CS5031 P3 Group Report

190005675, 220017897, 220032952

Contents

1 Introduction

In this practical, we developed an estate agency system. This system stores information about users, buildings and flats. Users can view, add to and update the database based on their role and their permission. There are three types of users:

- Guests can only view and edit their own user information. They can only view buildings and flats.
- Managers can only view and edit their own user information. They can view, add to and edit buildings and flats.
- Admins can view and edit all user information. They can view, add to and edit buildings and flats. They can view and update roles and permissions for users.

The system consists of a Java server, a web client and a terminal client. Users can interact with the system using the two clients.

1.1 Starting the Java server

To run the Java server on port 8080, run

```
java -jar cs5031p3code-server-0.0.1.war
```

1.2 Using the web client

To run the web client, make sure the Java server is running.

Run under the `EstateAgencySystemGUI` directory:

```
npm i
npm start
```

Then open the url displayed in the terminal in a browser.

1.3 Using the terminal client

To run the terminal client, make sure the Java server is running.

Run under the `EstateAgencySystemTerminal` directory:

```
mvn compile exec:java -Dexec.mainClass=
            "stacs.estate.cs5031p3code.TerminalClientMain"
```

Users should see a menu screen showing a list of options in the format of

```
[code] option
```

To select an option, input the code within the square brackets. Similar logic can be used to navigate between menus in the terminal client.

2 Overview of Implementation

2.1 Java Server

2.1.1 Technologies

The following technologies were used in the Java server:

- SpringBoot to build and deploy the Java Server.
- Spring Security to provide authentication and authorisation support.
- MySQL to support the storage of structured data in this system.
- MybatisPlus, a popular Structured Query Language (SQL) mapping framework for Java, to simplify the process of working with databases in Spring applications.
- Redis, which is a high-performance key-value database, to store data that the system needs to store and retrieve quickly, such as user authentication information.

2.1.2 Architecture

In this system, we adopted a development model based on the Model-View-Controller (MVC) design pattern, and the specific information is shown in Figure 1.

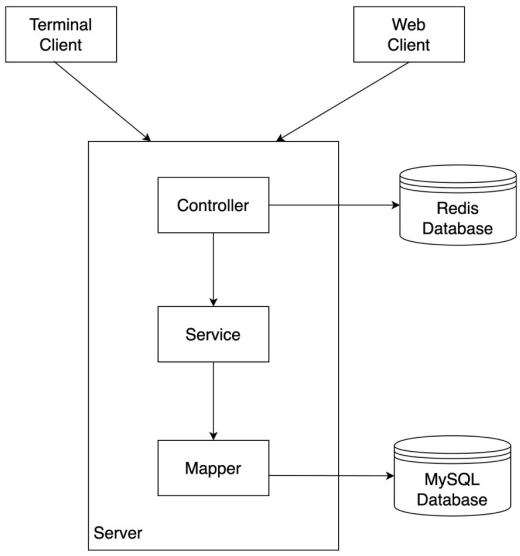

Figure 1: The system architecture graph

As Figure 1 shows, the server can be divided into three levels: the controller, the service, and the mapper. Whenever the client communicates with the server, the following process happens

1. Clients send service requests through the controller (API) methods exposed by the system.
2. The system API uses services by calling the corresponding service methods.
3. Service methods implement operations such as data exchange on the database by calling mapper methods.
4. Mapper methods represent a series of basic operations on the MySQL database (mainly representing a set of addition, deletion, modification, and query operations).

In addition, controller methods operate the data from Redis when the system needs to authenticate and authorise users.

2.1.3 Authentication and Authority

the Spring Security framework controls the authentication and authorization of the system. We used an authentication method based on JSON Web Token (JWT), which uses JSON-encoded tokens to authenticate users of web applications. In addition, we used the Role-Based Access Control (RBAC) for authorisation, which involves assigning roles to users and granting permissions to roles.

Figure 2 describes the login process and verification process about other APIs, and Figure 3 describes how the system uses Spring Security to control user authentication and authority.

