CS5031 - Practical 3

Group C
[160004376, 170019297, 190026921, 2100029990]

Word count: 1214

Note: For instructions on how to run the application, we refer to the README file in the root directory.

System Design

Spring Setup

In order to set up and protect our api endpoints we use the spring-boot-starter-web[1], -data-jpa[1], -jdbc[1] and -security[1] packages. Furthermore, for the api description we use the springdoc-openapi-ui[2] package. Most of the backend uses the default settings, with the security related settings being set in the SecurityConfiguration file in the utils package. The server is started through the Practical3Application class.

REST API

The REST API is mainly hosted in the FlatMateController class, with functionality for the A/B-testing hosted in the AbTestController class. Overall, we have 22 endpoints in the flatMateController class, as well as additional ones for the login and interactive api description. The AbtestController has a further endpoint for A/B test tracking. The endpoints in the FlatMateController class were all designed to perform certain workflows needed for the core functionality of our program. For the input of the endpoints, for the most part request parameters are used only if merely one value is needed, and JSON bodies if more than one is needed. This is in part due to more tidiness, both in the code as well as in the endpoints, and in part to the fact that all multi-parameter inputs correspond to a POJO anyway.

The API essentially works as an interface for the database, and propagates the inputs to the database. However, whenever needed, the API provides certain business logic. For example, in the createGroup endpoint, the API automatically adds the calling user to the group and makes them the admin. These logic implementations were implemented in order to keep both the SPA and CLI as slim as possible and provide consistent behavior through both interfaces. Wherever necessary the input to the API was validated using the InputValidatorUtils class. The REST API is tested extensively with 119 tests in the FlatMateControllerTest class.

Database

For persistent storage, the Flat Mate system uses a SQLite relational database. Initially, we chose SQLite for its simplicity and relative ease in terms of setting up, as it is a file-based database system. Figure 1 provides a relational diagram of the database.

[1] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.build-systems.starters
[2] https://springdoc.org/

