CS3219 Software Engineering Principles & Patterns

2022/2023 Semester 1

Peer Prep

Team Number: 36

Name	Matric Number
Lim Yuan Bing	A0200869B
Enerio Johannine Salvilla	A0204690J
Lok Ke Wen	A0192344E
Ong Chin Hang	A0200841W

1. Project Details

1.1 Background

Nowadays, a lot of students struggle with preparing for coding technical interviews. Although there is a plethora of platforms on the market such as Leetcode and Hackerrank to help students to practice coding problems, not many provide peer programming functionality, which could be very effective in helping students improve. Therefore, this project aims to help students prepare for their interviews by creating an interview preparation platform and peer matching system where students can find peers to practice whiteboard-style interview questions together. The objectives that our solution will try to solve include making the practice process less tedious and monotonous, improving their communication skills when doing whiteboard-style interview questions, and improving solving random coding interview questions of various difficulties.

In our project description, we were given the general description of the final application as follows:

A student who is keen to prepare for his technical interviews visits the site. He creates an account and then logs in. After logging in, the student selects the question difficulty level he wants to attempt today (easy, medium, or hard). The student then waits until he is matched with another online student who has selected the same difficulty level as him. If he is not successfully matched after 30 seconds, he times out. If he is successfully matched, the student is provided with the question and a free text field in which he is expected to type his solution. This free text field should be updated in near-real time, allowing both the student and his matched peer to collaborate on the provided question. After the students finish working on the question and are ready to end the session, any of them clicks on a "Finish" button which returns each student to the question difficulty selection page. From this page, the student logs out.

With these settings and objectives in mind, we embarked on this project to build out Peer Prep, a collaborative platform for students to practice coding questions together.

1.2 Purpose

The PeerPrep web application is built to aid students in preparing themselves better for technical interviews by allowing them to collaborate with another random user who also has the same goal in mind. By allowing students to collaborate with each other in real time, they are able to leverage each other's strengths and learn from each other. It will also drastically improve their self-confidence and communication skills which are crucial in an interview. To support such experience, we built our system based on core software engineering principles with high scalability, high performance and high maintainability in mind. Peer Prep is optimized for the desktop view but is also usable on mobile and tablet devices. A handful of useful functionalities are implemented to make the application more curated to the user's needs and details can be found in the subsequent sections.

2. Project Requirements Specification

The following sections specify the requirements of our project. Each table corresponds to a microservice or feature domain we wanted to implement and was handled and developed according to the priority legend below.

Table 1: Priority legend for functional/non-functional requirement

Priority Legend	
High	Mandatory for project success
Medium	Important for full user experience
Low	Implement only with sufficient time and resources.

2.1 Functional Requirements

User Service Requirements

Table 2: User service functional & non-functional requirements

S/N	**Functional Requirement**	**Priority**
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
FR1.7	The system should authenticate the user before allowing access to other actions in the system.	Medium
S/N	**Non-Functional Requirement**	**Priority**
NFR1.1	Users' passwords should be hashed and salted before storing in the DB.	Medium
NFR1.2	System should perform password strength validation before sign up/changing password (Passwords must be alphanumeric and of certain length)	Medium

Matching Service Requirements

Table 3: Matching service functional requirements

S/N	Functional Requirement	Priority
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 30s.	High
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
FR2.5	The system should provide a means for the user to leave a room once matched.	Medium
FR2.6	The system should allow users to rejoin the room if they got disconnected unintentionally.	Medium
FR2.7	The system should notify the user if their match leaves the room	Low

Question Service Requirements

Table 4: Question service functional & non-functional requirement

S/N	Functional Requirement	Priority
FR3.1	The system should provide whiteboard-style coding questions of the user's selected difficulty	High
FR3.2	The system should provide questions of at least 3 different types of difficulty (Easy, Medium, Hard)	High
FR3.3	The system should support the insertion of visual aids to help users better understand the question.	Medium
S/N	**Non-Functional Requirement**	**Priority**
NFR3.1	The system should have a question bank indexed by difficulty level	High
NFR3.2	The question bank should be large (>1000) so that the user can practice on many varieties of questions.	High

Collaboration Service Requirements

Table 5: Collaboration service functional & non-functional requirements

S/N	Functional Requirement	Priority
FR4.1	The system should have a commonly shared text editor that both connected users can edit in real time.	High
FR4.2	The system should have a chat box that allows both users to communicate with each other.	High
FR4.3	The system should show the real-time connection status of the user's partner in the same room.	Medium
FR4.4	The system should store the code contributed by both users for history viewing purposes.	Medium
FR4.5	The system should send a sound notification if they receive a chat message from their partner	Low
S/N	**Non-Functional Requirement**	**Priority**
NFR4.1	Latency between both users should be less than 1 second.	Medium

History Service Requirements

Table 6: History service functional & non-functional requirements

S/N	Functional Requirement	Priority
FR5.1	The system should allow users to view all the past questions attempted.	High
FR5.2	The system should allow user to view their final code on the text editor for past attempts	Medium
FR5.3	The system should show the date and time of attempt	Low
FR5.4	The system should allow the user to search for the past attempt question by question name	Low

User Interface Requirements

Table 7: User interface service functional & non-functional requirements

S/N	Functional Requirement	Priority
FR6.1	The system should have a signup and login page, allowing users to access their accounts using username and password.	High
FR6.2	The system should have a display of the code collaboration space and the interaction with other users.	High
FR6.3	In the code collaboration space, the system should be flexible and allow the user to resize the view of the code editor and question area.	High
FR6.4	The system should inform matched users if their partner is still connected and inside the room or has left.	Medium
FR6.5	The system should have a 30-second waiting page when searching for a match	Medium
FR6.6	The code editor should have basic syntax highlighting for better user experience.	Low
S/N	**Non-Functional Requirement**	**Priority**
NFR6.1	The design should be coherent throughout the entire web application and looks aesthetically pleasing	High
NFR6.2	The system is usable on a mobile system such as mobile phone/tablet	Medium

3. Project Plan

3.1 Planned Timeline

Table 8: Planned timeline for each deliverable

Week	Tasks
3	Held team meetings to have a common understanding of the project and come up with the requirements and roles for each member.
4	Come up with functional/non-functional requirements of different features of our app **User service** ● Implement User Service [FR1.1 - FR1.4] **Matching service** ● Implement Matching Service [FR2.1, FR2.2]
5	**User service** ● Implement User Service [FR1.5 - FR1.6, NFR1.6] **Matching service** ● Implement Matching Service [FR2.3, FR2.4]
6	**User service** ● Implement User Service [FR1.7, NFR1.2] **Matching service** ● Implement Matching Service [FR2.5 - FR2.7] **Milestone 1 Demo**
RW	Progress track and update the functionality requirements and timeline
7	**Question Service** ● Implement Question Service [FR3.1-FR3.2, NFR3.1] **Collaboration Service** ● Implement Collaboration Service [FR4.1, FR4.2, FR4.5]
8	**Question Service** ● Implement Question Service [FR3.3, NFR3.2] **Collaboration Service** ● Implement Collaboration Service [FR4.3, FR4.4, NFR4.1]
9	Testing and debug the services implemented **Milestone 2 Demo**

10	**History Service** ● Implement History Service [FR5.1] **CI/CD** ● Setup CI/CD pipeline and deploy the services to Google Cloud
11	**History Service** ● Implement History Service [FR5.2-FR5.4] **CI/CD** ● Setup CI/CD pipeline and deploy the services to Google Cloud **Documentation** ● Work on report ● Update README.md guides on Github repo
12	**Buffer time to:** ● Implement services that's not ready yet ● Test the functionalities locally & deployed
13	**Submissions** ● Finalize report, code submission ● Prepare presentation **Milestone 3 Demo**

3.2 Team Contributions

Table 9: Summary of contributions from each member

Name	Contributions	
	Technical	**Non-Technical**
Ong Chin Hang	● Backend of matching service ● Backend of collab service ● Docker containerization of the services ● Setup & deployment to Google Cloud ● CI/CD scripting	● Writing documentations ● Coordinate communications among the team whenever needed
Lim Yuan Bing	● Backend of user service ● Backend of question service ● Backend of history service ● Unit tests for the backend services developed	● Writing documentations ● Helps liaise communication with teaching team when required
Enerio Johannine Salvilla	● Frontend of login & signup page ● Frontend of profile settings page ● Frontend of history page and related question components ● Debugging user service backend	● Writing documentations ● Provide help to team members whenever needed
Lok Ke Wen	● Overall UI design for frontend ● Setup pre-development environment for frontend (eslint, prettier, husky) ● Frontend of matching page ● Frontend of dashboard page ● Frontend of collaboration page ● Integrate and modify backend code related to the responsible frontend pages ● Docker containerization of the user service	● Writing documentations ● Provide help to team members whenever needed

4. Application Architecture

4.1 Overall Architecture

The diagram below shows a high-level overview of the application's overall architecture and the interaction between the bounded context identified in the system.

