States

Goals

The objective of this lab is to discover and apply best practices for using XML/SOAP/WSDL WebServices.

First, we are going to implement two "helloworld" type webservices projects: a client and a server. In a second step, we will implement a small project in order to better master the notion of XML web service on the server side and on the client side.

Preliminary work

I think Tomcat and CXF are already installed in your computer if not tell me please

It is advisable to go through the slides attached to TP n°2.

To perform this lab, we need, in addition to Eclipse and Maven, the following tools:

- Tomcat
- The CXF library

A tutorial is available here: Installation and configuration of Tomcat and CXF in Eclipse

Hello World exercise

this is just an exampleI doit I will send you the zip contains the java classes

Step n°1: Creation of the WebService Server

Go to the directory of your choice then create a project using the command: mvn archetype:generate

The type of project to generate uses the following archetype: cxf-jaxws-javafirst
-> org.apache.cxf.archetype:cxf-jaxws-javafirst (Creates a project for developing a Web service starting from Java code))

When you have selected the right type of project, type the number corresponding to the version of CXF that you have installed (or that is already installed on your computer).

Finally, here are the different recommended options (to stay consistent with the examples of the lab) for the generation:

groupId: alom
artificatId: cxf-helloworld-server
version: Leave as is (1.0-SNAPSHOT)
package: alom.server

When your project is generated, import it into Eclipse.

You can notice that in the alom.server package, two files have been generated by Maven, they are a HelloWorld interface and a HelloWorldImpl class implementing the interface.

These two files are provided as examples by Maven. For this part of the exercise, we are going to use them, but in the majority of the builds made for this type of project, these two files have to be deleted/modified.

We can see that in the HelloWorld interface we have:

```
@WebService
public interface HelloWorld {
    String sayHi(String text);
}
```

The @WebService annotation indicates to the CXF library that our interface is indeed a WebService interface (entry point of our application). All the methods of this interface will therefore be accessible by the consumers of the service. Here, there is only the sayHi method.

On the implementation side, we have:

```
@WebService(endpointInterface = "alom.server.HelloWorld")
public class HelloWorldImpl implements HelloWorld {

    public String sayHi(String text) {
        return "Hello " + text;
    }
}
```

The @WebService annotation here is optional, it simply indicates that the interface corresponding to this implementation is alom.server.HelloWorld. This can be useful in order to quickly find which method of this class will be "exposed" if the class implements several interfaces (rare case).

Now, we are going to launch this WebService in an application server: Tomcat. To do this, in the Servers tab (take a look at the Tomcat installation tutorial if you don't have a Servers tab), right click on your Tomcat instance then Add and Remove.... Switch your project from the Available column to Configured. Now your project is one of the projects that can be launched in the Tomcat instance.