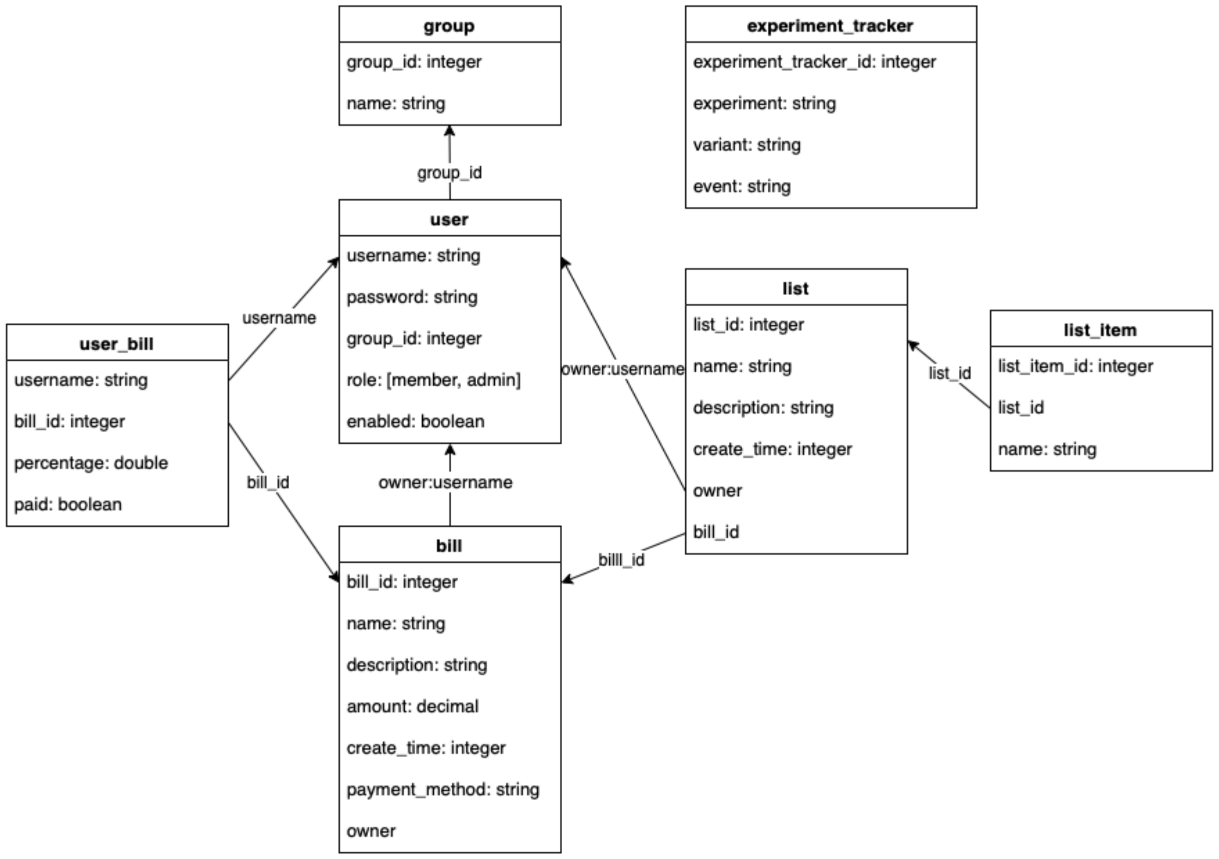


Figure 1: Database schema of *Flat Mate*

The setup of the database is handled from a SQL script (main/resources/db/schema.sql). To access the database the DataAccessObject class in the database package provides a number of methods that wrap SQL queries.

The testing of the DataAccessObject was done using JUnit. Unit tests were first written to confirm intended use cases functioned as expected. After these tests were finished tests probing the system with unintended use cases were written. Some examples of unintended use cases were creating duplicate database objects, retrieving objects that do not exist in the database, and testing methods with improper inputs, among others. All in all 59 tests were written for the DataAccessObject resulting in 100% code coverage for the DataAccessObject.

Clients

Single Page Application

We developed a single page application (SPA) which serves as the main client of our system. The SPA was developed using *Vue.js,* a framework for building reactive web applications. The app communicates with the REST API using the *axios* library. Further, *Vuex* is used for state management and *Vue Router* for routing of the different views.

Terminal Client

The terminal client is built on top of curl and consists of two scripts. One which can handle hashing passwords for login and account creation requests and one for all other requests. The client stores and uses session cookies and can use json files as the input body. The hashing is done using a separate javascript file to ensure it provides the same functionality as the frontend.

Security

On the server side, the endpoints of the REST api are protected by Spring to only allow access from authorized users. This is done via the Spring Security package through a custom authority system. While the endpoints for the server running test, the API description, and the user creation are accessible for all, every other endpoint requires authorization. The endpoints for adding a member to the group, removing a member from the group and changing the admin of the group are only accessible if the user holds admin rights. The login is handled through the security configuration, with both httpBasic auth and a form-based authentication available. To ensure data integrity, all passwords are hashed in the clients using SHA256 and a salt. Thus, the backend only handles hashed passwords. On top of that, every User object given back to an API caller will have their password nulled. This is handled through the DataProtection interface and the protect() method. All security features of the backend are accounted for in our FlatMateController tests.

On the SPA side, Vue router also does some authorization to prevent, for example, non-admins from accessing the admin page. When the user first logs in, the REST API returns the role of the user and is used to determine which views are accessible to the client. This is not the most secure way to handle this (since users could mock the login response to set a different role). However, this would not affect their privileges when using the REST API. Thus, while nefarious users could break the client (which would only hurt themselves) they cannot use this technique to escalate privileges on the server.

The CLI does not have specific security implementations as the endpoints are already covered through the REST api.

A/B Testing

A/B testing was handled mostly on the frontend in addition to a database table of the server to track experiment results.

On the web application, the A/B testing framework *AlephBet*[3] was integrated to define experiments. Each experiment defines the different variants of the tests, which are then randomly shown to users. In addition, we can define a goal for each experiment which, if completed, is tracked. *AlephBet* stores the variant shown to the user in local storage to always show the same variant to a user (and not switch upon reload). An experiment adapter is used to track events of the experiment (‘participate’, ‘goal completed’).

On the server, we defined a separate controller that exposes an endpoint to which the experiment adapter posts events. The events are stored in the *experiment_result* table for further analysis.

Figure 2 shows the variants of the first A/B test, which turns the default theme either to “simple” or “fancy”. The goal is defined by the user clicking the “switch theme” button in the top right corner. This test could be used to check if users prefer a simple or fancy background based on how likely they are to click the “switch theme” button.

