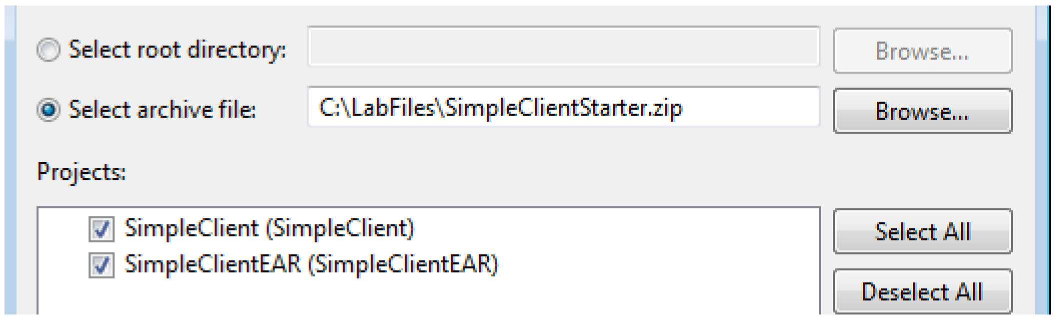
Part 2 - Import the Client Application

To save time, a skeletal version of the client application is given to you. You will import it now.

__1. From the Eclipse menu bar, select **File** → **Import**.

__2. Select **General** → **Existing Projects into Workspace** and click **Next**.

__3. Select the '**Select archive file**' radio button and use the **Browse** button to the right to select the **C:\LabFiles\SimpleClientStarter.zip** file.


Select root directory:
Select archive file: C:\LabFiles\SimpleClientStarter.zip
Projects:
☑ SimpleClient (SimpleClient)
☑ SimpleClientEAR (SimpleClientEAR)

__4. Make sure both projects are selected as shown above and click **Finish**.

This is just a plain web project. A servlet called QuoteController is created. Also, a few JSP files are added for the web front end. We will get to work with them later. Since the Wink client API is already part of the server which is on the classpath of the project no special settings are needed.

Part 3 - Build the Service Proxy Layer

Now, we will create the classes that will make up the proxy layer. Normally, this work is done by the same organization that offers the service. These classes are then packaged in a JAR file. To keep things simple, we will directly develop the classes in the SimpleClient project.

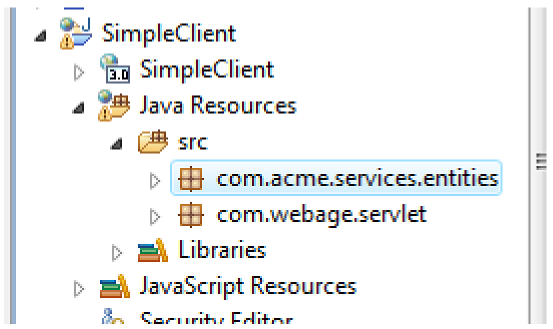
The proxy layer is made up of two types of classes:

1. The entity classes, like, QuoteRequest, QuoteResponse and Order.
2. The DAO classes that perform the network communication and data marshaling.

We can easily copy and paste the entity classes from the web service implementation project. We will do exactly that.

__1. From the **AcmeWeb** project, copy the **com.acme.services.entities** package. Make sure to copy the entire package by right clicking the package and selecting **Copy**.

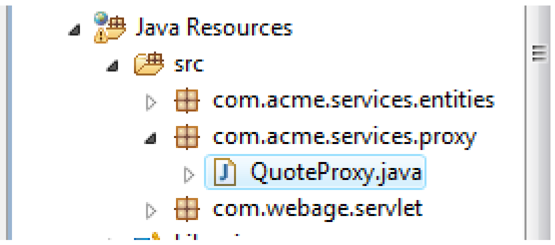
__2. In the **SimpleClient** project, paste it in the **src** folder.



Now, we will develop the DAO class.

__3. In the **SimpleClient** project, create a package called **com.acme.services.proxy**

__4. In that package, create a class called **QuoteProxy**



__5. In the QuoteProxy class, add a variable to store the service root URL.

```
public class QuoteProxy {
        String baseURL = "http://localhost:9080/AcmeWeb/svc/quotes";
```

__6. Add the shell of a method that will invoke the quote web service.

```
public QuoteResponse getQuote(QuoteRequest req) {

}
```

Now, we will develop the client code using the Wink client API.

__7. In the getQuote() method, add this line to instantiate an org.apache.wink.client.RestClient object.

```
RestClient client = new RestClient();
```

__8. Next, create a Resource object that points to the root web service resource.

```
Resource resource = client.resource(baseURL);
```

__9. Add these lines to set the content type of the request and the accept header.

```
resource.contentType(MediaType.APPLICATION_XML);
resource.accept(MediaType.APPLICATION_XML);
```

__10. Add this line to send a QuoteRequest XML using a POST request.

```
QuoteResponse res = resource.post(QuoteResponse.class, req);
```

This method will serialize the req object as XML. It will construct a QuoteResponse object from the reply XML.

__11. Finally, add this line to return the quote response.

```
return res;
```

__12. Organize imports and select these classes:

- org.apache.wink.client.Resource
- javax.ws.rs.core.MediaType

__13. Save changes.

Architecture Tip

When JAX-RS 2.0 is released, tools may be able to generate the proxy layer. That will save a lot of coding.

Part 4 - Build the Presentation Layer

This layer will be entirely developed by a client organization (Web Age for the purpose of this lab).

To save time, the HTML coding is already done for us. We will take a quick look at it.

__1. From the **WebContent** folder, open **index.jsp** and study the form.

__2. Open **result.jsp** and study how the QuoteResponse object properties are shown.

Now, we will develop the controller servlet.