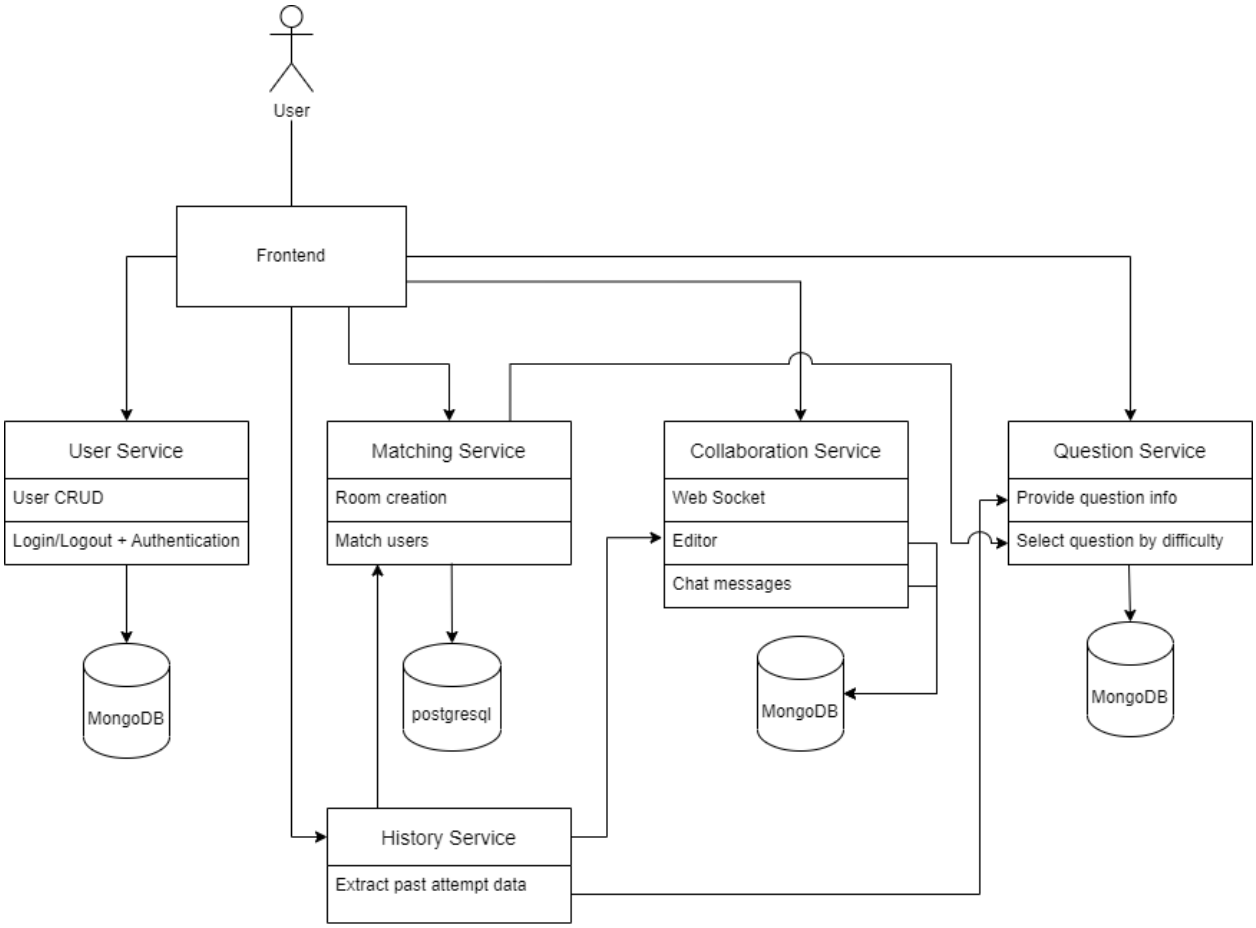

Diagram 1: Overall architecture diagram

From the diagram above, we can see that the user interacts with the system's functionalities through the frontend UI hosted as a web application. By interacting with the web application's UI, users are able to perform a series of actions such as registering an account, searching for a match, joining a room, collaborating with another user to solve the shown coding challenge in real-time and viewing their past attempts after the session. These functionalities are served to the users with the help of the respective backend services. We adopt microservices architecture when designing our entire system to ensure high scalability, maintainability and modularity. As such, each backend service is able to be developed and deployed independently without affecting the other services. A detailed explanation of each of the services and their implementation will be explained in the later part of this report.

4.2 Bounded context diagram

The following is the bounded context diagram to give a more detailed look at how components in the frontend interact with the different microservices in the backend.

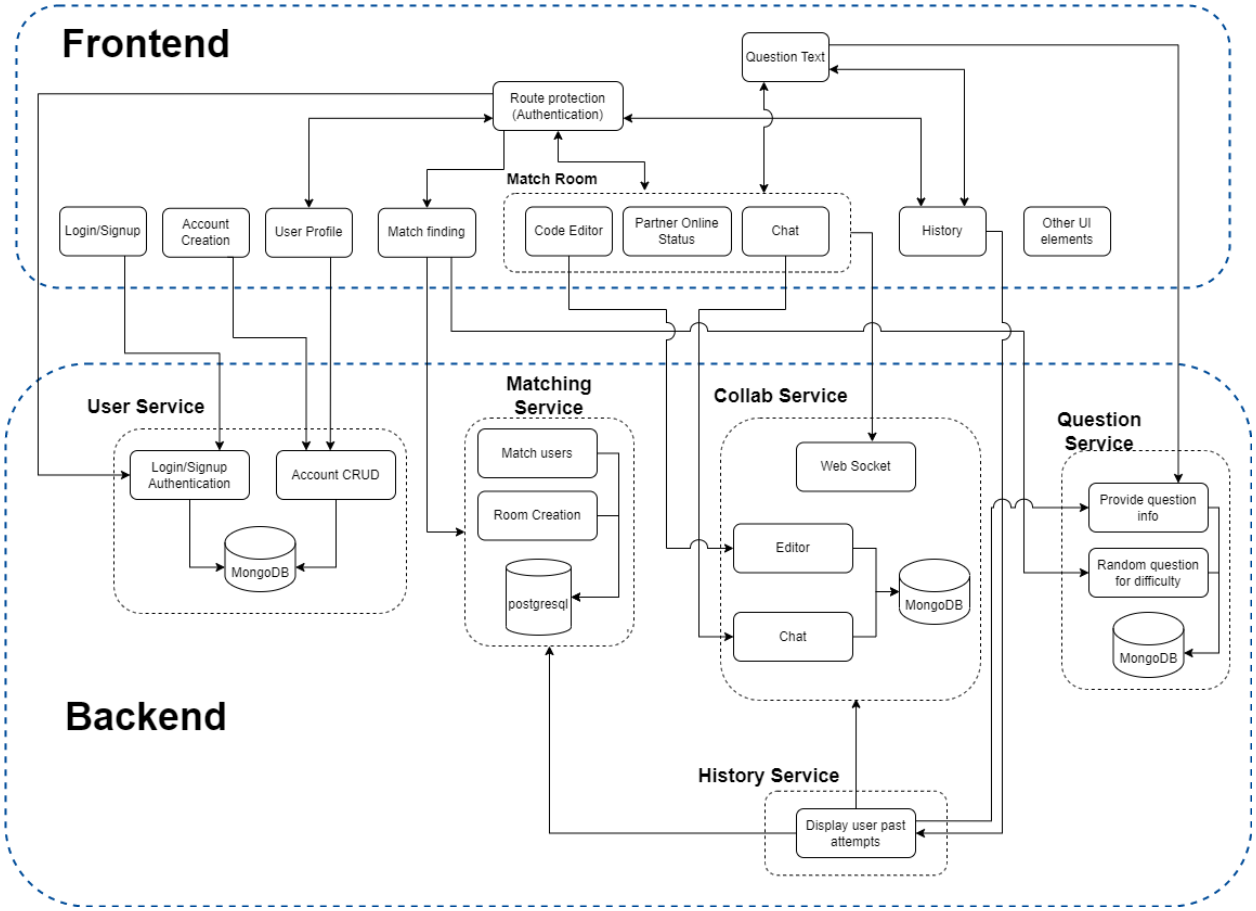

Diagram 2: Bounded context diagram

The bounded context diagram above shows the different components under the frontend and the functionalities provided by our different microservices. The diagram shows how each component in the front end interacts with the respective components in the backend.

The frontend components such as login/signup, profile page and authentication utilize the user service to handle user information. A notable aspect here is that we protect routes that require user login such as user profile, match finding, match room and attempt history by enforcing that the user passes the authentication route first before giving access to those routes. Besides, the match finding uses the matching service, and components under a match room utilize the collaboration service for them to share the same editor and communicate via text chats. The question text that is rendered in the match room and user attempt history is pulled from the question service. Lastly, the history service pulls data from matching, collab and question service and provides the data to the frontend history component.

4.3 Microservices Architecture

4.3.1 User-Service

The user service serves the purpose of allowing users to register an account to access the app [FR1.1-FR1.2]. Besides registering an account, our system allows users to perform other CRUD functions such as updating their account password and deleting their account [FR1.5-FR1.6]. It must also handle user login/logout and authenticate their session before they can use our services [FR1.3-FR1.4, FR1.7]. Our user service is built with these requirements in mind.

Implementation

This backend service is built using Node.js which exposes an HTTP endpoint. We designed the system in a model-controller structure. When the microservice receives HTTP requests, it routes the data to a controller which handles the high-level logic of the system functionality. The `user-controller` will subsequently use functions written and database setup under `user-model` to perform the required actions. This process is summarised in the diagram below.

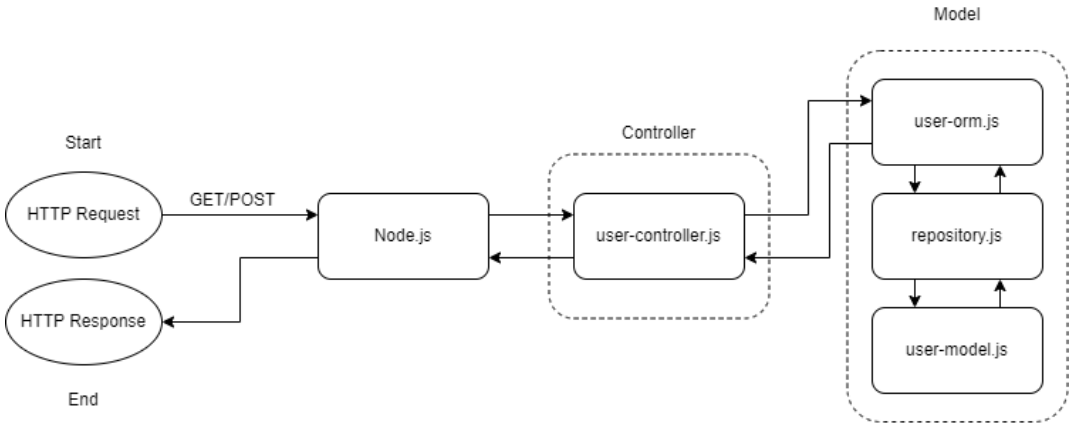

Diagram 3: User service model-controller structure

We utilized MongoDB and set up a user schema with the fields `username`, `password` and `token`. Depending on the request received, the `user-controller` will use functions in `user-orm` to:

- Register an user account
- Login
- Authenticate user
- Update account password
- Delete account
- Logout

Handling registration

To ensure the username used by the user is unique, we set up the MongoDB schema to enforce a unique username field and the system will also run a check to check for username existence before account creation is processed. We also used the npm package `bcrypt` to hash the

passwords before storing them in our database to prevent the original user password from being exposed in our database [NFR1.1].