Start your Tomcat. If you encounter an error related to a "NoClassDefFoundError: org.springframework.context.ApplicationContextException", add the following dependency in your pom.xml:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context< /artifactId>
    <version>5.1.16.RELEASE</version>
</dependency>
```

When Tomcat starts correctly, go to http://localhost:8080/cxf-helloworld-server/ . A page appears showing you all the accessible WebServices. You can also click on the link that appears, it will give you access to the WSDL corresponding to your service.

Step n°2: Creation of the Client WebService same things just an exampleI doit I will send you the zip contains the java classes

Recreate a project with maven, as for the server, but with the following options:

groupId: alom
artificatId: cxf-helloworld-client
version: Leave as is (1.0-SNAPSHOT)
package: alom.client

When your project is generated, import it into Eclipse.

Start by removing the maven generated classes in the alom.client package from src/test/java and the classes in the alom.client package from src/main/java.

We are now going to use the wsdl2java command to generate the source code for the client part of the WebService. To do so, you must first configure the JAVA_HOME of your console (unless it's already done...). If you don't know how to do it, search on the Internet (set command and export if necessary)

Then go to the bin directory of the apache-cxf-3.xy directory you previously unpacked (with x and y corresponding to the version number you downloaded). Using the wsdl2java command and the options below, generate the WebService client source code from the WSDL (provided by the server part, make sure the server from step 1 is started):

- -p indicates the package in which the generated sources should be created: alom.client
- -client indicates that you want it to generate the client part of the webservice
- -d indicates the output directory of the generated source files: fill in the full path to the src/main/java directory of your cxf-helloworld-client project (it will place the sources automatically in the directories linked to the package indicated above, this is to say alom.client)

then as an argument (it is therefore not an option), indicate the URL where the WSDL is located (hint: if you don't know where it is, read step 1 again)

Once the classes are generated, refresh the project under Eclipse. You can browse the generated classes to discover their content.

One of the generated classes is an example class used to call the WebService on the server side, based on the other generated classes. Modify the example class (it contains a main method) so that it calls the sayHi method by passing your first and last name as parameters, then execute this main method as a java application.

Exercise on a football web service Don't do it it was juste un exercice

Step 1: Configuring wsdl2java to include a proxy

In the bin directory of cxf, copy make a copy of the file wsdl2java (or wsdl2java.bat under windows) and name it wsdl2javaproxy (or wsdl2javaproxy.bat under windows).
Edit and replace in the file:

- On Linux: $JAVA_HOME/bin/java -Xmx128M
  by: $JAVA_HOME/bin/java -Dhttps.proxyHost=cache-etu.univ-lille1.fr -Dhttps.proxyPort=3128 -Xmx128M
- On windows: "%JAVA_HOME%\bin\java" -Xmx128M
  by: "%JAVA_HOME%\bin\java"-Dhttps.proxyHost=cache-etu.univ-lille1.fr -Dhttps.proxyPort=3128 -Xmx128M

Then, using this "proxyed" script, generate a java client with in your cxf-helloworld-client project, in the alom.foot package with the following WSDL: https://footballpool.dataaccess.eu/info. wso?WSDL

Step 2: Using the webservice

Inspired by the "example" client class, find and display the following information, in an Eclipse console or a terminal console, by querying the WebService (which uses data from the 2018 World Cup):

- The list of all participating players
- Date of all matches
- The name of all the coaches
- All the cities that hosted the competition
- The list of all the matches, with per match (several calls necessary to have the entire list):
  - Description of the match (final, semi-final, etc.)
  - The date of the game
  - The final score
  - The number of yellow cards
  - The number of red cards
  - The teams that faced each other

NB: Do not forget to configure the proxy if necessary, with the following code:

```
System.getProperties().put("https.proxyHost","cache-etu.univ-lille1.fr");
System.getProperties().put("https.proxyPort","3128");
```

Project to be returned Here begins the real project

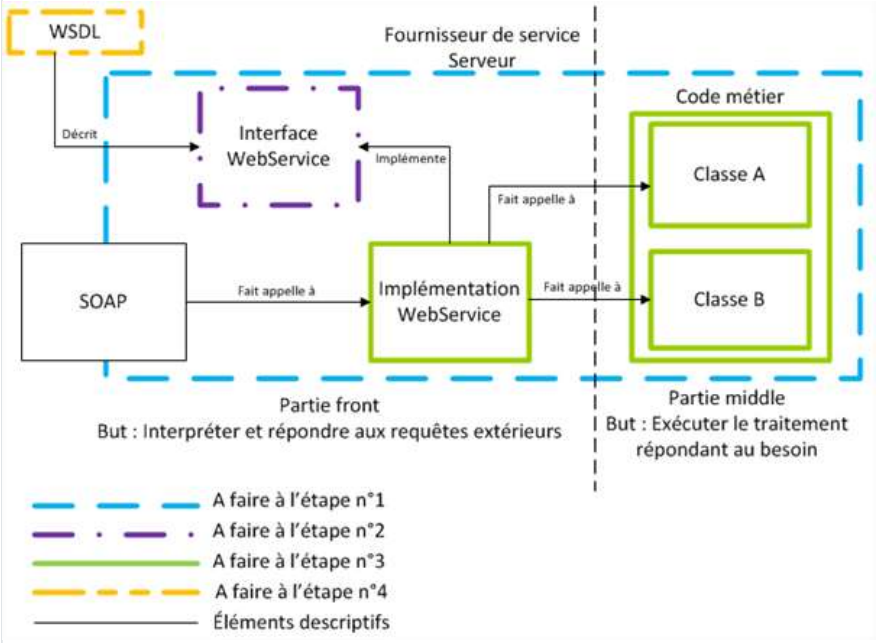
Now that we have seen how to generate a helloworld webservice (client and server) and a foot webservice (client), we will now see how to set up a webservice on the server side, then we will create a small client application using it .

Creation of a banking webservice (server)

Step n°0: Context of the subject

We are going to create a server webservice in the context of the bank. This webservice will notably make it possible to create customers, bank accounts on the server, to add and withdraw money, etc...

Below is a diagram describing the architecture of what will be achieved, with its legend:



The paragraph below is not specific to the TP, it helps to better understand the architecture above and it is a very common general convention for creating an SOA architecture.

As shown in the diagram, the front and middle parts are distinguished for two specific purposes. We set it up because the majority of WebService applications use this architecture. In the lab, these parts are simply used to properly structure the project, but on a daily basis in real applications, here are their functions:

- The front contains all the server's interfaces, which allow it to be called in different ways: Web for computers, mobile Web, WebService for information systems (other servers) or WebService for client applications (on computers or on mobile). It relies on the middle to execute the processes, and responds specifically to the caller depending on the interface used
- The middle centralizes the business code, so it can be called by different front interface to execute the same process, modulo some adaptations made in the front part

Step n°1: Creation of the CXF project via maven

Start by generating a new CXF web service project via maven (reread the Helloworld part of this lab if you don't remember how to do it), with the following configuration:

groupId: `alom`
artificatId: `cxf-bank-server`
version: Leave as is (`1.0-SNAPSHOT`)
package: `alom.bank.server`

Import the project into Eclipse and don't forget to remove the Helloworld classes that were generated by Maven (as mentioned in the previous lab, these classes are only there to discover CXF, so they are useless for other projects).

Step 2: Creating the interface

The purpose of an interface in a webservice is to create methods that can be called by clients consuming this webservice.

As a reminder, a WebService interface must contain the `@WebService` annotation attached to the class. Optionally (but strongly recommended) it can also contain `@WebParam` annotations attached to method parameters, with the name of the parameter as `attribute` . This mechanism allows the client who will generate code from your WSDL, to have parameter names as you have indicated them in these annotations, it is much more meaningful than `arg0` , `arg1` , `arg2` , etc., which are the names given when no parameter name is specified.

Example :

