Coursework: Distributed Application – Splitwise

Bachelors Software Engineering Course

Developed by Mihaela Ilieva 62550 and Nikoleta Beyska 62540

Table of contents

Introduction…………………………………………………………………………………3

Design………………………………………………………………………………………….3

Development……………………………………………………………………………….3

Deployment…………………………………………………………………………………4

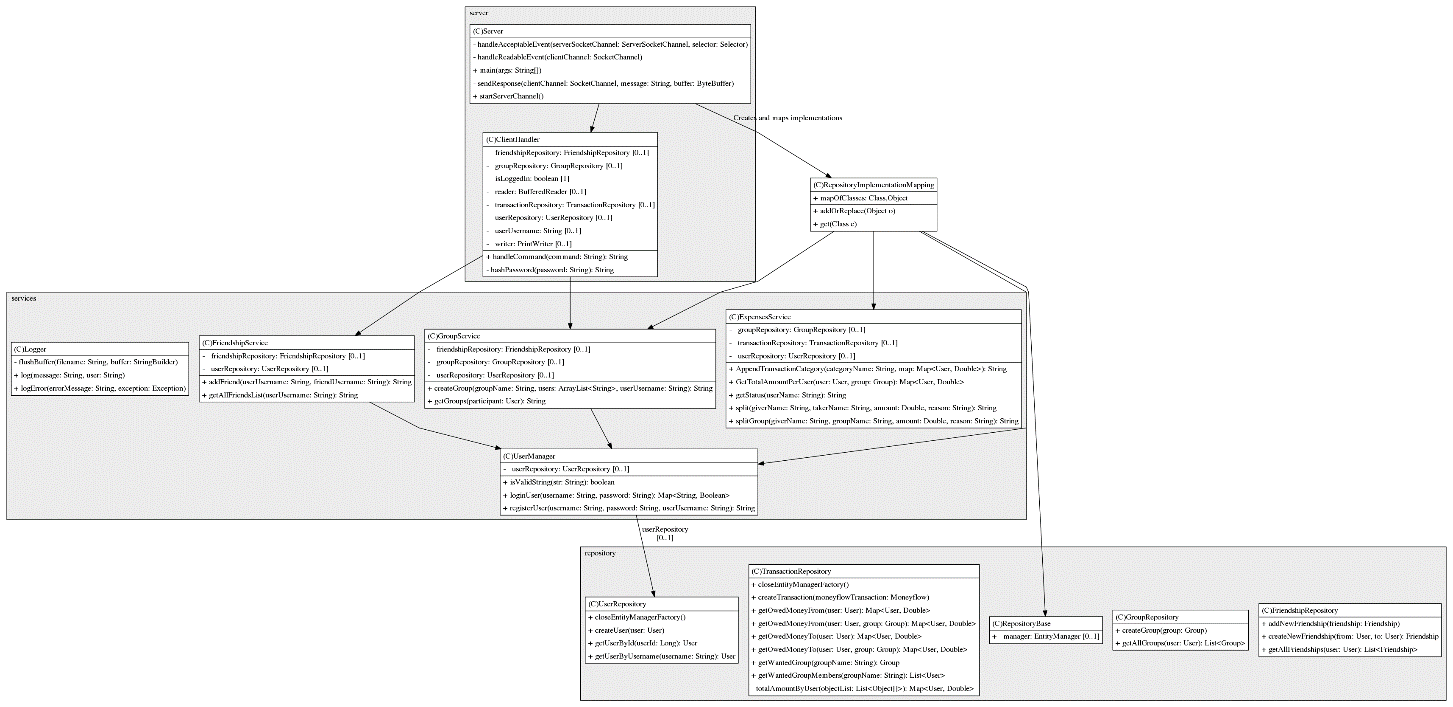
Testing………………………………………………………………………………………….4

