PeerPrep

CS3219 - Software Engineering Principles and Patterns

Project Report

Group 1

Source Code: https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g1

Member	Han Bin Lee	Noorul Azlina	Sagar Sureka	Thomas Mertens
Student Number	A0199835H	A0205302B	A0206443M	A0264284Y

Date: 09/11/2023

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Background

One of the major challenges for students when applying for internships and jobs are the increasingly challenging technical interviews. To prepare for this students practise a variety of coding questions on various platforms and try their best to get comfortable with the difficulty and types of questions. This grind can be frustrating and tedious. Further, this often falls short when the actual interview is run. This is largely because the interviewers also look at the ability of the candidate to articulate their thinking process and share their solution comprehensively. To help in this situation, PeerPrep aims to provide a platform to students where they can practice coding in a lighter interview setting. Students can use the platform to share their thinking processes and solutions with peers. The human interaction will additionally help make the process less frustrating and monotonous.

1.2 PeerPrep

PeerPrep is a platform which works on the motivation of helping students be better prepared for the challenging technical interviews. It provides a wide range of features to help students practise questions of varying difficulties, collaborate and communicate with other users and mutually benefit from each others' knowledge. It provides the functionality for administrators to edit the question database of the system - they can create new questions and edit or delete the existing questions. These functionalities will be elaborated further in the sections below.

The platform is developed primarily using a MERN stack (MongoDB - Express, ReactJS, NodeJS), GitHub as the source code manager, GitHub Issues for project management, GitHub Actions for continuous integration, and Socket.IO for pub-sub messaging.

2. REQUIREMENTS SPECIFICATION

2.1 User Roles

The platform allows for 2 types of users - administrators and regular users:
- Regular users can use the platform to practise coding questions of varying difficulty and collaborate with other users who they match with.
- Administrators can do everything that regular users can do. Additionally, administrators have the permissions to make modifications in the questions database of the platform.

2.2 Features and Services

Service	Features
Frontend	● Responsible for user interactions on the webpages and allows users to access the platform's functionalities in an intuitive setting.
User Service	● Responsible for user profile related workflows like signing up, signing in, changing password, deleting account, logout etc. ● As an additional layer of security, it makes use of JSON web tokens in the authentication flow. ● By default all users are regular users, administrator permission needs to be manually modified in the user database.
Matching Service	● Responsible for connecting together two users who wish to work on a question of the same difficulty. If no user is matched, then the current user can also work independently. ● When the user(s) is ready to work on a question, they are put inside a Room to work on the question. ● Responsible for providing a real time collaborative code editor once the user is inside the Room.
Collaboration Service	● Responsible for allowing two users who are working on a question to share their solutions. ● Users should be able to select their programming language of choice
Question Service	● Responsible for maintaining the store of questions for users to work on. ● Provides create/update/delete functionalities to administrators for modifying the questions store. ● Provides users with the functionality to retrieve a question to work on based on certain difficulty, once users are matched and inside a Room.
Chat Service	● Responsible for facilitating real time text communication between matched users once they are inside a Room.

2.3 Priority Legend

High	Mandatory for the success of the project
Medium	Important for the full user experience
Low	Implement if sufficient time and resources

2.4 Functional Requirements

FUNCTIONAL REQUIREMENTS		
ID	Requirement	Priority
	User Service	
FR1.1	The system should allow users to create an account with username and password.	High
FR1.2	The system should ensure that every account created has a unique username.	High
FR1.3	The system should allow users to log into their accounts by entering their username and password	High
FR1.4	The system should allow users to log out of their account	High
FR1.5	The system should allow users to delete their account	High
FR1.6	The system should allow users to change their password	Medium
	Matching Service	
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
FR2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room	High
FR2.3	If there is a valid match, the system should match the users within 30 seconds.	High
FR2.4	The system should inform the users that no match is available if a match cannot be found in 30 seconds.	High

FR2.5	System should have a code editor where the user can type	High
FR2.6	System should allow the code typed in the editor to be seen by the matched user	High
FR2.7	The system should provide a means for the user to leave a room once matched.	Medium
	Collaboration Service	
FR3.1	System should allow the user to choose a programming language to work on the given question	High
FR3.2	System should have a code editor where the user can type	High
FR3.3	System should allow the code typed in the editor to be seen by the matched user	High
FR3.4	System should allow the user to submit the solution when they have completed the task.	High
	Question Service	
FR4.1	System should store multiple questions which the users can work on	High
FR4.2	System should divide questions based on difficulties - easy/medium/hard	High
FR4.3	System should provide a question randomly based on a given difficulty level	High
FR4.4	System should provide CRUD functionalities for questions to administrator users.	Medium
FR4.5	System should have some example test cases for the user as reference	Medium
	Chat Service	
FR5.1	Matched users should be able to communicate with each other via text	High
FR5.3	System should allow user to close the chat tab whenever they desire	Low

2.5 Non-Functional Requirements

	NON-FUNCTIONAL REQUIREMENTS	
ID	Requirement	Priority
	User Service	
NFR1.1	**Security** - Users' passwords should be hashed and salted before storing in the DB	Medium
NFR1.2	**Security** - Implement an additional layer of security with JSON web tokens	Medium
	Collaboration Service	
NFR2.1	**Performance** - The collaborative editor should update with negligible latency - an update from one user should be reflected in less than 500ms to the other user.	High
	Question Service	
NFR3.1	**Localization** - The questions should cover a variety of different topics like string manipulation, hashing algorithms etc.	Medium
NFR3.2	**Security** - Creation/Deletion of questions can only be done by administrators	Medium
	Chat Service	
NFR4.1	**Performance** - The chat messages should reach the user within 200 ms at least 98% of the time. It should never take more than 300 ms.	High
	System NFRs	
NFR5.1	**Compatibility** - System should run on all the major browsers - Chrome, Firefox, Safari.	High
NFR5.2	**Performance** - System to respond to all requests within 5 seconds	High
NFR5.3	**Security** - Users must be registered to the system before being able to log in.	High

3. Software Development Process

3.1 Team Contributions

Name	Technical Contributions	Minor Contributions	Documentation Contributions
Lee Han Bin	Front End development User Service Question Service System Testing	Collaboration Service Chat Service	Sections 3.1, 4.3, 4.4.2, 4.4.3
Noorul Azlina	Matching Service Chat Service API Testing	System Testing	Sections 4.4.4, 4.4.6
Sagar Sureka	Collaboration Service CI Pipelines API/System Testing Scrum Master Code Quality PM Tools	User Service Matching Service	All sections of the report apart from 4.3
Thomas Mertens		User service System Testing	Section 4.5

3.2 Software Development Life Cycle

Our team used the Agile Framework for this project, as displayed in the image below.
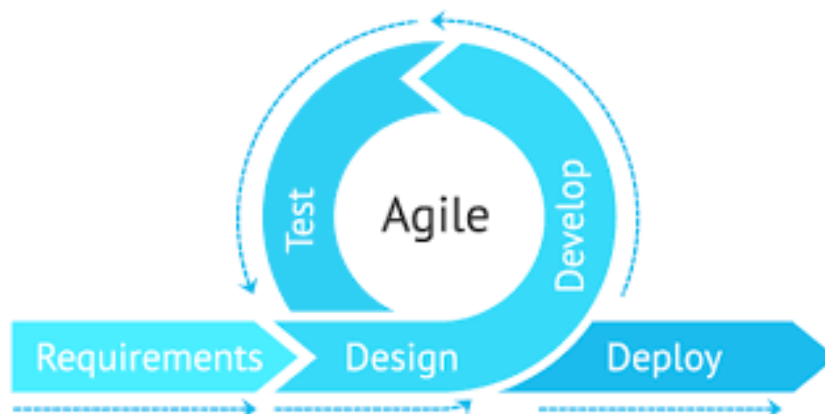

Figure 1: Agile Methodology Example

The length of our sprints were 1 week each. Every Thursday we held a meeting to update the team on each other's progress and estimate the work for the following sprint. At the beginning of all 3 milestones, we discussed and agreed on a timeline with sufficient buffer at the end to meet the deadlines set by the teaching team.