As shown in Figure 2, when the system processes the user's login request, the system verifies the user's email and password, and if the verification is successful, it generates a JWT, which is stored temporarily in the Redis database (meaning that the JWT has shelf life), and the system returns the JWT to the Client.

When dealing with other APIs that require JWT for identity verification, the client first passes the request to the server by including the JWT in the request header, and then the server parses and verifies the JWT. If parsing the JWT succeeds, the system can get the current user's id. Finally, the system obtains the user's permission information according to the current user id, and judges whether there is permission to access the resource according to its permission content, and then returns the corresponding result to the Client.

Figure 3 shows the process of authentication and authorisation. On receiving an HTTP request, the custom `JwtAuthenticationTokenFilter` interceptor in our system intercepts the request, and uses the tool class `JwtUtil` to parse and verify the JWT in the request header. Then, the system fills in the `SecurityContext`

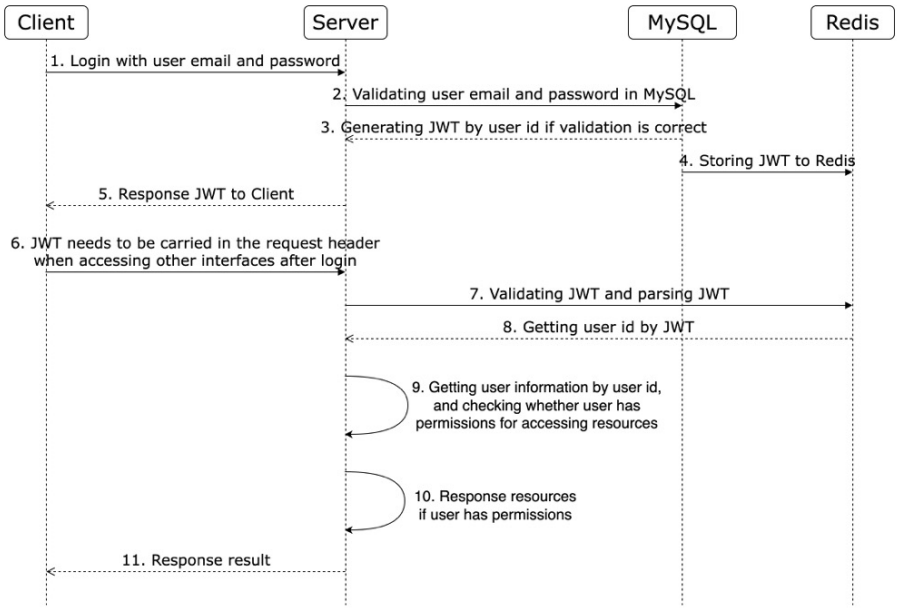

Figure 2: The process of login and other APIs verification

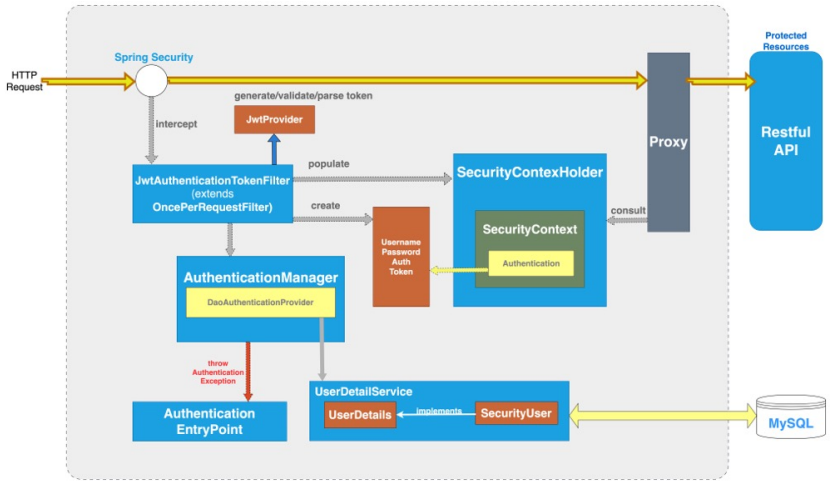

Figure 3: The process of authentication and authorisation

according to the parsed JWT (mainly populating the user information and corresponding permission information into the current `SecurityContext`). In addition, the system encapsulates a `UsernamePasswordAuthToken` object based on the information in the `SecurityContext` (if there is no JWT information in

the request header, the system directly creates the object based on the newly generated JWT information).