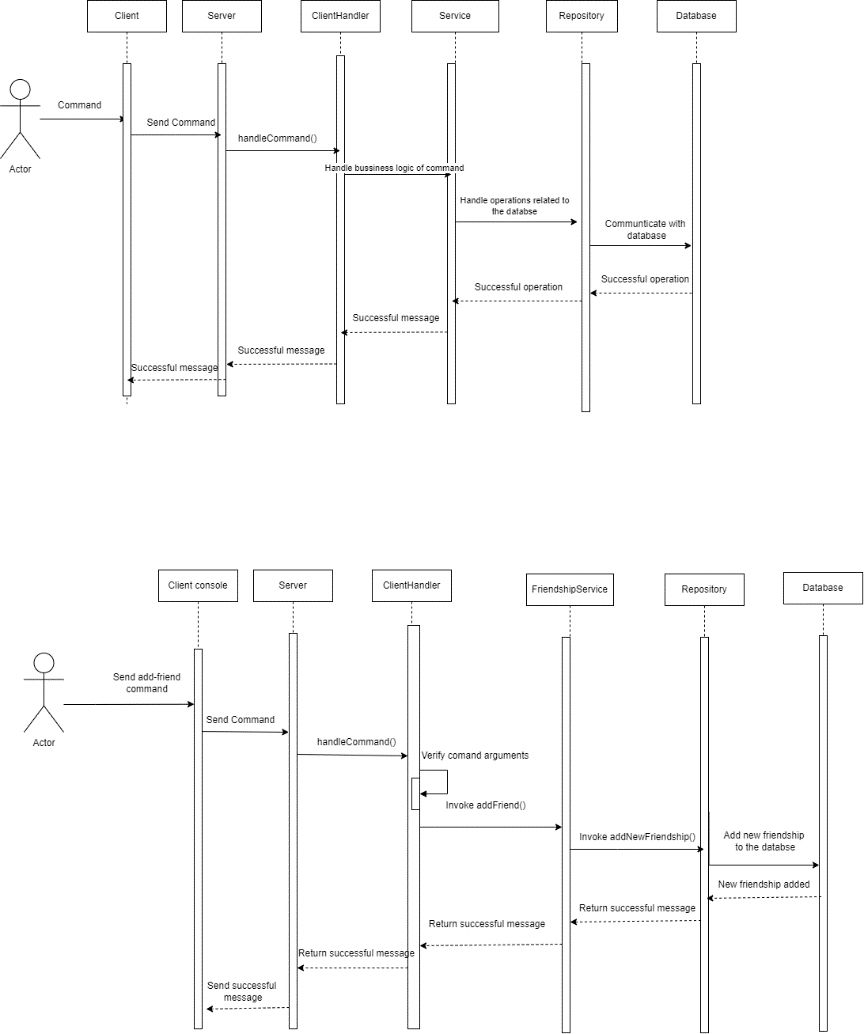
Application walkthrough………………………………………………………………5

Conclusion……………………………………………………………………………………7

1. Introduction

Our team has developed a distributed console application. The architecture we chose to design is client/server. The application is a simplified version of the popular expense-tracking app Splitwise. It aims to simplify sharing expenses with friends, individually or in groups. The client of our architecture is designed for our user’s needs, it accepts commands and sends them to the server. Then the server processes them, accepts the answer and gives it to the user in human-readable format. Our application uses a SQL database, established through object-relational mapping, using Jakarta Persistence. We also use Maven for project management. For testing we use Junit for writing Unit tests and Mockito for writing mockups. The communication protocol we chose to work with is TCP, as we need predictable and stable communication, as we exchange financial transactions.

1. Design 
2. Development



1. Deployment

You should first download Xampp and start the Apache and MySQL modules. After that open two command prompts, in one write java -jar %path\_to\_server\_jar\_file% for the server and write the equivalent for the client in the other one. When starting the server, the user should see “Server started on port 7777…”, and when connecting the client, the following should be seen in the command prompt – “Connected to server.”. After that the server should give a sign that the connection is established by the “Accepted connection from \127.0.0.1…”. Now the user can type their commands in the command prompt with the Client .jar file.

1. Testing

We used both Junit and Mockito for testing, and we tested both basic and more complicated functionalities. Our testing strategy was to test all possible situations with given commands, no matter how simple they appear.

As the first thing we coded was the client and server functionalities, we tested them manually with typing different inputs and checking if the server will return what we expected. We tested it with similar logic to the one in the Week3 Workshop Tasks.

After that we added the object-relational mapping with Jakarta Persistence, as well as some methods that would be helpful for the future development of the system. For all of those we wrote Unit tests, using Junit.

Later on, we started adding logic for the main functionalities, fully described in 6. Application walkthrough. As we did this, we created mocking tests for each one of them, using Mockito.