- For the first milestone, our deliverables were primarily User and Matching services. We split the work on the basis of the functional and non-functional requirements. For these services, the requirements were already provided to us so the team spent time on understanding the requirements, setting up the workspace and designing and developing the functionalities.
- For the second milestone, the focus was on implementing the remaining functional services (chat and question service, and collaboration functions in matching service). The team split the work on the basis of the services to be implemented and followed the same format of sprint meetings as milestone 1. Apart from this, Continuous Integration was also set up using GitHub Actions in this milestone to check for compile/build errors.
- For the final milestone, the team focused on documentation, improving the user interface and adding some lower priority functionalities to the system. Further, we also spent time doing end to end testing of the system to identify any bugs, and improving code quality as well. We additionally worked on some user authentication API testing using Mocha and Chai.

3.3 Project Management

We followed important software engineering practices in our development process. GitHub Issues and Projects were used to track progress and requirements of the project. Each feature/service development was done on a feature branch. Pull requests created to merge to the main branch were always reviewed by at least one other team member to ensure good code quality. Our team also met once a week to discuss the progress and the plans for the next sprint.
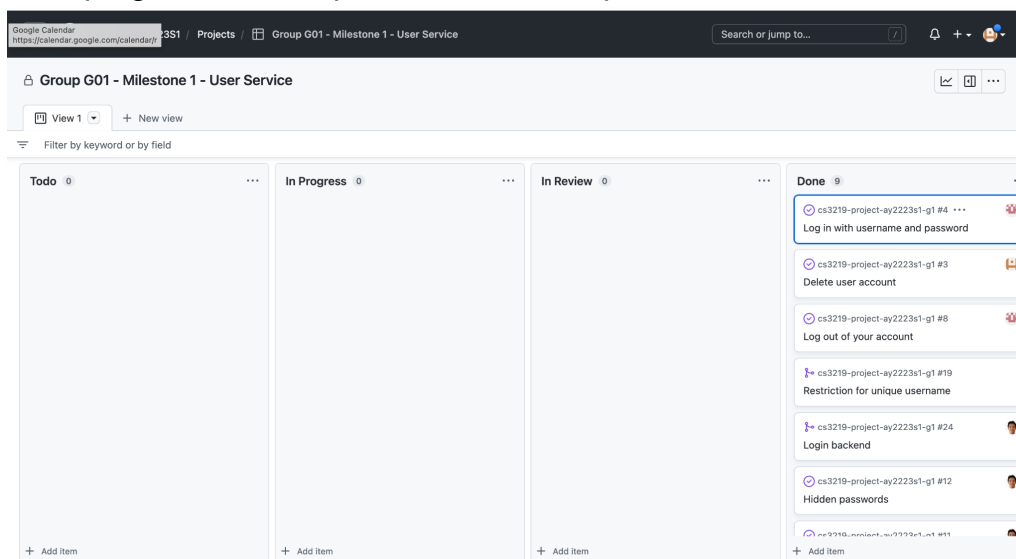

Figure 2: Projects in GitHub - User service

9

3.4 Continuous Integration

GitHub Actions was used for Continuous Integration. Separate workflows were defined for each of the services, which install all the npm packages for the service and build the service. Adding additional steps to the workflow for running tests would be simple. Since we were using local instances of MongoDB, we were unable to add tests to the CI workflow. This aspect of our project has room for improvement by extending it to run the tests, and in the future, setup Continuous Deployment.
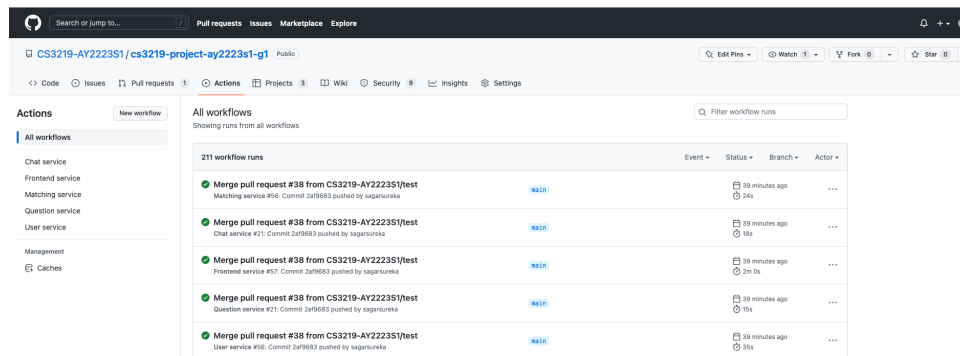

Figure 3: GitHub Actions Workflows

3.5 Testing

For API testing of our code, we used Chai and Mocha to test the authentication APIs of user service. The execution of these tests are not a part of the GitHub Actions workflow since we did not set up a cloud database. To run the tests locally, ensure that MongoDB is running locally. Then after navigating to the user service directory, run the command `npm run test`. It will run the test suite consisting of 6 test cases.

Figure 4: Local execution of the test suite

10

4. Application Design

4.1 Tech Stack

Frontend	ReactJS
Backend Services	NodeJS
Database	MongoDB SQLite (for Matching Service)
Pub-Sub Messaging	Socket.IO
API Testing	Chai, Mocha
Continuous Integration	GitHub Actions
Project Management Tools	GitHub Issues

4.2 Overall System Design

As mentioned earlier, PeerPrep has 6 services - Frontend, User service, Question service, Matching service, Collaboration service and Chat service.

The Matching and Collaboration service are implemented into the same backend service as Matching Service. The reason for this is specified in section 4.4.5 - Collaboration Service.

The overall structure of the system is displayed in the architecture diagram below.
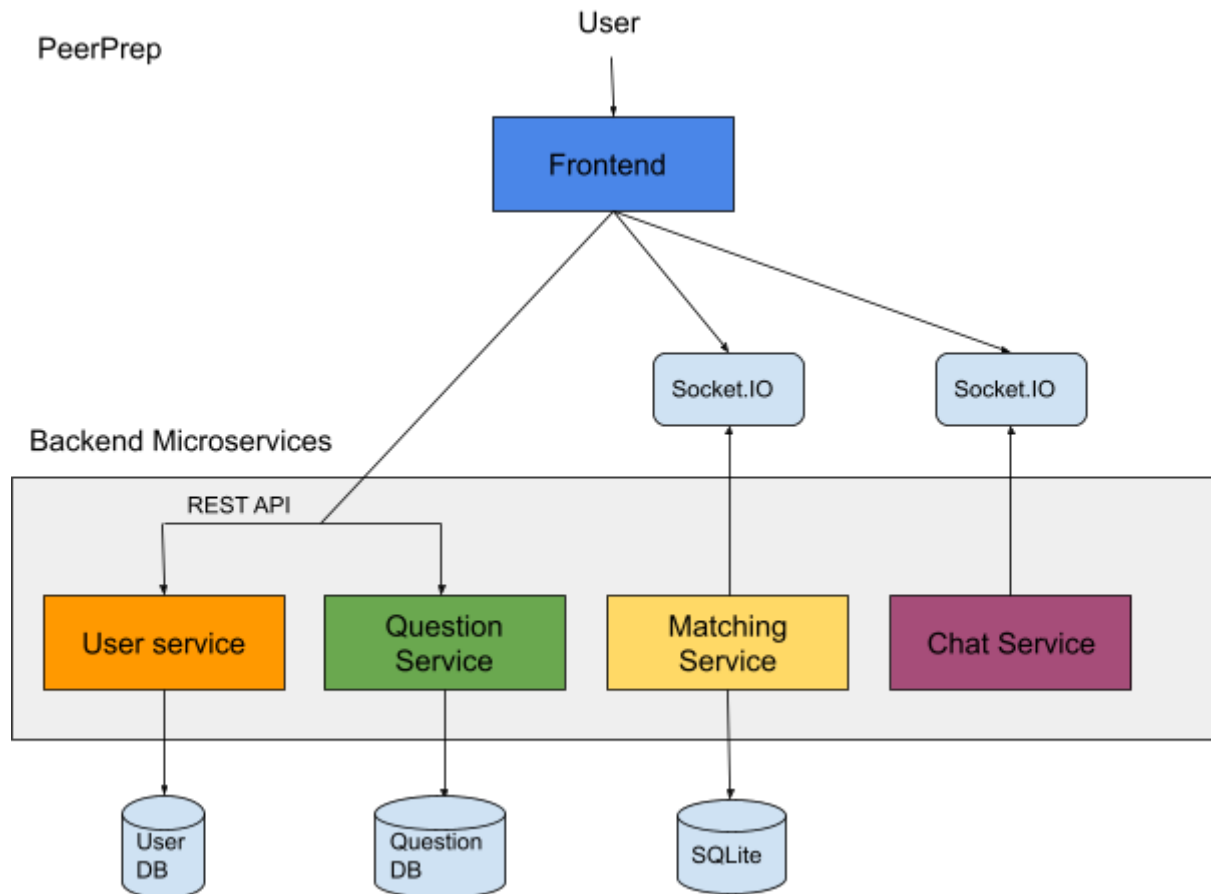
Figure 5: System Architecture Diagram

As shown in the architecture diagram, the Frontend service interacts with the backend services using REST API. The User and Question services interact with instances of MongoDB for their individual data storage needs. The Matching service interacts with SQLite for matching users into a Room in the system. Further, the Matching and Chat services depend on Socket.IO for the collaboration and communication functionalities. The design of PeerPrep is based on a microservices architecture which allows for extensibility, scalability and high availability.

