National University of Singapore

CS3219 Software Engineering and Principles

Team 12

Project Report

Source code: https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g12

Member	Student Number
Dione Goh Ru Ying	A0207977M
Tan Ying Hui	A0205260W
Papattarada Apithanangsiri	A0222677Y
Bryan Tan Jing Kai	A0200252A

Content Page

1. Individual Contributions

Tan Ying Hui	**Code** - Frontend UI (Login Form, Change Password Form, Form validation, error handling, feedback messages) for User Service - Implement API for account creation, change password, delete account - Set up database for User Service - Implement API for Question Retrieval - Set up database and question bank for Question Service (with Bryan) - Implement API for updating and retrieval of attempted questions - Set up database for History Service
	Report - General Organisation of Report Wrote and organised the following sections - [2.1, 2.2] Introduction - [4] Requirements Specifications - [5] Software Development Process - [6.1, 6.2, 6.5.3, 6.5.5, 6.5.6] Application Design - [7] Future Enhancements - [8] Learning Points - Appendix A: Use Cases
Dione	Code - Frontend UI structure of matching-service (match page, countdown view) and room page and handling of socket.io client logic - Set up collaboration-service with yjs - Made API call to question-service to fetch question to room page - Implemented refreshing of question in room Devops: - Dockerize application
	Report Contributed to the following section:

<table>
<tr><td></td><td>- [4] Requirements Specification
- [6.4, 6.5.4, 6.6, 6.7.2] Application Design</td></tr>
<tr><td rowspan="2">Papattarada Apithanangsiri</td><td>Code

Frontend:
- Handled UI enhancements for user, matching, question, history, collaboration services
- UI implementation for chat service
- Handled UX enhancements such as responsive design and adjustable splitscreen.

Backend/Real-time communication:
- Implement real-time client-server communication in matching and chat service
- Awareness and presence implementation for collaboration service in code editor

Database:
- Define schema and ORM layer for matching service
Devops:
- Conduct E2E testing using cypress framework
- Set up GitHub Actions workflow to activate CI upon push

Miscellaneous:
- Set up ESLint for style check
- Set up issue tracker, branch naming convention, squash merge flow</td></tr>
<tr><td>Report
Contributed to the following section:
- [3] Overall description
- [4.1, 4.2, 4.4] Requirement specifications
- [5] Software development process
- [6.1, 6.2, 6.3, 6.4, 6.5.1, 6.5.2] Application design

Diagram and implementation details:
- Overall architecture design
- Design patterns: MVC, pub-sub
- Microservices design: matching-service, chat-service
- Other design considerations: Extensibility

Devops:</td></tr>
</table>

	- Continuous Integration
Bryan	**Code** - Implement JWT authentication - Implement cookies for JWT and username - Implement protection to backend endpoints - Implement protection of frontend routes (pages that can only be viewed by an authenticated user) - Implement UI for Account Deletion - Set up Question Service (with Ying Hui) - Populate Question Bank - Implement storage and display of Question image in question bank and frontend respectively. - Implement update and retrieval of history records between frontend and backend. - Implement Home UI
	Report - [4] Requirements Specification - [5] Software Development Process - [6.5.3, 6.5.5, 6.5.6, 6.7.1] Application Design

2. Introduction

2.1 Background

Increasingly, students face challenging technical interviews when applying for jobs which many have difficulty dealing with. Issues range from a lack of communication skills to articulate their thought process out loud to an outright inability to understand and solve the given problem. Moreover, grinding practice questions can be tedious and monotonous.

2.2 Purpose of PeerPrep

We aim to develop a web application that can serve as an interview preparation platform and peer matching system, where students can find peers to practise whiteboard-style interview questions together. This platform allows students to practise their communication skills and

learn from one another. It can also break the monotony of revising alone, and generate a sense of camaraderie to get through the grind together.

2.3 Project scope

PeerPrep focuses on university students pursuing technology-related majors such as Computer Science. These people are usually the ones who seek technical internships or jobs after graduation. PeerPrep aims to provide a platform for these groups of people to prepare for their technical interviews and simulate the environment during the technical interviews, where students will be solving questions and editing code live with another person in the room.

Questions selected for this version of PeerPrep are a small subset of important LeetCode questions, based on 3 levels of difficulties defined by LeetCode. We think that these selected questions are important in technical interview preparations. Future versions of PeerPrep could be updated with more questions.

The current version of PeerPrep only allows collaboration between 2 users at any given point in time. Future versions of PeerPrep could allow for multiple collaborators in a room to help facilitate the experience of collaborative interview preparation.

3. Overall description

3.1 Product perspective

We envision a product that allows students to practise whiteboard style technical interviews together. Students would be matched in a real-time manner with another student that chose the same difficulty level (i.e. easy, medium, hard). They can practise tackling interview questions collaboratively through an online code editor that allows students to view changes to the written code in a real-time manner. Additionally, they can also communicate and discuss with each other through live chat. Having successfully completed the questions, the students can choose to generate another random question of the same difficulty to practise. They can also choose to leave the room when they are done. Students can also keep track of the number of questions they have practised, which are also categorised into easy, medium, and hard difficulty levels.

3.2 Product features

The following features were implemented:

1. User authentication
2. User account Management
3. Real-time matching: match users based on 3 difficulty levels: easy, medium, hard
4. Real-time code editor: syntax highlighting and awareness and presence system through cursor location
5. Real-time chat
6. Question display and generation
7. History of previous attempts

3.3 Operating environment

1. Development
   Our development took place in both Unix and Windows environments.

2. Server-side
   The server side component is able to operate within both Unix and Windows operating systems.

3. Client-side components
   Client side components are able to operate within common web browsers such as Google Chrome and Mozilla Firefox.

4. Requirements Specification

4.1 Functional Requirements

The following subsections details the requirement specifications for each microservice. Each table contains both the Functional Requirement (FR) and Non-Functional Requirement (NFR) specific to the microservice.

Each requirement is given a priority label - high, medium, low - that corresponds to the importance of the FR to achieve the purpose of the project.

High Priority - Must complete for Minimum Viable Product
Medium Priority - Important for smoother user experience

Low Priority - Implement only if there is sufficient time and resources

4.1.1 User Service

FRs	Description	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
NFRs		
NFR1.1 - Security	User password should be hashed and salted before storing in the DB.	Medium
NFR1.2 - Security	Users should be automatically logged out after 30 mins of inactivity.	Low
NFR1.3- Security	The password should be alphanumeric and be at least of length 8.	Medium
NFR1.4 - Security	The system should only allow users to access APIs that they are authorised to.	High

4.1.2 Matching Service

FRs	Description	Priority
FR2.1	The system should allow users to select the difficulty levels (easy, medium, hard) of the questions they wish to attempt	High
FR2.2	The system should be able to match two waiting users with same difficulty levels	High

FR2.3	The system should direct two users that have been matched to the same room.	High
FR2.4	If there is a valid match, the system should match the users within 30s	High
FR2.5	The system should inform the users that no match is available if a match cannot be found within 30 seconds	High
FR2.6	The system should provide a means for the users to leave a room once matched	Medium
FR2.7	The system should allow users to exit the matching process before the timer ends.	Medium
NFRs		
NFR2.1 - Scalability	The system should be able to support up to 1000 matched concurrent users	Low

4.1.3 Question Service

FRs	Description	Priority
FR3.1	The system contains questions with easy, medium and hard difficulty levels	High
FR3.2	The system should allocate questions that match with the user's selected difficulty level.	High
FR3.3	Within the same difficulty level, the system should generate selected question randomly to the users	High
FR3.4	The system should allow users to generate a new question while in the same room	Medium
FR3.5	The system should display an image for applicable questions to help the user understand the question.	Low
FR3.6	The system allows users to select questions of a particular topic	Low
FR3.7	For each question, the system should provide template code for users to fill in.	Low

NFRs		
NFR 3.1 - Performance	The question should be retrieved and displayed on the UI within 5 seconds.	Medium

4.1.4 Collaboration Service

FRs	Description	Priority
FR4.1	The systems should allow two users should be able to see each other's cursor in the code editor	High
FR4.2	The system should allow two users in the same room to edit code concurrently in the code editor in real-time	High
NFRs		
NFR4.1 - Performance	The changes in the code editor should be reflected in less than 1s.	Medium

4.1.5 Chat Service

FRs	Description	Priority
FR5.1	The system should allow users to send text messages to each other when in the same room.	Medium
FR5.2	The system should allow users to receive text messages from each other when in the same room.	Medium
FR5.3	The system should allow users to view the text conversation history in the same room.	Medium
NFRs		
NFR5.1 - Performance	The system should allow users to receive text in a real-time manner such that the information shall be displayed within 3 seconds.	Medium

4.1.6 History Service

FRs	Description	Priority
FR6.1	The system should be able to store a record of questions that the user has attempted previously.	Medium
FR6.2	The system should allow the user to view the number of questions that he has attempted so far for each difficulty level.	Medium

4.2 Other Non-Functional Requirements

The following table lists additional Non-Functional Requirements that are not specific to any microservice.

NFRs	Description	Priority
NFR7.1- Interoperability	Users should be able to run the service on different platforms (LTS of major browsers - Chrome, Firefox, Safari)	High
NFR7.2 - Performance	Each request from frontend should be processed by 5 seconds	High
NFR7.3 - Availability	The system should have an uptime of 99% at all times	Medium
NFR7.4 - Usability	The UI should be user-friendly enough for first-time users to easily navigate through main functions intuitively without prior knowledge about the application. (within 15 seconds).	Medium
NFR7.5 - Usability	The UI should have a consistent design scheme for a comfortable user experience.	Medium
NFR7.6 - Usability	The system should have a responsive design such that it could adjust to users' different screen sizes (i.e. the design should not make it completely impossible for users with smaller viewport e.g. smaller screen, tablet, mobile)	Low

4.3 Quality Attributes Prioritisation

Attributes	Score	Availability	Integrity	Performance	Reliability	Robustness	Security	Usability	Extensibility
Availability	1		^	^	^	^	^	<	^
Integrity	4			<	<	<	^	^	^
Performance	3				<	<	^	^	^
Reliability	1					^	^	^	^
Robustness	3						^	^	<
Security	6							^	<
Usability	6								<
Extensibility	4								

Quality Attributes Prioritisation Matrix

Based on the prioritisation matrix above, our group has decided to prioritise usability, security and extensibility.

When focusing on usability, we aim to make the User Interface (UI) easy to use for first-time users and fun to come back to for current users. The UI should provide informative feedback messages during error handling and successful operations to ensure a positive user experience.

Furthermore, when focusing on security, we want to ensure that unauthorised access to the web application, the API, and database are blocked. This is to prevent any loss of data, as well as unnecessary use of resources. Improving the security of the application also in turn maintains the integrity of the data that we use.

Lastly, when focusing on extensibility, we ensure that our implementation adheres to design principles, such as Single Responsibility Principle and loose coupling, as much as possible. This makes the code easier to understand, and also makes it easier for developers to debug the code when there are issues, or to add enhancements to current implementation.

4.4 Requirement Traceability

The following table traces each use case to their corresponding functional requirements and code elements (represented by our pull requests). Description of each use case can be found in Appendix A.

Use Case	Functional Requirement	Code Element
UC1 - Account Creation	FR1.1	Provided in given code template
	FR1.2	(#34) Implement logic to ensure unique usernames during account creation
UC2 - Login	FR1.3	(#23) Implement Login and Logout (#73) Implement refreshing of tokens if refreshToken is still valid (#63) Implement storage of returned JWT in cookie for successful logins (#70) Implement Authorizations to backend and frontend API with cookies (#42) Add simple sign up login page
UC3 - Logout	FR1.4	(#68) Implement UI and update backend for cookies for logout
UC4 - Account Deletion	FR1.5	(#35) Implement Account Deletion (#98) Implement UI for Delete Account (#68) Implement UI and update backend for cookies for logout
UC5 - Change Password	FR1.6	(#47) Implement ability to

		change password (#67) Add change password UI (#143) Improve UI for change password page
UC6 - Find Match	FR2.1	(#11) Add frontend view to select question difficulty
	FR2.2	(#84) Match frontend events to backend and abstract socket context (#12) Initialize socket server instance (#14) Implement match model and ORM (#53) Implement matching related event (#60) Milestone 1: matching service (#81) All rooms and events related implementation supporting many sockets
	FR 2.4	(#16) Add round countdown timer UI (#60) Milestone 1: matching service (#84) Match frontend events to backend and abstract socket context
	FR2.5	(#16) Add round countdown timer UI

		(#53) Implement matching related event (#60) Milestone 1: matching service
	FR2.7	(#84) Match frontend events to backend and abstract socket context (#53) Implement matching related event (#60) Milestone 1: matching service
UC7 - Join Room	FR2.3	(#26) UI layout for room page view (#84) Match frontend events to backend and abstract socket context (#81) All rooms and events related implementation supporting many sockets
	FR3.1	(#109) Add question bank (#131) Implement storing and displaying of pictures for questions
	FR3.2	(#114) Reflect question on room page (#119) Add Question Display component (#131) Implement storing and displaying of pictures for questions (#104) Implement Question

		Service - Retrieve Qn by difficulty API
	FR3.3	(#104) Implement Question Service - Retrieve Qn by difficulty API
	FR6.1	(#135) Implement History Service API (#173) Implement population of attempted questions
UC8 - Leave Room	FR2.6	(#81) All rooms and events related implementation supporting many sockets
UC9 - Code Collaboration	FR4.1	(#100) Set up yjs and codemirror (#107) Milestone 2: collaboration service
	FR4.2	(#92) Set up frontend UI for code editor (#96) Implement collaborative functionality of code editor
UC10 - Chat	FR5.1	(#125) Implement chat service
	FR5.2	
	FR5.3	
UC11 - Refresh Question	FR3.3	(#104) Implement Question Service - Retrieve Qn by difficulty API
	FR3.4	(#104) Implement Question Service - Retrieve Qn by difficulty API (#173) Implement population

		of attempted questions (#150) Refresh question
UC12 - View Attempted Questions History	FR6.2	(#135) Implement History Service API (#156) Implement Home UI Updates

Some code elements could not be linked to a specific use case, but they are necessary for the overall functioning of the application. See Section 5.3 on the team's work breakdown which details how each requirement (both FRs and NFRs) are mapped to their corresponding code elements.

5. Software Development Process

5.1 Weekly Sprints

The team followed the Scrum framework, and held weekly sprints on Sundays. At the start of the sprint, we discuss our progress for the week - sharing what has been completed, what is yet to be done, and what troubles we faced. We then come up with the backlog for the week based on the FRs and NFRs and assign tasks based on the capabilities of each member. If there are any group tasks for the week, we also worked on it during this meeting.

5.2. Development Process

5.2.1 Coding Standards and Style Checks

At the start of the development, the team has decided to adopt the Airbnb style guide for JavaScript with the following customization:

- Indentation will be 4 spaces
- Set generic line-break to support both Linux/Unix and Windows
- Limit number of characters per line to 120

ESLint was also set up to facilitate the process of identifying coding standard violations.

5.2.2 Issue Tracker

We made use of the Issue Tracker in GitHub to track our progress, as well as any bugs encountered.

The following tables show the tags used to tag PRs/Issues for issue tracking purposes.

Tag	Description
matching-service	PRs/Issues related to matching service
user-service	PRs/Issues related to user service
question-service	PRs/Issues related to question service
collaboration-service	PRs/Issues related to collaboration service
chat-service	PRs/Issues related to chat service
history-service	PRs/Issues related to history service
to-merge	PRs reviewed and ready to merge
to-review	PRs ready to be reviewed
backend	PRs/Issues related to backend
frontend	PRs/Issues related to frontend
database	PRs/Issues related to database
bug	Something is not working as expected
nice-to-have	An enhancement feature to the app
priority.High	Must do
priority.Medium	Must do but can be done slightly later
priority.Low	Nice to do, but not something urgent at this stage

5.2.3 Branching

The team has decided to create a branch for each microservice. We then branch off from each microservice branch to develop a particular feature.

Branches are named in this format: (issue number)-(keywords from issue title)

5.2.4 Pull Requests

After completing a feature, we make a PR to the respective microservice branch. PRs have to be reviewed and tested by a fellow team member before merging. Branch protection is also enabled for main and microservice branches.

To create the PR description, we followed the steps below to standardise our PR, and make it easy for our teammates to test and review.

1. Fixes #issue-to-fix
2. Summary: Contains the proposed commit message for squash merge in the following format:

```
{current situation} -- use present tense

{why it needs to change}

{what is being done about it} -- use imperative mood

{why it is done that way}

{any other relevant info}
```

Fig 1: Image showing description of a proposed commit message

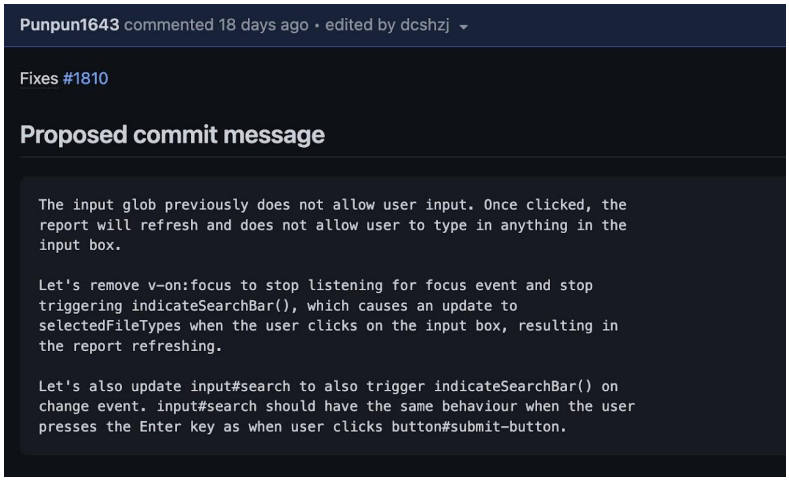


Fig 2: Example of a proposed commit message

3. Step-by-step guide on how to do manual testing for the feature

Once the PR is ready for review, we tag it with *ToReview*. After reviewing and performing acceptance testing, we change the tag to *ToMerge*.

When merging the PR, we make use of squash merge to keep a cleaner commit history on both microservice branch and main branch. For the commit title of the squash merge, we adhere to "[#issue number] keywords from the issue" when applicable.

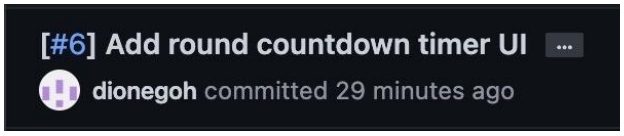


Fig 3: Example of squash merge and commit title

5.3 Gantt Chart (Work breakdown)

The following tables are Gantt Charts for each milestone and represent the FRs and NFRs implementation tasks done by each member. It does not include tasks related to bug fixes.