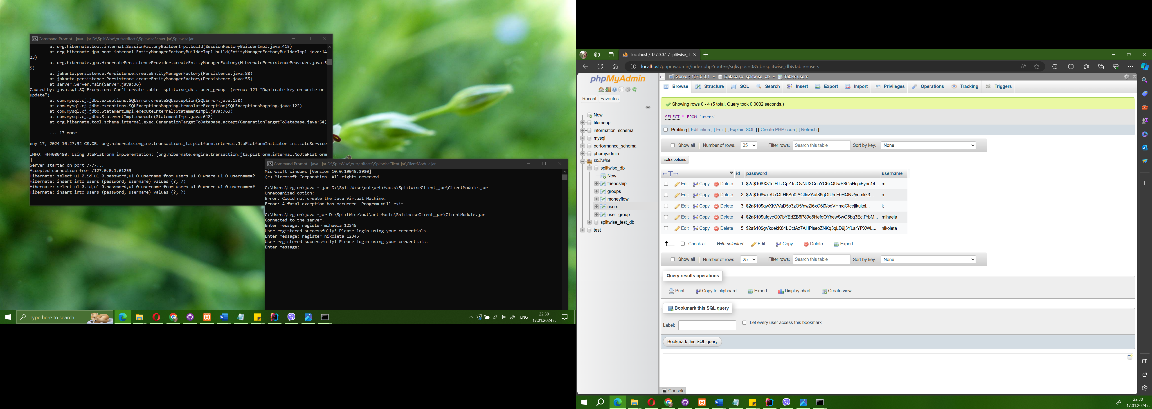
The unit tests are for the Service and Repository classes, the first of which heavily relies on mocks and the second of which relies on seeded test database.

All other logic, that we expect to be displayed for the user to see, we tested manually again, by purposely inputting invalid data (ex. Adding friend without being logged in first).

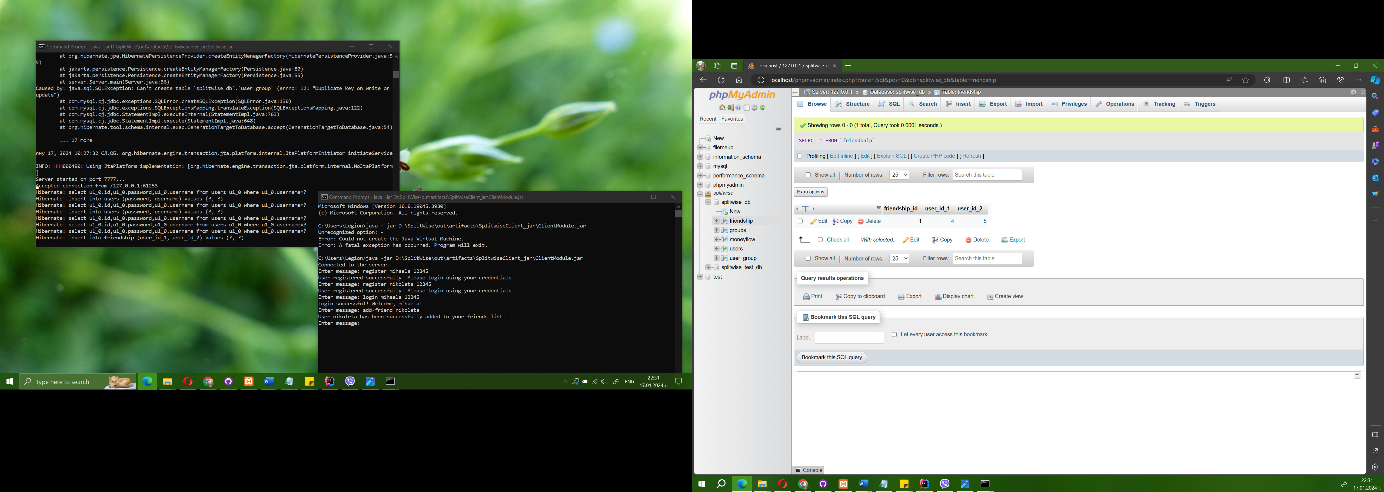
1. Application walkthrough

The functionalities our application supports are:

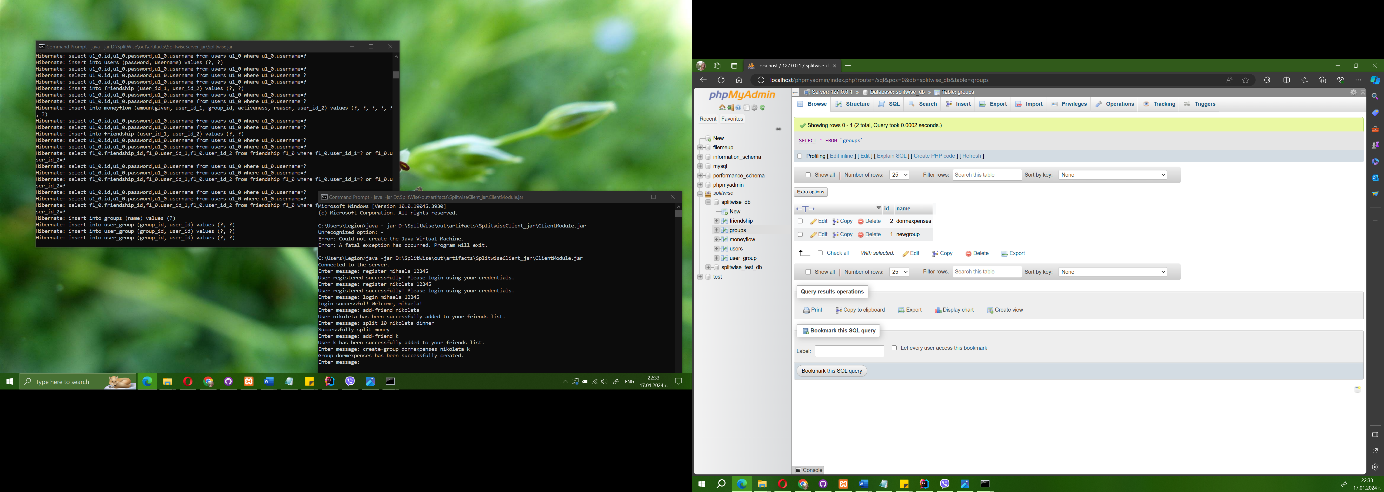
* 1. Registration – this happens with the command “register”, followed by the username and password the user wants to create an account with



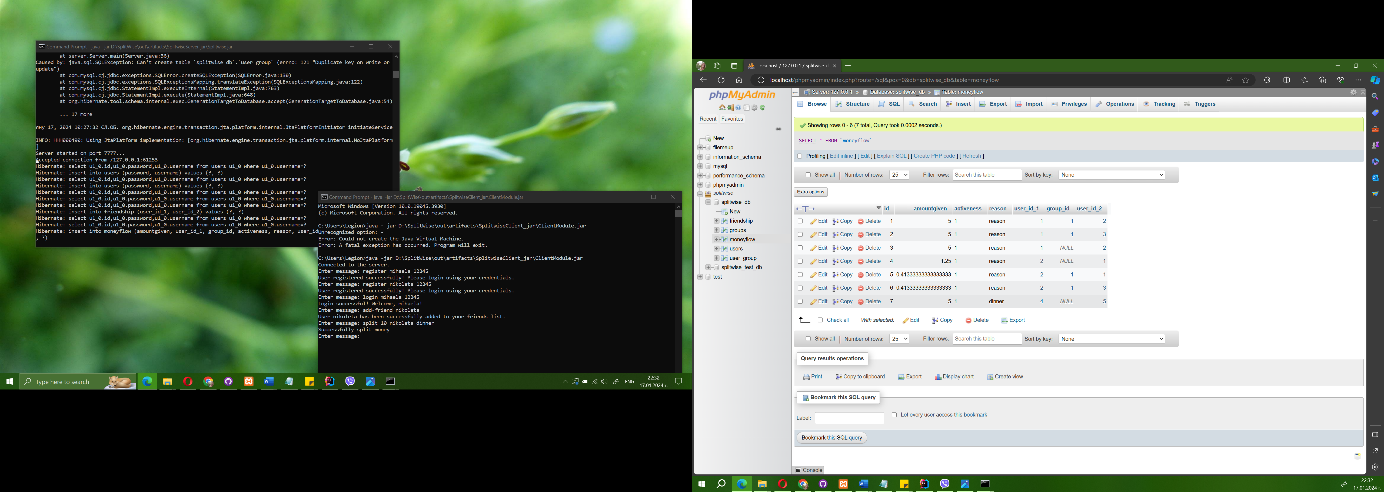
* 1. Log-in – functionality for already registered users. The user should write “login”, followed by username and password as well
  2. Logout – the user can log out of their profile by typing in the command “logout”
  3. Adding friend – a user can add another user as friend with the command “add-friend”, followed by the friend-to-be’s username



