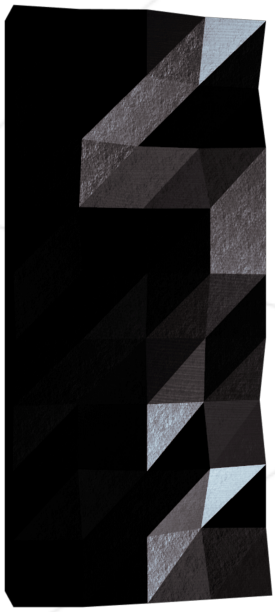


Module 08 – Piscine Java

Sockets

Summary: Today you will implement the basic mechanism of a client/server application based on Java—Sockets API

Contents

Chapter I

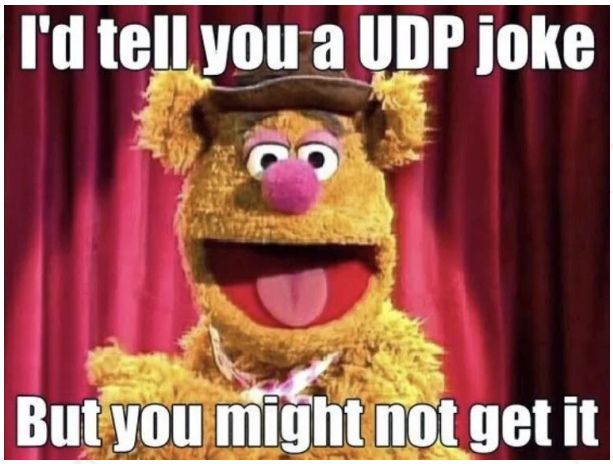
Foreword

The client/server interaction is the backbone of up-to-date systems. Server performs a large volume of business logic and information storage. As a result, the client application load is significantly reduced.

Dividing the logic into server and client components enables to flexibly build a general system architecture if server and client implementations are maximally independent.

Client and server communicate via numerous protocols described in OSI network model as different layers:

Layer	Example
7. Application	HTTP
6. Representation	ASCII
5. Session	RPC
4. Transport	TCP, UDP
3. Network	IPv4
2. Channel	Ethernet, DSL
1. Physical	USB, "twisted pair"



Chapter II

Instructions

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- Now there is only one Java version for you, 1.8. Make sure that compiler and interpreter of this version are installed on your machine.
- You can use IDE to write and debug the source code.
- The code is read more often than written. Read carefully the document where code formatting rules are given. When performing each task, make sure you follow the generally accepted Oracle standards
- Comments are not allowed in the source code of your solution. They make it difficult to read the code.
- Pay attention to the permissions of your files and directories.
- To be assessed, your solution must be in your GIT repository.
- Your solutions will be evaluated by your piscine mates.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your .gitignore to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google. And one more thing. There's an answer to any question you may have on Stackoverflow. Learn how to ask questions correctly.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- Use "System.out" for output

- And may the Force be with you!
- Never leave that till tomorrow which you can do today ;)

Chapter III

Exercise 00 : Registration

	Exercise 00
Registration	
Turn-in directory : $ex00/$	
Files to turn in : Chat-folder	
Allowed functions : All	

Before you start creating a full-scale, multi-user chat, you need to implement core functionality and build the foundational architecture of the system.

Now you need to create two applications: socket-server and socket-client. Server shall support connecting a single client and be made as a separate Maven project. Server JAR file is launched as follows:

```
$ java -jar target/socket-server.jar -- port=8081
```

Client is also a separate project:

```
$ java -jar target/socket-client.jar -- server-port=8081
```

In this task, you need to implement the registration functionality. Example of the client operation:

```
Hello from Server!
> signUp
Enter username:
> Marsel
Enter password:
> qwerty007
Successful!
```

Connection must be closed after Successful! message appears.

