Lab 4: 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 RAD if it's not running.

Ensure that RAD is in the Web Perspective by choosing *Window - Open Perspective - Web*.

_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 **lab04Web** for the *Name*. In the *EAR membership* section, put a checkmark in the *Add project to an EAR* box, and the For the *EAR Project Name*, enter **lab04EAR**.

Press Finish.

Answer No if prompted to switch to the JEE Perspective.

You can close the Quick Starts window.

_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 *lab04Web* 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)
public class SampleService ....
```

d. From the RAD menu, choose *Source - Organize Imports* to import the types for the annotations:

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

```
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;
```

e. Save to compile.

_4. Now you can deploy your Web service: In the RAD *Servers* tab at the bottom of the window, right-click on the WebSphere Application server and choose *Add and Remove* and move *lab04EAR* to the Configured list.

a. In the RAD *Servers* tab at the bottom of the RAD window, if the server's not running, right-click on the WebSphere server and choose *Start*, then wait for the server to start completely.

b. Right-click on the WebSphere Application Server again and choose *Add and Remove* and move *lab04EAR* to the Configured list. You can remove any previous projects.

_5. Now you can examine your Web service. Follow these steps:

a. Open an external browser, e.g. Internet Explorer, and enter the following URL:

```
http://localhost:9080/lab04Web/SampleServiceService?wsdl
```

b. Look in the WSDL's *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?

You can close the WSDL when you are finished looking at it.

_6. Next, test the new Web service. Follow these steps:

a. In the Enterprise Explorer, right-click on the *{http:sample/}SampleServiceService* entry and choose *Test with Web Services Explorer*.

b. The Web Services Explorer (WSE) is a generic Web services client provided by RAD 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 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.

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

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

d. 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:q0="http://sample/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:calcVoltage>
      <arg0>3</arg0>
      <arg1>4</arg1>
    </q0:calcVoltage>
  </soapenv:Body>
</soapenv:Envelope>
```

_7. 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: _____

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:

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

```
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 the Web Services Explorer.

_8. Now let's redeploy the service with the *Document/literal (wrapped)* style:

a. Back in RAD, edit SampleService.java.

Modify the SOAPBinding annotation so it looks like:

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

Import types as necessary and 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.

_9. In the same fashion as before, open a Web browser window with the WSDL URL (or refresh the existing window).

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:import* element. Find the *schemaLocation* attribute's value, then copy the URL into the clipboard. The URL should look something like:

```
SampleServiceService_schema1.xsd
```

Open a new browser window (or tab) and paste the schema's complete URL into the URL bar:

```
http://localhost:9080/lab04Web/SampleServiceService/
   SampleServiceService_schema1.xsd
```


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

Note: Please enter the above URL on a single line into the browser's address bar. We have split it here so it fits on the printed page.

c. Looking at the schema, what *elements* are defined?

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

d. What *complexTypes* are defined?

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

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.

_10. 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.

_11. Close the WSE and any open Web browser windows and tabs.

_12. Now let's attempt to redeploy the service with the *Document/literal (bare)* style and use.

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

Save and publish.

b. In the RAD console, scroll back to see error messages about the publishing. You should see something like:

```
runtime modeler error: SEI sample.SampleService has method calcVoltage
annotated as BARE but it has more than one parameter bound to body.
This is invalid. Please annotate the method with annotation:
@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```

This clearly shows us that according to the JAX-WS specification, "bare" services can only accept one parameter. Recall that "bare" services are most often used for document-oriented services.

_13. Change the service's style and use back to *rpc/literal* and republish.

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 *Lab04Web* project, create a new class that represents an insurance policy named **LifePolicy** in a new **insurance** package, then complete the class:

a. Add the following fields:

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


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

b. Use the RAD 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:

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.

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.

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 2014. 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*.

Note: There is currently a bug in the IBM JAX-WS runtime that prevents arrays from working in Web services, so if you try testing the service, it will not work.

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 *lab04Web* 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:

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

_2. Edit SampleService.java and re-work the *ratePolicy* method 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.

```
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.

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.

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
```

_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.

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

_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 ArrayList<LifePolicy> policies =
    new ArrayList<LifePolicy>();
```

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 after a moment or two.

Look in the RAD Console 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.

Part 6: Monitoring SOAP Messages

In this part, you will use the RAD 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 RAD menu, choose *Window - Preferences*.

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

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:   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 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 *SampleServicePortBinding* 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:9081/lab04Web/SampleServiceService
```

Note how the 9081 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 9081 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 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.

In the top pane, expand the *localhost:9081* entry and click on the *lab04Web/SampleServiceService* entry to highlight it. You should see the request and response messages in the lower panes.

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:

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:

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

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

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