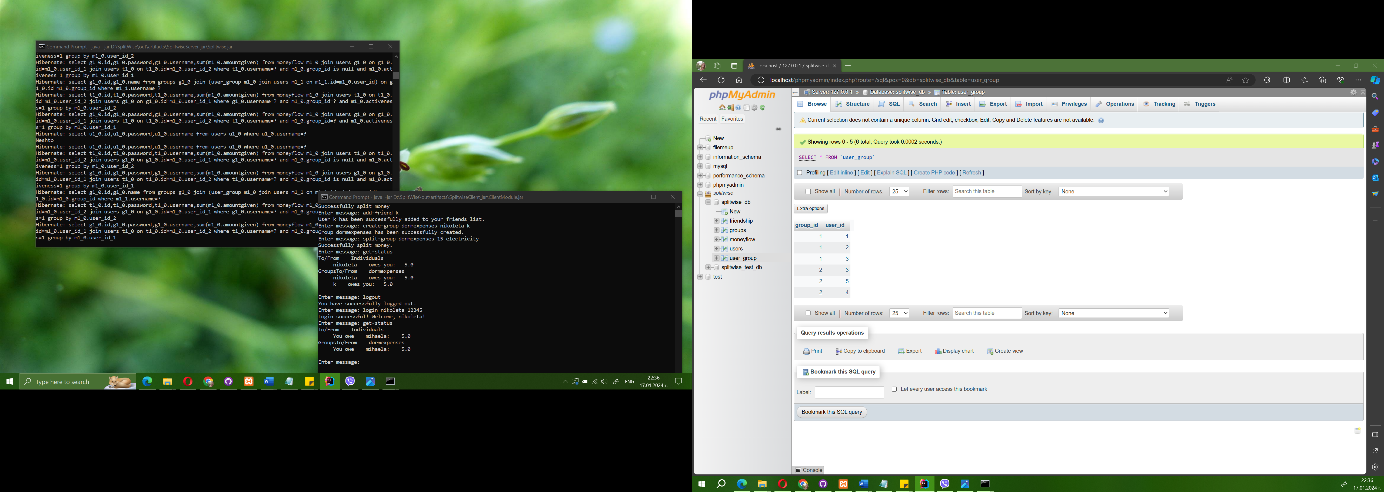
* 1. Creating a group – a user can create a group with other already registered users by typing in the command “create-group” after which they should write a name for the group and after that they should list all the participants



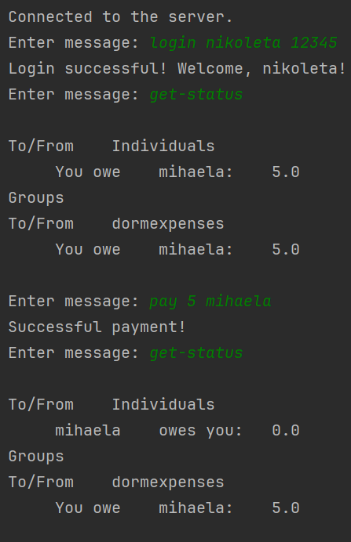
* 1. Adding expense to split with a friend – when the user types in “split”, followed by the friend’s username, the amount to be split and the reason for the split



* 1. Adding expense to split with a group – this functionality is accessed through typing the command “split-group” after which the user lists the group’s name, the amount to be split and the reason for the split
  2. Getting information about all the money the user owes or is owed to, grouped by types of relations – friendships or groups – the user types in “get-status” in order to receive this information



* 1. Paying a friendship expense – you can give the money you owe to your friend with the command “pay”, followed by the amount and the name of the friend
  2. Paying a group expense – you can pay what you owe to a group with the command “pay-group”, the amount and the group’s name



1. Conclusion

Our project supports basic functionalities for money transfer and debts between friends tracking. We have also added password hashing and operation logging. We use the technologies described in the introduction and we think our application has room for improvement with other directions, such as adding multithreading for example.