UNIVERSITY of STIRLING

Web Services

Practical 2 – Developing a Web Service with WSDL

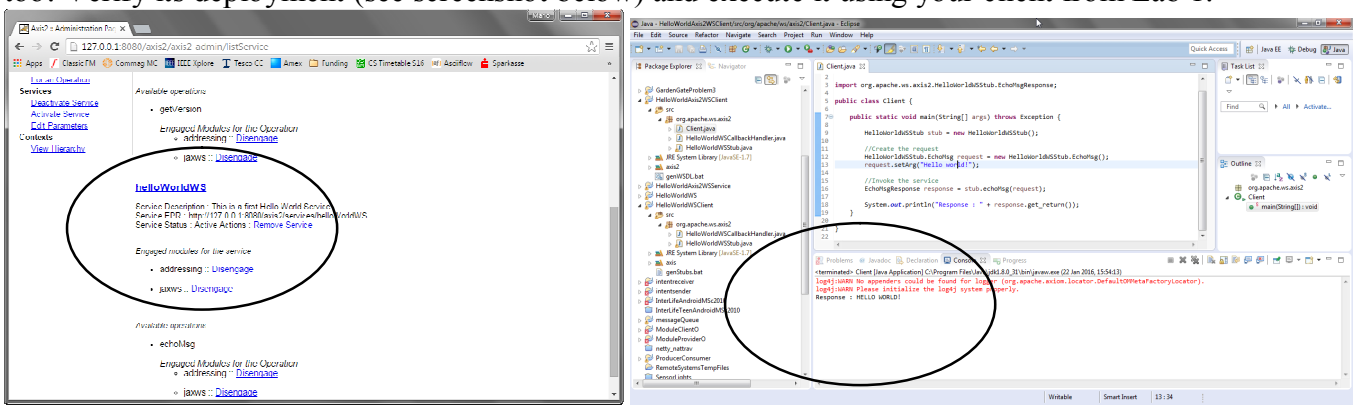
This lab contains a checkpoint towards the end. Please contact any lecturing staff when you reach this point to receive your credit.

It is important that you use the lab session to ask staff any questions you might have on the material. There are no tutorials!
Please do not sit and be stuck!

This practical will give you practice with developing web services. In the first practical, a simple Java class was turned into a web service by providing a service descriptor and packaging it as an .aar file which was then deployed on Axis2. That approach was called POJO (Plain ordinary (or old ☺) Java Object). In this practical you will create a web service and corresponding client starting with the WSDL description and then adding its behaviour in Java. This process is more complicated than the POJO approach from Lab 1. However, it also gives more flexibility.

Verifying the Web Service Environment

Start up Tomcat with Axis2 as in Lab 1. All being well, your Hello World service should still be there too! Verify its deployment (see screenshot below) and execute it using your client from Lab 1.



If you get output of HELLO WORLD in upper case, your web service environment is fine.

Starting with WSDL

In this lab, you will develop an approver service which determines a rate for lending money. The service receives a request with the loan applicant's name, address and required amount. The loan rate depends on all these factors.
Create a new Eclipse project. Name it ApproverWS. Make sure that the axis2 user library of classes you created in Lab 1 is linked to this project (under Build Path and then Libraries). Copy the file Approver.wsdl from the lab2 folder in the CSCU9YW folder on K: into your eclipse project. Refresh your project in eclipse by pushing F5. The wsdl file should now be visible as part of the eclipse project.

Open the wsdl file in eclipse and study it. The file is incomplete, but a skeleton of the abstract part and most of the concrete part with the binding are provided. Albeit being XML, WSDL is a complex language and the files can grow reasonably large very quickly. Thus in practise, you wouldn't start with a blank file, but adapt an existing one. This is the spirit of this step. You will need to provide the parts for the input and output messages together with the datatypes and then include the messages in the porttype definition.