When the request passes through the system `JwtAuthenticationTokenFilter` interceptor, the user's authentication and authorization is processed through the `AuthenticationManager` in `SpringSecurity`. If an exception occurs during this phase, it would be captured by the `AuthenticationEntryPoint` object. The specific authentication details are actually the implementation class `DaoAuthenticationProvider` of `AuthenticationManager` by calling the method in `UserDetailsService` for authentication. The actual authentication object in this system is `SecurityUser`, which implements `UserDetails`.

The above information only includes basic content about authentication and authorization in this system by using Spring Security. More details are shown on the following classes:

- CorsConfiguration: including such configurations about handling with the problem of CORS (Cross-origin resource sharing) in Spring Boot.
- RedisConfiguration: including the basic configurations of Redis, in fact, RedisTemplate will be used when storing data into Redis.
- SecurityConfiguration: including all configurations about Spring Security in this system, such as handling CORS problems, adding custom filters, and configuring custom exception handlers.
- JwtAuthenticationTokenFilter: the JWT filter/interceptor in this system for handling with JWT.
- AuthenticationEntryPointImpl: the handler for handling exceptions in the authentication process.
- AccessDeniedHandlerImpl: the handler for handling exceptions in the authorization process.
- SecurityUser: the user in the Spring Security context, for implementing UserDetails and doing authentication and authorization. Mainly including user information and permissions details of it.
- UserDetailsServiceImpl: the implementation of `UserDetailsService`, by the way, the default authentication in Spring Security is based on memory storage. Using `UserDetailsServiceImpl` in this system will change the default authentication way to a database-based way.
- FastJsonRedisSerialiser: the serialiser for serialising data when needed in Redis storage.
- JwtUtil: the utilised class for JWT, including parsing JWT, creating JWT and so on.
- RedisCache: series of methods for operating Redis cache.

2.1.4 Database

The database is the underlying system that stores relevant information about buildings, flats, users, roles, permissions as well information about the relationship between users to roles and roles to permissions. Figure 4 and Appendix A shows more details about the database.

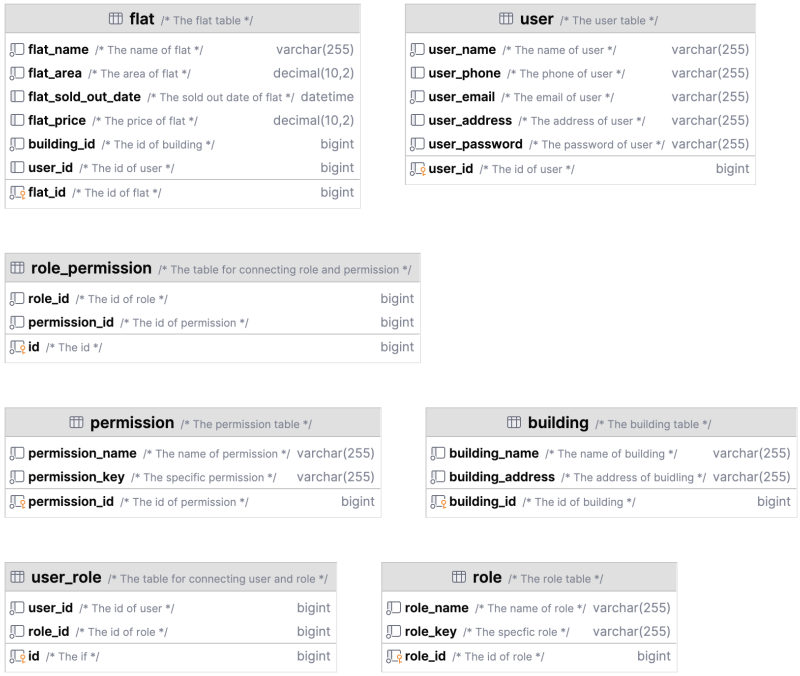

Figure 4: Design of database

2.1.5 Model (Entity, Mapper and Service)

As shown in the system architecture diagram, the system splits the model layer in MVC when designing it. It includes an entity layer representing system data (corresponding to the data in the database), a mapper layer representing multiple sets of mapping relationships (mapping between entity and database data), and the service layer representing actual business logic.

