Software Engineering - Practical Part

Matteo Mariani

Contents

01. JavaRMI

MISSING

02. Maven, web service interfaces - practical part of theoretical lesson

The middleware is provided in Java by a framework. In our case, we'll use: *Apache CXF*.
We'll use *Maven*, that is an automation tool. In particular, it is a tool in which we say - in the IDE - the step that it has to do in order to "build" the program. These steps are written in XML files, called `pom.xml`.

We're creating a web service that exposes three methods: `hello`, `getStudents` and `helloStudent`. `helloStudent` passes a Student to the web service and this one will store the student in the web service. `getStudents` retrieves the list of students stored in the web service. `hello` does nothing.

Firstly, I need to define the interface of the web service (`WSInterface.java`). We'll use the `javax.jws.*` and `javax.xml.*` imports. The first one is used with annotations (`@something`) in order to let understand the semantic that we're going to write to the underlying framework. For example, a class can be annotated like `@WebService` in order to inject the our need - "create and treat that class as a web service". In the interface, we put the signature of the three methods that we want to implement.
`@XmlJavaTypeAdapter` is an annotation put above a signature used to let the framework know how to build the XML for that method.

(...)

03. Lab 01 - SOAP

In this lab, there is a web service visible in the local network of the laboratory. The web service *BankInterface* exposes a two services - at 192.168.49.81:8080 :

- `java.lang.String[] getOperationsByClientID (int ClientID)` - given a `ClientID` (integer), which is the code for a client, returns the IDs of all bank operations performed in the last days by that client;
- `java.lang.String getOperationDetailsByID(int OpID)` - given an `OperationID` (integer) returns the full details of the given bank operation, as a unique String in the format `"[ID,ID of the performing client,date,amount, description]"`.

Goal:

1. implement a client that can query the *BankInterface* Web service;
2. implement a Web service AAAWS which offers the following operations:
   - `java.lang.String[] getClients()` - returns all the IDs and names of clients stored in the security sub-system. The result is returned as an array of strings, each string being the append of a client id, a comma and its name; as an example, a possible string might be `"1,Massimo Mecella"`.
3. implement the following logic in the client: "we want all the names of all clients who have performed an operation in the last days with description `"Benzina autostrada"` ".[1]
4. lastly, once downloaded also the *BankInterface* Web service, it is required that we refactor the two Web services in order to offer data structures of objects as returning values, and not anymore String (e.g., a map of clients, a map of operations, etc...).

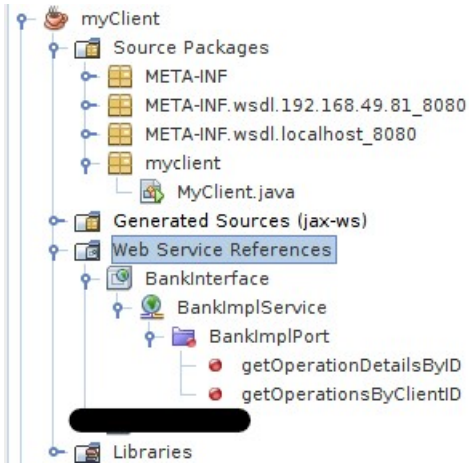
Step 1 - implement the client

First of all, we have to implement a Java application that works as a client. This will have the ability to query the Web services - the one exposed by Mecella (*BankInterface)* and the one exposed by us after the step 2).

Open NetBeans → New Project → Java → Java Application, and I created a project called *myClient*.[2]

Now, we want to use the services exposed by *BankInterface* Web service. In order to do that we need the IP address and the port in which that Web service is listening for some commands. In the laboratory, Mecella gives us the following address: `192.168.49.81:8080/BankInterface?wsdl`.
Note that the `?wsdl` part has to be always appended in order to correctly use the framework.[3]
To connect the client to the Web service: Right click on the package containing the client → New → Web Service Client → WSDL URL → then paste there the location of the Web Service (`192.168.49.81:8080/BankInterface?wsdl`).
This operation will retrieve the methods/services exposed by that Web service - *however* the server at that address should be running, otherwise can't do anything.[4] These will be listed directly in our Java project:

[1] In order to do so, the client has to query both web services.
[2] On the contrary, if we want to create a Server the procedure is slightly different, we'll see later on.
[3] `wsdl` stands for Web Services Description Language.
[4] Mecella gives us the code of BankInterface, so if we generate a new server project we can query that as localhost:8080/BankInterface?wsdl.



In particular under Web Service References → BankInterface → BankimplService → BankImplPort, we can see the two methods that the Web service exposes.