__3. From the **com.webage.servlet** package, open **QuoteController.java**.

__4. Locate the **doPost** method.

__5. Below the line:

```
//Compose the QuoteRequest object
```

__6. Add these lines to construct a QuoteRequest object:

```
//Compose the QuoteRequest object
QuoteRequest req = new QuoteRequest();

req.setClientId("C0001");
req.setDateRequiredBy(request.getParameter("dateRequiredBy"));
req.setPartNumber(request.getParameter("partNumber"));
req.setQuantity(Integer.parseInt(request.getParameter("quantity")));
```

__7. Below the line:

```
//Invoke the service
```

__8. Add these lines to invoke the service using the proxy class we created earlier:

```
//Invoke the service
QuoteProxy proxy = new QuoteProxy();

QuoteResponse res = proxy.getQuote(req);
```

__9. Below the line:

```
//Show the result
```

Add these lines to show the response in a JSP

```
//Show the result
request.setAttribute("quoteResponse", res);

request.getRequestDispatcher("/result.jsp").forward(request, response);
```

__10. Organize imports.

__11. Save changes.

Part 5 - Test

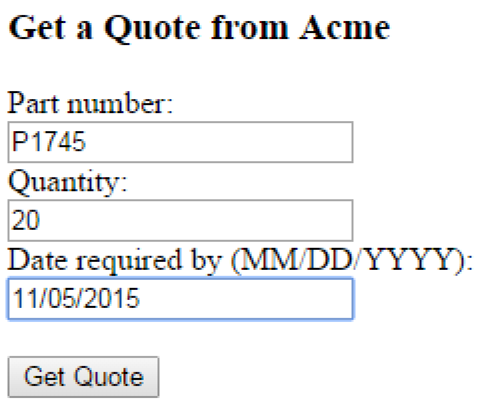
__1. Start the server if it is not running.

__2. Deploy **SimpleClientEAR** project to the server with the **Add and Remove..** option when you right click the server.

__3. Open a web browser and enter the URL:

```
http://localhost:9080/SimpleClient/index.jsp
```

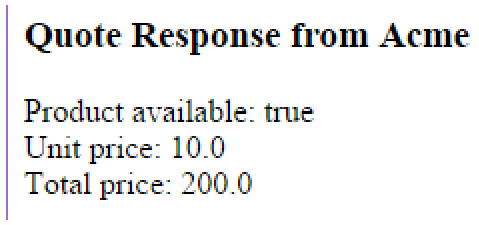
__4. Enter some values and click **Get Quote**.



Get a Quote from Acme

Part number:
P1745
Quantity:
20
Date required by (MM/DD/YYYY):
11/05/2015

__5. Verify the result.



Quote Response from Acme

```
Product available: true
Unit price: 10.0
Total price: 200.0
```

__6. Close the browser.

__7. Close all open files.

Part 6 - Review

In this lab, we developed an application that consumed the quote service. We followed proper architecture by developing a separate proxy layer. The proxy was then used from the presentation layer.

We could have also extended the client application as follows:

1. Create a proxy class for the order service. Add methods to place order and get order details.
2. From the QuoteController servlet class, place an order if the quote response says that product is available.

Lab 25 - Securing RESTful Services

So far, our quote and order services are working just fine. Except that there is no security configured. Anyone in the Internet can invoke these services. Also, we are not doing any kind of audit logging to keep track of which user is doing what. In case of a business dispute, audit logging can resolve many problems.

We will take care of both problems in this lab. We will secure the services so that only users belonging to the "Customer" role can invoke the services. In addition, when an order is placed, we will keep an audit log so that we know which user had placed that order.

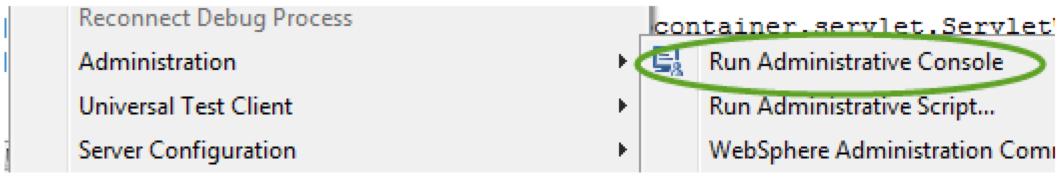
Part 1 - Enable Application Security

Many times security in a WebSphere test environment may be disabled. This is to simplify the local testing of an application. The problem is that with it disabled, even if we put security restrictions in the application they won't be enforced.

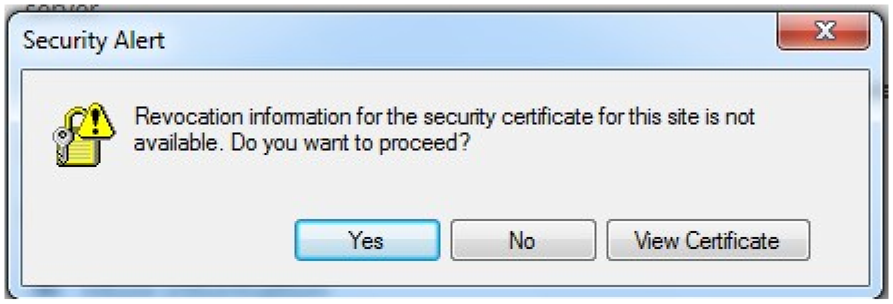
In order to enforce security restrictions we must enable application security in the WebSphere test server.

__1. Make sure the server is running and start it if it is not.

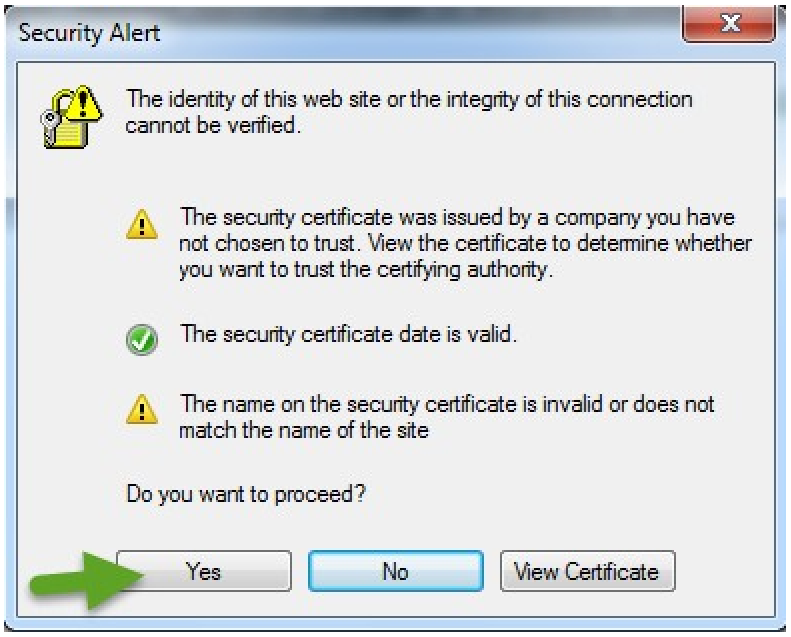
__2. In the **Servers** view, right click the server and select **Administration** → **Run Administrative Console**.