Package `stacs.estate.cs5031p3code.model` contains the entity layer code. In addition, the package `stacs.estate.cs5031p3code.mapper` contains the code of the mapper layer. The XML file in the mapper folder under the resources directory represents the mapping relationship. The interface and business logic implementation of the service layer lies in `stacs.estate.cs5031p3code.service` and its subpackage `impl`. Since the entity layer and mapper layer were auto-

matically generated by the MybatisPlus framework, they were not elaborated in this report.

Bundles of test cases about service level were devised based on the design of the system. For each service test class, edge cases as well as common cases were considered. For instance, when testing adding a new flat into a building, we considered four situations: 1) Flat is null or having empty data in attributes which should not be null. 2) The building does not exist. 3) Flat name exists within the building. 4) Adding a flat is successful. Due to the use of MySQL, we used an annotation named `@Transactional` in each service test to be responsible for database transaction management and data rollback. Each database table has a corresponding service and implementation of service.

2.1.6 Controller

The Controller layer comprises various controllers, including:

- Building Controller
- Flat Controller
- User Controller
- Permission Controller
- Role Controller
- RolePermission Controller
- UserRole Controller

In our software design, we followed several design principles to ensure the quality of our codebase. One of the key principles we applied was the Single Responsibility Principle (SRP), which states that each class should have only one reason to change. We also applied the Open-Closed Principle (OCP), which states that classes should be open for extension but closed for modification, and the Dependency Inversion Principle (DIP), which states that high-level modules should not depend on low-level modules but both should depend on abstractions. [2]

To implement these principles, we used the controller layer to connect various layers of the software, including RedisDatabase, Gui, and Service layers. The FlatController class, in particular, is an essential part of this layer, responsible for handling all APIs related to the flat. It provides authorization checks to ensure that only authorized users can perform certain operations, enhancing security.

Our design principles, along with the use of the controller layer and FlatController, allowed for better separation of concerns and easier maintenance of the codebase, resulting in improved software quality. [1]

2.2 Web Client

The web client of the estate agency platform was developed using the ant design component library, which provides a rich set of pre-designed UI components that are both functional and aesthetically pleasing. The use of ant design helped to streamline the development process, reduce the need for custom CSS and JavaScript code, and ensure consistency in the user interface across the application.

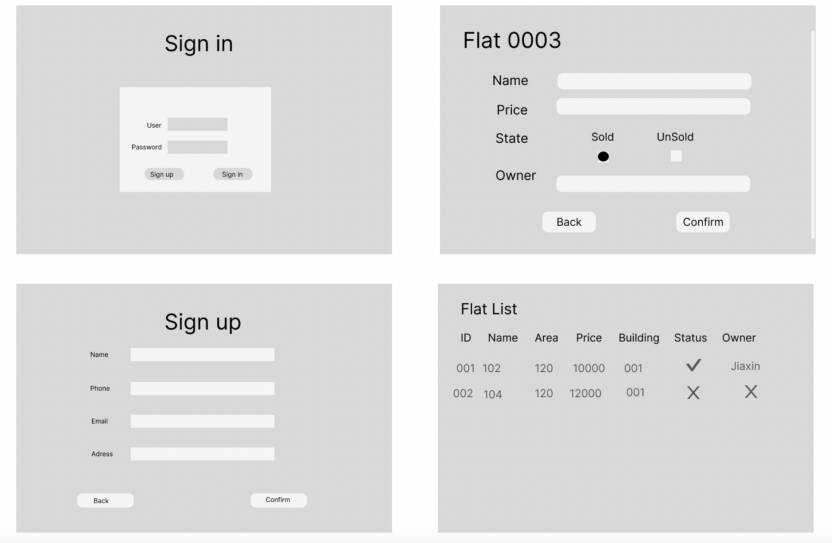

Figure 5: UI design via *figma* for login, registration, apartment list display, and apartment information modification

To handle the communication with the backend service, the web client uses `axios`, a popular JavaScript library for making HTTP requests. Axios is known for its ease of use, robustness, and support for modern features such as promises and async/await. By using axios, the web client can send and receive data from the backend service in a simple and efficient manner, making the overall user experience smooth and responsive.