Authentication

This process is needed to ensure that only authorised users are allowed to use our services and that their account is not being misused by malicious users [FR1.7]. A diagram showing the mechanism of issuing a JWT token during user login is shown below.

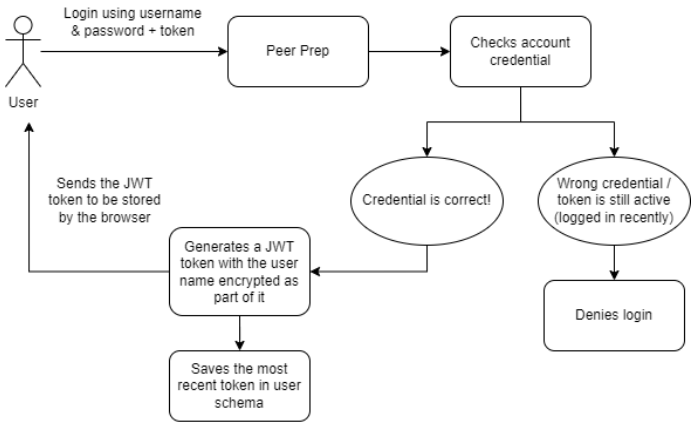

Diagram 4: Login process in user service

From above, we can observe that a successful login occurs when a user provides the correct credential and the token he sent is inactive/does not match the last issued token. The user will then be issued a JWT token signed with a private key stored in our system. When users want to access other parts of our service, they must attach the issued token in their payload to get authenticated before using our services.

The process of authentication is shown in the following diagram.

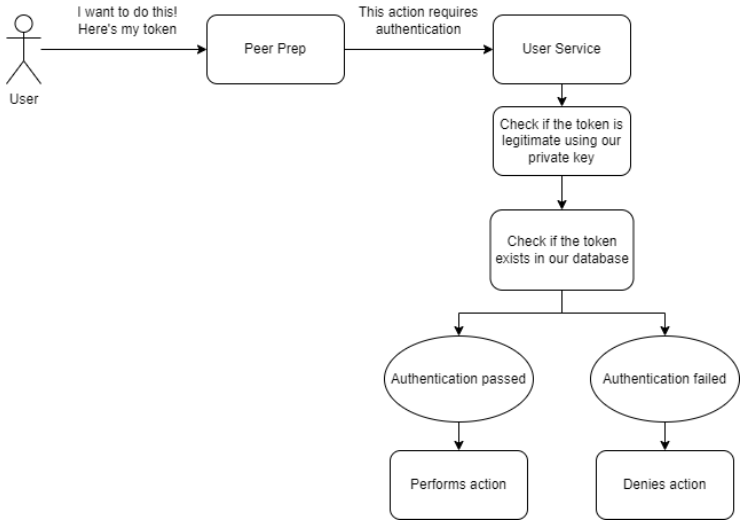

Diagram 5: Issuing JWT token when logging in

Before performing any action that requires authentication, our app will call upon the authentication function of user service to check if the token is valid before granting permission. The authentication functions as a middleware between the function call and the actual execution. In this process, a token will pass the JWT verification if it:

- Decodes correctly using our private key
- Has not expired (JWT token has an alive period set by our system)

To ensure that our system issued the token despite passing our JWT verification, we will run a check to see if the token exists in our database and matches the one the user provides.

Design considerations

The model-controller design employed in this service offers a good separation of concern between program logic and database calls/management. It helps us focus on writing the high-level and low-level logic respectively and is useful when debugging parts of our program. The other notable consideration is that we have decided to use the frontend service to protect other routes by enforcing the authentication before giving access to the routes called. We improved the security of this design by enforcing Cross-Origin Resource Sharing (CORS) to prevent API calls other than from our services.

4.3.2 Matching Service

The matching service is called whenever a user wants to find a match with another user [FR2.1-FR2.3]. The sequence diagram below shows the interaction flow between the user and the matching service.

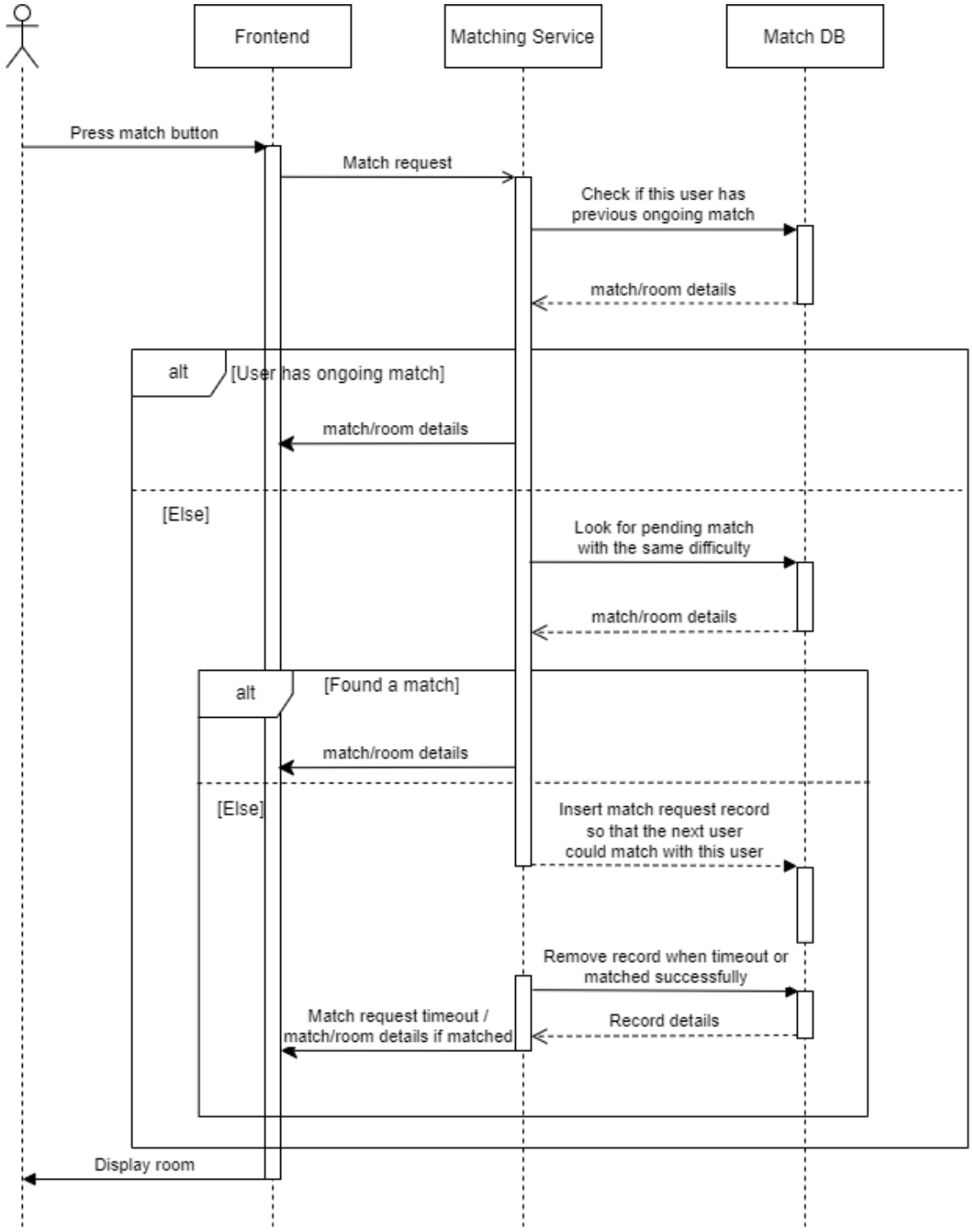

Diagram 6: Sequence diagram of the user matching process

When a user makes a match request, the matching service will first check if this user has any ongoing match with another user. This could happen for a number of reasons such as accidental browser closing and device power failure, and we think that in such situations the user should be able to rejoin the match [FR2.6]. If there is no ongoing match, the matching service looks for pending match requests from other users with the same difficulty and attempts to match them. If there is no match, the matching service will put the match request in the database so that when a new match request comes in, the service will be able to discover the user waiting for a match. There is also a timeout of 30 seconds, after which a request will time out and get invalidated and removed from the database [FR2.4]. If there is a match, the corresponding match record will be retrieved from the database and returned to frontend for further action. The timeout is implemented using the `setTimeout` function of NodeJS.

Design Considerations

A relational database is used to store the match requests to utilize the ACID properties, which are needed so that when two users try to find a match at the same time, the database will be able to serialize the database requests and return a successful match. Two main tables are used: PendingMatch and Match, where the former is for matching purposes and the latter is for storing match records. The past matches are kept tracked so that the information could be used by the history service to discover all past matches a user has.

Besides, we made the decision to allow a disconnected user to rejoin a room so that the match is not abruptly ended if a user has some temporary connection issue for a better user experience [FR2.6]. The users in a room could see the status of another user via the UI, so if one user disconnects, the other user could decide to wait or leave the room to look for another match [FR2.7].

4.3.3 Question Service

This service aims to provide the questions needed for the users to work on upon being matched [FR3.1]. History service will also need the question information when users are viewing their past attempts.

Implementation & considerations

To serve the above function and ensure that this service is always available and doesn't rely on other APIs, we have decided to store our question bank locally and load it onto our database during launch. The other benefit of loading the questions onto our database is that we can use MongoDB's functions to query for the question we are looking for.

To obtain questions for our question bank, we decided to web scrape the Leetcode question API and store all the questions (total: 1869) obtained in a single JSON file [NFR3.1-NFR3.2]. The script used for web scraping can be found on our repo under this link. We discarded the extra info provided by the API during the scraping process and only kept the fields `qid`, `question_title`, `difficulty` & `question_text` as these are the only information needed for our functionality.

The high-level view of the service's execution is shown in the diagram below.

