CS3219 Final Report
PeerPrep
Team 21

Alvin Tay Ming Hwee (A0218390E)
Lee Ze Xin (A0203252X)
Li Quan (A0200451B)
Tan De Shao (A0218351L)

1. Introduction

1.1. Background

In the current day and age where more and more companies are offering high-paying tech jobs, the screening process also becomes more rigorous. One such method that is popular among companies to screen for a highly-skilled candidate is called live coding, a type of technical interview where candidates are required to solve algorithmic and coding challenges on the spot while verbalizing their thought process at the same time. With live coding, companies are able to gain a deep insight into the technical skills of a candidate and from there, gain a better understanding of how well the candidate is able to perform on the job if he/she is accepted for the role. However, to many, live coding is challenging and it would take a lot of practice for one to perform well in a live coding setting. With the lack of interviewing experience, students often face the most challenge when applying for technical roles in the technology sector. Their difficulty in performing well in a live coding setting often stems from their lack of communication skills and inability to resolve a given coding problem under time constraint.

To provide students with a platform to practice their live coding skills, the objective of this project is to develop a web application that matches two users together, giving them an avenue to practice their live coding skills in a technical interview setting.

The web application, appropriately called PeerPrep, was created by a group of **National University of Singapore (NUS)** students as a project for **CS3219: Software Engineering Principles and Patterns** curriculum.

The following sections in this document will contain details regarding the technical implementation of PeerPrep and discuss some of the design considerations the team had when implementing some of the features for the application.

1.2. Useful Links

The deployed version of the application can be found here:
https://frontend-swougile5q-uc.a.run.app/

Note: Since the deployed version is running on a free cloud service, the initial load time of the application would be slow as the server would take some time to start up.

The Postman API documentation of the project can be found here:
https://documenter.getpostman.com/view/16419990/VVBTVT2J

The GitHub repository for this project can be found here:
https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g21

1.3. Glossary

Term	Definition
JWT	JSON Web Token
CA	Certified Authority
SSL	Secure Socket Layer
MVC	Model-View-Controller
CORS	Cross-Origin Resource Sharing
STUN	Session Traversal of User Datagram Protocol Through Network Address Translators
TURN	Traversal Using Relays around NAT

1.4. Project Overview

This section describes an overview of PeerPrep and provides an insight on the overall structure of the project and the approach the team used to implement different parts of PeerPrep.

1.4.1. Development Approach

PeerPrep can be broken down to different “services” where each service represents parts of the overall application that provides functionalities/features that are disjoint from one another. The different services and the key features they provide to the application are shown below:

Services	Features
Frontend	- Provides an interactive UI for users of the application to interact with.
User Service	- Handles the authentication of users. - Stores user information. - Stores the user’s attempted question history.
Matching Service	- Matches 2 users and redirects them to a room simulating a live coding environment.
Collaboration Service	- Handles the real-time collaboration between 2 users.
Question Service	- Provides an interface to fetch questions based on the difficulty selected by the user.
Compile Service	- Handles the compilation of code submitted by the user.

Since the application can be split up into separate disjoint services, with each service being able to independently function as an application by itself, the team developed PeerPrep by following the microservice architectural pattern. With this architectural pattern, development of PeerPrep can be done concurrently as there is little to no dependency between services.

1.4.2. Project Management

Our team adopted the Agile software development methodology, whereby our development process is split into 3 major sprints with a milestone at the end of each two-week sprint.

Before the start of each sprint, we conduct a round of deliverables planning, team retrospective, and assigning of tasks/issues. Those activities have helped us be more prepared for the upcoming sprints and have clearer short term goals in mind.

We also perform weekly sprint meetings to catch up on each other's progress, review pull requests, and conduct testing to ensure we have a working product at the end of each week.

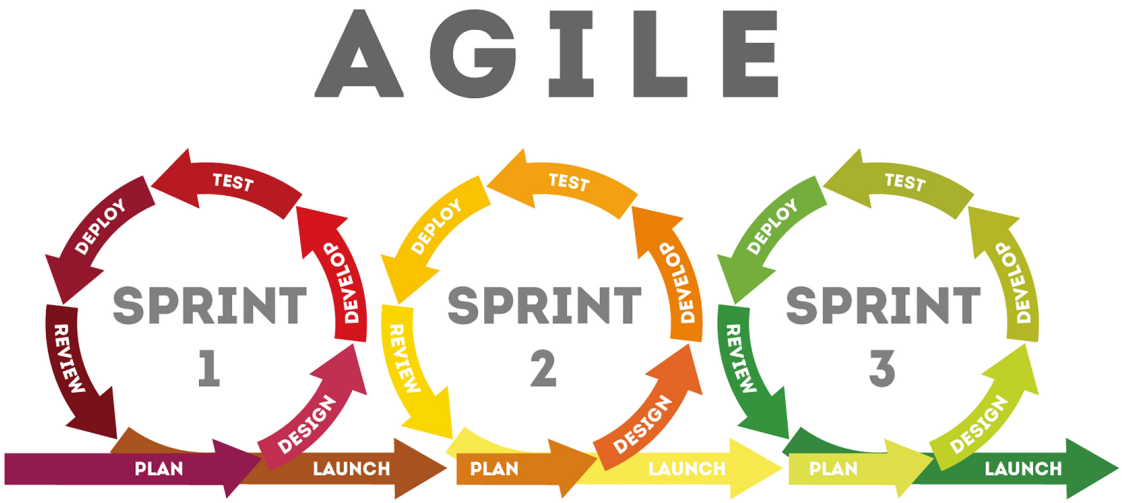


Figure 1.1: Agile Software Development Methodology

Github issues, board and milestones were used for project management, screenshots as follows.

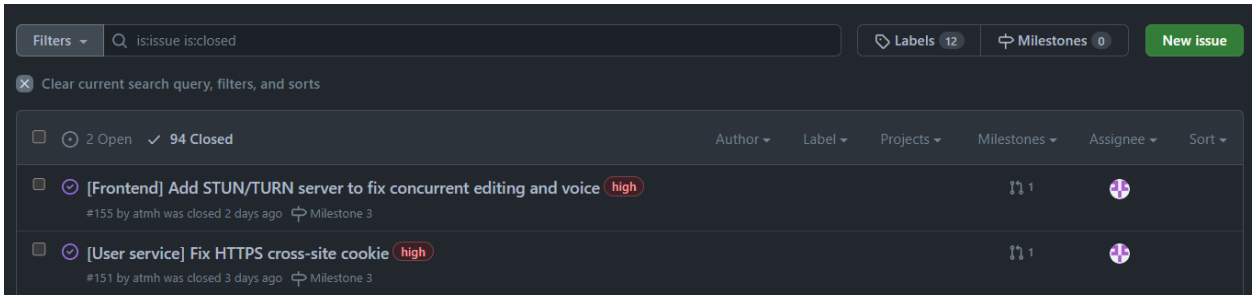


Figure 1.2: Github issues

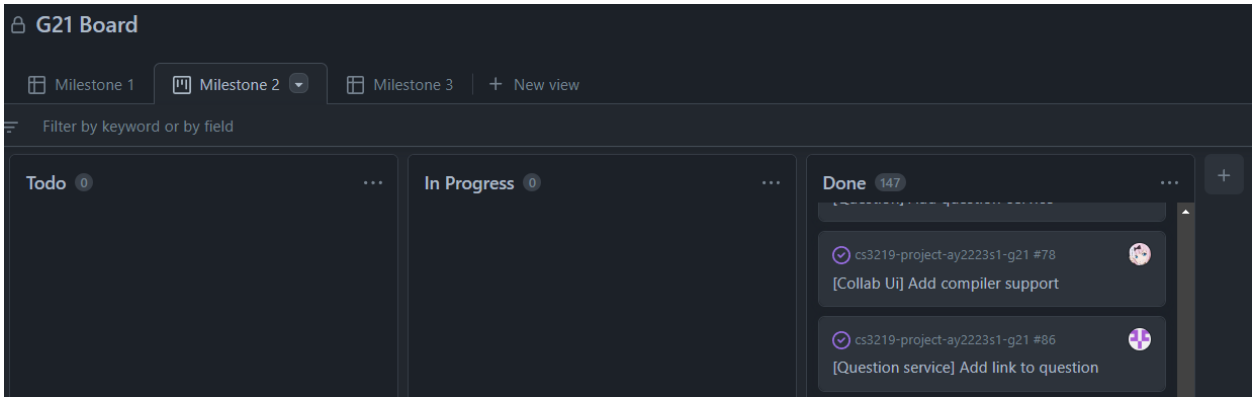


Figure 1.3: Github G21 Board

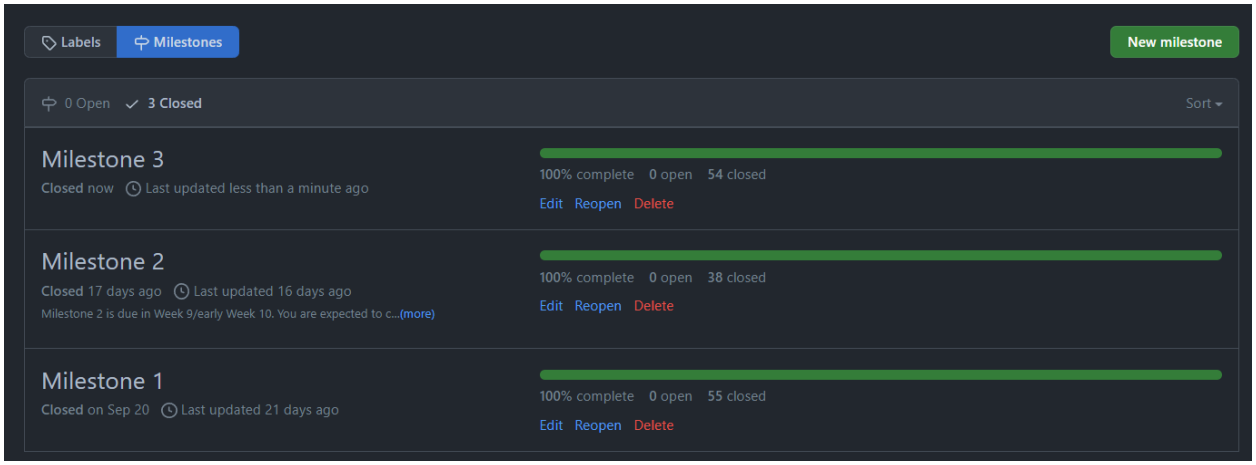


Figure 1.4: Github Milestones

1.4.3. Project Scope

Due to time constraints and the technical capabilities of the team, only a limited number of deliverables will be met for the project. This section aims to set the boundaries of the project and define the goals, deadlines and project deliverables the team will be working towards.

Objective

- Allow users to register for an account and login with PeerPrep..
- Allow users to select the difficulty level of their live coding question.
- Create a peer matching system that allows users to practice live coding with a peer.
- Provide users with a coding environment to practice their live coding skills with another peer.

Tech Stack

Purpose	Tech
Frontend	React
Redux	State Management Tool
Backend	Node.js
Database (Dev)	MongoDB
Database (Prod)	MongoDB Atlas
Testing	Mocha and Chai
Continuous Integration	Github Actions
Continuous Deployment	GCP Cloud Run

Dependencies

The following are a brief summary of the dependencies on the external libraries used. Note that this list is not inclusive of all the libraries used, but only includes the major ones.

Purpose	Dependency
Frontend	react
Backend	express
STUN/TURN server	xirsys
Pub-sub messaging	socket.io-client
Web sockets	ws
Concurrent editing and text chat	peerjs
Voice call	y-webrtc
Hash and salt	bcrypt
SHA-512 Hash	crypto
Access mongodb models	mongoose
Unit and integration tests	mocha
Unit and integration tests	chai
Access environment variables	dotenv

Roadmap

- **Week 3 - 4:** Read up on project requirements and learning of new technical tools.
- **Week 5:** Set up core features of user service & PeerPrep frontend.
- **Week 6:** Integration of user service & frontend web application.
- **Week 7:** Set up matching service & testing of PeerPrep application.
- **Week 8:** Integration of matching service to PeerPrep frontend.
- **Week 9**: Set up & integration of collaboration service to PeerPrep frontend.
- **Week 10:** Set up & integration of question service to PeerPrep frontend.
- **Week 11**: Set up & integration of compile service to PeerPrep frontend.
- **Week 12**: Deployment of application & documentation.

Out of scope

- Setting up video conferencing in the live coding environment.
- Handling of race conditions during concurrent code edits.
- Solving the firewall blockage of media streams in a public network (Section 6.2.4)
- Rejoining the live coding room after participants disconnect.
- Real-time collaboration for code compilation (i.e Compilation of code triggered by 1 peer can be seen by another peer.).

1.5. System Requirements

For PeerPrep to function well, your computer should meet the minimum system requirements specified. It is possible that Peerprep might work on other platforms or versions not listed below. However, if there is an issue due to not meeting the minimum requirement specified, the developers will not be able to provide any relevant support.