Once you are happy with the WSDL file, you can use the WSDL2JAVA tool, to generate most of the Java code required. Even though XML messages are used for communication between web services and clients, most Web services do not use XML as such. Instead, business data specific to the application is used. Thus XML is simply a convenient format to represent the business data at Web service interfaces. XML works well for this purpose, because it provides a platform-independent interface which can be used by various tools and languages and platforms. However, clearly applications need to convert the XML to or from their own internal data structures to use the data within the application. This is called data binding. It is possible to write your own data-binding code for a web service, but most developers use a data-binding framework that handles this conversion (you will see shortly why!). Axis2 supports a variety of data binding models, here we will use the ADB (Axis2 Data Binding) which is the default with Axis2.

To isolate application logic from the (rather messy) data binding code, Axis2 supports generating data binding code from WSDL held in separate files. The generated code handles the details of converting data structures to and from XML, giving the service and client direct Java access to the data encapsulated in XML. Axis2 generates code for both services and clients. The client code is in the form of a stub class, which extends the Axis2 org.apache.axis2.client.Stub class. The service code takes the form of a service-specific skeleton, along with a message receiver class that implements the org.apache.axis2.engine.MessageReceiver interface. WSD2Java can generate both stubs and skeleton as required.

Creating the Service

First we will complete the service. Call WSD2Java as follows:

```
K:\CSCU9YW\axis2-1.6.4\bin\WSDL2Java.bat -s -ss -sd -uri Approver.wsdl
```

This will create the service skeleton (-ss) for synchronous communication (-s), that is the client will stop and wait for the service to respond after a request. The command will also create the service descriptors (-sd). You may find it useful to put this command into a .bat file, so you can execute it repeatedly when you edit your WSDL. A good place for this .bat file would be in the root of your eclipse project so the generated files are automatically in your project folder. Verify that there are generated java files in the src folder in the subdirectory Approver (a package automatically created).

The server-side code centers around a message receiver. This receiver implements a flavour of the org.apache.axis2.engine.MessageReceiver (in our case InOut). The Message Receiver in turns calls the business logic. A "hook" for the business logic is provided in the ApproverServiceSkeleton class which has a single approveOperation (see WSDL definition) with a single parameter which encapsulates the request message parameters.

The downside of adding code directly to this class is that if the WSDL changes, you need to regenerate the class and merge your changes. You can avoid this by using a separate implementation class that extends the generated skeleton, allowing you to override the skeleton methods without altering the generated code. Create such a class in eclipse (File – New – Class). Name the class ApproverService. Extend the ApproverServiceSkeleton class and overwrite the method

```
public approver.ApproveOperationResponse approveOperation(approver.ApproveOperation approveOperation) {}
```

Note the types of the parameter and return value. You can now add the business logic into this method. You can use the approveOperation parameter to extract the three values for address, name and amount using predefined getXYZ() methods, such as `String name = approveOperation.getName();`