Milestone 1:

	3	4	5	6	Recess
Dione	Familiarising code base	FR2.1	FR2.2 - FE FR2.5 - FE	FR2.4 - FE	FR2.5
Pun Pun	Familiarising code base	FR2.2	FR2.3	FR2.3-2.4	Implement databases and backend for MVP (FR2.3) FE + BE Integration (FR2.4) Room events implementation (FR2.4) Leave the room once matched (FR2.5)

	3	4	5	6	Recess
Bryan	Familiarising code base	FR1.3	FR1.3, FR1.4, NFR1.2	FR1.3-Cookie, FR1.8, FR1.4(blacklist), FR1.5(blacklist),	FR1.4 FE + BE, FR1.8 FE + BE
Ying Hui	Familiarising code base	NFR1	FR1.2, FR1.5	FR1.6	FR1.3-FE, FR1.6-FE

Milestone 2:

	7	8	9	10
Dione	FR2.6, 2.7 - FE	FR2.2, socket context, integration of frontend and backend	FR4.2	FR3.3 - FE, documentation
Pun Pun	FR2.2, FR2.4, FR2.7, FR2.5, 2.6	FR2.2, 2.3, 2.6,	FR4.1,4.2	FR5.1, 5.2, 5.3, documentation
Bryan	FR1.5	NFR1.2	NFR1.4	FR3.1, documentation
Ying Hui	FR1.6-FE,	NFR1.3 FE + BE, FE-Navbar	FR3.1, FR3.2, FR3.3, FR3.4	FR3.2-FE, documentation

Milestone 3:

	11	12	13
Dione	FR4.1, Enhance matching-service UI	Dockerize, FR3.4	Documentation
Pun Pun	Enhance existing UI	CI	documentation
Bryan	FR3.1-FE, FR3.2-FE	FR6.1-FE, FR6.2-FE	documentation
Ying Hui	FR6.1-BE, FR6.2-BE	FR1.6-FE, documentation	FR6.1-FE, documentation

The following tables show a detailed breakdown of implementation progress by services.

User service

Milestone 1 (Frontend)

FR	Task	Date	Remarks
1.3	Add UI for login page and connect to backend	Recess week	(#42) Add simple sign up login page

Milestone 1 (Backend)

FR	Task	Date	Remarks
NFR1.1	Create Hashing Password and verify passwords functions	Week 4	(#17) Password Hashing and Salting
NFR1.2	Users should be logged out after 30 mins of inactivity.	Week 5	(#70) Implement Authorizations to backend and frontend API with cookies
FR1.2	Ensure Unique usernames during account creation	Week 5	(#34) Implement logic to ensure unique usernames during account creation
FR1.3	Implement login with JWT	Week 4, 5	(#23) Implement Login

			and Logout
FR1.3	Implement login, auth token, refreshToken, logout	Week 5	(#73) Implement refreshing of tokens if refreshToken is still valid
FR1.3	Save JWT to cookie	Week 6	(#63) Implement storage of returned JWT in cookie for successful logins
FR1.3-FR1.4	Update all user methods to use cookies instead of bearer token	Week 6	(#70) Implement Authorizations to backend and frontend API with cookies (#68) Implement UI and update backend for cookies for logout (#63) Implement storage of returned JWT in cookie for successful logins
FR1.4	Implement logout	Week 5	(#68) Implement UI and update backend for cookies for logout
FR1.5	Implement delete account	Week 5	(#35) Implement Account Deletion -blocked, need to blacklist the token
FR1.6	Implement change password	Week 6	(#47) Implement ability to change password
NFR1.4	Add authorization to routes (/logout, /home) FE+BE	Week 6, Week 9	(#70) Implement Authorizations to backend and frontend API with cookies (#118) Implement FE route protections

Milestone 2 (Frontend)

FR	Task	Date	Remarks
1.6	UI for change password • *Design Decision*: Store `username` of user after logging in, to use in API calls for user-related functions • `username` was initially stored using React Context, however, the value disappears when page is refreshed • Switched to session Storage, however race conditions were encountered • Switched to passing in `username` along with jwt tokens in cookies instead • This also allows users to be able to re-access the pages in the application without having to login again (within 30minutes)	Week 7	(#67) Add change password UI (#99 Update username handling from using Session to Cookies)
1.5	UI for delete account	Week 9	(#98) Implement UI for Delete Account
NFR1.3	Add error indicator UIs and client side validation for weak passwords on change password page and sign up page	Week 8	(#67) Add change password UI (#103 Fix SignUp Page password strength error display bug)
1.X	Navbar UI	Week 8	(#95) Add NavBar component to frontend

Milestone 2 (Backend)

FR	Task	Date	Remarks
1.5	Implement delete account with blacklist tokens	Week 6	(#68) Implement UI and update backend for cookies for logout
NFR1.3	Implement regex for password	Week 8	(#89 Implement backend

	strength requirement - change password, sign up API		validation for password strength)
FR1.8	Authorisation for delete account and change passwords	Week 7, 9	(#75) Add protection to changePassword endpoint (#118) Implement FE route protections

Milestone 3 (Frontend)

FR	Task	Date	Remarks
1.1, 1.3	Revamp sign up/ login UI	Week	(#128) Revamp UI
1.6	Enhance Change Password UI	Week 12	(#143) Improve UI for change password page

Matching service

Milestone 1 (Frontend)

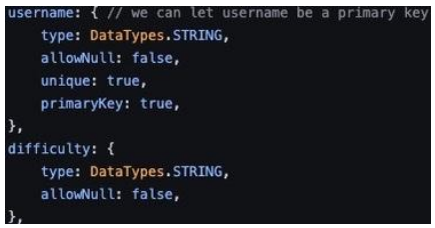
FR	Task	Date	Remarks
2.1	- Create frontend view for match selection	Week 4	(#11) Add frontend view to select question difficulty
2.4	- Create frontend view for countdown timer	Week 4	(#16) Add round countdown timer UI
2.5	- Create frontend view to notify the user when there is no match found within 30s	Week 4	(#16) Add round countdown timer UI
2.3	- Create frontend view for room page - Implement routing for frontend matchings - Create socket.io client instance	Week 5	(#26) UI layout for room page view
2.2	- Allow socket.io client to emit matching-related events	Week 6 & 7	(#84) Match frontend events to backend and abstract socket context

	The following are matching-related events **emitted** by the socket.io client - match-easy : when a client choses 'easy' as the difficulty level - match-medium : when a client choses 'medium' as the difficulty level - match-hard : when a client choses 'hard' as the difficulty level - match-cancel : when the user cancels a match before he has been matched and before the 30s mark The following are matching-related events **listened** by the socket.io client - match-success : when the client has been matched with another client with the same difficulty level, route the client to the room page after making an API call to question-service to retrieve a question. The parameters passed to FE will be the socket Ids of both client, as well as the question		
2.4	The following are matching-related events **emitted** by the socket.io client - no-match-found : when a client is still not matched by the 30s mark, client will be informed that the matching is unsuccessful.	Week 6 & 7	(#84) Match frontend events to backend and abstract socket context
2.7	The following are matching-related events **emitted** by the socket.io client	Week 6 & 7	(#84) Match frontend events to backend and abstract socket context

	- match-cancel : when the user cancel a match before he has been matched and before the 30s mark		
2.6	- Allow socket.io client to emit room-related events The following are room-related events **emitted** by the socket.io client - join-room : allocate socket instances to a room with a particular socketid - leave-room : socket instances to leave the room with a particular socketid	Week 6 & 7	(#84) Match frontend events to backend and abstract socket context
2.3	- Allow socket.io client to emit room-related events The following are room-related events **emitted** by the socket.io client - join-room : allocate socket instances to a room with a particular socketid	Week 6 & 7	(#84) Match frontend events to backend and abstract socket context

Milestone 1 (Backend)

FR	Task	Date	Remarks
2.2	- Create a socket server instance - Setup repository.js that will create a new connection with the database	Week 4	(#12) Initialize socket server instance
2.2	- Create a SQLite model schema for pending matches (this is our initial schema and is no longer used in the current implementation)	Week 4 & 5	(#14) Implement match model and ORM This schema is later changed in milestone 2 to allow for socket-based implementation of matching related events

	 - Implement an ORM layer for for basic logic which acts as direct interaction with the database, including add, read, delete, put		
2.2	- Create a database controller layer (pendingMatchController.js) which acts as an intermediary layer between the ORM and socket server. *This **design decision** helps to abstract out the lower level details in the ORM layer that interacts directly with the database. This allow the socket controller to interact with the database without having much concern about the changes in the database and ORM implementation.	Week 6	(#53) Implement matching related event
2.2, 2.4	- Create a socket controller (socketHandler) as a server side event listener. This controller reacts to different events emit from the client side, and emit necessary events to the client side. The following are matching-related events **listened** by the socket.io server - match-easy : creates a pending match with 'easy' as the difficulty level upon the receive of the event - match-medium : creates a	Week 6 & 7	(#53) Implement matching related event (#60) Milestone 1: matching service

	pending match with ‘medium’ as the difficulty level upon the receive of the event - match-hard : creates a pending match with ‘hard’ as the difficulty level upon the receive of the event The following are matching-related events **emitted** by the socket.io server - match-success : emit to both the matched users upon a successful match with the same difficulty level		
2.7	The following are matching-related events **listened** by the socket.io server - match-cancel : upon the receive of the event, deletes the created pending match from the database if the pending match cancels the match before the 30s mark	Week 6 & 7	(#53) Implement matching related event (#60) Milestone 1: matching service
2.5	The following are matching-related events **listened** by the socket.io server - no-match-found : upon the receive of the event , delete the created pending match from the database if the pending match has yet to be matched by the 30s mark	Week 6 & 7	(#53) Implement matching related event (#60) Milestone 1: matching service

Milestone 2 (Frontend)

FR	Task	Date	Remarks
2.2	Match frontend emitted events,	Week 8	(#84) Match frontend events to

	and listening events with backend server		backend and abstract socket context
	Abstract socket.io client instance to a SocketContext.js file to reduce need to pass the socket instance around different components. This **design decision** is to prevent potential errors from passing the socket instance between components, and to ensure that the socket will not be created again if there is an existing instance.	Week 8	(#84) Match frontend events to backend and abstract socket context

Milestone 2 (Backend)

FR	Task	Date	Remarks
2.2	Update SQLite model schema for pending matches to take in socketid `socketid: { type: DataTypes.STRING, allowNull: false, unique: true, }, username: { type: DataTypes.STRING, allowNull: true, unique: true, }, difficulty: { type: DataTypes.STRING, allowNull: false, },` *This **design decision** allow for implementation of matching related events based on socket instances as each socket instance is assigned a unique	Week 7 & 8	(#81) All rooms and events related implementation supporting many sockets

	Socket ID. The socket ID could be used to uniquely identified a client as well as facilitate matching processes driven by client-server communication		
2.3	Implement the creation of room on the server side upon matchSuccess event to direct matched users to the same 'room'. This is a necessary step for code collaboration, which will happen in the same 'room' The following are room-related events **listened** on the server side - join-room : allocate socket instances to a room with a particular socketid upon the receive of the event - leave-room : socket instances to leave the room with a particular socketid upon the receive of the event	Week 7 & 8	(#81) All rooms and events related implementation supporting many sockets
2.6	The following are room-related events **listened** on the server side - leave-room : socket instances to leave the room with a particular socketid upon the receive of the event	Week 7 & 8	(#81) All rooms and events related implementation supporting many sockets
	- Integration of matching-services on the client and server side	Week 8	(#87) Milestone 2: matching service

Collaboration service

Milestone 2

FR	Task	Date	Remarks

4.2	- Set up code editor UI to allow for writing of code with auto indentation and syntax colouring	Week 9	(#92) Set up frontend UI for code editor
4.2	- Implement collaborative functionality to allow the real-time code collaboration by two users	Week 9	(#96) Implement collaborative functionality of code editor This implementation is no longer used as the high overhead from the emission of the entire code block to the server result in flickering
4.1	- Implementation of UI cursor with username to facilitate real-time collaboration, allowing another user to know where the other is editing. This prevents duplicate edits - Finalise collaboration service	Week 9	(#100) Set up yjs and codemirror (#107) Milestone 2: collaboration service

Question service

Milestone 2 (Frontend)

FR	Task	Date	Remarks
FR3.2-FE	UI for question display ● Matching-service will be responsible for retrieving the question, instead of frontend. The question will then be passed to frontend on match success	Week 10	(#114) Reflect question on room page (#119) Add Question Display component

Milestone 2 (Backend)

FR	Task	Date	Remarks
3.1	Create database for storing question banks of varying difficulty levels	Week 9	(#109) Add question bank

	• *Design Decision*: Used json files to store the questions. Changes to questions can be tracked with git • Imported into mongodb using command line tool • Questions obtained from LeetCode		
3.2	Implement API that allows retrieval of questions from question bank based on difficulty level	Week 9	(#104) Implement Question Service - Retrieve Qn by difficulty API
3.3	Add randomness to the way the questions are being retrieved in the API	Week 9	(#104) Implement Question Service - Retrieve Qn by difficulty API
3.4	Implement API that generates a new question based on difficulty level, different from the current question displayed	Week 9	(#104) Implement Question Service - Retrieve Qn by difficulty API

Milestone 3 (Frontend)

FR	Task	Date	Remarks
FR3.4-FE	Include refresh button for regenerating question • Socket client will be responsible for emitting refresh-question event when user confirms to generate a new question • Socket server upon receiving refresh-question, will make an API call to question-service to generate a new question, before emitting update-question event with the new question.	Week 12	(#150) Refresh question

	The room page would update its existing question with the new question generated		

Chat service

FR	Task	Date	Remarks
5.1	Allow one user to send messages to another user, and have the message reflected in both chat. The message owner is also identified based on who the sender is.	Week 9	(#125) Implement chat service
5.2	Allow one user to receive messages from another user, and have the message reflected in both chat. The message owner is also identified based on who the sender is.	Week 9	(#125) Implement chat service
5.3	Store the text messages using a temporary storage on the frontend side.	Week 9	(#125) Implement chat service

History service

Milestone 3 (Frontend)

FR	Task	Date	Remarks
6.1	Store questions attempted during match-success and refresh-question	Wk 13	(#173) Implement population of attempted questions

6.2	Implement dashboard to view #questions attempted	Wk 12	(#156) Implement Home UI Updates (#177) Update question progress calculation (#169) Re-inject Attempted Difficulty cards with live data

Milestone 3 (Backend)

FR	Task	Date	Remarks
6.1	Set up new database for History Service	Wk 11	(#135) Implement History Service API
6.1	Create API to save an attempted question to a user's record ● PUT API where resource identifier is the username	Wk 11	(#135) Implement History Service API ● Ensure a question is only saved once if question was attempted previously
6.2	Create API to retrieve a user's record of attempted question ● Able to filter record based on questions' difficulty level	Wk 11	(#135) Implement History Service API

6. Application Design

This section details the design decisions and implementation details for our application.

6.1 Tech stack

This subsection describes the project's tech stack and explains the rationale behind our choice of tech stack.

6.1.1 Frontend

The following table lists the main technologies used and their rationale in the development of the application's frontend.

	Technologies	Rationale
Frontend	React.js	● The component-based architecture approach allow for the development of micro-frontends ● Each frontend component represents a specific feature/sub-component of the entire application. This allow the split of application into decoupled component codebase ● Allow for sharing of reusable components across the application ● A lot of community support
	MUI v5	● A lot of community support ● Fast learning curve ● Provides out-of-box stylised components - reduces frontend development time

6.1.2 Backend

Across the microservices, we have chosen to adopt the same technologies to develop the backend APIs and functionalities. The following table lists the technologies and the accompanying rationales.

	Technologies	Rationale
Backend	Node.js,	● High speed and scalability due to single threaded

	Express.js	model with event loop ● The event-driven architecture and non-blocking runtime environment allow for process of multiple requests at the same time ● Fast data sync is possible ● Using node to create lightweight backend model for asynchronous data-loading via callback functions with react for Single Page Application (SPA) ● A lot of community support

6.1.3 Microservices

As the project follows the microservices-based architecture, each microservice can adopt a different tech stack for development. The following subsections list the tech stack chosen and their rationales for each microservice.

User Service

	Technologies	Rationale
Database	MongoDB Atlas	● Does not require username and passwords data in a relational schema ● NoSQL databases are easier and cheaper to scale up ● Cloud-based - less hassle of setting up, resulting in faster pace of development ● A lot of community support

Matching Service

	Technologies	Rationale
Database	Sequelize, SQLite	● The storing of pending matches data is temporary and only requires a very small database. ORM helps to speed up the development of simple and repetitive queries

Pub/Sub Messaging	Socket.io	● Allow for easy implementation of real-time bi-directional communication between client and server

Question Service

	Technologies	Rationale
Database	MongoDB Atlas	● Database is used as a question bank to store and retrieve documents quickly ● NoSQL databases are easier and cheaper to scale up ● Cloud-based - less hassle of setting up, resulting in faster pace of development ● A lot of community support

Collaboration Service

	Technologies	Rationale
Frontend	React.js	● Same rationale as above services
Frontend javascript component	CodeMirror	● Rich programming API with a focus on extensibility and allow for easy integration of code editor with functionalities like syntax highlighting and auto-indentation into the existing UI
Modular framework for real-time syncing	Yjs	● Allow the binding of a third-party editor to a Yjs document (e.g. CodeMirror) ● Support of awareness & presence which allow communication of more information which aid collaboration such as cursor position

Chat Service

	Technologies	Rationale
Pub sub	MongoDB Atlas	● Same rationale as above services

History Service

	Technologies	Rationale
Database	MongoDB Atlas	• Same rationale as above services

6.2 Overall Architecture Design

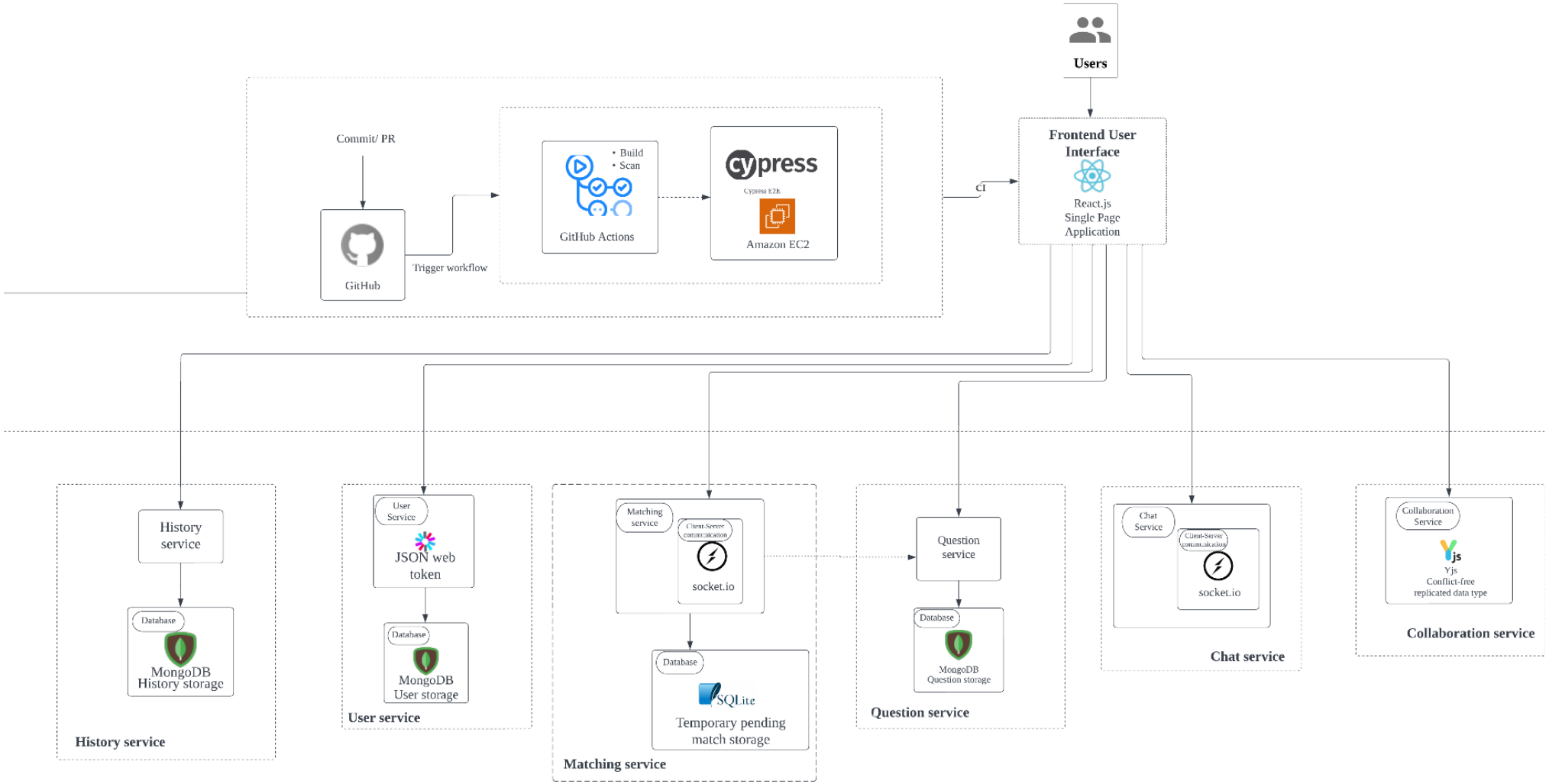


Fig 4: Architecture Diagram for PeerPrep Application

We adopt the microservices-based architecture in developing PeerPrep. After discussing the main features we want PeerPrep to have, we defined our microservices by adhering to Domain-Driven Design principle and modularity, where each microservice is responsible for one main function of PeerPrep. Each microservice also has an individual database where applicable. The design decisions for each component will be discussed in subsequent sections.