Reconnect Debug Process
Administration
Universal Test Client
Server Configuration
container.servlet.Servlet
Run Administrative Console
Run Administrative Script...
WebSphere Administration Comı

If you receive security alerts they are for the security certificate of the server. Click Yes or OK to accept the certificate and continue.


Security Alert

Revocation information for the security certificate for this site is not available. Do you want to proceed?



Security Alert

The identity of this web site or the integrity of this connection cannot be verified.

The security certificate was issued by a company you have not chosen to trust. View the certificate to determine whether you want to trust the certifying authority.

The security certificate date is valid.

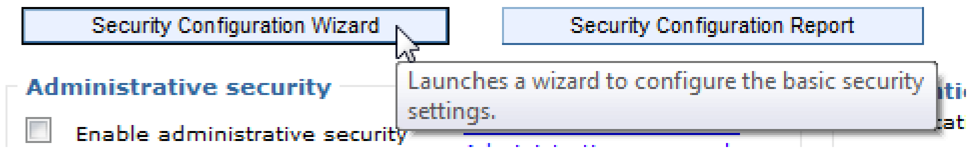
The name on the security certificate is invalid or does not match the name of the site

Do you want to proceed?

__3. Login as wasadmin for user and password if prompt.

__4. In the navigation on the left side, expand the **Security** section and click the link for **Global security**.

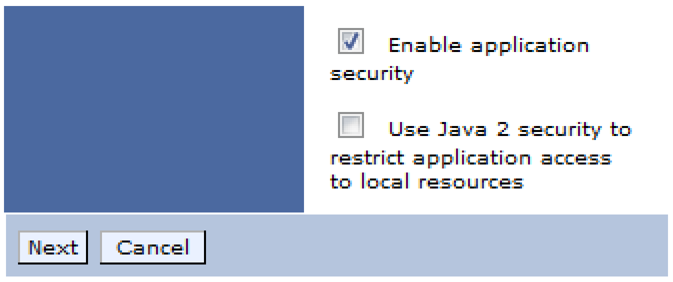
__5. Click the button for **Security Configuration Wizard**.



Security Configuration Wizard
Security Configuration Report
Administrative security
Enable administrative security

Launches a wizard to configure the basic security settings.

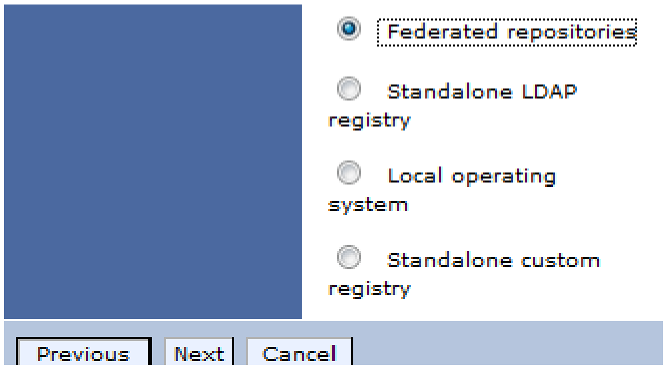
__6. On the first page, check **Enable application security** and click the **Next** button.



☑ Enable application security

☐ Use Java 2 security to restrict application access to local resources

__7. **Switch** the option to **Federated repositories** and click the **Next** button.



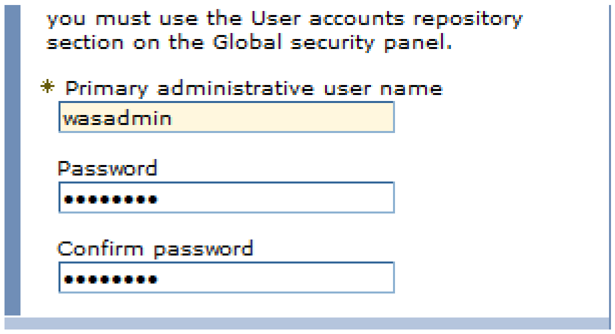
[✓ Federated repositories]

Standalone LDAP registry

Local operating system

Standalone custom registry

__8. Fill in a value of '**wasadmin**' without the quotes for the '**Primary administrative user name**' and both password fields and then click the **Next** button.



you must use the User accounts repository section on the Global security panel.

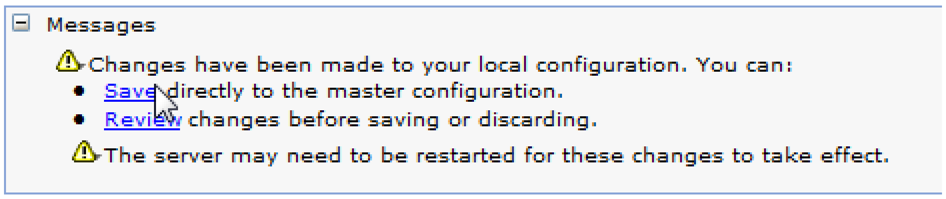
* Primary administrative user name: wasadmin

Password: ••••••••

Confirm password: ••••••••

__9. Click the **Finish** button to complete the security configuration.

__10. In the messages at the top of the Admin Console, click the link to **Save** the changes.



Messages

Changes have been made to your local configuration. You can:

- Save directly to the master configuration.
- Review changes before saving or discarding.

The server may need to be restarted for these changes to take effect.

__11. Click the **Logout** link in the upper right of the Admin Console.

__12. Close the Admin Console window.

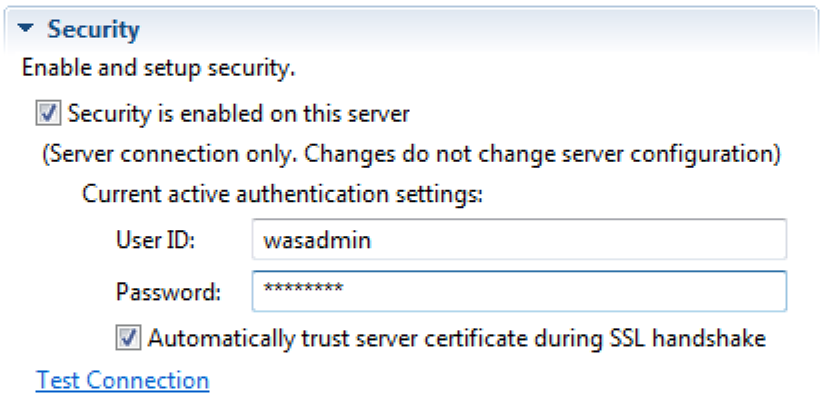
__13. Switch back to the Eclipse **Servers** view.