The system uses role-based access control (RBAC) to manage permissions for different types of users. This approach allows administrators to assign specific roles to users and control their access to various parts of the system. The choice of RBAC helps to maintain security and ensure that only authorized users can perform certain operations.

The frontend also provides a profile editing feature that allows users to modify their personal information such as name and email address. The use of a dropdown list to select the building of a flat helps to ensure data consistency and avoid errors that could arise from manually entering the building ID. The ability

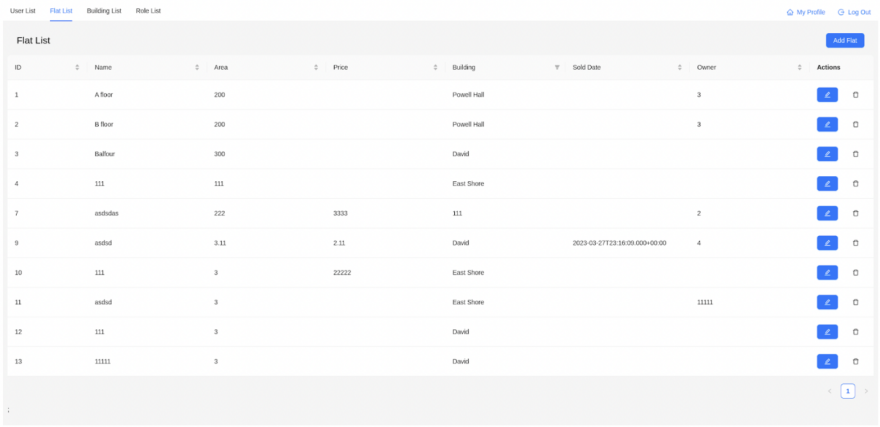

Figure 6: Screenshot of the web UI

to sort and filter lists enhances the user experience by enabling users to quickly find the information they need.

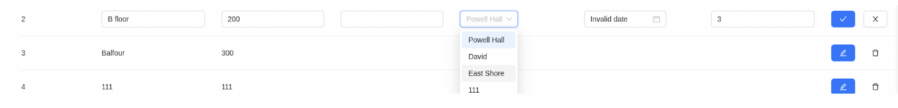

Figure 7: Usage of drop-down lists

Finally, the application provides clear feedback to users by displaying success or failure messages after each operation. This helps users to understand the outcome of their actions and take appropriate steps if necessary.

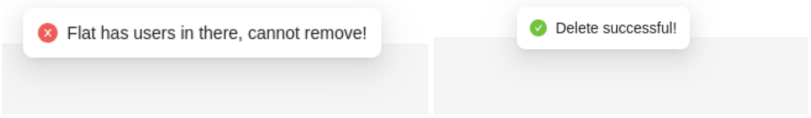

Figure 8: Message prompts on failing to delete a flat (left) and successfully deleting a flat (right)

2.3 Terminal Client

The terminal client interacts with the server via HTTP requests. It provides an user interface in the command line.

2.3.1 Design

Three main components were identified in the design of the terminal client:

- `TerminalClient` a client to make HTTP requests to the server;
- `ResponseHandler` a handler to interpret and extract data from the response received from the server;
- `Menu` a terminal interface used by users to interact with the system.

Since there were quite a few endpoints provided by the server, we decided to have many smaller sized sub-menus rather than a huge long menu. Each sub-menu provides a list of options and responds to the user based on user input.