Following Domain-Driven Design also allows us to easily add new features that correspond to new domains and thus, new microservices. This was especially helpful when we were adding new microservices in Milestones 2 and 3.

In the next section, the design pattern used in the development of PeerPrep is illustrated.

6.3 Design patterns

6.3.1 Model View Controller Pattern (MVC)

In certain services like user service and matching service, the main functionalities involve data storage, retrieval of data, displaying of data to the user, responding to users' actions. We adopt the MVC pattern for these services.

Our decisions in adopting the MVC pattern are as follow:

Reason	Justification
High modularity and reusability	MVC follows the separation of concern principle.
Accelerate development as model, view, and controller could be developed in parallel	As there is low coupling and high cohesion between model, view, and controller, each parts have little dependency to each other. This allows for parallel development as there is little coupling between components, allowing for a faster development process.
Enhance UI modification capabilities	View is not highly coupled with model and controller, therefore changes in the view components are less likely to require changes in the model and controller.

The following is an example of how the MVC pattern is being applied in our application.

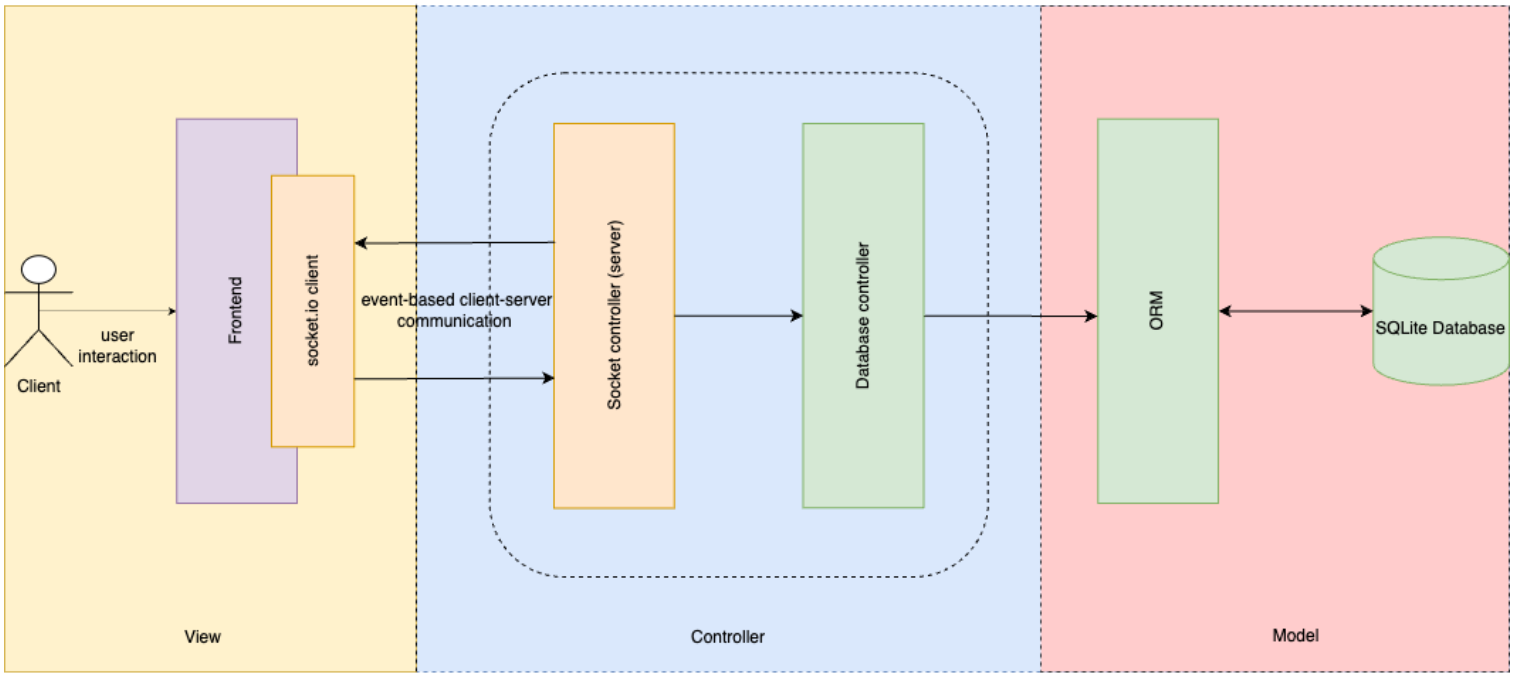


Fig 5: High-level view of MVC implementation in matching-service

6.3.2 Publisher-Subscriber Pattern

As our matching service and chat service rely on real-time communication between users, we make use of the Publisher-Subscriber (pub-sub) pattern. In pub-sub any message published to a topic is immediately received by all the subscribers of that topic. This facilitates our matching and chat services which are event-driven.

The following is the high-level overview of how pub-sub is applied in our application.

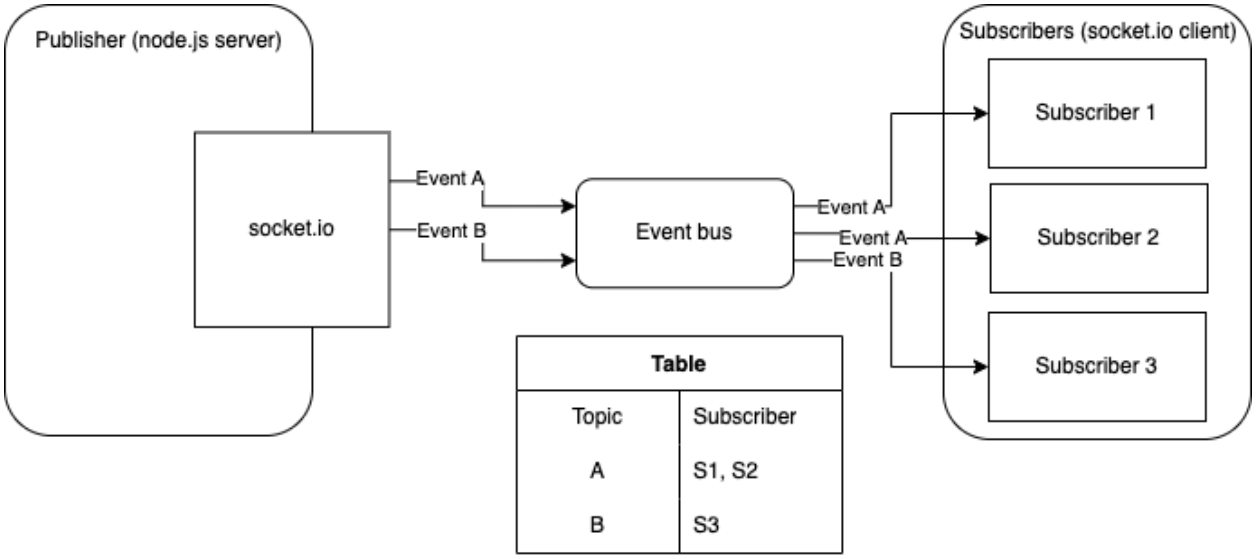


Fig 6: High-level view of pub-sub design pattern that is applied in our application. Note that the role could be switched (i.e. client is the publisher and server is the subscriber)

In the case where the server is acting as publisher and client is acting as subscriber, publisher can be modelled as a server who is sending messages where clients are those subscribing to messages. Events could be modelled as chat instances, request for match, or request to leave rooms.

We can see that there are no more than 2 subscribers to an event at any point in time, hence for each event topic there will only be at most 2 recipients.

6.4 Devops

6.4.1 CICD

Continuous Integration

We use Cypress as our frontend testing tool to conduct End-to-End testing to drive quality throughout the pipeline and replicate real-user scenarios.

Cypress Tests are automated and triggered by GitHub Actions upon push to the remote repository.

6.4.2 Using Docker

The application is deployed to a local staging environment on Docker. Each microservice has a Dockerfile with node:16 as a base image.

matching-service > Dockerfile

```
1   FROM node:16
2
3   #create app directory
4   WORKDIR /usr/app
5
6   #bundle app source
7   COPY . .
8
9   RUN npm install
10
11  EXPOSE 8001
12
13  CMD [ "npm", "run", "dev"]
```

Fig 7: An example of a Dockerfile for matching-service

A docker-compose.yml file is added in the root directory, listing all the microservices and frontend with its dependencies. Frontend has a port mapping of 3000:3000, and the following table shows the port mapping from the local machine to the Docker container for each microservice:

Microservice	Local port	Docker port
Chat service	8003	8003
History service	8004	8004
Matching service	8001	8001
Question service	8002	8002
User service	8000	8000

For each service in the docker-compose.yml file, the restart property is set to `always`. This ensures that if any container stops due to any reason, it will be immediately restarted.

6.5 Microservices Design

Our application comprises different microservices to provide different functionalities. Some of these microservices are hosted in-house, while others have some implementations that rely on external third-party APIs. We outline the current arrangement in the table below:

Microservice	In-House/Third-Party	External Service Relied On
Chat service	In-House	NA
History service	In-House	NA
Matching service	In-House	NA
Question service	In-House	NA
User service	In-House	NA
Collaboration service (Real-time peer programming)	Third-Party	Yjs

6.5.1 Matching Service

The matching service follows the MVC design pattern where users interact with the service through the shared frontend user interface.

The model represents a temporary data storage of users who have yet to be matched (pending match users). Users could be uniquely identified by their assigned *socketid* or their registered username.

Users are matched based on their position in the matching queue. Our implementation utilises the idea of temporary storage where the role is to store pending match users of each difficulty level. This means that there will only be a maximum of 1 user of each difficulty level waiting in a queue. Once there is a match for a specific difficulty level, the user of that respective difficulty level would be removed from the queue.

If a user is not matched within 30 seconds, the client will notify the server and the server will take actions to remove the respective user from the queue.

Similarly, if the user cancels the matching process before 30 seconds, the user's data will be deleted from the database.

Server structure

To follow the MVC design pattern where controller and model are clearly separated in their implementation. Each satisfies a specific role.

```
.
├── controller
|   ├── controller.js
|   ├── pendingMatchController.js
|   └── socketHandler
|       ├── globalHandler.js
|       └── pendingMatchHandler.js
├── index.js
├── model
|   ├── orm.js
|   ├── pendingMatchModel.js
|   ├── pendingMatchOrm.js
|   └── repository.js
├── node_modules  [190 entries exceeds filelimit, not opening dir]
├── package-lock.json
├── package.json
├── route
|   └── pendingMatchRoute.js
└── routes.js
```

Fig 8 : Server structure of matching-service

The duty of each file:

- Index.js : the entrypoint of the server which creates the components and initialises the applications. This is also where the server sockets are initialised
- socketHandler: the handlers of the events driven by the sockets communication between client and server
    - globalHander.js : handlers of events related to overall matching service
    - pendingMatchHandler.js : handlers of events related to matching logic
- Model: handles all data and its related logic
    - repository.js : handles database connection
    - pendingMatchModel.js : database schema for pending match
    - orm.js : basic low-level manipulation of data in the SQLite database
    - pendingMatchOrm.js : manipulation of data logic specific to each socket event

Each user details in the matching service is temporary stored in the following JSON format until a match is found:

```
[
    {
        "id": 2,
        "socketid": "CrYr_WNGdckN5nmNAAAD",
        "username": null,
        "difficulty": "hard",
        "createdAt": "2022-10-03T18:51:32.986Z",
        "updatedAt": "2022-10-03T18:51:32.986Z"
    },
    {
        "id": 3,
        "socketid": "nisJ690w6LauUgK4AAAF",
        "username": null,
        "difficulty": "medium",
        "createdAt": "2022-10-03T18:52:32.731Z",
        "updatedAt": "2022-10-03T18:52:32.731Z"
    }
]
```


Fig 9: User details in matching-service

Activity diagram for Matching-related Logic

The following is an activity diagram that captures the logical flow in which the match events are executed.

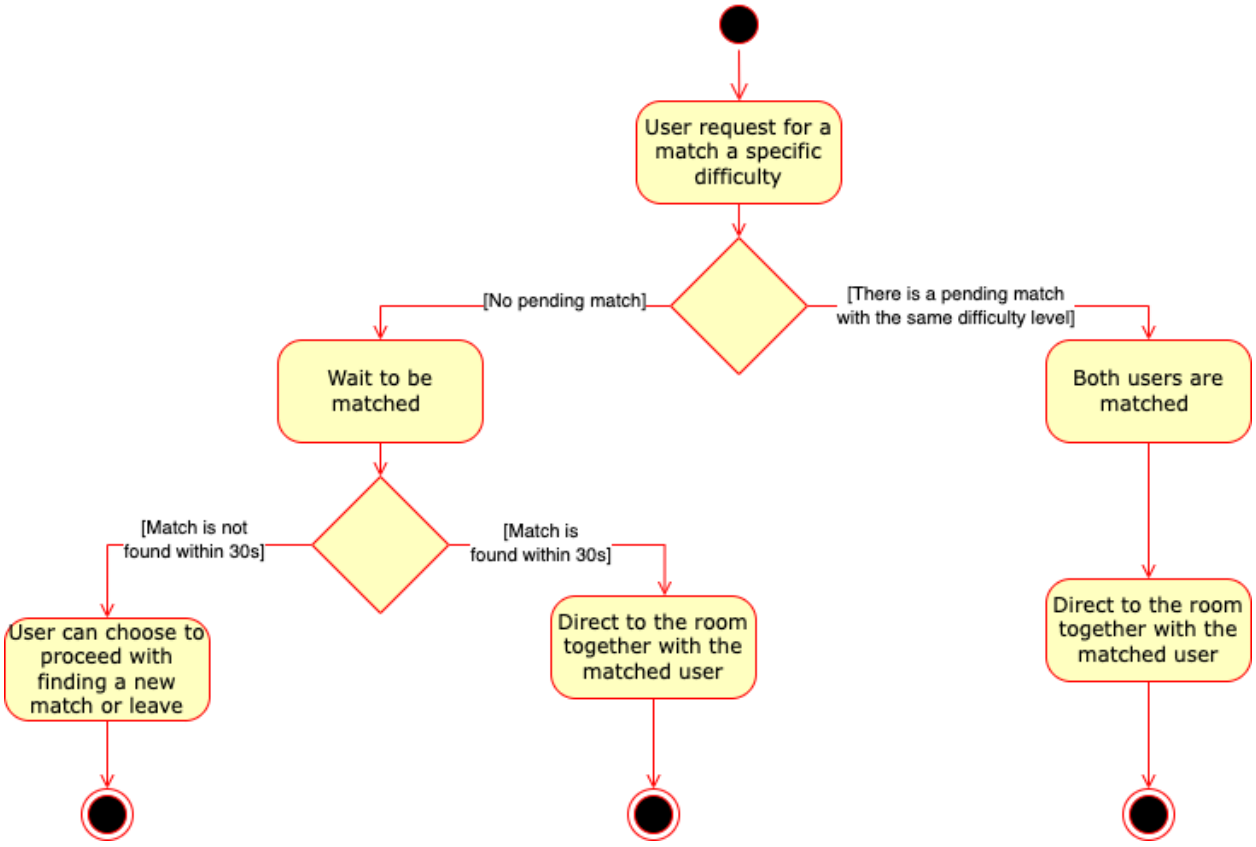


Fig 10: Activity diagram for matching related logic

Activity diagram for Room-related logic

The following is the activity diagram capturing the logical flow in which the leave room events are executed.

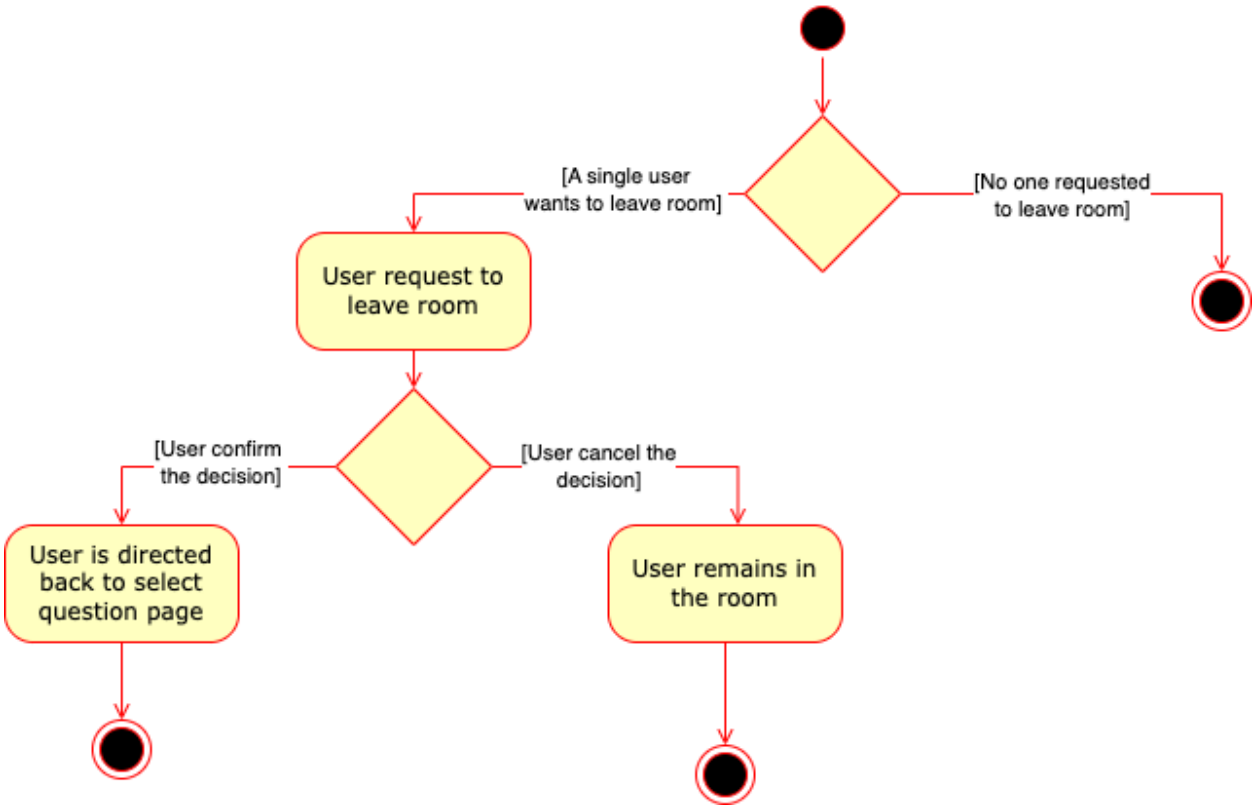


Fig 11: Activity diagram for room-related logic

Matching events emission and reception

The following is a sequence diagram capturing the interaction between the Client, Server and the database upon a user request to be matched with a specified level of difficulty.

