Lab 8: Document Style Messaging

In this lab, you will create a Web service that accepts an XML document and processes the document using DOM. You will then write a client program that uses SAAJ and the JAX-WS Dispatch API.

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

```
<m:policy xmlns:m="http://goliath.com/policy>
  <policyID>12</policyID>
  <policy-amount>100000</policy-amount>
  <effective-date>12-12-2005</effective-date>
</m:policy>
```

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 RAD WSDL editor to create the WSDL for your document-oriented service.

Steps:

_1. Create a new Dynamic Web project named **Lab08Web** associated with *lab08EAR*.

_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* to start the wizard.

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

   b. On the next page, set the *Target Namespace* to **http://goliath.com/policy**, then deselect the *Create WSDL Skeleton* box and press Finish. RAD will create the file and open it in the WSDL editor.

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

c. 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*. RAD 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 - RAD opens another WSDL editor window. Click the Design tab at the bottom of the new editor window.

f. Click the leftmost *processPolicy* symbol (with the 'e') and in the Properties, change its name to **policy**.

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

h. Right-click on the *(policyType)* symbol and choose *Add Element*.

Change the element's name to **policy-id** and its type to **int**.

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

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

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

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

l. 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. Back in the Design view for the policy.wsdl, repeat the above step to set the return document:

a. To the right of the *output* message, there's a right-facing arrow. Click on it so you edit the *processPolicyResponse* element definition - RAD opens another WSDL editor window. Click the Design tab at the bottom of the new editor window.

b. Click the leftmost *processPolicyResponse* symbol (with the 'e') and in the Properties, change its name to **policyResponse**.

c. In the *(processPolicyResponseType)* symbol, rclick on the *out string* entry, then in the Properties, change its name to **underwrite-result**.

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

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

_6. Before continuing, ensure that your port type looks like Figure 1:

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

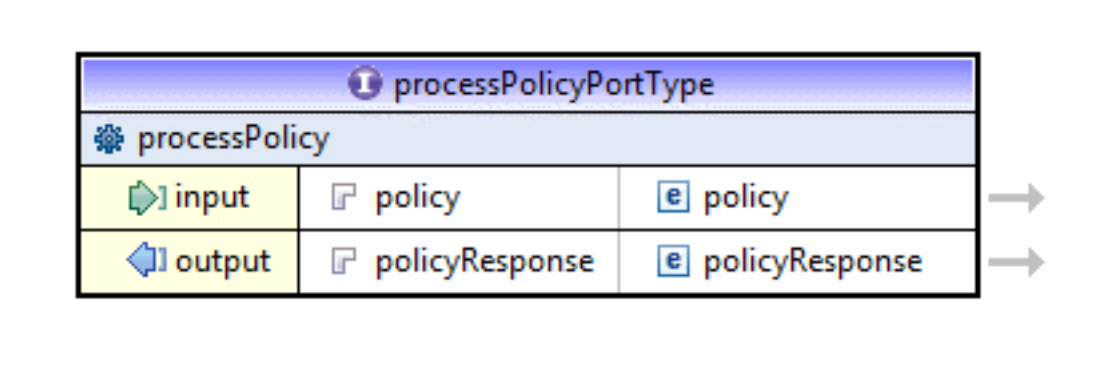


Figure 1: Port Type

Your policy element should look like Figure 2:

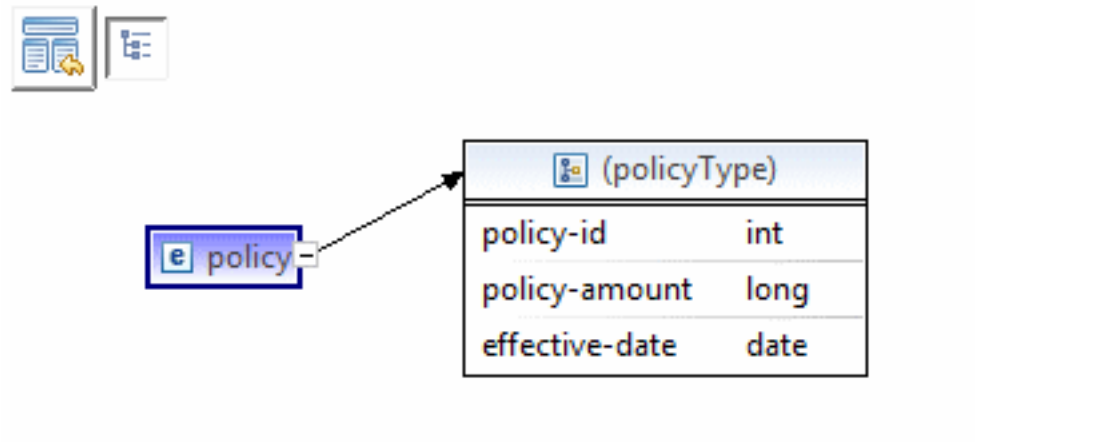


Figure 2: Policy Element

Your policy-response element should look like Figure 3:

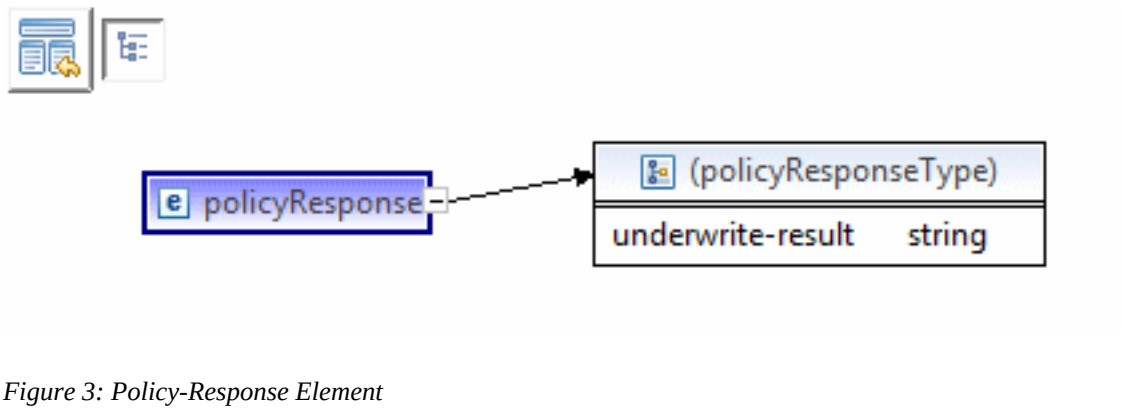


Figure 3: Policy-Response Element

_7. Next, you can create the binding for your port type. Follow these steps:

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

a. In Design View, right click on the white surface of the view and choose *Add Binding* -- RAD 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.

_8. 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 white 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:

```
Name:      ProcessPolicyPort
Binding:   ProcessPolicyBinding
Address:   http://localhost:9080/Lab08Web/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 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>
{

}
```

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:

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

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

_4. Use the *Source - Override/Implement Methods* menu so that RAD writes an empty *invoke* method.

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

_5. 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:

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

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 amtElment = (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.

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.

_6. 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 2014. 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 DOMSouce(doc);
```

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

Copyright © Descriptor Systems 2014. 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 *Source - Surround With - Try/Catch Block* to generate exception-handling code.

You should have no errors at this point.

_7. Next, you can deploy and test your service using the Web Services Explorer:

a. Add the *lab08EAR* project to the server. You can remove any older projects.

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

c. Click on the *processPolicy* Operation, then enter or select the following:

```
policy-id:       12
policy-amount:   100000
effective-date:  12-12-2010
```

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>
```

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

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

Optional Challenge Part 3: Writing a Client

In this part, you will write a Java client that uses SAAJ to build the request document and process the response document.

Steps:

_1. Create a new Dynamic Web project named **lab08ClientWeb** associated with *lab08ClientEAR*.

_2. In the new project, create a servlet named **SendPolicy** in the **servlets** package with a *doGet* method. Using the sample in the book as inspiration, complete the *doGet* method so that it uses SAAJ to send a *policy* document and display the *underwrite-response* to the resulting Web page.

Here are some hints:

- Create a *Dispatch* parameterized to SOAPMessage.
- Create a SOAPMessage containing the *policy* element and its child elements. The *policy* element should be namespace qualified with the service's target namespace, but the the child elements should not be namespace qualified.
- After using the Dispatch *invoke* method to send the request, check for a fault, then iterate to find the underwrite-result element and display its contents to the console.

_3. To test, run the servlet on the server. You should see either the String "Accepted" or "Rejected".

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