WA2103 Programming Java SOAP Web Services with JAX-WS - WebSphere 8 / RAD 8

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - 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 RAD 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**.

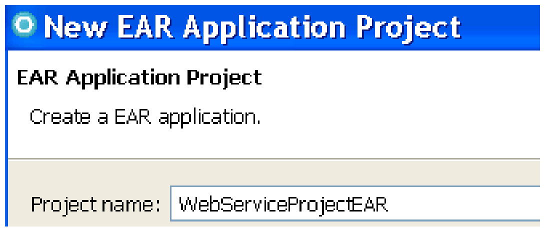
The *New Dynamic Web Project* screen will appear.

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

__3. At the bottom of the window, in the *EAR Membership* section, click the **New Project....** button (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**


New EAR Application Project

EAR Application Project

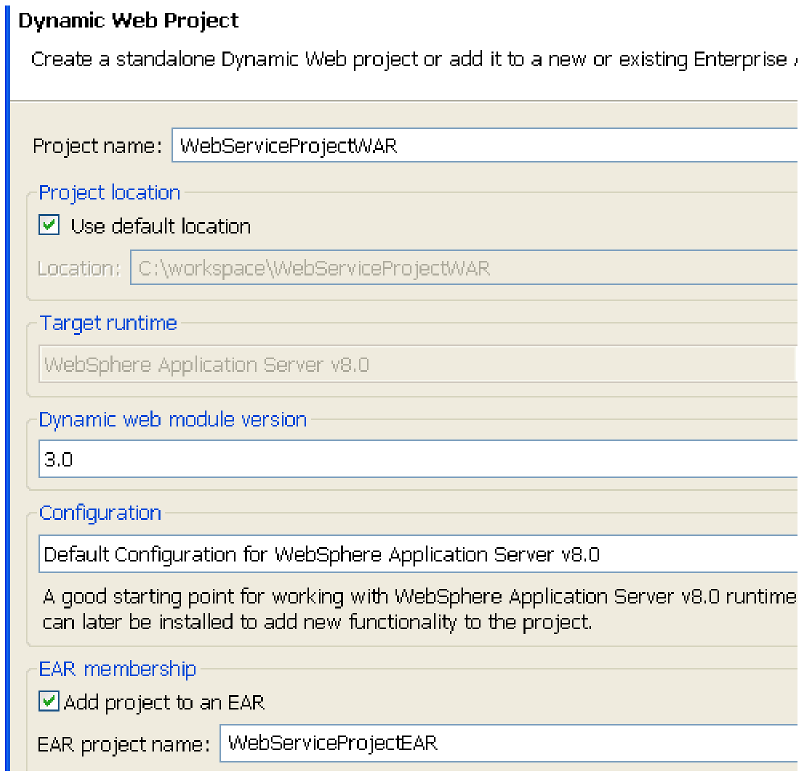
Create a EAR application.

Project name: WebServiceProjectEAR

__5. Click **Finish**.

You will be returned to the *New Dynamic Web Project* screen.

Your screen should now look like the following:


Dynamic Web Project

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

Project name: WebServiceProjectWAR

Project location

☑ Use default location

Location: C:\workspace\WebServiceProjectWAR

Target runtime

WebSphere Application Server v8.0

Dynamic web module version

3.0

Configuration

Default Configuration for WebSphere Application Server v8.0

A good starting point for working with WebSphere Application Server v8.0 runtime
can later be installed to add new functionality to the project.

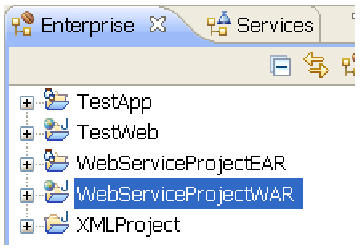
EAR membership

☑ Add project to an EAR

EAR project name: WebServiceProjectEAR

__6. Click **Finish**. Both the **EAR** and the **WAR** will be created.

RAD will displayed them in the *Enterprise Explorer* and will also open a *Technology Quickstarts* view in the editor area.



__7. Close the *Technology Quickstarts* view. You will not need it.

__8. Start the Server. Wait until finish publishing.

__9. Right click on the server and select **Add and Remove Projects..**

__10. Click **Add All**.

__11. Click **Finish**.

__12. Right click the Server and select **Publish**.

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

Note. Be sure your server is started and the project has been deployed in the server before continue; this steps are important because there is a bug in RAD that doesn't update the Service soap address if the server is not started.

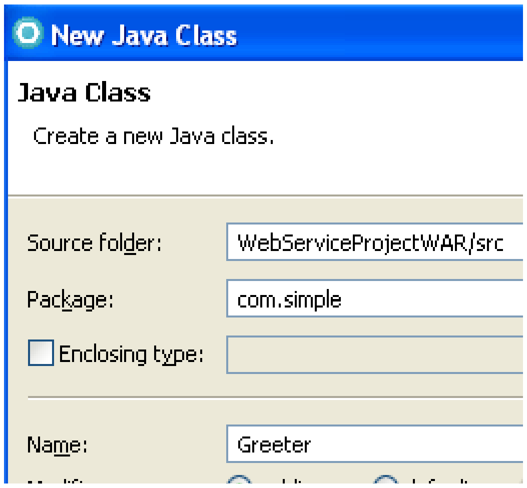
Part 2 - Create the Implementation Class

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).



Java Class

Create a new Java class.

Source folder: WebServiceProjectWAR/src

Package: com.simple

Enclosing type:

Name: Greeter

__4. Click **Finish**.

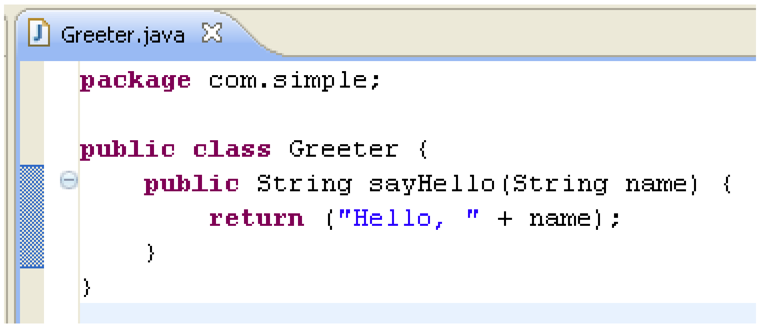
An editor will open on the newly created source file.

__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: RAD 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

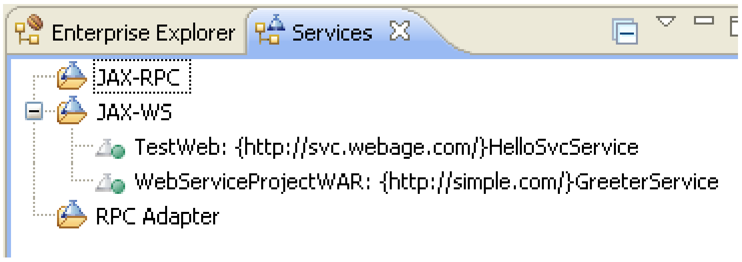
__1. Open the *Servers* view.

__2. Right click the server and select **Publish**.

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. This will open the *Services* view.

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



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

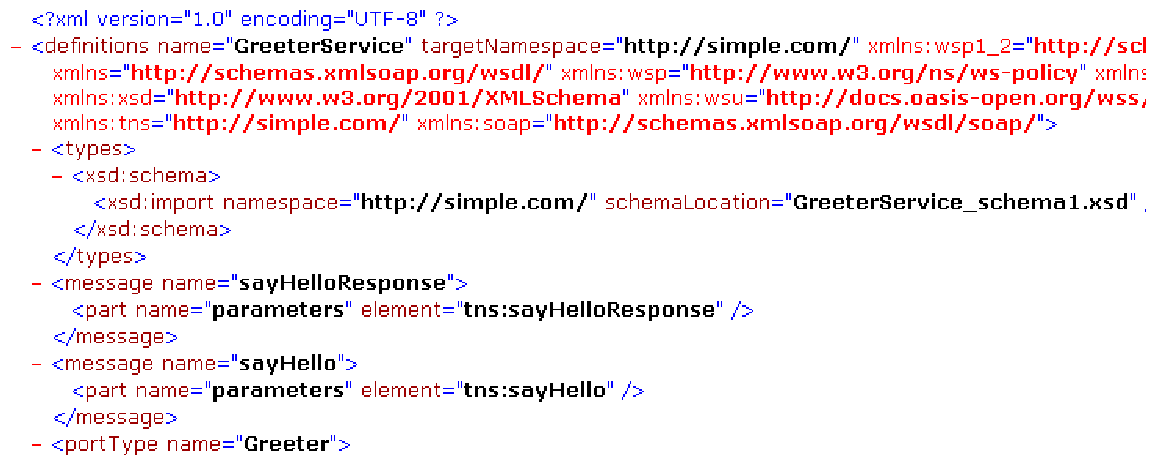
Notice that two services are listed. One is the **HelloSvcService** service from the **TestWeb** project (which we examined in Lab 1), and the other is **GreeterService** we have just created, in the **WebServiceProjectWAR** project. RAD has automatically read our annotated code and is aware they are services.

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

__5. Open a web browser and navigate to:

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

The browser should show the WSDL file for the service.



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <definitions name="GreeterService" targetNamespace="http://simple.com/" xmlns:wsp1_2="http://scl
    xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsu="http://docs.oasis-open.org/wss,
    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 in Lab 1.