Windows

- Windows 10 (8u51 and above)
- Windows 8.x (Desktop)
- Windows 7 SP1
- Windows Vista SP2
- RAM: 128 MB
- An Intel Pentium 4 processor or later that's SSE3 capable

Mac OS X

- Intel-based Mac running Mac OS X 10.8.3+, 10.9+

Modern Browser

- Chrome version 107.0.5304.88 or later.
- Firefox version 106.0.5 or later.

Note that for voice communication to work, Firefox has a better chance of establishing a connection to send media streams than Chrome.

2. Requirements

Below are the functional requirements and non-functional requirements for the project. For requirements that are not achieved in this project, they will be highlighted in red.

2.1. Functional Requirements

2.1.1. User Service

S/N	Functional Requirement	Priority
FR1.1	The system should allow users to create an account with username and password.	High
FR1.2	The system should ensure that every account created has a unique username.	High
FR1.3	The system should allow users to log into their accounts by entering their username and password.	High
FR1.4	The system should allow users to log out of their account.	High
FR1.5	The system should allow users to delete their account.	High
FR1.6	The system should allow users to change their password.	Medium
FR1.7	The system should allow the user to reset their password.	High
FR1.8	The system should allow users to get their history.	High
FR1.9	The system should track the history of questions that a user did.	High
FR1.10	The system should allow users to view past questions in detail.	High

2.1.2. Matching Service

S/N	Functional Requirement	Priority
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
FR2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High
FR2.3	If there is a valid match, the system should match the users within 30s.	High
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
FR2.5	The system should allow users to select topic of the question they wish to attempt	Medium
FR2.6	The System should be able to match two waiting users with the same topic and put them in the same room	Medium

2.1.3. Question Service

S/N	Functional Requirement	Priority
FR3.1	The system should allow users to retrieve a question of the respective difficulty level (indexed by Easy, Medium, Hard)	High
FR3.2	The system should show the same question to both users in the workspace	High
FR3.2	The system should allow the user to retrieve a random question	High
FR3.3	The system should allow users to retrieve a new question that is different from the previous questions.	High
FR3.4	The system should provide a skeleton code for users in multiple programming languages.	High

2.1.4. Collaboration Service

S/N	Functional Requirement	Priority
FR4.1	Allow participants to connect to the same room	High
FR4.2	Show identical question to users matched in the same room	High
FR4.3	Participants must be able to edit code concurrently on the shared editor	High
FR4.4	Participants must be able to leave the room	High
FR4.5	Participants should be informed when 1 participant has left the room	High
FR4.6	Participants should be able to undo and redo their code edits	Medium
FR4.7	Participants should be able to talk to one another	High
FR4.8	Participants should be able to engage in a video call with each other.	Medium
FR4.9	Participants should be able to rejoin the room when they are disconnected.	Medium

2.1.5. Compile Service

S/N	Functional Requirement	Priority
FR5.1	Users should be allowed to compile their code.	High
FR5.2	The service should be able to indicate errors in the submitted code.	High
FR5.3	The service should be able to support compilation of different programming languages.	Medium

2.2. Quality Attributes (NFRs)

2.2.1. Security

S/N	Non Functional Requirement	Priority
NFR1.1	Users' passwords should be hashed and salted before storing in the DB.	High
NFR1.2	Password must be between 6 to 20 characters with at least 1 number, one uppercase and one lowercase letter to minimize brute force attacks	High
NFR1.3	Expired JWT tokens should be refreshed via refresh token	High
NFR1.4	Sensitive access tokens like JWT should be stored in a non-persistent storage like redux	High
NFR1.5	Refresh token should be stored in the client and sent to the server to refresh the JWT token	High
NFR1.6	Refresh token should be hashed before storing in the DB.	High
NFR1.7	JWT token should be verified before allowing access to user specific APIs .	High
NFR1.8	Sensitive requests to the services need to be authenticated by the JWT token.	High
NFR1.9	The system should automatically delete expired refreshToken in the database.	Medium

2.2.2. Performance

Currently, for **NFR2.3**, **NFR2.4** and **NFR2.5**, we are able to guarantee that the performance requirements are met. However, in the future as the application gains more users, we are not able to guarantee that the application will be able to meet those performance requirements.

This is because:

- **NFR2.3**: Relies on the network latency of the application.
- **NFR2.4**: Relies on the user's browser capabilities.
- **NFR2.5**: Relies on the size of the database and the type of query that is being processed.

NFR2.1	The time delay of the code edit should not exceed 50ms	Medium
NFR2.2	The time delay of the question appearing should not exceed 300ms	Low
NFR2.3	The time delay of page loading should not exceed 10 seconds.	High
NFR2.4	The time delay of page renders should not exceed 2 seconds.	High
NFR2.5	The time delay of fetching results from the database should not exceed 1 second.	High

2.2.3. Usability

NFR3.1	Users should be able to know how to interact with the application without reading any documentation.	High
NFR3.2	Users should be able to access the application as long as there is an internet connection.	Medium
NFR3.3	There should be a smooth transition between different aspects of the application.	Medium
NFR3.4	Users should be able to determine the functionality/feature of the web component without any additional help.	Medium

2.2.4. Reliability & Availability

NFR4.1	If system failure occurs, there should be no loss of application data in the database.	High
NFR4.2	Under normal operational conditions, the application should not fail more than 2 times a day.	Medium
NFR4.3	The system should be able to restore code edits in case the user has been disconnected from the internet.	Low

3. System Architecture

As mentioned in 1.4.1. Development Approach, the main architectural pattern used by PeerPrep is the microservice architectural pattern. Below is an overview of the architectural diagram for the PeerPrep web application.

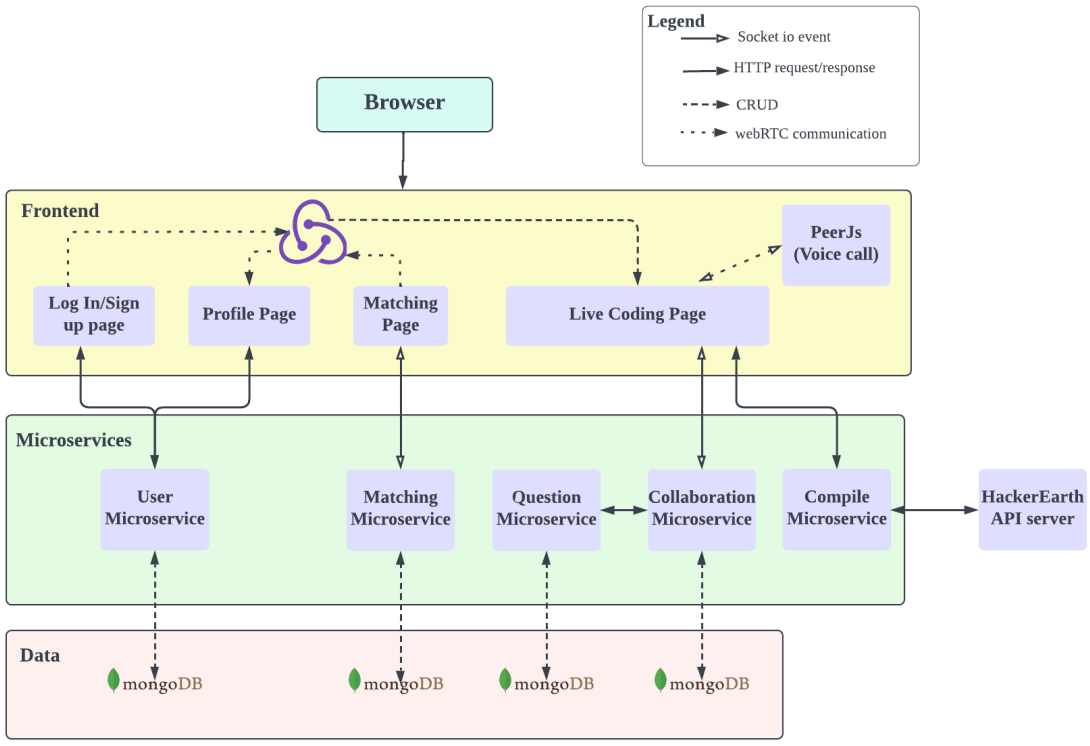


Figure 3.1: Overview of PeerPrep System Architecture

Even though the main architectural pattern adopted in this project is the microservice architecture pattern, traces of other architectural patterns can be observed in the diagram. By incorporating different architectural patterns into the project, we can reduce/eliminate anti-patterns in the development process and ensure all components/workflows are well organized.

In this section, we will identify and break down the different architectural patterns used in the project and discuss how the patterns are able to help us in achieving PeerPrep's software requirements.

3.1. Microservice Architectural Pattern

In the microservice architectural pattern, the entire application is structured as a set of loosely coupled, collaborating services where each service is able to function independently without any additional dependencies. This implies that development of such services can be conducted in parallel, speeding up the development process and offering better agility in terms of deploying updates as and when required. Also, in an AGILE development process, incremental code changes are favored more than large changes to the code base and in a microservice application, the loosely coupled nature of the services allow developers to add new features/upgrades without having to change a large segment of the code base.

During the development of PeerPrep, there was a need for regression testing to ensure that no defects/bugs are introduced into the system when new code is introduced into the codebase. The loosely coupled nature of PeerPrep made testing easy to accomplish as different services can be tested independently from one another and even when there are inter-dependencies between services, it is easy to replace the dependencies with stub classes to produce a more consistent testing result.

Below are some other benefits of the microservice pattern is also experienced during the development of PeerPrep but they will not be the focus of discussion for this section:

- Better code maintainability.
- Easier deployment of services.
- Improved fault isolation.

However, one trade-off experienced by the team when developing PeerPrep using the microservice architecture is the additional latency experienced when deploying PeerPrep to a public network. The primary communication protocol used by PeerPrep during inter-service communication is the HyperText Transfer Protocol (**HTTP**) and as HTTP relies on a network to transmit data between services, increased level of network traffic can result in slower response time and as a result, slow down the interactive features of the application. This issue is magnified even further when the application is deployed onto the internet, where there is an increasingly high amount of network traffic.

3.2. Layered Architectural Pattern

In the layered architectural pattern, the application can be segregated into one or more layers with each layer having a distinct and specific set of responsibilities. Looking back at the overall architectural diagram of PeerPrep, it is obvious that the application can be segregated into 3 layers. Namely, the Frontend layer, the Microservice layer, and the Data layer. The logical structuring of the components into different layers ensures that there is a distinct separation of concern where each logical layer is assigned a set of responsibilities that are different from the other layers. This method of segregation provides better changeability to the codebase, where any layer can be replaced by any other logical layer as long as they both implement the same interface for inter-layer communication.

This implies that whenever there is an improvement or release of new tools that are able to allow developers to create a better implementation of the logical layers in PeerPrep, these tools can be easily adopted and integrated with the current system architecture, making PeerPrep an application that is easily adaptable to future technological advancements.

4. Design Patterns

4.1. Model-View-Controller (MVC) Pattern in Frontend

A typical web development pattern seen in frontend applications is the Model-View-Controller (**MVC**) pattern. This design pattern emphasizes a division between the business logic and the appearance of the software and this “separation of concerns” provides a better division of computational labor and also, is widely used to improve the extensibility of the codebase.

The three main components in the MVC pattern are as follows:

- **Model:** Holds the business logic and data of the application.
- **View:** Controls and shows the presentation and layout of the application to the user.
- **Controller:** Distributes and updates instructions to the view and model elements.

Below is a diagram showing the interaction between PeerPrep’s frontend and one of PeerPrep’s backend servers.

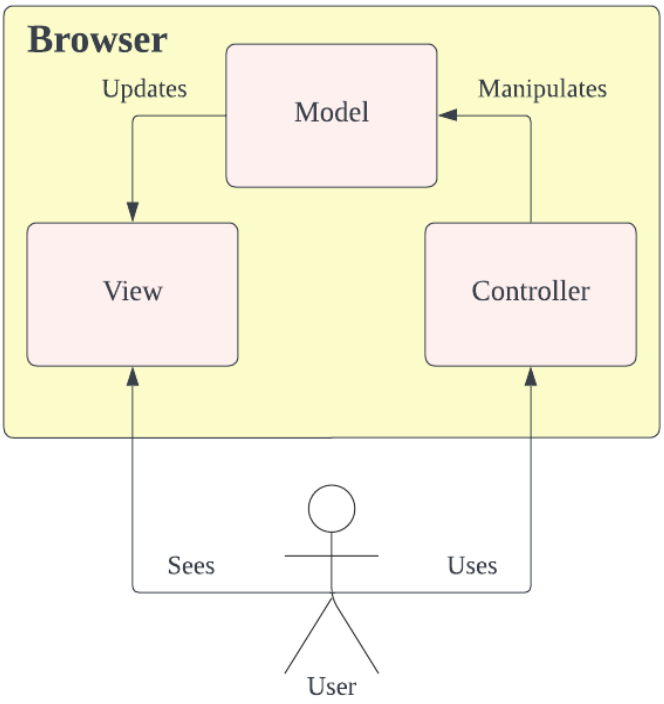


Figure 3.2: MVC Design Pattern in Frontend

React was used for the development of PeerPrep's frontend application and the MVC pattern was used during the development process. In PeerPrep's frontend application, the view is being represented by the Javascript Expression (**JSX**) components that are rendered onto the Document Object Model (**DOM**) during runtime and the model is being represented by redux, a state management tool for React.

