Building REST and SOAP Web Services with Java

Lab Exercises

Eclipse Photon/JBoss EAP 7.1 Version

Version 1.0.0 June 9, 2020



Copyright © Descriptor Systems, 2012
All Rights Reserved.

This document may not be copied or reproduced in any form without prior written consent of Joel Barnum of Descriptor Systems.

All trademarks belong to their respective companies.

You can contact Descriptor Systems at:

http://www.descriptor.com
info@descriptor.com
319-362-3906
P.O. Box 461

Marion, IA 52302

Building REST and SOAP Web Services with Java Labs

Lab 1: Hello, World Web Service

- Part 1: Creating a Web Service
- Part 2: Writing a Standalone Client
- Part 3: Re-deploying

Lab 2: XML Review

- Part 1: XML Namespaces
- Part 2: XML Schema

Lab 3: Examining SOAP Messages

- Part 1: Examining SOAP Messages
- Part 2: Complex Types
- Part 3: Arrays
- Part 4: SOAP Faults
- Part 5: Object Graphs
- Part 6: Monitoring SOAP Messages

Lab 4: WSDL

- Part 1: Reading a WSDL Document
- Part 2: Writing WSDL Top Down
- Optional: Writing a Standalone Client

Lab 5: Writing a JAX-WS Service

- Part 1: Creating the Service
- Part 2: Using a Service Endpoint Interface
- Part 3: Using Annotations
- Part 4: Service Method Parameters
- Part 5: Service Lifecycle and Stateful Services

Lab 6: Writing a JAX-WS Client

Part 1: Creating the Service

Part 2: Using a Proxy from a Standalone Client

Part 3: Writing a Managed Client

Part 4: The TCP/IP Monitor

Part 5: The BindingProvider Interface

Lab 7: Document Style Messaging

Part 1: Writing the Top-Down WSDL

Part 2: Creating the Service

Lab 8: Attachments

Part 1: Writing the Service

Part 2: Writing the Client

Lab 9: Writing Message Handlers

Part 1: Writing a Server-Side Handler

Optional Part 2: Logging More Information

Lab 10: EJBs and Web Services

Part 1: Creating the Service

Part 2: Writing a Web Service Client

Lab 11: Simple RESTful Service

Part 1: Creating the Service

Lab 12: RESTful CRUD Service

Part 1: Creating the Service

Part 2: Implementing the R in CRUD

Part 2: Implementing Update

Part 3: Completing CRUD

Lab 13: Additional JAX-RS Topics

Part 1: Writing a JAX-RS Client

Part 2: Writing a Server-Side Filter

Lab 14: JAX-RS Security

Part 1: Creating the Service

Part 2: Securing the Service with JWT

Part 3: Writing a Security Filter

Case Study Lab

Lab 1: Hello, World Web Service

In this lab, you will write the "Hello, world" of Web services to get you started in Web services development.

Objectives:

- To write a simple Web service
- To become familiar with the Web service development process
- To write a simple standalone Web service client

Part 1: Creating a Web Service

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

Eclipse Installation directory: _____

JBoss Installation directory: _____

If your computer is running Windows 7 or later, the standard directories are:

Setting	Value
Lab Installation directory:	c:\Users\username\websrvclass
Eclipse Installation directory:	c:\Users\username\java\eclipse
JBoss Installation directory:	c:\Users\username\java\jboss-eap-7.1

Please substitute your actual user name for the "username" above.

_2. Use Windows Explorer to navigate to the **{Eclipse Installation Directory}**. Launch Eclipse by double-clicking on the **eclipse.exe** file.

When Eclipse prompts you, enter **{Lab Installation Directory}/workspace** for the *Workspace location*. If your computer has the standard lab setup, the workspace location should be:

```
c:\Users\username\websrvclass\workspace
```

Substitute your actual user name for the username.

_3. Your next job is to install the JBoss Tools package so that Eclipse can control the JBoss server. This will take two steps: first install the so-called "target platform" package, then the tools themselves.

a. From the Eclipse menu, choose *Help - Install New Software* to start the wizard.

b. At the top of the wizard, click the *Add* button, then in the *Add Repository* dialog, click the *Archive* button and browse to **{Lab Installation Directory}/lib**. If your computer has the standard lab setup, the directory is:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
c:\Users\username\websrvclass\lib\jboss
```

In the directory, select *jbosstoolstarget-4.60.0.Final.zip* and press Open followed by Add.

The dialog should indicate that "There are no categorized items" - this is odd, but normal. Press Cancel - the target platform has been installed.

c. From the Eclipse menu, choose *Help - Install New Software* again, then press the Add and Archive buttons as before.

Again, browse to **{Lab Installation Directory}\lib\jboss**, but this time select *jbosstools-4.6.0.Final-updatesite-core.zip*, then press Open followed by Add.

d. Back in the Available Software dialog, expand the entry *Abridged JBoss Tools*, then scroll down to find the entry *JBoss AS, Wildfly & EAP Server Tools* and put a checkmark next to it, then press Next.

e. On the next wizard page, simply press Next.

f. On the next wizard page, accept the license, then press Finish. The installation will take a few minutes. If you are prompted about "unsigned content", press Install Anyway, then wait for the tools to install, then when prompted, restart Eclipse.

_4. Next, create a reference to the JBoss server by following these steps:

a. Find the *Servers* tab at the bottom of the Eclipse window and click it to bring it to the front.

b. Right-click in the empty Servers window and choose *New - Server* to start the wizard.

c. On the first wizard page, expand the *JBoss Community* category, then select *JBoss AS 7.1*, then press Next.

d. On the next wizard page, press Next.

e. On the next wizard page, for the *Home Directory*, press the *Browse* button and navigate to **{JBoss Installation Directory}**. If your computer has the standard lab setup, this directory is:

```
c:\Users\username\java\jboss-eap-7.1
```

Press Finish - the new server should appear in the Servers tab.

_5. Next, configure the JBoss tools so projects deploy to the correct application server directory:

a. In the Servers tab, double click on the JBoss Server entry to open the Properties page.

b. At the bottom of the properties editor, click the *Deployment* tab.

c. Select the choice *Use the JBoss deploy folder*, then put a checkmark next to the *Deploy projects as compressed archives* box.

d. Examine your screen and ensure that it matches Figure 1:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

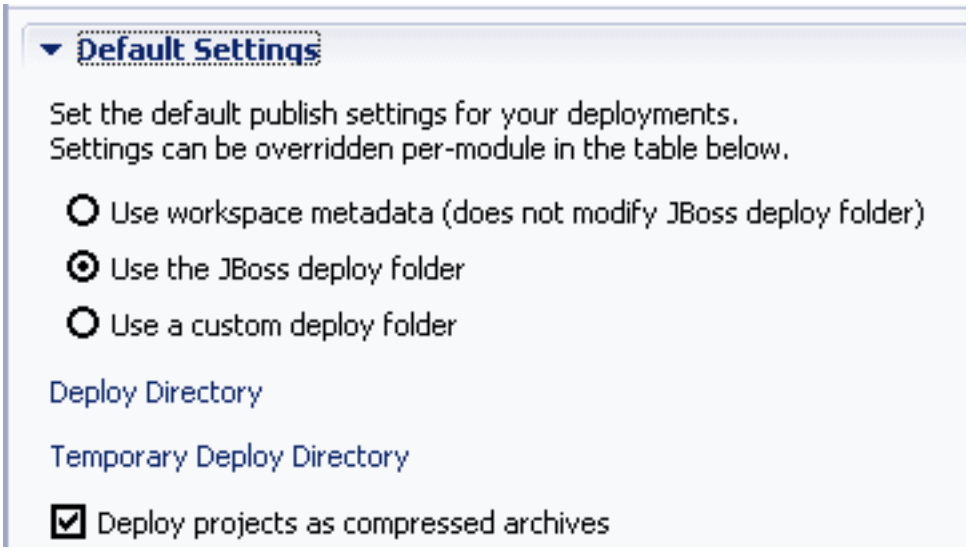


Figure 1: Deployment Settings

Save and close the properties editor.

e. In the Servers view, right-click on the JBoss server and choose *Start* and ensure that the server starts successfully. You'll know when it's started by looking in the Console pane to see the message:

```
JBoss EAP 7.1.0.GA (WildFly Core 3.0.10.Final-redhat-1) started in xxx
```

_6. Your first job is to create a *project* for the lab. Follow these steps:

a. Choose **File - New - Dynamic Web Project** to start the wizard.

b. On the first wizard page, enter **Lab01Web** for the *Name*. For the *EAR Membership*, put a checkmark on *Add project to an EAR*, then CHANGE the suggested name to **Lab01EAR**.

Press Finish.

_7. Examine the *Lab01Web* project in the *Enterprise Explorer* window pane by clicking on the "+" symbol to expand it. Note the following:

- The *Java Resources/src* folder is where you put your Java packages and code.
- The *WebContent* folder is where you put HTML, CSS and so forth.

_8. Eclipse makes it easy to expose Java methods as RPC SOAP Web services. Follow these steps to write a class containing a method that you will configure as an RPC:

a. In the Project Explorer, right-click on the *Lab01Web* project and choose *New - Class*.

For the *package*, enter **service** and for the *Name*, enter **HelloWorld**, then press Finish.

b. Within the new class, write a method that accepts a String and returns a String:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

c. Above the class definition, write an *annotation* to indicate that this class contains one or more Web service methods:

```
@WebService
public class HelloWorld ....
```

d. From the Eclipse menu, choose *Source - Organize Imports* to import the type for the annotation.

e. Press Ctrl+S or choose *File - Save* to save and compile your class. Look in the *Problems* tab at the bottom of the Eclipse window to ensure that you have no compile errors. If you don't see the Problems tab, from the Eclipse menu, choose *Window - Show View - Other* and then start typing **Prob** then select *Problems*.

_9. Now you can deploy your Web service:

a. In the Eclipse *Servers* tab at the bottom of the Eclipse window, right-click on the JBoss server and choose *Start*, then wait for the server to start completely.

b. Right-click on the JBoss server again and choose *Add and Remove* and move *Lab01EAR* to the Configured list.

c. Look in the Eclipse Console - near the end of the messages, you should see a line that looks like:

```
address=http://localhost:8080/Lab01Web/HelloWorld
```

This is the endpoint address of your service. Copy the URL (starting with the "http:" to the clipbard.

_10. Next, test the new Web service. Follow these steps:

a. From the Eclipse menu, choose *Window - Web Browser - Chrome* so that Eclipse uses Chrome as its default browser.

b. From the Eclipse menu, choose *Run - Launch The Web Services Explorer* and wait for it to open in Chrome.

c. The Web Services Explorer (WSE) is a generic Web services client provided by Eclipse that lets you easily test your services. **Note:** We recommend that you double-click on the WSE titlebar tab to maximize the view. You can restore it by double-clicking on the tab again.

d. In the WSE toolbar, in the top right, hover your mouse over the icons to find the icon for *WSDL Page*, then click on it.

Then in the Navigator pane (on the left side), click *WSDL Main*. You should see an "Open WSDL" form into which you can enter a WSDL URL.

e. Paste the service URL from the clipboard into the *WSDL URL* entry field, append the text **?WSDL**, then press Go.

f. In the WSE right-side pane, there's a section labeled *Operations* that contains a clickable list of the methods exposed as Web services -- in this case, only the *hello* method.

Click on the *hello* link, and in the resulting window, click the *Add* link, then enter your name to provide the argument to your Web service method.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Press the *Go* button to invoke the service -- you should see the returned string in the *Status* pane. Does it match your expectations?

g. The WSE also lets you examine the raw SOAP messages that make up an RPC. Double-click on the *Status* pane's titlebar to make it bigger, then press the *Source* link in the pane's upper-right corner to display the SOAP request and response messages.

Don't worry too much about syntax, but see if you can find the parameter string that you entered in the request message and returned string in the response message.

h. You can double-click on the Status pane titlebar again to restore it and press the *Form* link in the upper-right corner to see just the returned string.

You can close the Web Browser for the WSE when you are finished experimenting.

Part 2: Writing a Standalone Client

In this part, you will write a Java program that will act as a client to your Web service.

Steps:

_1. Create a new *Java Project* for the client. Follow these steps:

a. Choose **File - New - Project - Java Project** to start the wizard.

b. Enter **Lab01Client** for the *Project name*, then press Finish.

When Eclipse asks you if you want to change to the Java Perspective, press the No button.

_2. Your next job is to generate the client artifacts so that your client program can invoke the Web service via SOAP. Follow this procedure:

a. Right-click on the *src* folder and choose *New - Other - General - File* and create a new file named **build.xml**.

This will be an *Ant script* that you will run to generate a service from the WSDL.

b. Enter the following into the new build.xml file:

```
<project name="lab01" default="genclient">

  <property name="WSDL"
 value="http://localhost:8080/Lab01Web/HelloWorld?WSDL"/>

    <target name="genclient">
      <exec executable="wsimport">
       <arg value="-keep"/>
       <arg value="${WSDL}"/>
      </exec>
    </target>

</project>
```

c. Right-click on the build.xml file and choose *Run As - Ant Build* - the script should execute with no errors.

d. The *wsimport* command generates the client artifacts into the *service* subdirectory.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

In the Project Explorer, right-click on the *Lab01Client/src* folder and choose *Refresh* -- you should see the generated files in the *service* package.

_3. Next, create a class with a *main* method for the Java client. Follow these steps:

a. Right-click on the *Lab01Client* project and choose **New - Class**.

b. In the wizard, for the *Package*, enter **main** and for the *Name*, enter **Client**. Ensure that *public static void main(String[] args)* checkbox is selected, then press Finish.

_4. Complete the *main* method to invoke the Web service via the generated proxy:

a. Instantiate the proxy factory and retrieve the proxy object itself:

```
HelloWorldService service = new HelloWorldService();
HelloWorld stub = service.getHelloWorldPort();
```

Use *Source - Organize imports* to get rid of compile errors about "XXXX cannot be resolved".

b. After the last line you added in main, add the following code to invoke the Web service and print the results to the console:

```
System.out.println(stub.hello("xxxx"));
```

Substitute your name for the xxxx.

c. Press Ctrl+S to save and compile. Look in the Problems tab at the bottom of the Eclipse window and fix any errors before continuing.

If you don't see the Problems tab, from the Eclipse menu, choose *Window - Show View - Other* then start typing **Prob** into the *type filter text* box, select *Problems* then press Open.

_5. To run your client, in the Project Explorer, right-click on on HelloWorldClient.java and choose *Run As - Java Application*. You should see the returned String from the Web service in Eclipse's Console pane.

Part 3: Re-deploying

In this part, you will learn the procedure to modify and re-deploy a Web service.

Steps:

_1. In the *Lab01Web* project, edit HelloWorld.java and modify the *hello* method's return so it looks like:

```
return "Hello, " + name + ", the time is: " +
   java.time.LocalTime.now();
```

Be sure to save the class before continuing.

_2. Near the bottom of the Eclipse window, find the *Servers* tab and click on it.

Right-click on the JBoss server and choose *Publish*. This ensures that the server is in synch with the changes you made to the Web service.

_3. Run your standalone Java client as a Java program in the *Lab01Client* project -- you should see the updated string that includes the current time in the Eclipse console.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_4. From the Eclipse main menu, choose *File - Close All* to close all of the editor windows.

_5. In the Servers tab, right click on the server and choose *Add and Remove* and remove the EAR project from the Configured list.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 2: XML Review

In this lab, you will review the basic XML concepts required for understanding Web services.

Objectives:

- To understand XML namespaces
- To write an XML Schema for valid XML

Part 1: XML Namespaces

In this part, you will demonstrate your mastery of XML namespaces, which is a required skill for understanding SOAP Web services.

Steps:

_1. The following is a well-formed XML document. Please study it:

```
<env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <env:Body   env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <m:validate    xmlns:m="http://example.com/Lab05">
   <purchase     xmlns:n1="java:creditcard"
     xsi:type="n1:Purchase">
    <amount
       xsi:type="xsd:double">20.2</amount>
    <cardNumber
      xsi:type="xsd:string">sample string</cardNumber>
    <expiryDate
      xsi:type="xsd:dateTime">2005-07-03T16:52:11.218-05:00</expiryDate>
   </purchase>
  </m:validate>
 </env:Body>
</env:Envelope>
```

_2. Answer the following questions:

a. What is the namespace URI of the *Envelope* root element?

_____

b. What is the namespace URI of the *Body* element?

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_____

c. What is the namespace URI of the *validate* element?

_____

d. What is the namespace URI of the *purchase* element?

_____

e. What is the namespace URI of the *encodingStyle* attribute within the *Body* element?

_____

f. What is the namespace URI of the *type* attribute within the *purchase* element?

_____

Part 2: XML Schema

In this part, you will write an XML Schema to validate XML content that represents an exam.

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

Eclipse Installation directory: _____

JBoss Installation directory: _____

If your computer is running Windows Vista, Windows 7 or later, the standard directories are:

Setting	Value
Lab Installation directory:	c:\Users\username\websrvclass
Eclipse Installation directory:	c:\Users\username\java\eclipse
JBoss Installation directory:	c:\Users\username\java\jboss-as-7.1.1.Final

Please substitute your actual user name, e.g. *libtrain1* for the "username" above.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_2. Your first job is to create a *project* for the lab. Follow these steps:

a. Choose **File - New - Project - Java Project** to start the wizard.

b. Enter **Lab02** for the *Project name*, then press Finish. Decline any invitation to switch perspectives.

_3. Next, you will import files. Follow these steps carefully:

a. Start Windows Explorer or My Computer and navigate to **{Lab Installation Directory}/starters/Lab02**.

If your computer has the standard lab setup, this directory is:

```
c:\Users\username\websrvclass\starters\lab02
```

b. In Windows Explorer, select to highlight *exam.xml*, and *exam.xsd*, then choose *Edit - Copy*.

c. In Eclipse, right-click on the *Lab02* project and choose *Paste*.

_4. Open the *exam.xml* that you imported and familiarize yourself with this *instance document*. Note that it contains no namespaces and does not reference a schema.

Note: If necessary, click the *Source* tab at the bottom of the editor to see the raw XML.

_5. Next, edit the starter schema named **exam.xsd** that you imported.

Note: If necessary, click the *Source* tab at the bottom of the editor to see the raw XML.

Note how it defines the root element and required namespace information for a schema.

_6. In *exam.xml*, modify the *test* start tag so it references the schema:

```
<test id="XM30-XML"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="exam.xsd">
```

In the Project Explorer, right-click on the exam.xml file and choose *Validate* -- you will see an error due to the fact that you have not completed the schema. Your job is to complete the schema so that it validates this instance document.

Note: To best see error messages, look in the *Problems* tab at the bottom of the Eclipse window. If you don't see the *Problems* tab, from the Eclipse menu, choose *Window - Show View - Other* then type **Problems** and press Open.

_7. Edit the schema. Start by defining globally scoped *element* definitions for the simple PCDATA elements in the content:

- **description**
- **questionText**
- **choiceText**
- **correct**

For example:

```
<xs:element name="description" type="xs:string"/>
```

As you edit the schema, save it, then go back to the instance document and use the Validate menu.

You should see no errors on the lines that use the definitions you just created.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_8. Next, define the "aggregator" elements as *complex types*, referencing the elements you defined in the last step. Use *minOccurs* and *maxOccurs* as appropriate. Do not worry about attributes yet. Note that you can use the *Outline* pane to help you visualize the source XML's tree structure. (Use Eclipse menu *Window - Show View* to see the Outline pane.)

- **choice**
- **choices**
- **question**
- **testQuestions**
- **test**

For example:

```
<xs:element name="choice">
  <xs:complexType>
    <xs:all>
      <xs:element ref="correct"/>
      <xs:element ref="choiceText"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Note: To specify an unlimited number of child elements, set *maxOccurs* to *unbounded*.

At this point, you should only have errors in *exam.xml* that refer to attributes. If you have any other errors, fix them before continuing.

_9. Now it's time to define the attributes. Define *xs:string* attributes for each of the elements that require them. For example, for the *choice* element:

```
<xs:attribute name="id" type="xs:string"/>
```

_10. Then test your validation as before -- you should have no errors!

_11. From the Eclipse main menu, choose *File - Close All* to close all of the editing windows.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 3: Examining SOAP Messages

In this lab, you will study SOAP messages using a variety of tools.

Objectives:

- To be able to read SOAP messages

Part 1: Examining SOAP Messages

Steps:

_1. Start Eclipse if it's not running.

_2. Your next job is to create a *project* for the lab. Follow these steps:

a. Choose **File - New - Dynamic Web Project** to start the wizard.

b. On the first wizard page, enter **Lab03Web** for the *Name*. In the *EAR membership* section, put a checkmark in the *Add project to an EAR* box, and thhe For the *EAR Project Name*, enter **Lab03EAR**.

Press Finish.

_3. JAX-WS makes it easy to expose Java methods as Web services. Follow these steps to write a class containing a method that you will configure as an service method:

a. In the Project Explorer, right-click on the *Lab03Web* project and choose *New - Class*.

For the *package*, enter **sample** and for the *Name*, enter **SampleService**, then press Finish.

b. Within the new class, write a method that calculates Ohm's Law:

```
public double calcVoltage(double current, double resistance)
{
    return current * resistance;
}
```

c. Above the class definition, write *annotations* that configures the class as containing Web service methods using the *rpc/literal* style and use:

```
@WebService
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
```

d. From the Eclipse menu, choose *Source - Organize Imports* to import the type for the annotations from *javax.jws* packages.

e. Save to compile.

_4. Now you can examine and test your Web service. Follow these steps:

a. In the Servers tab, ensure that the JBoss server is running and start it if it's not.

b. In the Servers tab, right-click on the JBoss server and choose *Add and Remove* and move *Lab03EAR* to the Configured list (you can remove any older EARs), then press Finish.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. Look in the Eclipse Console - near the end of the messages, you should see a line that looks like:

```
address=http://localhost:8080/Lab03Web/SampleService
```

This is the endpoint address of your service. Copy and past the URL (starting with the "http:" to the clipboard.

Note: If you don't see the endpoint address in the console, remove the EAR and let the server republish, then right-click on the server and choose *Clean* to completely republish the project and then look in the Console for the endpoint address.

d. Open a Web browser and paste the endpoint address into the URL bar.

Append the text **?wsdl** to the end of the listening address, then press Enter to display the WSDL for the service.

e. Look in the *binding* section and find the *style* and *use* attributes and write note them here:

__ RPC/encoded

__ RPC/literal

__ Document/encoded

__ Document/Literal

Does that match what you configured in the annotation in the service class?

f. In the browser's URL bar, highlight the WSDL's URL and copy it into the clipboard. It should be something like:

```
http://localhost:8080/lab03Web/SampleServiceService?wsdl
```

_5. Next, test the new Web service. Follow these steps:

a. From the Eclipse menu, choose *Run - Launch The Web Services Explorer* and wait for it to open.

b. The Web Services Explorer (WSE) is a generic Web services client provided by Eclipse that lets you easily test your services. **Note:** We recommend that you double-click on the WSE titlebar tab to maximize the view. You can restore it by double-clicking on the tab again.

c. In the WSE toolbar, in the top right, hover your mouse over the icons to find the icon for *WSDL Page*, then click on it.

Then in the Navigator pane (on the left side), click *WSDL Main*. You should see an "Open WSDL" form into which you can enter a WSDL URL.

Note: If you don't see the form at this point, click the left-arrow icon in the upper toolbar.

d. Paste the WSDL URL from the clipboard into the *WSDL URL* entry field, then press Go.

e. In the WSE right-side pane, there's a section labeled *Operations* that contains a clickable list of the methods exposed as Web services -- in this case, only the *calcVoltage* method.

Click on the *calcVoltage* link, and in the resulting window, enter your values to provide the arguments to your Web service method.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Press the *Go* button to invoke the service -- you should see the returned value in the *Status* pane. Does it match your expectations?

f. The WSE also lets you examine the raw SOAP messages that make up an RPC. Double-click on the *Status* pane's titlebar to make it bigger, then press the *Source* link in the pane's upper-right corner to display the SOAP request and response messages.

The request message should look something like:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns0="http://sample/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns0:calcVoltage>
      <arg0>3</arg0>
      <arg1>4</arg1>
    </ns0:calcVoltage>
  </soapenv:Body>
</soapenv:Envelope>
```

_6. Examine the request message, then answer the following questions.

a. What is the namespace URI of the Envelope element?

`Envelope namespace URI:` _____

b. Which version of SOAP does that namespace URI indicate?

`SOAP version:` _____

c. What is the namespace URI of the Body element?

`Body namespace URI:` _____

Does this URI match the Envelope's namespace URI as is required by the SOAP specification?

d. What is the name of the *payload* element?

`Payload element name:` _____

The *payload* element's name should be **calcVoltage**, which is the name of the service method. This is a requirement of the RPC style.

e. What is the namespace URI of the payload element?

`Payload element namespace URI:` _____

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

This namespace URI identifies the Web service. Look back in the WSDL's *definition* element and find the *targetNamespace* attribute and verify that it matches.

f. Still examining the request message, note the number of child elements of the payload element:

```
Payload element child count:  _____________________________
```

There should be two child elements of the payload element, which is also consistent with the RPC style. In an RPC, each method *parameter* corresponds to a child element in the message.

g. You can double-click on the Status pane titlebar again to restore it and press the *Form* link in the upper-right corner to see just the returned string.

You can close the Web Browser for the WSE when you are finished experimenting.

h. Close any Web browser windows, including the admin page and WSDL pages.

_7. Now let's redeploy the service with the *Document/literal (wrapped)* style:

a. Back in Eclipse, edit SampleService.java.

Modify the SOAPBinding annotation so it looks like:

```
@SOAPBinding(style=Style.DOCUMENT, use=Use.LITERAL,
  parameterStyle=ParameterStyle.WRAPPED)
```

Be sure to save the file.

b. In the Servers tab, right-click on the server and choose *Publish* to ensure that everything's up to date.

_8. In the same fashion as before, open a new Web browser window with the WSDL URL.

Answer the following about the WSDL:

a. Look in the *binding* section and find the *style* and *use* attributes and write note them here:

```
__ RPC/encoded

__ RPC/literal

__ Document/encoded

__ Document/Literal
```

Does that match what you configured in the annotation in the service class?

b. At the top of the WSDL, note the *xsd:schema* element. Looking at the schema, what *elements* are defined?

```
doc/literal (wrapped) elements:  _____________________________________
```

c. What *complexTypes* are defined?

```
doc/literal (wrapped) types:  _____________________________________
```


Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Note how the elements and types define XML that mirror the structure of the *rpc/literal* service you deployed earlier. This is a hallmark of the document/literal wrapped style and use combination.

_9. Test your service in the WSE and verify that the request message matches the schema structure you just examined and are equivalent to the rpc/literal request message you noted in an earlier step.

_10. Close the WSE and any open Web browser windows and tabs.

_11. Now let's redeploy the service with the *Document/literal (bare)* style and use.

a. In SampleService.java, modify the *parameterStyle* to be **ParameterStyle.BARE**.

Save and publish.

b. Open a new Web browser with the WSDL URL.

Note the schema and how the elements are different from the previous examples.

c. Test the service in the WSE and note that the request message doesn't have a method name, so it's not a legal SOAP RPC message. JBoss's SOAP implementation works with it even though it's not legal according to the SOAP specification.

_12. Change the service's style and use back to *rpc/literal*.

Part 2: Complex Types

In this part, you will expose a Java class parameter in a service that will be a complex type in the SOAP message.

Steps:

_1. In the *Lab03Web* project, create a new class that represents an insurance policy named **LifePolicy** in the **insurance** package, then complete the class:

a. Add the following fields:

```
private int policyID;
private String holderName;
private double amount;
private boolean isSmoker;
```

b. Use the Eclipse menu *Source - Generate Getters and Setters* to generate get/set methods for all of the fields.

_2. In SampleService.java, write a new method that accepts a LifePolicy object and returns a "rating":

```
public boolean ratePolicy(LifePolicy lp)
{
    if (lp.getAmount() < 50000 && lp.isSmoker() == false)
        return true;
    else
        return false;
}
```

_3. Save and publish the project.

_4. In the same fashion as the last part of lab, examine the service's WSDL and its imported schema in your Web browser, then answer the following:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. What is the name of the schema complex type that maps the LifePolicy Java class?

```
Complex type for LifePolicy:  _____________________________________
```

b. For the properties in the class, JAX-WS maps the Java type to a corresponding schema type. What are the schema types for the Java types?

```
Java Type            Schema Type
---------            -----------

String               ____________________

double               ____________________

int                  ____________________

boolean              ____________________
```

_5. As in the last part of lab, test the service in the WSE, running the *ratePolicy* method. Note that you may need to do a hard refresh on the browser to see the new operation (for most browsers, you can do a hard refresh by holding the Shift key while clicking the Refresh button).

Examine the request message and ensure you understand how the Java class's properties are mapped to XML in the SOAP message.

_6. Close the WSE and all open browser windows.

Part 3: Arrays

In this part, you will expose a service method that uses arrays.

Steps:

_1. In SampleService.java, write a method that accepts an array of integer and returns an array of LifePolicy objects:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
public LifePolicy[] getPolicies(int[] ids)
{
    LifePolicy[] retVal = new LifePolicy[ids.length];

    int index = 0;
    for(int id : ids)
    {
        LifePolicy lp = new LifePolicy();
        lp.setPolicyID(id);
        lp.setAmount(id * 2000);
        lp.setHolderName("Holder" + id);
        lp.setSmoker(id % 2 == 0);
        retVal[index++] = lp;
    }

    return retVal;
}
```

_2. Save and publish the project.

_3. As before, examine the WSDL in a Web browser and note that it imports two schemas. Examine both schemas and ensure you understand how Java arrays map to a schema complex type with *maxOccurs* set to *unbounded*.

_4. As before, test the *getPolicies* operation in the WSE (you can click the *Add* link to compose the input array of integers).

Examine both the request and response messages and ensure you understand how Java arrays map into XML in the SOAP messages.

_5. Close the WSE and all open browser windows.

Part 4: SOAP Faults

In this part, you will expose a service method that throws an exception and investigate how SOAP Faults work.

Steps:

_1. In the *Lab03Web* project, create a new exception class:

a. Right-click on the *insurance* package and choose *New - Class* to start the wizard.

b. For the *Package*, use *insurance*.

For the *Name*, enter **InvalidPolicyException**

For the *Superclass*, press the *Browse* button and start typing **Excep**, then choose *Exception - java.lang*.