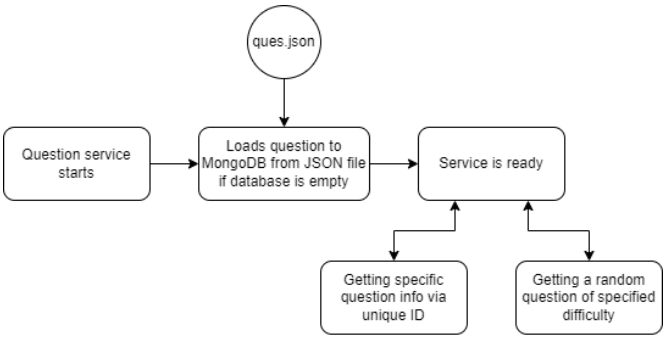

Diagram 7: Activity sequence of question service

When the service is launched, it will check if the connected database is populated by questions already and proceed to import the questions from the `ques.json` file if it's empty. When it is running, the service can be called to obtain the question's data by its unique id or difficulty [FR3.2]. When searching by difficulty, we utilized MongoDB's aggregate function to help us group the questions with similar difficulty and randomly select a question of that difficulty. We also designed this service using the model-controller method discussed in user service to separate high-level logic and lower-level database calls.

4.3.4 Collab Service

The collab service handles the real-time pair programming editor collaboration between users. It uses Socket.IO to connect two users to a room (returned by matching service) and broadcast keystrokes to everyone in the room whenever a user types something in the editor or sends a chat message as shown in the diagram below [FR4.1-FR4.2].

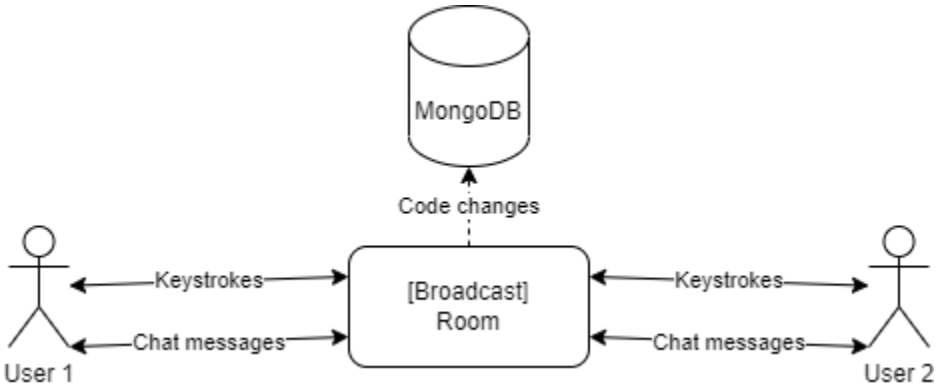

Diagram 8: Interaction diagram for collaboration service

The service also stores the users' code periodically (triggered by frontend) to MongoDB so that the users could retrieve it when viewing histories [FR4.4]. Besides, the service broadcasts a user connection status event to the other user in the room when a user is online, disconnected and leaves the room [FR4.3].

Design Considerations

Socket.IO is used to reduce the latency of the real-time collaboration because it keeps an open TCP connection to the server instead of redoing TCP requests every time when there are changes in the code editor [NFR4.1]. As it is a write-heavy application, MongoDB is chosen because it could handle the write workload well. Besides, MongoDB stores data as documents, which fits our needs as each code is essentially a document on its own.

We also decided to store the code directly in the database instead of storing it in files and linking them to the database. It is because code files are generally small in size, so they are less likely to impact the performance of the database when dealing with large files. By storing the code directly in the database, the latency of reading and writing is reduced as well.

4.3.5 History Service

This service provides the functionality of allowing users to view a record of their past questions attempts and the code they have written for those attempts [FR5.1-FR5.3].

Implementation

This service functions as an intermediary to help the front end history component pull data from the other services, compile and clean up the data before sending it over and displayed. The interactions of history service and other services are shown in the diagram below.

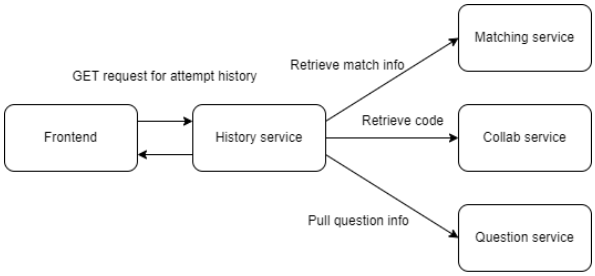

Diagram 9: Interaction diagram of history service

Upon pulling, frontend will receive a JSON message containing a list of all past attempt info as shown below

```
{
    [
        {
            "id":1, // match id, created in sequence
            "roomId":"7290b605-a967-4feb-8eaa-bfae89a48e63",
            "questionId": 123,
            "difficulty":"easy",
            "ongoing":false,
            "createdAt":"2022-09-22T10:56:30.722Z",
            "updatedAt":"2022-09-22T12:14:23.470Z",
            "partner":"someuuid"
        },
    ]
}
```

Diagram 10: Example data provided by History Service

Design consideration

When starting on the design for this service, we realize that we have to pull data from various other services which are not directly under the history service's domain. For instance, a matching service has the responsibility to store match information whereas a collaboration service needs to store the code data to fulfill its requirement. Therefore, we designed a history service to act as an intermediary to help the frontend obtain past attempt data more easily. Data parsing is done in the backend so that the front end can easily obtain and show the past attempts just by making a single API call to this service.

We have also considered other options such as pub-sub messaging for history service to pull and store the data whenever users are done with using other services. A redis caching would also help speed up the data delivery process if the program scales up. However, we didn't have the time and resources to build these functionality suggestions and decided to take this approach to solve the functionality of this service.

4.4 Frontend

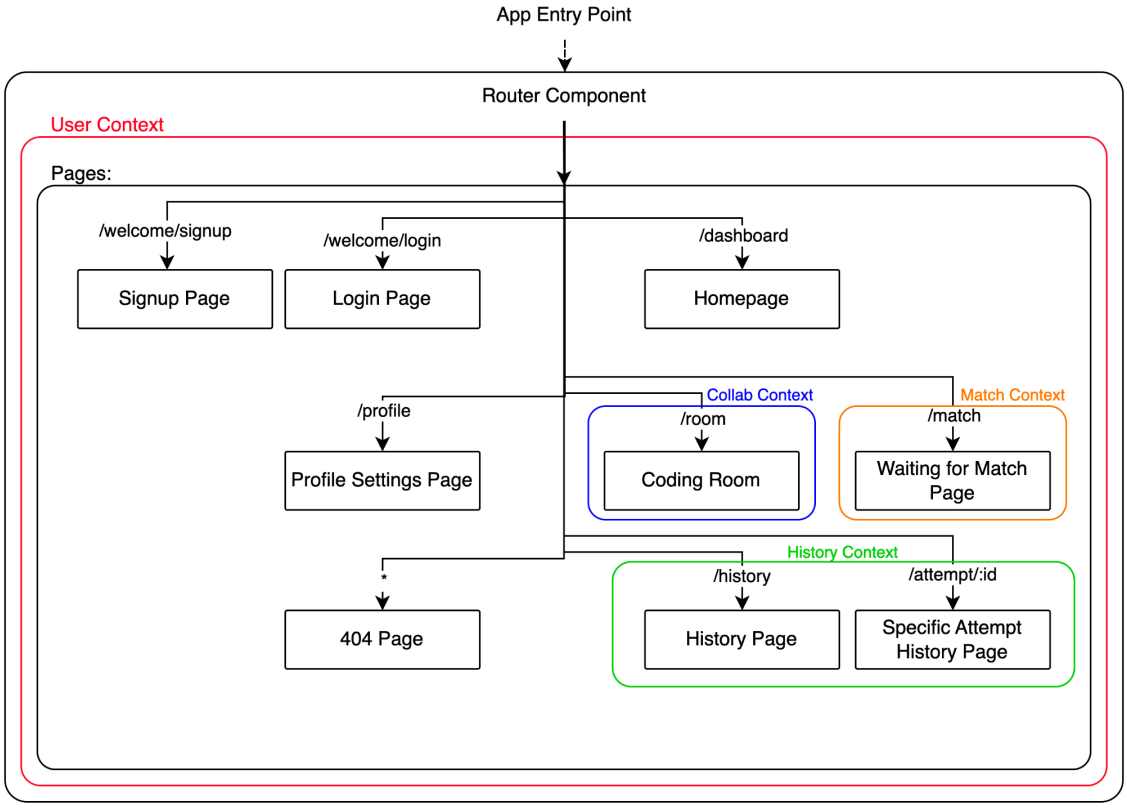

Diagram 11: High-level Frontend Architecture

We have developed our app to be a Single Page Application and used routing to determine the layout of the page to be rendered. With this design, the application is more efficient in transitioning between pages, since there is only a one-time load of the webpage file at the start.

The Router component wraps all other pages and components and is responsible for routing to the correct views depending on the path. It does so with user authentication and route protection, which applies [FR 1.7], by making use of cookies and tokens contained in Local Storage to determine whether a user is logged in and allowed to view a page, otherwise, they are redirected to the login page. This is why the user context holds information related to the current user session, given by a username and a token received from the backend when the user logs in. This access token is then used for making API calls, especially to the user service, in scenarios such as changing one's password, deleting one's account, or logging out.

In the other contexts available, for match, collab and history, React Context is used to share values. This is especially useful in our hierarchical structure with multiple layers, as it does not require us to pass down each individual prop to nested children. With Contexts, all components below the Context Provider easily have access to the data and are able to handle them as needed.

To allow real-time interaction between users, we made use of web socket communication provided by the Socket.IO library. Web socket connection allows the client browser and server to establish a long-term connection rather than the short-term connection in normal Transmission Control Protocol (TCP). The web socket connection is used for exchanging real-time information such as chat messages, code changes, language changes and user online status between the users in the same room. Some other background information is also exchanged to make sure that the web application is working as intended. In our current implementation, we have a web socket used when communicating with the collab service and matching service from the frontend. All web socket connections (with the exception of connection to matching service when a user leaves the room) are established on the layouts (parent) component and passed down to the child components that are using the connection. This is to prevent the re-establishment of web socket connection with the backend services whenever a child component re-renders so as to preserve one and only one living connection between frontend and the respective backend services.

4.4.1 Sign-up/Login Pages

New users with no account can sign up for a new account by entering a username and password of their choice on the Sign Up page. Password strength validation was implemented on the frontend such that it checks the password that was inputted using regex, ensuring that it first meets the criteria of being at least 6 characters long and containing at least one each: uppercase letter, lowercase letter, and number. The checking for form completion is also done on the frontend before a user submits to reduce the number of API calls to the backend.