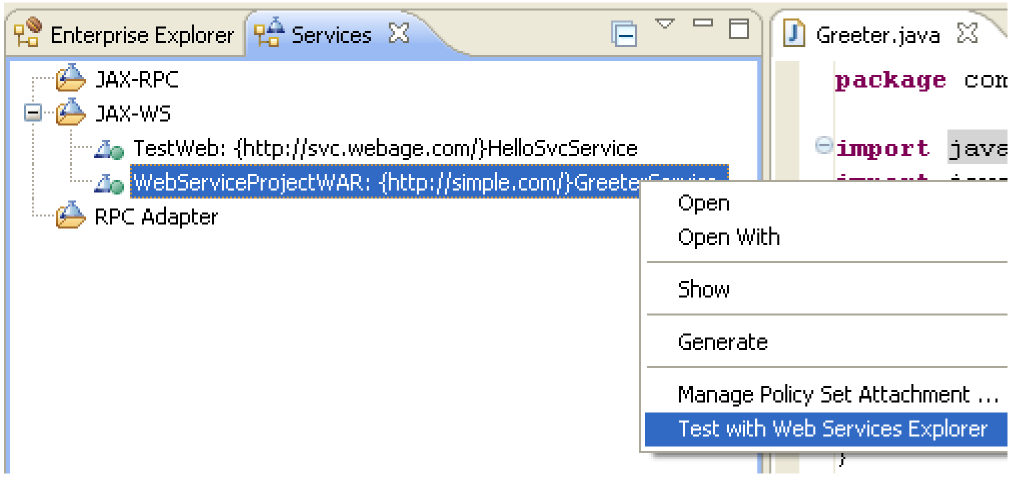
__6. In the *Services* view, right-click on the **WebServiceProjectWAR** service and select **Show > WSDL Interface**.

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

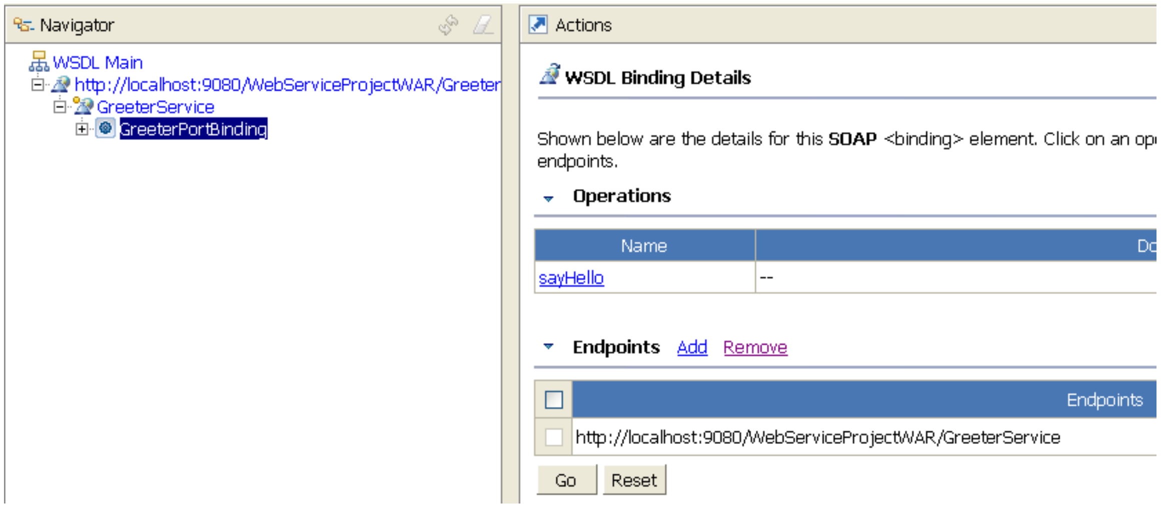
__8. Scroll down to the bottom of the code and verify the soap address has been updated to the service URL. If it is not showing the above URL continue with the steps, there is a note about this later (Step 10).

```
<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**.


Enterprise Explorer
Services
Greeter.java
JAX-RPC
JAX-WS
TestWeb: {http://svc.webage.com/}HelloSvcService
WebServiceProjectWAR: {http://simple.com/}Greete
RPC Adapter
package com
import java
Open
Open With
Show
Generate
Manage Policy Set Attachment ...
Test with Web Services Explorer

The WSE will open.



Shown below are the details for this SOAP <binding> element. Click on an op
endpoints.

Operations

Name	Do
sayHello	--

Endpoints Add Remove

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

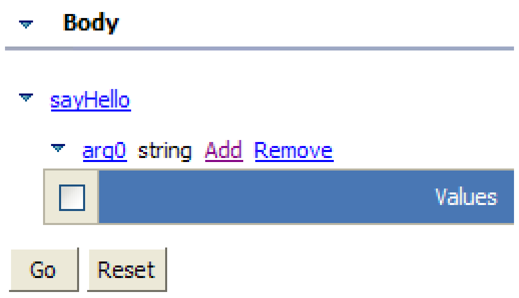
__10. Verify the Endpoint is configured as shown above. If not then close all open files, stop the server, start the server and run the service again. If the endpoint is not updated then you will need to create a new workspace and perform this Lab again. Ask your instructor for help, there is the solution for previous Lab in C:\LabFiles\Solutionsthat 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.

__11. Look in the *Actions* pane, on the right, click in the **sayHello** operation.

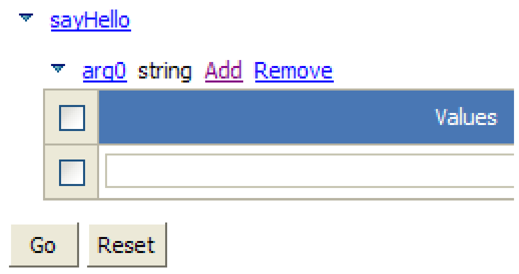
The *Actions* pane changes to display possible inputs to the service.

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.

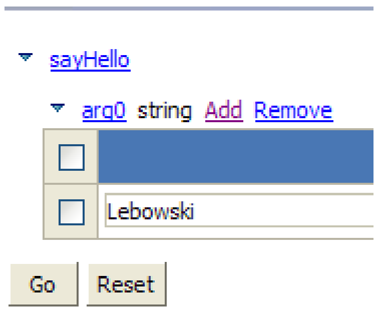


Why is this happening? This is because the WSDL file that was automatically generated did not specify that a parameter *must* be there.

__12. For now, however, we can still test our service. Click the **Add** link, and a field will appear.

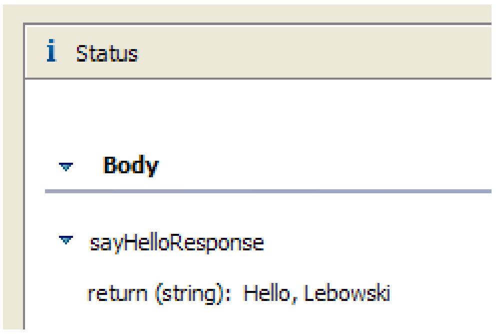


__13. Enter a name into the field.



__14. Click **Go**.

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



```
return (string): Hello, Lebowski
```

(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)

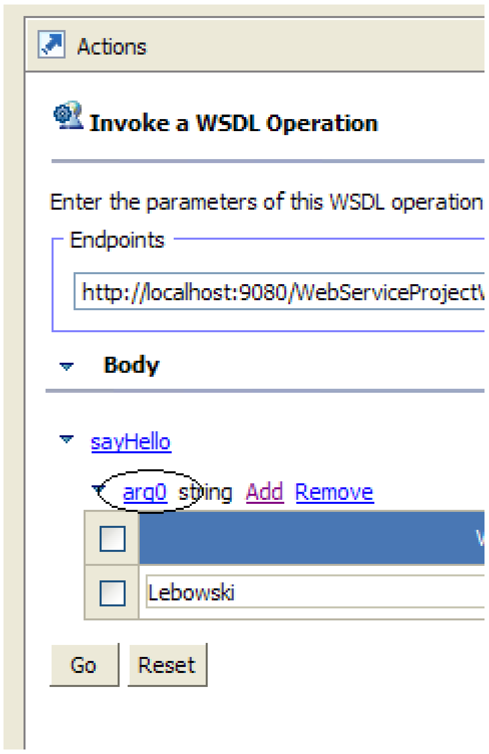
Our web service has been invoked successfully!

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.



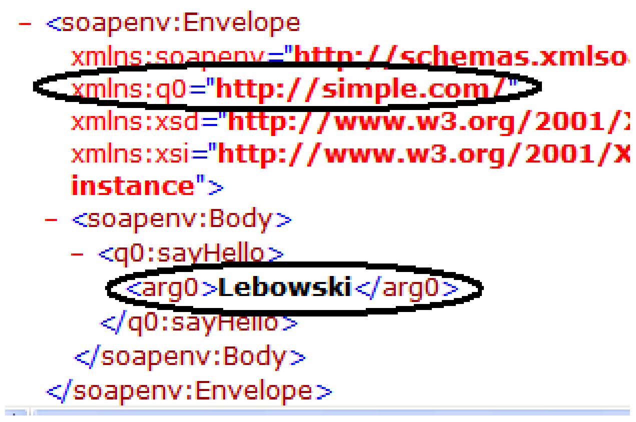
Notice that the name of the parameter is **arg0**. What exactly is the impact of this? To see, we have to look at the underlying SOAP messages. To be precise, we want to take a look at the SOAP request message that the WSE generated and was submitted to the service for processing.

__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.

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.