Press Finish.

c. In the new class, write a constructor that accepts the exception's human-readable descriptive string:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
    public InvalidPolicyException(String msg)
    {
        super(msg);
    }
```

_2. Edit SampleService.java and re-work the *ratePolicy* method so it looks like:

```
public boolean ratePolicy(LifePolicy lp)
    throws InvalidPolicyException
{
    if (lp.getAmount() < 0)
        throw new InvalidPolicyException(
                "Amount must not be negative");

    if (lp.getAmount() < 50000 && lp.isSmoker() == false)
        return true;
    else
        return false;
}
```

_3. Save and publish the project.

_4. As before, examine the WSDL and schemas in a Web browser and note that in the *portType*, the *operation* for ratePolicy references a fault message.

_5. As before, test in the WSE, running the *ratePolicy* operation and providing a negative amount. Note that you will need to provide a legal policy ID.

Look in the Status pane at the response message source and verify that it contains a SOAP fault and that the *fault-string* is human-readable string provided by the InvalidPolicyException class.

_6. Close the WSE and all open browser windows.

Part 5: Object Graphs

In this part, you will investigate exposing a service with a method that uses a Java object that references other Java objects.

Steps:

_1. Create a new class in the *insurance* package named **Agent** with the following properties, complete with get/set methods:

```
Type            Name
----            ----

int             agentID
String          name
double          accruedCommissions
```


Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_2. Modify the LifePolicy class so that it has a property named **theAgent** of type *Agent*, complete with get/set methods.

_3. Save and publish.

_4. Test the service in the WSE, running the *ratePolicy* operation. You will need to press the *Add* link so you can enter the Agent information.

Examine the source of the request message and note how theAgent complex type is embedded within the policy.

_5. Close the WSE.

_6. Modify the Agent class so it contains a property that references a list of policies:

a. First, define a field for a collection of LifePolicy objects:

```
private List<LifePolicy> policies =
        new ArrayList<>();
```

b. Generate get/set methods for the new field.

_7. Modify SampleService.java so it has a new method:

```
public LifePolicy getPolicy(int id)
{
    LifePolicy retVal = new LifePolicy();
    retVal.setPolicyID(id);
    retVal.setAmount(20000);
    retVal.setHolderName("Sam Customer");
    retVal.setSmoker(false);

    Agent a = new Agent();
    a.setAccruedCommissions(500000);
    a.setAgentID(33);
    a.setName("Sue Agent");
    a.getPolicies().add(retVal);

    retVal.setTheAgent(a);

    return retVal;
}
```

Note that this method defines a *cyclical* object graph: the Agent references the LifePolicy and vice versa.

_8. Save and publish.

_9. Test the *getPolicy* method in the WSE. It should fail.

Look in the Console at the very end of the output and note that the JAX-WS runtime detected the cyclical object graph and aborted the request.

_10. Close the WSE and all open browser windows.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 6: Monitoring SOAP Messages

In this part, you will use the Eclipse TCP/IP monitor to examine the entire HTTP request for a SOAP message, including the HTTP headers.

Steps:

_1. To configure and start the monitor, follow this procedure:

a. From the Eclipse menu, choose *Window - Preferences*.

In the upper-left corner, in the "type filter text" box, start typing **TCP**, then double-click the entry for *TCP/IP Monitor*.

b. In the TCP/IP monitor pane, click the Add button to configure a monitor.

Then enter or select:

```
Local monitoring port:   8081
Host name:               localhost
Port:                    8080
Type:                    HTTP
```

Press the OK button. This configures the monitor to listen for HTTP requests on port 8081 and forward them to port 8080. Port 8080 is the standard port on which the JBoss Web server listens for requests.

c. Press the Start button to enable the monitor, then OK and close the Preferences.

_2. Test the service in the WSE, sending the request to the monitor, which will then forward the request to service:

a. Start the WSE, specifying the service's WSDL.

b. In the Navigator pane, click the *SampleServiceSoapBinding* link and note that in the *WSDL Binding Details*, there's a section for Endpoints.

c. In the Endpoints section, click the Add link, and then edit the newly added endpoint so it uses this URL:

```
http://localhost:8081/lab03Web/SampleServiceService
```

Note how the 8081 port references the monitor.

Press the Go button to apply the change.

d. In the method list, click the *calcVoltage* method, then choose the new 8081 endpoint from the list.

Provide values for the arguments, then press Go to invoke the service via the monitor.

_3. If all went well, the Web service result should appear in Eclipse's console and the TCPMonitor should've intercepted the request. Look around in the Eclipse window -- you should see a new tab titled *TCP/IP Monitor*. Click on the tab, then double-click its titlebar to maximize it.

In the top pane, expand the *localhost:8081* entry and click on the *lab03Web/SampleServiceService* entry to highlight it. You should see the request and response messages in the lower panes. For each, set the *viewer type* to **XML**.

On the TCP/IP Monitor titlebar's right side, find the button whose icon is a downward-pointing triangle (it's the third from the right edge of the titlebar). Click the icon and choose *Show Header* -- the monitor should open two more panes that show the headers.

Examining the headers, answer the following:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. What is the *Content-Type* of the request?

`Content-Type:` _____

Is this the proper value for the version of SOAP in use?

b. What is the *SOAPAction* header's value:

`SOAPAction:` _____

Does this make sense for the version of SOAP in use?

_4. Examine the rest of the request and response. It should match what you saw in the Web Services Explorer in the last part of lab.

You can close the monitor when you are finished.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 4: WSDL

In this lab, you will exercise the functionality of WSDL.

Objectives:

- To read an existing WSDL file
- To write a WSDL from scratch and generate a starter service

Part 1: Reading a WSDL Document

In this part, you will examine the WSDL for an existing service. The service is the Weather service from the National Oceanic and Atmospheric Administration(NOAA).

Steps:

_1. The homepage for the service is at:

```
http://graphical.weather.gov/xml
```

Open a Web browser and enter the URL and read about the Web service. Note that there's a link to the WSDL for the service. Click on it and download it.

Note: If your computer doesn't have Internet access, we've downloaded the WSDL and placed it in **{Lab Installation Directory}/starters/lab04**.

_2. You can use Notepad to view the WSDL. Open Notepad and then choose *File - Open* and browse to the directory in which the WSDL file resides. At the bottom right of the Notepad file-open dialog, change the file type to *All Files (*.*)*.

_3. Examine the WSDL, then answer the following questions:

a. What is the *target namespace* of the service (make sure you are looking at the *definitions* root element, not a *schema* element)?

_____

b. What are the names of types that this service uses (you only need to list the first three types)?

_____

c. What is the *portType* name?

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_____

d. What is the *style* and *use* of the *NDFDgenByDay* operation? If encoded, what encoding scheme is used? Hint: Look in the *binding* element.

_____

e. What messages comprise the *NDFDgenByDay* operation?

_____

f. Describe the *parts* for the *NDFDgenByDay* input method.

_____

g. As best you can, write the Java interface that corresponds to the *portType* (you only need to write the first two methods).

h. What is the service endpoint address?

_____

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 2: Writing WSDL Top Down

In this part, you will write a WSDL from scratch, generate a starter service, then complete and test the service.

A WSDL file describes a Web service -- commonly, developers first create the Web service and then use their Web services toolkit to generate the WSDL file so that client programmers can access the service. This process is referred to as *bottom up*, since the WSDL is generated after the fact.

However, it is also useful to use a *top down* process where you write the WSDL first, and use it to generate the Web service itself. That process lets architects design the system and hand off the WSDL for implementation by developers. And that's what this part is all about.

The scenario is that you wish to expose an *order status* application as a Web service so that client programs can query the status of an order, given an order number.

Note: The most common cause of problems in this lab is simple typographical errors, so be careful!

Steps:

_1. Start Eclipse if it's not already running.

_2. Create a new Dynamic Web Project named **Lab04Web** associated with *Lab04EAR*.

_3. Right-click on the new project's *WebContent/WEB-INF* folder and create a new folder named **wsdl**.

_4. To get started, right-click on the *wsdl* folder and choose *New - Other - Web Services - WSDL File* to start the wizard.

   a. On the first wizard page, for the *File name*, enter **orderstatus.wsdl**.

      Set the *Target Namespace* to **http://bigcompany.com/orderstatus**, then deselect the *Create WSDL Skeleton* box and press Finish. Eclipse will create the file and open it in the WSDL editor.

   b. At the bottom of the editor window, note the *Source* and *Design* views. Try switching between the views to see the raw XML and a (for now empty) graphical view. In this lab, you will use the Design view to make writing the WSDL from scratch easier.

_5. Start by creating a *port type* for your service. Follow this procedure:

   a. Right-click on the Design view and choose *Add Port Type*. Eclipse creates a symbol and lets you enter its name: type **OrderStatus** and press Enter.

   b. Click the topmost *NewOperation* placeholder text and change it to **getOrderStatus** and press Enter.

   c. At the bottom of the Eclipse window, surface the *Properties* tab (if you don't see it, choose *Window - Show View - Properties* from the Eclipse menu).

   d. In the port type symbol, for the *Input* message, click the the *parameters* cell to select it.

      Then in the Properties tab, change the *Reference Kind* to *Type* and then change the *Type* to *int*.

   e. Repeat the above to change the *Output* message to *Type* with a type value of *string*.

   f. For the *Input* message, click the *parameters* placeholder text to select it, then in the Properties, change its Name to **orderNumber**.

      At this point, your WSDL should look like Figure 1:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

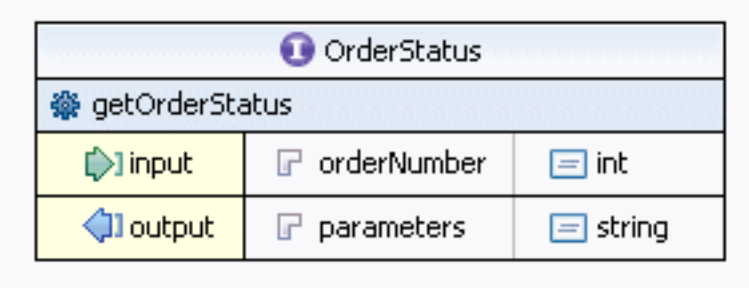


Figure 1: Port Type

g. Examine the WSDL Source at this point and ensure that you understand it.

_6. Next, you can create the binding for your port type. Follow these steps:

a. In Design View, right click on the empty surface of the view and choose *Add Binding* -- Eclipse draws a small symbol that represents the binding.

b. In the Properties, change the *Name* to **OrderStatusBinding**.

Still in the Properties, for the *Port Type*, select *OrderStatus*.

c. In the Properties, click the *Generate Binding Content* button.

For the *Protocol*, select *SOAP*.

For the *SOAP Binding Options*, select *rpc literal*, then press Finish.

d. Use the Source view to examine the newly added WSDL content and ensure that you understand it.

_7. Finally, configure the Service:

a. In the Design View, first left-click anywhere in the empty surface to deselect the binding, then right-click on the white surface of the view and choose *Add Service*.

Enter **OrderStatus** for the Name.

b. Click on the *NewPort* placeholder text to select it, then in the Properties, enter or select:

```
Name:       OrderStatusPort
Binding:    OrderStatusBinding
Address:    http://localhost:8080/Lab04Web/OrderStatus
```

At this point, your WSDL should look something like Figure 2:

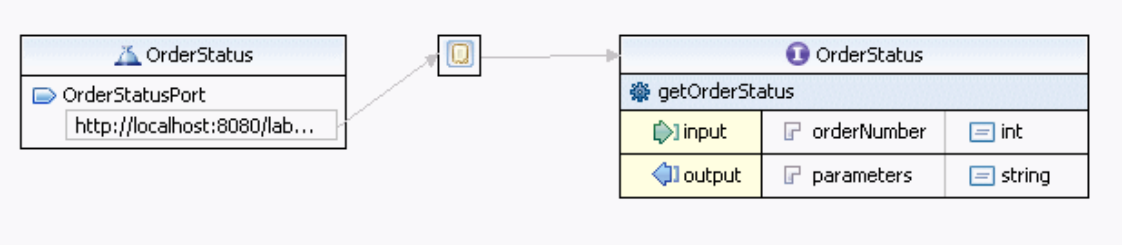


Figure 2: Service

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. Use the Source view to examine the newly added WSDL content and ensure that you understand it.

d. In the Source view, find the *xsd:schema* element at the top of the file, then delete the entire *xsd:schema* element, including the *getOrderStatus* and *getOrderStatusResponse* child elements.

Eclipse create the schema automatically when you created the WSDL, but it's not needed since you changed the messages from "Element" to "Type".

e. Save and close the WSDL.

_8. Now generate a service from the WSDL. Follow these steps:

a. Right-click on the *Java Resources/src* folder and choose *New - Other - General - File* and create a new file named **build.xml**.

This will be an *Ant script* that you will run to generate a service from the WSDL.

b. Enter the following into the new build.xml file:

```
<project name="lab04" default="genservice">

  <property name="WSDL"
 value="../WebContent/WEB-INF/wsdl/orderstatus.wsdl"/>

    <target name="genservice">
      <exec executable="wsimport">
       <arg value="-keep"/>
       <arg value="${WSDL}"/>
      </exec>
    </target>

</project>
```

c. Right-click on the build.xml file and choose *Run As - Ant Build* - the script should execute with no errors.

d. Right-click on the *Lab04Web* project and choose *Refresh* - you should see a new package named *com.bigcompany.orderstatus* that contains two files:

- **OrderStatus.java** - An interface for the Web service you will code in a moment. The *wsimport* task generated this interface from the WSDL.
- **OrderStatus_Service.java** - This is a client-side factory class that you will not use when creating a new service. Please delete this file.

_9. Create the service class that implements the OrderStatus interface:

a. Right-click on the *com.bigcompany.orderstatus* package and choose *New - Class* to start the wizard.

b. For the *Name*, enter **OrderStatusImpl**.

For the *Interfaces*, click the Add button and start typing **Order**, then choose *OrderStatus* and press OK followed by Finish.

_10. Edit OrderStatusImpl.java and complete it:

a. Copying and pasting from OrderStatus.java as necessary, add the following annotations before the class definition:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@WebService(
      endpointInterface="com.bigcompany.orderstatus.OrderStatus"
 , targetNamespace="http://bigcompany.com/orderstatus"
 , serviceName="OrderStatus"
 , portName="OrderStatusPort"
 , wsdlLocation="WEB-INF/wsdl/orderstatus.wsdl")
@SOAPBinding(style = SOAPBinding.Style.RPC)
```

b. Add code to the *getOrderStatus* method so that it returns a String like:

```
The status of order XX is PENDING
```

where XX is the integer passed to the method.

You should have no compile errors at this point.

_11. Now you can examine and test your Web service. Follow these steps:

a. In the Servers tab, ensure that the JBoss server is running and start it if it's not.

b. In the Servers tab, right-click on the JBpss server and choose *Add and Remove* and move *Lab04EAR* to the Configured list (you can remove any older EARs), then press Finish.

c. At the bottom of the Eclipse window, click the Console tab, then look for the Web service's endpoint address **http://localhost:8080/lab04Web/OrderStatus**. Copy this text into the clipboard.

Note: If you don't see the endpoint address in the console, remove the EAR and let the server republish, then right-click on the server and choose *Clean* to completely republish the project and then look in the Console for the endpoint address.

d. Open a Web browser and paste the listening address into the URL bar.

Append the text **?wsdl** to the end of the listening address, then press Enter to display the WSDL for the service.

You should see the WSDL you created.

e. Copy the WSDL URL into the clipboard.

_12. Next, test the new Web service. As you have done in earlier labs, use the Eclipse Web Services Explorer to invoke the service and examine the SOAP messages.

_13. Use Eclipse's *File - Close All* menu to close all open windows.

Optional: Writing a Standalone Client

Steps:

_1. Using the same techniques as in Lab01, create a simple client for the OrderStatus service that invokes the getOrderStatus method in a new Java project named **Lab04Client**.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 5: Writing a JAX-WS Service

In this lab, you will write a JAX-WS service and experiment with several annotations.

Objectives:

- To generate a Web service using JAX-WS in a bottom-up approach.
- To work with exceptions in JAX-WS-based Web services
- To experiment with Web service annotations
- To write a stateful service and client

Part 1: Creating the Service

In this part, you will create a service that validates purchases made by a credit card. Your service will throw an exception if the card number doesn't meet the technical rules for credit-card numbers using the so-called Luhn formula.

Steps:

_1. In the same fashion as in earlier labs, create a Dynamic Web project named **Lab05Web** associated with *Lab05EAR*.

_2. In the new project, create a class named **Purchase** in a new **creditcard** package. You will later write a class with a service method that accepts a Purchase as an argument.

Define the following fields:

```
private String cardNumber;
private Date expiryDate;
private double amount;
```

Use the Eclipse *Source - Generate Getters and Setters* menu to create the get/set methods.

_3. In the *creditcard* package, create a class that subclasses *java.lang.Exception* named **InvalidCardNumberException**:

a. Right-click on the *creditcard* package and choose *New - Class* to start the wizard.

b. For the *Package*, use *creditcard*.

For the *Name*, enter **InvalidCardNumberException**

For the *Superclass*, press the *Browse* button and start typing **Excep**, then choose *Exception - java.lang*.

Press Finish.

c. Define a field that contains the card number:

```
private String cardNumber;
```

Generate get/set methods for the field.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Note that the InvalidCardNumberException class effectively has TWO properties: the cardNumber (which you just implemented) and a *message* property inherited from the Exception superclass.

d. Write a constructor that accepts the card number and assigns a value to the *message* property:

```
public InvalidCardNumberException(String cardNumber)
{
    super("Card number is not legal");
    this.cardNumber = cardNumber;
}
```

_4. In a new **service** package, create a new class named **ValidatePurchase** that will act as the service endpoint implementation class.