The system responds gracefully whenever there are errors in the user's input by displaying it on a pop-up Dialog component.

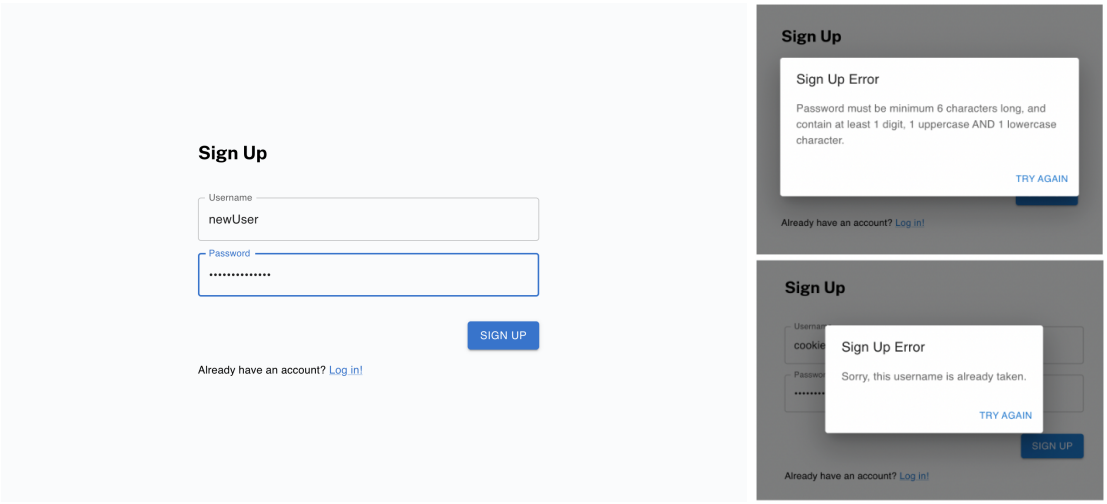

Diagram 12: Sign Up Page

These pages apply [FR1.1-FR1.3] of user service, through the user interface [FR6.1].

4.4.2 Dashboard

Each user will have his/her own dashboard page that will be shown when the user is logged in. On the dashboard page, a find match button and total submissions statistics are shown here for simplicity. The total submission count is retrieved from the matching service database and the count is updated every time a user leaves a match. Detailed information on each submission can be found under the history tab. Other statistics such as the breakdown of attempts by question difficulty could also be added in the future.

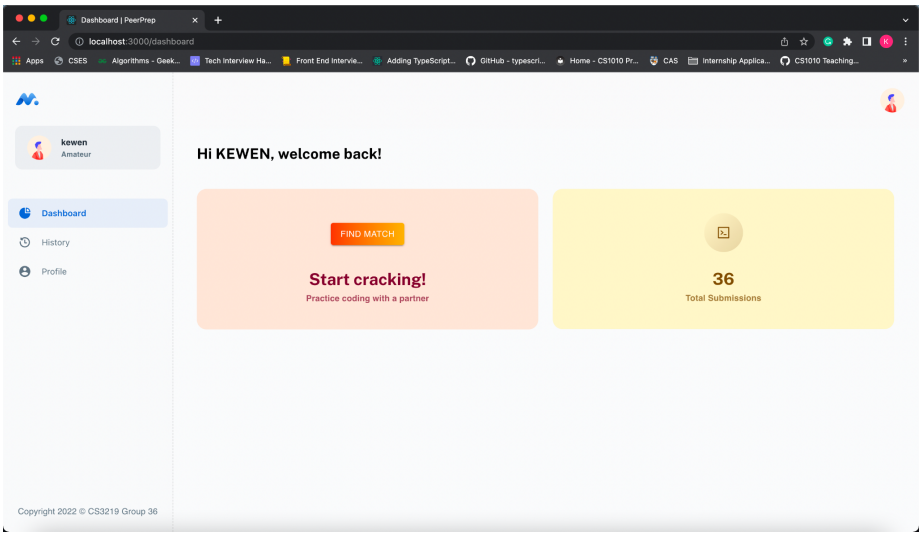

Diagram 13: Dashboard Page

When a user wants to start practicing his/her coding and interview skills, he/she can click on the “FIND MATCH” button located in the first box on the dashboard page.

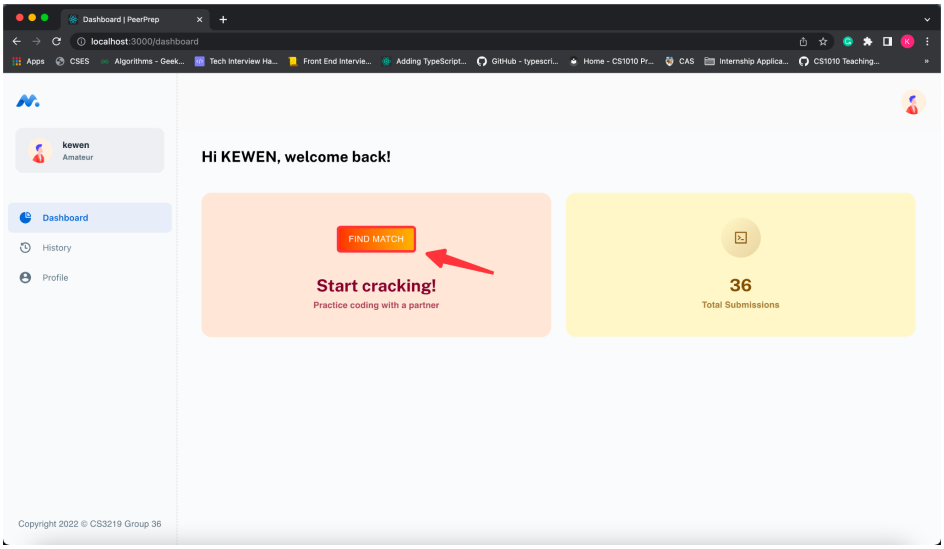

Diagram 14: “FIND MATCH” button on Dashboard Page

After clicking the “FIND MATCH” button, a modal popup will be shown to the user for him/her to select the difficulty of the question he/she wishes to attempt. There is also a disclaimer at the bottom of the modal to tell the user that by joining the match he/she agrees that whatever code is typed in the code editor in the room will be saved. The code is saved for historical reference and code persistent across multiple user sessions due to reasons like network disconnection.

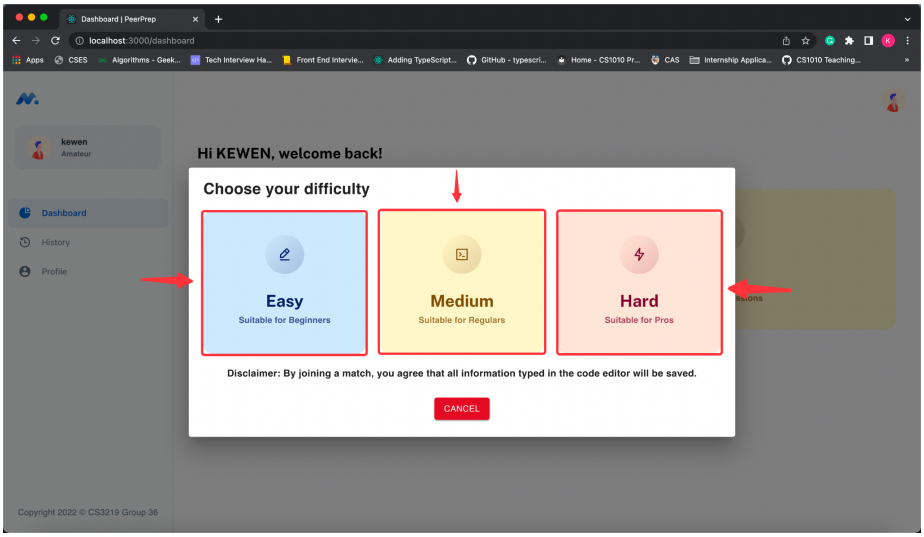

Diagram 15: Popup modal showing difficulty choices

When a difficulty is selected, a confirmation dialog will popup to double confirm that the user is choosing a correct difficulty before proceeding to the matching process. If the user chooses the wrong difficulty, he/she can select the “NO” button and select a new difficulty. Otherwise, clicking “YES” will bring the user to the matching page.

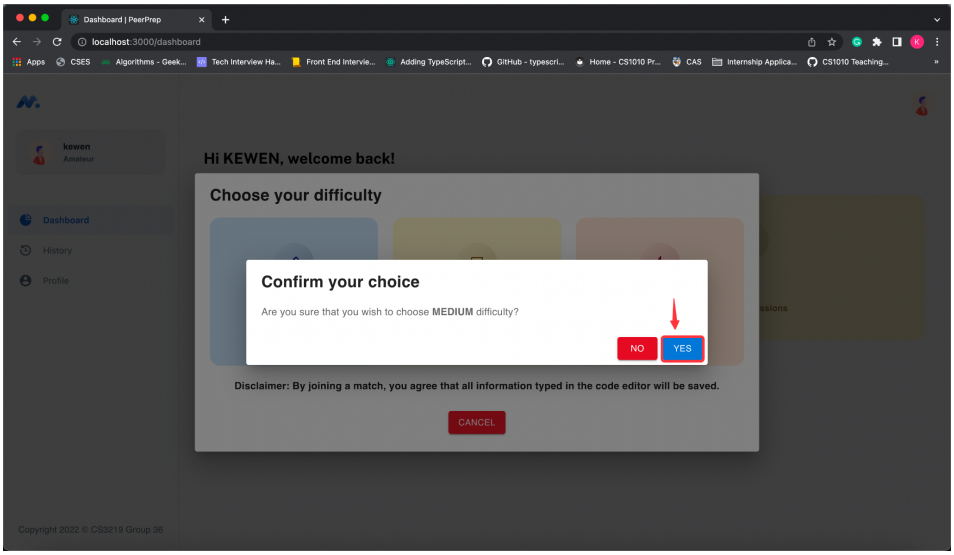

Diagram 16: Difficulty Confirmation Popup Dialog

4.4.3 Matching Page

User will be redirected to the matching page once he/she is logged in and selected his/her question difficulty. The matching page will show a countdown timer of 30 seconds to show the user that matching is in progress while sending a match request to the matching service on the backend side. The page will then listen to the web socket event from the matching service for the match results. If a match succeeds, the user will be redirected to the matched room along with his/her partner. Otherwise, the user will be redirected back to the dashboard 3 seconds after the 30 seconds timer is up [FR6.5].