```
- <soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlso
    xmlns:q0="http://simple.com/"
    xmlns:xsd="http://www.w3.org/2001/
    xmlns:xsi="http://www.w3.org/2001/X
    instance">
  - <soapenv:Body>
    - <q0:sayHello>
        <arg0>Lebowski</arg0>
      </q0:sayHello>
    </soapenv:Body>
  </soapenv:Envelope>
```

Examine the *SOAP Request Envelope*. This is the top pane. We want to note two things here:

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.

__5. Return the *Status* pane to its normal size by double clicking on it again.

__6. Switch back to the *Enterprise Explorer* view and open **Greeter.java** in the editor again (under **com.simple** package).

__7. 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**

__8. 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.

__9. Organize imports. There should be no errors.

__10. Save the file.

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

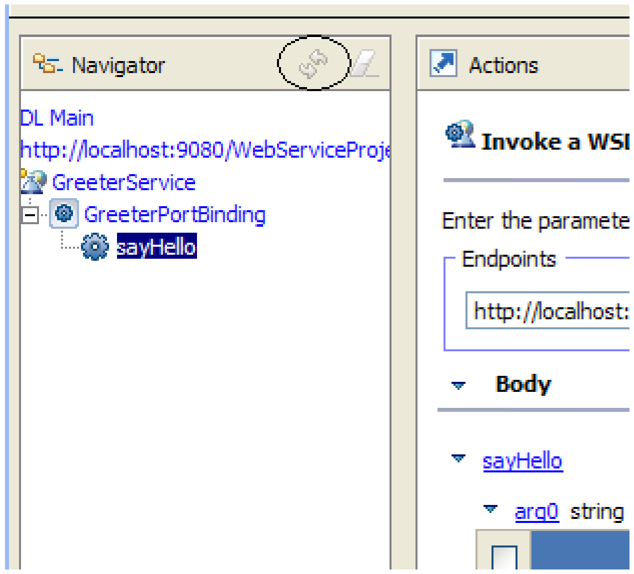
__11. We now need to push these latest code changes out to the server. Locate the *Servers* view and ***Publish*** the server by right-clicking on it and selecting **Publish**.

Part 6 - Examine the Updated Service

We should now see if the annotations worked.

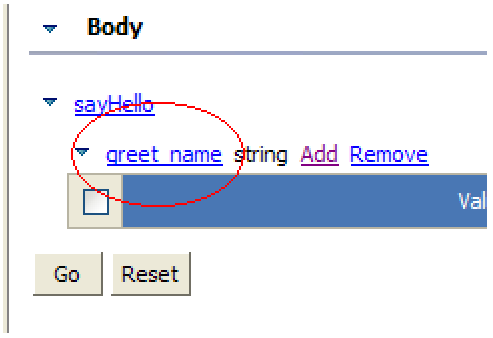
__1. Go back to the WSE which should still be pointing at the GreeterService.

__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.

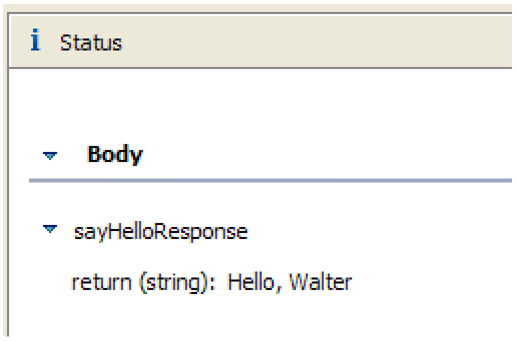
The WSE updates:



__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**.

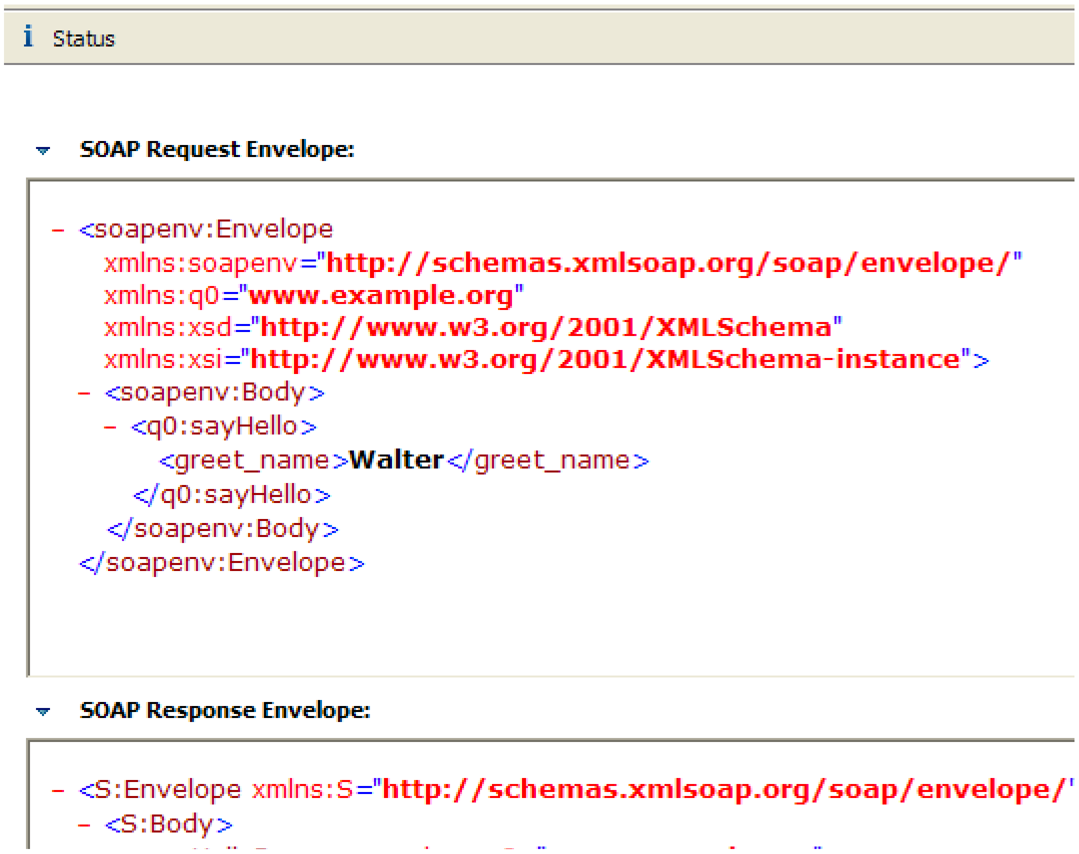
The response will come back as before.



```
return (string): Hello, Walter
```

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

__6. Click the **Source** link in the *Status* pane, and then maximize the *Status* pane.



```
- <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>Walter</greet_name>
      </q0:sayHello>
    </soapenv:Body>
  </soapenv:Envelope>
```

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

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**.

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

__7. Stop the server.

__8. Close all open editor windows, including the WSE.

__9. Close the web browser that you opened.

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 2 - 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 **WebSphere Application 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. Stop the server.

__5. We will now import the given projects and study before apply any security setting.

__6. From the menu select **File > Import**.

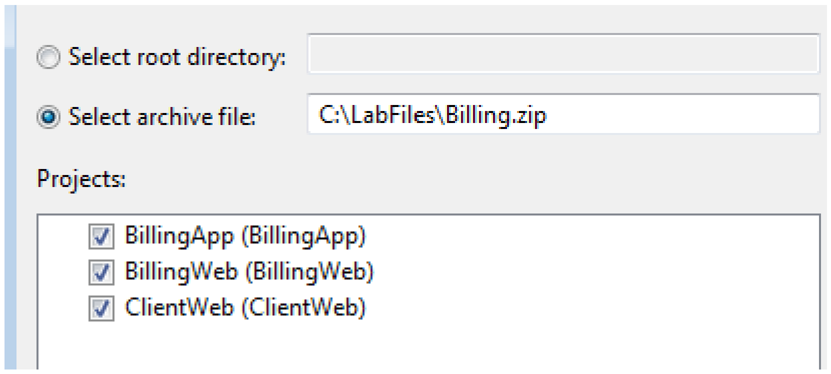
__7. Expand **General** and select **Existing Projects into Workspace**.

__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:

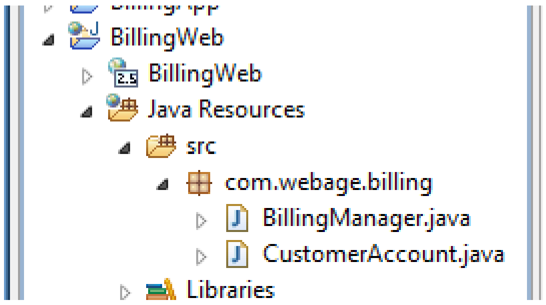
- [✓] BillingApp (BillingApp)
- [✓] BillingWeb (BillingWeb)
- [✓] ClientWeb (ClientWeb)

__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.

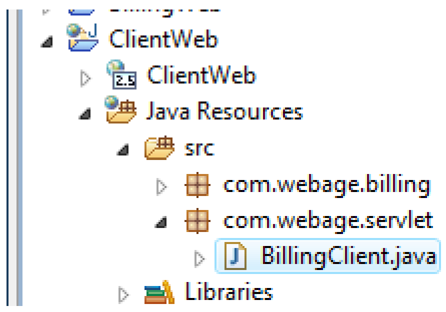
__18. Note that the addAccount operation simply prints out a few lines in the log file and finally returns "OK".

__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 RAD. 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.

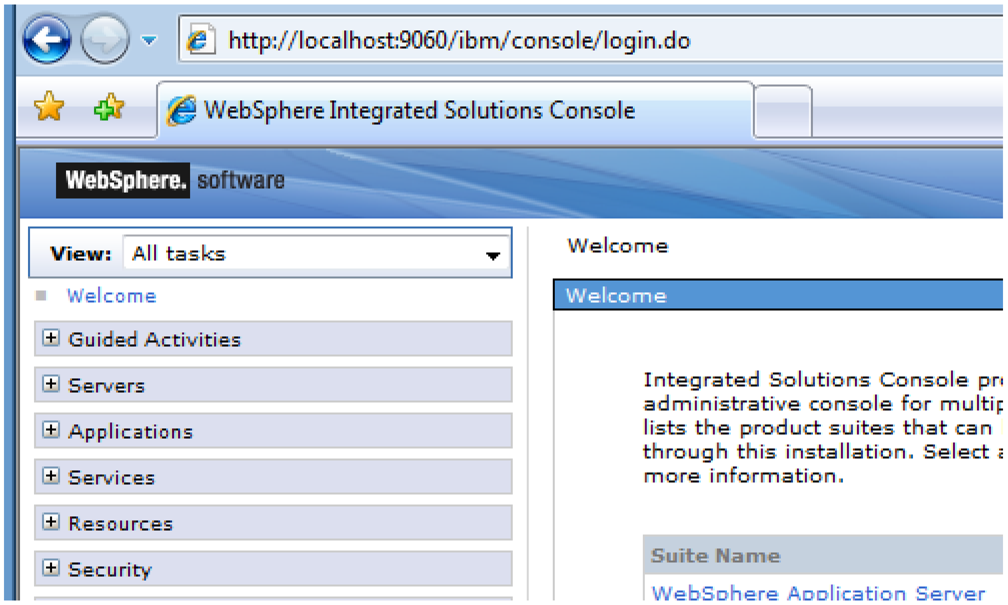
__3. Launch a browser window like IE or FireFox.

__4. Enter the URL:

```
http://localhost:9060/ibm/console/
```

__5. Enter anything as user ID (wasadmin) and log in.

The admin console will open.



http://localhost:9060/ibm/console/login.do

WebSphere Integrated Solutions Console

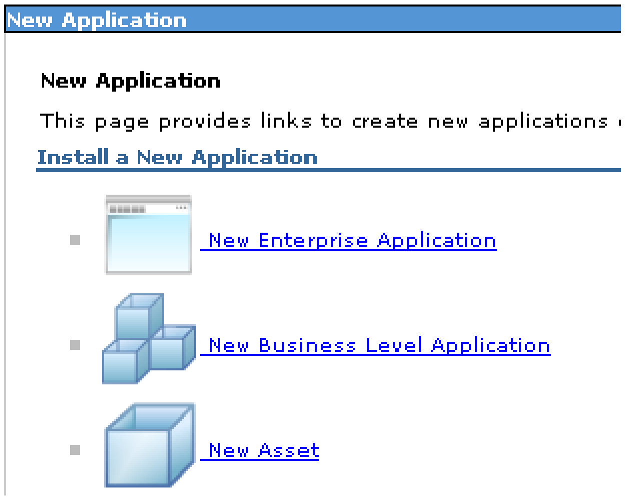
Integrated Solutions Console pr
administrative console for multip
lists the product suites that can
through this installation. Select a
more information.

Suite Name
WebSphere Application Server

__6. Expand **Applications**.

__7. Click **New Application**.

__8. Click **New Enterprise Application**.



New Application

New Application

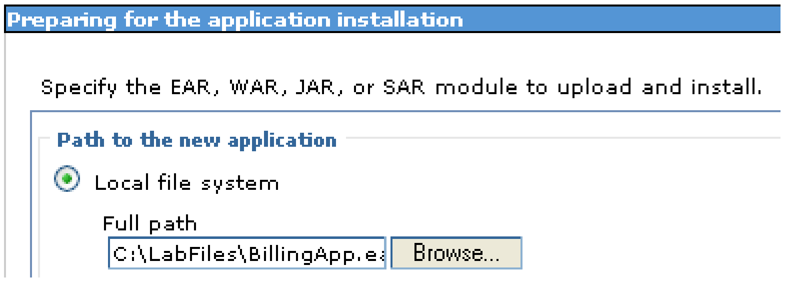
This page provides links to create new applications

Install a New Application
New Enterprise Application
New Business Level Application
New Asset

__9. Make sure that **Local file system** is selected.

__10. Click **Browse**.

__11. 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

C:\LabFiles\BillingApp.ea

__12. Click **Next**.

__13. Leave the default in the **How do you want to install the application?** page and click **Next**.

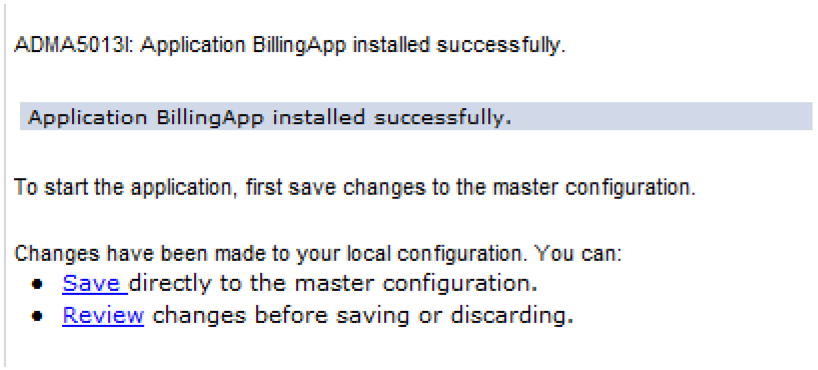
__14. In the **Step1: Select installation options** page just click **Next**.

__15. In the **Step 2: Map modules to servers** page, again just click **Next**.

__16. In the **Step 3: Metadata for modules** page, again just click **Next**.

__17. Click **Finish**.

__18. 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.

__19. Click **Save**.

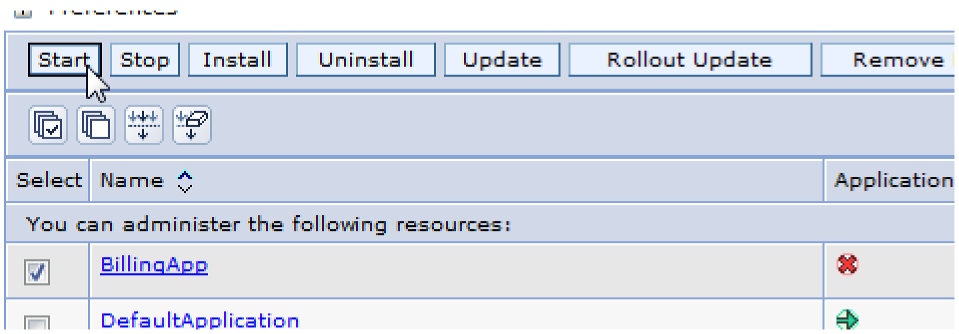
Part 3 - Test the Application

__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.



You can administer the following resources:

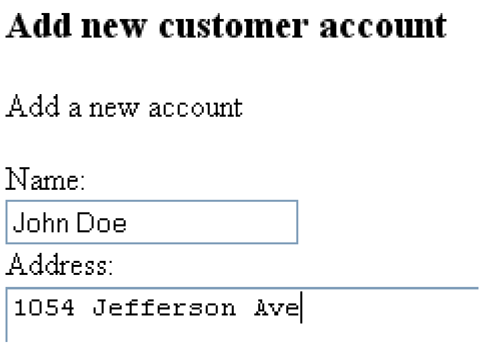
Select	Name	Application
✓	BillingApp	✗
	DefaultApplication	➔

__5. Launch a new web browser window.

__6. Enter the URL:

```
http://localhost:9080/ClientWeb/index.jsp
```

__7. The page will be displayed. Add some input as shown below.



Add new customer account

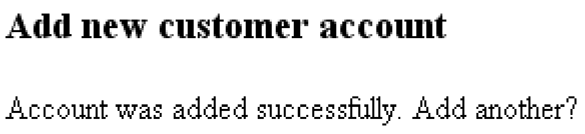
Add a new account

Name:
John Doe

Address:
1054 Jefferson Ave

__8. Click **Add**.

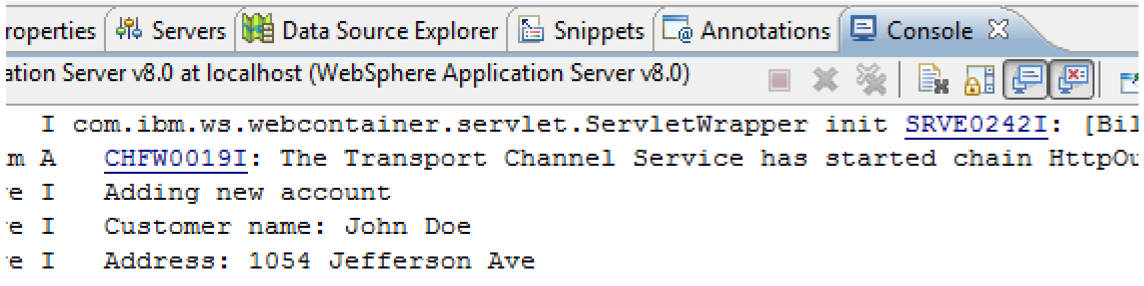
__9. The JSP will call the web service. Make sure that the page shows the success message:



Add new customer account

Account was added successfully. Add another?

__10. Back in RAD, the **Console** view will show the log output from the web service.



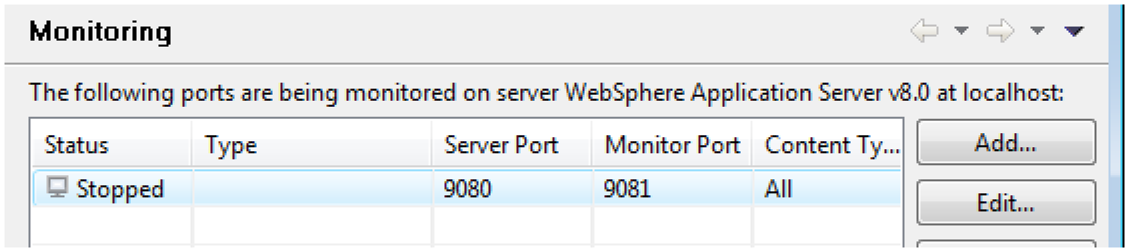
```
ation Server v8.0 at localhost (WebSphere Application Server v8.0)
   I com.ibm.ws.webcontainer.servlet.ServletWrapper init SRVE0242I: [Bi]
m A   CHFW0019I: The Transport Channel Service has started chain HttpOu
e I   Adding new account
e I   Customer name: John Doe
e I   Address: 1054 Jefferson Ave
```

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.

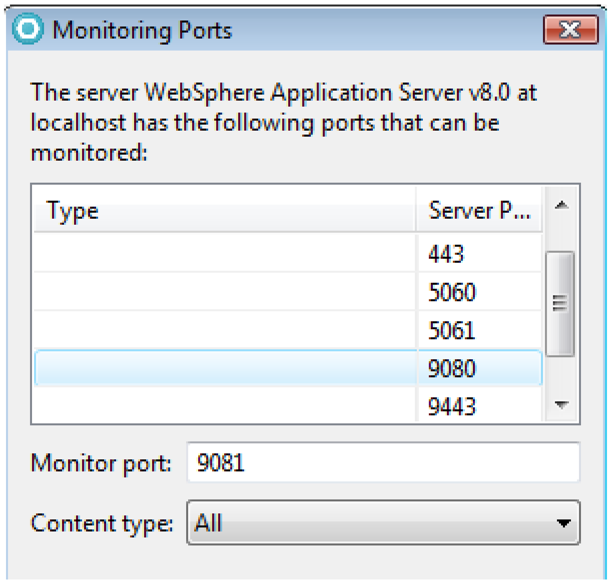


The following ports are being monitored on server WebSphere Application Server v8.0 at localhost:

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

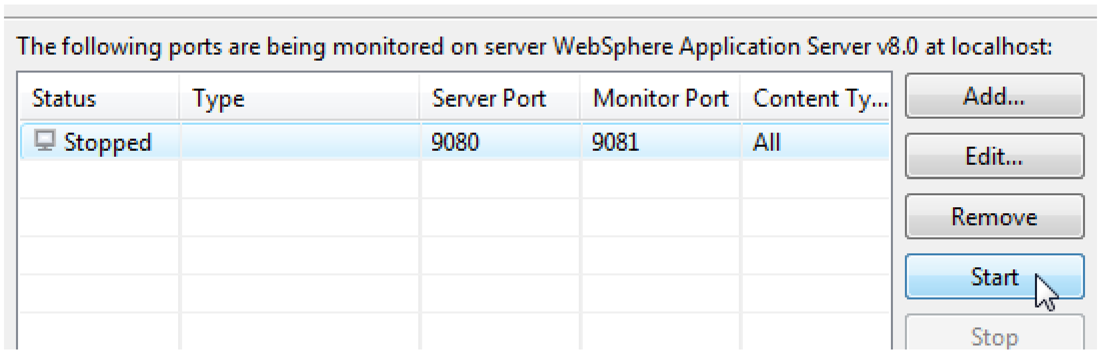
Monitor port: 9081

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 **OK** to close the monitoring properties.

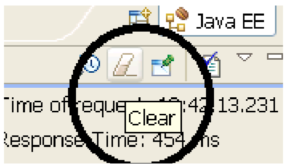
__8. Select **Window > Show View > Other**.

__9. Expand **Debug** and select **TCP/IP Monitor**.

__10. Click **OK**.

__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.

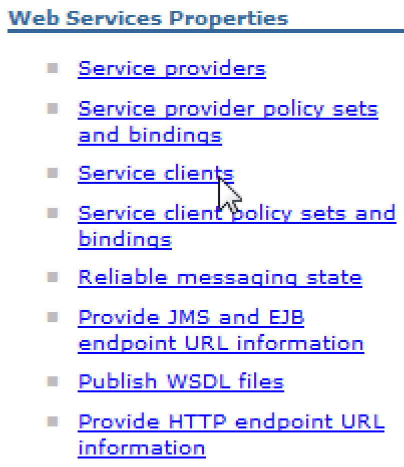
__13. If you don't have one open already, open a web browser to the Admin Console URL and login with any user:

```
http://localhost:9060/ibm/console/
```

__14. In the Admin Console, navigate to **Applications** → **Application Types** → **WebSphere enterprise applications**.

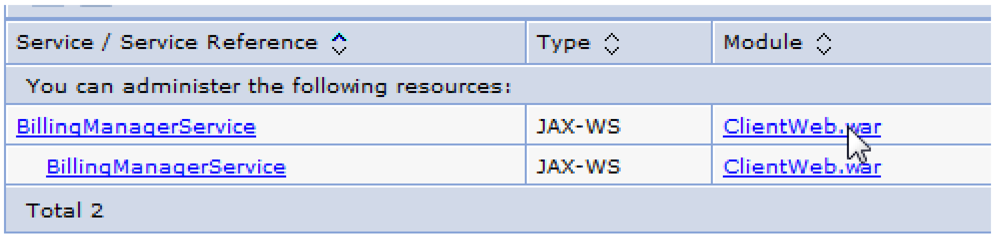
__15. Click the link for the **BillingApp** application.

__16. Scroll down and in the list of 'Web Services Properties' click the link for **Service clients**.


Web Services Properties

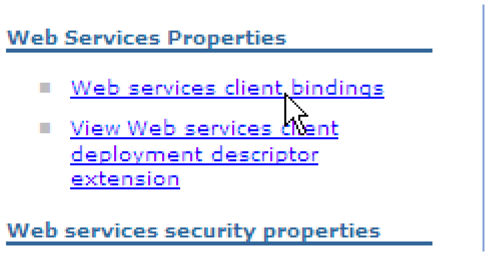
- Service providers
- Service provider policy sets and bindings
- Service clients
- Service client policy sets and bindings
- Reliable messaging state
- Provide JMS and EJB endpoint URL information
- Publish WSDL files
- Provide HTTP endpoint URL information

__17. 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		

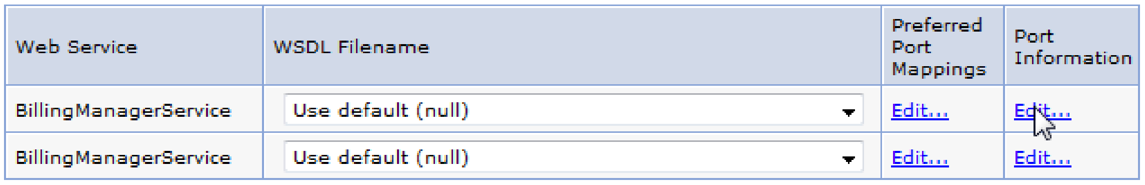
__18. Under 'Web Services Properties' click the link for **Web services client bindings**.


Web Services Properties

- Web services client bindings
- View Web services client deployment descriptor extension

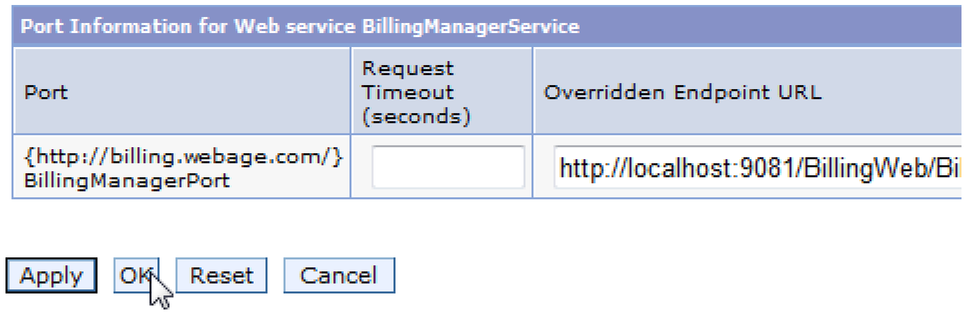
Web services security properties

__19. 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...

__20. 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

__21. Click the **Save** link in the messages at the top of the Admin Console.

__22. Click the **Logout** link at the top of the Admin Console and close it.

__23. Go back to the consumer web page (http://localhost:9080/ClientWeb/index.jsp).

__24. Enter some input data and click **Add**.

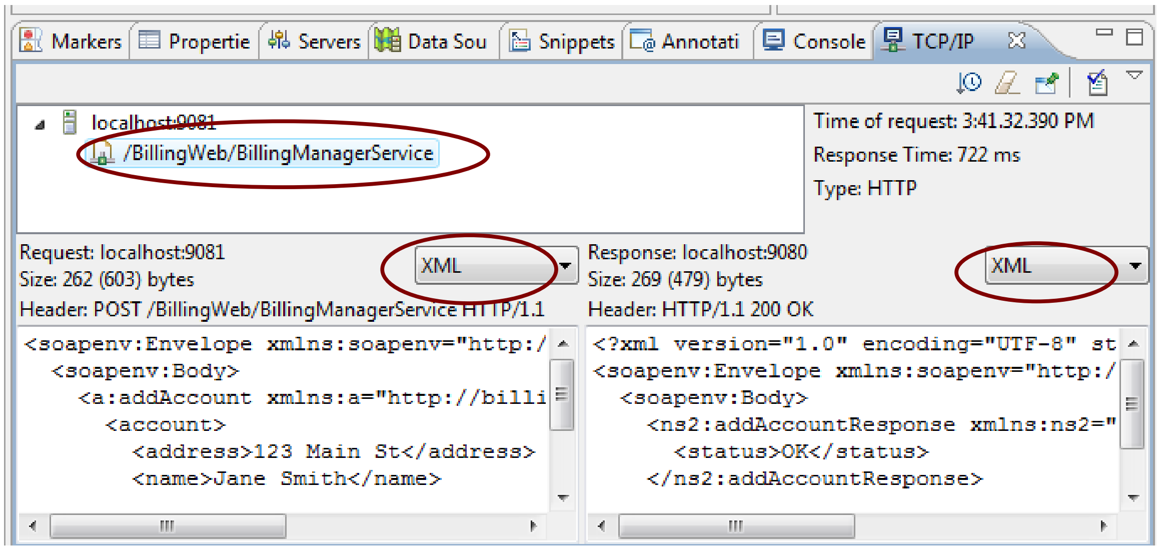
__25. Make sure that the success message is shown.

We will now inspect the SOAP messages.

__26. Select **TCP/IP Monitor** view.

__27. Maximize the **TCP/IP Monitor** view.

__28. You should see a request already captured. Select it as shown below. Also, set the display type to XML.



Time of request: 3:41.32.390 PM
Response Time: 722 ms
Type: HTTP

Request: localhost:9081
Size: 262 (603) bytes
Header: POST /BillingWeb/BillingManagerService HTTP/1.1

XML

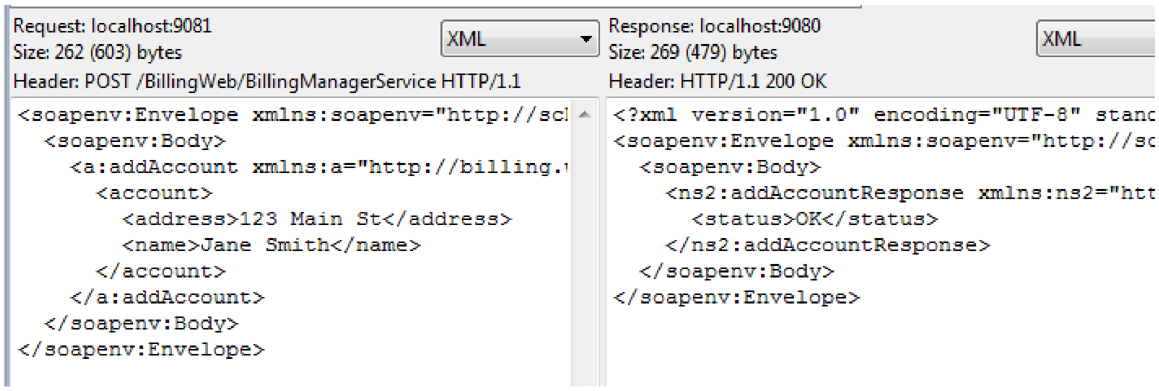
```
<soapenv:Envelope xmlns:soapenv="http:/
  <soapenv:Body>
    <a:addAccount xmlns:a="http://billi
      <account>
        <address>123 Main St</address>
        <name>Jane Smith</name>
```

Response: localhost:9080
Size: 269 (479) bytes
Header: HTTP/1.1 200 OK

XML