__14. Stop the server. It may take too long and will need to be Terminated.

__15. Once the server is stopped, double click on the server to open the server properties.

__16. On the right, expand **Security**.

__17. Check the option for '**Security is enabled on this server**' and fill in a value of '**wasadmin**' without the quotes for the **User ID** and **Password** options.


Security
Enable and setup security.
☑ Security is enabled on this server
(Server connection only. Changes do not change server configuration)
Current active authentication settings:
User ID: wasadmin
Password: ********
☑ Automatically trust server certificate during SSL handshake

__18. Save and close the server properties.

__19. Start the WebSphere server from the **Servers** view. If you see errors starting the server (e.g., "Unable to initialize SSL connection"), then stop the server and close Eclipse. After that, start Eclipse again and then start the server. The server should start with no errors. If you have any problems, please contact your instructor. You may request to give privileges to continue.

Part 2 - Secure the Quote Service

There are many ways one can secure a JAX-RS service. The easiest way is to use the annotations available from the javax.annotation.security package. These annotations are also used to secure Servlet, EJB and JSF resources. As a result, this approach will be very consistent throughout your Java EE code.

__1. In the **AcmeWeb** project expand **com.acme.services** package.

__2. Open **QuoteService.java**.

__3. Add the following annotation in bold to the class.

```
@Path("/quotes")
@RolesAllowed({"Customer"})
public class QuoteService {
```

Note the following things:

- You can provide multiple role names. Example: @RolesAllowed({"Customer", "Admin"})
- You can apply the annotation to individual resource methods to setup different security levels.

__4. Organize imports.

__5. Save changes.

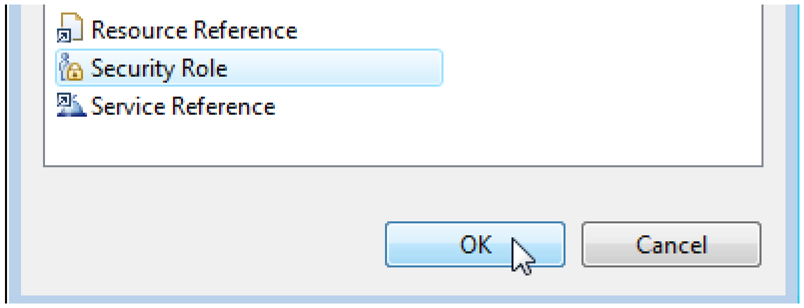
__6. In the **Enterprise Explorer** view, right click the **AcmeApp** EAR project and select **Java EE Tools → Generate Deployment Descriptor Stub**. Make sure it is the AcmeApp project as the AcmeWeb project will have similar options.

__7. In the **Enterprise Explorer** view, expand **AcmeApp → META-INF** and double click the **application.xml** file to open it.

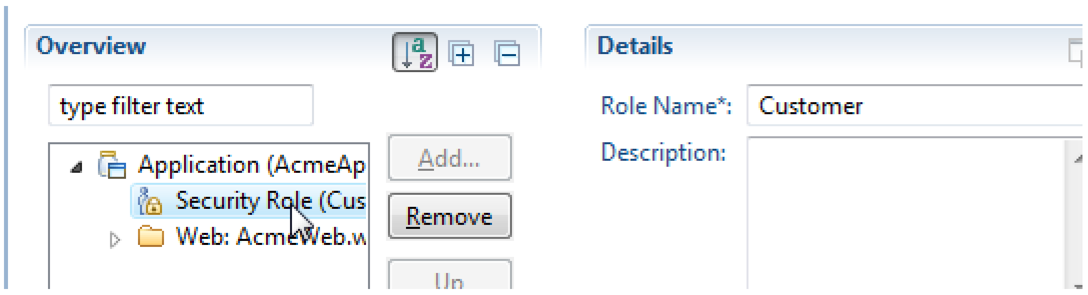
__8. Select the **Design** tab.

__9. In the **Overview** section on the left, click the **Add** button.

__10. Select **Security Role** and click the **OK** button.



__11. With the new security role selected on the left, fill in a **Role Name** of '**Customer**' without the quotes on the right.



Role Name*: Customer

__12. Save and close the file.

Part 3 - Test

__1. Publish the server.

__2. Open a browser and enter the URL:

```
http://localhost:9080/SimpleClient/index.jsp
```

__3. Enter some data and then click **Get Quote**.

__4. The browser will show an error.

Although it is not easy to tell, this error occurs because security is now enabled on the service.

__5. Close the browser.

Part 4 - Run Server with Resources on Server

In the following part, you will make some WebSphere Test Environment configuration changes to avoid a problem that occurs in the next part, when trying to map security roles to user/groups. Specifically, you will change the configuration to run the server with resources on the server.

Note: If you were running a standalone copy of WebSphere, like you would be doing in a staging or production environment, these steps are not required.

__1. Make sure the server is started.

__2. Right click on the server and choose **Add and Remove**.

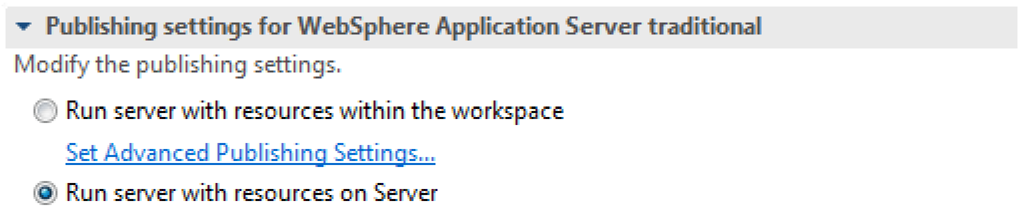
__3. In the **Add and Remove** dialog, click **Remove All** and click **Finish**. Then click **OK** to continue.

__4. Right click the server and select **Clean**. Then click **OK** to continue.

__5. Stop the server.

__6. Double click on the server to open the server configuration editor.

__7. Under **Publishing settings for WebSphere Application Server**, click the **Run server with resources on Server** radio button.