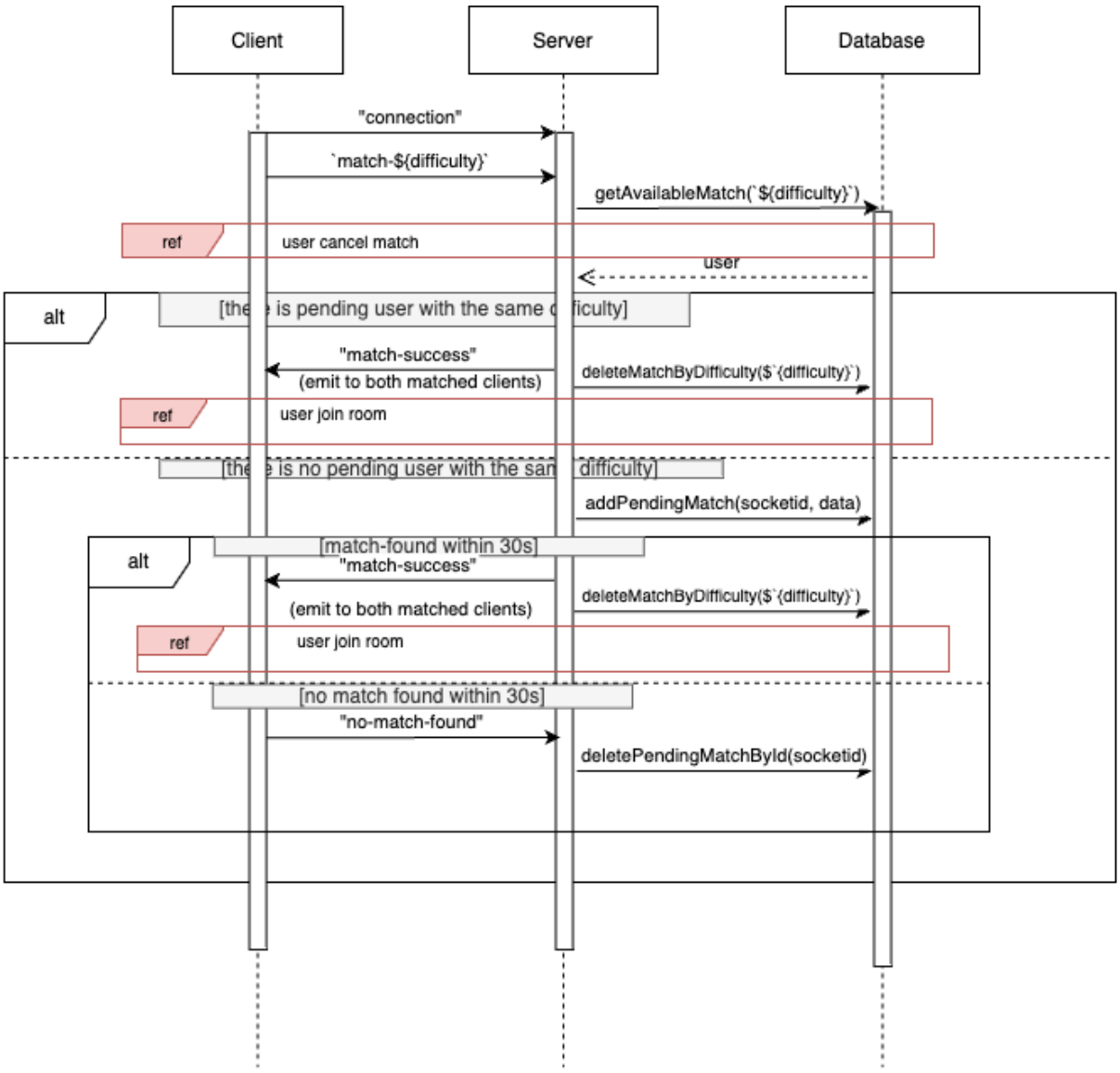


Fig 12: sequence diagram showing interaction between client, server, and the database for matching-related events (diagram: https://drive.google.com/file/d/1t9fXhL7WOx5EgA0J48ICw77141BOqh45/view?usp=sharing)

Reference frame with reference to Figure 12 – user cancel match

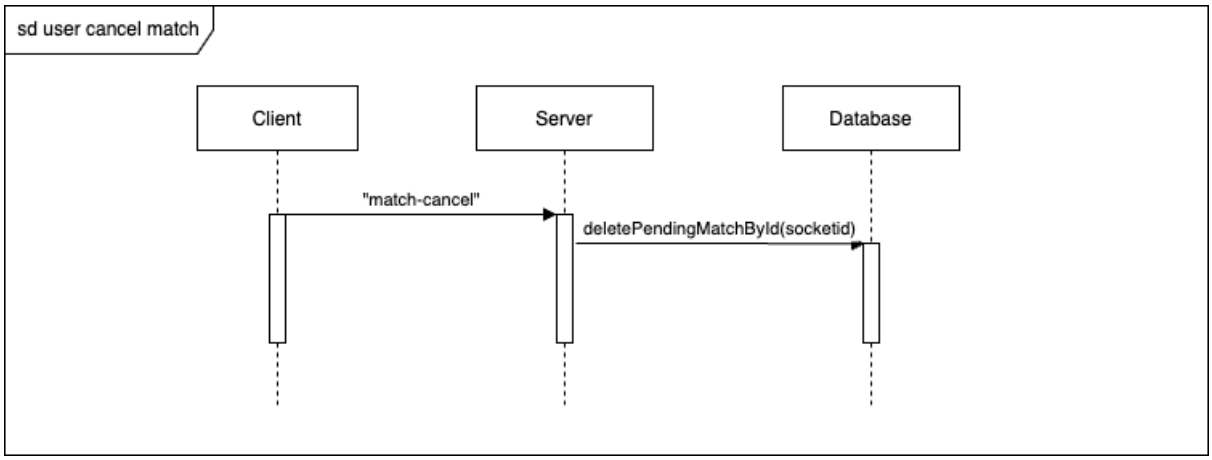


Fig 13: sequence diagram showing interaction between client, server, and the database for match-cancel event upon user cancelling the matching process
(diagram: https://drive.google.com/file/d/1t9fXhL7WOx5EgA0J48ICw77141BOqh45/view?usp=sharing)

Reference frame with reference to Figure 12 – user join room

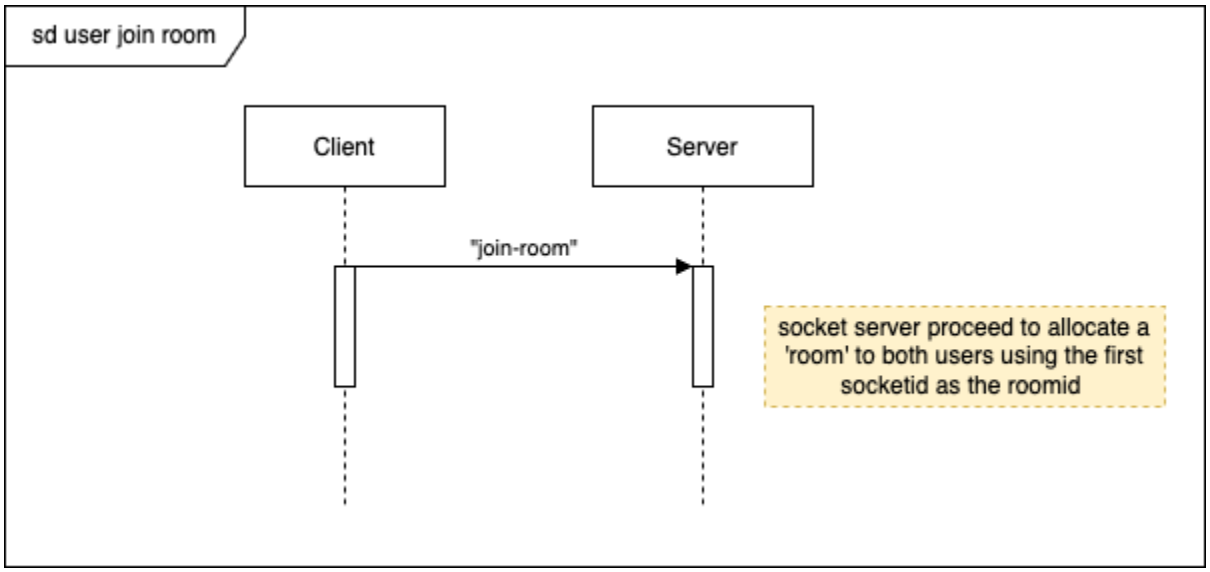


Fig 14: sequence diagram showing client-server communication to facilitate 'room' allocation upon successful match
(diagram: https://drive.google.com/file/d/1t9fXhL7WOx5EgA0J48ICw77141BOqh45/view?usp=sharing)

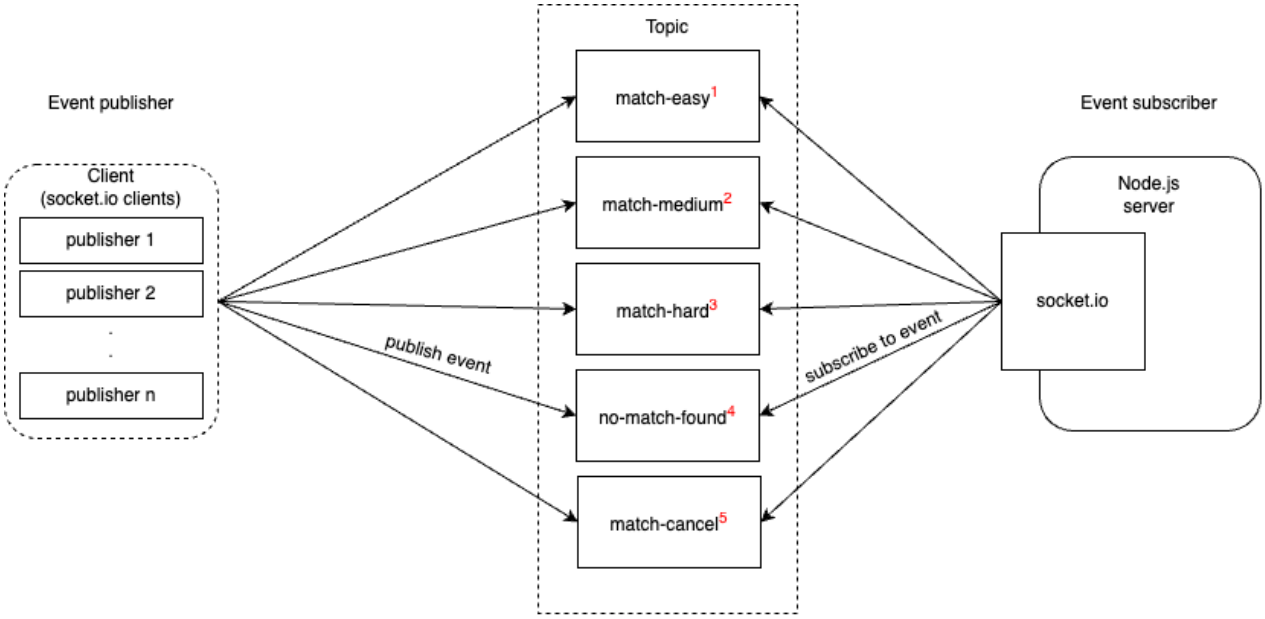
The matching service utilises *socket.io*, an event-driven real-time library that facilitates bi-directional client-server communication.

As discussed in Section 6.3.2 on Pub-Sub Pattern, socket.io utilises pub-sub design pattern to facilitate communication between the client-side that runs in the browser and the server-side run within Node.js.

Application of pub-sub design pattern

We can see that pub-sub design pattern helps to reduce latency between client and server communication as the client would not need to continuously poll the server for any message updates and is important for matching service which focuses on real-time communication.

The pub-sub design pattern also facilitates separation of concern where each event is only responsible for one single purpose, containing data and triggering actions related to that particular purpose. Each subscriber could subscribe to the topics that they are interested in.



1. when a client choses 'easy' as the difficulty level
2. when a client choses 'medium' as the difficulty level
3. when a client choses 'hard' as the difficulty level
4. when a client is still not matched by the 30s mark
5. when the user cancel a match before he has been matched and before the 30s mark

Fig 15: socket.io client instance acting as an event publisher, the server subscribe to these events (diagram:https://drive.google.com/file/d/1Sn7fuRsyP7OaLHG37LEK8bKgToh8MXRO/view?usp=sharing)

Once connected, clients are uniquely identified based on an assigned unique *socketid*. This means that the server side is able to emit events to specific client-side subscribers identified by their unique *socketid*.

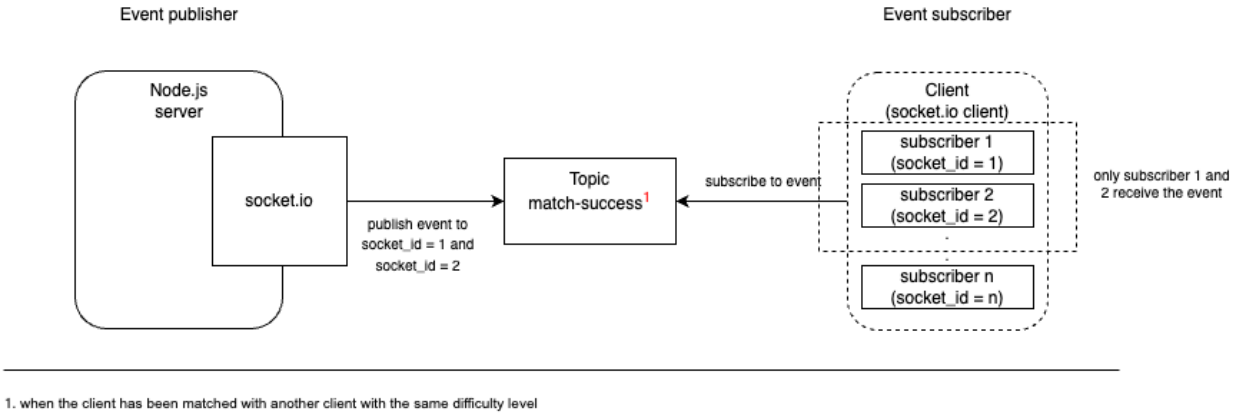


Fig 16: the server instance acting as an event publisher, socket.io clients subscribe to these events (diagram: https://drive.google.com/file/d/1Ab1yMOQIAE9XSvS3_u-91z166gC1WMrV/view?usp=sharing)

Room allocation events

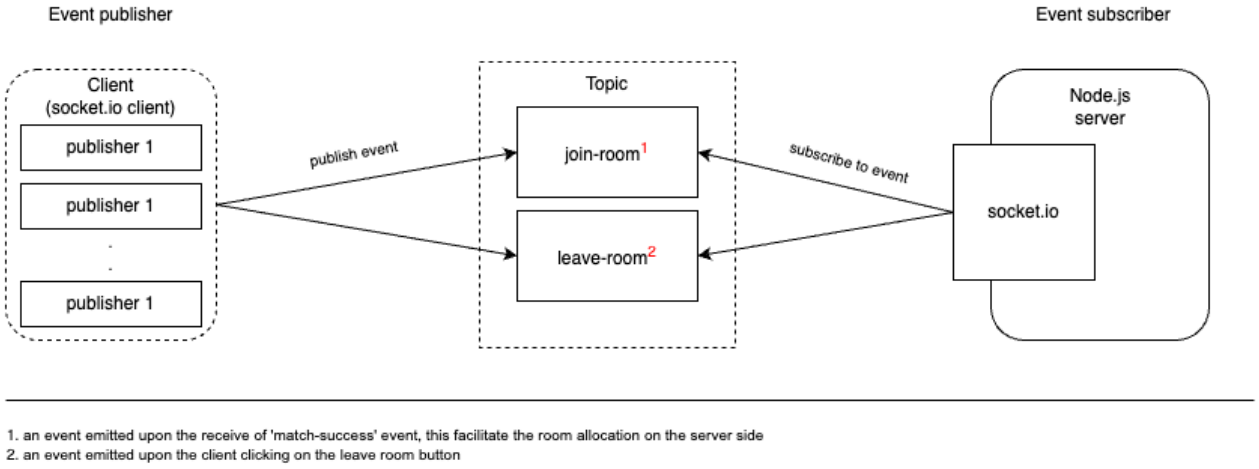


Fig 17: room-related events where the client instaces act as an event publisher, the server subscribe to these events (diagram: https://drive.google.com/file/d/1th8_IqpWgqzhsq4z_PKXyCLqH6iHwQK3/view?usp=sharing)

Triggering of database updates

The following describes how the `socketController` is interacting with update and deletion of pending match data in the database.

```
socket.on('match-easy', async (data) => {
    const user = await pendingMatchController.getAvailableMatch('easy');

    // if no match --> add to db
    if (user === null) {
        pendingMatchController.addPendingMatchEasy(socket.id, data);
    } else {
        const currentSocketId = user.dataValues.socketid;

        let question;

        //retrive easy question by sending a GET API to question-service
        axios.get('http://localhost:8002/api/questions/?level=easy')
            .then(response => {
                question = response.data;
                // emit succcess event to the matched users
                io.to(socket.id).emit('match-success', currentSocketId, socket.id, question);
                io.to(currentSocketId).emit('match-success', currentSocketId, socket.id, question);

                // else --> match and delete
                pendingMatchController.deleteMatchByDifficulty('easy');

            })
            .catch(error => {
                console.log(error);
            });
    }
});
```

1
2
3

Fig 18: Code snippet on database updates

This is a lower-level implementation details which also align with the matching events sequence diagram above. We will use the example of a request to match with ‘easy’ as the level of difficulty.

1. Upon request for a match with ‘easy’ as the level of difficulty, the `*getAvailableMatch(*difficulty)` of `pendingMatchOrm` is called.

2. This checks whether there is a pending match user of the same level of difficulty (labelled as 1 in Figure 18).

3. If there is no pending match with the same difficulty, there is a match with the same difficulty level to the requested match. The `*addPendingMatchEasy*(*socketid*, *data*)` is called to add the pending match to the database.