```
<?xml version="1.0" encoding="UTF-8" st
<soapenv:Envelope xmlns:soapenv="http:/
  <soapenv:Body>
    <ns2:addAccountResponse xmlns:ns2="
      <status>OK</status>
    </ns2:addAccountResponse>
```

__29. Study the request and response messages. They are in plain text since we have not enabled any message security.



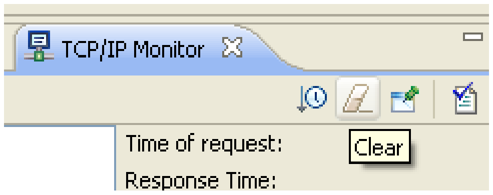
Request: localhost:9081
Size: 262 (603) bytes
Header: POST /BillingWeb/BillingManagerService HTTP/1.1

```
<soapenv:Envelope xmlns:soapenv="http://sc
  <soapenv:Body>
    <a:addAccount xmlns:a="http://billing.
      <account>
        <address>123 Main St</address>
        <name>Jane Smith</name>
      </account>
    </a:addAccount>
  </soapenv:Body>
</soapenv:Envelope>
```

Response: localhost:9080
Size: 269 (479) bytes
Header: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8" stanc
<soapenv:Envelope xmlns:soapenv="http://sc
  <soapenv:Body>
    <ns2:addAccountResponse xmlns:ns2="htt
      <status>OK</status>
    </ns2:addAccountResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

__30. Click the eraser button in the TCP monitor view toolbar to remove all captured messages.



__31. Restore the size of the **TCP/IP Monitor** view.

__32. Close all web browsers.

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 3 - 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. RAD 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 RAD 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 **policySet.xml** referred to. Basically, the run-time will read the **policySet.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.

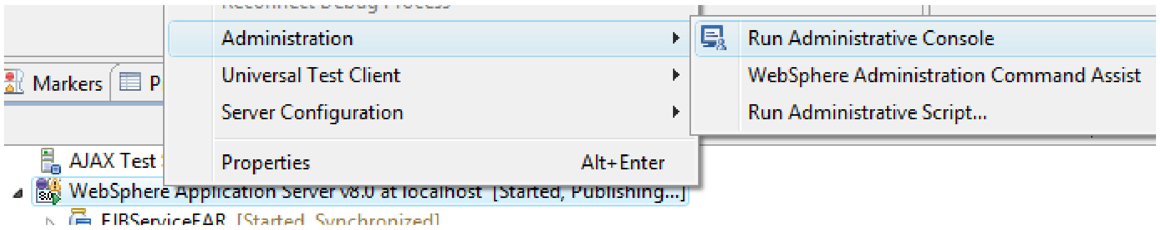
Part 2 - Import the Policy Set

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 RAD, locate the **Servers** view.

__2. Start the server if it is not running.

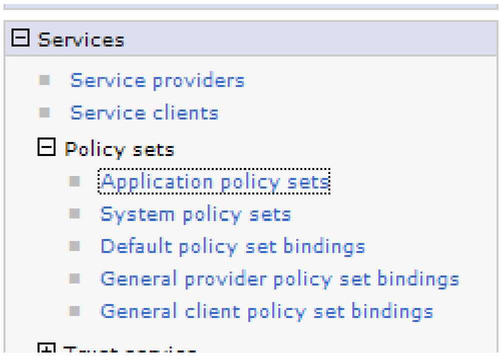
__3. Right click the server in the **Servers** view and select **Administration** → **Run administrative console**.


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]

The admin console will open inside RAD.

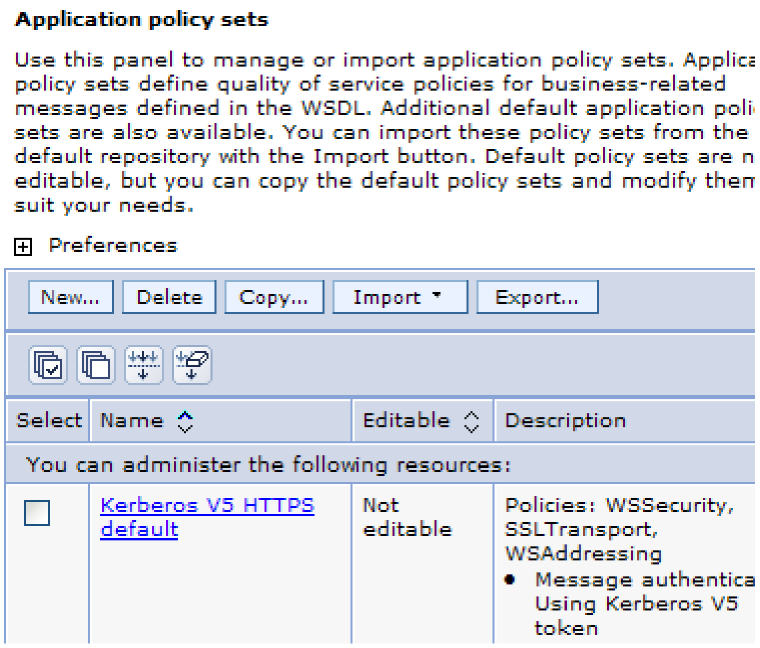
This is the administrative front end to WebSphere that you saw in the previous Lab, and is where a WebSphere administrator would perform the majority of administration tasks.

__4. 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

__5. Click **Application policy sets**.

The right pane will display the *Application policy sets* screen.

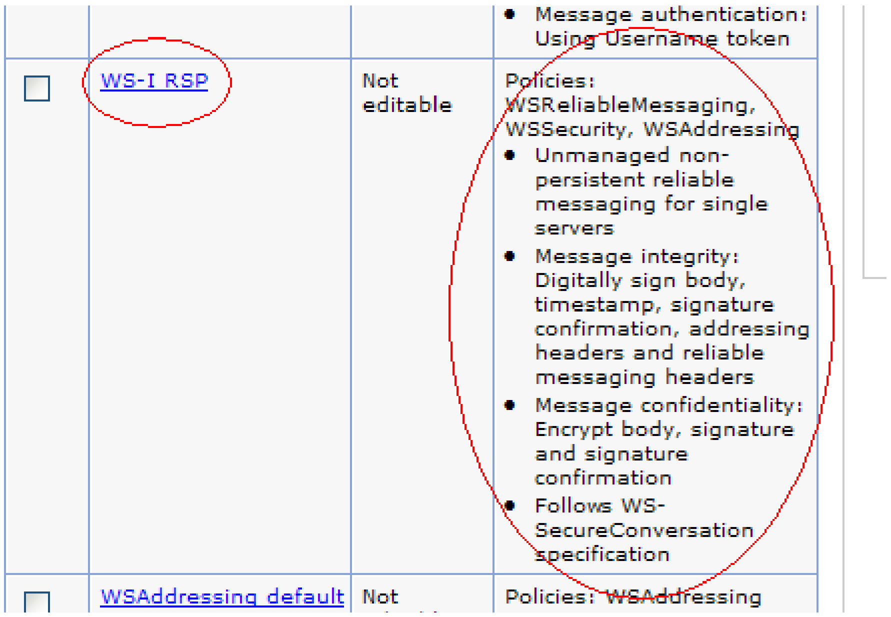

Application policy sets

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.

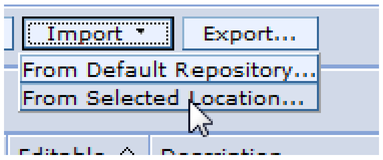
__6. Scroll down the list to examine some of them and what they offer. For example, examine the **WS-I RSP** profile, and look at what it offers.



			• 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.

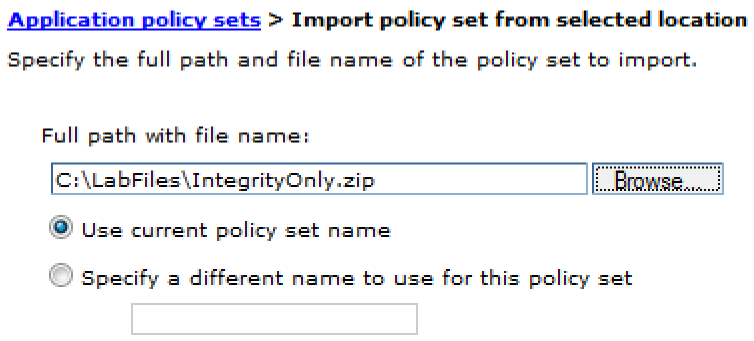
__7. Scroll up. Click the **Import** button, and select **From Selected Location...** from the drop-down.



The screen will change to allow you to specify a location for the policy set.

__8. Click the **Browse...** button.

__9. Select **C:\LabFiles\IntegrityOnly.zip** and click **Open**.



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:

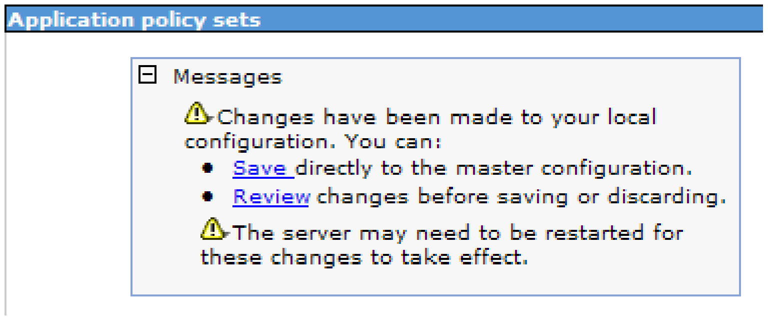
C:\LabFiles\IntegrityOnly.zip Browse...

(●) Use current policy set name

( ) Specify a different name to use for this policy set

__10. Click **OK**.

A *Messages* pane will appear.