Publishing settings for WebSphere Application Server traditional

Modify the publishing settings.

Run server with resources within the workspace

Set Advanced Publishing Settings...

[✓ Run server with resources on Server]

__8. Save your changes.

__9. Close the file.

__10. Add the **AcmeApp** and **SimpleClientEAR** projects back to the server.

__11. Start the server.

Part 5 - Configure Security at the Server

Now that functions of the service are secured, you must map users or groups to the security roles within the application. This is done administratively.

__1. Make sure the WebSphere server is running and start it if it is not.

__2. Right click the server and select **Administration > Run Administrative Console**.

__3. You may see a security certificate alert, click **Yes** in all the windows to continue.

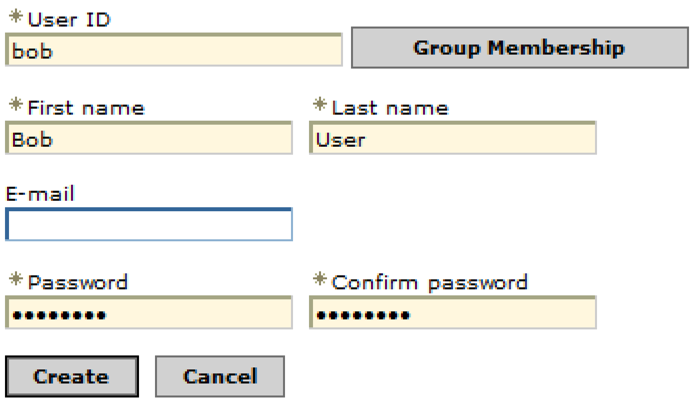
__4. Login to the WebSphere Admin Console with a user ID and password of **wasadmin**.

__5. In the navigation section on the left side of the Admin Console navigate to **Users and Groups** → **Manage Users**.

__6. Type * in the Search for field and click the **Search** button to list the current users.

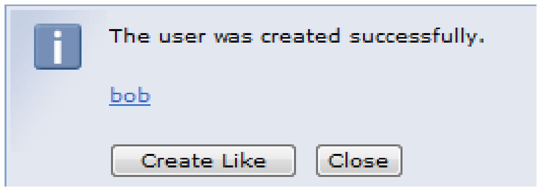
__7. Click the **Create** button and fill in the following user information:

User ID:	bob
First name:	Bob
Last name:	User
Password:	pa55word



__8. When the options are filled in as above click the **Create** button to create the user.

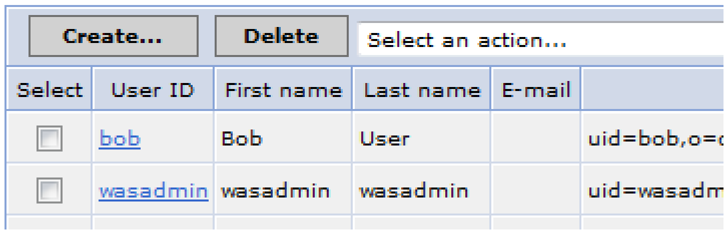
__9. You will get a confirmation message, click **Close**.



The user was created successfully.

bob

The new user will show up in the list.

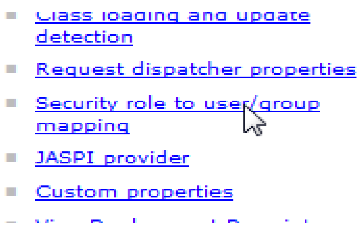


Select	User ID	First name	Last name	E-mail	
☐	bob	Bob	User		uid=bob,o=c
☐	wasadmin	wasadmin	wasadmin		uid=wasadm

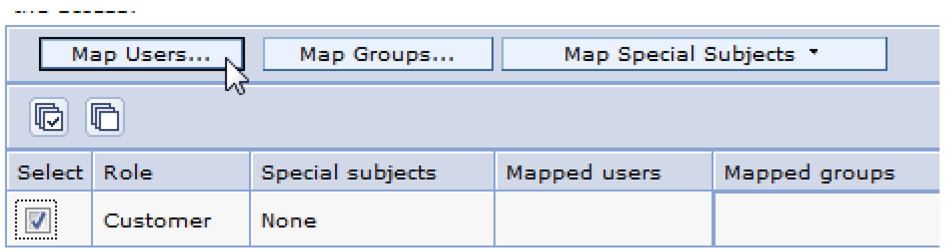
__10. Navigate to **Applications** → **Application Types** → **WebSphere enterprise applications**.

__11. Click the link for the **AcmeApp** application.

__12. Under the **Detail Properties** section, click the link for **Security role to user/group mapping**.


Class loading and update detection
Request dispatcher properties
Security role to user/group mapping
JASPI provider
Custom properties

__13. Check the checkbox next to the **Customer** role and click the **Map Users** button.

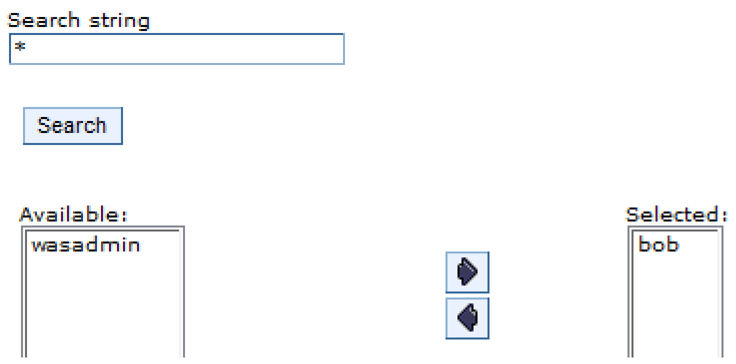


Select	Role	Special subjects	Mapped users	Mapped groups
✓	Customer	None		

Note: In a large application it is more likely that you will be mapping security roles to groups in the user registry rather than individual users. The benefit of this is that if a new user is added that needs to be given access to secure applications all that needs to be done is to add that user into the appropriate group in the user registry. Nothing about the WebSphere configuration would need to change. We are mapping to users since there will only be one used for demonstration purpose.