4. If there is an existing pending match with the same difficulty, the existing pending match is deleted from the database (labelled as 3 in Figure 18).

```
92          // no match found after 30s ends
93          socket.on('no-match-found', () => {
94      4       pendingMatchController.deletePendingMatchById(socket.id);
95          });
96
97          // pending match is cancelled before 30s ends
98      5   socket.on('match-cancel', () => {
99              pendingMatchController.deletePendingMatchById(socket.id);
100         });
```

Fig 19: Code snippet on database updates

5. Similarly, `*deletePendingMatchById*(*socketid*)` is called when there is no successful match within 30 seconds (labelled as 4 in Figure 19) or when the user requests to cancel matchmaking before the 30 seconds end (labelled as 5 in Figure 19).

6.5.2 Chat Service

Similar to Matching Service, Chat Service utilises *socket.io* for real-time bi-directional client-server communication.

<u>Decoupling of microservices: decoupling socket instance from matching-service</u>

```
11      // get socket
12      const { getSocket } = useContext(SocketContext);
13      let socket = getSocket();      client-side, chat service
```

Fig 20 Code snippet showing decoupled socket instance

To decouple Matching Service and Chat Service, new socket instances are instantiated for each connected client inside the chat service. The `*roomId*` is being used to join the newly instantiated socket instances together.

```
8       // get roomId
9       const roomId = props.roomId;
10
11      // get socket
12      const { getSocket } = useContext(SocketContext);
13      let socket = getSocket();
14
15      const [newMessage, setNewMessage] = useState("");
16      const [messages, setMessages] = useState([]);
17      const [arrivalMessage, setArrivalMessage] = useState(null);
18      const scrollRef = useRef();
19
20      useEffect(() => {
21          socket.on('connect', () => {
22              console.log(socket.connected);
23          });
24
25          socket.emit('join-chat-service', roomId);
26      }, []);
```

Fig 21 shows how roomId is used

The following activity diagram captures how messages are communicated to specific users and the communication between client and server in emitting messages to users.

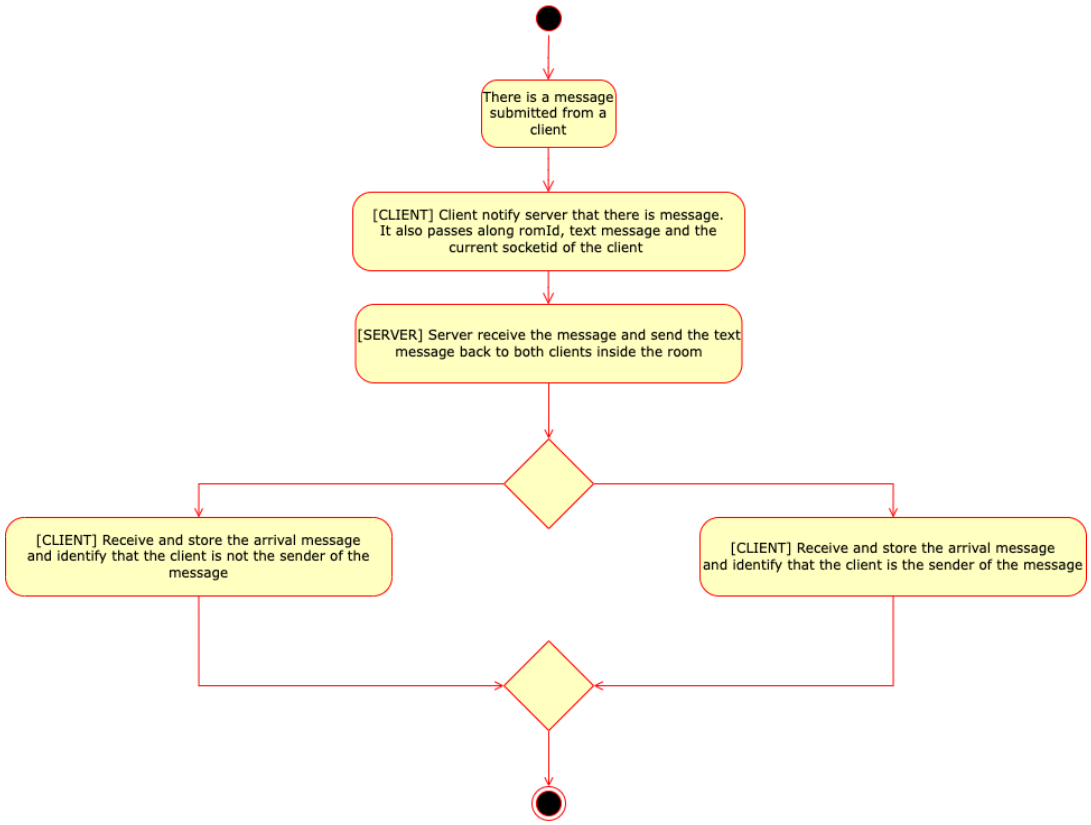


Fig 22 Activity Diagram for how messages are communicated between CLIENT and SERVER

The following is a sequence diagram capturing the interaction between the Client, Server and the database upon a user request to be matched with a specified level of difficulty.

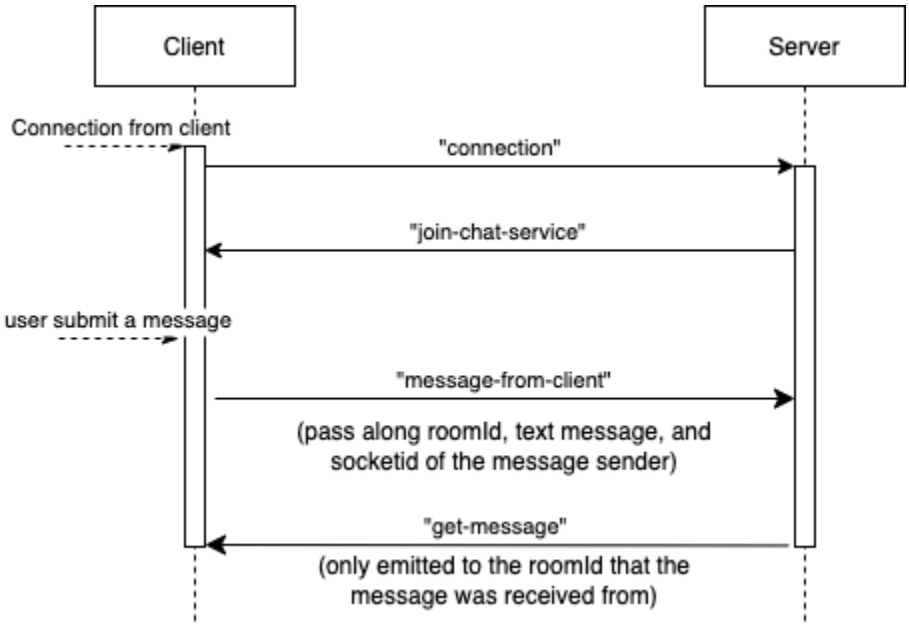


Fig 23: Sequence Diagram capturing interaction on a match request

Application of pub-sub design pattern

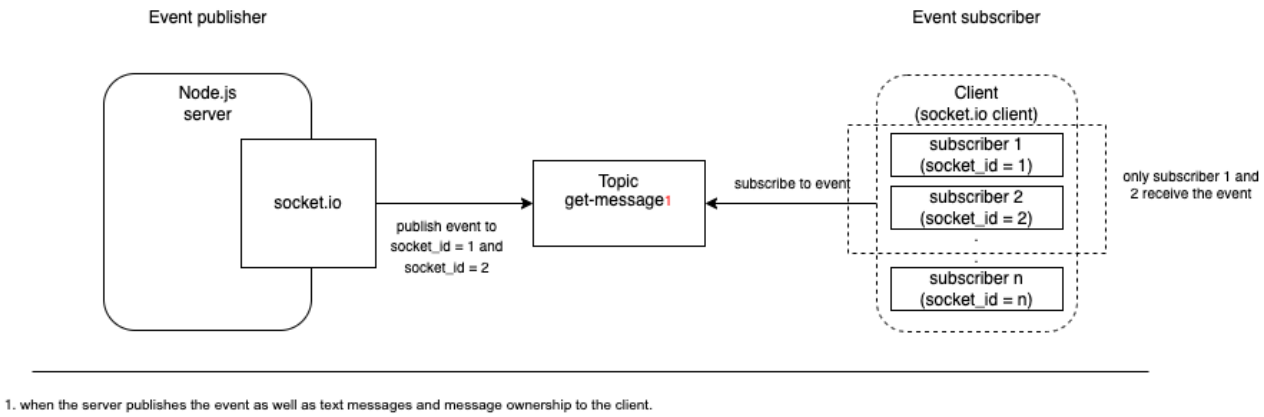


Fig 24: the server instance acting as an event publisher, socket.io clients subscribe to these events

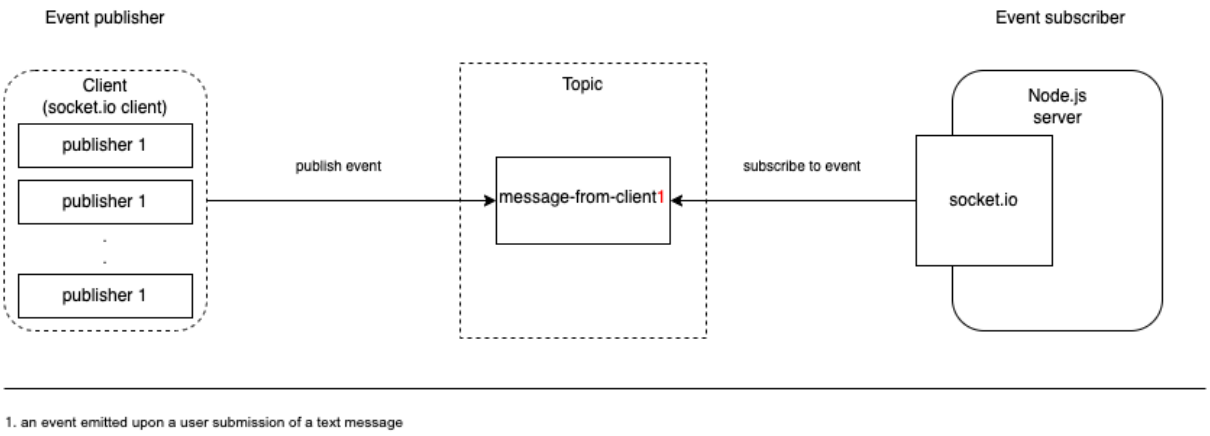


Fig 25: the server instance acting as an event publisher, socket.io clients subscribe to these events

Similar to the matching-service, chat-service applies the pub-sub design pattern where both client and server acts as event publisher and subscriber to certain events.

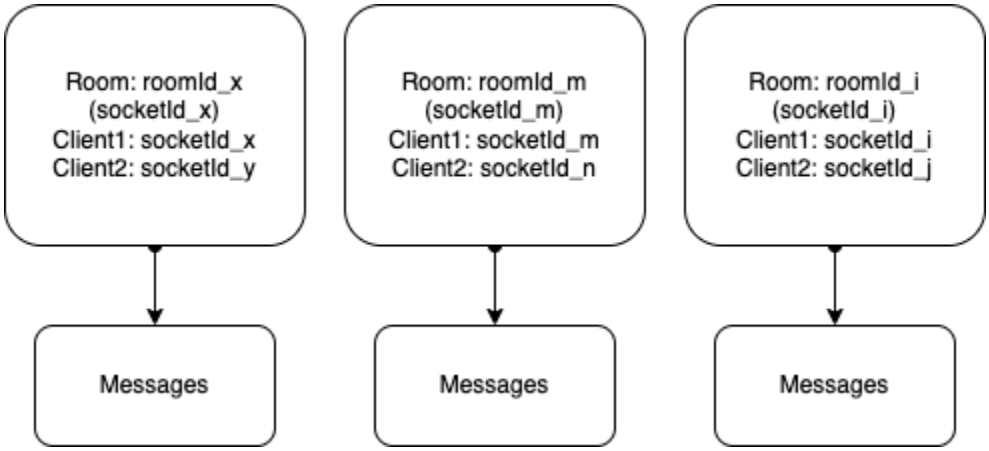


Fig 26: Pub-Sub Pattern in Chat Service

All 'rooms' in PeerPrep are identified with a unique *socketId* based on the *socketId* assigned to the first pending match users among the 2 users in the room.

The *socketid* of each user can still be accessed within the room. This allows the implementation to identify the owner of the message by comparing the socket id instance of the client and the socket id of the room, facilitating the implementation of the chat-service in identifying who the message belongs to as messages received by the clients are emitted by the server.

6.5.3 User Service

The User Service is responsible for the management of user-account related information. The main features implemented under User Service are:

1. Create Account
2. Login
3. Logout
4. Change Password
5. Delete Account

User Model

The information that is required for creating an account with PeerPrep are namely, *username* and *password*. Thus, a user is modelled as shown below:

- *username*: unique string identifying the user
- *password*: hashed and salted password string to authenticate user

Design

The user-service follows the MVC design pattern to implement these functionalities.

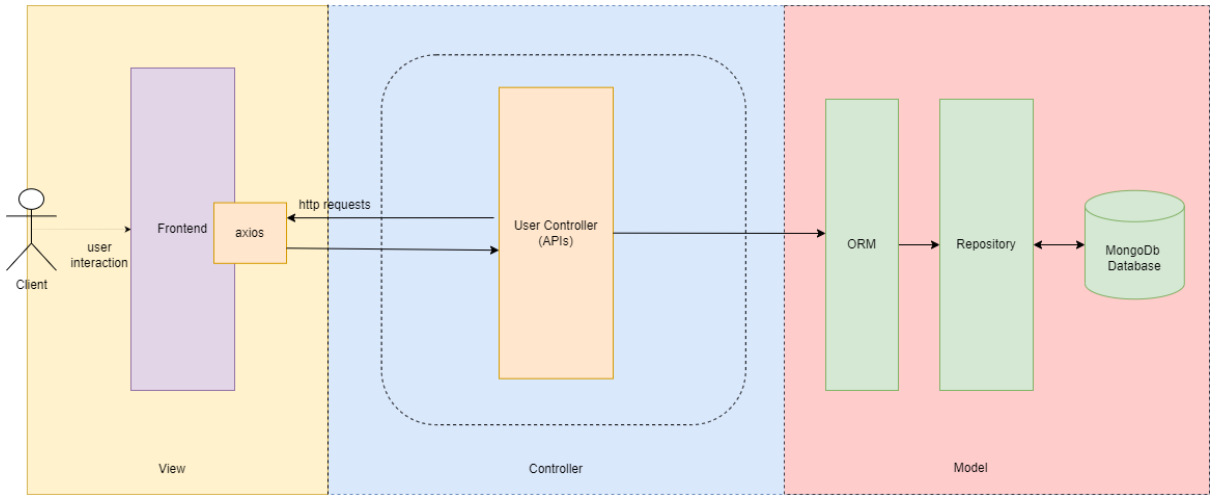


Fig 27: Architecture diagram of user-service, showing MVC pattern

As explained in Section 6.3.1, the MVC model is chosen as the main functionalities involve CRUD operations that interface with a database.

With reference to Figure 27, the Repository layer encapsulates methods that connect to MongoDB and interfaces with the *user model*. The ORM layer contains processing logic related to database operations. The User controller layer provides APIs to the functionalities of user-service and contains the logic for handling API calls.

Furthermore, to adhere to Separation of Concerns principle, utility-related methods, such as bcrypt hashing and JWTtoken verification functions, are also abstracted into a separate Services.js file.

Design of User Authentication

As the User Service is responsible for authentication of users, it also manages access into the application's main features and API.

We utilise jwt tokens for authentication purposes. As JWT is self-contained, all the necessary information for authentication is stored tokens, reducing the need for a separate authentication server/database to keep track of the authentication state of users.

In our application, JWT tokens are stored in httpOnly cookies. This allows us to send the tokens over to the server via HTTP, while preventing JavaScript code in the browser from reading or writing to our tokens. This design choice of storing tokens in httpOnly cookies over alternatives such as localStorage, sessionStorage and normal cookies is deliberate as it helps to maintain the integrity of the JWT tokens, in turn improving the security of our application.

There are 2 kinds of JWT tokens employed in our application: *Access Tokens* and *Refresh Tokens*. Upon a successful login, an *Access Token* and *Refresh Token* will be stored in the user's browser in their respective httpOnly cookies. *Access Tokens* have a short lifespan of 30 seconds after which they will expire. *Refresh Tokens* are set to last longer with an expiry time of 30 minutes.

As JWT tokens cannot be invalidated directly at will due to their stateless nature, the next best alternative to achieve such a functionality would be to set a short expiry time for each *Access Token.* In contexts where an *Access Token* has expired, but the user does not wish to be logged out yet, the *Refresh Token* is then used as a special token to obtain/refresh a new *Access Token*, provided the *Refresh Token* is still valid. The new *Access Token* again has an expiry time of 30 seconds.

When a user is logged out, the user's *Access Token* is blacklisted and their Refresh Token is removed from a whitelist of *Refresh Tokens* on the server. On the frontend, both httpOnly cookies are wiped from the browser. This prevents the *Access Token* of a logged out user from being used for authentication again, and also prevents their *Refresh Token* from generating new *Access Tokens*.

Login and Logout - *Authentication procedure*

The following sequence diagram depicts the authentication procedure involved in login and logout of users, as well as the authentication checks while the user is using the application.