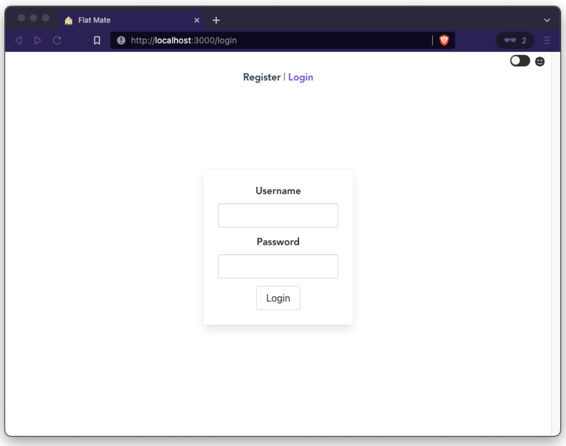
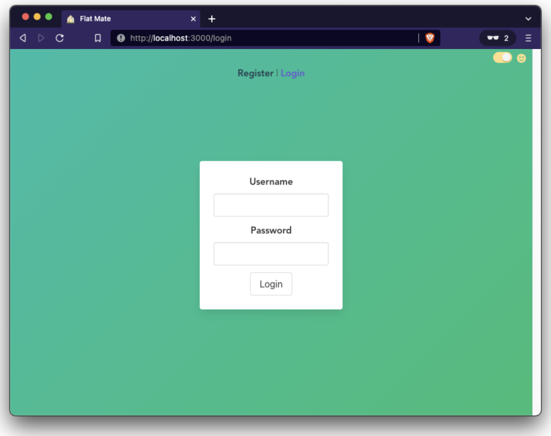
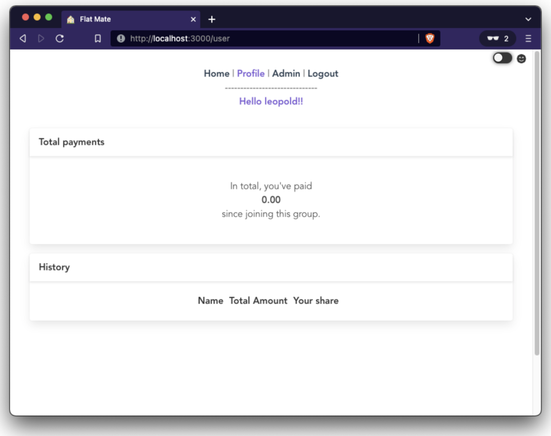
	
a) Simple theme	b) Fancy theme

Figure 2: “Simple” and “Fancy” variants of the *Toggle Theme* A/B testing experiment.

Figure 3 shows the second A/B test, which activates/deactivates a component on the “User” page of the SPA that allows users to pay all due bills in one click. The goal is marked as completed if the user clicks on the “Pay all due bills” button. This could be used to test if users would make use of this functionality.

[3] https://github.com/Alephbet/alephbet



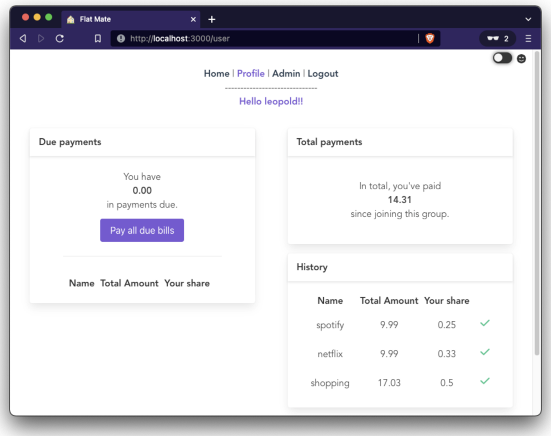
Total payments

In total, you've paid
0.00
since joining this group.

History

Name	Total Amount	Your share

a) Inactive



Due payments

You have
0.00
in payments due.

Name	Total Amount	Your share

Total payments

In total, you've paid
14.31
since joining this group.

History

Name	Total Amount	Your share	
spotify	9.99	0.25	✓
netflix	9.99	0.33	✓
shopping	17.03	0.5	✓

b) Active

Figure 3: Variants of the *Pay All Bills* A/B testing experiment.

REST interface

→ see docs/api-documentation.pdf