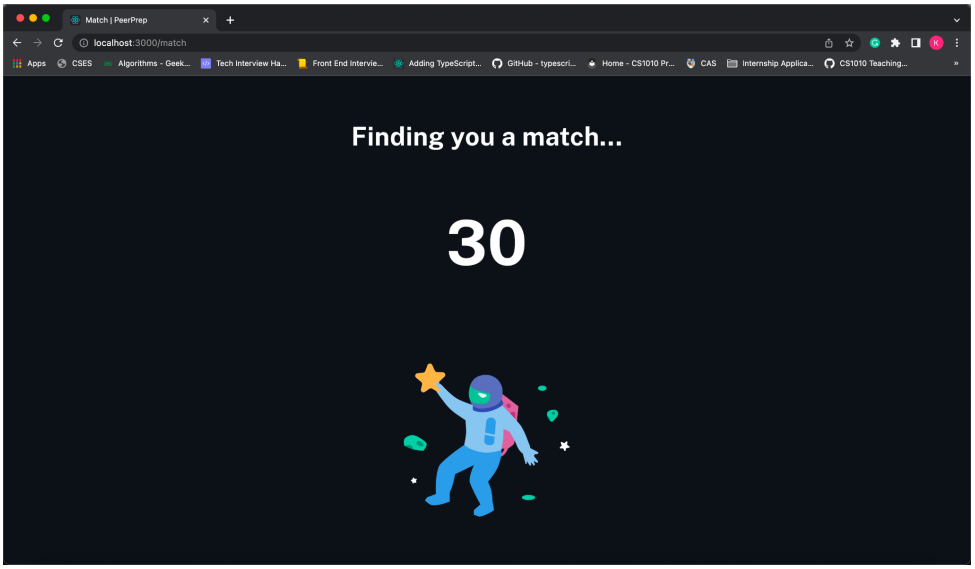

Diagram 17: Matching Page

4.4.4 Room Page

When users are matched, they will be redirected to the room page for collaboration. The room page will display a random question with the specified difficulty, code editor, chat box, user connection status avatar, and a leaving room button. Here, both users can discuss their thoughts on the question through the chat box feature. The chat box provides a real-time communication means for both users to exchange ideas and the user will be notified of a new message through notification sound [FR4.5]. This notification can be muted by clicking the speaker icon on top of the chat box header section. We also allow the users to resize the question and code editor section using the split bar between the sections so that they have the flexibility to allocate more space to the section that they are currently focusing on.

After having a good idea of what the solution is like, both users can start coding on the code editor and see each other's code change in real time. This code is also saved to the database to allow both users to revisit their code in the history tab. Users can also select to code in the language that they prefer by changing the code language. In the current implementation, JavaScript is the only language that has a code suggestion feature, and code suggestion support for other languages will be added in the future.

On the top right corner of the code editor, a user avatar with a status badge is shown. This status badge is used to show the user their partner's connection and whether they remain in the room. If a user disconnects from the room, the status badge will turn grey whereas leaving the room will make the user's avatar disappear [FR6.4]. It lets the user know whether there are any other users in the room. In a case where a user is disconnected from the room, he/she will still be able to join the room as long as he/she does not leave the room.

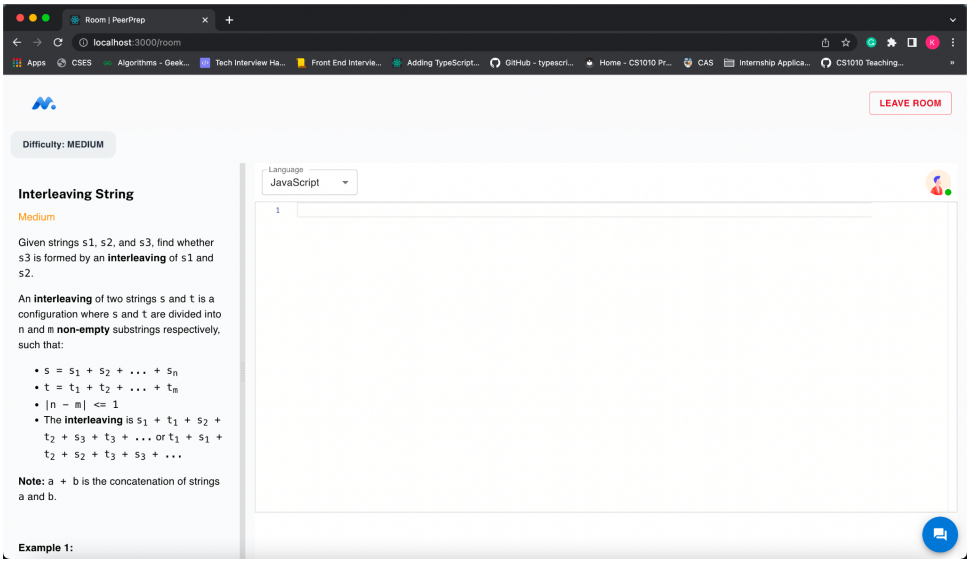

Diagram 18: Room Page

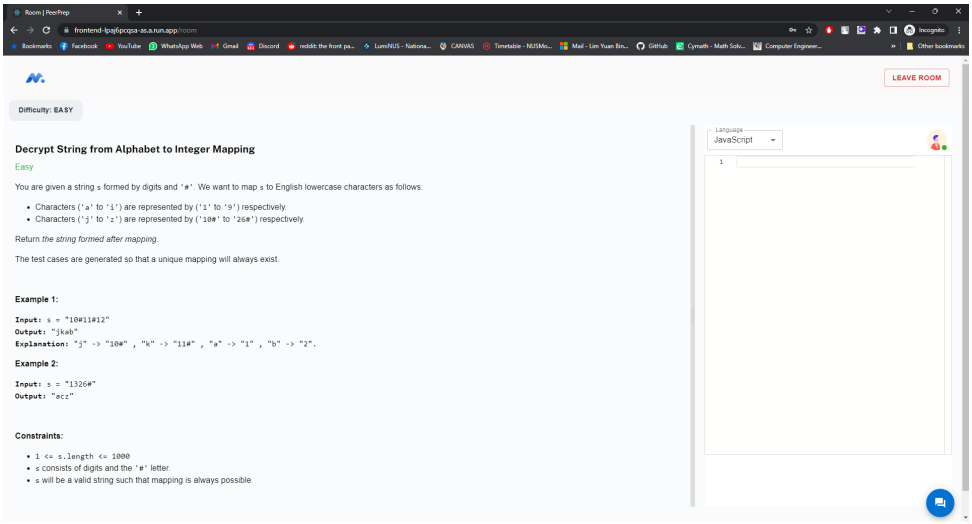

Diagram 19: Resizing the code editor/question space

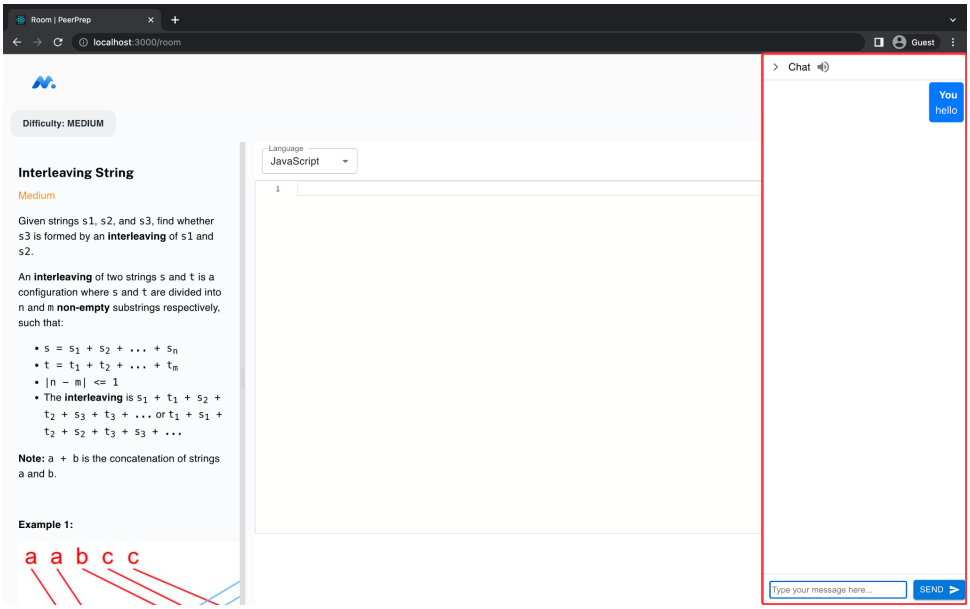

Diagram 20: Chat Box in Room Page

4.4.5 Profile/Settings Page

Here the user can view their basic profile details and perform account-specific actions such as changing their password or deleting their account.

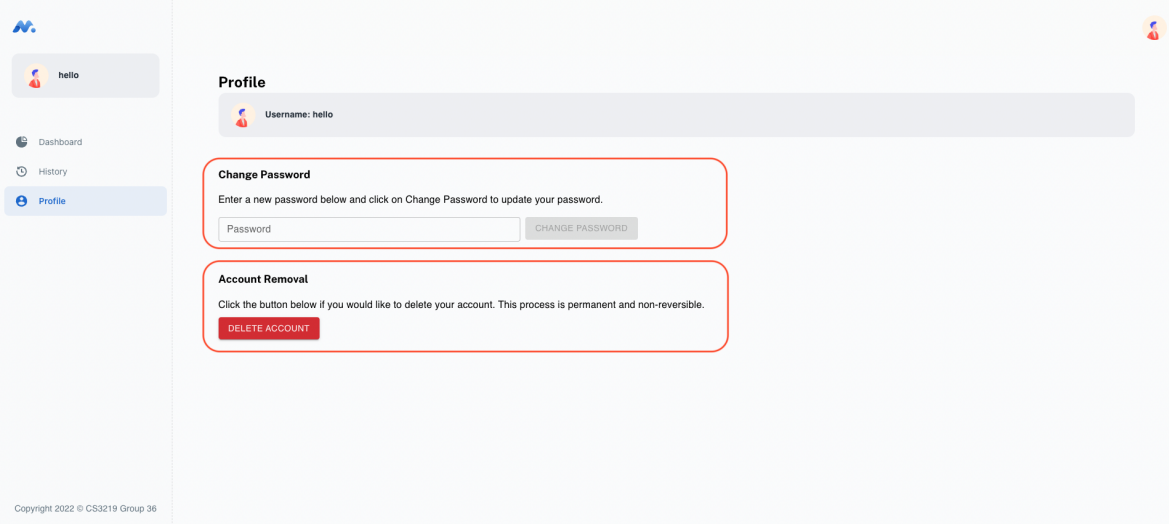

Diagram 21: Profile page

Similar to when a user signs up, password strength validation is also applied for the new password that is inputted by the user here.