2.3.2 Test

A test-driven approach was applied when implementing the terminal client. For each of the APIs provided by the server, a group of tests were written. These tests were written in such a way that they did not have to interact with the actual server; instead, `MockWebServer`[1] was used to mock a server. We also mocked the response returned by the server. Various scenarios were considered, including when server responds with a successful HTTP status code (200 OK) and unsuccessful codes.

As new features were added to the system, a set of new tests regarding the new features were added in the following way:

- A mock server was created, with mock responses for each endpoint (both successful and unsuccessful)
- The function in the terminal client which calls the endpoint was executed
- The expected response from the terminal client was asserted

We also ensured that commits were only made when all test cases passed, and it was easy to revert our code to a previous state if anything failed.

2.3.3 Implementation

The implementation of the client involved using a `WebClient`[2]. Each of the APIs provided by the server has a corresponding function in the client code. It takes arguments of an action and encodes them in a format that could be sent

[1] `https://square.github.io/okhttp/3.x/mockwebserver/index.html?okhttp3/mockwebserver/MockWebServer.html`

[2] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html`

to the server via HTTP requests. It then passes the response from the server to `ResponseHandler` as a map.

The `ResponseHandler` only contains static methods, which means that we did not need to create an instance of this class and we could use the methods directly. The methods in this class parse a response received from the server. Since the response map is parameterized with wildcard type, each of the methods attempt to do type casting based on what we expect the response to contain.

The `Menu` is an interface with one method `run()`. All sub-menus implemented this interface. The `run()` method decides the logic of the current sub-menu. It reads in user input using a `Scanner` object, and gives various responses to the user based on the input. The same `Scanner` object was passed among the sub-menus when navigating through them. The scanner was closed on exiting the system to avoid wasting resources.

3 Software engineering practice

3.1 Test-driven approach

A test-driven approach was used throughout the project. For each feature of the system, a set of test cases were devised. New tests were added as we added more features to the system. Normal test cases as well as edge test cases were considered, to guard our system against erroneous input. Commits to GitLab were only made when the current code passed all tests.

3.2 Version control

We used `git` for version control and `GitLab` for repository management and merge requests. Each of us worked independently in our own branch to avoid conflicts, and we created merge requests to the main branch when needed. Code review happens before merging a feature branch into the main branch. Each commit has a clear commit message which reflects what has been done.

4 Agile Development

4.1 Task management

We used Trello[3] to manage our project. Figure 9 shows the Kanban board used throughout this project. We used this board to track our project status and update each other what we have done, what we were doing and whether we were stuck. Each task was assigned to at least one group member and a deadline was provided to push our progress.

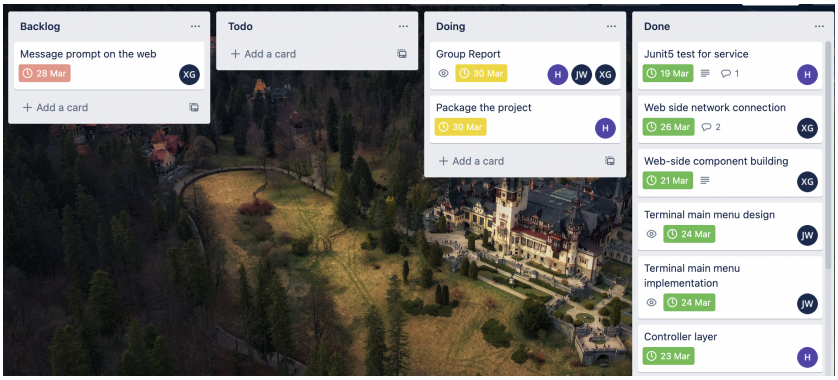

Figure 9: Kanban board created at the beginning of the task

[3] `https://trello.com/`

4.2 Scrum meetings

We utilised three methods for conducting stand-up meetings, including face-to-face meetings, online meetings through teams, and collecting updates through forms (as shown in Figure 10). These different approaches were used according to the schedules of team members. However, daily communication is ensured through these methods. A summary of our stand-up meetings is listed in Appendix B.