The controllers in the PeerPrep's frontend application are callback functions that are invoked whenever the user interacts with the JSX components on the DOM. These callback functions then trigger a series of actions that will ultimately update the application state in redux and whenever there is a change in the application state, the view automatically updates based on event listeners implemented within the React library.

By separating the different aspects of the frontend code based on their responsibilities specified in the MVC pattern, we are able to create modules with high cohesion and this concept of modularity is essential in creating reusable modules that can reduce the develop time by increasing the efficiency in the development process through code reuse.

4.2. Web Model-View-Controller (MVC) Pattern for PeerPrep

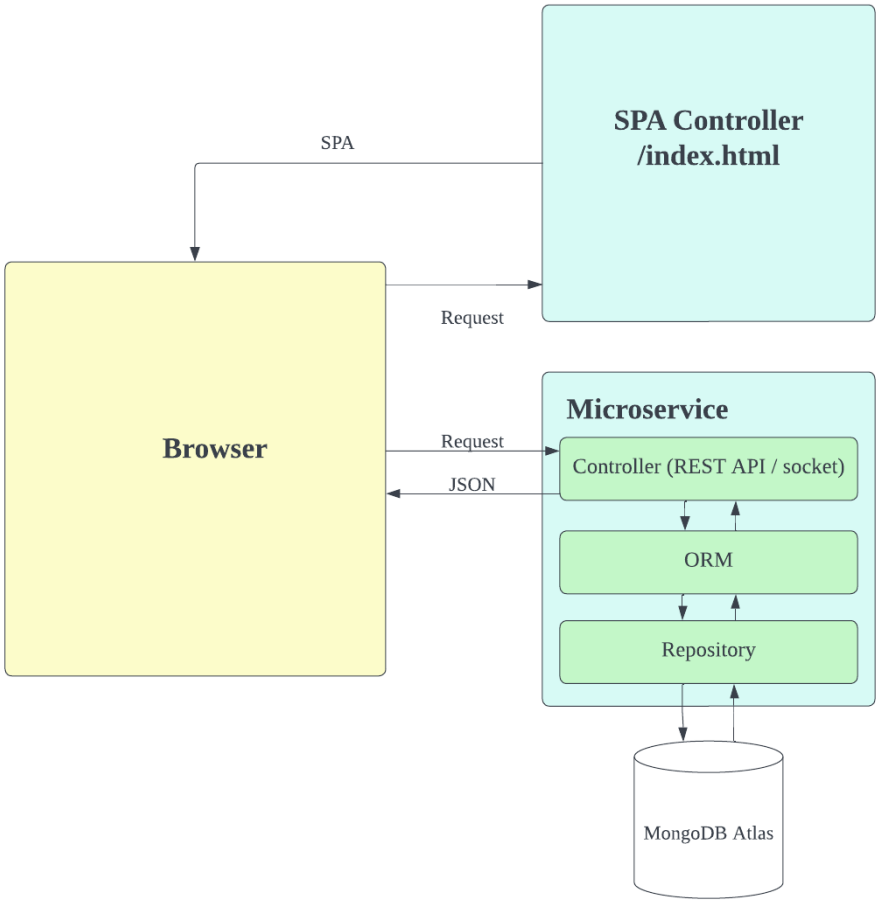


Figure 3.3: Web MVC Design Pattern in PeerPrep

At the application level, traces of the MVC pattern can be observed as well and this can be illustrated in figure 3.3. In PeerPrep, whenever a user navigates to the frontend url from the browser, a single html file is transmitted as a response and the browser would render the html file as a single page application. When users interact with this application, javascript written in the application would communicate with the various microservices in PeerPrep and this in turn is translated to CRUD operations that operate on the records in MongoDB Atlas. The benefits of employing this design pattern into the design of the application is similar to the benefits mentioned in 4.1. Model-View-Controller (MVC) Pattern in Frontend.

.

4.3. Mediator Pattern

The Mediator Pattern is employed in the Frontend microservice to provide a common point of communication. This restricts explicit reference between the other microservices (except between question and collaboration microservice), and forces them to collaborate only via the Frontend microservice, thus promoting loose coupling.

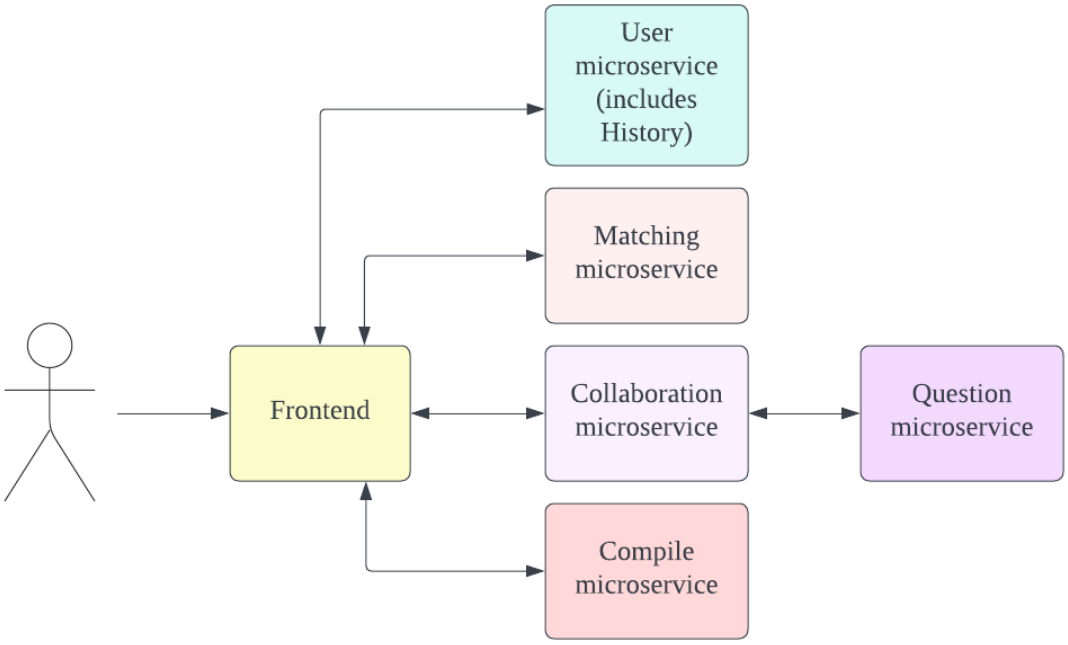


Figure 3.4: Mediator Design Pattern in PeerPrep

Also, Mediator simplifies object protocols. It replaces many-to-many interactions with one-to-many interactions between the microservices, making the code base easier to understand, manage, and extend. More specifically, if an object is updated with new interaction rules or a new object is added, only the Mediator needs to be updated.

However, it needs to be acknowledged that the Mediator object can become complex if the number of microservices in PeerPrep becomes large. Since PeerPrep is likely to remain small in time to come, we have decided that trading complexity in the mediator for less complexity in the interactions between microservices is worthwhile.

4.4. Pub-Sub pattern

Websockets are used in the Matching and Collaboration microservices. For example, webSockets are used to inform the frontend in real time that a match has been found, or the user has joined a room successfully.

This is an implementation of Pub-Sub pattern whereby the frontend will listen to topics (i.e. events in Socket IO) and the microservices will publish changes to the observers (frontend) by making use of the emit feature of Socket IO.

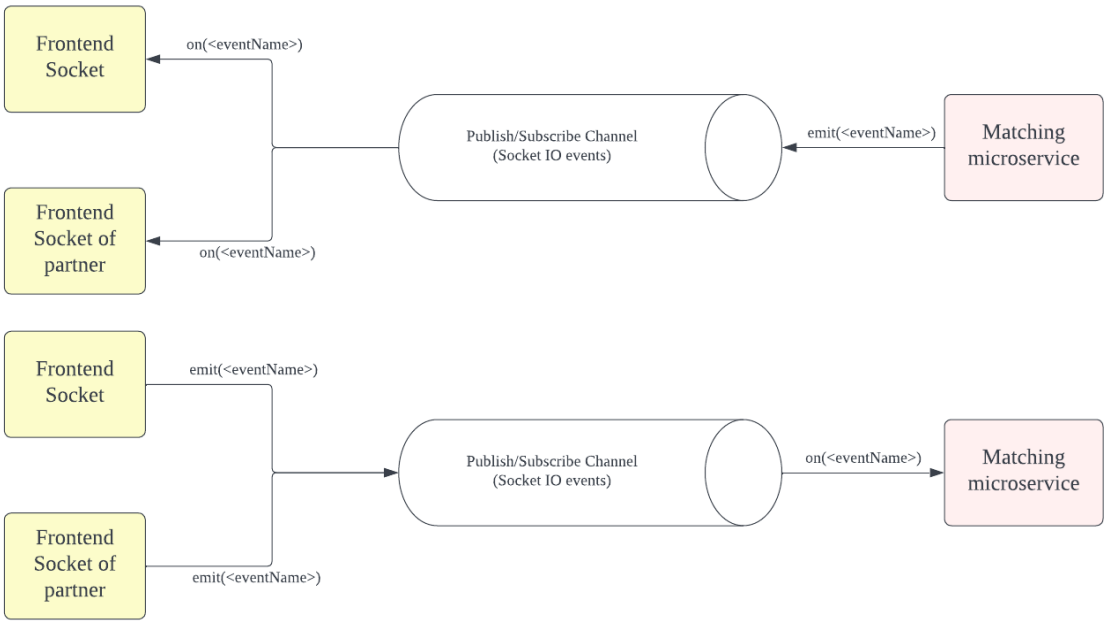


Figure 3.5: Pub-Sub Design Pattern in PeerPrep

This approach makes it possible to create event-driven services (matching microservice and collaboration microservice) without constantly querying the message queue for messages. Furthermore, the Pub-Sub pattern isolates publishers from subscribers. Subscribers do not need to know where the messages come from, and publishers do not need to know who consumes the messages, thus enforcing **Information Hiding** in our system.

5. Design Considerations

5.1. Security

5.1.1. Storage of JWT token

In PeerPrep, we primarily use JWT tokens as the means to authenticate the user and to allow the user access to the various services in the application. These JWT tokens are sensitive access tokens that have to be securely stored in the client and in this section, we will discuss the design considerations made in coming up with a secure way to store the JWT token in the frontend application.

There are several ways to store JWT token in a frontend application and they are namely:

1. Local Storage.
2. Session Storage.
3. Cookies.
4. In-memory storage.

Local storage, session storage and cookies are storages that remain persistent across page refresh and browser tabs/windows. By storing the tokens in these storages, the token will be retained in the browser even after the application is closed, resulting in a higher exposure to attacks made by malicious parties. If an attacker can inject malicious Javascript into the frontend application using a cross-site scripting (XSS) attack, they can potentially retrieve the access token stored in these persistent storages, thereby compromising the user's PeerPrep account.

In-memory storage is a type of non-persistent storage where the data would be removed after the frontend application is closed or reloaded. This means that after the PeerPrep page is closed, the browser will not retain any information regarding the JWT token and in general, it would present less of a security risk as compared to persistent storage options. However, because the JWT token is lost every time the user refreshes/closes the browser, the user would have to login on every tab/window and everytime a browser is reopened. This can hamper the user's experience of the application. Hence, there is a need to implement a "silent login" feature to ensure that the user remains logged into their account without compromising the security of the application.

To implement the silent login feature, we can use the refresh token authentication pattern for the authentication of the user. This pattern advocates the use of short-lived access tokens that can only be refreshed through the use of a refresh token. Specifically, the authentication server can send a refresh token to the browser as a http-only cookie every time the user manually login into their account. Since the refresh token is opaque (meaning it does not give away any sensitive information to an attacker), it is safe to store the refresh token in a persistent storage like a cookie and with the added benefit of making the cookie http-only, it would be difficult for an attacker to retrieve the refresh token via a XSS attack on the browser. As it is only possible for the browser to have the refresh token only after the first successful login, the refresh token can act as a persistent state that describes the login status of the user. As a result, just by checking the existence of the refresh token in the browser, it is possible to automatically authenticate the user without having them to manually login into their account.

Note: Since the refresh token is stored as a http-only cookie, it is not possible to query for the existence of the token in the frontend application. The cookie is automatically attached to every request sent to the authentication server by the browser. Hence, the server can take up the role of checking for the existence of the cookie and respond with an error message if the cookie does not exist in the header of the request.

Below is an activity diagram depicting the control flow of the login feature for PeerPrep.