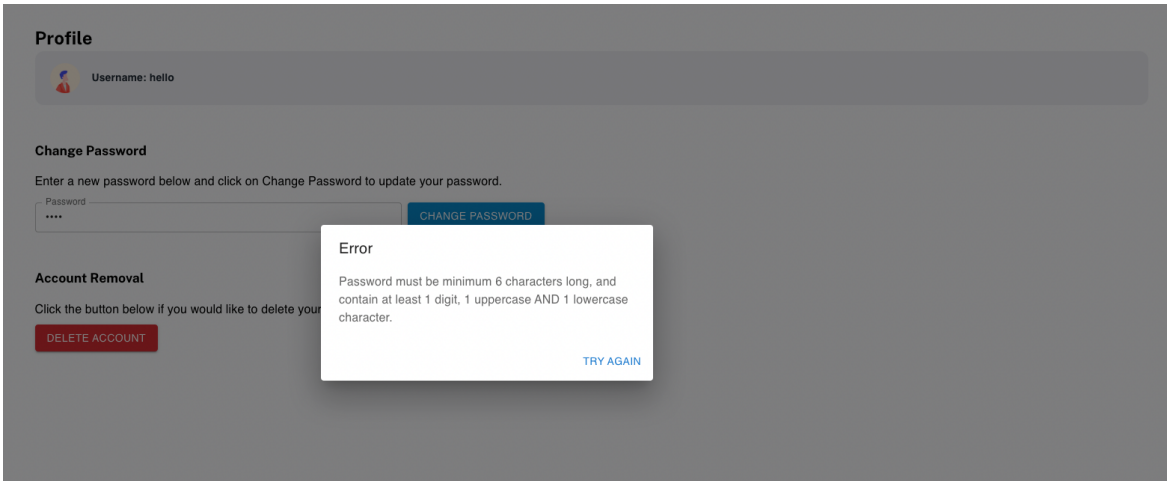

Diagram 22: Password strength validation

To avoid accidental deletions, the Delete Account feature triggers a pop-up confirmation for the user to click on to confirm that they want to delete their account.

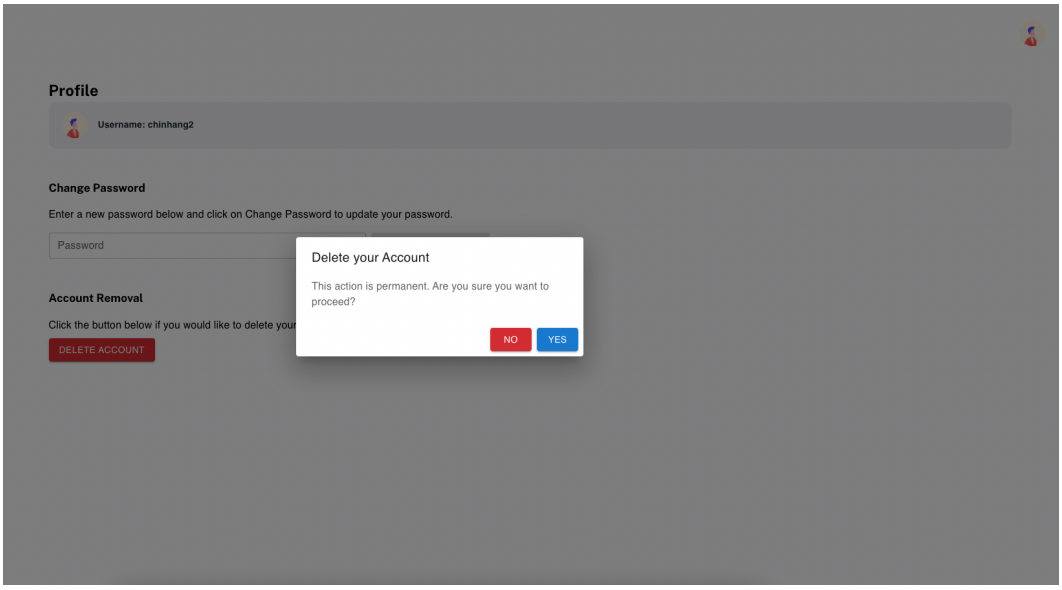

Diagram 23: Delete Confirmation Popup Dialog

These two functionalities make use of the user context as it is where user authentication is most critical. When the user clicks on Change Password or Confirm Delete, it sends an HTTP request to the user service, which first goes through an authentication function that checks on the token currently associated with the user's account, to verify it, before proceeding to send the actual request to the respective functions in the user service backend that handles the account details and carries out the action. This page applies [FRs 1.5-1.6] of user service.

4.4.6 History Page

To view their match and question history, users can go to the History page, which displays a list of past attempts, along with details such as which user they matched with, the programming language used, and the difficulty of the question. From here the user is able to sort the list by any one of these columns and search for a record.

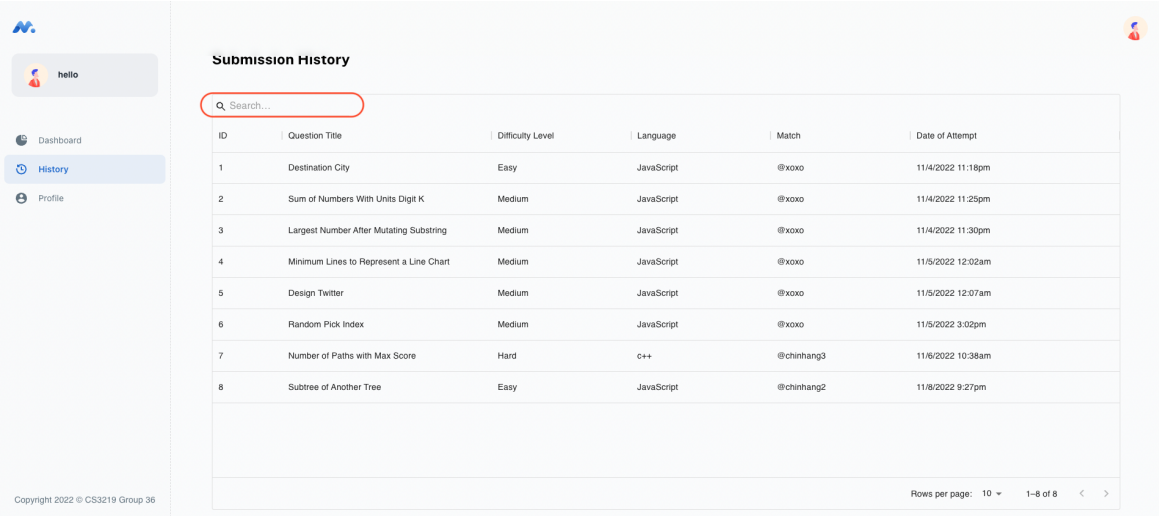

Diagram 24: Past attempts history summary

To view the full details of the attempt, including the question and the code written, the user can click into the respective row which will bring them to another page.

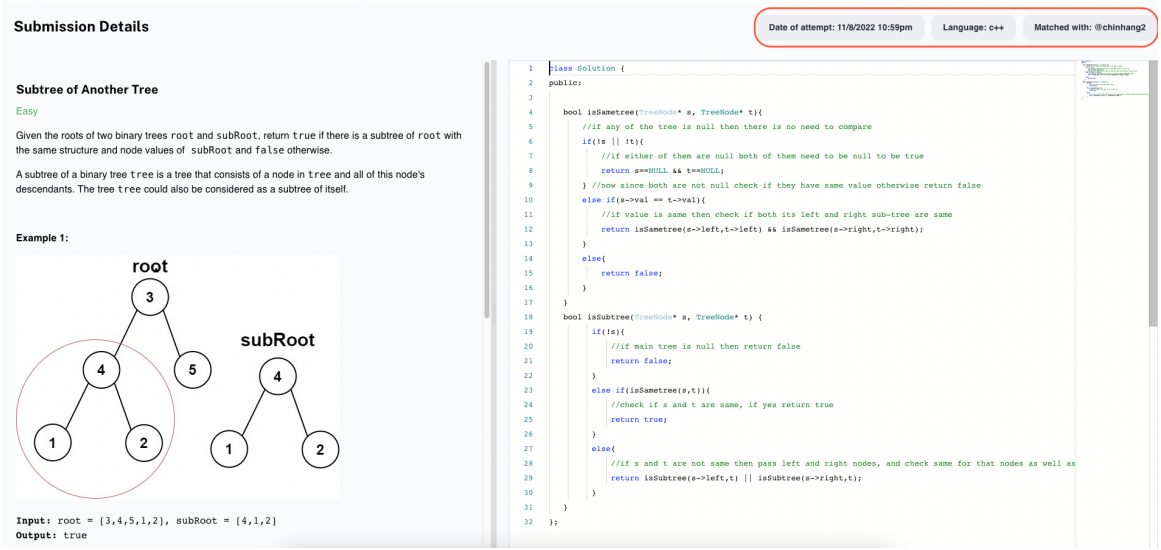

Diagram 25: Example of a past attempt record

This section applies [FR 5.1-5.4] of history service.

