Advanced Java 2018
Assignment 3 (2 points)
Deadline Thursday, 23 August 23:59

Learning goals. This assignment is about communication using jetty in Java. You are expected to learn the following skills by completing this assignment:

- How to use Jetty library to embed an HTTP server inside an application.
- How to use Jetty library to construct HTTP clients and send HTTP messages to servers.
- How to send HTTP GET/POST requests and receive responses.
- How to serialize/de-serialize data-structures to text and back using XStream library.
- How to construct an application which can send and receive messages over the network.

Format. This assignment set consists of two parts:

1. A problem description on 4 numbered pages, including this page. Make sure you have them all, and read this page first.
2. A zip archive containing a stub implementation of the assignment.

The implementation portion of the assignment text has three parts. Each part contains a number of questions which you must solve by filling out the missing parts of the stub implementation. Your only deliverable is the solution code, no report is required. *Please make sure to comment your code appropriately.* This is a pass/fail assignment. You are allowed to work in groups (max group size is 3). *Remember that the instructors are there to help you during the session, ask questions/clarifications if you need it.*

Hand in. You must submit your solution via the Absalon course page under the appropriate assignment page. Your submission must adhere to the following requirements:

- It must consist of a single zip file containing your solution code. You can omit submitting the lib/ directory handed out in the zip archive of the exam.
- All files must be able to compile (i.e., be accepted by the Java 10 compiler without errors), and any code required for building must be included in the submission. You are only allowed to use the jars handed out in the exam zip archive.
- You are not allowed to change the handed out interfaces. You are free to add new classes and methods as you require without violating the interfaces.

The deadline at the top of this page is strict. You may submit multiple times, the *last* submission counts and the previous will be ignored.

Happy hacking!

Employee Database

An *Employee database* stores employee records distributed across independent and isolated servers. Clients can access the database using the client libraries to add/read/modify employee records. The employee data are partitioned by the department an employee works in, over various servers. All the employee records for a department will be in one server. A server can contain employee records for multiple departments. In order to simplify the design, the partitioning layout is statically configured in two configuration files "serverdepartmentservermapping.xml" and "clientdepartmentservermapping.xml". The first one is read by the server and the second one by the client when they startup. They contain identical configuration, but it may be changed to test real-world scenario when client configuration is not up-to-date with the server configuration. The clients communicate with the server using remote procedure calls (RPCs) available through client libraries which are implemented using HTTP for communication [1].

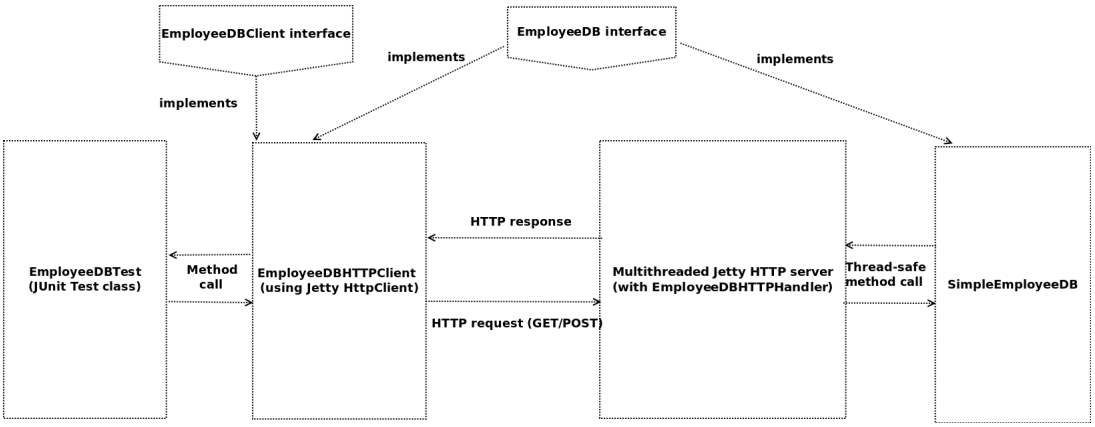

Figure 0.1: Architecture of Employee Database application

The methods (RPC) available in the employee database are defined in *assignment3.EmployeeDB* interface. The interface is implemented by the client library using *assignment3.EmployeeDBHTTPClient* class and the server using *assignment3.SimpleEmployeeDB* class. The *assignment3.SimpleEmployeeDB* class provides the implementation of the employee database and provides methods defined in the interface which are invoked by the handler class *assignment3.EmployeeDBHTTPHandler* which multiplexes HTTP requests received by the instance of Jetty HTTP server. To instantiate Jetty HTTP server you may use *assignment3.EmployeeDBHTTPServerFactory* class (see usage example in *assignment3.EmployeeDBHTTPSampleServer*). The *assignment3.EmployeeDBClient* class defines the interface that client libraries must implement to retrieve department to server mappings to simplify lookups. This interface is implemented by the *assignment3.EmployeeDBHTTPClient* class using the helper methods from *assignment3.Utility* class. *You do not have to implement EmployeeDBClient interface, we have already implemented the department to server lookup method defined in the interface for you.* This mapping is read from the "clientdepartmentservermapping.xml" file. Make sure you change this file to update the server and department mappings you want in your application. We have also provided a JUnit test case class *assignment3.EmployeeDBTest* which can be used to test both the client (EmployeeDBHTTPClient) and the server (SimpleEmployeeDB) classes which implement the *EmployeeDB* interface.