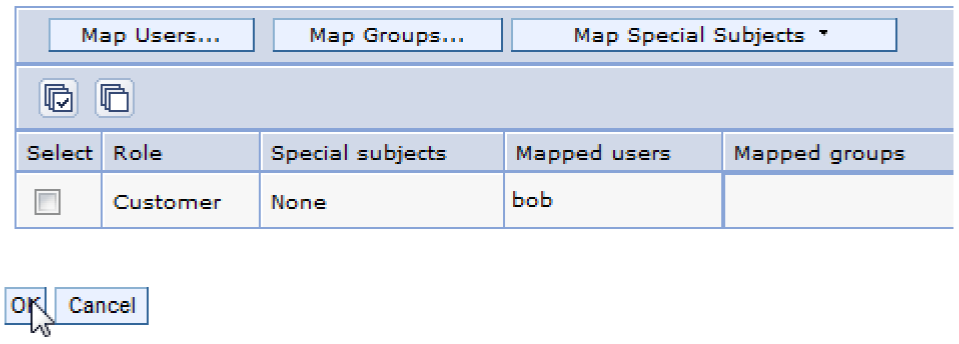
__14. Press the **Search** button to list available users.

__15. Select the **bob** user and press the right arrow button to move the user over to the "Selected" list on the right.



__16. Press the **OK** button on the user search page to accept the user.

__17. Check that the user is listed on the row with the **Customer** role and press the **OK** button.



Select	Role	Special subjects	Mapped users	Mapped groups
☐	Customer	None	bob	

Troubleshooting:

If you don't see the **OK** button, then you won't be able to save this configuration. If this happens, follow these steps:

Log out and close the admin console.

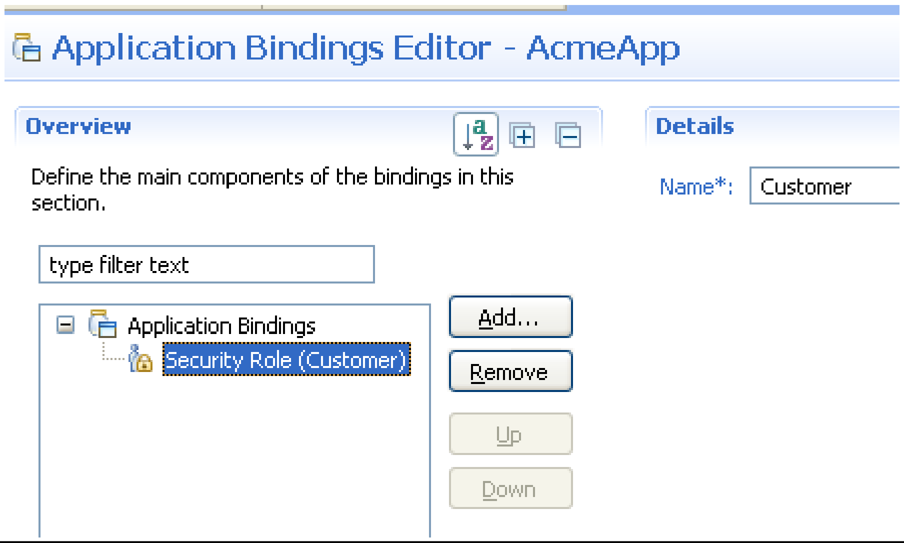
Right click **AcmeApp** and select **Java EE->Generate WebSphere Bindings Deployment Descriptor**.

In the editor, click the **Design** tab.

Make sure **Application Bindings** is selected.

Click the **Add** button, select **Security Role**, and click **OK**.

In the **Name** field, enter 'Customer' (without the quotes).



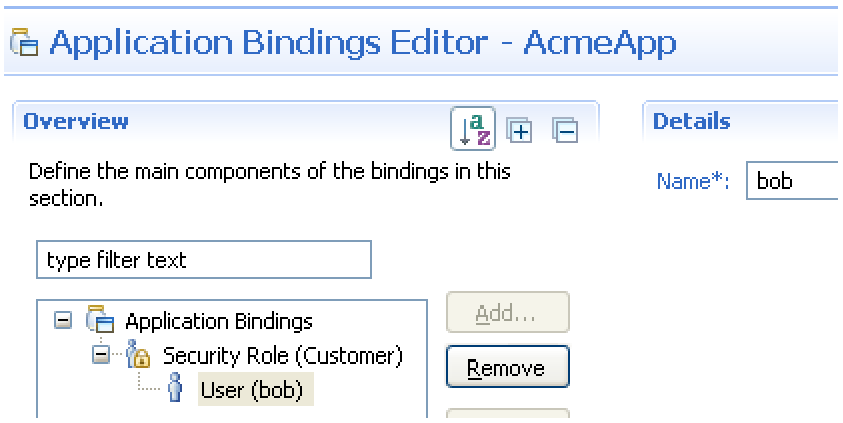
Define the main components of the bindings in this section.

Name*: Customer

Make sure **Security Role (Customer)** is selected.

Click the **Add** button, select **User**, and click **OK**.

In the **Name** field, enter 'bob' (without the quotes).



Save and publish your changes.

Click the **Source** tab in the editor and verify you see the following XML:

```
<security-role name="Customer">
   <user name="bob" />
</security-role>
```

Open the admin console again and review that the mapped user bob is already configured.

Log out and close the console.

Continue to Part 6 of this lab. If you have any problems, contact your instructor.

Note: Even though you hit an **OK** button on the group search page that was only to get back to this page with the new group selected. It is not until you hit the **OK** button on the main page for the security roles that the settings are actually updated.

If after clicking the **OK** button the first time, you see an error at the top of the page, instead of the **Save** link, check the user and click **OK** again.

__18. Click the **Save** link in the messages at the top of the Admin Console.

Note: If you were to remove this application from the server and deploy it another time you would need to come in and configure security settings like the role mapping again.

__19. Click the **Logout** link in the upper right corner of the Admin Console.

__20. Close the Admin Console window.

Part 6 - Modify the Test Client

Clients will now need to supply user ID and password credentials over basic HTTP authentication scheme. Right now the web application that is using the proxy class is using the Apache Wink client API. Ideally we'd like to be able to use code like this:

```
ClientConfig config = new ClientConfig();
BasicAuthSecurityHandler basicAuthSecurityHandler =
          new BasicAuthSecurityHandler();
basicAuthSecurityHandler.setUserName("bob");
basicAuthSecurityHandler.setPassword("pa55word");
config.handlers(basicAuthSecurityHandler);
```

Unfortunately with this API the "BasicAuthSecurityHandler" will only supply user credentials after receiving a 401 response code. The secure REST service will never return this code though. This means that we can't use the Wink client API with our secure service.