In order to enable to client to use them, we have to drag-n-drop both of them in our `MyClient.java` class. This will automatically generate two methods that can be used in the main method of our class. They wrap the call needed to use the services of BankInterface. After the drag-n-drop, we have this two new methods in `MyClient.java`

```
private static String getOperationDetailsByID(int arg0) {
    it.sapienza.softeng.bankws.BankImplService service = new it.sapienza.softeng.bankws.BankImplService();
    it.sapienza.softeng.bankws.BankIFace port = service.getBankImplPort();
    return port.getOperationDetailsByID(arg0);
}

private static java.util.List<java.lang.String> getOperationsByClientID(int arg0) {
    it.sapienza.softeng.bankws.BankImplService service = new it.sapienza.softeng.bankws.BankImplService();
    it.sapienza.softeng.bankws.BankIFace port = service.getBankImplPort();
    return port.getOperationsByClientID(arg0);
}
```

> Note that: these two new methods will return a List of Strings - not an array of Strings, as can be seen in the specification. I noticed that, the drag-n-drop operation does automatically this change of the return type. Indeed, when I created my Web service with the method `getClients` that return an array of Strings; then after the drag-n-drop it became a List of Strings. We'll see that I also changed the implementation of my Web service in order to be compliant - in particular, the method `getClients` doesn't return anymore a `java.lang.String[]` but a `java.util.List<java.lang.String>`. Ask to Mecella why this happens and what we have to do in this case.

Now the client can use the Web service functionalities - i.e. I written in the main a simple call to `getOpearationDetailsByID(5)` and it returned the result. In order to run the client, just click on Run Project on `myClient.java`.

Now the client can correctly use BankInterface Web service.

Step 2 - implement AAAWS Web service

Let's create our Web service, AAAWS.

Open NetBeans → New Project → Maven → Java Application, and I created a project called *myServer*.

First of all, we have to manipulate the `pom.xml` file.
At the beginning, we have to add the following dependencies:

```
(...)
<dependencies>
        <dependency>
                <groupId>org.apache.cxf</groupId>
                <artifactId>cxf-rt-frontend-jaxws</artifactId>
                <version>3.1.6</version>
        </dependency>
        <dependency>
                <groupId>org.apache.cxf</groupId>
                <artifactId>cxf-rt-transports-http-jetty</artifactId>
                <version>3.1.6</version>
        </dependency>
</dependencies>
(...)
```

Then click on Build Project, not Run. This will download some dependencies needed to run the server, in particular: the first one is used in order to have access to Apache CXF.

The Web service will be composed by three Java files:

- `AAAWSIFace.java`, that is the interface that groups all the services that the Web service will expose;
- `AAAWSImpl.java`, that is the class that implements the interface `AAAWSIFace.java`;
- `MyServer.java`, that is the class the represents the "front-end" of the Web service.

Web service interface

Let's consider `AAAWSIFace.java` interface.
This is a very simple interface with the signature of the method `getClients()`. The only thing to take care of is the annotation `@WebService`, that is used for "creating Web Services with JAX-WS". Note that some imports are needed in order to use that annotation.

```
@WebService
public interface AAAWSIFace {

    public java.util.List<java.lang.String> getClients();

}
```

Web service implementation

Let's consider `AAAWSImpl.java` class.
This is the class that implements the method seen before. The only important thing to notice is the `@WebService(endpointInterface = "com.mycompany.myserver.AAAWSIFace")` annotation. This is used in order to connect the Web service interface.

```
@WebService(endpointInterface = "com.mycompany.mynewserver.AAAWSIFace")
public class AAAWSImpl implements AAAWSIFace {
```

```
    @Override
    public java.util.List<java.lang.String> getClients(){
        List<String> l = new ArrayList();
        l.add("1,Massimo␣Mecella");
        l.add("2,Maurizio␣Lenzerini");
        l.add("3,Giuseppe␣De␣Giacomo");

        return l;
    }
}
```

The method is very stupid and compliant to requirements.

Web service "front-end"

Lastly, let's consider the `MyServer.java` class.
This class is used to run our Web service and put it in listen; waiting for some clients that ask its services.

```
public class myServer {
     public static void main(String[] args) throws InterruptedException {
        AAAWSImpl implementor = new AAAWSImpl();
        String address = "http://localhost:8080/AAAWS";
        Endpoint.publish(address, implementor);
        //System.exit(0);
        while(true){
            Thread.sleep(60 * 1000);
        }
    }
```

Note that `http://localhost:8080/AAAWS` is the location of the Web service and it will be managed *exactly* in the same manner we had seen for `192.168.49.81:8080/BankInterface`, concatenating it with `?wsdl` keyword.