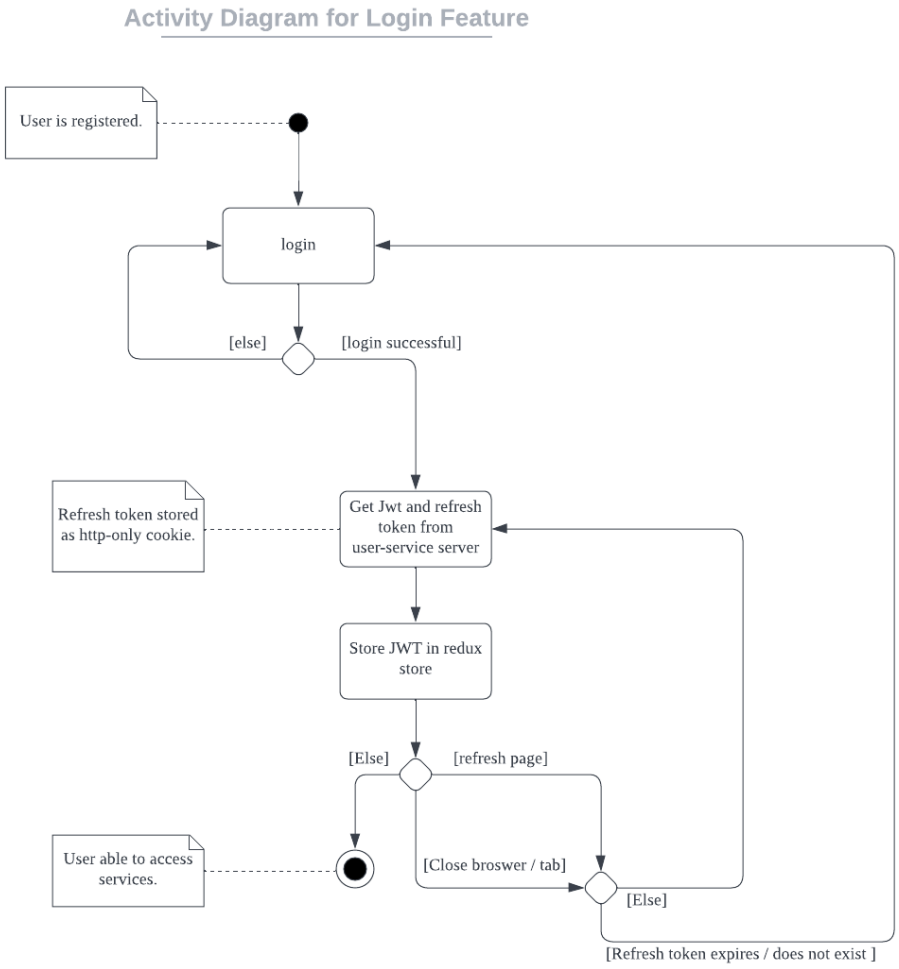


Figure 5.1: Activity Diagram for login feature

With the control flow shown in the activity diagram, the PeerPrep application is still vulnerable to CSRF attacks. As long as the attacker possesses the refresh token, they are able to request for a new JWT token that can allow them access to the services in the PeerPrep application. To circumvent the vulnerability, cross-origin-resource-sharing (cors) is set to only allow requests coming from the PeerPrep client domain. This effectively forces the authentication server to accept requests that come only from the PeerPrep client and as such, prevents attackers from requesting a JWT token from a different domain.

Lastly, to ensure that the storage of refresh tokens are securely stored in the authentication server's database, every refresh token generated by the authentication server is hashed before it is sent to the database for storage.

5.1.2. Password strength

Password strength validation is performed to ensure that every password conforms to the criteria where the password must be between 6 to 20 characters with at least 1 number, 1 uppercase and 1 lowercase letter.

Enforcing this criteria ensures a decently complex password to protect from brute force attacks.

5.1.3. Hashed and salted password

Users' passwords are hashed and salted prior to insertion into the database. Password hashing and salting makes storage and management more secure. Hashed and salted passwords make it harder for bad actors to crack passwords at scale. The unique hash created with salted passwords defends against attack vectors, including dictionary, brute force and hashtable attacks. Hashing with salting is non-deterministic, making the password even harder to crack. Bcrypt is used for hashing and salting.

5.1.4. JWT and Refresh Token expiry

Every JWT token expires in 1 hour which can be refreshed by the refresh token. The refresh token expires in 1 day and has to be re-generated by login in again. This is to provide an extra layer of security in the case where either is compromised.

To prevent unnecessary accumulation of expired refresh tokens, the database has been configured to automatically delete the expired tokens.

5.1.5. Protected APIs

To protect the user sensitive APIs and APIs which require the user to be logged in, JWT token authentication is performed for these APIs. For these APIs, a middleware is implemented to verify the JWT token is valid, belongs to the user and has not expired. If it passes the checks, then the user can access the APIs. Otherwise, if the JWT token is invalid, does not belong to the user, or has expired, then a 401 Unauthorized will be returned.

5.2. Efficiency

5.2.1. Matching Process (Efficiency)

Instead of creating an REST API server to handle the matching between users, we have decided to use Socket IO as part of the tech stack. Compared to traditional REST API , Socket IO server allows the user and server to act as subscriber and publisher respectively to achieve a non-blocking matching experience for the user.

In the first version of the matching service , when user **A** sends a matching request to the matching service using Socket IO. The server will attempt to find a match for the user in the database every 5 seconds interval. Shortly after, user **B** comes by and sends a matching request to the matching service. Since user **A** comes first, the server will match user **A** with **B** and an interview model is created and saved into the database.

The server will then emit an event to user **A** to notify the user about the match. User **B** will only be notified by the matching service when the interval of 5 seconds is up and the matching service triggers a lookup in the interview table for a match.

Below is an activity diagram depicting the matching process ,

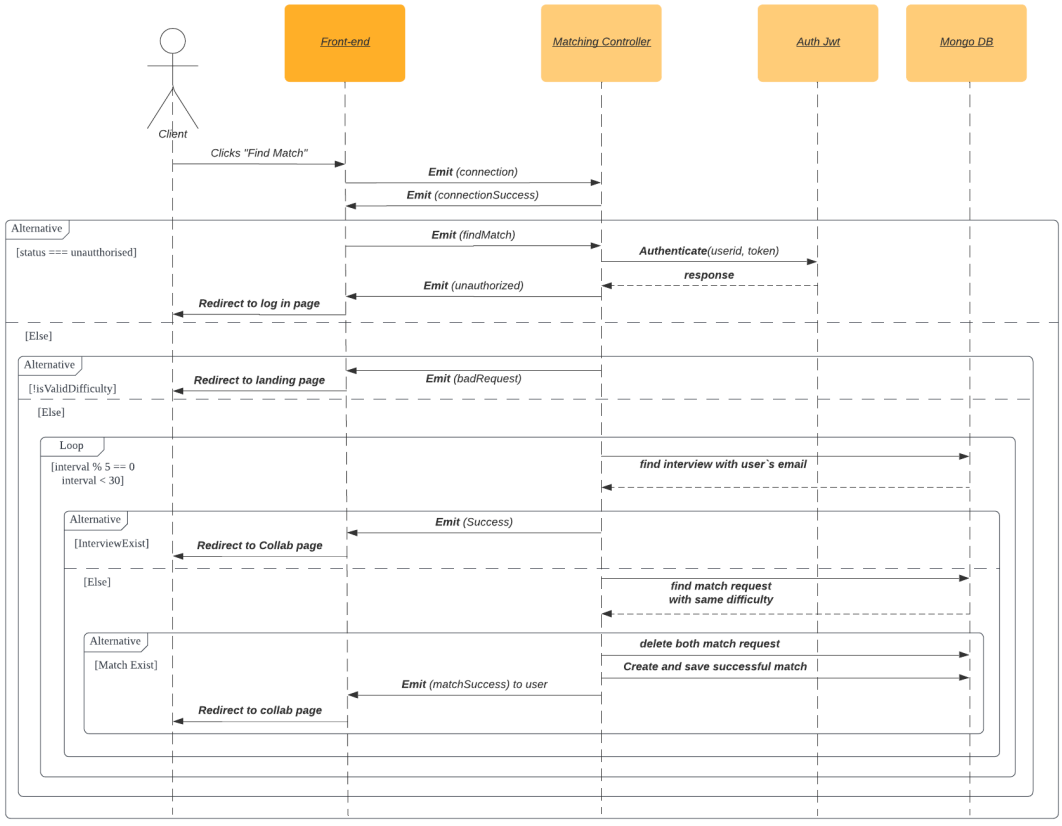


Figure 5.2: Sequence diagram for old matching service implementation

The initial version of the matching has a big flaw whereby the server has an interval created for every match request sent which is an inefficient use of resources. Furthermore it is a blocking process whereby the specific socket is blocked due to the interval.

To resolve the flaw, the matching service logic has been redesigned to be passive as opposed to aggressive matching in the first version. In the redesigned logic , instead of having an interval to find a match , the matching is now triggered by every match sent to the server. If a match cannot be found by the current user , the matching service will not attempt to match the user until a new user sends a match request to the server.

The diagram below depicts the redesigned matching process:

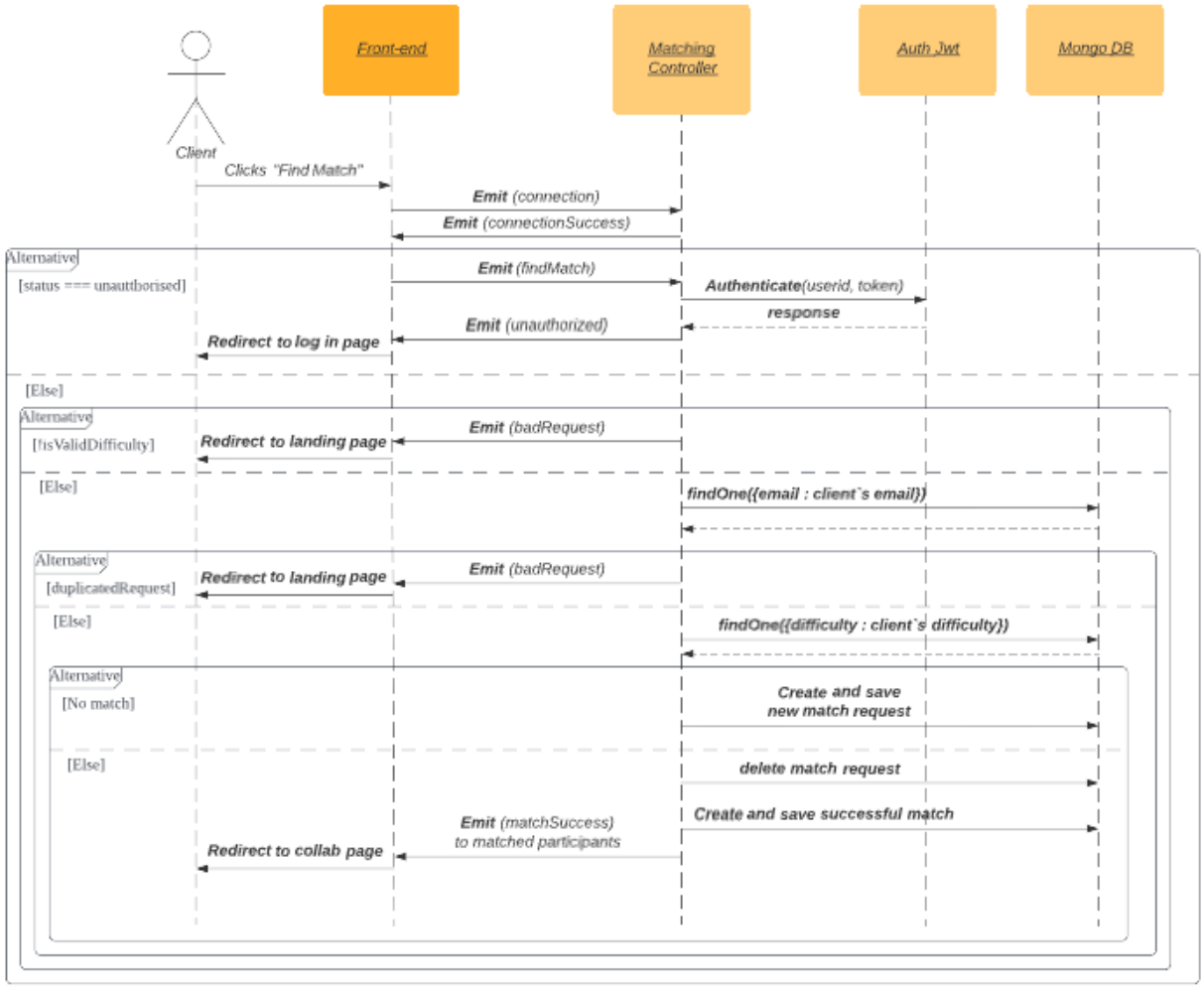


Figure 5.3: Sequence diagram for current matching service implementation

5.2.2. Hashed refresh token

Similarly, the refresh token is hashed for the same reason as above. However, it is not salted as the salting is non-deterministic. The refresh token is used as a key in the database, to access it quickly, only the hashed value of the token is used as hashing is deterministic.