Complete the new class as follows:

a. Use annotations to define this class as an RPC/literal Web service.

b. To make this lab easier, we have provided you with the code for a method that implements the Luhn formula algorithm to validate card numbers. Copy and paste the *checkNumber* method from **{Lab Installation Directory}/starters/lab05/starters.txt** into the ValidatePurchase class.

c. Write a method that will act as the RPC service method:

```
public boolean validate (Purchase p)
     throws InvalidCardNumberException
{
    return true;
}
```

Note that the method accepts a Purchase object and throws InvalidCardNumberException using the classes you wrote in an earlier step (you will need to import them).

d. Code the *validate* method:

```
if (p.getCardNumber() == null || p.getExpiryDate() == null)
    return false;

if (p.getAmount() < 0 || p.getAmount() > 100000)
    return false;

if (!checkNumber(p.getCardNumber()))
    throw new InvalidCardNumberException(p.getCardNumber());

return true;
```

_5. Ensure that the JBoss server is running, then add the *Lab05EAR* project to the server (you can remove any older projects).

Look in the Console for the endpoint address. Then copy and paste it into a Web browser, appending **?WSDL** and examine the WSDL.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Note: If you don't see the endpoint address in the console, right-click on the server and choose *Clean* to republish the project and then look in the Console for the endpoint address.

Copy the WSDL URL into the clipboard.

_6. As you have done in earlier labs, use the service's WSDL to open it into the Eclipse Web Services Explorer. Then test the service:

a. Click on the *validate* link, then enter the following into the input form, clicking *Add* as necessary:

```
amount:       33
cardNumber:   49927398716
```

For the expiryDate, click *Add*, the press the Browse link and select today's date.

The card number, 49927398716 follows the rules for valid numbers. Click the *Go* button - the result should be *true* if your service is working properly. (Be sure you don't have any trailing or leading spaces in the card number.)

b. Then repeat the test with 49927398717 - the last digit is different, which makes the number invalid. Your service should throw the InvalidCardNumberException. Please examine the returned SOAP message and ensure that you understand it.

Especially note how the *faultstring* and *detail* elements match the InvalidCardNumberException class you wrote.

c. Repeat the test with a legal card number and an invalid amount to see if your service's logic is correct.

Part 2: Using a Service Endpoint Interface

In this part, you will modify the project so it defines and uses a service endpoint interface (SEI) for the credit-card-validation service.

The advantage of having a separate SEI is that it would allow an architect to first define the service interfaces and then let developers implement them (we are doing it out of order in this lab).

Steps:

_1. In the Project Explorer, right-click on the *ValidatePurchase.java* entry and choose *Refactor - Rename* and rename the endpoint class to **ValidatePurchaseImpl**.

_2. Right-click on the *service* package and create a new interface named **ValidatePurchase**.

_3. Complete the interface:

a. Copy the *validate* method's declaration from the endpoint class to the interface.

b. Copy the *@WebService* and *@SOAPBinding* annotations from the endpoint class to the interface.

Your completed interface should look like:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@WebService
@SOAPBinding(style = Style.RPC, use = Use.LITERAL)
public interface ValidatePurchase
{
 public boolean validate(
   Purchase p) throws InvalidCardNumberException;
}
```

_4. Update the endpoint class to reference the interface:

   a. Modify the @WebService annotation to reference the interface's fully-qualified name.

   b. Delete the @SOAPBinding annotation.

_5. Use the Web Services Explorer to verify that the service works as before. Note that the endpoint address changed as a result of your modifications (the WSDL URL is different, too)!

Part 3: Using Annotations

In this part, you will work with Web services annotations.

Steps:

_1. Open a Web browser and display the service's WSDL, then write down the following:

```
Target namespace:          _____________________________________
Request message name:      _____________________________________
Request message part name: _____________________________________
Response message name:     _____________________________________
Response message part name:_____________________________________
PortType name:             _____________________________________
Operation name:            _____________________________________
SoapAction value:          _____________________________________
Service name:              _____________________________________
Port name:                 _____________________________________
```

_2. Your job is to use annotations on the endpoint class and the endpoint interface to customize the WSDL as follows:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Target namespace:              http://goliath.com/creditcard/validate

Request message name:          verifyCard

Request message part name:     purchaseInfo

Response message name:         verifyCardResponse

Response message part name:    verificationResult

PortType name:                 VerifyCard

Operation name:                verifyCard

SoapAction value:              verifyCard

Service name:                  VerifyCardService

Port name:                     VerifyCardPort
```

_3. Here are some hints:

a. Start by modifying the ValidatePurchase interface, updating the @WebService annotation and writing @WebMethod and @WebParam annotations.

b. Write the same *targetNamespace* element in the @WebService annotation on BOTH the endpoint class and interface.

According to the JavaDoc for @WebService:

```
If the @WebService.targetNamespace annotation is on a service
endpoint interface, the targetNamespace is used for the namespace
for the wsdl:portType (and associated XML elements).

If the @WebService.targetNamespace annotation is on a service
implementation bean that does NOT reference a service endpoint
interface (through the endpointInterface attribute), the
targetNamespace is used for both the wsdl:portType and the
wsdl:service (and associated XML elements).

If the @WebService.targetNamespace annotation is on a service
implementation bean that does reference a service endpoint
interface (through the endpointInterface attribute), the
targetNamespace is used for only the wsdl:service
(and associated XML elements).
```

c. The *portName* and *serviceName* elements in @WebService can ONLY be placed in an endpoint class, not an endpoint interface.

d. Because you changed the service name, the service's endpoint URL will change - look in the Console's server.log for the updated URL. It should be something like:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
http://localhost:8080/lab05Web/VerifyCardService
```

e. Verify that your changes were effective by displaying the WSDL at the new service URL.

Optional Part 4: Service Method Parameters

In this part, you will work with interesting parameters to a service including OUT parameters and dates.

Steps:

_1. In the *service* package, create a new class named **QuadraticFormula**.

Complete the class:

a. Annotate the class so its methods will be exposed as rpc/literal Web services.

b. Your job is to write a method that solves the quadratic equation, which is shown in Figure 1:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 1: Quadratic Equation

Note that to solve this formula, the method will need to accept three IN parameters: *a*, *b* and *c*, and returns two roots (note the plus/minus). The roots will be OUT parameters. The type of all parameters should be *double*.

Write a method named **solveQuadratic** that returns *void* and accepts the five parameters shown above, using *Holders* and annotations as required. The method should return the two roots, implementing the quadratic equation shown in Figure 1.

Be sure to use annotations to name the parameters *a*, *b*, *c*, *root1* and *root2*.

_2. Examine the service's WSDL and ensure you understand how the two OUT parameters are mapped.

_3. Test your service with the Web Services Explorer.

For testing, if you use *a=1*, *b=3*, *c=-4* (note the minus sign), the roots should be 1.0 and -4.0.

Optional Part 5: Service Lifecycle and Stateful Services

In this part, you will write a service that implements a simple shopping cart.

Steps:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_1. Create a new class in a new *shoppingcart* package named **Product** that represents a product in the shopping cart. The Product should have two fields: **productNum** (int) and **price** (double), both exposed with get/set methods.

_2. Create a new class in the *shoppingcart* package named **Item** that represents a line item in the cart. This class should have two fields: **quantity** (int) and **theProduct** (Product), both exposed with get/set methods.

_3. Create a new class in the *service* package named **ShoppingCart** that represents the actual shopping cart.

Complete the *ShoppingCart* class:

a. Use annotations to define this class as an RPC/Literal Web service.

b. Define a field of type *java.util.HashMap* named **products**, initializing the field to reference a new HashMap. The HashMap should be typed to <Integer,Product>: the Integer key will be a product number and the Product will be an object of that type.

c. Use *dependency injection* to initialize a field of type *WebServiceContext* named **wsCtx**.

d. Write a method named **init** annotated so that the JAX-WS runtime calls it after the endpoint object is created.

Then create Product objects and put them into the HashMap as described in the following table:

ProductNum	Price
14	56.78
73	89.22
133	142.92

This HashMap will act as a "database" of available products.

e. Add two empty service methods to the ShoppingCart class:

```
public double addItem(int productNum, int quantity)
{
    return 0;
}

public double getTotal()
{
    return 0;
}
```

You will complete the service methods in the next couple of steps.

_4. Complete the *addItem* method:

a. Create an Item object and initialize it from the input parameters. You can look up the Product object from your HashMap, given the productNum.

b. Retrieve an HTTP session object reference from the WebServiceContext object.

c. Determine if an Items list is already in the session. If not, create one and store its reference in the session:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
ArrayList<Item> items =
    (ArrayList<Item>)session.getAttribute("ItemList");
if (items == null)
{
    items = new ArrayList<Item>();
    session.setAttribute("ItemList", items);
}
```

d. Add the newly created Item object to the ArrayList.

e. Return the subtotal for the newly added Item (i.e. the quantity * the product's price).

_5. Complete the *getTotal* method:

a. As in the *addItem* method, retrieve the ItemList from the HTTP session. If it's not there, simply output a diagnostic to the console indicating that the session is expired and return -1.

b. If the ItemList is in the session, iterate it and return the total amount of the shopping cart.

_6. Annotate the *init*, *addItem* and *getTotal* methods with **@WebMethod**. You should EXCLUDE the *init* method from the list of service methods.

_7. Do a quick test of your service using the Web Service Explorer.

Only test the *getTotal* method -- you should get a return of -1 and see your informative message in the Console. The WSE cannot use sessions, so you cannot test the full functionality of your service.

_8. If you have not already done so, create a new create a new Java Project named **lab05Client**.

_9. As you did in Lab01, generate client artifacts into the client project.

Refresh the client project.

You should see the generated *service* package in the *src* folder.

_10. In the client project, create a new class named **ShoppingClient** in the *main* package with a *main* method. Complete the *main* method:

a. Create an instance of the stub in a similar fashion as in *lab02Client*.

b. Add a couple of items with quantities and product numbers of your choice. Each time you add, print the returned subtotal to the console.

c. Print the shopping cart total to the console.

_11. Run your client as a Java application. Each of the subtotals should be correct, but don't be alarmed if the total is incorrect -- look in the server's console and you should see your message that a session could not be retrieved.

To get sessions to work, the client must indicate that it wishes to use a session by configuring the stub. For example, assuming that you named your stub's variable **stub**:

```
BindingProvider bp = (BindingProvider)stub;
Map<String,Object> ctx = bp.getRequestContext();
ctx.put(BindingProvider.SESSION_MAINTAIN_PROPERTY, true);
```

Add this code AFTER retrieving the stub, but before issuing any of the Web service calls. Then try testing your client again. It should maintain the correct total.

_12. Optional: Use the TCP/IP monitor to examine how the session requires an HTTP cookie.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 6: Writing a JAX-WS Client

In this lab, you will write JAX-WS clients.

Objectives:

- To write a standalone client that uses a proxy.
- To write a managed client that uses dependency injection
- To work with binding-provider properties

Part 1: Creating the Service

In this part, you will import, deploy and start a service for which you later write clients. Since you will be playing the role of a client programmer, we intentionally do not provide the source for the service.

This AgentManager Web service provides methods that lets insurance agents rate policies, process claims, pay bonuses and perform other calculations.

Steps:

_1. First, you will import the service:

   a. From the Eclipse menu, choose *File - Import - Java EE - EAR file*, then press Next.

   b. For the *EAR file*, Browse to **{Lab Installation Directory}/starters/lab06**, then select **lab06Service.ear**.

   c. Press Finish to complete the import.

_2. Deploy the service by right-clicking on the JBoss server and choosing *Add and Remove*, then moving *lab06Service* to the configured list. You can unconfigure any earlier projects.

_3. Ensure that the server is started, then look in the Console's server.log pane - you should see the service endpoint address, which should be something like:

```
http://localhost:8080/lab06ServiceWeb/AgentManager
```

_4. The Web service is a service for use by insurance agents so they can rate policies, pay claims, collect a bonus and perform various other financial calculations.

   Examine the WSDL in a Web browser at:

```
http://localhost:8080/lab06ServiceWeb/AgentManager?wsdl
```

   Looking at the WSDL and the schema, answer the following questions about the service:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Target namespace: _____

Types defined: _____

Operations: _____

PortType name: _____

Style and use: _____

Port name: _____

Service name: _____

Service endpoint: _____

For the *Target namespace*, make sure you used the attribute on the *definitions* element, not from a schema.

_5. Still examining the WSDL, use the following space to write a Java interface that corresponds to the *portType*:

Part 2: Using a Proxy from a Standalone Client

In this part, you will generate client artifacts, then write a client to invoke the service.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Steps:

_1. Your next job is to create a new *project* for the client. Follow these steps:

a. Choose **File - New - Project - Java Project** to start the wizard.

b. Enter **Lab06ProxyClient** for the *Project name*, then press Finish.

Decline any invitation to switch perspectives.

_2. Your next job is to generate the client artifacts so that your client program can invoke the Web service via SOAP.

a. Right-click on the *src* folder and choose *New - Other - General - File* and create a new file named **build.xml**.

This will be an *Ant script* that you will run to generate a service from the WSDL.

b. Enter the following into the new build.xml file:

```
<project name="lab06" default="genclient">

  <property name="WSDL"
value="http://localhost:8080/lab06ServiceWeb/AgentManager?wsdl"/>

    <target name="genclient">
      <exec executable="wsimport">
       <arg value="-keep"/>
       <arg value="${WSDL}"/>
      </exec>
    </target>

</project>
```

c. Right-click on the build.xml file and choose *Run As - Ant Build* - the script should execute with no errors.

d. The *wsimport* command generates the client artifacts into the *service* subdirectory. In the Project Explorer, right-click on the *Lab06ProxyClient/src* folder and choose *Refresh* -- you should see the generated files in the *service* package.

_3. JAX-WS generates client artifacts using the following naming conventions:

a. For the *service* defined in WSDL, there will be a Java class with a matching name that acts as a factory for proxies.

Using the information you read from the WSDL in the last part, find the service class in the *service* package.

b. For each *portType* defined in the WSDL, there will be a Java interface with the same name. Find this interface in the *service* package.

This interface is the client-side equivalent of the Service Endpoint Interface - you will use this as the type of the proxy itself. Please examine it and compare to the Java interface you wrote down in an earlier step.

c. The generated code also includes class definitions of types used by service methods - please examine the Agent, Policy and Claim classes and see if you can make any sense of them.

_4. Right-click on the project and choose *New - Class* to create a new class named **ProxyClient** in the *client* package. The class should have a *main* method.

Using the information you discovered in the last step, and the proxy example in the course notes, write code in *main* to:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

- Instantiate a service object using the zero-argument constructor.
- Use the factory to retrieve a proxy.
- Create a *service.Policy* object and initialize its *age*, *amount*, *holderName* and *smoker* properties to values of your choice (call the *set* methods).
- Call the *ratePolicy* method on the service, displaying its results to the console.

_5. To run, right-click on *ProxyClient*, then choose *Run As - Java Application*. Look in the Console pane -- do you see your client's output?

What are the results? Hint: if you want the *ratePolicy* method to return *true*, you will need to provide Policy properties that would be a good risk to an insurance company!

_6. Modify the client so it also invokes the service's *payClaim* and *payBonus* operations and displays the results to the console.

Part 3: Writing a Managed Client

In this part, you will write a Web-based front end to the AgentManager Web service. The Web application will contain a servlet that uses *dependency injection* to obtain a service reference.

Steps:

_1. Create a new Dynamic Web project named **Lab06ClientWeb** associated with *Lab06ClientEAR*:

a. Choose **File - New - Dynamic Web Project** to start the wizard.

b. On the first wizard page, enter **Lab06ClientWeb** for the *Name*. In the *EAR membership* section, put a checkmark in the *Add project to an EAR* box, and thhe For the *EAR Project Name*, enter **Lab06ClientEAR**.

Press Next.

c. On the next wizard page, press Next.

d. On the final wizard page, put a checkmark next to the *Generate web.xml deployment descriptor* option, then press Finish.

Note: If you neglected to generate the web.xml deployment when creating the project, you can generate it now by right-clicking on the project and choosing *Java EE Tools - Generate Deployment Descriptor Stub*.