Deployment of my Web service

Now, we have to deploy our Web service.
First of all, we have to insert in the `pom.xml` a plugin:

```
(...)
<build>
        <plugins>
                <plugin>
                        <groupId>org.codehaus.mojo</groupId>
                        <artifactId>exec-maven-plugin</artifactId>
                        <configuration>
                                <mainClass>com.mycompany.mynewserver.myServer</mainClass>
                        </configuration>
                </plugin>
        </plugins>
</build>
(...)
```

where, in `mainClass` we have to insert the path to `myServer`.
Build Project again.

To finalize the deployment, Run Project on `myServer.java`. If everything goes OK, the server should run indefinitely:

```
Output - Run (myNewServer)

------------------------------------------------------------------------
Building myNewServer 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ myNewServer ---
Mar 23, 2019 11:02:51 AM org.apache.cxf.wsdl.service.factory.ReflectionServiceFactoryBean buildServiceFromClass
INFO: Creating Service {http://mynewserver.mycompany.com/}AAAWSImplService from class com.mycompany.mynewserver.AAAWSIFace
Mar 23, 2019 11:02:53 AM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be http://localhost:8080/AAAWS
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

Let the client access the services exposed by AAAWS

As we did for BankInterface Web service, the services exposed by AAWS has to be "encapsulated/imported" by the client.

To connect the client to the AAAWS Web service: Right click on the package containing the client → New → Web Service Client → WSDL URL → then paste there the location of the Web Service (`http://localhost:8080/AAAWS?wsdl`).
This will generate a new Web Server References; so we have to drag-n-drop the new method as we did previously. So, in `MyClient.java` we'll the additional method:

```java
private static java.util.List<java.lang.String> getClients() {
    com.mycompany.mynewserver.AAAWSImplService service = new com.mycompany.mynewserver.AAAWSImplService();
    com.mycompany.mynewserver.AAAWSIFace port = service.getAAAWSImplPort();
    return port.getClients();
}
```

> If we have multiple servers in our machine that have to be deployed contemporary, then they cannot be configured to start at the same port. Change it in something ≥ 1024.

Now the client can talk to both Web services.

Step 3 - implement the client logic

Now, we have to implement the logic in `myClient.java` client.
Remember that we want to "to write a client program which outputs all the names of all clients who have performed an operation in the last days with description `"Benzina autostrada"` ".

The code is long and trivial, check directly `myClient.java` class.

Step 4 - refactor of AAAWS BankInterface Web services

For this last step, Mecella gives us the code of the Web service that was online in the laboratory.
First of all, we have to:

1. create a new Maven project in which store the three classes belonging to BankInterface WS;
2. modify the related `pom.xml` - in the same way we saw before;
3. modify the location of the Web service inside its `Server.java` class, i.e. `http://localhost:1024/BankInterface`;
4. modify the location of the Web service inside the configuration of our client (go to Web Service References → BankInterface → right click → Edit Web Service Attributes → Wsimport Options → change the port at wsdlLocation, i.e. `http://localhost:1024/BankInterface?wsdl`;

5. Build Project and then we can Run the server.

Since in Step 3) the logic was implemented relying heavily con String manipulation, this step has the goal to alleviate that pain. :)
In particular, we have to "refactor the two Web services in order to offer data structures of objects as returning values, and not anymore String (e.g., a map of clients, a map of operations, etc...)"
This can be done by using Adapters.

TO DO

03. Lab 02 - REST

(...)
Check the code inside the VM in order to see some notes directly in the code.

04. Lab 03 - JMS

The lab is divided into two exercises. At first we need to implement a client which is able to subscribe to the topic exposed by Mecella's machine.
To do this lab I used both the code on the PDF `Lab3.pdf` and the tutorial at [**?**].
At first, we need to create a Maven project. Then, in the `pom.xml` file we need to add the following dependency:

```
<dependencies>
    <!-- ActiveMQ -->
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-all</artifactId>
        <version>5.14.5</version>
    </dependency>
</dependencies>
```

Basically, ActiveMQ will be our JMS provider and makes possible to use the `javax.*` library.
The code implementing such exercise is the class `Client.java`. For simplicity, I decide to implement the first exercise with one class and the other with another class. Clearly, it is possible to do them using a single class.

For the complex exercise, everything is implemented by `ClientOrdini.java`. Here the only difference is that there is a producer which sends a message to another topic that is, Ordini. This message is sent only when receiving a particular quotation (in my case it is Vodafone) and contains something like my name, the price at which buying quotations and how many quotations to buy.
At this point, the consumer which has been created on the topic Ordini, is listening until a reply message is sent from Mecella. For this reason, I created another listener overriding the `onMessage()` method.