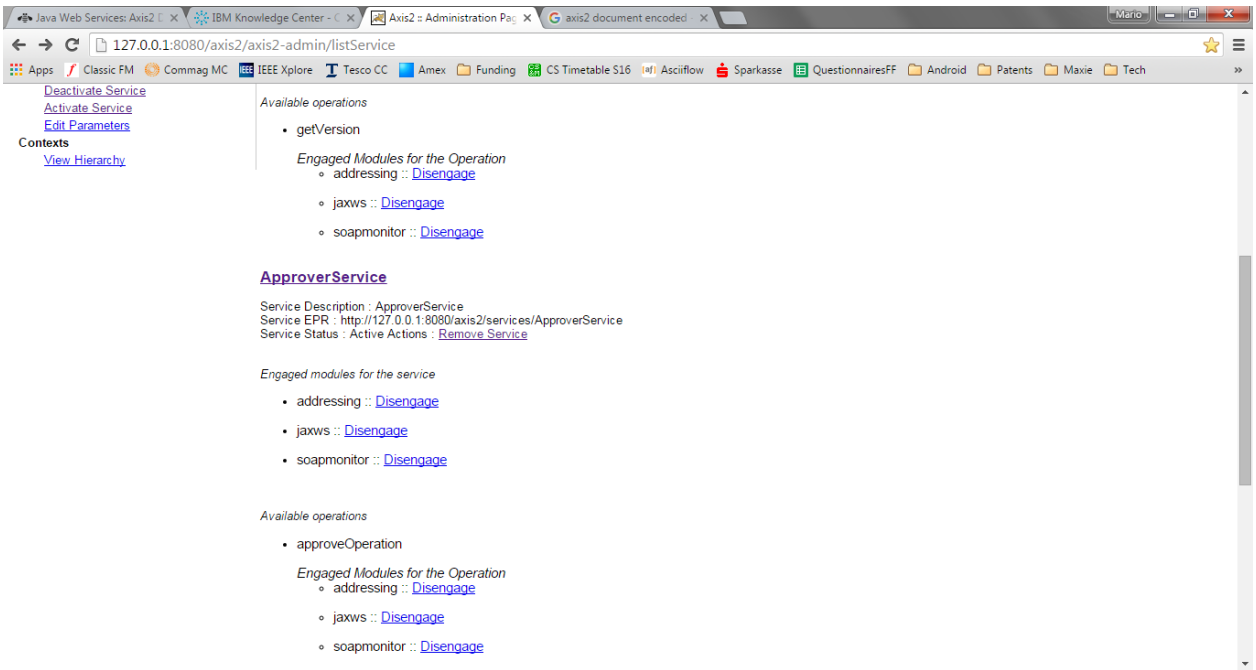
Next define a float value (assuming you used float as the return data type in the WSDL) and assign it some value. You can then write some logic to alter this value based on the request parameters. For instance: if (name.startsWith("Mario")) rate = 3.7f; You should define else branches checking the values for address and amount in a similar fashion.

Finally we will need to prepare the return value which needs to have the type approver.ApproveOperationResponse. Create an object of this type and use the predefined setXYZ() method to insert the float value into this object and then use a return statement with it.

```
approver.ApproveOperationResponse resp = new approver.ApproveOperationResponse();
resp.setRate(rate);
return resp;
```

Finally, we will need to prepare the service descriptor. This has already been generated by WSDL2Java, but in a directory resources, rather than META-INF. Create a META-INF directory under your eclipse project and copy the files from resources in it (it includes a nicely formatted WSDL file also). You will need to make a minor change to the descriptor file, changing the name of the ServiceClass to approver.ApproverService (rather than the Skeleton class which we have overwritten).

You can now create the approverService.aar file as you did in Lab 1 (export your project to a jar file). Make sure the META-INF directory and the generated class files are included. Deploy the service on Axis2. Afterwards you should see it as an available service on the Axis2 admin console. If you need to redeploy the service because something was not right or you made some other changes, sometimes Axis2 does not redeploy a service correctly. In that case close down Tomcat and restart. You should see it in Axis2 similar to:



Creating the Client

Next we need a client to try our service. Create a new eclipse project and link it to the Axis2 user library as you did with the service. Copy the WSDL file into this project. Next we will need to create the client stub code. Use WSDL2Java for this.

K:\CSCU9YW\axis2-1.6.4\bin\WSDL2Java.bat -uri ApproverService.wsdl -d adb -s

This creates the client stub class using synchronous communication and ADB binding. You then need to define your client code. This is similar to what you have already done in Lab 1. You will need to create a stub object and use this to create an object of the operation:

```
ApproverServiceStub stub = new ApproverServiceStub();
ApproverServiceStub.ApproveOperation request = new ApproverServiceStub.ApproveOperation();
```

After you can use predefined setXYZ() methods to set the parameters on the operation:

```
request.setName("Mario");
```

Then invoke the service expecting a response object back:

```
ApproveOperationResponse resp = stub.approveOperation(request);
```

Finally you can output the returned rate extracting it from the response object using the getXYZ() method.

```
System.out.println("Response : " + resp.getRate());
```

Compile and run the client. Verify that you get different responses from the service for different parameter values as defined in your service logic.

Checkpoint