The overall system adheres to the Model-View-Adapter architectural pattern. The specifications for the individual services are specified below. The project development was also done keeping the SOLID Principles in mind. We separated our services according to the Single Responsibility Principle, and did our best to balance responsibilities and functionality in the services. We also largely adhered to the Dependency Inversion Principle as we eliminate tight coupling between each service by separating them. Hence the application has low coupling and high cohesion in its design.

4.3 Frontend Design

For our frontend, we choose to use React as our main frontend framework. The reason why we chose React was because React is remarkably flexible. We could easily create react components based on their given library and their API is very simple to learn. What is even greater is the availability of React Hooks which makes it easier for our developers to have easier control over the components lifecycle. The various lifecycle methods in React include mounting, updating and unmounting. With React Hooks, organising components by lifecycle methods was made extremely easy.

4.3.1 Not Using Redux

Redux is an extremely popular library commonly used with React to manage and centralise application state. We did not use Redux in our application.

We decided not to use Redux as there was not much of a need for state management for our application and based on our set requirements, react hooks would be sufficient. On top of that, Redux makes it harder to reuse components. Since state that is passed between components is stored in the Redux store, components are tightly coupled with the Redux store (which acts as a singleton). Components that are dependent on a global state also means that they are not properly encapsulated, which makes them harder to reuse and test. Often, the Redux store has to be modified if we want to add another instance of the component. We believe that not using the Redux store will make us more mindful about how we design our components to be properly encapsulated and reusable, which will pay off in the long run. For state that has to be shared across different components, we decided that custom Hooks and where necessary, the Context API were sufficient for our needs

4.3.2 User Context (Use Context Hook)

The React context provides data to components no matter how deep they are in the components tree. The context is used to manage global data, e.g. global state, theme, services, user settings and more. As such, for our project, we used the react context to keep track of our User Context.
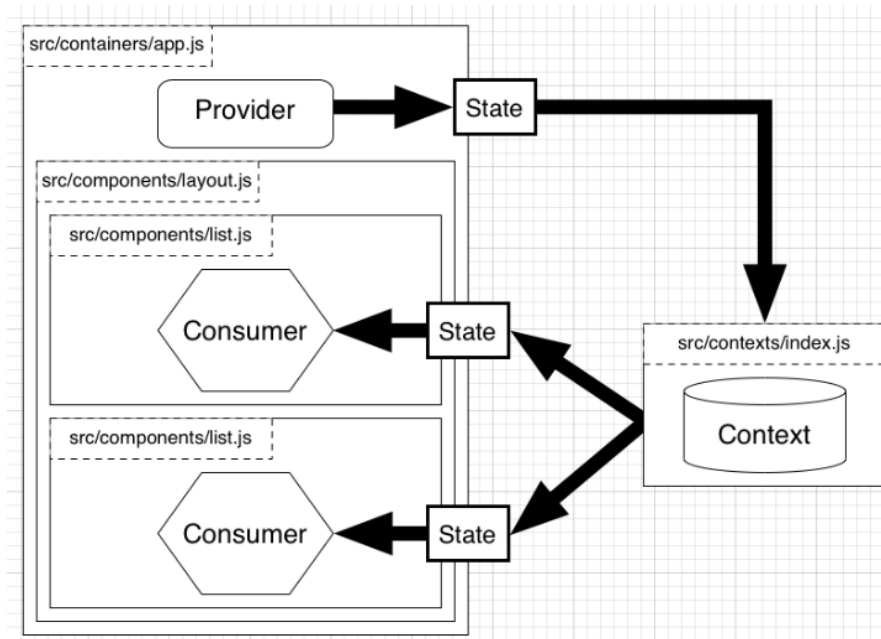
Figure 6: User Context Concept

The main idea of using the context is to allow our components to have access to the global user data and re-render when that global data is changed. This is so we can easily manage when the user is logged in, allowing the user to have access to all of the services, or when the user is logged out and denying users the access to these services. In fact, this context data is also used to differentiate the type of services that are allowed to be provided to the user based on their administrative permissions. Example, only admins are allowed to have access to edit or delete questions in the database.

4.3.3 React Router Dom

React Router is the most popular routing library for React. It allows you to define routes in the same declarative style. It includes three main packages:
- react router, the core package for the router
- react-router-dom, which constraints the DOM bindings for React Router. In other words, the router components for websites
- react-router-native, which contains the React Native bindings for React Router. (Not used in our web development)

React Router DOM enables developers to implement dynamic routing within the web application. Unlike the traditional routing architecture in which the routing is handled in a configuration outside of a running app, React Router DOM facilitates component based routing according to the needs of the app and platform.
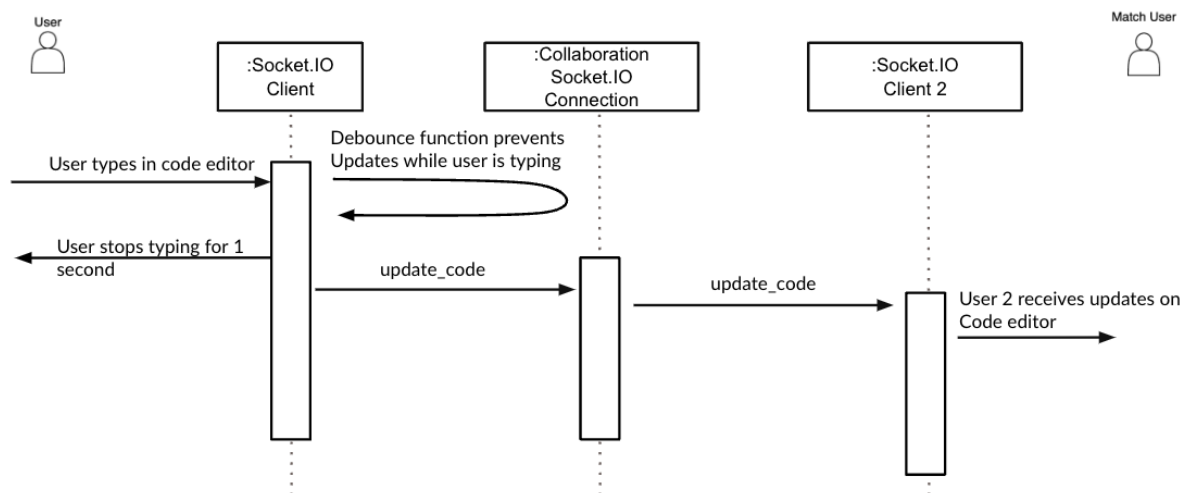
14

4.3.4 Debounce Function

Figure 7: Debounce Function Sequence Diagram

The debounce function forces a function to wait a certain amount of time before running again. The function is built to limit the number of times a function is called and aims to reduce overhead by preventing a function from being called several times in succession. This is built onto the frontend of our collaboration services as we wanted to reduce the overhead each time a user updates the code editor. As such, a debounce function is incorporated to prevent emission to socket.io if the user types continuously in succession. Instead, a batch update of the code editor is emitted once the user pauses for 1 second (1000ms). This provides a better user experience as it reduces the overhead that the socket is receiving and prevents socket.io from crashing.

4.4 Backend Design

4.4.1 Services Directory and File Structure

This section describes the general structure of the backend services in PeerPrep, except for Chat service. The structure of Chat service is fairly simple, with the functionalities implemented in the file `index.js`.