The architecture of the application is outlined in Figure 0.1.

Implement the server methods

Implement assignment3.EmployeeDB interface

- Implement the following stubbed out methods in `assignment3.SimpleEmployeeDB`[2]

[1]The Jetty library (http://www.eclipse.org/jetty/) is used to provide the HTTP server and the HTTP client libraries. We will be using Jetty 8 (http://download.eclipse.org/jetty/stable-8/dist/). The jars have been provided in the lib directory in the assignment handout. Make sure you add the lib directory in your your buildpath.

[2]The stubbed out methods have the synchronized keyword to make them thread-safe.

a. `addEmployee`: Adds an employee record in the employee database. The method throws an exception if an employee record is being added for a department that the server is not initialized to handle in the constructor. You need to propagate this exception to the client.

b. `listEmployeesInDept`: Returns the list of employee records who belong to the departments passed as argument to this method in the employee database. The method throws an exception if one or more of departments passed as argument are not in the list of departments that the server is initialized to handle in the constructor. You need to propagate this exception to the client.

c. `incrementSalaryOfDepartment`: Increments the salaries of the employees in the departments passed as argument by the amounts passed as argument. Ensure all/nothing semantics[3]. Possible causes of failure can be :

a) One or many departments passed as argument do not exist on the server.

b) The amount by which the salary is to be incremented is negative.

This method throws exceptions to signal which of the above errors happened. You need to propagate this exception to the client.

d. `cleanupDB`: Resets the state of the employee database to an empty state (newly created employee database).

- ***OPTIONAL*** Create JUnit test cases in the `assignment3.EmployeeDBTest` class and then test the implementation of SimpleEmployeeDB.

Implement the client methods

Implement assignment3.EmployeeDB interface

- Implement the following stubbed out methods in `assignment3.EmployeeDBHTTPClient` class.

a. `addEmployee`: Send a request to add an employee record in the employee database. Use *HTTP POST* [4] request to implement this method. You must add the employee record on the server determined by the `getServerURLForDepartment` method.

b. `listEmployeesInDept`: Send a request to the servers hosting employee records belonging to departments passed as argument (use `getServerURLForDepartment` to determine target servers) to list employee records belonging to that department on that server. Use *HTTP POST* request to implement this method.

c. `incrementSalaryOfDepartment`: Send a request to the servers hosting employee records belonging to departments passed as argument (use `getServerURLForDepartment` to determine target servers) to increment employee salaries of those departments by the specified amounts on that server. Use *HTTP POST* request to implement this method. You do not have to ensure all/nothing semantics across servers.

d. `cleanupDB`: Send a request to reset the state of the employee database to an empty state (newly created employee database). Use *HTTP POST* request to implement this method.

- *You also need to add code in the EmployeeDBHTTPHandler to multiplex the requests based on the request URI and invoke the corresponding methods in SimpleEmployeeDB on the server side.*

- *You also need to modify the "filePath" variable in EmployeeDBHTTPClient and EmployeeDBHTTPServer classes so that it stores the absolute path of the clientdepartmentservermapping.xml and serverdepartmentservermapping.xml file in your file-system. In the constructor of EmployeeDBHTTPClient, you should also create and start the HTTPClient.*

[3] Either all records are incremented or none are incremented on the same server. You do not have to ensure all/nothing semantics across servers.

[4] Use XStream (http://xstream.codehaus.org/) to serialize and de-serialize objects to xml strings and back before packing them in request or response objects. The XStream jar (https://nexus.codehaus.org/content/repositories/releases/com/thoughtworks/xstream/xstream/1.4.7/xstream-1.4.7.jar) files have been provided in the lib directory in the assignment handout. Make sure you add the lib directory in your buildpath.

- ***OPTIONAL*** You can now rerun the test cases you wrote in `assignment3.EmployeeDBTest` JUnit class and test the implemented `assignment3.Employe -eDBHTTPClient` class which will test the whole communication system. You can simply do this by switching the object in the setUpBeforeClass() method of the JUnit class (look at the hints in the comments in setUpBeforeClass method :-)).

***OPTIONAL* Build a client application - The JUnit test class**

- We have already built a client application i.e. `assignment3.EmployeeDBTest` JUnit test class. So, there should not be anything more to do. Just make sure that you have written sufficient test cases to test the SimpleEmployeeDB interface.