_2. In Eclipse, copy and paste the *src/service* package from the Lab06ProxyClient project into the *src* folder in the new project. This gives the new project access to the client artifacts you generated in the last project.

_3. Right-click on the client project's WebContent folder and create a new HTML file named **CalcAnnuity.html**.

Complete CalcAnnuity.html so it looks like:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
<html>
  <head>
    <title>Calculate Annuity</title>
  </head>
  <body>
    <form action="CalcAnnuityServlet"
        method="post">
      <p>
     Payment per period:
     <input type="text" name="payment">
   </p>
   <p>
     Interest rate:
     <input type="text" name="interestRate">
   </p>
   <p>
     Number of periods:
     <input type="text" name="periods">
   </p>
   <p>
     <input type="Submit" value="Submit">
   </p>
    </form>
  </body>
</html>
```

_4. Next, you will write a *servlet* that responds to the form submission. The servlet will act as a client to the Web service you wrote earlier in lab.

Right-click on the *Lab06ClientWeb* project and choose *New - Servlet* to start the wizard.

a. On the first wizard page, for the *Java package*, enter **servlets**.

For the *Class name*, enter **CalcAnnuityServlet**, then press Next.

b. On the next page, note the defaults, then press Next.

c. On the next page, in the *method stubs* section, de-select the *doGet* checkbox, then press Finish.

_5. In the servlet class, define an annotated field so that the Web container can inject a reference to the AgentManager Web service:

```
@WebServiceRef
private AgentManagerService svc;
```

_6. Complete the new servlet's *doPost* method:

a. Initialize the response:

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

b. Retrieve values from the input form and convert from String to appropriate type:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
String sPayment = request.getParameter("payment");
String sInterestRate = request.getParameter("interestRate");
String sPeriods = request.getParameter("periods");

double payment = Double.parseDouble(sPayment);
double interestRate = Double.parseDouble(sInterestRate);
int periods = Integer.parseInt(sPeriods);
```

c. Retrieve an AgentManager SEI reference from the injected service reference:

```
AgentManager proxy = svc.getAgentManagerPort();
```

d. Invoke the service, then send the result to the *out* variable, which represents the response sent back to the browser, using code like:

```
double annuity =
  proxy.calcAnnuity(payment, interestRate, periods);
out.println("Annuity: " + annuity);
```

_7. Test the program by running CalcAnnuity.html on the server.

You can set the operation input parameters to:

```
Payment per period:     100
Interest rate:          0.08
Number of periods:      36
```

With the input values listed above, the result should be: 4053.5557743097606.

Part 4: The TCP/IP Monitor

In this part, you will configure the Eclipse TCP/IP monitor and use the BindingProvider interface to change a proxy's endpoint.

Steps:

_1. If you didn't do so in an earlier lab, you can configure and start the monitor, following this procedure:

a. From the Eclipse menu, choose *Window - Preferences*.

In the upper-left corner, in the "type filter text" box, start typing **TCP**, then double-click the entry for *TCP/IP Monitor*.

b. In the TCP/IP monitor pane, click the Add button to configure a monitor.

Then enter or select:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Local monitoring port:   8081
Host name:               localhost
Port:                    8080
Type:                    HTTP
```

Press the OK button. This configures the monitor to listen for HTTP requests on port 8081 and forward them to port 8080. Port 8080 is the standard port on which the JBoss Web server listens for requests.

c. Press the Start button to enable the monitor, then OK and close the Preferences.

_2. Create a new Java project named **Lab06MonitorClient**.

_3. In the same fashion as you did earlier, generate client artifacts from the AgentManager service or copy and paste the *src/service* package from an earlier project to the new project.

Be sure to refresh the project if you generated the client artifacts using an Ant script.

_4. Create a new class named **MonitorClient** in a new **client** package with a *main* method.

_5. Complete the *main* method:

a. Create a service reference and retrieve a proxy for the AgentManager service in the same fashion as before.

b. Copying and pasting from the last part of lab as necessary, invoke the *calcAnnuity* operation and display the results with System.out.println. You can hard code the input values.

c. Run the client as a Java Application to ensure that it works.

_6. Modify the client to send requests to the monitor, which will capture the request and then forward to the actual service:

a. In *main*, after retrieving the proxy, but before invoking the calcAnnuity operation, configure the proxy's endpoint:

```
BindingProvider bp = (BindingProvider)proxy;
Map<String, Object> rc = bp.getRequestContext();
rc.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
  "http://localhost:8081/lab06ServiceWeb/AgentManager");
```

Note the 8081 port number, which matches the monitor configuration.

b. Run the client as a Java Application.

If all went well, the Web service result should appear in Eclipse's console and the TCPMonitor should've intercepted the request. Look around in the Eclipse window -- you should see a new tab titled *TCP/IP Monitor*. Click on the tab, then double-click its titlebar to maximize it.

c. In the top pane, expand the *localhost:8081* entry and click on the *lab06ServiceWeb/AgentManagerService* entry to highlight it. You should see the request and response messages in the lower panes.

d. You should see two drop-down menus both with *Byte* as the current setting. Change both to *XML* so you can better see the SOAP messages.

Examine the request and response messages and ensure that you understand them.

e. On the TCP/IP Monitor titlebar's right side, find the button whose icon is a downward-pointing triangle (it's the third from the right edge of the titlebar). Click the icon and choose *Show Header* -- the monitor should open two more panes that show the headers.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Examine the request and response headers and ensure that you understand them.

Optional Part 5: The BindingProvider Interface

In this part, you will write a client that uses the BindingProvider interface to supply credentials to a Web service. For purposes of this lab, the service will not actually use the credentials, but this will give you experience working with the BindingProvider interface.

Steps:

_1. Create a new Java project named **Lab06CredentialClient**.

_2. In the same fashion as you did earlier, generate client artifacts from the AgentManager service or copy them from an earlier project.

_3. Create a new class named **CredentialClient** in a new **client** package with a *main* method.

_4. Complete the *main* method (copy and paste from the last part of lab as necessary):

   a. Create a service reference and retrieve a proxy for the AgentManager service in the same fashion as before.

   b. Copying and pasting from the last part of lab as necessary, invoke the *calcAnnuity* operation and display the results with System.out.println. You can hard code the input values.

   c. Run the client as a Java Application to ensure that it works.

_5. Use the *BindingProvider* to set the following request properties:

```
ENDPOINT_ADDRESS_PROPERTY:  The monitor's URL
USERNAME_PROPERTY:          A username such as "Sam"
PASSWORD_PROPERTY:          A password such as "password"
```

_6. Run the client as a Java Application and then look in the monitor to see if you can find the credentials in an HTTP header.

   Note that the credentials are encoded with base-64 encoding, which is not very hard to decode. For all intents and purposes, the credentials are in plain text, which is why HTTP Basic Authentication by itself is not very secure.

   If you have Internet access and are feeling ambitious, search the Web for "base64 decode" and see if you can find an online site that will decode the encrypted credentials.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 7: Document Style Messaging

In this lab, you will create a Web service that accepts an XML document and processes the document using DOM.

The XML document you will pass from client to service is for an insurance policy, and will look like:

```
<m:processPolicy xmlns:m="http://goliath.com/policy>
  <policyID>12</policyID>
  <policy-amount>100000</policy-amount>
  <effective-date>2014-12-12</effective-date>
</m:processPolicy>
```

The business rule that this method will implement is:

```
if (policy amount > 50,000)
  return Rejected
else
  return Accepted
```

The response will contain a simple document that looks like:

```
<m:policyResponse xmlns:m="http://goliath.com/policy>
  <underwrite-result>Accepted</underwrite-result>
</m:policyResponse>
```

Objectives:

- To write a document-style JAX-WS endpoint that uses DOM
- To write a Dispatch client that uses SAAJ

Part 1: Writing the Top-Down WSDL

In this part, you will use the Eclipse WSDL editor to create the WSDL for your document-oriented service.

Steps:

_1. Create a new Dynamic Web project named **Lab07Web** associated with *Lab07EAR*.

_2. Right-click on the new project's *WebContent/WEB-INF* folder and create a new folder named **wsdl**.

_3. To get started, right-click on the *wsdl* folder and choose *New - Other - Web Services - WSDL File* to start the wizard.

   a. On the first wizard page, for the *File name*, enter **policy.wsdl**.

      Set the *Target Namespace* to **http://goliath.com/policy**, then deselect the *Create WSDL Skeleton* box and press Finish. Eclipse will create the file and open it in the WSDL editor.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. At the bottom of the editor window, note the *Source* and *Design* views. Try switching between the views to see the raw XML and a (for now empty) graphical view. In this lab, you will use the Design view to make writing the WSDL from scratch easier.

_4. Start by creating a *port type* for your service. Follow this procedure:

a. Right-click on the Design view and choose *Add Port Type*. Eclipse creates a symbol and lets you enter its name: type **ProcessPolicyPortType** and press Enter.

b. Click the topmost *NewOperation* placeholder text and change it to **processPolicy** and press Enter.

c. Click the topmost *parameters* placeholder text and change it to **policy**.

d. Click the other *parameters* placeholder text and change it to **policyResponse**.

e. To the right of the *input* message, there's a right-facing arrow. Click on it so you edit the *policy* element definition - Eclipse opens another WSDL editor window. If necessary, click the *Design* tab at the bottom to open Design view.

f. In the *(processPolicyType)* symbol, right-click on the *in -string* entry and choose *Delete* to get rid of it.

g. Right-click on the *(processPolicyType)* symbol and choose *Add Element*.

Change the element's name to **policyID** and its type to **int**.

h. Add another element named **policy-amount** of type **long**.

Hint: You will need to Browse for the *long* type.

i. Add another element named **effective-date** of type **date**.

j. Save, then close the *Inline Schema* window.

k. Examine the WSDL Source at this point and ensure that you understand it. Especially ensure that the schema you just wrote matches the input document shown at the beginning of this lab write up.

_5. Repeat the above step to create the **policyResponse** schema. Refer to the the sample response document shown at the beginning of this lab write up. Note that the *underwrite-result* element contains a *string*.

_6. Next, you can create the binding for your port type. Follow these steps:

a. In Design View, right click on the empty surface of the view and choose *Add Binding* -- Eclipse draws a small symbol that represents the binding.

b. In the Properties, change the *Name* to **ProcessPolicyBinding**.

Still in the Properties, for the *Port Type*, select *ProcessPolicyPortType*.

c. In the Properties, click the *Generate Binding Content* button.

For the *Protocol*, select *SOAP*.

For the *SOAP Binding Options*, select *document literal*, then press Finish.

d. Use the Source view to examine the newly added WSDL content and ensure that you understand it.

_7. Finally, configure the Service:

a. In the Design View, first left-click anywhere in the white surface to deselect the binding, then right-click on the empty surface of the view and choose *Add Service*.

Enter **ProcessPolicyService** for the Name.

b. Click on the *NewPort* placeholder text to select it, then in the Properties, enter or select:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Name:       ProcessPolicyPort
Binding:    ProcessPolicyBinding
Address:    http://localhost:8080/Lab07Web/ProcessPolicyService
```

c. Use the Source view to examine the newly added WSDL content and ensure that you understand it.

d. Save and close the WSDL.

Part 2: Creating the Service

In this part, you will write a JAX-WS endpoint that uses the DOM to work with documents.

Steps:

_1. Create a new class named **ProcessPolicy** in the **service** package. The new class does not need a *main* method.

_2. Modify the class definition so the class implements the Provider interface:

```
public class ProcessPolicy implements Provider<Source>
{

}
```

Import javax.xml.transform.Source and javax.xml.ws.Provider.

You will have an error at this point since the class doesn't completely implement the Provider interface. You will fix that in a later step.

_3. Annotate the class so that it's a document-literal endpoint provider:

```
@WebServiceProvider(wsdlLocation = "WEB-INF/wsdl/policy.wsdl",
  serviceName = "ProcessPolicyService",
  targetNamespace = "http://goliath.com/policy",
  portName = "ProcessPolicyPort")
@ServiceMode(Mode.PAYLOAD)
```

_4. Use *Source - Organize Imports* to import *javax.xml.ws.Service.Mode*. You will still have an error at this point.

_5. Use the *Source - Override/Implement Methods* menu so that Eclipse writes an empty *invoke* method. Change the name of the parameter to *invoke* to **request**.

You should have no errors at this point (after importing the correct types).

_6. Your first job in completing the *invoke* method is to use the DOM API to process the input document and retrieve the *policy-amount* value and then perform the business logic described above.

a. Convert the input *Source* parameter to a DOM tree:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
DOMResult dr = new DOMResult();
Transformer trans = TransformerFactory.newInstance()
        .newTransformer();
trans.transform(request, dr);
Document payload = (Document) dr.getNode();
```

b. You can retrieve a NodeList of *policy-amount* elements from the root element using the *getElementsByTagName* method:

```
NodeList nodes = payload.getElementsByTagName("policy-amount");
Element amtElement = (Element)nodes.item(0);
```

c. Once you have retrieved the *policy-amount* element, you can retrieve the text from its child nodes using code like:

```
NodeList list = amtElement.getChildNodes();
StringBuffer sb = new StringBuffer();
for(int i=0; i<list.getLength(); i++)
{
    Node n = list.item(i);
    sb.append(n.getNodeValue());
}

String amtString = sb.toString();
amtString = amtString.trim();
```

After this code, the variable *amtString* should contain the policy amount as a String.

d. Convert the policy-amount String to an *int* using the *Integer.parseInt* method:

```
int amount = Integer.parseInt(amtString);
```

e. Define a String named **retString** that will hold either "Accepted" or "Rejected". Initialize the String based on the policy-amount as described in the business logic above:

```
String retVal = "Accepted";

if (amount > 50000)
    retVal = "Rejected";
```

_7. Next, it's time to use DOM to build the result document as described above:

a. First, create a String that has the response document XML:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
StringBuffer sbXML = new StringBuffer();
sbXML.append("<m:policyResponse ");
sbXML.append(" xmlns:m='http://goliath.com/policy'>\n");
sbXML.append("<underwrite-result>" + retString +
        "</underwrite-result>\n");
sbXML.append("</m:policyResponse>");
String xml = sbXML.toString();
```

b. Next, create an InputSource from the XML string:

```
StringReader sr = new StringReader(xml);
InputSource is = new InputSource(sr);
```

c. Next, create a DOM tree from the InputSource:

```
DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(is);
```

d. Finally, return the DOMSource that represents the DOM tree:

```
return new DOMSource(doc);
```

e. Use the Eclipse *Source - Organize Imports* menu. Your imports should include:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
import java.io.StringReader;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.dom.DOMSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service.Mode;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;
```

f. Use the Eclise *Source - Surround With - Try/Catch Block* to generate exception-handling code.

You should have no errors at this point.

_8. Next, you can deploy and test your service using the Web Services Explorer:

a. Add the *Lab07EAR* project to the server. You can remove any older projects.

b. In the Console, find the service's endpoint address and copy the endpoint address into the clipboard.

c. As in earlier labs, use the Web Services Explorer to load the service's WSDL.

d. Click on the *policy* Operation, then enter or select the following:

```
policy-id:       12
policy-amount:   100000
effective-date:  Browse for today's date
```

Press the Go button. In the WSE Status pane, click the Source link, then find the SOAP response. It should look like:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <m:policyResponse xmlns:m="http://goliath.com/policy">
      <underwrite-result>Rejected</underwrite-result>
    </m:policyResponse>
 </S:Body>
</S:Envelope>
```

e. Try running the service again with a policy-amount of $25,000. The underwrite-result should be "Accepted".

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 8: Attachments

In this lab, you will create a Web service that accepts a PDF file. Since PDFs are not well formed XML, you will pass the PDF content as an attachment outside of the SOAP envelope. You will then write a client program that sends the PDF.

Objectives:

- To work with non-well-formed data in Web services

Part 1: Writing the Service

In this part, you will create a bottom-up Web service that accepts a PDF attachment.

Steps:

_1. Create a new Dynamic Web project named **Lab08Web** associated with *Lab08EAR*.

_2. In the new project, create a class named **ProcessPDF** in a new package named **com.goliath.pdf** package.

