Lab 6: 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 **lab06Web** associated with *lab06EAR*.

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

This class should be written in the form of a JavaBean, exposing the following properties via get/set methods:

```
Property name        Property type
-------------        -------------
cardNumber           String
expiryDate           java.util.Date
amount               double
```

Hint: If you define fields (instance variables) with the names and types shown above, you can use the RAD *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**.

Complete the class:

a. Define a field that contains the card number:

```
private String cardNumber;
```

Generate get/set methods for the field.

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

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

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
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/lab06/starters.txt** into the ValidatePurchase class.

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

```
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 using the following pseudo-code as a guide:

```
If expiryDate or cardNumber is null, return false.

If amount is less than equal to zero or greater than 100,000,
   return false.

If the cardNumber is not valid, throw a new InvalidCardNumberException.

Otherwise, return true.
```

Note that you can call the *checkNumber* method to determine if the cardNumber is valid.

_5. Ensure that the server is running, then add the *lab06EAR* project to the server (you can remove any older projects).

_6. Now let's do a quick test.

a. As in earlier labs, start the Web Services Explorer.

b. 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.

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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.

c. 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.

d. 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 for the credit-card-validation service.

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 *@SOAPBindingImpl* annotations from the endpoint class to the interface.

_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 @SOAPBindingImpl annotation.

_5. Republish, then use the Web Services Explorer to verify that the service works as before.

Part 3: Using Annotations

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

Steps:

_1. In the Web Services Explorer, in the Navigator pane, click on the *ValidatePurchaseImplPortBinding* entry, then copy and paste the service endpoint URL into the clipboard.

_2. Open an external Web browser and enter the service endpoint URL you noted in the previous step, appending the text **?wsdl** to display the service's WSDL.

   Write down the following:

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
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:                 _____________________________________
```

_3. Your job is to use annotations on the endpoint class and the endpoint interface to customize the WSDL as follows:

```
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
```

_4. 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:

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
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 URL, and thus the WSDL's URL will change. Close any Web Services Explorer window, then re-test with the WSE. Note the new service URL and copy it to the clipboard. Then in your external browser, use the service URL with **?wsdl** appended.

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

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

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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.

_4. If you have time, write a standalone Java client that invokes the service.

Part 5: Service Lifecycle and Stateful Services

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

Steps:

_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:

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
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:

```
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 **lab06Client**.

_9. Generate the client "stub" files into the new project for the shopping-cart service. See Lab02 if you cannot remember how to do this.

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

_10. In the *lab06Client* project, create a new class named **ShoppingClient** in the *main* package with a *main* method. Complete the *main* method:

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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.

Copyright © Descriptor Systems 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.