Consider the example of a service by the name of `feature-service`. The entry point of the service would be the file `index.js`. This file mentions the REST API endpoints and the handler methods which are referenced from the file `feature-controller.js` in the controller directory. This layer consists of the main business logic of the feature service. For database related references, the controller layer makes use of the `feature-orm.js` file from the model directory. The `feature-orm.js` file calls the `repository.js` file which directly uses the `feature-model.js`. The feature model contains the specification of the database schema for the feature and the

15

repository.js file executes direct database queries and commands.

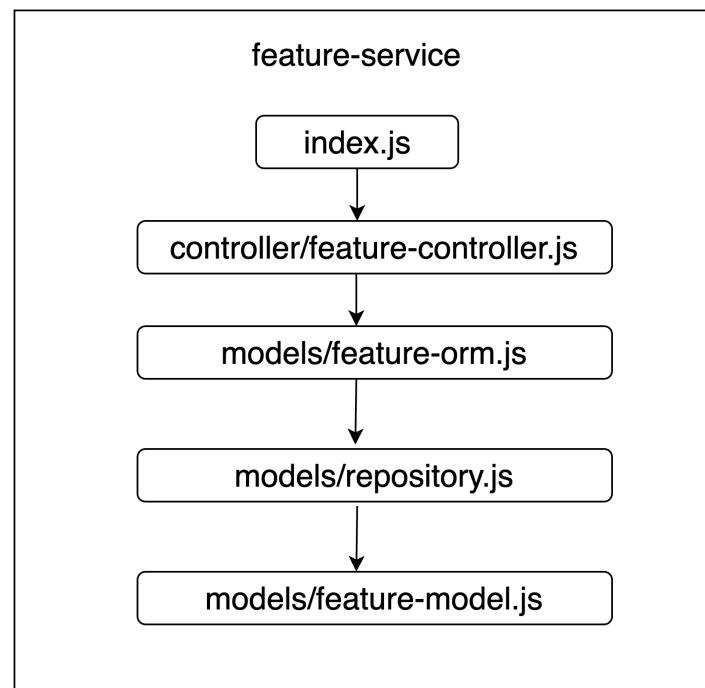This layering of files is described visually in the diagram below.

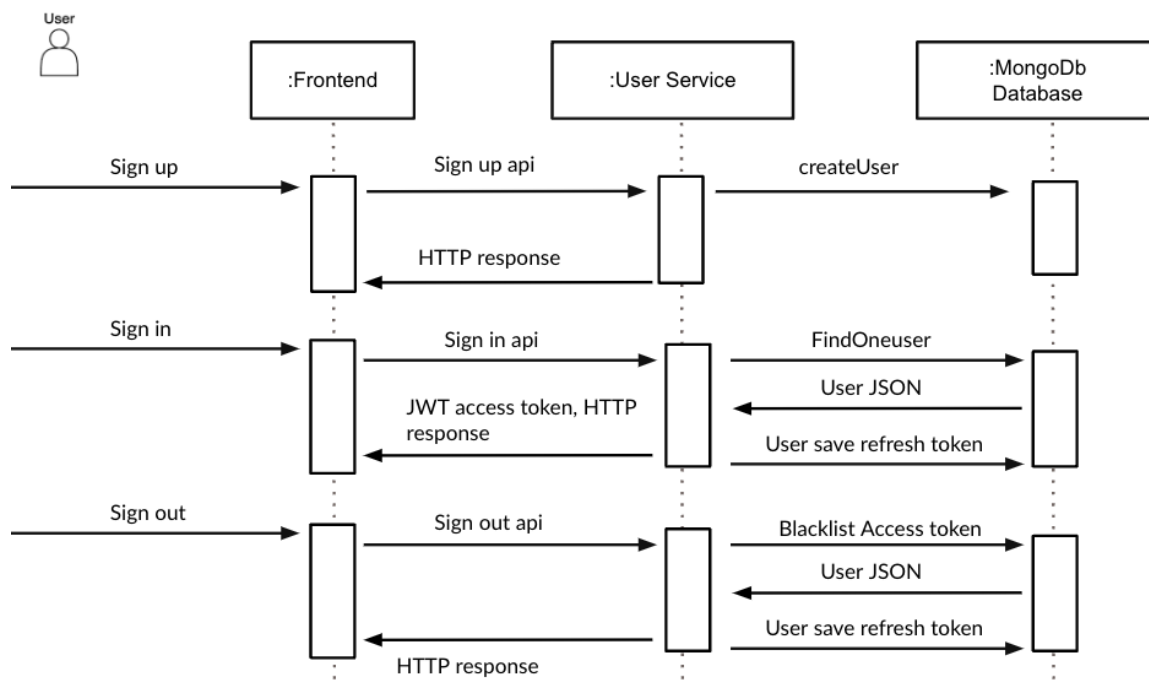Figure 8: Backend services file structure and layers

4.4.2 User Service

Figure 9: User Service Sequence Diagram

The User Service follows a Model-View-Adapter Design Framework. It is responsible for user authentication, authorisation and modification of details. This service communicates with the Frontend using REST APIs. As referenced in the figure below, all communication between the View and the Model is done through the Adapter. The Model and View do not exchange any data directly. The Model represents the data storage of the service and the View is the User Interface. The Adapter accepts the commands or instructions given by the user from the View. Then it does the required operations on data retrieved from the Model, or creates the relevant data. Finally the Adapter returns a response to the View which in turn updates the user.
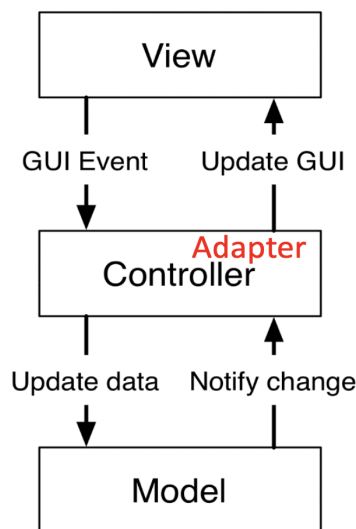
Figure 10: Model-View-Adapter Framework

In the context of the User Service, the Adapter/Controller specifies the handler methods for each of the REST API endpoints. These handler methods make use of the data in the Model to respond to the api requests. The response sent to the View from the Adapter updates the user.

The User Service was enhanced in terms of security by the usage of JSON Web Tokens, and by salting + hashing of the user passwords. The user passwords are salted with the length of the username of the user. With the JWT, we have generated two types of token, one would be the refresh token, which would be stored in the user in the backend to allow users to acquire new access tokens in an event of access token expiration and the other would be the access token itself which is attached to the cookie of HTTP response. The access token will be given an expiration length of an hour and will be stored in the localstorage in the frontend. This access token would then be attached to the header of each HTTP request to ensure proper authentication before calls on the backend api.