Application policy sets

Messages

Changes have been made to your local configuration. You can:

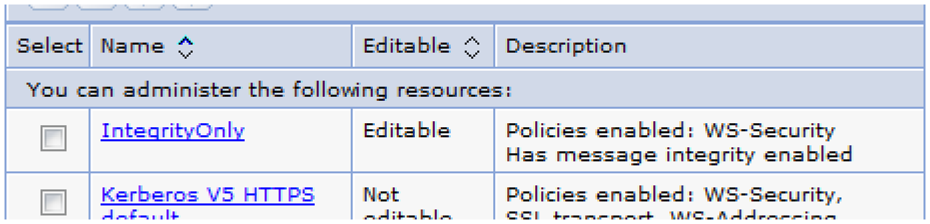
- Save directly to the master configuration.
- Review changes before saving or discarding.

The server may need to be restarted for these changes to take effect.

__11. Click the **Save** link.

The policy set will be imported.

__12. 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.

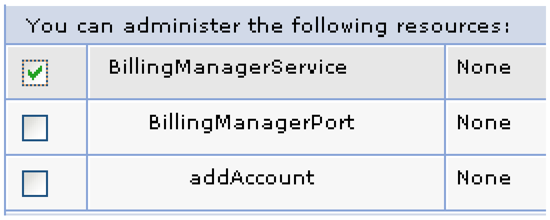
First, we will attach the policy set to the provider.

Note: A policy set can be attached to a provider or consumer using either RAD or WebSphere admin console. 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 admin console.

__1. On the left hand side of admin console, expand **Services** and click **Service providers**.

__2. Click **BillingManagerService**.

__3. Select the checkbox for **BillingManagerService**.

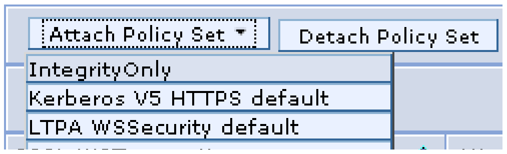


You can administer the following resources:

✓	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.

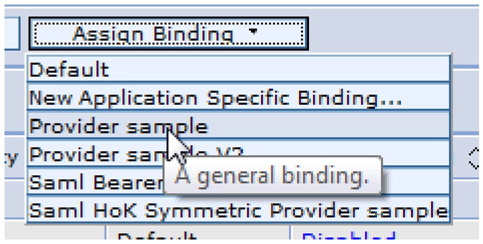
What is a **policy set binding**? A policy simply states what communication behavior should be enabled. For example, integrity and encryption. Various encryption and digest algorithms are also specified in the policy. A policy *does not* specify machine or installation specific details. For example, 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.

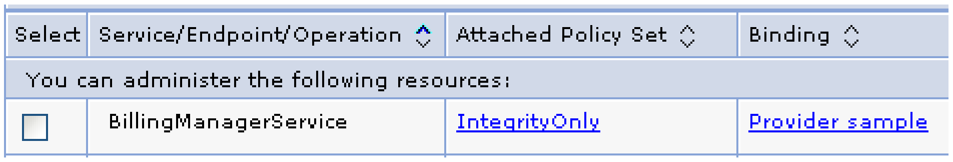
In this and the next few labs, we will use a predefined policy set binding. This binding uses pre-generated keys for the provider and consumer. This is good enough for development. In a production environment, you must generate your own keys. Preferably a key should be signed by a well known certificate 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.

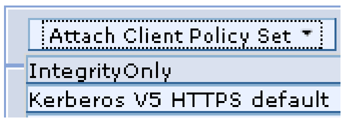
Now, we will configure the consumer.

__9. On the left hand side, click **Service clients** under **Services**.

__10. Click the first **BillingManagerService**.

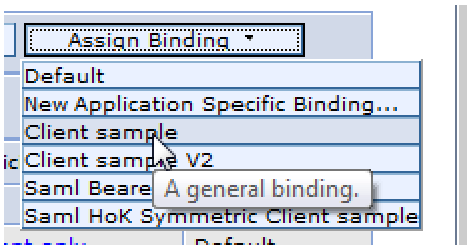
__11. Select the checkbox for **BillingManagerService**.

__12. Click **Attach Client Policy Set > IntegrityOnly**.

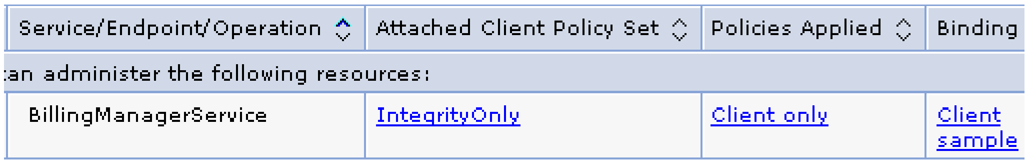


__13. Select the checkbox for **BillingManagerService** again.

__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. Press Control+Alt+R to restart the server or right click the server and select **Restart**.

__21. 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 RAD, 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.

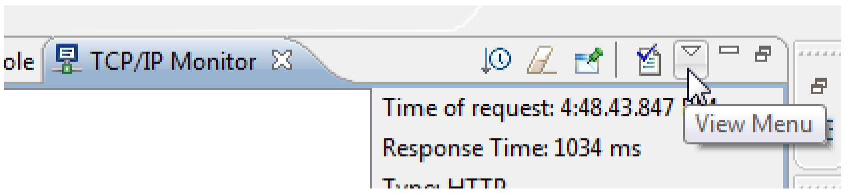
__10. Similarly, locate and observe the SignatureValue, DigestValue and BinarySecurityToken elements in the SOAP response message.

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.

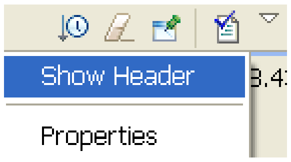
First, we need to view the HTTP header of the existing request. If we do not do that, headers will be omitted from the modified request. Which appears to be a bug in RAD.

__1. In the **TCP/IP Monitor** view, move your mouse over the low-arrow at the top-right of the menu bar, it will display **View Menu**.

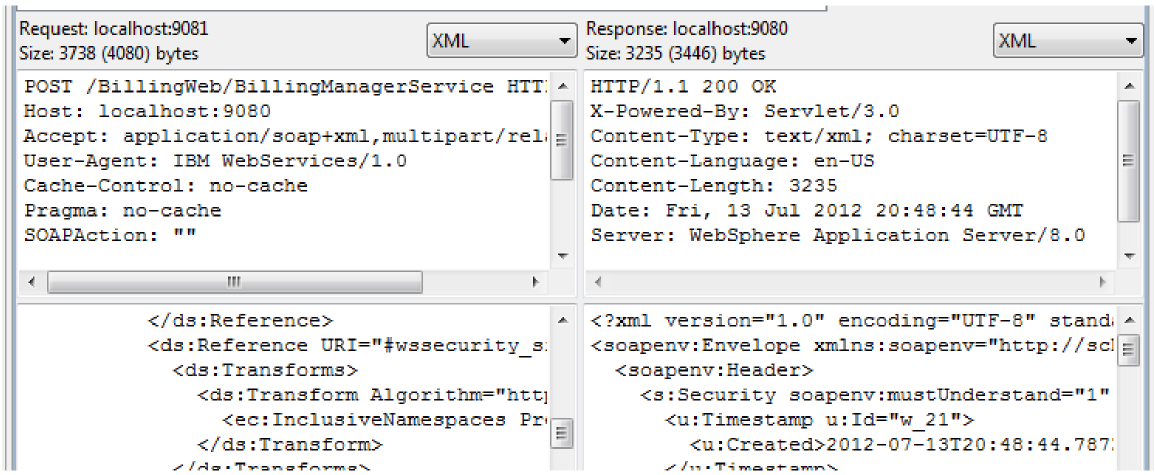


Time of request: 4:48.43.847
Response Time: 1034 ms

__2. Click the **View Menu** button and select **Show Header**.



__3. The **TCP/IP Monitor** view now will show the header for the Request and Response.


Request: localhost:9081
Size: 3738 (4080) bytes