_3. Complete the class:

a. Write @WebService and @BindingType annotations on the class so that it uses SOAP 1.1 and MTOM.

The import for SOAPBinding should be:

```
javax.xml.ws.soap.SOAPBinding
```

b. Write a method named **processPDF** that accepts an argument of type String for a filename, and an argument of type *javax.activation.DataHandler* for the PDF. The method should return a String.

c. Annotate the method argument with **@XmlMimeType**:

```
public String processPDF(String fname,
        @XmlMimeType("application/pdf") DataHandler src)
{
}
```

This will cause the generated WSDL to specify "application/pdf" as the *xmime:expectedContentTypes* for the parameter. Since the parameter's type is DataHandler, the schema type will be *xs:base64Binary*. The combination of these inform any client exactly what the service expects.

d. Check the content-type of the request and if it's not PDF, return a message to that effect:

```
String contentType = src.getContentType();
if (!contentType.equals("application/pdf"))
{
    return "Input is not PDF: " + contentType;
}
```


Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

e. Write code to copy the input PDF stream into a file. Your code might look something like (exception handling omitted):

```
File outFile = new File(fname);
System.out.println("Output file: " + outFile.getAbsolutePath());
FileOutputStream fos = new FileOutputStream(outFile);
InputStream is = src.getInputStream();
int abyte = is.read();
while(abyte != -1)
{
    fos.write(abyte);
    abyte = is.read();
}

is.close();
fos.close();
```

Highlight all of the code you just wrote, then from the Eclipse menu, choose *Source - Surround With - Try/Catch Block* to generate exception-handling code.

f. Return the string "Success".

_4. Right-click on the JBoss server and add the project to the server. You can remove any older projects.

_5. In the Console, find the service endpoint URL and copy it to the clipboard, then examine the service's WSDL and schema in a Web browser.

Find the *processPDF* schema complexType and verify that it specifies a type of *base64Binary* and expectedContentTypes of *application/pdf*.

Part 2: Writing the Client

In this part, you will write a client that sends a PDF to the service.

Steps:

_1. Create a new Java project named **Lab08Client**.

_2. As you did in earlier labs, generate client artifacts from the WSDL into the project's *src* folder, then refresh the project in Eclipse.

_3. Use Windows Explorer to find a PDF file on your hard disk (if you can't find one, you can use the *serverStartup.pdf* file in the *lab08* "starters" folder).

Use Windows Explorer to copy the PDF file to the **root** directory of the project (not the *src* folder).

Then, back in Eclipse, refresh the project to see the PDF file.

_4. Create a new class named **SendPDF** in a new package named *client*. The class should have a *main* method.

Complete the *main* method:

a. Create a service object and retrieve a proxy for the service as you have done in earlier labs.

b. Create a File object that represents the PDF:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
File pdfFile = new File("xxxxxx.pdf");
```

Substitute your PDF file's name for the xxxxxx.

c. Create an object that associates a MIME type with a file extension:

```
MimetypesFileTypeMap mftm = new MimetypesFileTypeMap();
mftm.addMimeTypes("application/pdf pdf PDF");
```

d. Create a *java.activation.DataSource* that references the file:

```
FileDataSource ds = new FileDataSource(pdfFile);
ds.setFileTypeMap(mftm);
DataHandler dh = new DataHandler(ds);
```

e. As a debugging aid, output the file's content-type and size:

```
System.out.println("PDF length: " + pdfFile.length());
System.out.println("Content type: " + dh.getContentType());
```

f. Enable MTOM:

```
BindingProvider bp = (BindingProvider)proxy;
SOAPBinding sb = (SOAPBinding)bp.getBinding();

sb.setMTOMEnabled(true);
```

Note: The SOAPBinding type is from the *javax.xml.ws.soap* package.

g. Invoke the service method and display the returned String to the console.

_5. Run the client as a Java application and note the returned String.

Look in the Console to find the full pathname of the PDF "on the server", then use Windows Explorer to verify that the PDF was copied.

Hint: In the Eclipse Console, you can press the double-X icon to close the client's output so you can see the server's console.

_6. As in earlier labs, ensure that the TCP/IP monitor is running, then add code to the client so that it sends requests via the monitor.

Run the client and then look in the monitor and examine the request message. Note how the attachment part follows the Envelope.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 9: Writing Message Handlers

In this lab, you will write a service that uses handlers.

Objectives:

- To write handlers

Part 1: Writing a Server-Side Handler

In this part, you will code and configure a handler that measures the elapsed time of requests to the Web service.

Steps:

_1. As you have done in earlier labs, create a new dynamic Web project named **Lab09Web** associated with *Lab09EAR*.

_2. In the Lab09Web project, create a bottom-up "Hello, world" service. You can copy from the Lab01Web project as necessary.

Deploy (add) lab09EAR to the server and ensure that it's published.

Test the service in the Web Services Explorer to ensure that it works.

_3. To ensure that there's a measurable elapsed time for the handler you will write to measure, edit the service method and add the following (before the *return*):

```
try
{
    TimeUnit.MILLISECONDS.sleep(100);
}
catch(InterruptedException e) {}
```

_4. In a new package named **handlers**, create a class named **LoggingHandler** that implements the *SOAPHandler<SOAPMessageContext>* interface. This class will need the following methods (hint: use the *Source - Override/Implement Methods* menu after you create the class):

- close
- handleFault
- handleMessage
- getHeaders

The handleFault and handleMessage methods should return *true* and getHeaders should return *null*.

Ensure that you have no errors at this point.

Your code should look like this at this point:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
package handlers;

import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class LogginHandler implements SOAPHandler<SOAPMessageContext> {

 @Override
 public void close(MessageContext ctx) {
 }

 @Override
 public boolean handleFault(SOAPMessageContext ctx) {
  return true;
 }

 @Override
 public boolean handleMessage(SOAPMessageContext ctx) {
  return true;
 }

 @Override
 public Set<QName> getHeaders() {
  return null;
 }
}
```

_5. For debugging purposes, put an informative *System.out.println* in each of the overriden methods, for example:

```
System.out.println("In handleFault");
```

_6. Complete the *handleMessage* method:

   a. Rename the arg0 parameter to **ctx** if you haven't already done so.

   b. Write an *if* statement that determines if the message is an inbound (request) or outbound (response) message.

   c. If it's an inbound (request) message, simply use System.out.println to write a message to the console indicating an inbound request was received.

_7. In the same package where the service implementation resides (e.g. *service*), create a file named **handler-chain.xml**. Be sure to use that exact name and ensure that it's in the package where the service implementation class resides.

_8. Complete the XML file so it looks like:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<javaee:handler-chains
    xmlns:javaee=
"http://oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/web-app_3_0.xsd"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <javaee:handler-chain>
  <javaee:handler>
   <javaee:handler-class>
     handlers.LoggingHandler
   </javaee:handler-class>
  </javaee:handler>
 </javaee:handler-chain>
</javaee:handler-chains>
```

Note: You can ignore any warnings in the XML regarding "No grammar constraints".

_9. In your service implementation class, add the following annotation above the class definition:

```
@HandlerChain(file = "handler-chain.xml")
```

_10. Test with the Web Services Explorer. You should see your handler's message in the Eclipse console.

_11. Now update the handler so it measures the service's performance. Modify the *handleMessage* method:

a. If the message is a request message, note the current time and store it in the message context:

```
long startTime = System.currentTimeMillis();
ctx.put("startTime", startTime);
```

b. If the message is a response message, note the new current time, retrieve the start time from the context, and calculate and display the elapsed time:

```
long endTime = System.currentTimeMillis();
long startTime = (Long)ctx.get("startTime");
System.out.println("Elapsed time: " + (endTime - startTime) +
  " milliseconds");
```

c. Test in the WSE - you should see the elapsed time in the Eclipse console.

Optional Part 2: Logging More Information

In this part, you will use the SAAJ API in your request handler to log the name passed to your "hello world" service.

Steps:

_1. Using the SAAJ chapter in the book and the SAAJ JavaDocs, modify your request handler so that it prints the request message's name argument value to the console.

Hint: When you iterate elements using SAAJ, you need to filter out spurious Text nodes and find SOAPElement nodes. You can use code like:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
SOAPElement payload = null;
Iterator iter = body.getChildElements();
while (iter.hasNext())
{
    org.w3c.dom.Node n = (org.w3c.dom.Node)iter.next();
    if (n instanceof SOAPElement)
    {
        payload = (SOAPElement)n;
    }
}
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 10: EJBs and Web Services

In this lab, you will expose an EJB as an interoperable Web service. You will then write a simple client.

Objectives:

- To work with EJB-based Web Services

Part 1: Creating the Service

Steps:

_1. First, you need to create an EJB project for your EJB. Follow these steps:

   a. Choose **File - New - EJB Project** to start the wizard.

   b. On the first wizard page, for the *Project name*, enter **Lab10EJB**.

      At the bottom of the wizard page, put a checkmark next to the *Add project to an EAR* box, then for the *EAR Project Name*, enter **Lab10EAR**.

      Press Next.

   c. On the next wizard page, note the defaults, then press Next.

   d. On the next wizard page, uncheck the box labeled *Create an EJB Client JAR Module...*, then press Finish.

      You should see the new EJB and EAR projects in the Enterprise Explorer view.

_2. In the Project Explorer view, expand the *lab10EJB/Deployment Descriptor: lab10EJB* entry -- you should see an empty folder for *Session Beans*.

_3. Right-click on the *Session Beans* folder and choose **New - Session Bean (EJB 3.x)**. Complete the wizard:

   a. On the first wizard page, for the *Java package*, enter **ship**.

      For the *Class name*, enter **ShippingBean**.

      Uncheck the *No-interface View* box, then put a checkmark next to the *Remote* box, then, for the remote interface name, change the name to **ship.ShippingRemote** and press Next.

   b. On the next wizard page, uncheck the boxes for *Inherited abstract methods* and *Constructors from superclass*, then press Finish.

   c. Eclipse creates the business interface and the implementation class and opens the class into the Java editor.

   d. Look in the Project Explorer and note that you can see the source files under the *ejbModule* folder in the *ship* package.

_4. Double-click *ShippingRemote.java* to open it into the Java editor. Note how the wizard inserted the @Remote annotation to configure this as a remote interface.

   In the interface, define the *calcDomestic* method that returns a hypothetical shipping-charge calculation:

```
double calcDomestic(String zip);
```

   Press Ctrl+S to save and compile. You should have no errors in the interface, but you will have errors in the implementation class.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_5. Open ShippingBean.java (the class). Note how the wizard inserted the @Stateless annotation to configure the EJB as a stateless session bean. In the class, write a simple implementation method that returns a hard-coded shipping charge:

```
public double calcDomestic(String zip)
{
    return 3.55;
}
```

Press Ctrl+S to save and compile.

_6. Complete the *ship/ShippingBean.java* implementation and expose it as a Web service:

a. Add the **@WebService** annotation to the class.

_7. Add the EAR to the server and look at the console and note the Web service's endpoint address.

_8. As you have done in earlier labs, test the service using the Web Services Explorer.

Part 2: Writing a Web Service Client

In this part, you will write a simple client for your shipping-calculation Web service.

Steps:

_1. As you have done in earlier labs, create a standalone Java client for the "shipping" Web service.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 11: Simple RESTful Service

In this lab, you will create a simple RESTful service using JAX-RS.

Objectives:

- To demonstrate RESTfulness.

Part 1: Creating the Service

Steps:

_1. Create a new Dynamic Web project named **Lab11Web** associated with *Lab11EAR*.

_2. Create a new class named **HelloREST** in a new *services* package.

_3. Complete the class so it can act as a source of RESTful content:

a. Write an annotation on the class that assigns a RESTful URL:

```
@Path("/hellorest")
```

b. Write a simple method that returns content:

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

c. Annotate the method so that it responds to an HTTP *GET* and produces a text HTML result:

```
@GET
@Produces("text/html")
```

Import the javax.ws.rs.Produces type.

d. Annotate the *name* parameter so it's accessible to the service as an HTTP GET parameter:

```
@QueryParam("name")
```

e. Import types from the *javax.ws.rs* package.

_4. Create another class in the *services* package named **HelloApplication** that extends *javax.ws.rs.core.Application*:

a. Right-click on the *services* package and choose *New - Class* to start the wizard.

b. On the first wizard page, for the *Name*, enter **HelloApplication**.

Then for the *Superclass*, press the Browse button and start typing **Appli**, then choose *Application - javax.ws.core*.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Press Finish.

_5. Complete the class:

a. Annotate the class to provide a prefix for the RESTful URL:

```
@ApplicationPath("/rest/")
```

b. Use the *Source - Override/Implement Methods* menu to override the *getClasses* method.

You can delete the generated call to the superclass.

c. Add code to the method so that it creates a Set containing all of your RESTful sources (only one in this case):

```
public Set<Class<?>> getClasses()
{
    Set<Class<?>> s = new HashSet<Class<?>>();
    s.add(HelloRest.class);
    return s;
}
```

_6. Deploy the application by adding the EAR to the JBoss server.

_7. Test the RESTful service by opening a Web browser and entering this URL:

```
http://localhost:8080/Lab11Web/rest/hellorest?name=xxxx
```

Substitute your name for the xxxx. You should see a simple Web page saying "Hello" to your name.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 12: RESTful CRUD Service

In this lab, you will create a RESTful application that supports the CRUD (Create, Retrieve, Update and Delete) operations using JAX-RS.

Objectives:

- To write a RESTful CRUD application.

Part 1: Creating the Service

Steps:

_1. Create a new Dynamic Web project named **Lab12Web** associated with *Lab12EAR*. On the last page of the wizard, be sure to put a checkmark in the box to create the Web deployment descriptor.

_2. First, create a Customer class that your service will retrieve and store:

   a. In a new **com.goliath.domain** package, create a class named **Customer**.

   b. Decorate the class definition with the **XmlRootElement** annotation and import javax.xml.bind.annotation.XmlRootElement.

   c. Add the following fields:

```
private int id;
private String name;
private String address;
private String city;
private String state;
private String zip;
```

   d. Generate get/set methods for the fields.

   e. Use *Source - Generate hashCode() and equals()* to generate those methods. Deselect all of the fields in the wizard except for the *id* field.

   f. Use *Source - Generate toString()* to generate that method that displays all of the fields.

_3. Next, let's write a servlet context listener that will create an in-memory "database" of Customer objects and store them at application scope.

   Note: This technique of using an in-memory list and using the ServletContext (application scope) is OK for a lab, but should NOT be used in a production app since it's not thread safe. We are doing it here for simplicity, so we don't need to install, configure and code to an actual database.

   a. Right-click on the project and create a new class in the *com.goliath.listeners* package named **MyContextListener** that implements *ServletContextListener* from the *javax.servlet* package.

   b. Annotate the class with **WebListener** to register the class as a context listener.

   c. In the *contextInitialized* method, write the following to create a Customer list and store it at application scope:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
List<Customer> customers = new ArrayList<Customer>();

Customer c = new Customer();
c.setId(12);
c.setAddress("123 Elm");
c.setCity("Madison");
c.setName("Bill Smith");
c.setState("WI");
c.setZip("53220");
customers.add(c);

c = new Customer();
c.setId(44);
c.setAddress("456 Oak");
c.setCity("Marion");
c.setName("Sue Jones");
c.setState("IA");
c.setZip("52302");
customers.add(c);

arg0.getServletContext().setAttribute("customers", customers);
```

Part 2: Implementing the R in CRUD

In this part, you will implement retrieving Customer as XML, given a customer ID.