17

4.4.3 Question Service

Figure 11: Question Service Sequence Diagram

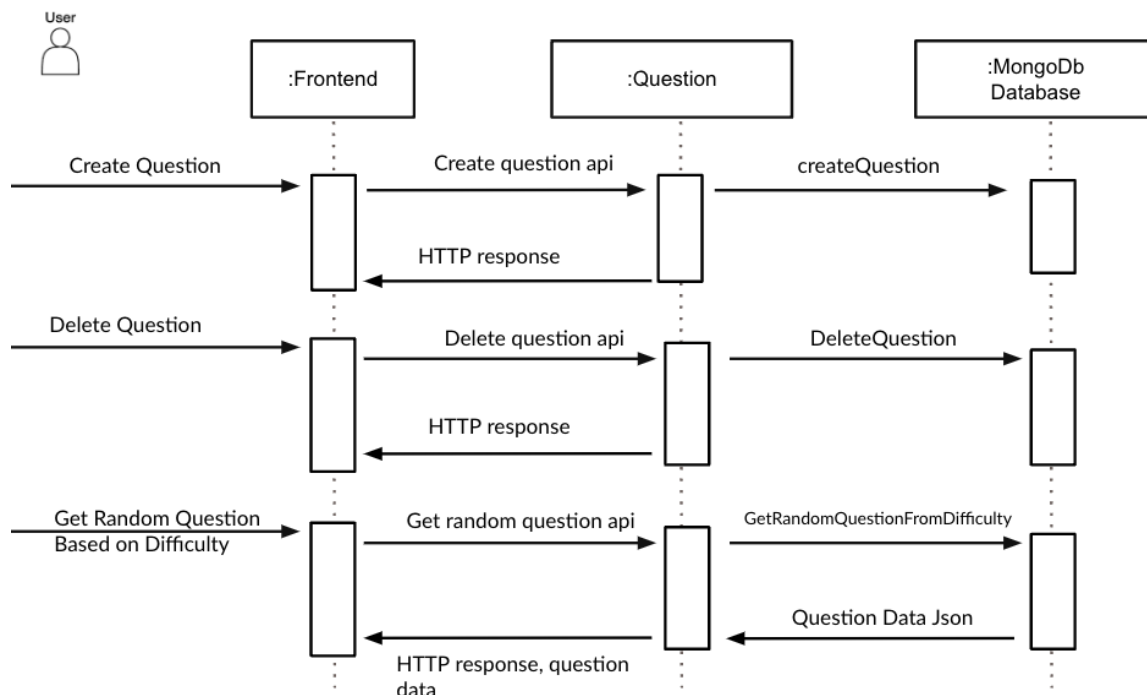The Question Service also follows the same MVA framework as the User Service (refer to Figure 8). This service provides endpoints for admin users of the system to create-read-update-delete documents from the question service database. Again, the handler methods for these CRUD functionality REST APIs are specified in the Adapter. The Adapter uses the Model for operations on the data and passes the response to the Frontend for user updation.

4.4.4 Matching Service

The Matching service follows a similar MVA framework as User and Question service. Additionally, it also works with a Socket.IO connection which facilitates matching of users and collaboration in the code editor.
In Socket.IO, the server (backend) is connected to the client (frontend) with the cors-origin of the server set to the URL of the client which is where the frontend runs. On the client side, a client is created with the URL of the server (backend). Socket.io is chosen for the implementation of matching service as it offers real-time communication based on whenever an event is fired. With a new user selecting their difficulty being treated as an event, two users can be matched almost immediately without any latency.

The design is represented in the figure below which is an extension of Figure 10.
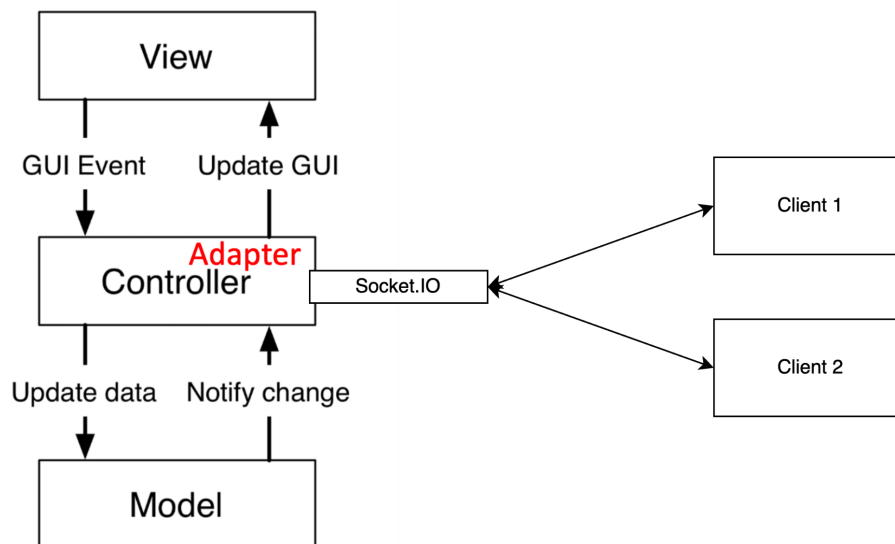
Figure 12: Matching Service Design
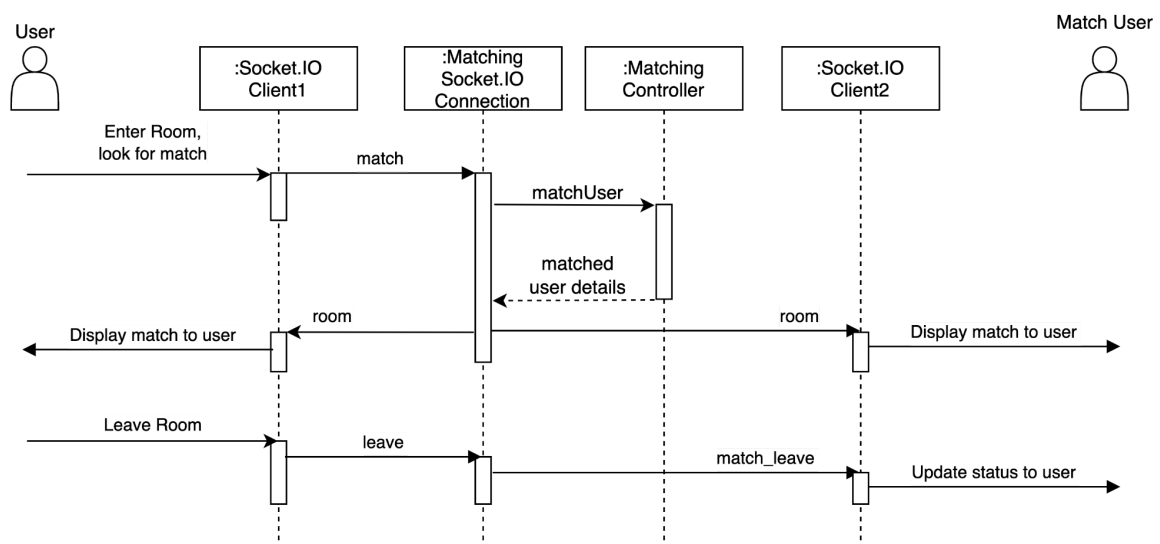
Figure 13: Matching Service Sequence Diagram

As shown in the sequence diagram, whenever a user selects the difficulty that they want to practise, the client sends a socket emission to the server with the difficulty level. If there is another existing user in the database that has chosen the same level of difficulty, the match is successful and the existing user is deleted from the database. The server sends the socket id of both the new user and existing user their respective matches, where the socket id of each user is stored as a hashmap in the server. If there is not an existing user with that difficulty, then this new user is added to the database. Suppose that there is no new user entering for the same difficulty within 30 seconds, then the client sends a socket id to the server to remove this user from the data and no match is found for this user. This is how socket.IO is used for the matching users functionality.

4.4.5 Collaboration Service

The Collaboration Service provides the users with the functionality of real-time collaboration in an editor where they can code the solution of the question. Further, it allows the user to choose from different editor themes, choose a question to solve and different languages in which they want to work on the question. The collaboration service has been implemented as part of the same Socket.IO connection as the matching service. This was done to avoid creating an extra service which would not give any benefit. The Socket.IO connection of the Matching service was utilised for the collaboration service and this prevents having to establish an extra connection between the matched users. Moreover, since once the connection between users is established, the connection remains idle till it has to be terminated. Adding the collaboration to the same connection conserves resources without any demerits.
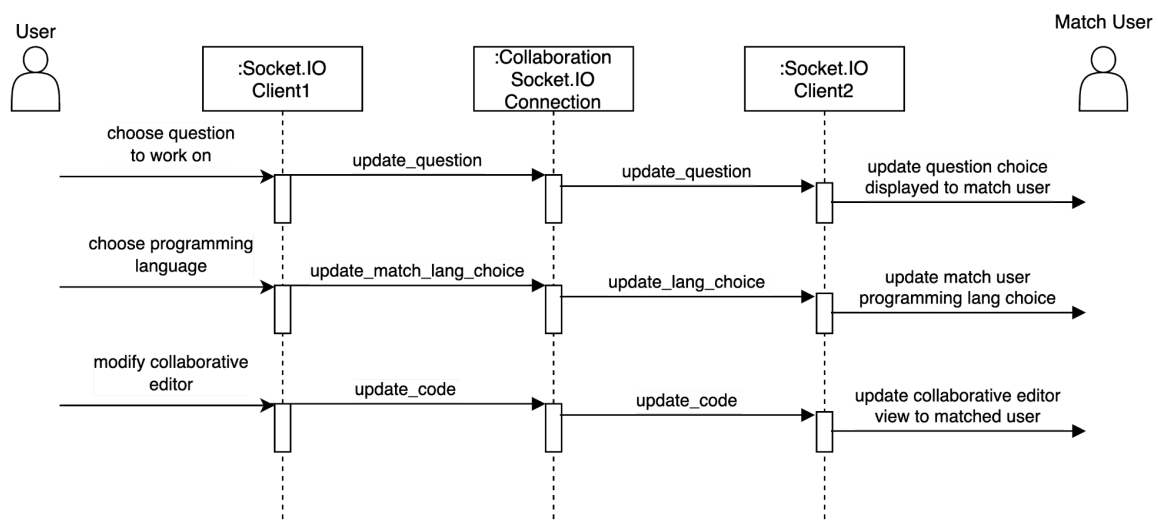
Figure 14: Collaboration Service Sequence Diagram