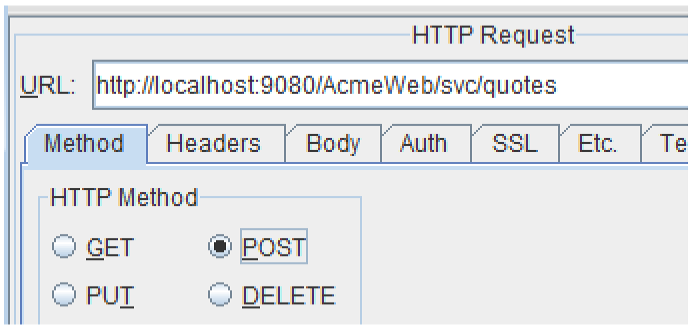
You will use the RESTClient testing tool which can do "pre-emptive" authentication and not wait for a 401 response that will never come.

__1. Launch the RESTClient tool if it is not already running. Use steps from previous labs to set the 'JAVA_HOME' variable in the command prompt to launch the tool.

__2. In the URL text box, enter:

```
http://localhost:9080/AcmeWeb/svc/quotes
```

__3. Select **POST** method.



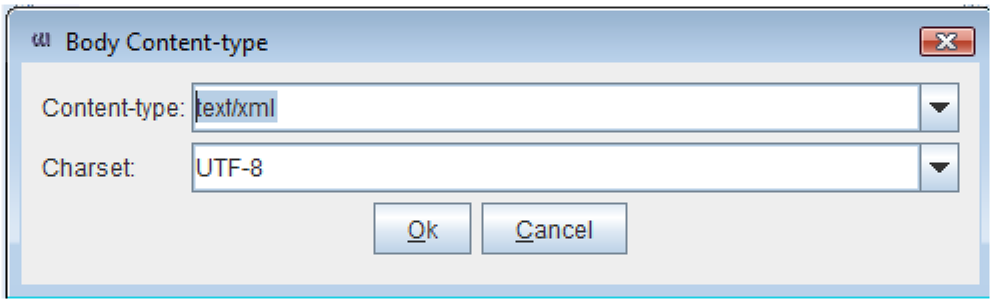
URL: http://localhost:9080/AcmeWeb/svc/quotes

HTTP Method: GET [✓ POST] PUT DELETE

__4. Click the **Body** tab.

__5. Click the icon.

__6. From the drop-down select the **text/xml** content type.



Body Content-type

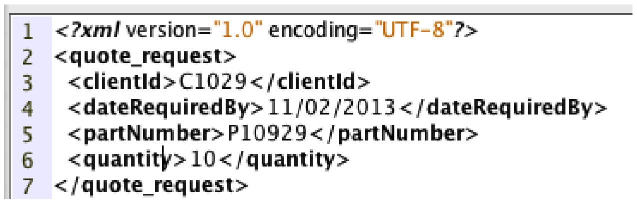
Content-type: text/xml

Charset: UTF-8

__7. Click **OK**.

__8. Click the icon to open the file. Click Yes to erase the existing code.

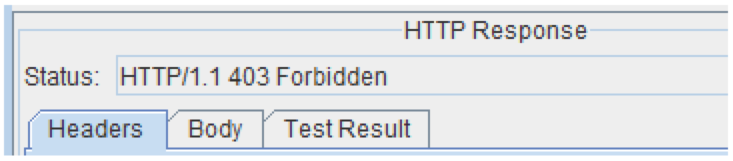
__9. Select **C:\LabFiles\quote_request.xml** and open it. The tool will offer you an option to change the MIME type to application/xml. This is not necessary (either text/xml or application/xml will work fine), click **NO**.



```
<?xml version="1.0" encoding="UTF-8"?>
<quote_request>
  <clientId>C1029</clientId>
  <dateRequiredBy>11/02/2013</dateRequiredBy>
  <partNumber>P10929</partNumber>
  <quantity>10</quantity>
</quote_request>
```

__10. Click the icon next to the URL.

__11. Make sure that the HTTP reply status code is 403.



HTTP Response

Status: HTTP/1.1 403 Forbidden

__12. Click the **Auth** tab.

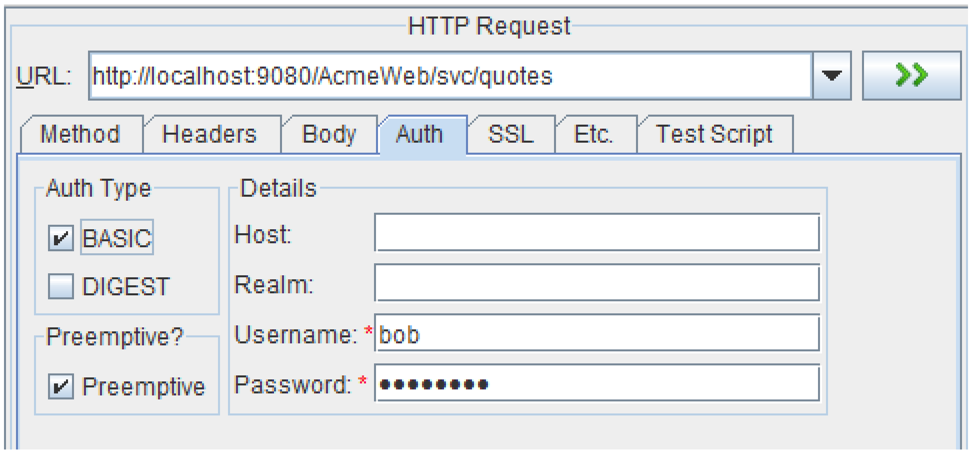
__13. Enter these values:

Check **BASIC**.

Username: bob

Password: pa55word

Check **Preemptive**. This way, the test too will send the credentials with the first request. Otherwise, RESTClient will wait for the first request to fail with a 401 status code and then retry the request with credentials. Since WebSphere returns a 403 status code, that scheme will not work.