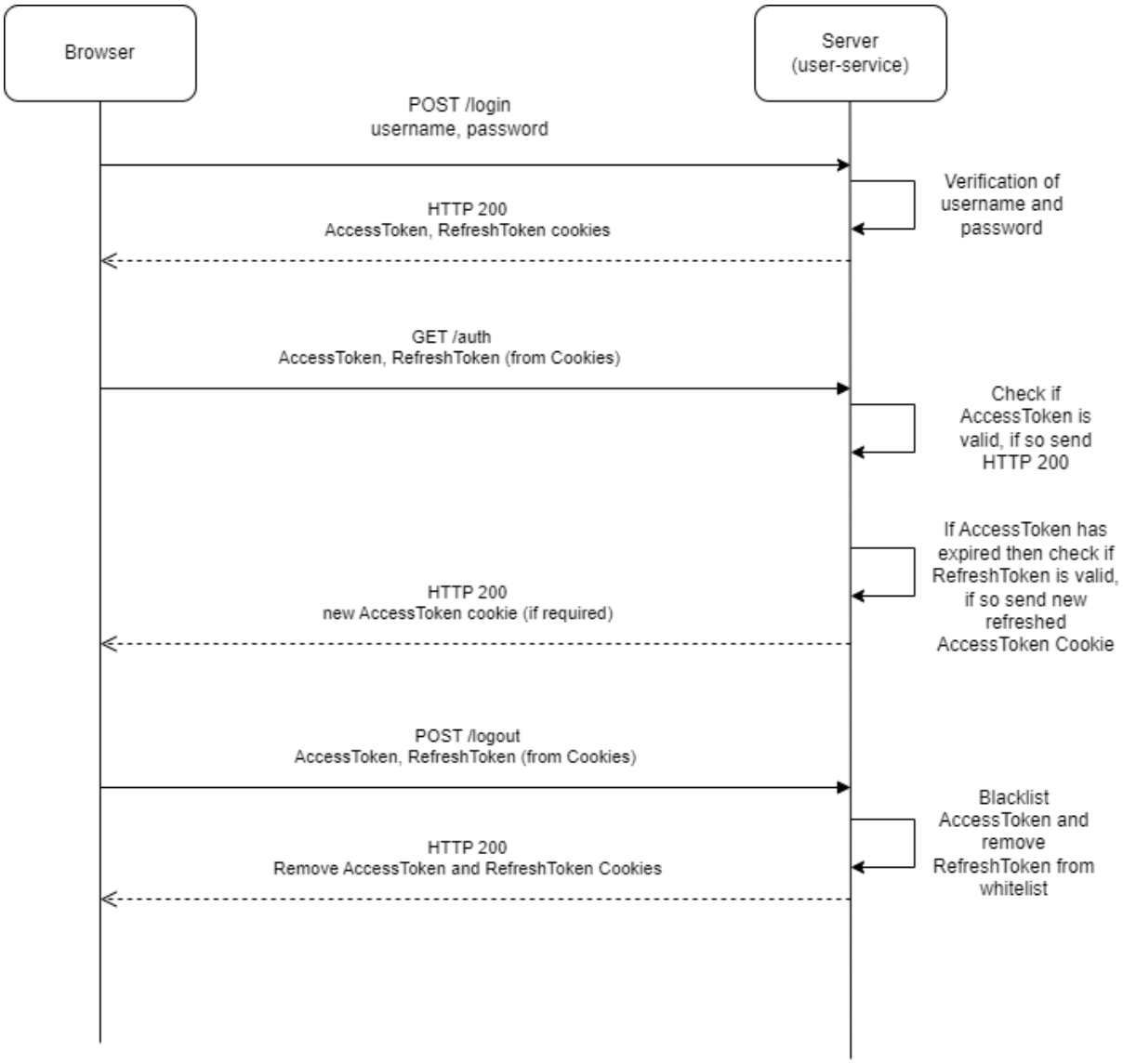


Fig 28 : Authentication procedure involved in login, performing authentication checks and logout of users

Password Strength

To improve user account security, we added a basic password strength requirement. A user's password must contain at least one alphabet, one digit, and be at least 8 characters in length. Users are free to add special symbols to strengthen their password, but it is optional. This check is implemented using regex:

```
24  export function verifyPasswordStrength(password) {
25      const pwRegex = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d@$!%*?&]{8,}$/;
26      return password.match(pwRegex) !== null;
27  }
```

Fig 29 : showing code snippet of password strength verification.

6.5.4 Collaboration Service

As the collaboration service allows users to edit the same code simultaneously at the same time, conflict handling is necessary. We adopted a conflict-free replicated data type (CRDT) so that inconsistencies will be handled and resolved by the algorithm.

Frontend implementation

An API call to YJS, a modular framework and high-performance CRDT for building collaborative applications that sync automatically, will be made when a pair of matched users enter the room page. We used CodeMirror as a third-party editor and bind it to a YJS document. To make the editor syncable and reflect the changes of the other user, we established a local connection at signalling server ‘localhost:3000’ between the two users through a WebRTC provider.

Awareness and presence

Awareness and presence features (linked to FR4.1) refers to the ability to know what the other user is typing / which part of the document he is at. We used the provider’s implemented Awareness CRDT to render remote cursor locations in the editor.

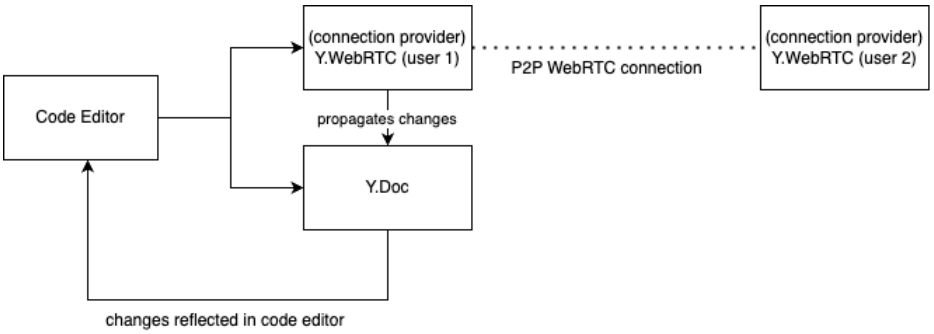


Fig 30 : Collaboration service architecture

6.5.5 Question Service

The question service stores a question bank and is responsible for retrieving questions based on the selected difficulty level.

Question Model

A Question is modelled as follows:

- *QuestionTitle*: Unique string identifying the question.
- *QuestionBody*: Formatted string with new line and quotation marks escaped characters to represent the main body information of a question. This includes the description of the question, example inputs and outputs, and constraints.
- *QuestionDifficulty*: String representing the difficulty of a question (i.e. “easy”, “medium”, “hard”).
- *QuestionImage*: base64 encoded string representation of an image pertaining to a question, if applicable.

Design of Question Bank

As our application does not have roles (other than user), we do not have an admin console equivalent to perform insertion of questions into the question bank, which is hosted on MongoDB. As such, we achieve the population of the question bank by importing JSON files containing our questions into MongoDB via the mongoimport command provided by MongoDB Database Tools.

The omission of an admin console to perform CRUD operations on the question bank was a deliberate design choice, as multiple roles (such as admin and user) and the extension of the question bank to contain more than the predefined 60 questions by our development team is not within the scope of our project, at least in its current iteration.

Additionally, we have researched how to perform web-scraping on LeetCode website to automatically gather the questions, but we decided to invest the limited time and effort the development team had into developing other essential aspects of the application.

Design

Similar to User Service, Question Service also uses an MVC pattern as it involves retrieval of data from a database. It also facilitates extensibility when we want to extend the functionalities of Question Service.

API

`question-service` provides 2 APIs for retrieving questions:

- **`GET /api/questions/`** with query param '**`?level={VALUE}`**' can be used to retrieve a question randomly based on the given difficulty level. If level is not provided, the API will return a random question from the question bank
- **`GET /api/questions/generateNew/`** with query param '**`?level={VALUE}`**' and requires input of 'currQuestionTitle' in the request body. This API returns a random question of the given difficulty level that is not the same as currQuestionTitle. If level is not provided, the API will return a random question from the question bank

Design of Question Display Process

To generate a question to be displayed in the correct room on the frontend, the server socket in Matching Service interacts with Question Service to retrieve the question.

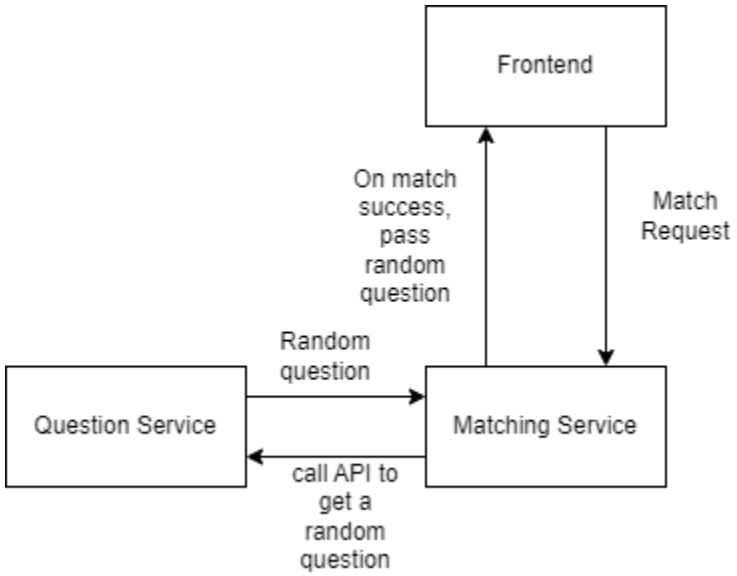


Fig 31: Diagram showing flow when displaying a question on match success

With reference to Figure 31, the server socket in Matching Service will make an API call to Question Service to retrieve a question when a successful match is found. The question is then passed to the client socket in the room in the frontend. The retrieved question is then formatted and rendered in a component on the frontend. Question retrieval also occurs when the user chooses to refresh the question.

6.5.6 History Service

The History Service is responsible for maintaining a record of a user's attempted questions. The main features implemented in History Service are:

1. Create and Update user's record of attempted questions
2. Retrieve user's record of attempted questions

History Model

A History Model is modelled as follows:

- *username*: Unique string identifying the user.
- *records*: Array of objects containing "questionTitle" and "questionDifficulty" fields. These objects represent the attempted questions.

Design

Similar to User Service, History Service also uses an MVC pattern as it involves storage and retrieval of data from a database. It also facilitates extensibility when we want to extend the functionalities of History Service in the future.

API

`history-service` provides APIs for retrieving and updating the history records of a given user:

- **`GET /api/history/{username}`** with query param '**`?level={VALUE}`**' retrieves all questions attempted by a user (*username*) based on the given difficulty level (*level*). If level is not provided, the API will return all questions attempted by the user, inclusive of all difficulties.
- **`PUT /api/history/{username}`** updates the user's history with an attempted question, which is supplied in the request body (**`{ "questionTitle": "...", "questionDifficulty": "..."}`**). If the user's record does not exist, it will create a new record for the user.

Design of updating user record of attempted question process

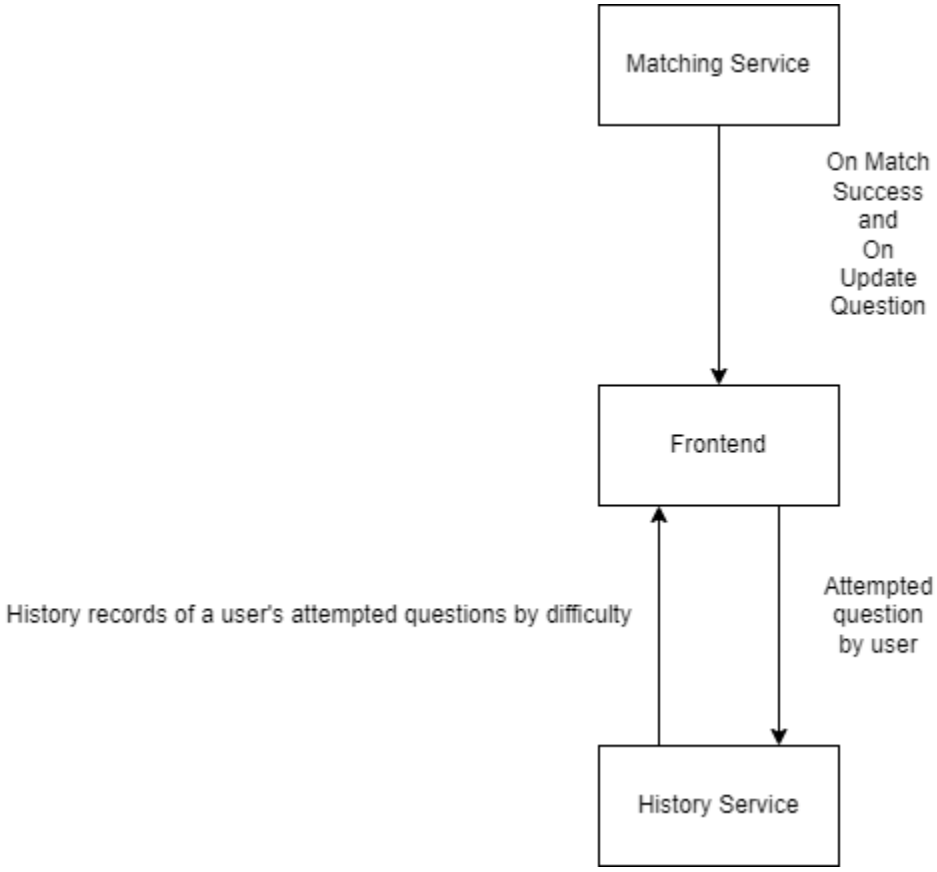


Fig 32: showing the flow when updating a user's record of attempted questions

When there is a new match, the "match-success" event triggers the Frontend to pass the initial question and username to History Service for updating of records. This flow of events then occurs again on each "update-question" event, which happens when the user requests for a new question whilst still being in the same room.

6.6 Frontend Design

We adopted a component-based design structure for our frontend. This allows our React components to be easily reused across different parts of the application, ensuring consistency, reducing duplicate codes and making it easier for changes to be made in the future.

6.7 Other Design Considerations

6.7.1 Security

We opted to use JWT authentication to fulfil the security aspect of our application, with the primary reason being JWT is stateless. As JWT is self-contained, all the necessary information for authentication is stored in the token, without a need for a separate authentication server/database to keep track of the authentication state of users. This capability of JWT allows us to bypass the need for querying a database to authenticate a user for every api call.

6.7.2 Usability

We ensured a consistent, fun and user-friendly user interface throughout the application.

First-time users should be able to navigate through the application easily, through the buttons on the page as well as a top navigation bar. Section 6.8 details out the entire user flow of the application from start to end. We used the MUI (Material-UI) component library and came up with a colour scheme to ensure a consistent interface for users to interact with.

Confirmation dialogs were added for significant events (such as upon clicking the button to leave the room page, account deletion) to reduce user errors, especially for first-time users who may not be familiar with the application.

Error checks were added where necessary to guide the user to enter correct details, such as when the user retypes a different password. Also, notification toasts were added upon success events such as a successful change of password, to show confirmation to users while not interrupting the user experience.

6.7.3 Extensibility

Low coupling

Our microservices are designed such that they are able to stand on their own with limited dependency from another service. This means that changes caused by one service would have very little impact on another service.

For example, our chat service is decoupled from matching service through the re-initialization of socket instances in the chat service. This means that both services are able to stand on their own. The failure of the chat service will not affect the functionality of the matching service and the entire application. There will only be missing functionality due to missing chat service upon matching, however the rest of the application would still be able to function.

Separation of Concerns

Our microservices are implemented such that each layer in the application is only responsible for one thing and should not contain any code that deals with other things.

An example is in the controller layer of matching service that is internally divided into sub-layers.

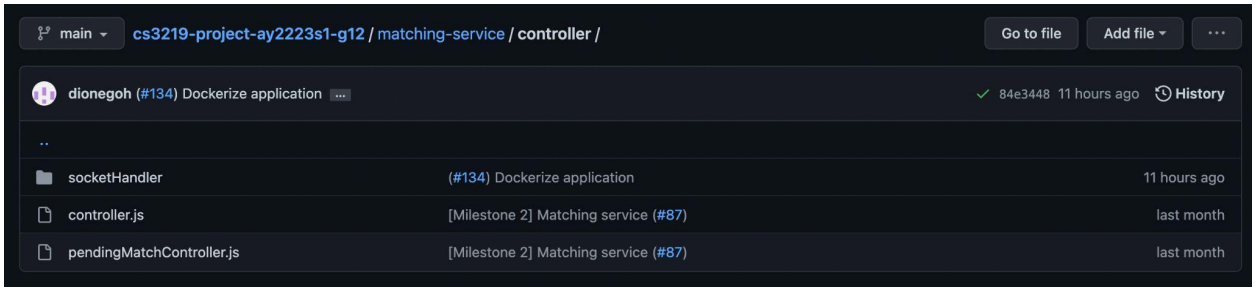


Fig 33 shows the different layers/ files in Matching Service

The `socketHandler` only deals with events from client-server communication, whereas the logical implementation of how these events will interact with the database is responsible by the `pendingMatchControlller`.

Separation of concern is also evident in the application of the pub-sub design pattern in chat service and matching service.

As data travels one-way from the publisher to subscribers, each event/topic/message type is designed such that they fulfil a simple and straightforward purpose.

Abstraction

Matching service abstraction of lower-level implementation of logic from higher-level functions so that those methods could function without having to worry about the internal mechanisms of how the lower-level functions are implemented.

An example would be how the controller in matching service is being implemented. The lower level internal mechanism of how to retrieve, update, delete user information from the database (these functions are defined in the `pendingMatchController`) is hidden away from the `socketHandler`.

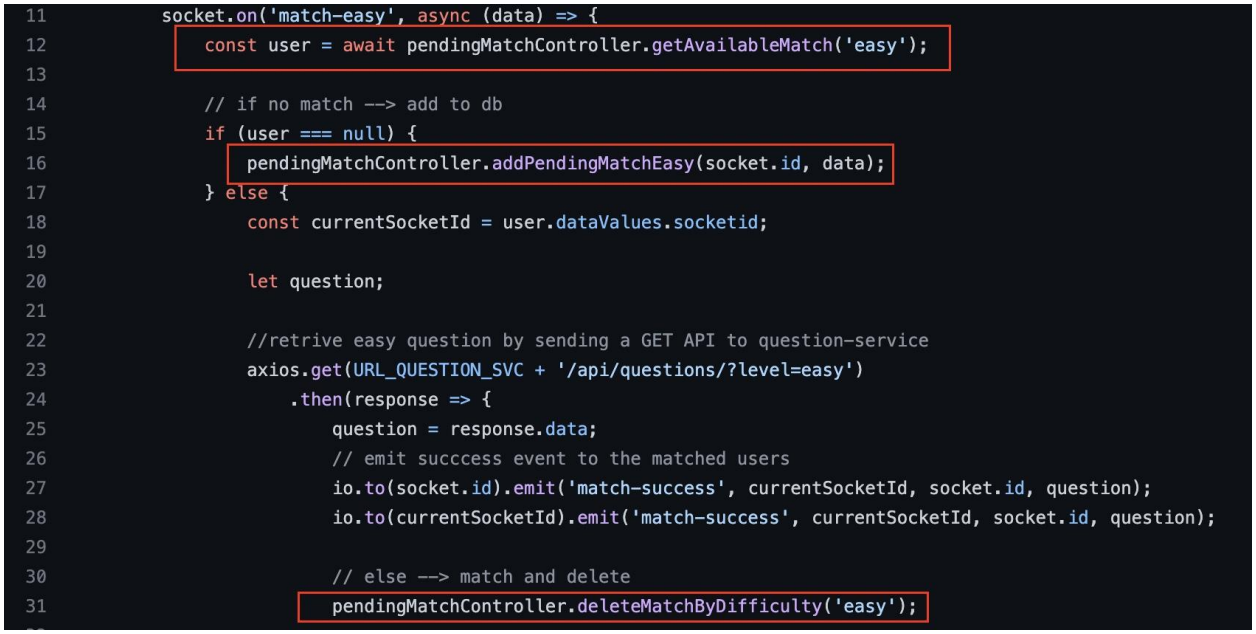


Fig 34 shows the abstraction of lower level implementation details

The `socketHandler`, which has the role to just handle the events in the client-server communication, will only need to know about the role of each lower level function but not the implementation details.

This means that we can change the implementation of these lower-level functions without having to change the implementation of the `socketHandler`.

We can also see a similar abstraction in the `pendingMatchController`. The role of `pendingMatchController` is to handle the logic related to each event in the client-server communication.

In this example, the lower level internal mechanism of how the details of the user (i.e. pending match) is being added to the database is abstracted away from the `pendingMatchController`. This means that if there were to be a change in the design of how user details are added to the database, there would not need to be a change in the implementation of `pendingMatchController`.

```
13  function addPendingMatchEasy(socketid, username) {
14      pendingMatchOrm.addPendingMatchEasy(socketid, username);
15  }
16
17  function addPendingMatchMedium(socketid, username) {
18      pendingMatchOrm.addPendingMatchMedium(socketid, username);
19  }
20
21  function addPendingMatchHard(socketid, username) {
22      pendingMatchOrm.addPendingMatchHard(socketid, username);
23  }
```

Fig 35 showing the lower level implementation details

6.8 User Flow

The following is an activity diagram of a typical user flow from the point a user logs in to the point the user logs out.

