Web Services-I Assignment

Bottom Up Approach of Web services Development.

The bottom up approach (Code First) is used where the developer starts with the business logic, which is the code, and then develops and deploys the code as a Web service.

1. Simple Class Deployment (POJO Deployment):

I. Create a simple java class called MyFirstService.java (have no package), the class should include two simple method:
   - o String sayHello (String name) : returns the string Hello <given name>
   - o int add (int a, int b): returns a + b

II. Deploy the new class as a web service: very simple, just create a new folder called **pojo** under the <AXIS_HOME>/repository folder and copy the .class file into it. Axis has the full responsibility to deploy the given class as a full service.

III. Ensure that the server configuration supports POJO deployment. Check the file <AXIS_HOME>/conf/axis2.xml

IV. Test the deployed service:
   a. Run the server: <AXIS_HOME>/bin/axis2server.bat
   b. Type the following URL in the browser

      http://localhost:8080/service

      You should see a list with the available services.

   c. Test the first method:

   http://localhost:8080/axis2/services/MyFirstService/sayHello?name=Axis2

   d. Test the second method:

   http://localhost:8080/axis2/services/MyFirstService/add?a=10&b=20

2. JAR file deployment (POJO Deployment):

Suppose that we have a service with two classes

I. Class Address (package book.sample) : a simple java bean with two fields, street and number
II. Class AddressService:

```
package book.sample
import javax.jws.WebService;
@WebService      //JSR181 Annotation
public class AddressService{
        public Address getAddress(){
                Address address=new Address();
                address.setStreet("my Street");
                address.setNumber("15");
                return address;
        }
}
```

III. Create a **jar file** containing the two classes.
IV. Deploy the jar file as a web service: copy the jar file into the previously created **pojo** folder.
V. Adjust the server configuration to allow jar deployment: add the following line into the <AXIS_HOME>/conf/axis2.xml file and restart the server.
    a. <deployer extention=".jar" directory="pojo" class="org.apache.axis2.deployment.POJODeployer"/>
VI. Test the deployed service:
    a. Run the server: <AXIS_HOME>/bin/axis2server.bat
    b. Type the following URL in the browser

    http://localhost:8080/axis2/services/listServices

    You should see a new service listed. Just call the getAddress method as previously described.

3. Deploying a service using a service archive file

The POJO approach cannot be considered the most flexible approach for web services deployment. Even Axis2 does not recommend the POJO deployment approach. The recommended one is the ***service archive-based*** approach. Suppose we would like to create a service archive file from the **HelloWorld** java class.

I. Writing the services.xml file: it represents the deployment descriptor for the service. It tells the deployment module how to deploy and configure the service. There are two things you have to keep in mind while creating such file:

    a. The fully qualified class name of the service implementation class
    b. The message receiver/s that we are going to use.

    Axis2 has a set of built in message receivers:

    - Handle only XML-In and XML-Out: **RawXML** message receivers.
    - Handle any kind of java beans, simple java types and xml: **RPC** message receivers.

```
<service name="HelloService">

        <messageReceivers>

                <messageReceiver  mep="http://www.w3.org/2004/08/wsdl/in-only"

                class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>

                <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
                class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>

        <messageReceivers>

        <parameter name="ServiceClass">HelloWorld</parameter>

        <operation name="sayHello" mep="http://www.w3.org/2004/08/wsdl/in-out"/>

        <operation name="add" mep="http://www.w3.org/2004/08/wsdl/in-out"/>

</service>
```

II. Creating a service archive file:

```
HelloWorld.aar
    META-INF
            Services.xml
    HelloWorld.class
```

III. Deploy the service: copy drop the archive file into the services directory in our Axis2 server repository.

4. Writing a client to access a web service

I. Creating a ***ServiceClient*** and setting its options.

```
ServiceClient sc= new ServiceClient();
//create option object
Options opts=new Options();
opts.setTo( new EndpointReference("http://localhost:8080/axis2/services/MyService"));
//setting action
opts.setAction("urn:sayHello");
//setiing created option into service client
sc.setOptions(opts);
```

II. Creating an OMElement (AXIOM) for the payload (content for the soap message). After looking at the code, look at the WSDL and see how this payload is created form it.

```
Public OMElement createSayHelloPayload(){
   OMFactory fac=OMAbstractFactory.getOMFactory();
   OMNamespace omNs =fac.createOMNamespace("http://ws.apache.org/axis2", "ns1");
   OMElement method=fac.createOMElement("sayHello", omNs);
   OMElement value=fac.createOMElement("name", omNs);
   value.setText("somebody");
   method.addChild(value);
   return method;
}
```

III. Invoking the service:

a. Blocking (Synchronous) manner:

sendReceive is the API for invoking a service in a blocking manner. When we use this API, the program gets blocked until it gets the response.

```
OMElement res= sc.sendReceive(createSayHelloPayload());
System.out.println(res);
```

b. Non-Blocking (Asynchronous) manner:

Axis2 uses the callback mechanism to provide asynchronous support. Therefore, we need to implement “AxisCallback” and pass that object as the method parameter.

```
OMElement res= sc.sendReceive(createSayHelloPayload());
System.out.println(res);
        //create a callback object
        //AxisCallback can be used started from version 1.3
        AxisCallback callback=new AxisCallback(){
                public void onComplete() {
                        complete=true;
                }
                public void onFault(MessageContext msgContext) {
                        System.err.print(msgContext.getEnvelope().toString());
                }
                public void onMessage(MessageContext msgContext) {

                        System.out.println(
                        msgContext.getEnvelope().getBody().getFirstElement());
                        complete=true;
                }
                public void onError(Exception e) {
                        e.printStackTrace();
                }
        };
        try {
                sc.sendReceiveNonBlocking(createPayload(),callback);
                System.out.println("-----------invoke the service---------");
                int index=0;
                while(!complete){
                        Thread.sleep(1000);
                        index++;
                        if (index>100){
                                throw new AxisFault("Time out");
                        }
                }
                } catch (AxisFault e) {
                        e.printStackTrace();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
```

Exercise: Adjust the client to call the add method rather than sayHello method. Provide both the blocking and the non-blocking manner.