This is a reasonable trade off, efficiency over security, as the plain refresh token is a UUID of length 36 generated by uuidv4, which naturally prevents against dictionary and brute force attacks. Moreover, SHA-512 is the hashing method used and there are no known attacks against SHA-512.

5.3. Integrity

5.3.1. Cancel Match Process

There are several ways that the cancel match feature can be implemented

1) A period of x seconds for cancellation before the request is send to the microservice
2) User can cancel match anytime during the 30 secs period

In the matching User interface , the team has decided to let the user have a 5 seconds period to cancel the matching request before emitting an event to the matching service to register the intent to match. This idea was inspired by the grab interface for booking a ride whereby the user is given some time to cancel the match before the ride request is sent to the server for matching.

The team has decided on this idea because it guarantees that the matching service will not match users that have canceled the match but the matching service has yet to receive the update to delete the match request in the DB.

5.3.2. Collaborative Editing (Usability over Integrity)

The initial idea for collaborative editing was to make use of socket IO to propagate the edits to the other participant in the room. Even though the idea is working , the team has encountered race conditions when the participants make simultaneous edits. For instance , the current text is ABC , User A removes A while User B removes B.

Assuming that the server receives User A edit of AB , the server will then propagate the text , AB to User B. Shortly after , the server receives User B`s edit of AC , the server will also propagate the text AC to user A.. Instead of the text A as A and B has been removed from the text ABC, the final text in the editor will be AC.

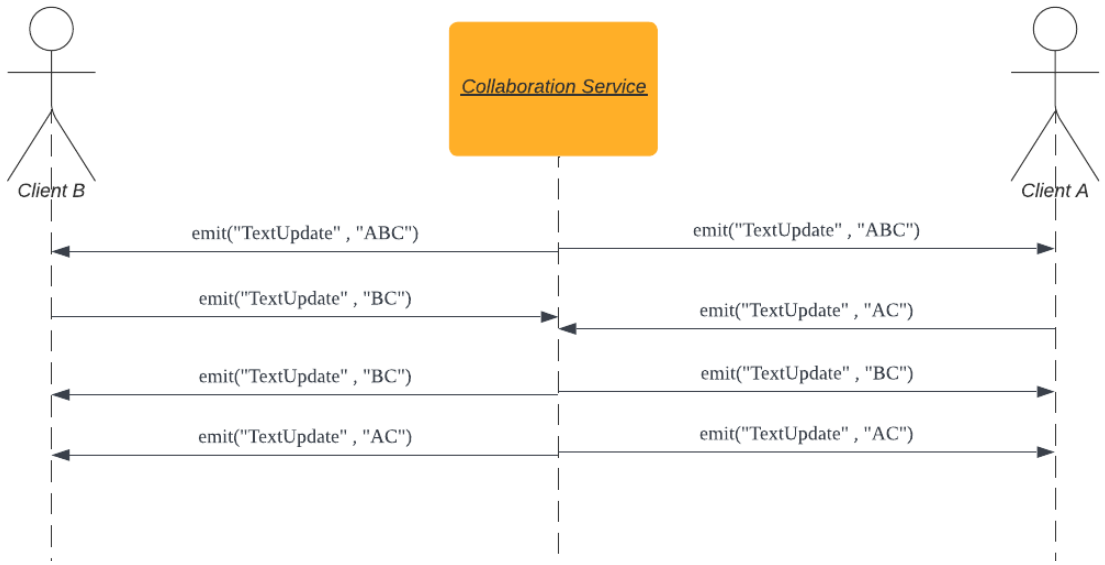


Figure 5.4: Race condition sequence diagram

There are 2 methods that can resolve the race condition shown above.

1) Operational Transformation
2) Conflict-free replicated data type

Operational Transformation (OT) is a technique / algorithm about the transformation of operations such that these operations can be applied to the documents that have different states, bringing the documents back to the same state. Every edit by the user will be represented as an (insert / delete) operation. These operations will then be sent to the server whereby the server will apply the operation to the document and broadcast the document to all users.

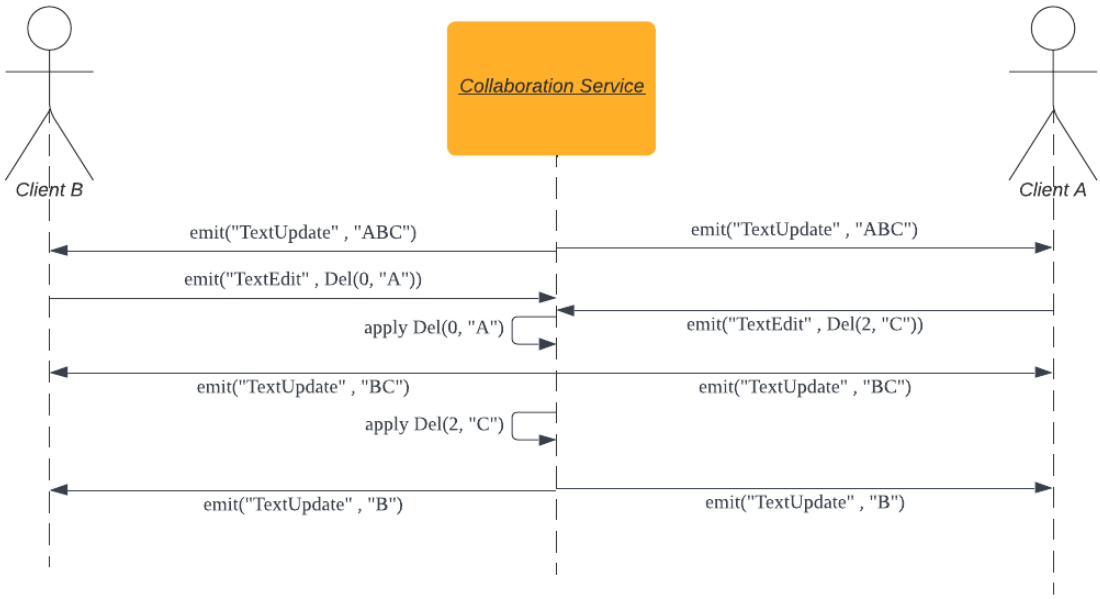


Figure 5.5: Operational Transformation conflict resolution sequence diagram

This technique is used in various products such as Google Docs , Google Slides , Wave and many more. The Javascript library that is based on this technique is Sharedb.

The 2nd method is Conflict-free (CRDT) replicated data type is a data structure that can be replicated across a network and the data is guaranteed to be consistent and correct eventually. The general technique is to broadcast all operations to all the clients and the conflict is resolved in a way whereby T(operation 1 , operation 2) == T(operation 2 , operation 1) where T is applying the operation on the text.

CRDT is used by Figma , Apple notes and many more. The javascript library that uses CRDT is **Yjs**.

The team has decided to use Yjs as sharedb requires the text edit to be the structure as shown below,

Figure 5.6: Shared operation structure diagram

Furthermore , the team could not find a javascript library that is recent and actively maintained that has the capabilities to transform the text edit to the required structure.

On top of selecting the javascript library to handle the conflicts , Yjs requires a provider to exchange the document updates with other peers. The team has chosen to use yjs-webrtc which uses simple-peer to connect participants in the room.

Note that for collaborative editing to work , the network used by the user has to allow UDP packets as the yjs-webrtc uses a ICE/STUN server to discover and connect the user. To increase the stability of the collaborative editing , the team has changed the default ICE/STUN server to using the twilio`s ICE/STUN server.

Despite that, there might be cases whereby the collaborative editing will not work due to the Internet service provider blocking connection to the twilio ICE/STUN server. In order to provide the user of PeerPrep a seamless user experience , the team has decided to fallback to the Socket IO method of collaborative chat.

6. Implementations

6.1. Features

The following table is a summary of the features in PeerPrep and the functional requirements (FR) that it satisfies:

Feature	FR
Authentication	1.1 to 1.7
Question history	1.8 to 1.10
Matching of two users	2.1 to 2.4
Random question based on difficulty	3.1 to 3.4
Concurrent code editing	4.1 to 4.6
Voice	4.7
Code compilation	5.1 to 5.3

6.2. Microservices

The Postman API documentation of for the project can be found here: https://documenter.getpostman.com/view/16419990/VVBTVT2J

6.2.1. User service (with history)

User microservice provides authentication and history functionality. Authentication is implemented using NodeJS and MongoDB with JWT Authentication.

Users' history is also stored in the same MongoDB as intuitively history is tightly coupled with the user. Storing history with the user information helps to facilitate clean-up of data. If the user were to delete his account, his history will be deleted as well. Alternatively, a separate microservice could be implemented for history. However, to prevent unnecessary accumulation of stale data, extra handling will be needed to delete the history if a user deletes his account. Hence, the history functionality is implemented with the user microservice.

There are nine routes: signup, login, delete, changePassword, refreshToken, requestPasswordReset, resetPassword, getHistory and updateHistory.

When a user creates an account, if the email is unique (case-insensitive) and the password passes the password validation, an account will be created and stored in MongoDB. Passwords will be hashed and salted using bcrypt prior to storage. Upon login, the credentials are checked against the data in MongoDB, and only if valid, then the user will be logged in. A JWT token will then be stored in the client's redux store and a refresh token will be stored in the browser cookie and in MongoDB (refer to Storage of JWT token).

If a user is logged in, the user can delete his account or change his password. Deletion of account will delete the respective user data in MongoDB and route the user back to the login page. For changing of password, if the user typed in the correct current password and a valid new password, his password will be changed in MongoDB. Similarity, the new password will be hashed and salted prior to storage.

The refreshToken route is used by the client to refresh the JWT token of the user. If the token does not exist, a 400 bad request will be sent. If the token exists but is either expired or invalid, an unauthorized 401 will be sent. Otherwise, if the refresh token is valid, a new JWT token will be generated, sent back to the client, and stored in the redux store.

If a user forgets his password, he can reset his password by entering his email. An email will be sent to his email containing a link to reset his password. This link contains a token that will expire within 30 minutes. If the user does not reset his password within 30 minutes, the link will no longer be valid. If a user passes the password validity check and successfully resets his password within 30 minutes, the token will be deleted and his password will be updated in MongoDB. Similarity, the new password is hashed and salted prior to storage.
The history of questions that a user attempts is also stored. Whenever a user matches and attempts a question, the question will be added to the history of questions that the user has attempted. On his profile page, he will be able to view his history, including the title, difficulty, time and a link to the question to view more information about the question.

To protect user-sensitive APIs and APIs that require the user to login, a middleware is implemented to verify the JWT token prior to access to these APIs, for example, delete account, change password, etc. Most of the APIs in peerprep are protected with this verification. There are three checks, firstly to check if the JWT token is expired. Secondly, it checks if it is a valid JWT token using the secret and lastly if it matches the user id. If all are valid, then these APIs can be accessed. Otherwise, a 401 unauthorized will be returned.

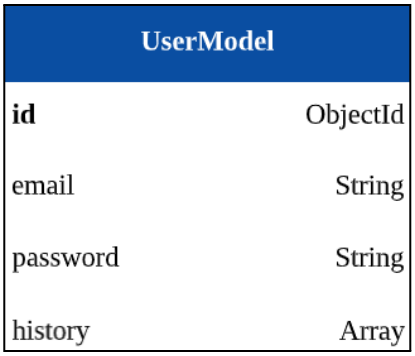


Figure 6.1: UserModel schema

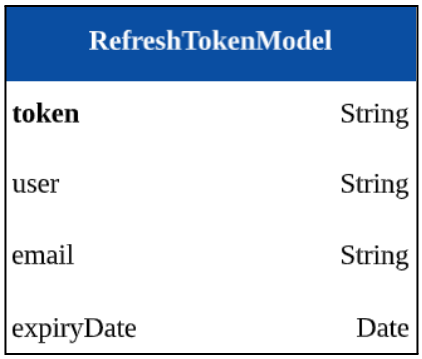


Figure 6.2: RefreshToken schema

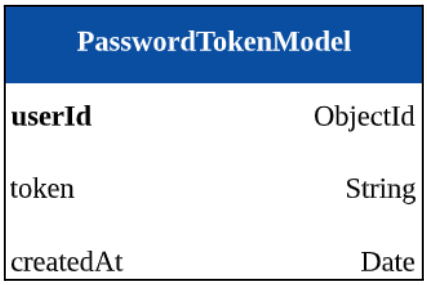


Figure 6.3: PasswordToken schema

6.2.2. Question service

Question microservice serves as the service to interact with the question database on MongoDB. There is only one route for this service, which is to retrieve a random question based on difficulty, with an optional exclusion list of question ids. There are 3 difficulties, Easy, Medium and Hard. The optional exclusion list is used for the new question functionality in the collaboration service, allowing it to retrieve new questions that are different from the previously retrieved questions in the same session.