```
@WebService
public interface HelloWorld {
    String sayHi( @WebParam (name="text")String text);
}
```

To see this for yourself, go back to the generated helloworld client, you will find that the parameter available for the client-side `sayHi` method is indeed `arg0` . By adding the `@WebParam` annotation as shown in the example and restarting your server, you get a new WSDL. Using this new WSDL, you can regenerate a client that will contain the `sayHi` method with a parameter named `text` , which gives much better information than `arg0` .

Before embarking on the creation of this interface, take the time to fully read what it should contain (all of step 2 therefore), in order to allow you to design your interface as well as possible.

Create an interface with the following methods:

- a method to create a client on the bank server:
  - Taking as a parameter:
    - The first name of the account holder
    - The name of the account holder
    - The date of birth (of Calendar type) of the account holder
  - Returning the bank customer thus created
  - Throwing an exception when a customer already exists with the same information

- a method to recover a customer from the bank:
  - Taking as a parameter:
    - The first name of the account holder
    - The name of the account holder
    - The date of birth (of Calendar type) of the account holder
  - Returning the customer from the bank if it exists
  - Throwing an exception when no customer exists with this information

- a method to create a bank account for a bank customer:
  - Taking as a parameter:
    - The bank customer
    - Type of bank account:
      - CHECKS
      - A BOOKLET
      - SUSTAINABLE_DEVELOPMENT_BOOKLET
      - YOUTH_BOOKLET
  - Returning the bank account thus created
  - Throwing an exception when the specified bank bank customer is incorrect or does not exist, the specified account type does not exist, or an account of that type already exists for this customer

- a method to recover a bank account:
  - Taking as a parameter:
    - The bank customer
    - The type of bank account
  - Returning the bank account if it exists
  - Throwing an exception when the specified bank customer is incorrect or does not exist, when the specified account type does not exist, or when the account you are looking for does not exist

- a method for adding money to a bank account:
  - Taking as a parameter:
    - The bank account
    - The amount to be added to the bank account
  - Returning the new bank account balance
  - Raising an exception when the specified bank account does not exist, or the amount to add to the bank account is incorrect

- a method to know the balance of a bank account:
  - Taking as a parameter:
    - The bank account
  - Returning the bank account balance
  - Throwing an exception when the specified bank account does not exist

- a method for withdrawing money from a bank account:
  - Taking as a parameter:
    - The bank account
    - The amount to be withdrawn from the bank account
  - Returning the new bank account balance
  - Throwing an exception when the specified bank account does not exist, or the amount to be withdrawn from the bank account is incorrect or does not allow the withdrawal (no overdraft allowed)

- a method for making an internal transfer between two accounts of the same customer:
  - Taking as a parameter:
    - The customer concerned
    - The type of account to be debited
    - The type of account to be credited
    - The amount to transfer
  - Returning nothing
  - Throwing an exception if the specified customer does not exist, if one of the two specified account types is invalid, if one of the two specified account types does not exist for the specified customer, the amount to be transferred is greater than the balance of the account to be debited (no overdraft authorized) or that the amount to be transferred is incorrect

- a method for making a transfer between two accounts of two different customers:
  - Taking as a parameter:
    - The debited customer
    - Debited bank account
    - The credited customer
    - The credited bank account
    - The amount to transfer
  - Returning nothing
  - Throwing an exception if the specified debited or credited customer does not exist, the debited or credited bank account does not exist, the debited bank account does not belong to the debited customer, or the credited bank account does not does not belong to the credited customer (security rules to ensure that the information entered corresponds to the actual transfer request, allowing the control of input errors,

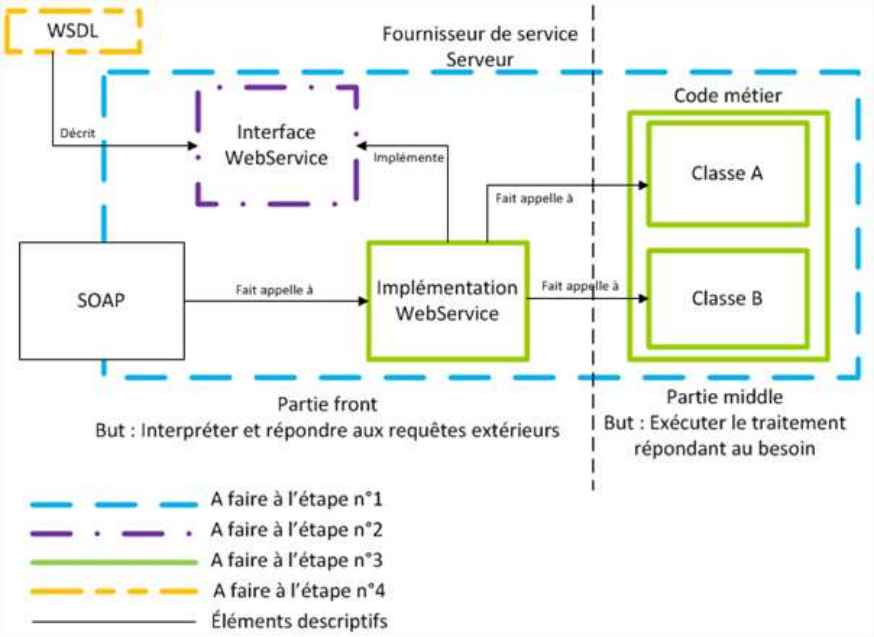
etc.), if the amount to be transferred is greater than the balance of the account to be debited (no overdraft authorized) or that the amount to be transferred is incorrect

- a method for settling (closing) a bank account:
  - Taking as a parameter:
    - The bank account
  - Returning the closed account balance
  - Throwing an exception if the bank account does not exist

- a method to remove the customer from the bank:
  - Taking as a parameter:
    - The bank customer
  - Returning nothing
  - Raising an exception if the customer does not exist or still has a bank account

Step 3: Creating the implementation

Now, create a class implementing the interface you have just set up, as well as the classes linked to the business code (in this case the banking rules set out in step 2), while respecting the following architecture (the webservice part only contains code calling the business code):



Step n°4: Deployment of the WebService and access to the WSDL

Open the beans.xml file located in the src/main/webapp/WEB-INF/ directory of your project. This file was generated by Maven, and it contains in particular a part specific to the project:

```
<jaxws:endpoint
    id = "helloWorld"
    implementor = "alom.bank.server.HelloWorldImpl"
    address = "/HelloWorld" />
```

These lines define an endpoint, which works on a principle similar to that of servlets. An endpoint therefore makes it possible to connect the methods of the WebService, to a given URL. In the example below, we have:

- id: Defines an endpoint name
- implementor: Defines a class implementing an interface annotated by `@WebService`
- address: Defines the url on which the WebService is connected (this is the end of the url, which can be preceded by a directory or other connections, such as the name of the webapp for example)

Replace the values of these lines (which concerned the helloWorld generated by Maven) with consistent values for your WebService, which will allow it to work properly.

When you're done, add your project to the Tomcat instance in your Eclipse (like in the Helloworld part of the lab), then start the server. If you used the configurations from step 1, and you correctly modified the endpoint configuration, your endpoint should be visible at http://localhost:8080/cxf-bank-server/ .

All you have to do is display the WSDL, which will allow you to move on to the next part, the webservice client.

Creation of a banking webservice (customer)

Step n°1: Creation of the project, import into Eclipse and generation of sources from the WSDL

As in the Helloworld part of the lab, create the `cxf-bank-client` webservice client project , import it into Eclipse and generate the source code from the WSDL you just accessed in the previous step.

Step n°2: Creation of a class allowing the instantiation and configuration of the WebService client

Create a BankServiceClient class.

In the constructor, we will instantiate a `ClientProxyFactoryBean` by passing an instance of `JaxWsClientFactoryBean` as a parameter . The `ClientProxyFactoryBean` is used to configure, then instantiate a WebService client of our choice, provided that we have available the generated classes corresponding to the WSDL of the server that we want to call.

Once this object is instantiated, we can call the following methods:

- `setServiceClass` : which takes the WebService interface as a parameter
- `setAddress` : which takes the server URL as a parameter
- `create` : instantiates an object implementing the interface passed to the `setServiceClass` method . All methods called on this instance will launch an invocation on the WebService Server

The call to the last mentioned method, create, should look like this:

```
this .service = (BankService) factory.create();
```

Then you can use CXF objects to configure HTTP connection elements, which are used as follows:

```
Client client = ClientProxy.getClient ( this.service ) ; HTTPConduit http = (HTTPConduit) client.getConduit();
HTTPClientPolicy httpClientPolicy = http.getClient ();
```

The `HTTPClientPolicy` object allows you to call many methods, including:

- setConnectionTimeout: sets the timeout after which the connection attempt will be considered failed
- setReceiveTimeout: sets the timeout after which the connection attempt will be considered failed
- setProxyServer: defines the proxy host through which you want the client to go to call the server (optional for the lab because you are in localhost, provided for information only)
- setProxyServerPort: defines the port of the proxy through which you want the client to go to call the server (optional for the TP because you are in localhost, provided for information only)

As well as other methods that can be used as needed.

Step 3: Using the client and testing

Create a class consisting of a main method, instantiating the class you created in step 2, then experiment by calling each of your methods. Take care to write down your test scenarios because they are required of you in the report to be submitted with this practical work. Do not forget to provide test cases for each of the exceptions that can be returned by the methods.

Modified on: Tuesday 28 September 2021, 00:31

◄ Render

Go to...

Installing and configuring Tomcat and CXF in Eclipse ►