Steps:

_1. First, create the Java class that will act as the service's endpoint:

a. In the **com.goliath.service** package, create a class named **CustomerService**.

b. Annotate the class to assign a URI to the service:

```
@Path("/customer")
```

Import the *javax.ws.rs.Path* type.

c. Define a field and annotate it so your service can access the ServletContext object that represents application scope:

```
@Context
ServletContext servletContext;
```

Import the *javax.servlet.ServletContext* and *javax.ws.rs.core.Context* types.

d. Start writing a method that will provide the "R" in CRUD, i.e. it will retrieve a customer in JSON form, given a customer ID:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/{id}")
public Response find(@PathParam("id") int id)
{
    Response ret = null;

    return ret;
}
```

Import *javax.ws.rs.core.Response* and *javax.ws.rs.Produces*.

Note how we annotate this method: @GET and @Produces configure the HTTP verb and content type, @Path configures a path parameter and @PathParam maps the "{id}" URI template to the parameter on the method.

Note that we use the "{id}" syntax to specify a placeholder in the URI for the customer's ID. In other words, URIs from the client will look like:

```
http://localhost:8080/Lab12Web/rest/customer/xxxxx
```

Where 'xxxxx' is the customer ID. The "/customer" portion of the URL is specified on the class itself.

e. In the method, after the *String ret = null;* line but before the *return*, retrieve a reference to the customer list that your listener stored at application scope:

```
List<Customer> customers = (List<Customer>) servletContext
        .getAttribute("customers");
```

f. Then iterate the list, looking for a customer with the input ID:

```
Customer theCust = null;
for(Customer c : customers)
{
    if (c.getId() == id)
    {
        theCust = c;
        break;
    }
}
```

g. Next, write an *if-else* statement that checks if a there was a customer with the specified ID:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
if (theCust != null)
{

}
else
{

}
```

h. If theCust isn't null, use JAXB automatic object mapping to create an XML response from the customer object:

```
ret = Response.ok(theCust).build();
```

i. In the *else* block (for theCust == null), send back HTTP status code "404 Not Found":

```
ret = Response.status(
        Response.Status.NOT_FOUND).build();
```

_2. Next, in the *com.goliath.service* package, create a class named **CustomerApplication** that extends *Application* from the *javax.ws.rs.core* package.

Complete the class:

a. Annotate the class to provide a prefix for the RESTful URL:

```
@ApplicationPath("/rest/")
```

b. Use the *Source - Override/Implement Methods* menu to override the *getClasses* method.

You can delete the generated call to the superclass.

c. Add code to the method so that it creates a Set containing all of your RESTful sources (only one in this case):

```
Set<Class<?>> s = new HashSet<Class<?>>();
s.add(CustomerService.class);
return s;
```

_3. Deploy the application by adding it to the JBoss server. You can remove any previous projects.

_4. Test the RESTful service by opening a Web browser and entering this URL:

```
http://localhost:8080/Lab12Web/rest/customer/44
```

You should see a simple Web page showing the JSON for customer #44.

_5. Modify the URL so that the customer ID is 566 - you should get a 404 Not Found or an empty page.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_6. Next, instead of just displaying the raw JSON, let's write a JavaScript client that will make an AJAX request to get the Customer info and then display it in a simple web page. To make the JavaScript simpler, you will use the *axios* JavaScript library to make the AJAX calls. Follow these steps:

a. Right-click on the project's *WebContent* folder and create an HTML file named **GetCustomer.html**.

b. Modify the page title to **Customer Info**, then immediately following the close tag for title, reference the *axios* JavaScript AJAX library:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

c. Immediately below the line you just wrote, write another *script* element in which you use axios to make an AJAX request for the Customer info. When the request returns, this script will use the HTML DOM to update a page element so the user can see it (you'll create the page element shortly):

```
<script>
function getCustomer() {
    axios.get('rest/customer/44')
    .then(function (response) {
     // handle success
        document.getElementById("customerInfo").innerHTML =
            JSON.stringify(response.data);
    })
    .catch(function (error) {
     // handle error
     document.getElementById("customerInfo").innerHTML =
           error.message;
    })
}
</script>
```

d. In the *body* element, define a button and attach a JavaScript event handler that calls the *getCustomer* function you wrote in the last step:

```
<button onclick="getCustomer()">Get Customer</button>
```

e. Immediately following the button, define an HTML *div* that will hold the customer info. Note how we use an HTML *id* attribute so the getCustomer() function you wrote can find the *div* and update its contents when the AJAX response comes back:

```
<div id="customerInfo">No data yet</div>
```

_7. To test, right-click on the GetCustomer.html file and choose *Run As - Run on Server*, then select the JBoss server and press Finish.

After the page opens in the browser, click the button to see the customer info.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

If you're feeling ambitious, modify the page so that it displays each part of the Customer info (id, name and so forth) in a separate page element (*div* or *span*).

Part 2: Implementing Update

In this part, you will implement the Update verb in a CRUD RESTful service.

Steps:

_1. Start by editing CustomerService.java and add a new method that responds to HTTP PUT:

a. Write the stub of a method that accepts a Customer parameter and returns a JAX-RS Response:

```
public Response update(Customer c)
{
   ResponseBuilder rb = null;

   return rb;
}
```

Note how the method accepts a Customer Java object - clients will send a JSON object in the request body and JAX-RS will marshal it into a Customer.

b. Annotate the method so it:

- Responds to HTTP PUT
- Consumes JSON content

Note that since we didn't write @Path, the method will respond to a PUT to the URL *http://localhost:8080/Lab12Web/customer*. Examine the @Path annotation on the class and ensure that you understand how this works.

c. Retrieve the List of Customers from the ServletContext in the same fashion as you did in the find() method in the last part of lab.

d. Search the list looking for a Customer with the ID that matches the Customer passed as a parameter to this method. If found, set the found Customer's properties to match the Customer properties passed as a parameter to the method. Then set the response to 204 No Content:

```
rb = Response.noContent();
```

If there's no matching Customer in the list, set the response to 404 Not Found:

```
rb = Response.status(Status.NOT_FOUND);
```

Your completed update() method should look something like:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
public Response update(Customer c)
{
    System.out.println("In update: " + c);
    ResponseBuilder rb = null;

    if (c != null)
    {
        Customer found = findCustomer(c.getId());

        if (found != null) {
            found.setAddress(c.getAddress());
            found.setCity(c.getCity());
            found.setName(c.getName());
            found.setState(c.getState());
            found.setZip(c.getZip());
            rb = Response.noContent();
        }
        else
        {
            rb = Response.status(Status.NOT_FOUND);
        }
    }
    else
    {
        rb = Response.status(Status.CONFLICT);
    }

    return rb.build();
}
```

_2. Next, you will work on a web page and a JavaScript client that will issue the PUT request. To give you a running start, we've provided you with a starter file. In Windows Explorer, browse to **{Lab Installation Directory}/starters/lab12**, then copy and paste *UpdateCustomer.html* into the Eclipse project's *WebContent* folder.

Open UpdateCustomer.html in Eclipse and note:

- In the *head*, there's a script element that references the *axios* JavaScript AJAX library and another script element containing an empty JavaScript function named *updateCustomer()*. You will complete this function shortly.
- Also in the *head*, there's some simple Cascading Style Sheet (CSS) rules to style the form in a rudimentary fashion.
- In the *body*, there's a *div* with the ID *result* into which your script will write a status message that reflects whether the AJAX request was successful or not.
- Also in the *body*, there's a *form* - note how the form defines *input* elements for each of the Customer properties. Also note how the form's *onsubmit* event handler references the updateCustomer() function - when the user presses the form's submit button, JavaScript will call the function so you can harvest data from the form and issue a PUT AJAX request.

_3. Complete the *updateCustomer()* function:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. First, create an empty JavaScript object, then populate it with data from the HTML form:

```
let customer = {};
customer.id = updateCustomerForm.customerID.value;
customer.name = updateCustomerForm.customerName.value;
customer.address = updateCustomerForm.address.value;
customer.city = updateCustomerForm.city.value;
customer.state = updateCustomerForm.state.value;
customer.zip = updateCustomerForm.zip.value;
```

b. Use axios to issue a PUT to the service, passing the customer object in the request body. When the response comes back, update the *result* div appropriately:

```
axios.put('rest/customer', customer)
  .then(function (response) {
      document.getElementById("result").innerHTML =
          "Customer updated successfully"
  })
  .catch(function (error) {
      document.getElementById("result").innerHTML =
          error.message;
  });
```

_4. Test your code by right-clicking on UpdateCustomer.html and choosing *Run As - Run on server*. Wait for the web page to appear in the browser, then fill in data for an existing customer (use ID of 12, for example). Press the submit button and observe the results.

You can verify that the update worked using the GetCustomer.html page you wrote in the last part of lab.

Note: Since we are using an in-memory customer List that's recreated every time you republish the application on the server, if you change the application code and save it, the app will reload the original hard-coded customer list - in other words, the update isn't really permanent like it would be in an actual database.

Part 3: Completing CRUD

In this part, you will complete the application by creating web pages and JavaScript that implement Create (POST) and Delete (DELETE).

Steps:

_1. Add a new method to the CustomerService.java class that responds to an HTTP POST, accepting a JSON representation of a customer. The method should ensure that that the specified customer doesn't exist in the List, and then add it. If successful, return 201 Created with a Location header with the new customer's resource URL. In that case, also return a complete customer representation in the response body. If the customer already exists, return 400 Bad Request.

Create a web page named **CreateCustomer.html** that does an HTTP POST via AJAX, retrieving the customer information from an HTML form. Use the UpdateCustomer.html page you wrote in the last part for inspiration.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Note: In axios, if the response is successful, you can access the *location* header using the expression **response.headers.location**. You can display the location header in the *result* div.

Note: Since we are using an in-memory customer List that's recreated every time you republish the application on the server, if you change the application code and save it, the app will reload the original hard-coded customer list - in other words, the insert isn't really permanent like it would be in an actual database.

_2. Add a new method to the CustomerService.java class that responds to HTTP DELETE, accepting the ID of the customer to delete as a path parameter. Search the customer list for a Customer with the specified ID - if found, delete it from the List and return 204 No Content. If not found, return 404 Not Found.

Create a web page named **DeleteCustomer.html** that deletes a Customer. It's OK to hard-code the customer ID in the same fashion as you did in GetCustomer.html, or if you're ambitious, update GetCustomer.html so that the user enters the customer ID into a form and use the same technique in DeleteCustomer.html.

Note: Since we are using an in-memory customer List that's recreated every time you republish the application on the server, you can get the deleted customer "back" by removing and re-adding the project to the JBoss server.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 13: Additional JAX-RS Topics

In this lab, you will practice JAX-RS techniques.

Objectives:

- To use the JAX-RS client API
- To write a JAX-RS filter.

Part 1: Writing a JAX-RS Client

In this part, you will write a client using the JAX-RS Client API. While most RESTful clients are written in JavaScript, it can be useful to write Java clients that retrieve RESTful resources and perform data processing.

Steps:

_1. First, create a simple service that your client can access. The service will provide a hard-coded list of JSON data that represents segments of airline travel.

   a. Create a new Dynamic Web project named **Lab13Web** associated with **Lab13EAR**.

   b. A TravelSegment represents a portion of an airline reservation. Create a class named **TravelSegment** in a new package named **domain** with the following fields:

```
private int segmentID;
private String startingAirport;
private String endingAirport;
```

      Use Eclipse's *Source* menu to generate:

      - Get/set methods for all of the fields
      - Constructor from fields (all fields)
      - Constructor from superclass (zero-argument constructor)
      - toString()

   c. Create a class named **TravelService** in a new **service** package. Complete the new class:

      - Annotate the class with **@Path("segments")**.
      - Write a method named **getAllSegments** that accepts no arguments and returns a hard-coded List of TravelSegment objects. Annotate the method with **@Path("all")**, **@GET** and **@Produces(MediaType.APPLICATION_JSON)**. Your code might look like:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@Path("all")
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<TravelSegment> getAllSegments()
{
    List<TravelSegment> segments = new ArrayList<>();

    segments.add(new TravelSegment(12, "ORD", "SFO"));
    segments.add(new TravelSegment(14, "SJC", "SEA"));
    segments.add(new TravelSegment(17, "CID", "DEN"));

    return segments;
}
```

d. Create a class named **TravelApplication** in the *service* package that extends javax.ws.rs.core.Application. Override the *getClasses()* method and return a HashSet containing a reference to the TravelService class.

   Annotate the class with **@ApplicationPath("rest")**.

e. Add Lab13EAR to the JBoss server, then test your service by browsing to the resource URL for all segments to see the JSON. Note the URL for when you write the client.

_2. Now let's work on a client that uses the JAX-RS client API. First, create a new Java project named **Lab13Client**. Decline Eclipse's request to switch perspectives.

_3. Next, set up the client's CLASSPATH to reference the JAX-RS client API JARs and the TravelSegment:

a. Right-click on the Lab13Client project and choose *Build Path - Configure Build Path*, then click the *Libraries* tab.

b. Press the *Add External JARs* button, then browse to **{Lab Installation Directory}/lib/rest-client**. If your computer has the standard lab setup, this directory is:

```
c:\Users\username\websrvclass\lib\rest-client
```

   Substitute your actual user name.

   Select all of the JARs in this folder, then press Apply followed by Close. You should see the JARs in the *Referenced Libraries* folder in the Lab13Client project.

c. Copy and paste the *domain* package from the Lab13Web project to the new project's *src* folder.

_4. Create a new class named **SegmentsClient** in a new package named **client**. Select the checkbox to create a *main()* method.

Using the example in the Additional JAX-RS Topics chapter as a guide, complete *main()* so it:

a. Creates an instance of javax.ws.rs.client.Client, using the javax.ws.rs.client.ClientBuilder.

b. Creates an instance of javax.ws.rs.client.WebTarget, referencing the URI for the service.

c. Retrieves the List of TravelSegments from the service:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
List<TravelSegment> segments = webTarget.request("application/json")
    .get(new GenericType<List<TravelSegment>>(){});
```

Note how we create an inline subclass of javax.ws.rs.core.GenericType: this enables the JAX-RS client runtime to determine the type of objects in the List.

d. Iterates the *segments* collection, printing each TravelSegment to the console.

```
for (TravelSegment seg : segments)
{
    System.out.println(seg);
}
```

_5. Run the client as a Java Application - you should see the hard-coded list of TravelSegments that your client retrieved from the service.

Part 2: Writing a Server-Side Filter

In this part, you will write a JAX-RS filter that runs on a server, intercepting requests to the service you created in the last part of lab.

Your filter will log all of the HTTP headers in the request to the console.

Steps:

_1. Create a new class named **HeaderDumpFilter** that implements *javax.ws.rs.container.ContainerRequestFilter*. Put the class in a new package named **filters**. **Hint:** When you create the class, click the Add button for *Interfaces* and start typing **ContainerRequestFilter**, then select *ContainerRequestFilter - javax.ws.rs.container*.

For readability, rename the argument to the *filter* method to **reqCtx**.

_2. Annotate the class with @Provider.

_3. Add the HeaderDumpFilter.class to the set of classes returned by the getClasses() method in TravelApplication.java.

This step really shouldn't be necessary since you annotated the filter class with @Provider, but apparently the JBoss JAX-RS runtime requires it.

_4. Complete the filter() method:

a. Retrieve the list of HTTP headers passed in the request:

```
MultivaluedMap<String, String> headers = reqCtx.getHeaders();
```

MultivaluedMap is a collection in which the each key is associated with a List. In this case, the keys are header-name strings and the List for each key is the set of header-value strings for that key.

b. Retrieve the set of header-name keys from the map:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Set<String> headerNames = headers.keySet();
```

c. Iterate the header names, and then iterate the values for the given header. Print a formatted string to the console:

```
for(String headerName : headerNames)
{
    List<String> headerValues = headers.get(headerName);

    for(String headerValue : headerValues)
    {
        System.out.println(headerName + "->" + headerValue);
    }
}
```

_5. Test your filter by browsing to the resource URL to fetch all segments in a browser. You should see the header names and values in the Eclipse console.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 14: JAX-RS Security

In this lab, you will secure a simple RESTful service using JWT.

Objectives:

- To use token-based security for a stateless RESTful service.

Part 1: Creating the Service

Steps:

_1. Create a new Dynamic Web project named **Lab14Web** associated with *Lab14EAR*. On the last page of the wizard, put a checkmark on the *Generate web.xml deployment descriptor* option.

Note: If you forget to select the option to generate the web deployment descriptor, right-click on the Lab14Web project and choose *Java EE Tools - Generate Deployment Descriptor Stub*.

_2. Create a new class named **SavingsAccount** in the *domain* package.

Complete the class so it defines fields for **accountNum (int)**, **balance (double)** and **holderName (String)**. Generate get/set methods for the fields and a *toString()* method.

Also generate a constructor that accepts arguments for each of the fields.