A question consists of a title, difficulty, link to the question on LeetCode, instructions, examples, constraints and skeleton code for java, javascript and python.

QuestionModel	
id	ObjectId
title	String
difficulty	String
link	String
instruction	String
examples	Array
constraints	Array
java	String
javascript	String
python	String

Figure 6.4: QuestionModel schema

6.2.3. Matching service

Matching microservice serves as the service to match users with the same difficulty into a room. The service listens to 2 events upon a successful connection with the frontend`s socket which are findMatch and disconnect.

The findMatch event indicates that the user would like to submit a match request to the server and if there is a match request that has the corresponding difficulty to also inform the user. First and foremost ,the service will perform verification of the JWT token and userId included in the data in the socket event.

Next, the service will check if the supplied parameter , difficulty , is valid. If the supplied parameter, difficulty , is not valid ,a badRequest event is emitted by the server. After the difficulty check, the service will query into the MongoDB if the user has an existing valid matching request. If there is an existing valid matching request, a badRequest event is emitted by the server.

Events

connectionSuccess

Connected to backend successfully

unauthorized

Unauthorized request

badRequest

Request has invalid difficulty or Existing valid match request

createSuccess

Match Request created and saved in DB

findMatch

Find Match

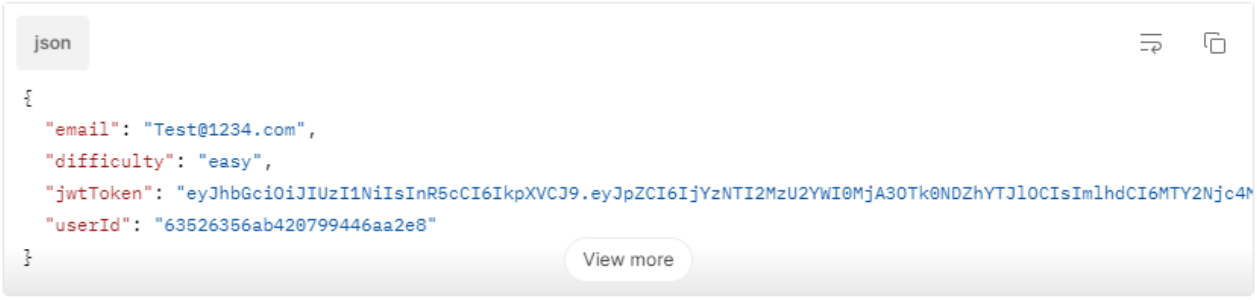


Figure 6.5:Matching Service Socket IO documentation

After all the verifications above , the service will perform a query into the MongoDB for a request with the same difficulty to match. If there is a match , the microservice will create an InterviewModel document and save into the mongoDB. This interviewModel document provides a unique interviewId that will be propagated back to the user that is making the match request currently as a roomId for the collaboration room.

InterviewModel	
id	ObjectId
interviewId	String
difficulty	String
email1	String
email2	String

Figure 6.6:InterviewModel schema

Other than the current matching user , the user that has been matched will be informed by the microservice as the microservice will emit the createSuccess event directly to the socket. Directly emitting to the other user that is involved in the match is possible as the MongoDB schema, match, includes the socketId.

MatchModel	
id	ObjectId
email	String
difficulty	String
socketId	String
timeCreated	Date

Figure 6.7: MatchModel schema

The 2nd event that the microservice listens to is disconnect , an event that will be emitted by every client socket when the socket is closed. When receiving data for this event , the server will delete the MatchModel document by socketId , allowing the user to submit another match request.

6.2.4. Collaboration service

The collaboration microservice is in charge of room creation , assigning users to rooms , question assignment to the room , voice call coordination. It also coordinates many interactions that the user can perform in the room such as requesting for the next question , changing coding language , sending a text message and exiting the room. The collaboration microservice makes use of Socket IO to achieve the above functionalities.

When the user is successfully matched and given a roomId from the matching service, as mentioned in Section 6.2.3, the user will attempt to connect to the collaboration service to be assigned to a room created by the microservice. Depending on the order of arrival, the user will receive data on different event listeners. For the first user to enter the room , the user will receive data from the joinSuccessFirst event listener while the subsequent users will receive data from the joinSuccess event listener.

For the joinSuccessFirst event listener , the client emits an event , TriggerFetchQn , to notify the microservice that it needs to fetch a question and save it in the roomModel for subsequent users. On the other hand, clients that receive data for joinSuccess will emit an event to signal to the microservice that the microservice needs to notify the rest of the participants to call this particular client. Below is the sequence diagram showing the interactions between the user and the server upon joining the room.

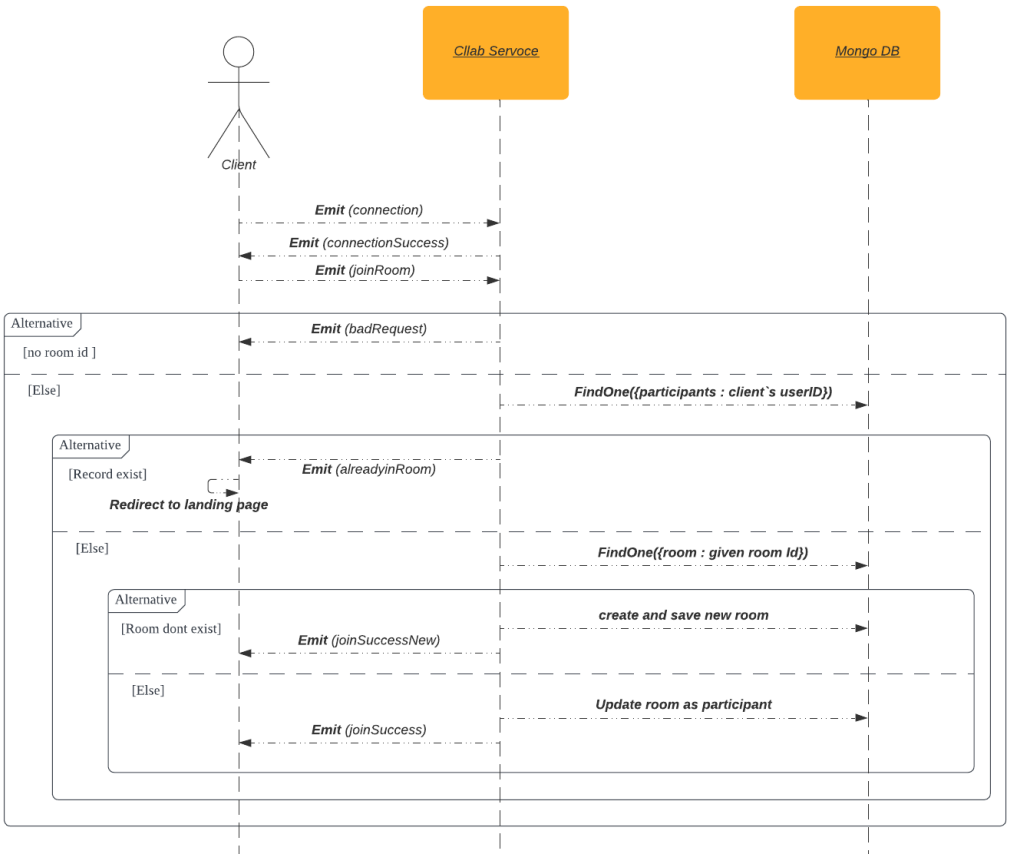


Figure 6.8:Sequence diagram for joinRoom feature

During the above sequence diagram , the microservice creates a roomModel document and saves it into the MongoDB. To enable a better user experience , the collaboration microservice has exposed a REST API to allow the frontend to check if a user is available for a match. This is possible as the MongoDB keeps track of the participants in the room , therefore enforcing the constraint that a user can only have an active room at any given moment.

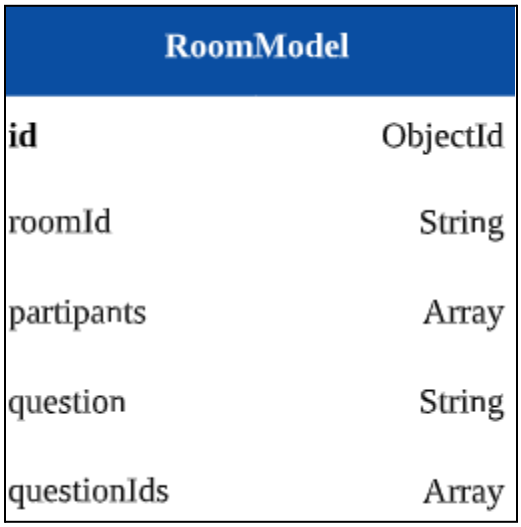


Figure 6.9:RoomModel schema

The collaboration service also provides voice call coordination between participants in the room. In particular , the microservice signals to the other participants that a new participant has joined and the participant should call him/her using the PeerJs Library.

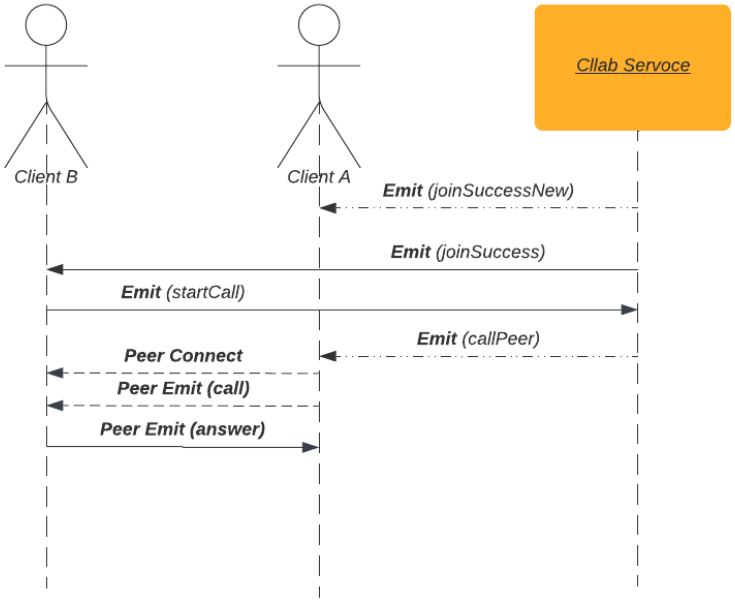


Figure 6.10:Sequence diagram for voice call feature

Do take note that there are cases whereby the voice chat feature will not work due connectivity issues with the signaling server that prevents the participants from discovering each other to establish the call. The connectivity issues may include network firewall , congestion in the signaling server and many other unforeseen circumstances.

In addition, for the voice chat to work , both participants are required to give mic and sound permissions. The team also encourages that both participants remove the permissions after leaving the call to allow the app to ask for the permissions again to ensure greater stability of the voice chat feature.

The figure below shows the events that the client sockets will listen to for updates from the microservice for the different possible interactions. Followed by the events that the client sockets will publish data to for the microserver to listen to.

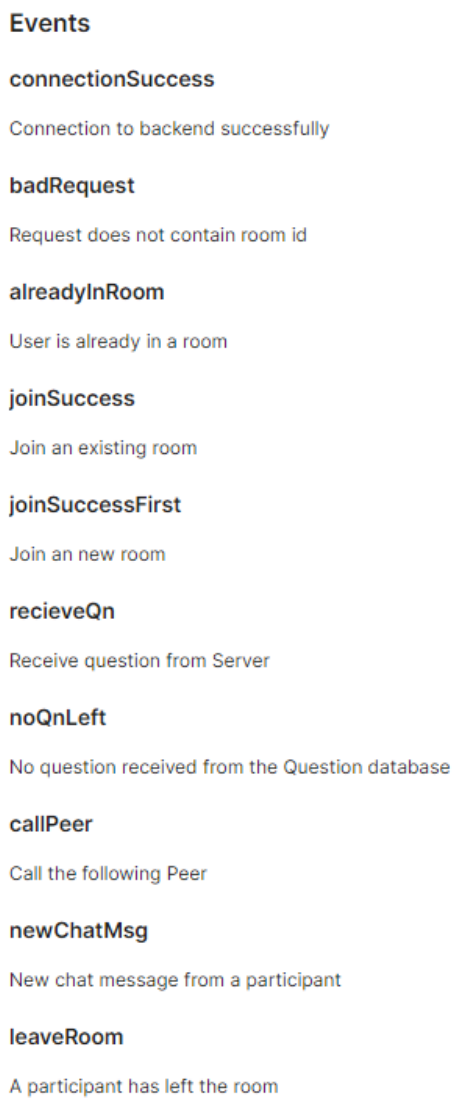


Figure 6.11: Collaboration Service Event List

joinRoom

Join Room

json


```
{
  "roomId": "63593254f117fedff5c763f",
  "jwtToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNTI2MzU2YWI0MjA3OTk0NDZhYTJlOCIsIml
  "userId": "63526356ab420799446aa2e8"
}
```


View more

TriggerFetchQn

Fetch Question

json


```
{
  "roomId": "63593254f117fedff5c763f",
  "difficulty": "Easy",
  "jwtToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNTI2MzU2YWI0MjA3OTk0NDZhYTJlOCIsIml
  "userId": "63526356ab420799446aa2e8"
}
```


View more

Figure 6.11.1: Event list emitted by collaboration page

startCall

Start Call

json


```
{
  "peerId": "636a6abf112c5fa3628a6497-63663b5489923587568c6bc6",
  "roomId": "63593254f117fedff5c763f",
  "jwtToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNTI2MzU2YWI0MjA3OTk0NDZhYTJlOCIsIml
  "userId": "63526356ab420799446aa2e8"
}
```


View more

sendChatMsg

Send Chat Message

json


```
{
  "newMessage": "Hello",
  "roomId": "63593254f117fedff5c763f",
  "jwtToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNTI2MzU2YWI0MjA3OTk0NDZhYTJlOCIsIml
  "userId": "63526356ab420799446aa2e8"
}
```


View more

exitRoom

Exit Room

json


```
{
  "roomId": "63593254f117fedff5c763f",
  "jwtToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNTI2MzU2YWI0MjA3OTk0NDZhYTJlOCIsIml
  "userId": "63526356ab420799446aa2e8"
}
```


View more

Figure 6.11.2: Event list emitted by collaboration page

6.2.5. Compile service

To implement the code compilation feature that allows users to compile and execute their code written in the collaboration page, the team decided to use the HackerEarth API, an API server that supports asynchronous code compilation and execution.