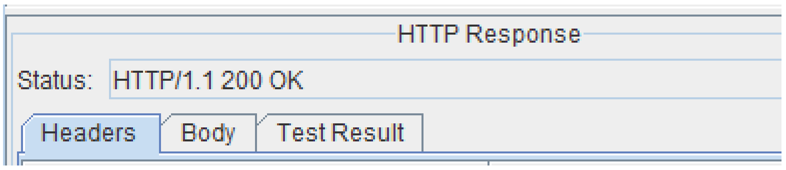
URL: http://localhost:9080/AcmeWeb/svc/quotes

[✓] BASIC
[ ] DIGEST
[✓] Preemptive

Host:
Realm:
Username: * bob
Password: * ●●●●●●●●●

__14. Run the test again.

__15. Verify that status code 200 is returned.



Status: HTTP/1.1 200 OK

Part 7 - Enable Audit Logging

We will now protect the order service and add audit logging support. The audit log will show the name of the user that is placing the order.

__1. In the **AcmeWeb** project, open **OrderService.java**.

__2. Configure the valid role for the service as shown in bold face below.

```
@Path("/orders")
@RolesAllowed({"Customer"})
public class OrderService {
```

__3. Add a member variable to the class as follows.

```
public class OrderService {
          @Context
          SecurityContext sec;
```

This javax.ws.rs.core.SecurityContext object will give us access to the user name.

Note: There is already a @Context annotation for the UriInfo object. You need to add another @Context annotation before the SecurityInfo object that you added in this step. The @Context annotation applies to only one injection, not multiple injections.

```
public class OrderService {

          @Context

          SecurityContext sec;


          @Context

          UriInfo ui;
```

Note that if you don't add a separate @Context annotation for the SecurityContext object, then the existing annotation will only apply to the SecurityContext object. That means that the JAX-RS runtime will only inject the SecurityContext object inside the resource class and not the UriInfo object. Because the UriInfo object was not injected, ui will be null, which will lead to a NullPointerException when trying to construct the order URL.

__4. In the **placeOrder**() method, below the line:

```
long orderId = dao.createOrder(o);
```

__5. Add:

```
logger.info("Order " + orderId +
    " created by: " + sec.getUserPrincipal());
```

__6. Organize import and select this class:

- javax.ws.rs.core.SecurityContext

__7. Save changes.

__8. Publish the server.

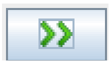
Part 8 - Test

__1. Launch the REST test client if it is not already running.

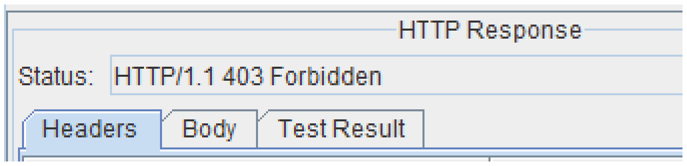
__2. Select **File** > **Open Request** from the menu.

__3. Open **C:\LabFiles\order_create.rcq**

Right now, no security credential is entered in the request. Let's test for authentication failure.

__4. Click the  button to send the request.

__5. Make sure that a status code 403 is returned.



Status: HTTP/1.1 403 Forbidden

Now we will configure credentials.

__6. Click the **Auth** tab.

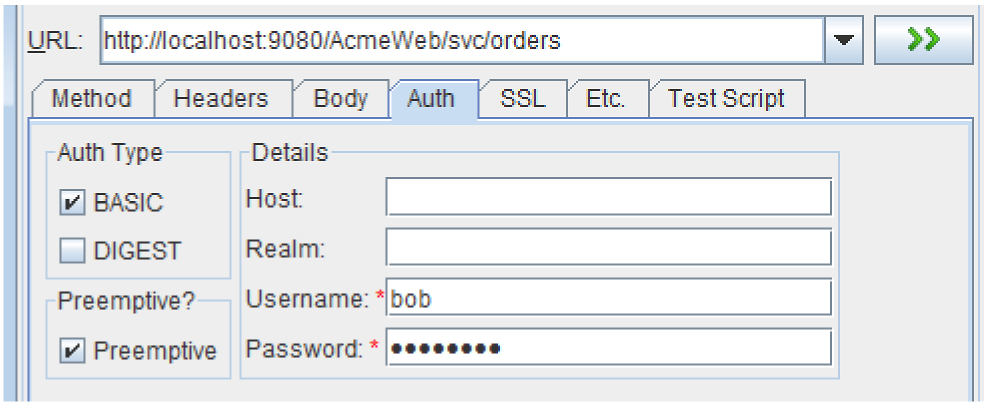
__7. Enter these values:

Check **BASIC**.

Username: bob

Password: pa55word

Check **Preemptive**.



URL: http://localhost:9080/AcmeWeb/svc/orders

Auth Type: [✓ BASIC] DIGEST

Preemptive? [✓ Preemptive]

Host:

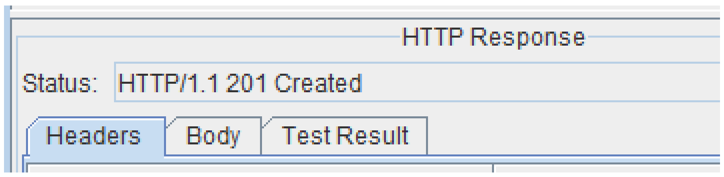
Realm:

Username: * bob

Password: * ●●●●●●●●

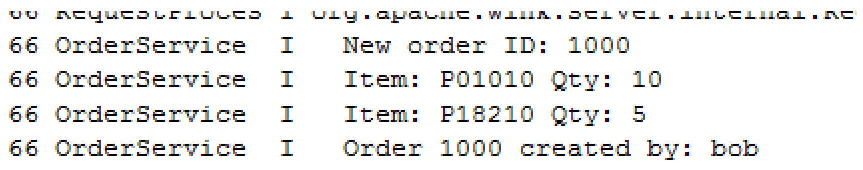
__8. Run the test again.

__9. Verify that status code 201 is returned.



Status: HTTP/1.1 201 Created

__10. Check Eclipse console for the audit log. It will look like this:



```
66 OrderService  I   New order ID: 1000
66 OrderService  I   Item: P01010 Qty: 10
66 OrderService  I   Item: P18210 Qty: 5
66 OrderService  I   Order 1000 created by: bob
```

If you an Unable to initialize SSL Connection in the console, scroll up to find the order messages.

__11. Close all open browsers.

__12. Close all open files.

__13. Close REST client.

__14. Stop the server.

Part 9 - Review

In this lab, we learned how to enable role based protection of service resources. We protected the OrderService and QuoteService classes using the @RolesAllowed annotation. Optionally, you can configure the roles for individual methods.

We also learned how to perform audit logging.