_3. Create a new class named **AccountService** in the **service** package.

_4. Complete the class:

a. Write an @Path annotation on the class with **/accounts** as the URL.

b. Write an @Produces annotation on the class so that it produces *application/json* content.

c. Write a simple method that returns a hard-coded SavingsAccount domain object. Annotate the method so it responds to a **GET** request with a PathParam for the account number. Your code might look like:

```
@GET
@Path("/{accountNum}")
public SavingsAccount getAccount(@PathParam("accountNum") int accountNum) {
    return new SavingsAccount(accountNum, 44.55, "Sam Smith");
}
```

_5. Create a class in the *app* package named **AccountApplication** that extends *javax.ws.rs.core.Application*.

Complete the class:

a. Annotate the class to provide a prefix for the RESTful URL:

```
@ApplicationPath("/rest/")
```

b. Use the *Source - Override/Implement Methods* menu to override the *getClasses* method.

You can delete the generated call to the superclass.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. Add code to the method so that it creates a Set containing all of your RESTful sources (only one in this case):

```
Set<Class<?>> s = new HashSet<Class<?>>();
s.add(AccountService.class);
return s;
```

_6. Deploy the application by adding the EAR to the JBoss server.

_7. Test your work so far by opening a Web browser and entering this URL:

```
http://localhost:8080/Lab14Web/rest/accounts/12
```

Part 2: Securing the Service with JWT

In this part, you will use the *jjwt* library to secure the service using a Java Web Token (JWT).

Steps:

_1. First, add the *jjwt* JARs to the project's class path:

a. In Windows Explorer or My Computer, browse to **{Lab Installation Directory}/lib/jwt**. If your computer has the standard lab setup, this directory is:

```
c:\Users\username\websrvclass\lib\jjwt
```

Highlight all seven of the JARs there, then press Ctrl+C to copy to the clipboard.

b. Back in Eclipse, in the *Lab14Web* project, expand *WebContent/WEB-INF*, then right-click on the *lib* folder and choose *Copy* to copy the jjwt JARs to the project's class path.

Note: Be sure to copy the JARs to the *lib* folder, not the WEB-INF folder!

_2. To give you a running start, we will provide you with a partially completed HTML file that uses the *axios* JavaScript library to perform AJAX requests to your service. Import the HTML file:

a. In Windows Explorer or MyComputer, navigate to **{Lab Installation Directory}/starters/lab14** and select *index.html* and press Ctrl+C to copy to the clipboard.

b. Back in Eclipse, in the Lab14Web project, right-click on the *WebContent* folder and choose Paste to import the HTML file.

Examine the starter HTML file and note:

- Within the *head* element, it defines two *script* elements: one that brings in the *axios* AJAX JavaScript library, and one that contains empty JavaScript functions: *login* and *getAccount*. You will complete the JavaScript functions in later steps.
- Within the *body* element, there are two *sections*. The first section defines a form into which a user can enter login credentials and submit them - note how the form's *onsubmit* property invokes the JavaScript *login()* function. The second section contains a button to retrieve Account information from your JAX-RS service - note how the button's *onclick* property invokes the JavaScript *getAccount()* function.

_3. Let's start by completing the JavaScript *login()* function:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. In the *script* element for the two functions, ABOVE the functions, define a global JavaScript variable to hold the JWT:

```
var jwt;
```

b. Within the curly braces for the login() function, above the *return* statement, retrieve the data that user enters into the login form:

```
const username = document.loginForm.username.value;
const password = document.loginForm.password.value;
```

c. Still above the *return*, create a JavaScript object that lets you pass form parameters in an AJAX request:

```
const params = new URLSearchParams();
params.append('username', username);
params.append('password', password);
```

d. Still above the *return*, use the axios library to issue an AJAX request and process the response, retrieving the return JWT and storing it in the global variable, then updating the web page to show the user the result:

```
axios.post('rest/accounts/login', params)
    .then(function(response) {
        jwt = response.data;
        document.getElementById("loginResponse").innerHTML =
           "Login successful";
    })
    .catch(function(response) {
        document.getElementById("loginResponse").innerHTML =
           "Login failed";
    });
```

Note how the request is a POST to the *rest/accounts/login* URL. You will write a method in the JAX-RS AccountService in a moment to respond to this request.

_4. Next, complete the JavaScript *getAccount()* function:

a. Within the curly braces for the getAccount() function, start by defining an object that contains the JWT embedded in the HTTP *Authorization* header:

```
const config = {
    headers: {"Authorization": "Bearer " + jwt}
};
```

b. Use axios to send an AJAX request to your service's getAccount() method for Account number 12:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
axios.get('rest/accounts/12', config)
.then(function (response) {
 // handle success
    document.getElementById("accountInfo").innerHTML =
        JSON.stringify(response.data);
})
.catch(function (error) {
 // handle error
 document.getElementById("accountInfo").innerHTML =
       error.message;
})
```

_5. Next, you will update AccountService.java to support the login request and JWTs.

a. Within the AccountService class, above the getAccount() method, define a *static* block to create, initialize and digitally sign a JWT using the *jjwt* library:

```
private static final Key key;

static {
    key = Keys.secretKeyFor(SignatureAlgorithm.HS256);
}
```

Import *io.jsonwebtoken.SignatureAlgorithm* and *java.security.Key*.

b. Within the class, write the stub of a new method that responds to the AJAX login request:

```
@POST
@Path("/login")
public String login(@FormParam("username") String username,
                    @FormParam("password") String password) {

}
```

Note how we use @FormParam so that JAX-RS extracts form parameters from the request and assigns them to method parameters. Also note how the @Path URL matches the URL you earlier coded in the JavaScript login function.

c. Complete the login() method so that it:

- Checks the user name and password against hard-coded values on which you decide.
- If the credentials are OK, create a JWT and return it from the method. To create a JWT using *jjwt*, you can use code like:

```
String jws = Jwts.builder()
   .setSubject("Joe")
   .signWith(key)
   .compact();
```


Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

- If the credentials are bad, throw a new NotAuthorizedException with the string "Login failed".

_6. Next, update AccountService's getAccount() method to check for a proper JWT:

a. Add another parameter to the getAccount() method so the method can access the HTTP authorization header:

```
public SavingsAccount getAccount(@Context HttpHeaders headers,
    @PathParam("accountNum") int accountNum) {
```

b. In the method, above the existing code, retrieve the header:

```
String authorizationHeader =
   headers.getHeaderString(HttpHeaders.AUTHORIZATION);
```

c. Ensure there's a proper Authorization header and then extract the JWT:

```
if (authorizationHeader == null ||
    !authorizationHeader.startsWith("Bearer ")) {
      throw new NotAuthorizedException("No authorization header");
}

String token = authorizationHeader.substring(
    "Bearer".length()).trim();
```

d. Ensure that the JWT is properly formed, complete with two separator characters (the period):

```
long count = token.chars().filter(ch -> ch == '.').count();
if (count != 2) {
    throw new NotAuthorizedException("Invalid JWT");
}
```

e. Use *jjwt* to parse the token, extracting the claim *subject*. In this simple example, we won't do any authorization based on the subject, so simply print it to the console:

```
Jws<Claims> jws = Jwts.parserBuilder()
    .setSigningKey(key)
    .build()
    .parseClaimsJws(token);
String subject = jws.getBody().getSubject();
System.out.println("In getAccount(), subject: " + subject);
```

f. Highlight the lines above and the *return* statement. Then from the Eclipse menu, choose *Source - Surround with - try/multi-catch block*.

In the catch block, delete the generated code and replace it with:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
        throw new NotAuthorizedException("Invalid JWT signature");
```

_7. Save all files, then open a browser and enter the following URL:

```
http://localhost:8080/Lab14Web/index.html
```

Test your service:

a. First try clicking the Get Account button, without logging in - you should see an "Request failed with status code 401" message.

b. Enter the user name and password that you hard-coded in the AccountService login() method, then click the Login button. You should see a "Login successful" message.

Then retry the Get Account - it should work now.

Back in Eclipse, look in the Console pane - you should see the login user name displayed.

Part 3: Writing a Security Filter

In this part, you will improve the AccountService by moving the JWT-checking code to a JAX-RS filter so that service methods are simpler.

Steps:

_1. Start by adding another resource method to the service. Open AccountService.java and write and annotate a method that returns "all" SavingsAccounts (you can hard-code a list and return it). You don't need to write any JWT verification code in this method at this point.

Test the new method in a browser.

_2. Now think about securing this new method with JWT and all of the code that you'd need to copy and paste from the getAccount() method - clearly that's wasteful and error-prone - what if your service had ten methods instead of two? What happens if you need to modify the JWT-verification code? What if you have multiple resource classes?

The better approach is to externalize the JWT-verification code into a JAX-RS filter so it can apply to ALL methods.

a. Start by creating a new class named **AuthorizationFilter** that implements the *javax.ws.rs.container.ContainerRequestFilter* interface. Put the new class in a package named **filters**.

Since the filter implements ContainerRequestFilter, it will need to provide the *filter* method:

```
public void filter(ContainerRequestContext reqCtx)
   throws IOException {

}
```

b. Annotate the filter class with @Provider.

c. Add a little diagnostic to the filter() method so we can see what it's doing by looking in the Eclipse console:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
String resourcePath = reqCtx.getUriInfo().getPath();
System.out.println(
   "Auth filter, processing request for " + resourcePath);
```

Then add a check for the login() URL - since it provides the JWT, we don't want the filter to work on it:

```
if (resourcePath.endsWith("/login"))
    return;
```

_3. The next step normally would be to MOVE the JWT code from the getAccount() method in AccountService to the filter, but there's a problem - we need to share the secret encryption key between the filter and the AccountService (the AccountService will keep the login() method that creates the JWT).

The approach we'll use is to define a separate Singleton class that keeps the key. The easiest, and perhaps best way to write a Singleton is to use an *enum*. Follow these instructions:

a. Right-click on the project and create an enum named **KeyEnum** in the *util* package.

Modify the enum so it looks like:

```
public enum KeyEnum {
 theInstance;

 private final Key theKey;

 public Key getTheKey() {
  return theKey;
 }

 private KeyEnum() {
  theKey = Keys.secretKeyFor(SignatureAlgorithm.HS256);
 }
}
```

b. In AccountService, delete the *key* field and the *static* block that initializes it. That will generate a couple of errors where the key is referenced.

c. To fix the error in the login() method, at the top of the method, retrieve the key:

```
Key key = KeyEnum.theInstance.getTheKey();
```

You can ignore the error in the getAccount() method for now.

_4. Next, move the JWT-verification code from the getAccount() method in AccountService into the filter() method in AuthorizationFilter. There will a couple of things you'll need to fix in AccountService:

- The only code remaining in getAccount() should be creating a hard-coded SavingsAccount object and returning it.
- You can remove the *@Context HttpHeaders headers* argument from the getAccount() method.

Things to fix in AuthorizationFilter:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

- You'll need to get the key from the KeyEnum in the filter in the same way you did in the login() method.
- A filter can retrieve headers from the ContainerRequestContext parameter:

```
String authorizationHeader =
  reqCtx.getHeaderString(HttpHeaders.AUTHORIZATION);
```

_5. Open the app.AccountApp class and add the AuthorizationFilter class to the set of classes.

_6. Test your program as you did in the last part of the lab via the index.html page.

Note: You will not be able to successfully run the "get all accounts" method at this point from the browser, since there's no easy way to specify the Authorization header in a browser (i.e. it will fail with a 401 error). As far as I'm concerned, the 401 error is indication enough that the filter's working as designed, but there are a couple of ways you could successfully test it if you are determined to do it:

- Use the cURL utility from the command line to test the http://localhost:8080/Lab14Web/rest/accounts/all URL. You will likely need to download cURL from https://curl.haxx.se/windows/. Read to cURL docs to see how to specify an HTTP request header. You'll also likely need to modify the Java code to print the JWT to the Eclipse console.
- Update the index.html page so it makes the *rest/accounts/all* as an AJAX request using Axios in a similar fashion to how existing JavaScript code in the getAccount() function. You'll have to define a button to kick the request off and figure out how to display a list of items in a web page via JavaScript!

_7. As it stands, the application sends the JWT between the browser and the server essentially unencrypted (JWT uses Base64URL encoding, which is trivial to decode). That means that an attacker could potentially use a network "sniffer" to intercept traffic and decode the JWT and potentially mount an attack.

To mitigate this, we can route all application traffic over HTTPS, which strongly encrypts all requests and responses.

The good news is that JBoss includes a self-signed certificate suitable for testing HTTPS. You can demonstrate it by browsing to:

```
https://localhost:8443/Lab14Web/index.html
```

You will need to bypass the browser's warning about an invalid certificate, but note that the request and response are now encrypted. If we were doing this in the real world, we'd obtain a proper (not self-signed) certificate and install it into JBoss, but that's too much work for this lab.

_8. **OPTIONAL**: Forcing HTTPS

The above works, but note that the user can still use "normal" HTTP as you've done in previous steps.

To force the application to always run on HTTPS, open the *WEB-INF/web.xml* file, and after the close of the welcome-file-list, configure the application to require HTTPS:

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
 <security-constraint>
   <web-resource-collection>
       <web-resource-name>All</web-resource-name>
       <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
 </security-constraint>
```

_9. Save the file and let JBoss reload.

Then enter in your browser:

```
http://localhost:8080/Lab14Web/index.html
```

You may need to do a "hard reload" on your browser (typically by pressing Ctrl+Shift+r), but you should see the URL change to the HTTPS equivalent.

If you're ambitious, in Eclipse, configure a TCP/IP monitor to on port 8444, forwarding to 8443 and then test again and see the encrypted traffic in the monitor.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Case Study Lab

In this lab, you will write a complete web application that uses most of the techniques you learned in the class. We do not expect you to finish this lab in the time allotted, but you should be able to architect the system and start working on it.

Here's the situation. You work in Information Technology for *Bike Trails Forever*, a large, non-profit organization that accepts donations from individuals. In the past, BTF has used a standalone PC-based system to keep a record of donations, but in this day and age, the officers are demanding a new, Web-based solution. You have been given the job of creating such a system.

Here are the specifications:

- For each donation, you must track the date and amount, as well as the donor's name and address information
- A single individual might make several donations over time -- your program needs to track each one separately
- To save time, we will provide you with the web page for the application - your job is to code the RESTful resources

Objectives:

- To design and implement a RESTful service from scratch

Steps:

_1. Since the goal of this lab is to let you practice the techniques you learned in class, this lab guide will not give you complete step-by-step instructions as in earlier labs, but we will provide the steps to to help you get started.

Start by importing a starter project that includes the web pages for the application:

a. In Eclipse, from the menu, choose *File - Import* to start the wizard. On the first page, select *General - Existing Projects into Workspace*, then press Next.

b. On the next page, click the *Select archive file* button, then browse to **{Lab Installation Directory}/ starters/lab15** and select *lab15.zip*. Press Open followed by Finish - you should see *Lab15Web* and *Lab15EAR* projects in the Project Explorer.

Note: If there's a red X indicating errors on the Web project, right-click on *Lab15Web* and choose *Build Path - Configure Build Path*, then click the *Libraries* tab. If you see a red X on the JRE System Library entry, double-click it and choose *Workspace default JRE* then press Apply followed by Close.

c. Examine the imported Lab15Web project's WebContent folder - it has several JavaScript files that use the React library to create a single-page web application that uses *axios* to fetch RESTful data. Open each JavaScript file and find the axios.get() lines and note the resource URLs - your job is to write a JAX-RS service that provides these resources. Write down each of the resource URLs for future use.

d. The WebContent folder also has a *sample.json* file that you can open to see what kind of data the web pages expect. Examine it closely - the RESTful service you will write must provide JSON in this basic format.

e. Even though there's no RESTful service yet, you can right-click on *index.html* and run it on the server. You will see a web page that is partially functional. When you are finished writing your RESTful service, the page should display a list of donors and have the ability to search for and display a list of donations for a given donor.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_2. Using the techniques you learned in this class, create a JAX-RS service that responds to the URLs you noted above, providing donor and donation data as shown in the sample JSON. To save time, you can hard-code the donors and donations.

Good luck!

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.