In the first attempt to implement the feature, API requests are called directly from the frontend and this leads to CORS errors as the API server rejects requests from unauthorized domains. Since CORS is only enforced when the request is made from a browser, the team decided to proxy the request through a NodeJS server such that it is able to make valid requests to the HackerEarth API server. From there, to integrate the code compilation feature into the frontend application, the frontend would have to request for code compilation through the newly created NodeJs server (aptly named Compile service).

6.3. Tests

The services were tested using the Mocha and Chai library. For example, for the user service, integration tests were implemented with an in-memory MongoDB (mongodb-memory-server). There are 32 test cases to cover most citations and edge cases. These test cases are added as part of the Continuous Integration to serve as regression tests.

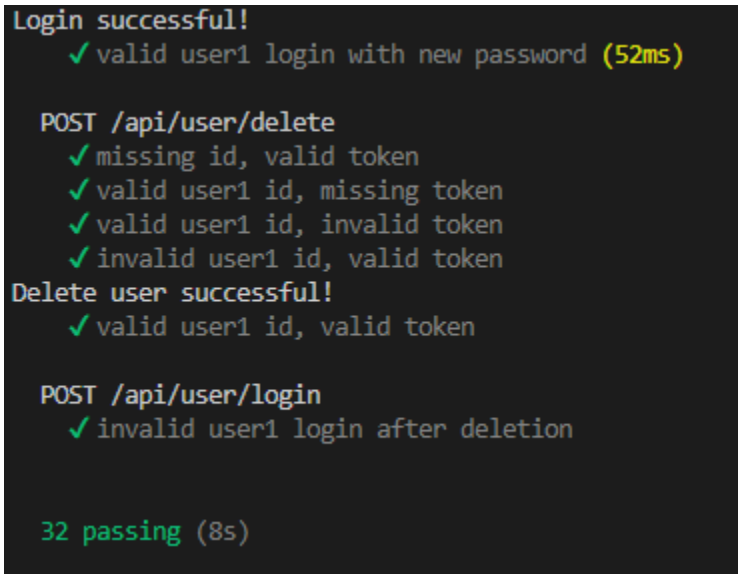


Figure 6.12: Test result screenshot

7. CI/CD

7.1. Cloud Database (Scalability)

For data resilience and future scaling, we deployed our databases as a cluster on MongoDB Atlas. Under greater load, our database cluster can automatically scale to accommodate greater CPU/memory utilization.

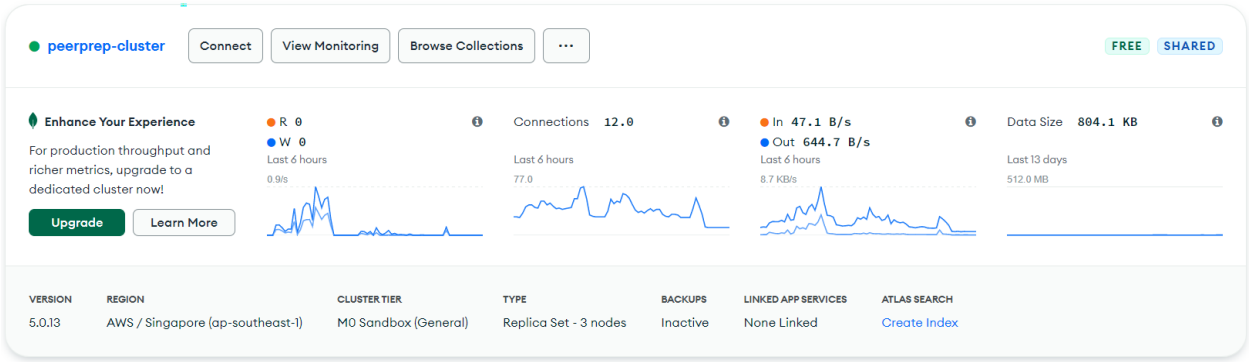


Figure 7.1: PeerPrep MongoDB Atlas

7.2. Continuous Integration (CI)

All pushes and pull requests to our github repository are run on a CI pipeline.

A linter is implemented into our CI pipeline in order to ensure consistent coding style and a smoother development experience. The SuperLinter is used as it is configurable to lint multiple file types, including Javascript, HTML, JSON, YAML, etc.

Figure 7.2: Github CI pipeline super-lint

Aside from the linter, the CI pipeline also verifies that our project builds successfully on Node 16 and runs the unit tests.

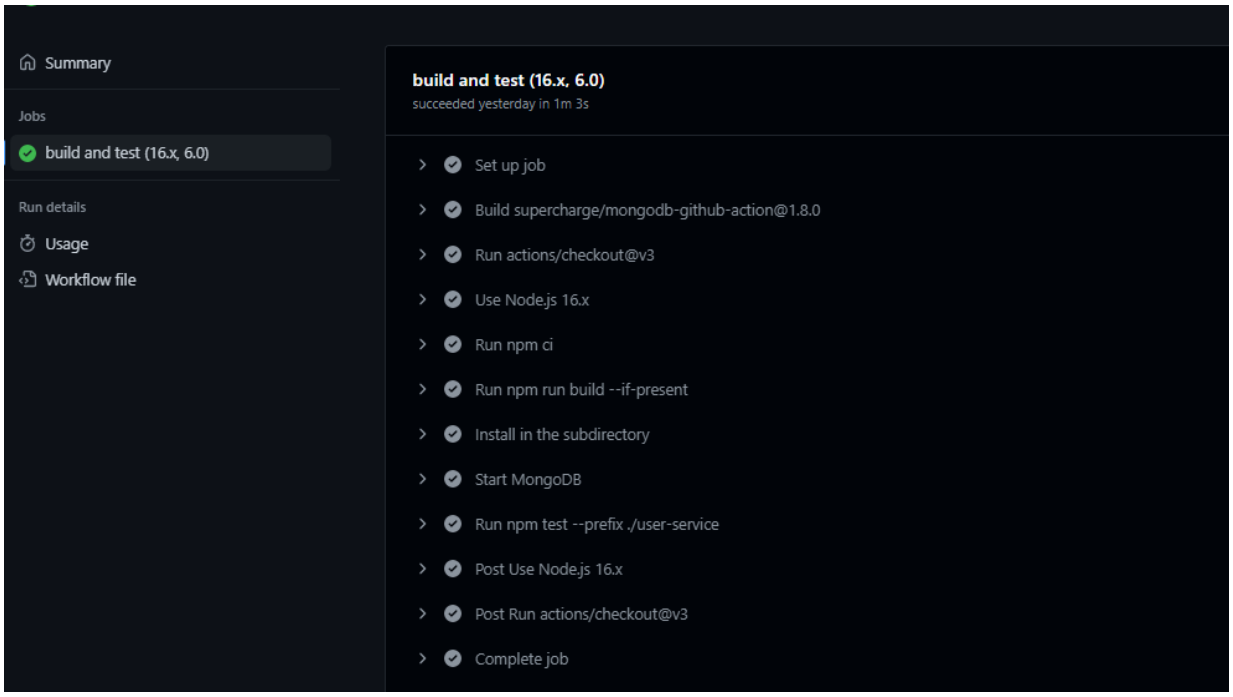


Figure 7.3: Github CI pipeline build and test

To simulate a testing environment with database service:

1) Secrets are preconfigured on Github Actions for starting the microservices.

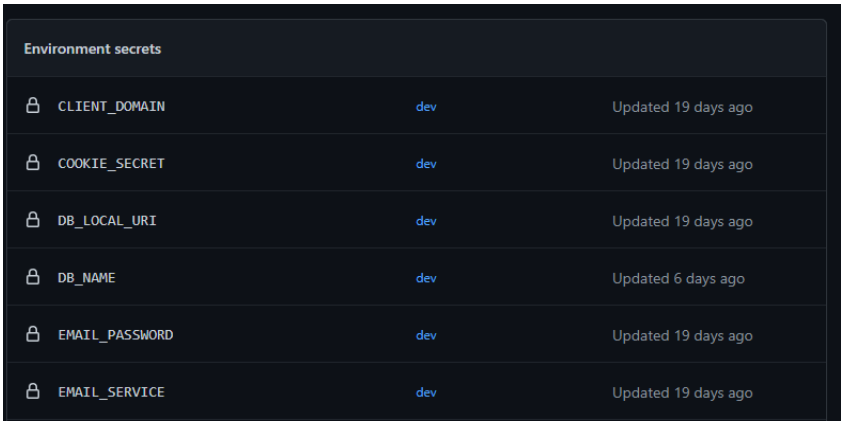


Figure 7.4: Github Action secrets

2) A MongoDB server is started in Github Actions.

Figure 7.5: Github Action MongoDB server

7.3. Continuous Deployment (CD)

7.3.1. GCP Cloud Run

Comparison between different Cloud Providers based on integration requirements:

	GCP Cloud Run	AWS EC2	Vercel
Supports Continuous Deployment	Yes, direct link to github repository.	Yes, but requires additional setup.	Yes, direct link to github repository.
Provides HTTPS endpoint	Yes	More customizable but sophisticated. Needs to go through Certificate Authority (CA) for SSL certificate issuance.	Yes
Support Websockets	Yes	Yes	No, serverless.
Automatically Builds Container	Yes	No	No

GCP Cloud Run is a managed compute platform that allows running of containers directly above Google's scalable infrastructure. As GCP Cloud Run matches all of our integration requirements shown above, we chose GCP Cloud Run as our final deployment approach.

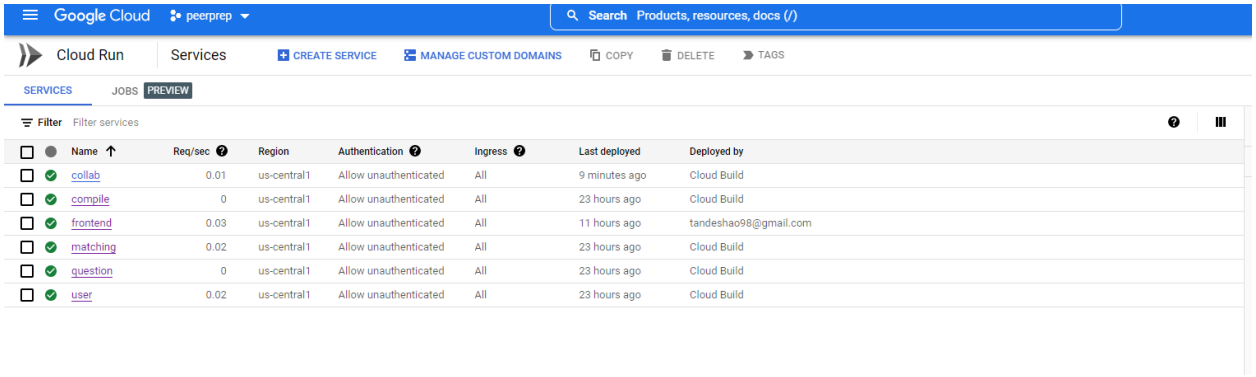


Figure 7.6: GCP Cloud Run deployment

The above image shows an overview of our microservices containers deployed on GCP Cloud Run. PeerPrep is deployed on **GCP Cloud Run** and can be accessed via this link: https://frontend-swougile5q-uc.a.run.app/.