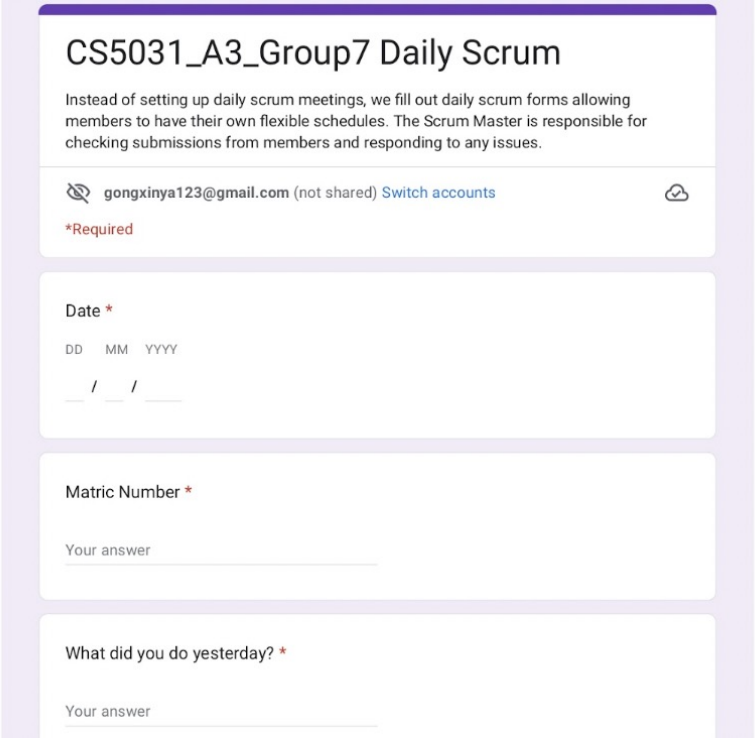

Figure 10: Forms for Standup Alternatives

4.3 Sprints

Sprint Record - End of Sprint 1 (March 19th - March 21st)

Sprint Goals:

- Implement basic functionality for managing and viewing real estate information
- Create a working prototype of the system
- Develop a testing strategy and write unit and integration tests

Tasks accomplished during the sprint:

- Designed and implemented the system architecture
- Set up the development environment and databases
- Created a basic user interface for the web client
- Implemented user authentication and authorization
- Implemented CRUD operations for buildings and apartments
- Added validation and error handling for input data
- Wrote unit and integration tests for the implemented functionality

Challenges faced during the sprint:

- Lack of experience with certain technologies
- Difficulty in coordinating work between different team members
- Time constraints due to having other deadlines and commitments

Sprint Retrospective:

- Overall, the team made good progress towards the sprint goals
- Communication and coordination between team members could be improved
- More emphasis on testing and quality assurance is needed going forward

Sprint Record - End of Sprint 2 (March 25th - March 27th)

Sprint Goals:

- Refactor and optimize the existing codebase
- Add more functionality and polish to the user interface
- Write documentation and conduct a code review
- Prepare for deployment

Tasks accomplished during the sprint:

- Refactored and optimized the existing codebase
- Added error logging and monitoring functionality
- Implemented pagination and sorting functionality for listings
- Created a more polished user interface for the web client
- Wrote documentation for the system
- Conducted a code review and addressed any issues or feedback

- Finalized the system and prepared for deployment

Challenges faced during the sprint:

- Need for more in-depth testing and quality assurance
- Difficulty in meeting all sprint goals within the remaining time
- Potential issues with deployment and server configuration

Sprint Retrospective:

- The team made good progress towards the sprint goals and completed most of the planned tasks
- More emphasis on testing and quality assurance is still needed going forward
- Communication and coordination between team members has improved, but could still be further improved
- Planning and prioritization of tasks could be more effective to ensure all goals are met within the available time

References

[1] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1st edition, 2009.