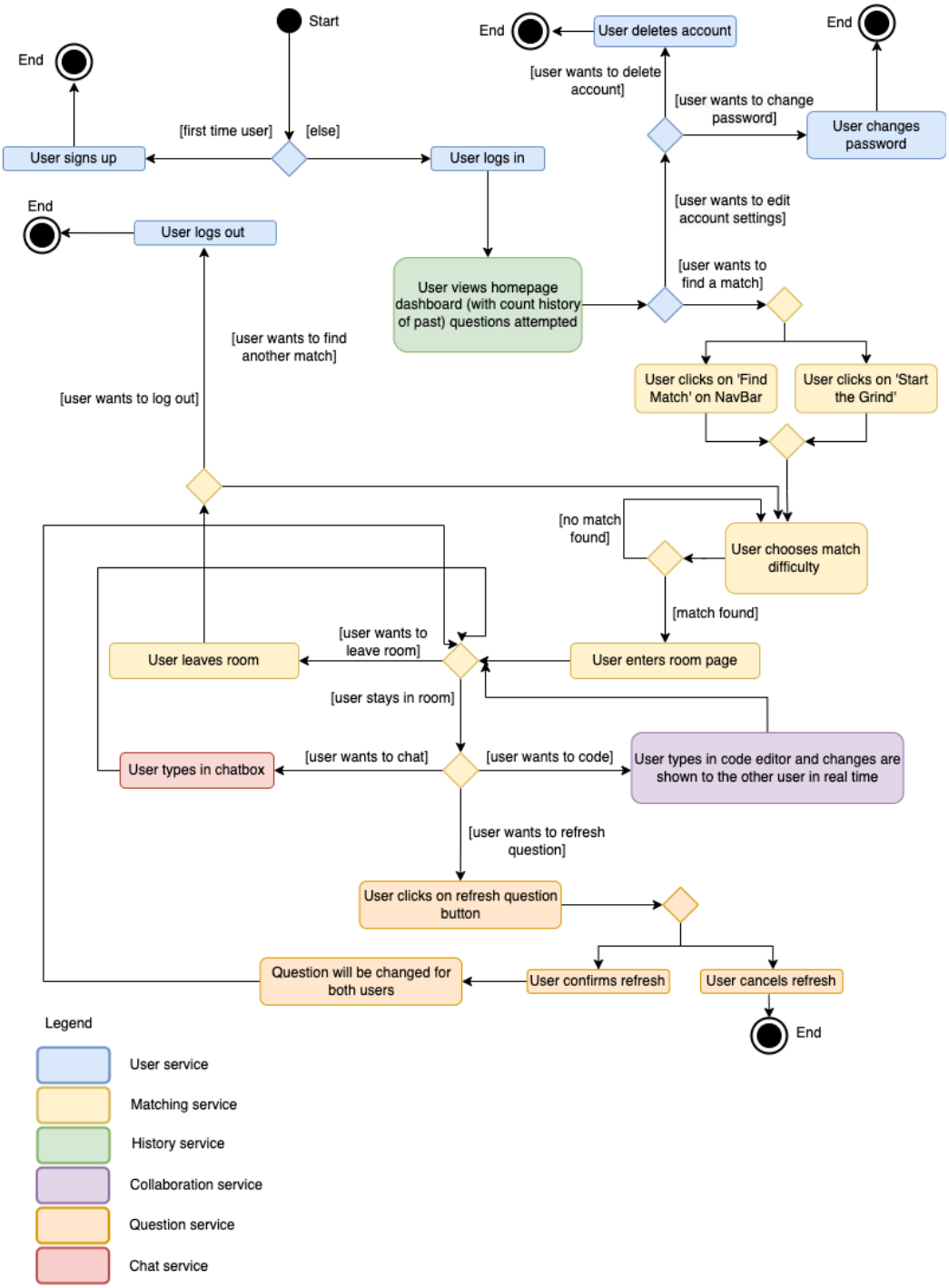


Fig 36: User flow activity diagram

6.8.1 UI Flow

This subsection describes and shows the UI flow in pictures.

<u>Sign up and Login UI</u>

Users will first be directed to the login page. For first-time users who do not have an account, they can click on the redirect text below the login button to be redirected to the sign up page.

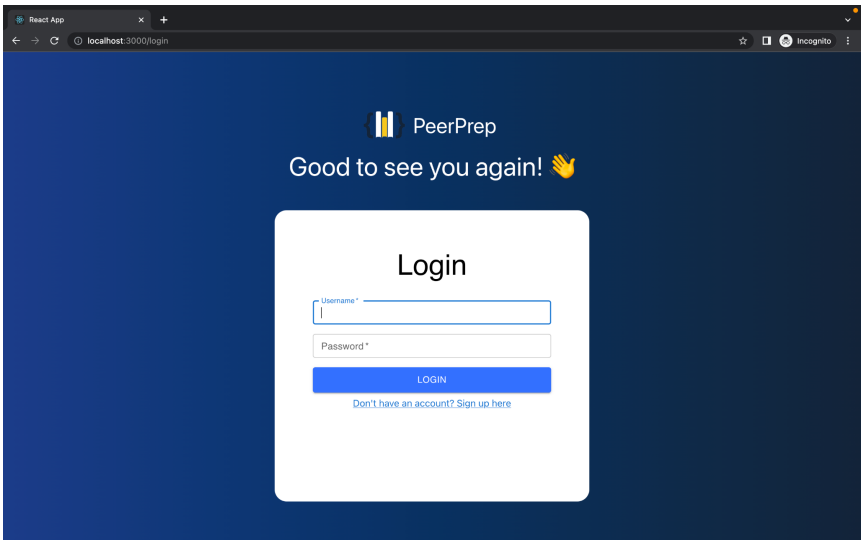


Fig 37: Login page

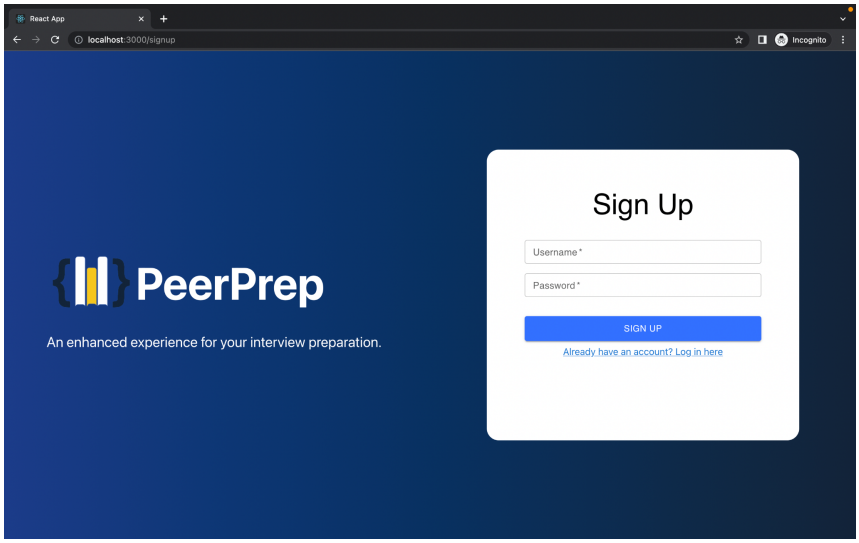


Fig 38: Sign Up page

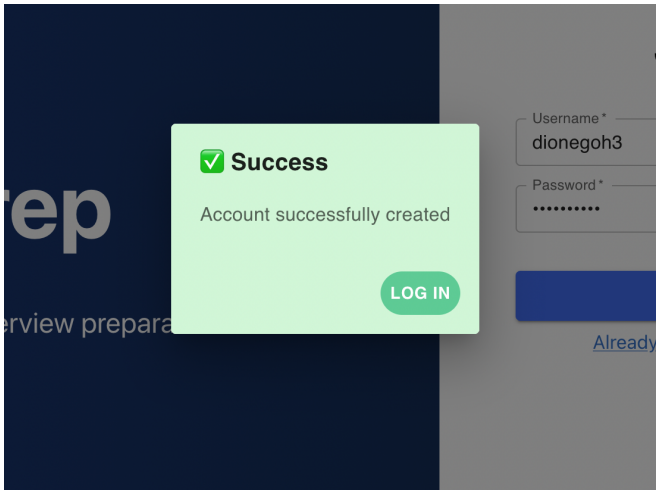


Fig 39: Successful sign up

Home page UI

The home page dashboard keeps track of how many questions per difficulty level the user has previously attempted. Users can click on `START THE GRIND` button or `FIND MATCH` text on the navigation bar to be redirected to the match page.

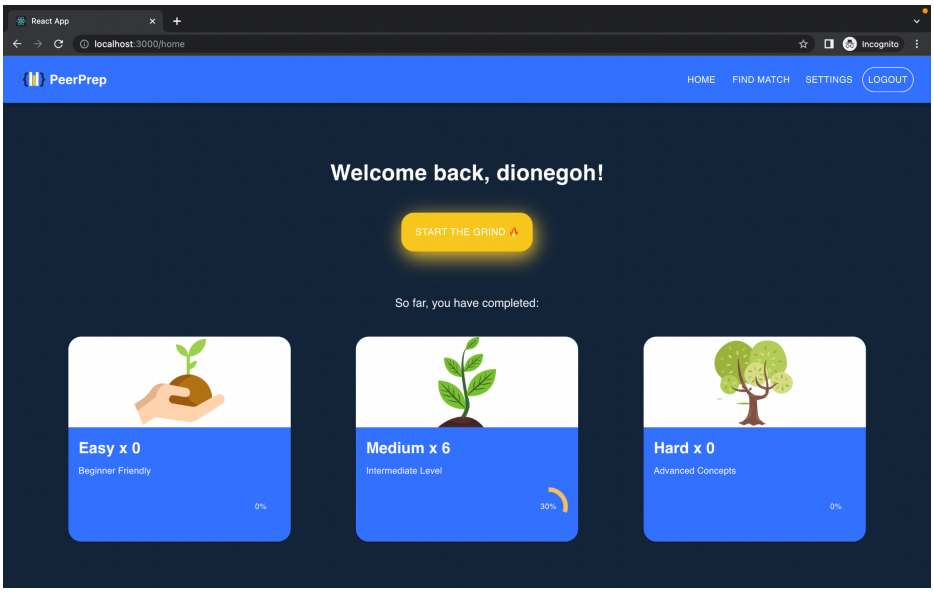


Fig 40: E.g: homepage of a user who has attempted 6 medium questions

Match page UI

Users can choose which difficulty level they want to attempt. A countdown timer which changes colour according to the time left to match will be shown as a modal overlay upon selecting a difficulty level.

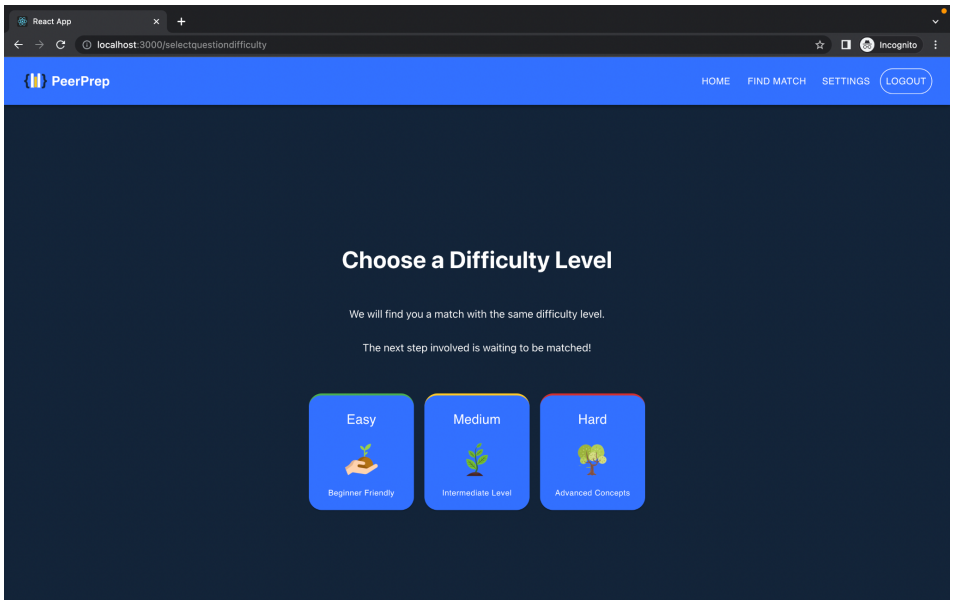


Fig 41: Match page

Countdown timer before time is up	Countdown timer after time is up

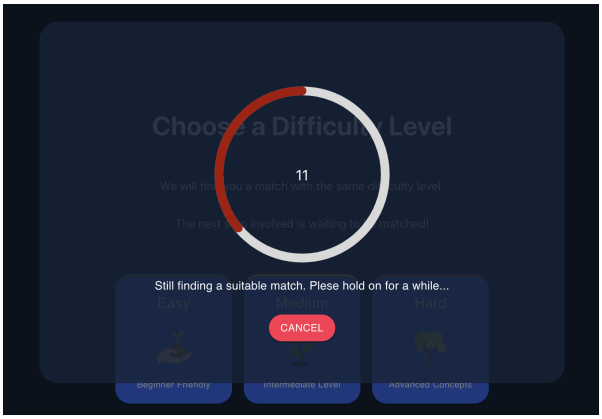
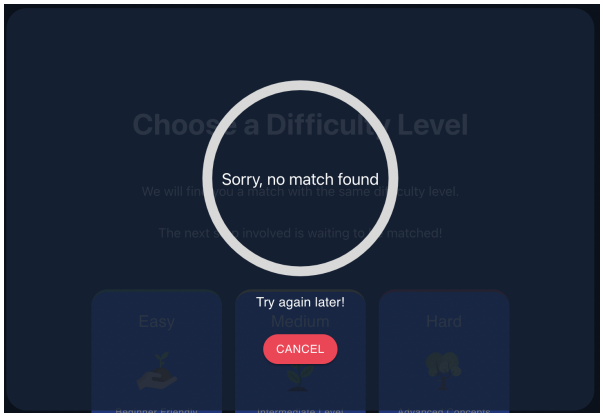

	

Fig 42: Countdown timer running	*Fig 43: Countdown timer ended*

<u>Room page UI</u>

Upon a successful match, users will be redirected to a room page. Users can chat, code, and change to another new question in the room page.

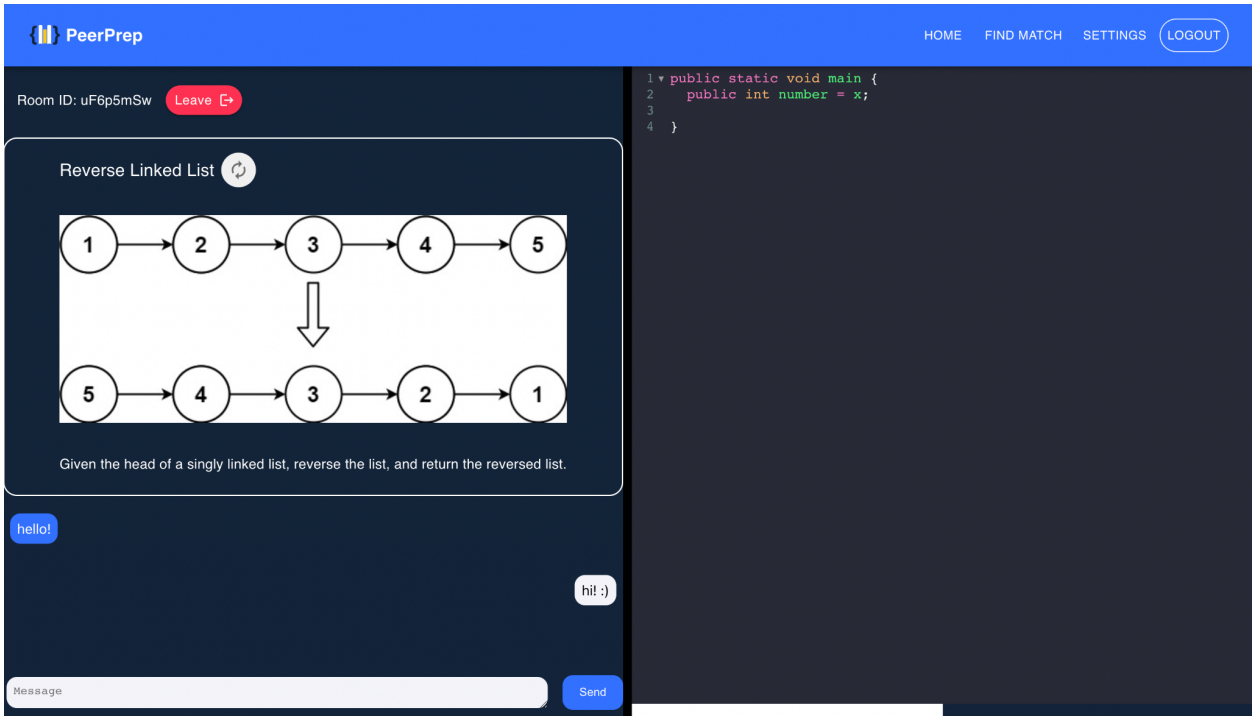


Fig 44: Room page

The code changes will be reflected in real time for both users. Additionally, a user will be able to see the cursor of the other user to know where he is editing.

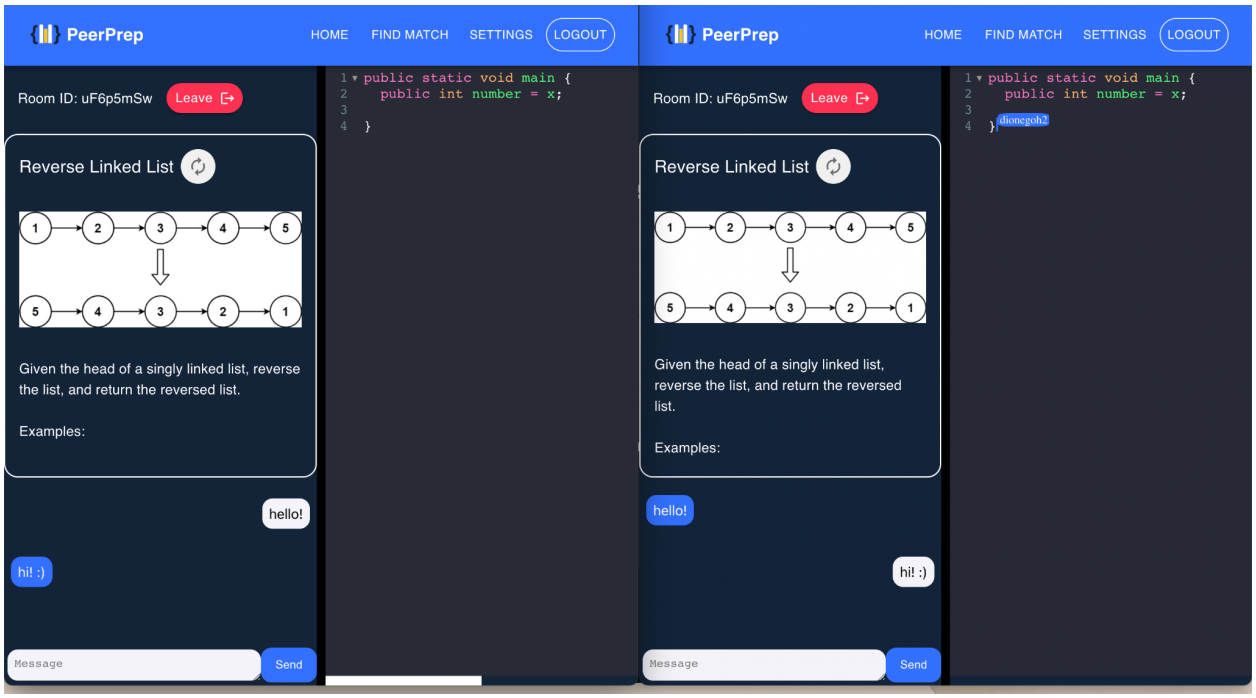


Fig 45: Code changes being reflected in real time

A user can request to change to a new question of the same difficulty level in the room by clicking on the refresh button next to the question title. A confirmation dialog will open up.