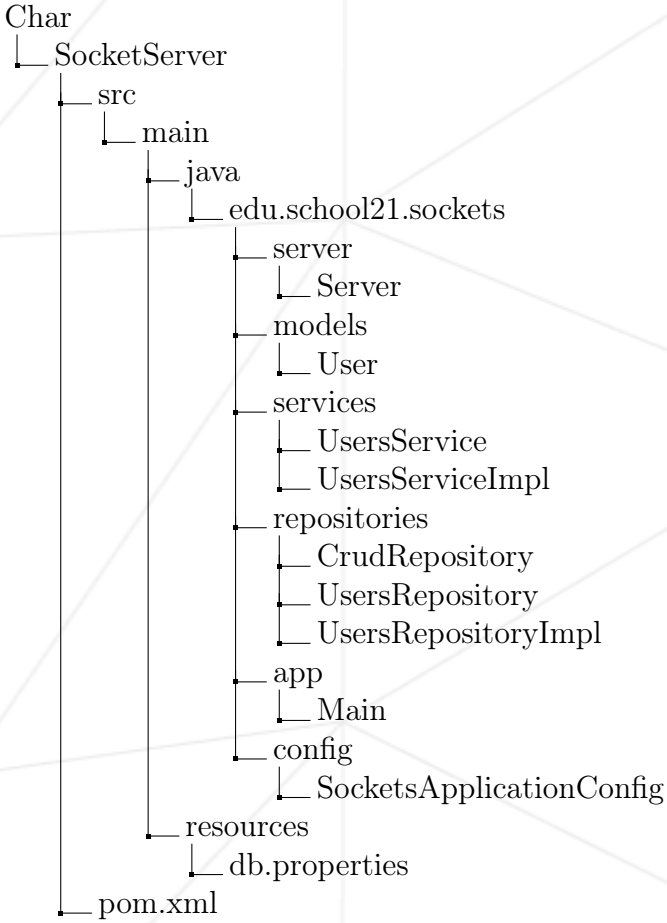
To ensure secure storage of passwords, use a hashing mechanism with PasswordEncoder and BCryptPasswordEncoder (see Spring Security components). Bin for this component shall be described in a class of SocketsApplicationConfig configuration and used in UsersService.

Key client/server interaction logic and the use of UsersService via Spring Context shall be implemented in Server class.

Additional requirements:

- Use a single DataSource—HikariCP
- Repository operation shall be implemented via JdbcTemplate
- Services, repositories, utility classes shall be context bins.

Server application architecture (client application is at the developer's discretion):



Chapter IV

Exercise 01 : Messaging

	Exercise 01
Messaging	
Turn-in directory : *ex*01/	
Files to turn in : Chat-folder	
Allowed functions : All	

Once you have implemented the application backbone, you should provide multi-user message exchange.

You need to modify the application so that it supports the following chat user life cycle:

1. Registration
2. Sign in (if no user is detected, close a connection)
3. Sending messages (each user connected to the server must receive a message)
4. Logout

Example of the application operation on the client side:

```
Hello from Server!
> signIn
Enter username:
> Marsel
Enter password:
> qwerty007
Start messaging
> Hello!
Marsel: Hello!
NotMarsel: Bye!
> Exit
You have left the chat.
```

Each message shall be saved in the database and contain the following information: Sender

- Message text

- Sending time

Note:

- For comprehensive testing, several jar files of the client application shall be run.

Chapter V

Exercise 02 : Rooms

	Exercise 02
Rooms	
Turn-in directory : $ex02/$	
Files to turn in : Chat-folder	
Allowed functions : All	

To make our application fully-featured, let's add the concept of "chatrooms" to it. Each chatroom can have a certain set of users. The chatroom contains a set of messages from participating users.

Each user can:

1. Create a chatroom
2. Choose a chatroom
3. Send a message to a chatroom
4. Leave a chatroom

When the user re-enters the application, 30 last messages shall be displayed in the room the user visited previously.

Example of the application operation on the client side:

```
Hello from Server!
signIn
SignUp
Exit
> 1
Enter username:
> Marsel
Enter password:
> qwerty007
Create room
Choose room
Exit
> 2
Rooms:
First Room
SimpleRoom
```

```
JavaRoom
Exit
> 3
Java Room ---
JavaMan: Hello!
> Hello!
Marsel: Hello!
> Exit
You have left the chat.
```

Using JSON format for message exchange will be a special task for you. In this way, each user command or message must be transferred to the server (and received from the server) in the form of a JSON line.

For example, a command for sending a message may look as follows (specific contents of messages are at the discretion of a developer):

```
{
    "message" : "Hello!",
    "fromId" : 4,
    "roomId": 10
}
```