[2] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

A Database Structure

Attributes	Type	Requirement
user_id	bigint	Primary key
user_name	varchar	Not null
user_phone	varchar	
user_email	varchar	Not null and not repeated
user_address	varchar	
user_password	varchar	Not null

Table 1: User table

Attributes	Type	Requirement
building_id	bigint	Primary key
building_name	varchar	Not null and not repeated
building_address	varchar	Not null

Table 2: Building table

Attributes	Type	Requirement
flat_id	bigint	Primary key
flat_name	varchat	Not null
flat_area	decimal(10, 2)	Not null
flat_sold_out_date	datetime	
flat_price	decimal(10, 2)	
building_id	bigint	Not null
user_id	bigint	

Table 3: Flat table

Attributes	Type	Requirement
permission_id	bigint	Primary key
permission_name	varchar	Not null
permission_key	varchar	Not null

Table 4: Permission table

Attributes	Type	Requirement
role_id	bigint	Primary key
role_name	varchar	Not null
role_key	varchar	Not null

Table 5: Role table

Attributes	Type	Requirement
id	bigint	Primary key
user_id	bigint	Not null
role_id	bigint	Not null

Table 6: User Role table

Attributes	Type	Requirement
id	bigint	Primary key
role_id	bigint	Not null
permission_id	bigint	Not null

Table 7: Role Permission table

B Scrum Meetings

Standup Record - Day 1 (March 17th)

Tasks to be accomplished today:

- Brainstorm potential project ideas
- Decide on a project
- Project planning and requirement gathering

Potential roadblocks:

- Time constraints due to having other deadlines

Standup Record - Day 2 (March 18th)

Tasks accomplished since last meeting:

- A project was decided (estate agency system)
- System requirements were discussed and agreed

Tasks to be accomplished before next meeting:

- Setting up the development environment
- Designing the architecture of the system
- Implementing CRUD (Create, Read, Update, Delete) operations for buildings and apartments
- Adding validation and error handling for input data
- Writing unit tests for the implemented functionality

Potential roadblocks:

- Difficulty in implementing complex business logic
- Lack of experience with certain technologies

Standup Record - Day 3 (March 23rd)

Tasks accomplished since last meeting:

- Created basic RESTful API endpoints
- Set up the Redis and MySQL databases
- Implemented user authentication and authorization
- Implemented CRUD operations for buildings and apartments
- Added validation and error handling for input data

- Wrote unit tests for the implemented functionality

Tasks to be accomplished before next meeting:

- Implementing user roles and permissions
- Adding search and filtering functionality for buildings and apartments
- Writing integration tests for the system
- Creating a basic user interface for the web client
- Creating a basic user interface for the terminal client

Potential roadblocks:

- Time constraints due to having other deadlines

Standup Record - Day 4 (March 26th)

Tasks accomplished since last meeting:

- Implemented user roles and permissions
- Added search and filtering functionality for buildings and apartments
- Wrote integration tests for the system
- A simple web interface was created
- A simple terminal interface was created

Tasks to be accomplished before next meeting:

- Refactoring and optimizing the existing codebase
- Adding error logging and monitoring functionality
- Continuing to write more tests for edge cases and error scenarios
- Continuing to work on the web interface
- Continuing to work on the terminal interface

Potential roadblocks:

- Complexity of the business logic and data model
- Need for more in-depth testing and quality assurance

Standup Record - Day 5 (March 28th)

Tasks accomplished since last meeting:

- Refactored and optimized the existing codebase
- Added error logging and monitoring functionality

- Wrote more tests for edge cases and error scenarios
- Web interface was completed
- Terminal interface was mostly completed

Tasks to be accomplished before next meeting:

- Implementing pagination and sorting functionality for listings
- Creating a more polished user interface for the web client
- Writing documentation for the system
- Connecting web interface to the server
- Connecting terminal interface to the server

Potential roadblocks:

- Need for more in-depth testing and quality assurance
- Difficulty in meeting all sprint goals within the remaining time

Standup Record - Day 6 (March 29th)

Tasks accomplished since last meeting:

- Implemented pagination and sorting functionality for listings
- Created a more polished user interface for the web client
- Wrote documentation for the system

Tasks to be accomplished today:

- Conducting a code review and addressing any issues or feedback
- Finalizing the system and preparing for deployment
- Planning and preparing for the next sprint

Potential roadblocks:

- Need for further testing and quality assurance
- Potential issues with deployment and server configuration