Assignment 1.1 - Web Service (WDSL/SOAP) - Lab Report

Katerina Chinnappan, Rohit Shaw, Kjell Zijlemaker

April 11, 2019

Design & Approach

We have chosen Java to implement the web service & client for this part of the assignment. We have used the *Bottom-Up* approach(started off by writing the Java class/methods and then generated the WSDL file from it) to implement the WDSL/SOAP service. Essentially, SOAP means that the URL remains the same for all invocations. The only part that changes are the parameters for the Java methods. We set up the project in the NetBeans IDE because it provides a convenient integration with the GlassFish server. We chose "create web service" when creating the project in NetBeans and this created a nice project structure. Maven is used in NetBeans in order to package the entire project (web service) in a .WAR file. This file is then used to deploy the web service to a server (in our case we went with GlassFish, which is what NetBeans offered.)

Implementation

Web Service - `CalculatorService.java`

We started off by creating a Java class (`CalculatorService.java`) for the web service in the **src /main /java/ package** directory. Since we were implementing a web service, we used the dependency injection `@WebService` right before the class definition. We then proceeded to implement the methods add, sub, mul & div. Just like the `CalculatorService.java` class, each method used a `@WebMethod` dependency injection in order to explicitly state that this is indeed a web method inside a web service. Each method takes in two parameters of type `float` and returns the result from the mathematical operation performed on the two parameters.

After finishing implementing the web service, NetBeans takes care of packaging the project in a .WAR file (the type of file we wish to package our project too is specified in the `pom.xml` file in the root of the project). This .WAR file is then deployed to GlassFish server (through NetBeans) and the web service can be accessed through `http://localhost:8080/calculator_maven/CalculatorServiceService?Tester`

Client - `CalculatorClient.java`

Of course, `http://localhost:8080/calculator_maven/CalculatorServiceService?Tester` this url is a client itself however we wrote our own client to illustrate that we understand how to use our web service. The client folder and Java classes are generated from the WSDL file which was generated in the beginning using the Bottom-Up approach. The command `wsimport -keep -p client calculator.wsdl` generates all the needed Java classes in order to write the client. The generated file `CalculatorServiceService.java` initiates a connection to the web service so in our case we had to change the url to `http://localhost:8080/calculator_maven/CalculatorServiceService?WSD`. The generated file texttCalculatorService.java describes the web methods. We must create a new `service` object - `CalculatorServiceService service = new CalculatorServiceService();`. We then create the `calc` object - `CalculatorService calc = service.getCalculatorServicePort();` This specifies which port number to use for the protocol. We then can proceed to write our client by calling the web methods described in `CalculatorService.java` using the `calc` object. We decided to provide an option for the user to test the web service using the client in an interactive and non-interactice mode. The non interactive mode simply invokes the web methods with hard-coded numbers and returns the output. The interactive mode provides a menu with the list of mathematical operations and gives the option to the user to chose which operation to perform on the chosen numbers.

Question 3 - Proposed solution to a stateful calculator

Overview

A stateless web service does not keep track of information from one invocation to another one. In other words, it does not keep track of the state from the previous request. In general, a stateless web service is not a bad thing because many web services don't actually need to keep track of the previous state. However, some applications like Grid applications generally require statefullness so we will propose a solution on how to implement a stateful calculator service.

In order to make the calculator stateful, we need to add the state information as a variable in our web service. We are going to describe briefly the implementation and the process for the "add" method because this process & implementation logic will be exactly the same for other methods.

Process

Figure 1 illustrates the process of the "add" web method which keeps track of the previous state.

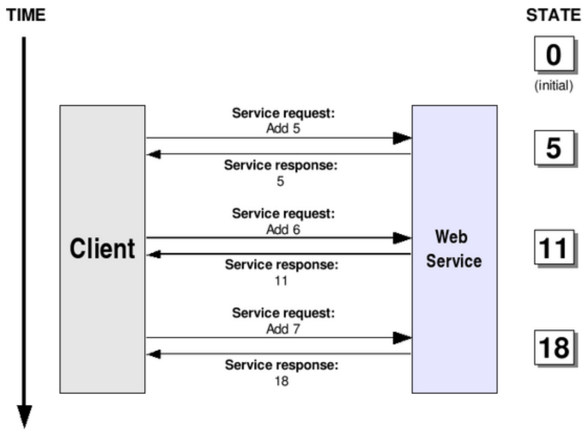

Figure 1: The communication between client and a stateful web service - taken from https://docs.huihoo.com/globus/gt4-tutorial/ch01s03.html

The client sends a service request to add the number 5. The web service responds by returning the number 5 back to the client. The client then wishes to add the number 6(since this is a stateful service, we keep track of the previous state (add number 5)). The service responds back to the client with the number 11 because it remembered the previous request and added 6 to 5. The next steps illustrated in Figure 1 are very intuitive.

Implementation

The simplest way to implement a stateful web service is to defind a global `state` variable in the Java class. Each web method in the calculator service would then take one parameter `x`. This parameter would then be added or subtracted or whatever the mathematical operation is from `state` which tracks the state of the previous request. Another additional method `result` would be added which would return the final state of the variable `state`. Here is a sample pseudo-code(python)of how this would look:

```
Class Calculator:
   # initialize the state
   state = 0
   def add(x):
      return state = state + x
   def result:
      return state
```