```
POST /BillingWeb/BillingManagerService HTT
Host: localhost:9080
Accept: application/soap+xml,multipart/rel
User-Agent: IBM WebServices/1.0
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
```

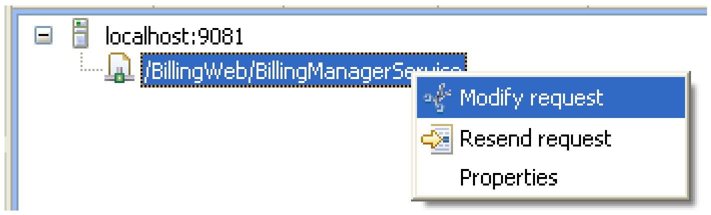
```
          </ds:Reference>
          <ds:Reference URI="#wssecurity_s
            <ds:Transforms>
              <ds:Transform Algorithm="http
                <ec:InclusiveNamespaces Pr
              </ds:Transform>
```

Response: localhost:9080
Size: 3235 (3446) bytes

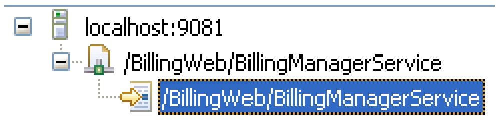
```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0
Content-Type: text/xml; charset=UTF-8
Content-Language: en-US
Content-Length: 3235
Date: Fri, 13 Jul 2012 20:48:44 GMT
Server: WebSphere Application Server/8.0
```

```
<?xml version="1.0" encoding="UTF-8" stand
<soapenv:Envelope xmlns:soapenv="http://scl
  <soapenv:Header>
    <s:Security soapenv:mustUnderstand="1"
      <u:Timestamp u:Id="w_21">
        <u:Created>2012-07-13T20:48:44.787
```

__4. In the top pane of the **TCP/IP Monitor** view, right click the request and select **Modify request**.

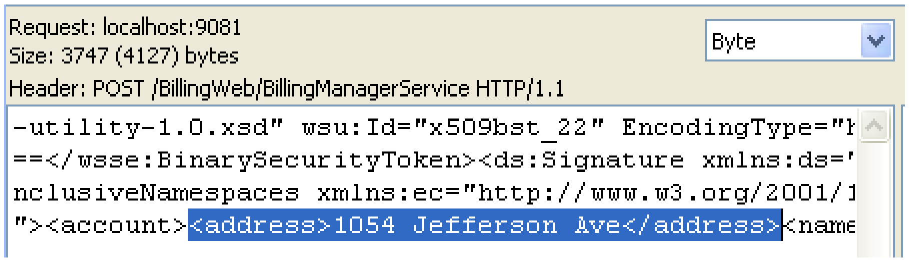


The tool will create a modified request.


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.

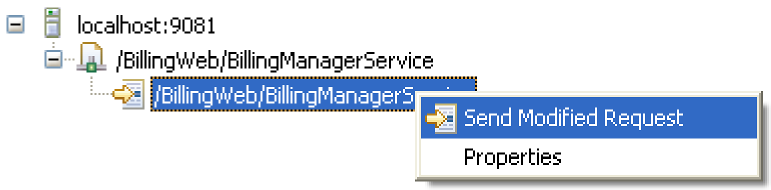
__5. In the **Request** editor, scroll carefully to the right and locate the <**address**> element in the bottom line.


Request: localhost:9081
Size: 3747 (4127) bytes
Header: POST /BillingWeb/BillingManagerService HTTP/1.1
Byte

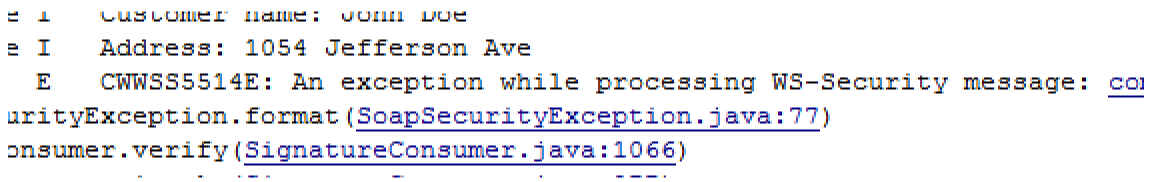
```
-utility-1.0.xsd" wsu:Id="x509bst_22" EncodingType="h
==</wsse:BinarySecurityToken><ds:Signature xmlns:ds='
nclusiveNamespaces xmlns:ec="http://www.w3.org/2001/1
"><account><address>1054 Jefferson Ave</address><name
```

__6. 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.

__7. Right click the modified request and select **Send Modified Request**.


localhost:9081
/BillingWeb/BillingManagerService
/BillingWeb/BillingManagerS
Send Modified Request
Properties

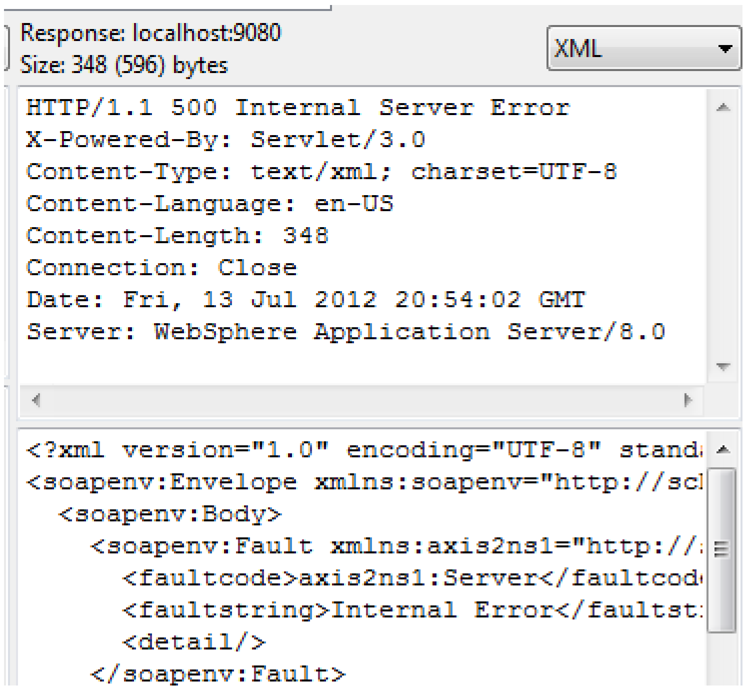
__8. The **Console** view will show an exception from the service.



```
e I   Customer name: John Doe
e I   Address: 1054 Jefferson Ave
  E   CWWSS5514E: An exception while processing WS-Security message: co
urityException.format(SoapSecurityException.java:77)
onsumer.verify(SignatureConsumer.java:1066)
```

__9. Back in the TCP/IP Monitor view, for the **Response** pane, change the display mode to **XML**.

__10. Notice that a fault has been returned by the server.



Response: localhost:9080
Size: 348 (596) bytes
XML

```
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
Connection: Close
Date: Fri, 13 Jul 2012 20:54:02 GMT
Server: WebSphere Application Server/8.0
```

```
<?xml version="1.0" encoding="UTF-8" stand
<soapenv:Envelope xmlns:soapenv="http://sc
  <soapenv:Body>
    <soapenv:Fault xmlns:axis2ns1="http://
      <faultcode>axis2ns1:Server</faultcod
      <faultstring>Internal Error</faultst
      <detail/>
    </soapenv:Fault>
```

__11. Clear all requests in TCP/IP Monitor by clicking the eraser toolbar button.

__12. Hide the Header since you don't need it anymore, remember that you opened the header to modify a request.

__13. 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.