This sequence diagram above specifies the signals sent and received by the 2 users when working in the collaborative service. All the interactions shown are valid in the other direction as well. For example, in the second interaction - the user chooses a programming language, say Java, as the choice to work on the question. This will trigger a signal to the socket.io connection with the label 'update_match_lang_choice'. The backend of the collaboration service receives this signal and sends a signal to the other client with label `update_lang_choice`. When the client receives this signal, it updates the choice displayed to the user in the frontend.

Socket.IO provides real-time communication between a server and client and is built around the websocket protocol. This protocol is a duplex communication protocol built over a single TCP connection. Since it does not require the HTTP overhead for

20

sending messages, this protocol is much faster and permits for real time communication (accepted to be <500 ms).

4.4.6 Chat Service

The Chat service is simply the Socket.IO connection between clients which facilitates the sending and receiving of text messages for communication between the matched users. When a user sends a message, the client then emits a socket to the server with its matched user's name (receiver) and its message. The server contains a hashMap which maps a user's name to its socket id. The server retrieves the socket id for the receiver then sends the message sent to only that socket id. There is no Model component in the Chat service. Socket.IO was chosen for the implementation of Chat Service because it allows for a real-time bidirectional event based communication. A bidirectional event based communication is what a chat service is and therefore, it is used.

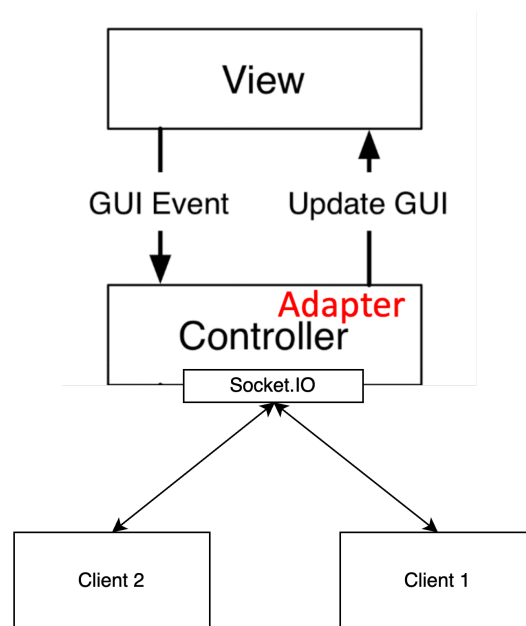Connection of backend and frontend is done in a similar way to that of Matching Service.
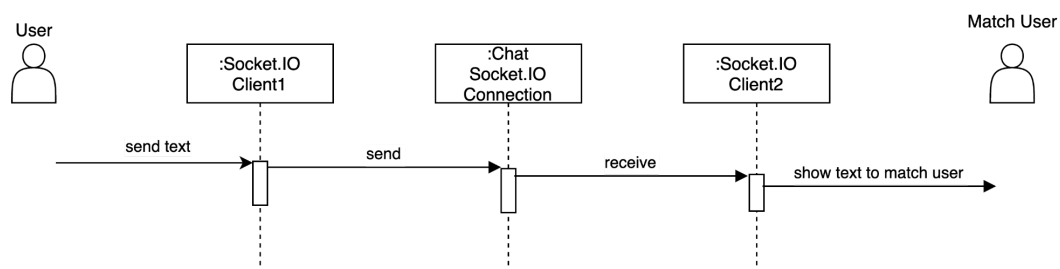

Figure 15: Chat Service Design

Figure 16: Chat Service Sequence Diagram

21

4.5 Database Design

Regarding the User service and Question service we use MongoDB.

The user service database stores the username and password of every user, along with a boolean signifying if the user has admin access. All the fields of this schema are required and the username has to be unique.
For the question service we store a question's index as an identifier shown to the admin users, a title, a degree of difficulty and a question description. All of these are required and the last three attributes have to be strings.

The database schemes for both User and Question services are displayed below.

```
let QuestionModelSchema = new Schema({                      3    let UserModelSchema = new Schema({
    index: {                                                4        username: {
        type: Number,                                       5            type: String,
        required: true,                                     6            required: true,
    },                                                      7            unique: true,
    title: {                                                8        },
        type: String,                                       9        password: {
        required: true,                                    10            type: String,
    },                                                     11            required: true,
    difficulty: {                                          12        },
        type: String,                                      13        admin:{
        required: true,                                    14            type:Boolean,
    },                                                     15            required:true
    question: {                                            16        }
        type: String,                                      17    })
        required: true,                                    18
    }                                                      19
})                                                         20
                                                           21
```

Figure 17: Database schemas for User and Question Models (MongoDB)

The reason why MongoDB is used for those services is because the integration with Node.js is seamless. The JavaScript driver automatically maps JavaScript objects to BSON documents (Binary Javascript Object Notation), so that developers can easily work with their data.

For the matching service we used SQLite to develop the database. SQLite works with relational databases. For relational databases it is very easy to compare different users and check if we can match multiple users. Queries for this can be written in a relatively concise way in comparison to MongoDB. In the matching service database, every user looking for a match is stored with an index, user name, difficulty choice, matching and availability status, requested timestamp and most recent update timestamp. A picture of an example database is shown below.

```
id  name    difficulty  match  available  createdAt                        updatedAt
--  ------  ----------  -----  ---------  -------------------------------  -------------------------------
1   Noorul  1           1      1          2022-09-03 18:15:20.426 +00:00   2022-09-03 18:15:20.426 +00:00
2   noorul  1           1      1          2022-09-03 19:19:16.062 +00:00   2022-09-03 19:19:16.062 +00:00
3   noorul  1           1      1          2022-09-03 19:19:16.070 +00:00   2022-09-03 19:19:16.070 +00:00
4   noorul  1           1      1          2022-09-03 19:40:16.995 +00:00   2022-09-03 19:40:16.995 +00:00
5   noorul  1           1      1          2022-09-03 19:40:17.007 +00:00   2022-09-03 19:40:17.007 +00:00
6   noorul  1           1      1          2022-09-03 19:42:15.100 +00:00   2022-09-03 19:42:15.100 +00:00
7   noorul  1           1      0          2022-09-03 19:42:15.111 +00:00   2022-09-03 19:42:15.111 +00:00
```

Figure 18: Database schema for Matching Service (SQLite)