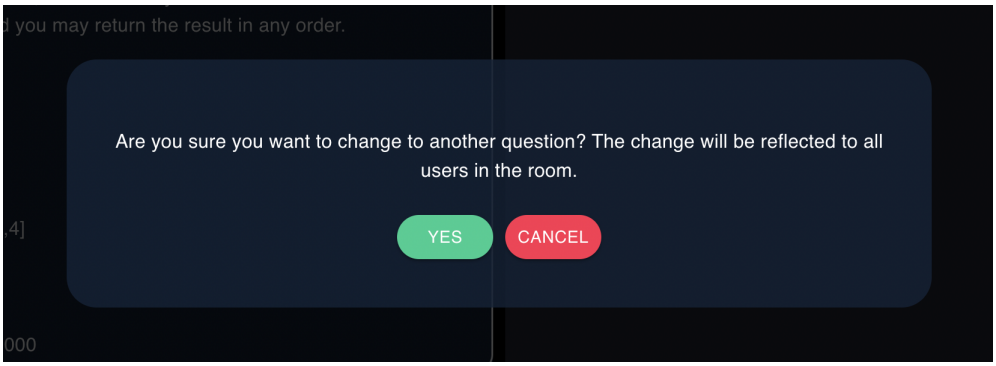


Fig 46: Confirmation modal

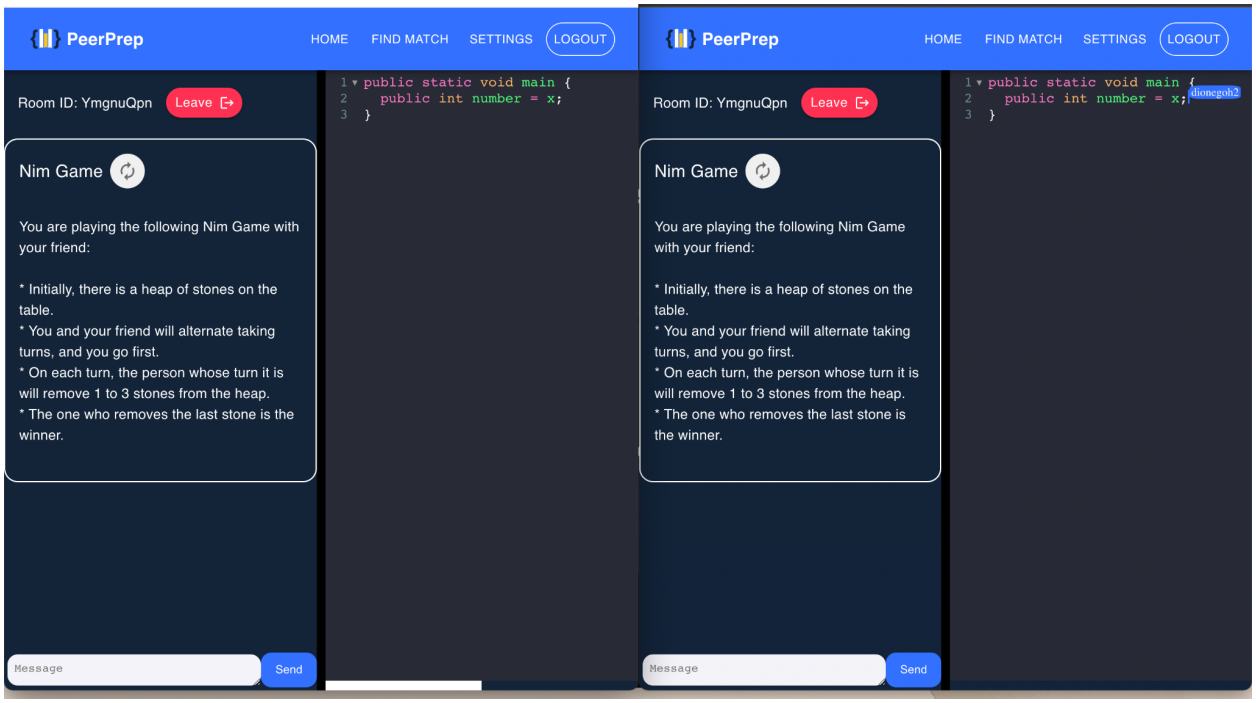


Fig 47: New question generated

A user can leave the room anytime he wants by clicking on the leave button next to the RoomId. Similar to when refreshing a question, a confirmation dialog will appear. The user can choose to proceed or cancel, and will be redirected to the match page upon the former.

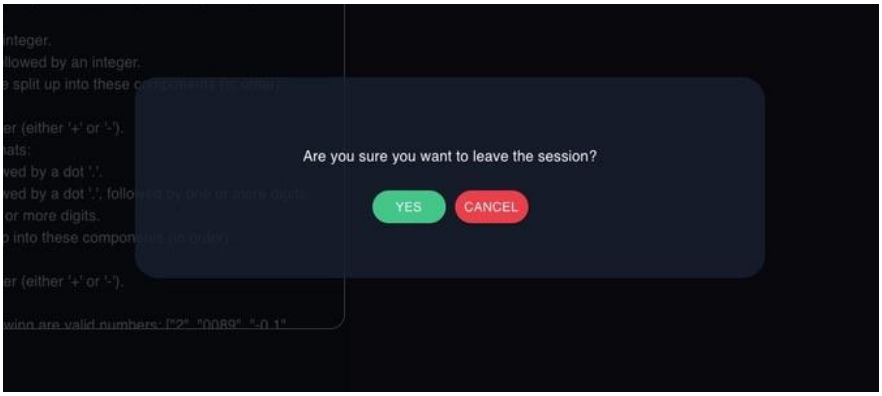


Fig 48: Leave room confirmation dialog

Account settings UI

A user can change password or delete his account via Settings on the navigation bar.

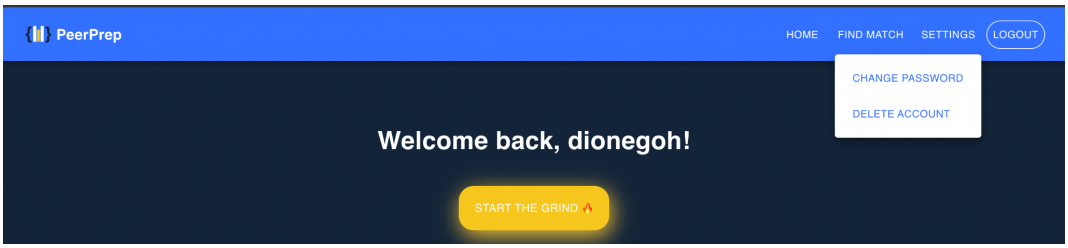


Fig 49: How a user can edit account settings

Change Password	Delete Account

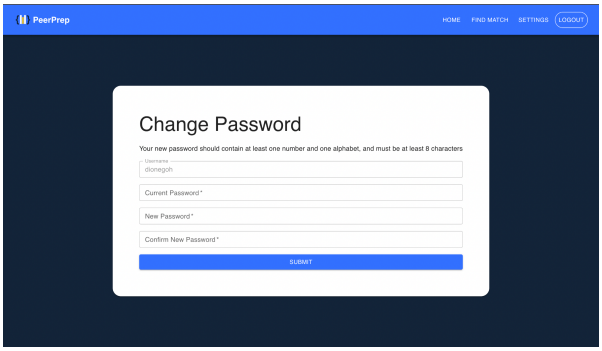
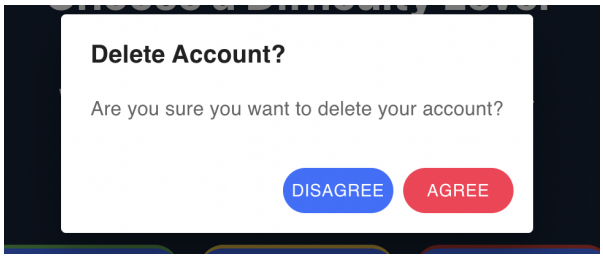

	

Fig 50: Change password UI	*Fig 51: Delete account UI*

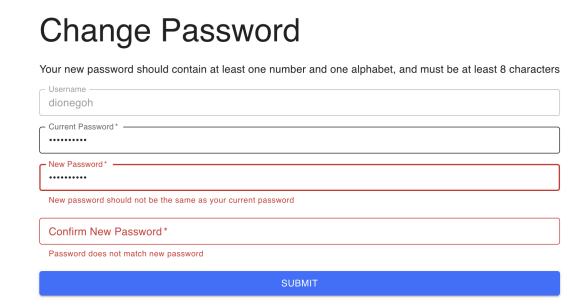


Fig 52: Error shown when new password matches old password

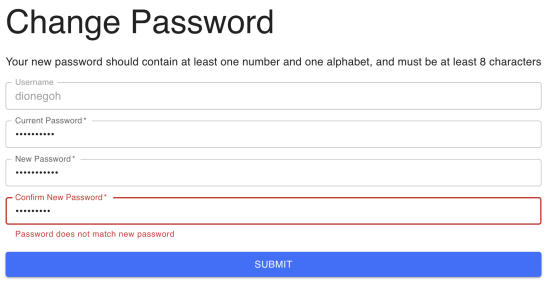


Fig 53: Error shown when retyped new password does not match new password

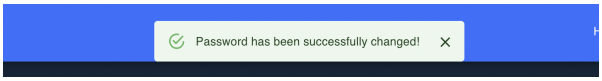


Fig 54: Toast showing successful password change

7. Future Enhancements

This section details possible enhancements to improve the experience of using PeerPrep.

7.1 Features

7.1.1 Video Chat

During an interview, one's non-verbals, such as eye contact, body posture, and facial expressions, are also important in signalling his confidence levels about his ability, and displaying one's genuine interests in the company. Thus, a video chat function can enable peers to do mock interviews with each other, on top of being able to do whiteboard-style technical interviews. This function would also be very useful in helping one prepare for a technical interview.

7.1.2 Gamification and Achievements

To better engage users and improve user retention, we can add a gamification element to PeerPrep. For example, we can have achievements for completing a certain number of questions or for doing a certain number of matches (that exceeds a defined amount of time). The latter ensures that the user adequately works on the question with his peer and prevents him from trying to game the system to get achievements. In obtaining achievements, one can obtain experience points and level up. They may also obtain badges based on the achievements gained.

7.1.3 Login using Google and GitHub

To minimise the trouble of having to remember another username and password, we can integrate with external providers, such as Google and GitHub authentication.

7.2 Deployment

Currently, our application can only run on the local machine, either through npm start, or via docker-compose.yml. To allow this application to reach our target audience, we would need to deploy the microservices and frontend application on one of the cloud service providers. Furthermore, this would also enable us to scale the microservices and frontend application to better manage the number of users using the application in the future.

7.3 Cache

We identified that performing authentication is one of the most frequent operations throughout the application. This is because we need to ensure that the application is protected at all times and thus, block any unauthorised access. When many users are active at the same time, this may lead to slow response, a longer loading time for users, and a poor user experience. Thus, to improve performance, the middleware required for authentication can be cached. For example, we can cache the blacklist of Access Tokens and whitelist of Refresh Tokens in a Redis Cache. The user's account data still remains secure in the database. As these white and black lists are currently stored in-memory, this also helps to avoid the situation where users are forcefully logged out if the user-service server crashes and/or restarts.

Upon rendering of the homepage, a GET request will be made to the History Service API to fetch all the questions that a user has previously attempted. If the user has attempted a large number of questions, there will be a longer response time to fetch the large amount of data before the home page can be rendered. In this case, a Redis Cache can be used to store the questions attempted by the user to improve efficiency of the application. The value in the Redis key-value pair (userId, questions) could be appended to whenever a new question is attempted by the user. This would improve user experience through a faster rendering time.

8. Learning Points

Before this module, our team is mostly new to microservices, CI/CD, Dockerisation, as well as other concepts taught. Thus, we felt that we have received many takeaways, both technical and non-technical, while working on this project together.

8.1 Microservices Architecture

Through the project, we learnt how different features can be decoupled into different microservices. Although we did not manage to deploy the microservices and frontend successfully, we do understand the importance of using microservices architecture, particularly for helping to scale different aspects of the application, as well as minimising resource usage.

8.2 New Technologies

It was the first time that most of us have used technologies in our chosen tech stack such as Socket I/O for pub/sub, JWT authentication and Cypress for frontend testing. Furthermore, for some of us, it was also the first time we used Node.js and React.js to develop an application. It is definitely a rewarding learning experience to learn to implement our desired features with these technologies.

8.3 Deployment

CI/CD and containerisation using Docker are also new to us. We see how using CI for testing can help to ensure that changes made do not affect the current version of the application. Although we were not successful at deploying the application with CD, through the module, we also understood the significance of making incremental changes instead of one large change.

8.4 Communication within the team

We learnt to work with one another's schedules and differing levels of skills as we develop the application. Whenever we encountered difficulties with implementation or design decisions, we consulted with one another to resolve these issues. It is important that we have open communication in the team, so we can come to a consensus about a particular design or use case for our application.

~ End of Report! Thank you for reading 🙂 ~

Appendix A: Use Cases

Use Case: UC1 - Account Creation
Actor: User
MSS:

1. User navigates to the application webpage.
2. System redirects User to the sign up page.
3. User enters his desired username and password.
4. User confirms sign up.
5. System informs User that account creation is successful.
6. System prompts User to login.
   Use case ends.

Extensions:
3a. User enters a weak password.
    3a1. System informs user that password is weak
    3a2. User enters a strong password.
    Use case resumes from step 4.

3b. Desired username is already taken.
    3b1. System informs user that username is taken
    3b2. User enters a different username.
    Use case resumes from step 4.

Use Case: UC2 - Login
Actor: User
Precondition: User has an existing account.
MSS:

1. User navigates to the login page.
2. User enters username and password.
3. System logs user in and redirects User to home page.
   Use case ends.

Extensions:
2a. User enters the wrong username or password.
    2a1. System informs the user that the username or password is incorrect.
    Use case resumes from step 2.

Use Case: UC3 - Log out
Actor: User

Precondition: User is logged in.
MSS:

1. User chooses to logout.
2. System logs User out and redirects to the login page.
   Use case ends.

Extensions:
*a. User has been inactive for 30 mins.
   *a1. System automatically logs User out.
   Use case ends.

Use case: UC4 - Account Deletion
Actor: User
Precondition: User is logged in.
MSS:

1. User chooses to delete his account.
2. System prompts for confirmation.
3. User confirms.
4. System deletes account and redirects User to login page.
   Use case ends.

Extensions:
3a. User decides not to delete the account.
   3a1. User chooses to cancel.
   Use case ends.

Use Case: UC5 - Change Password
Actor: User
Precondition: User is logged in.
MSS:

1. User chooses to change his password.
2. System requests User for his current password.
3. User enters his current password.
4. System requests User for his new password.
5. User enters his new password.
6. System requests User to type his new password again.
7. User enters his new password again.
8. User confirms change password.
9. System informs User that the password change is successful.
   Use case ends.

Extensions:

5a. User enters a weak new password.

5a1. System informs user that new password is weak

5a2. User enters a strong new password.

Use case resumes from step 6.

5b. User's new password matches his current password.

5b1. System informs User that the new password should not match his current password.

5b2. User enters a different new password.

Use case resumes from step 6.

7a. The second new password does not match the first new password.

7a1. System informs User that the two new passwords do not match.

Use case resumes from step 7.

8a. User enters the wrong current password.

8a1. System informs User that the current password is wrong.

Use case resumes from step 3.

Use Case: UC6 - Find Match

Actor: User

Precondition: User is logged in.

MSS:

1. User navigates to the find match page.
2. User chooses his desired question difficulty from 'easy', 'medium' and 'hard' levels.
3. System displays a waiting screen while matching is in progress.
4. System successfully finds a match and redirects User (and matched User) to join the same room (UC7).

   Use case ends.

Extensions:

3a. User decides not to continue matchmaking.

3a1. User chooses to cancel the matching.

3a2. System cancels matching and closes the waiting screen.

Use case ends.

3b. System could not find a match in 30s.

3b1. System informs User that a match could not be found.

3b2. User acknowledges.
Use case ends.

Use Case: UC7 - Joins Room
Actor: Users
Precondition: Users are logged in and matched together.
MSS:

1. Users join the same room.
2. System displays a random question of the chosen difficulty level.
3. Users write code to answer the question in the code editor (UC9).
4. Users chat with each other in the same room (UC10).
   Use case ends.

Use Case: UC8 - Leave Room
Actor: User
Precondition: User is logged in and in a room.
MSS:

1. User chooses to leave the room.
2. System prompts for confirmation
3. User confirms.
4. System redirects User to the find match page.
   Use case ends.

Extensions:
3a. User decides not to leave the room.
3a1. User chooses to cancel.
Use case ends.

Use Case: UC9 - Code Collaboration
Actor: User A and B
Precondition: Users are logged in and in the same room.
MSS

1. User A enters a code snippet into the code editor on the room page.
2. System syncs the changes and reflects them onto User B's screen.
3. User B sees where User A is editing.
   Use case ends.

Extensions:
*a. User B can be the one making changes in the code editor.

*a1. User B enters a code snippet into the code editor on the room page.
*a2. System syncs the changes and reflects them onto User A's screen.
*a3. User A sees where User B is editing.
Use case ends.

Use case: UC10 - Chat
Actor: User A and B
Precondition: Users are logged in and in the same room.
MSS:

1. User A sends a message.
2. User B receives the message.
   Use case ends.

Extensions:
*a. Sender and recipient of messaging can swap.
*a1. User B sends a message.
*a2. User A receives the message.
Use case ends.

Use Case: UC11 - Refresh Question
Actor: User A and B
Precondition: Users are logged in and in the same room.
MSS:

1. User A chooses to refresh the question.
2. System prompts User A for confirmation.
3. User A confirms.
4. System displays a new question of the same difficulty on both User A and User B’s room page.
   Use case ends.

Extensions:
3a. User A decides not to refresh the question.
3a1. User A chooses to cancel.
Use case ends.

Use Case: UC12 - View Attempted Questions History
Actor: User
Precondition: User is logged in
MSS:

1. User navigates to the home page.
2. System displays the number of questions completed for each difficulty level.
   Use case ends.