Lab 7: Writing a JAX-RPC 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 RAD menu, choose *File - Import - Java EE - EAR file*, then press Next.

   b. For the *EAR file*, Browse to **{Lab Installation Directory}/starters/lab07**, then select **lab07ServiceEAR.ear**. If your computer has the standard lab setup on Windows XP, the directory is:

      ```
      c:\websrvclass\starters\lab07
      ```

      On Windows Vista and later:

      ```
      c:\Users\username\websrvclass\starters\lab07
      ```

   c. Press Finish to complete the import. You can decline any invitation to switch perspectives.

_2. Deploy the service by right-clicking on the WebSphere server and choosing *Add and Remove*, then moving *lab07ServiceEAR* to the configured list. You can unconfigure any earlier projects.

_3. 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.

   You can browse the WSDL in a Web browser at:

   ```
   http://localhost:9080/lab07ServiceWeb/AgentManagerService?wsdl
   ```

   And browse the service's schema at (perhaps in a new browser tab):

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

```
http://localhost:9080/lab07ServiceWeb/AgentManagerService/
   AgentManagerService_schema1.xsd
```

Note that we have split the schema's URL into separate lines so it fits on the printed page.

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

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.

_4. Still examining the WSDL, use the following space to write a Java interface that corresponds to the *portType*:

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

Part 2: Using a Proxy from a Standalone Client

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

Steps:

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

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

   b. Enter **Lab07ProxyClient** for the *Project name*, then press Finish.

      Decline any invitation to switch perspectives.

_2. Next, generate a Java *proxy* for your Web service into the new project. A proxy or stub is a client-side Web services "stand-in" for the remote service. Follow these steps:

   a. Right-click on the lab07ProxyClient project and choose *New - Other - Web Services - Web Services Client* and press Next to start the wizard.

   b. On the first wizard page, for the *Service definitions*, enter:

      ```
      http://localhost:9080/lab07ServiceWeb/AgentManagerService?wsdl
      ```

      Note that you might be able to copy and paste this URL from the Web browser.

      Press Next.

   c. On the next wizard page, note the defaults, then press Finish.

   d. In the Enterprise Explorer, expand the *lab07ProxyClient* project and look in the *service* package -- these generated files comprise the Java *proxy* for your Web service.

_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.

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

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:

- 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 **lab07ClientWeb** associated with *lab07ClientEAR*:

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

b. On the first wizard page, enter **lab07ClientWeb** 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 **lab07ClientEAR**.

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.

Answer No when prompted to switch to the JEE Perspective.

_2. In the same fashion as in the last part, generate a client stub into the lab07ClientWeb project.

Look in the Web project's *Java Resource/src* folder for the *service* package with the generated classes.

_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 2014. 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 *lab07ClientWeb* 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 2014. 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. Here's some code to help you get started:

```
AgentManager proxy =
```

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 =
out.println("Annuity: " + annuity);
```

Note: The calcAnnuity operation parameters should be in this order: *payment*, *interestRate*, *periods*.

_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 RAD TCP/IP monitor and use the BindingProvider interface to change a proxy's endpoint.

Steps:

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

a. From the RAD 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 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Local monitoring port:   9081
Host name:               localhost
Port:                    9080
Type:                    HTTP
```

Press the OK button. This configures the monitor to listen for HTTP requests on port 9081 and forward them to port 9080. Port 9080 is the standard port on which the WebSphere 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 **lab07MonitorClient**.

_3. In the same fashion as you did earlier, generate client stubs into the new project.

_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:9081/lab07ServiceWeb/AgentManagerService");
```

Note the 9081 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 RAD's console and the TCPMonitor should've intercepted the request. Look around in the RAD 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:9081* entry and click on the *lab07ServiceWeb/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.

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

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

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 **lab07CredentialClient**.

_2. In the same fashion as you did earlier, generate client stubs into the new 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 2014. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.