4.6 User Flow

This section describes a typical successful interaction and code flow for a user using the PeerPrep system. In the sequence diagram below, we assume that there is another user who is waiting to match with a user to work on a question of Medium difficulty, and the current user also enters the system and chooses a Medium difficulty question.
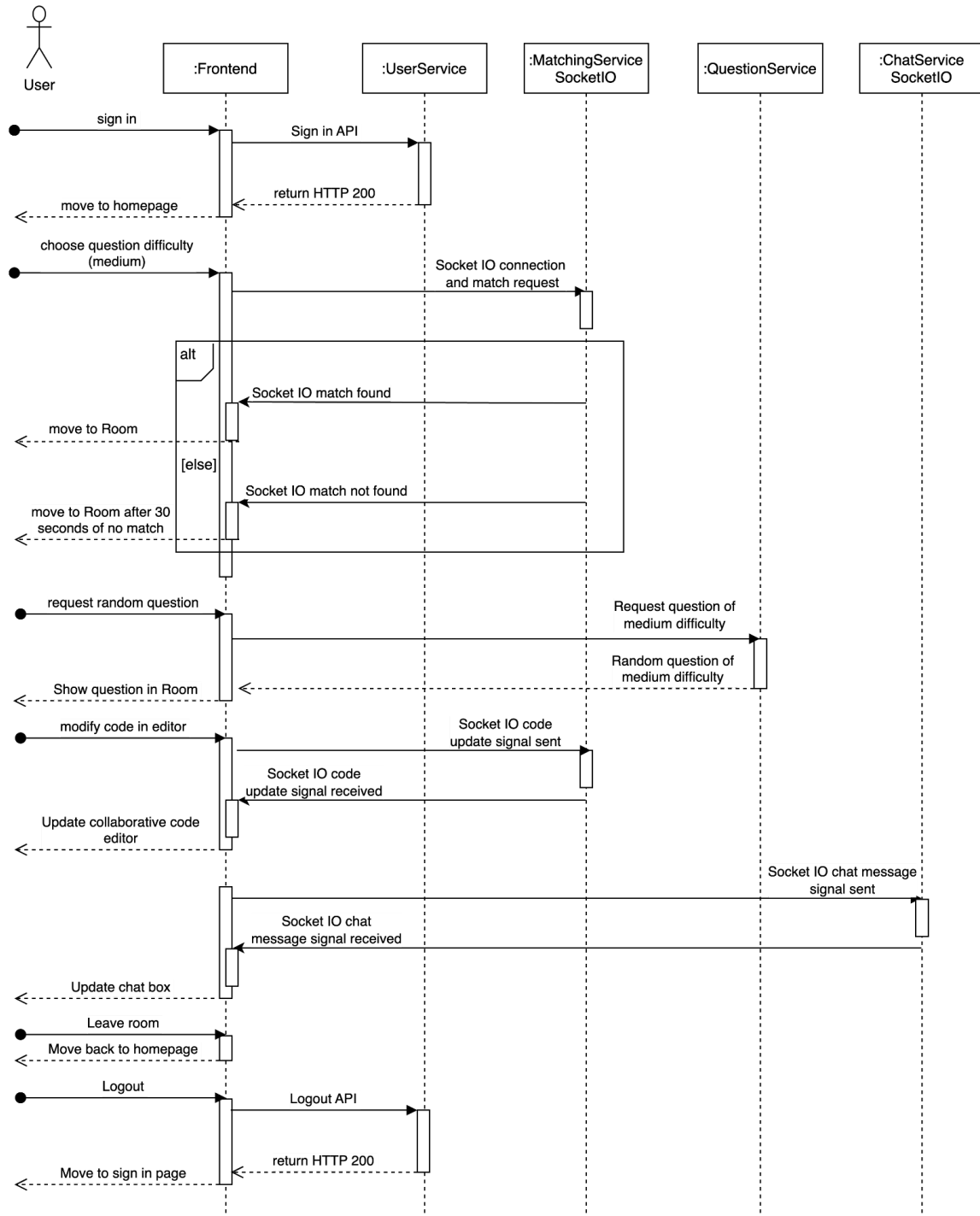

Figure 19: PeerPrep Sequence Diagram

The figure above starts with specifying the user interaction of logging into the PeerPrep. On a successful login, the user is moved to the homepage of the system where they are provided with options between Easy, Medium, and Hard for questions difficulty. In this situation the user chooses Medium and is then moved to a waiting page for a user match. Once the socket.io connection with matching service signals a matched user, or once 30 seconds have elapsed in searching for a match, the user is moved into the Room to work on the coding question. Once in the Room, the user can request a question, and the question service will provide a randomly chosen question of Medium difficulty. Then the user(s) work on the question on the collaborative code editor which is implemented using the same socket.io connection in the matching service. Note that even though this functionality is part of the Collaboration service, since it is implemented in the Matching service, the diagram shows the interaction with the same object.

Next, the user(s) can also make use of the chat service to communicate with each other. Once the users have completed working on the question, they can leave the room and log out of the system. This covers a successful workflow of the system. Screenshots of the application covering the user flow process are attached in the appendix.

5. Extension Considerations

This project was developed with the time and resources constraints of our team. Given more time to work on this project, our team would like to focus on the following features and system enhancements.

- The team would like to work on the deployment of the application such that it would be available on a domain for anyone to try out. For this development, the team would possibly use the Google Cloud Platform. After working on deployment, Continuous Deployment would be added to the practice to make this process automatic and seamless.
- An API gateway for the frontend redirection to the backend services would be implemented. Currently, there are multiple backend service urls stored in the config files of the frontend directory. If this application was scaled further, this would be inconvenient to keep track of. An API Gateway would then handle the responsibility of redirecting the frontend REST API calls to the relevant backend services.
- The team would have liked to add the learning history feature. This feature is very helpful in improving the overall experience of the platform as the students using the system can then keep track of their previous attempts and hence monitor their improvement and growth.
- Additional good-to-have features for the user experience would be on the frontend user interface component. For example, enhancing the collaborative code editor to compile, allowing the code base to have specified Inputs and outputs. This would help users test their code to see if it matches the ideal test cases.
- We spent limited time on API testing. More exhaustive API and unit testing and testing of socket.io services would be beneficial for the project.

These extensions would go a long way in improving the quality and experience of using the system, as well as the quality of the code base from a developer perspective

6. Reflection and Learning Points

The team had many technical and non-technical learning points which we took away from working on this project.

Technical Takeaways:
- Use of JSON Web Token (JWT) in user authentication was an important industry practice which we implemented in our User Service.
- Differentiating between concepts of Authentication and Authorisation.
- Learned about Socket.IO as a bidirectional and low latency communication to facilitate collaboration and communication.
- Familiarity with API testing frameworks like Chai and Mocha.
- Exposure to CI/CD workflows and its benefits.
- Learning how to work with new technologies. Different team members had some experience with different technologies. Everyone got a chance to work on something new and learn a new technology.

Non-Technical Takeaways:
- Practising the role of a Scrum Master and usage of Project Management Tools gave us an increased appreciation of these aspects of the project.
- Having project requirements very well defined is extremely good for the development of the project.
- Capitalising on your team member's skills is vital to the success of the project.

As the team reflects back on the development of this project, we are proud of our effort to meet the requirements and deliver a product which displays this effort. Having said that, we recognise aspects of this project where we could have done better. Primarily:
- The team should have allocated time more consistently over the course of the semester. The workload was higher towards the end as the team was catching up to the goals we had set for ourselves.
- Working on deploying the application is something we would have liked to work on. It would be a great learning opportunity, however we did not plan for that ahead and were eventually unable to work on it.

The team is glad to have worked on PeerPrep and we appreciate all the learnings - good and bad, that we got from this project!

7. Appendix
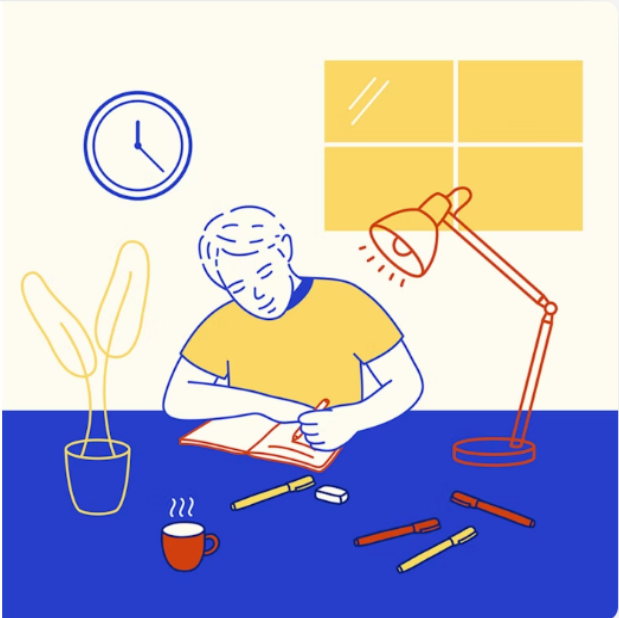
Figure A1: Register or Sign Up Page

Figure A2: Login Page

Group 1

Welcome to Group 1's Peer prep

To Test out matching service, click on Match Tab and select a difficulty

To Test out reset of password, click on Profile Tab and Account Settings

Take note, the Question modification is only allowed for Admins

To Add / Edit / Remove Questions in the question bank, click on the Questions Tab