4.4.7 Mobile site

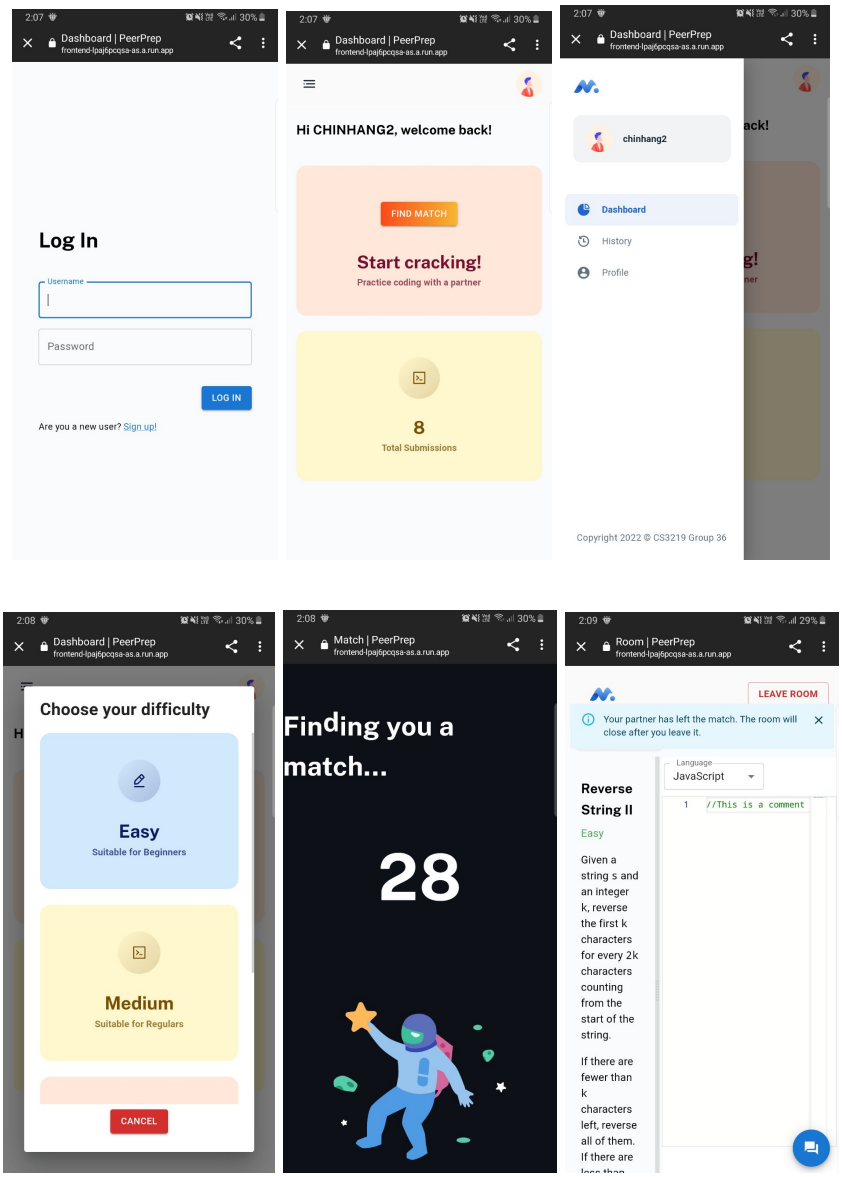

Diagram 26: Our Peer Prep mobile site

As shown from the images above, the user can also access our web application by using their mobile phone [NFR6.2]. Even though it might not be ideal to code on phone, they can still use this feature to access and revise their past attempts on the go.

5. Software Development Process

For this project, our team used an Agile development process. We started off by listing all the features that we wanted to implement and the FRs for each microservice and their associated priorities. We then planned out a timeline, as outlined in Section 3.1, with deadlines for the implementation of each microservice and its subfeatures, as well as other non-feature-related tasks like documentation and deployment. FRs of a higher priority were developed first, and each person took charge of their own feature assignments, according to the roles allocated in Section 3.2.

Thereafter, we conducted weekly meetings to update on our individual progress and sprints to push out features. In each meeting, we will also report on bugs we have found while testing the overall deployed application (in the last month or so), as well as reassign tasks on who would be responsible for fixing what issue.

Before pushing the code to production, the team performed all the testing for individual microservices, both through local testing and Continuous Integration through GitHub Actions. Once this was done and we were ready for a new version, we deployed the project onto Google Cloud Run.

5.1 Tech Stack

Here is a list of frameworks/technology (non-exhaustive) we chose to use to develop our services along with the rationale.

Table 10: Tech stack used in our app

Choice	Rationale
Frontend: **React**	• Fast render due to virtual DOM • Easier learning curve, ramp-up time is shorter • Facilitates creation of highly reusable components
Backend: **Node.js**	• Lightweight and fast processing • Very effective for REST APIs
Database: **MongoDB**	• Multiple deployment options through MongoDB Atlas • Fully scalable • Easy to interact with JSON-like data structure
CI/CD tool: **Github Actions**	• Seamless integration with GitHub repository • Easy to set up and use
Development & Deployment tool: **Docker**	• Manages complexity of multiple microservices and allows them to be developed and launched independently • Easily deployed due to isolated dependencies.
Deployment Platform: **Google Cloud Run**	• Simple and scalable

5.2 Testing Mechanisms

Mocha and Chai are our choices of framework for backend testing since all the services are written in NodeJS. It allows us to set up a mock HTTP server and make HTTP requests to the service and analyse if the output status and content matches our expected values. Some interesting technology we utilized here is `mongodb-memory-server` which helps us set up a temporary MongoDB server to hold our test data needed during the test which will be deleted after the test. To standardize and enable integration testing, docker-compose is used as well to run up ephemeral services that resemble the real services for testing.

As for the frontend side, we made use of the popular JavaScript testing framework, Jest, to test the code written. Unlike the backend services, we decided to employ a higher-level approach for frontend code testing using the snapshot testing technique. The frontend aspect of the system provides a way for the users to interact with the system's functionality. This means that user experience and whatever is visible to the user is crucial. The snapshot testing technique is particularly useful when we want to test for UI changes. During the initial run, the Jest framework will generate a snapshot file of the target UI components and make it a base file for the subsequent run's comparison. When the test is run, the Jest framework will compare the latest UI component render with its previous render result in the snapshot file. If there are changes to the rendered UI component, Jest will throw a test case failed message and that could help us to identify unintended UI changes. We can simply update the snapshot file with the latest UI component renders if it is confirmed to be intended by using the `jest --updateSnapshot` command.

5.3 DevOps

5.3.1 Local development

Docker is used to package each microservice so that the services could be launched independently and easily, whereas Docker Compose is used to orchestrate the service containers and facilitate their communications. Each service has its own docker-compose file so that a service can be developed and tested independently. Bash scripts are written to build and run all the docker-compose files to bring up all the services more easily for the purpose of integration tests. Each service is configured to be in the same docker network and exposes a port so that the services could communicate with each other via a service's IP address and port. A detailed guide on how to run up the development environment is included in the README.md on the project GitHub repository.

5.3.2 Production

A Github Action CI/CD pipeline is set up such that whenever new code is pushed to any branch of the repository, it will run tests for the services that have code/file changes. In other words, the unit tests for services that have no changes will not be run. This is because all microservices are located in the same repository, so it makes sense for services that have no changes to be skipped to reduce the testing and deployment time. An example of the CI/CD graph is shown below.

Diagram 27: Github Actions workflow page

If the branch is the production branch, the deployment job for a service will be run after its unit tests are successful and push the latest Docker Image to Google Artifact Registry. Each service is deployed as a Cloud Run service, and Cloud SQL is configured for services that use relational databases. MongoDB Atlas clusters are set up as well so that services that require MongoDB could connect to and use them. Once the latest Docker image is pushed to the registry, the deployment script will configure Cloud Run to use the latest image, thus achieving continuous deployment. This follows the style of big-bang deployment, which we determined to suit our project best as we only have a small user base.

Cloud SQL is used as it integrates well with the rest of the Google Cloud services and provides managed services that could automatically scale up if the workload increases. By using it, our application could handle heavy workloads without manually configuring and adding the storage.

GitHub Actions was used because of its seamless integration with GitHub repositories, and we chose to deploy to Google Cloud Run because of its simplicity and scalability. Kubernetes is a good option as well, but more maintenance is needed and we think Cloud Run fits our needs well enough. Besides, we also considered the cost of the deployment, and Cloud Run is clearly the winner as it provides free-tier services. It also has the ability to scale down to 0 instances when there is no traffic, so our cost is minimized. Kubernetes, on the other hand, runs on Compute Engine and hence incurs a higher cost, which is not as ideal for the purpose of this project.

6. Suggested Improvements

Due to time constraints, there are some features that we did not manage to implement in time. Nevertheless, there are several enhancements that could improve our application:

Table 11: List of suggested improvements for our app

Suggestions
Allow the users to add each other as friends and allow them to initiate matches with a friend. This would allow the users to match with users that they like or found useful to match with.
Add code compilers so that the users could compile the code in the editor to test out the correctness.
Add a “next question” functionality so that the users could skip a question or work on a new question once the original question is done.
Allow the users to pick a question based on the topic category of the questions they wish to attempt so that they could focus on the topics they are weak at.
Add voice and video call features to the matched room so that users in the same room will have an enhanced interaction experience and easier collaboration.
Add a ranking system where users can be ranked by their effort and attempt to boost the motivation of the users.
Add a discussion forum where users can exchange ideas publicly on the platform.
Implement pub-sub messaging and redis caching for history service to enable a better performance if scaled up

7. Reflections & Learning Points

This project has helped us tremendously in gaining experience in all parts of the Software Development Life Cycle (SDLC). One of the most important aspects that we have learnt is the implementation of microservices architecture and its benefits. In our past projects, we are used to building a single application within the same structure but the exposure to microservices architecture helps us realize the benefits of decoupling components by the functionality they provide. We managed to develop an appreciation for this architecture as we were able to focus on developing each service by itself and the ease of integration between services.

For almost everyone in the group, this is our first time building a web application using React + NodeJS framework and we managed to learn a lot of the skill sets that we took up this module to learn about. We definitely upskilled our technical skill sets through this module with the use of the aforementioned framework and implementing other skill sets from this module into this project such as deployment, CI/CD and system design. It was an impactful experience too to be able to build something functional that can be used by many others to serve a purpose.

Lastly, this project helped each individual in the group develop better communication skills as a developer. From communicating regarding the FRs and NFRs of the project to running agile processes to churn out features, we grew more confident in communicating with one another for a software project development, whether it be on something technical or non-technical.

8. Appendix

Repo link: https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g36