To Log Out, Click on Profile Tab and Log Out
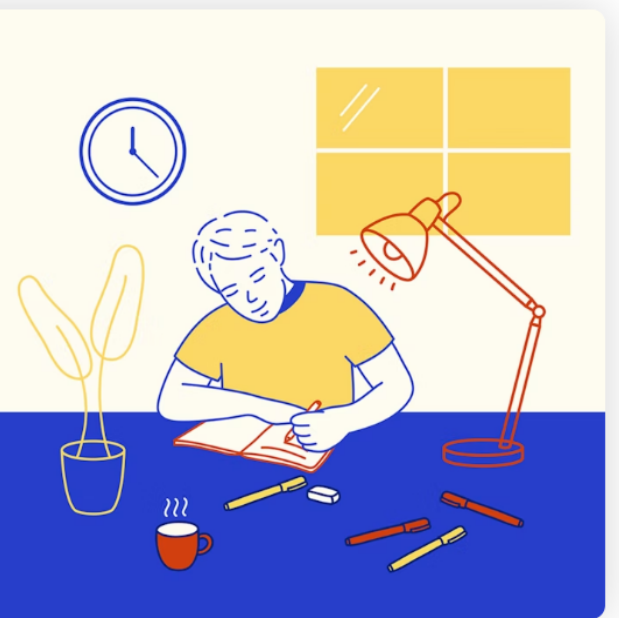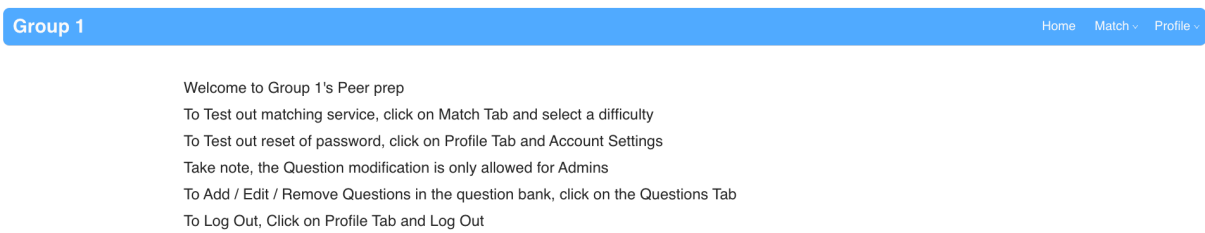
Figure A3: Dashboard Page for regular user
(no questions tab in navigation bar)

Group 1

Welcome to Group 1's Peer prep

To Test out matching service, click on Match Tab and select a difficulty

To Test out reset of password, click on Profile Tab and Account Settings

Take note, the Question modification is only allowed for Admins

To Add / Edit / Remove Questions in the question bank, click on the Questions Tab

To Log Out, Click on Profile Tab and Log Out
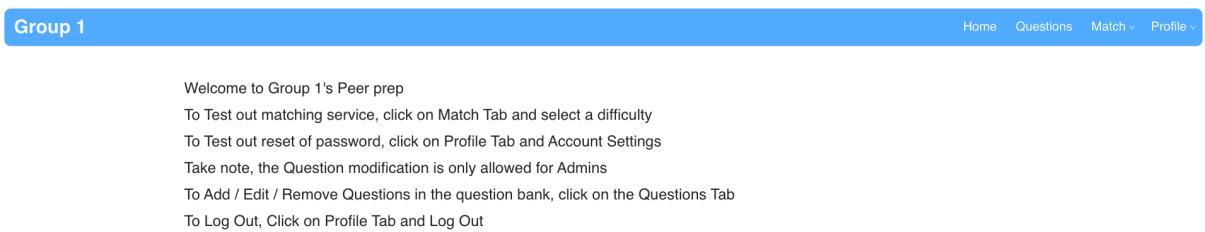
Figure A4: Dashboard Page for admin user
(questions tab in navigation bar)

Searching for Users

Chosen Difficulty: Easy

Figure A5: User waiting for a match

Matched with Username: sagar

Question details is as follows:

Index: 1

Title: Linear Search

Question: Linear Search array for element

Difficulty: Easy

Get a question

JavaScript (Node.js 12.1…

Oceanic Next

```
1   /**
2    * Problem: Binary Search: Search a sorted array for a target value.
3    */
4
5   // Time: O(log n)
6   const binarySearch = (arr, target) => {
7     return binarySearchHelper(arr, target, 0, arr.length - 1);
8   };
9
10  const binarySearchHelper = (arr, target, start, end) => {
11    if (start > end) {
12      return false;
13    }
14    let mid = Math.floor((start + end) / 2);
15    if (arr[mid] === target) {
16      return mid;
17    }
18    if (arr[mid] < target) {
19      return binarySearchHelper(arr, target, mid + 1, end);
20    }
21    if (arr[mid] > target) {
22      return binarySearchHelper(arr, target, start, mid - 1);
23    }
24  };
```

sagar2

hello

sagar
hello to you too

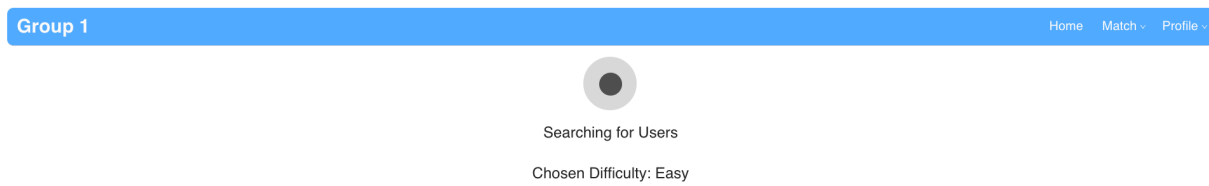Message   SEND

Figure A6: Users matched in a room with a question, a chat box, a collaborative editor, a theme selection box, and a language selection box
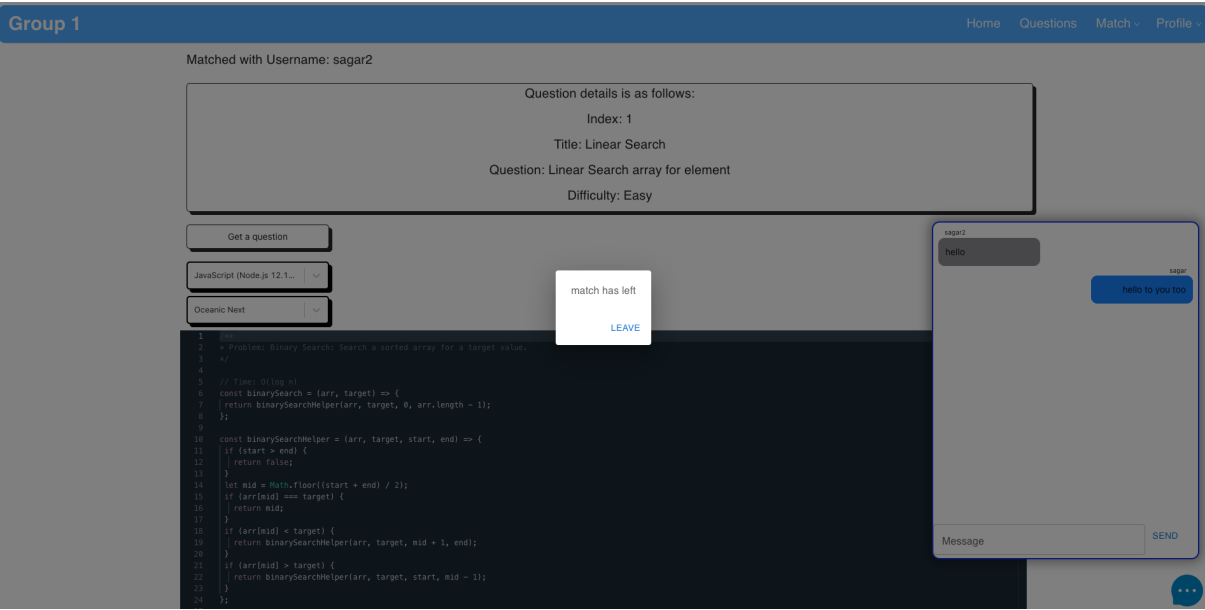
29

Figure A7: Room page after match has left

Figure A8: Admin access for Questions page

Figure A9: Account setting page to change password, delete account , and Profile dropdown for Logout