7.3.2. Reflection: Deployment attempt on Amazon Web Service (AWS)

Initially, we deployed our microservices onto separate AWS EC2 Linux instances manually via SSH File Transfer Protocol (SFTP) from our local machine onto the remote linux instance.

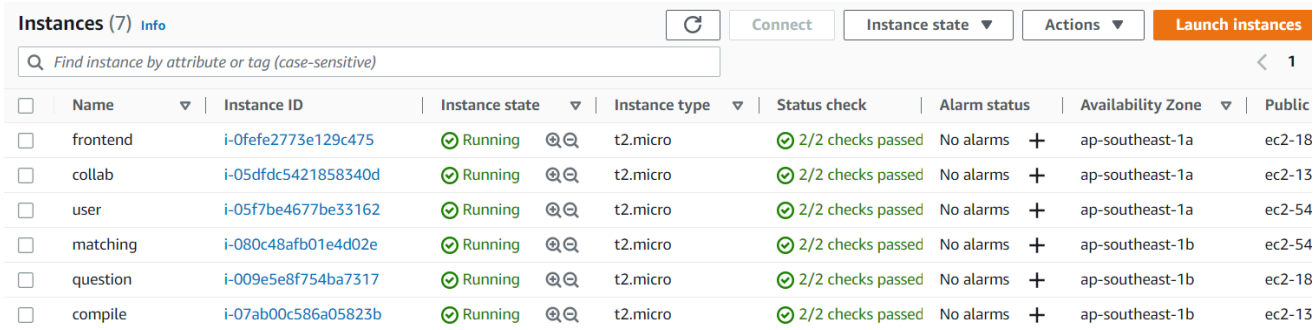


Figure 7.7: AWS deployment

However, due to the voice channel feature in our collaboration service, the service has to be secured with SSL. As a result, all other services have to be secured with SSL as well. Due to our microservices being hosted on the free-tier EC2 instances, it is not viable to submit our digital certificate for verification by a Certificate Authority.

Our initial fix attempt is to register a temporary domain (peerprep.tk) on Freenom, and configure NGINX on our EC2 linux instances to forward https requests received on port 443 to our server port. Then, we map our temporary domain to the EC2 ip addresses on Cloudflare, which offers a free SSL certificate with DDoS protection. However, this fix did not prove to be useful as our microservices are unable to communicate with one another due to the https address being resolved back to the actual http EC2 ip addresses within the code.

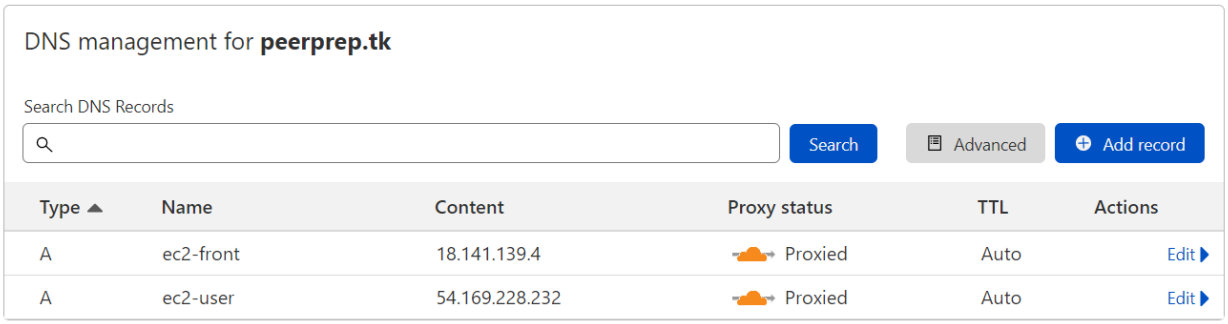


Figure 7.8: AWS EC2 DNS management

Although our initial deployment did not work as expected, we learned to troubleshoot failures when migrating microservices from local to cloud. It was a great learning experience overall.

8. Possible Extensions

8.1. User initiated session

Currently, users can only enter a session if they match with another user. We plan to add a feature where users can create their own session and invite a friend to their room. This feature could be implemented allowing users to create their own sessions and also exposing room API endpoints that allow invitation of users to a room without being paired.

8.2. Video call

Currently, PeerPrep only supports concurrent editing, text chat and voice chat. We plan to add video calls in the future to increase the interactivity between users as non-verbal communication is important as well. This can be implemented by accessing the video media channel and passing the stream to PeerJs over to the other participant.

8.3. Rejoin Room Feature

Currently, users cannot rejoin a room that the user has previously left and the room will be closed when all the users have exited the room. We plan to add the rejoin room feature in the future to increase the user experience between users as they might have faced some internet issues and might want to go back to the room. This feature can be implemented by adding an option for the user to denote that this room should not be closed after all users have exited.

9. Individual Contributions

The following table summarizes the technical and non-technical contributions of the members involved in the project.

Name	Technical Contributions	Non-Technical Contributions
Alvin Tay Ming Hwee	Implement User service Implement integration test (User service) Implement Question service Assist in Collaboration service	Requirements documentation Project final report REST API documentation
Tan De Shao	Implement Frontend Implement Compile service Assist in Collaboration service Set up CD in GCP	Requirements documentation Project final report
Lee Ze Xin	Implement Matching service Set up Cloud Database Implement CI pipeline (linter, build & test) Set up CD in AWS	Requirements documentation Project final report README.md
Li Quan	Implement Frontend Implement Matching service Implement Collaboration service	Requirements documentation Project final report Socket IO documentation

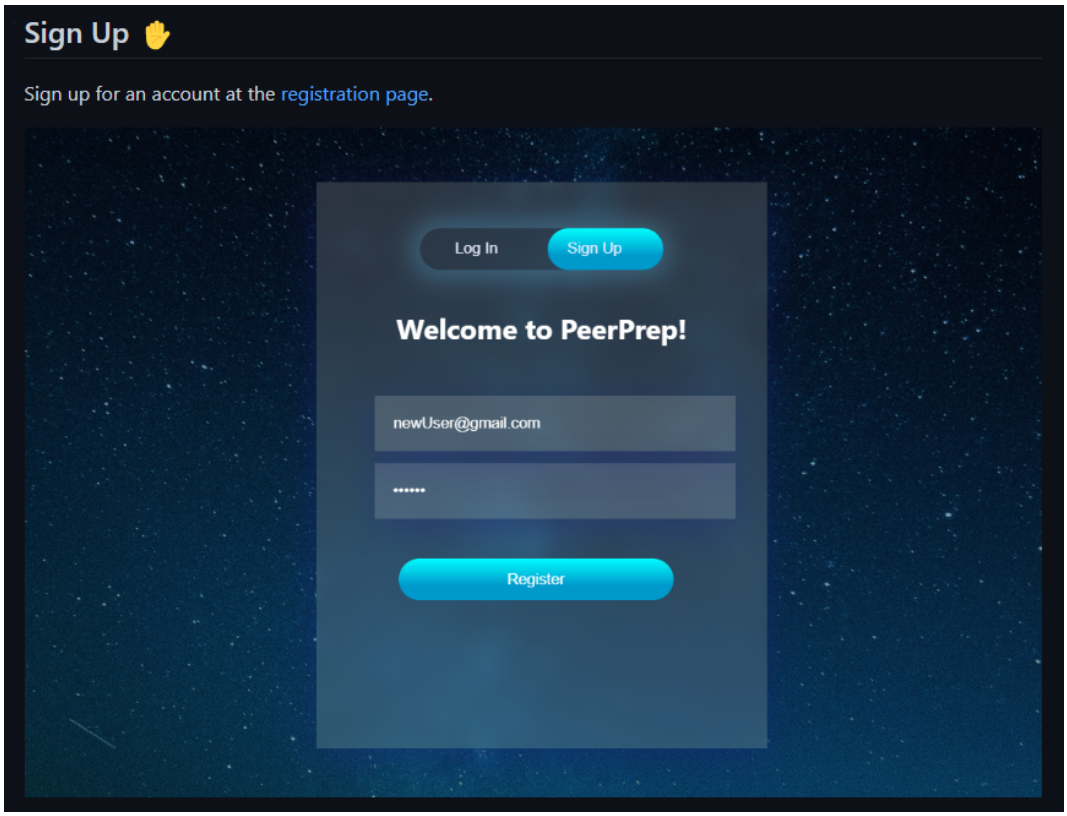
10. Backlogs

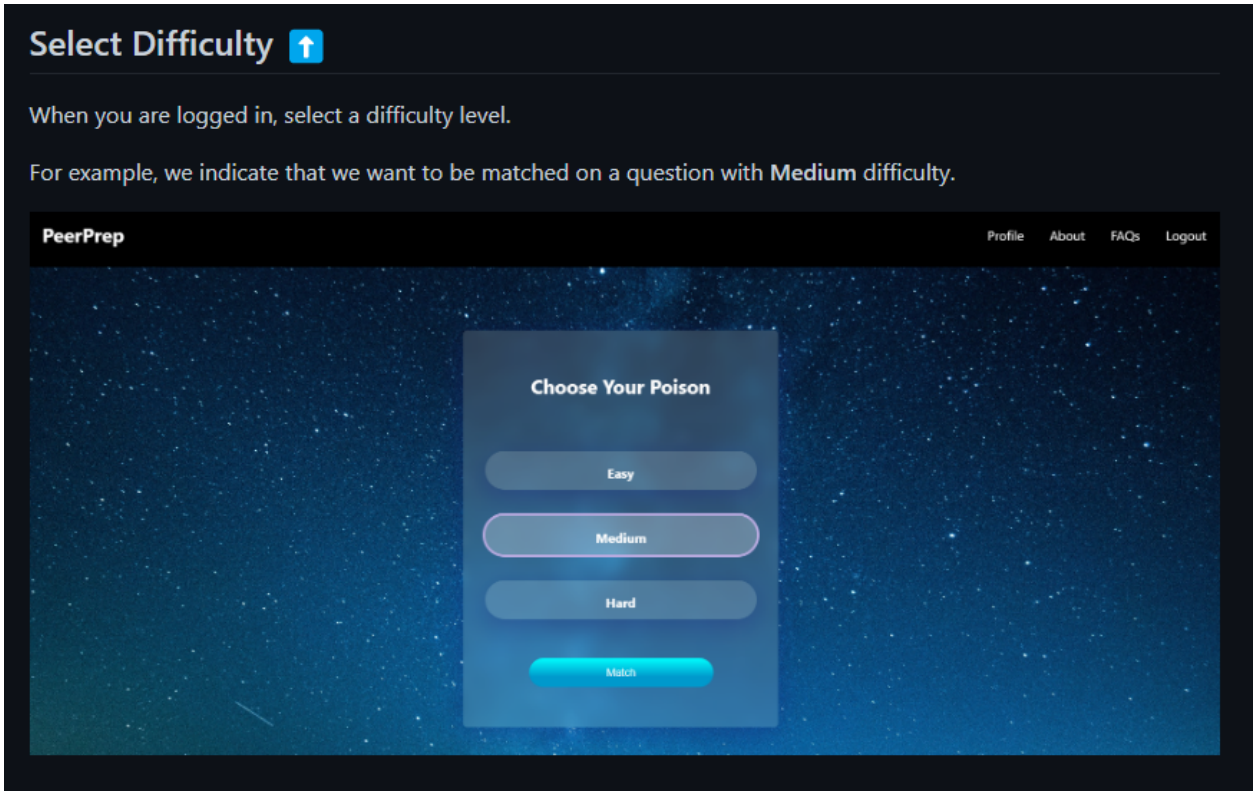
Week	Alvin	De Shao	Lee Ze Xin	Li Quan
3	Setting up and understanding codebase			
4	[User service] - Unique username - Login - Log out - Delete account - Hash & salt password - API documentation	[Ui] - Create Login Page - Create Signup Page	[Matching service] - Difficulty selection backend - Able to match 2 waiting users with similar difficulty	[UI] - Create looking for match page
5	[User service] - Refresh token - Hash & salt refresh token - Silent login discussion & integration - Strict password criteria - Change password - API documentation	[Ui] -Add a loading screen that provides confirmation that the user has submitted the login form. -Create landing page. -Create Ui that allows users to change passwords. [Ui] -Create Ui to enable users to login and signup for an account.	[Matching service] - MongoDB setup - Match users within 30 seconds if valid match - Inform users that no match is found if exceed 30 seconds	[UI] - Integration with landing page and matching page [Documentation] - Edit diagram for matching service
6	[User service] - JWT token verification - Reset password - Integration tests - API documentation	[Ui] -Implement silent login feature. -Create redux storage to store jwt token. -Link up change password backend functionality to frontend. [Documentation] -Write up design considerations regarding the storage of JWT tokens in the client.	[Matching service] - Validation - Testing - Bug fixing	[Matching Service] - Bug fixing [UI] - Integrate Matching Service with Matching Page
7	[Question service] - Design question model. - Create questions with HTML formatting. - Create skeleton code for Java, JS & Python.	[Ui] -Setup Ui to allow users to reset passwords. -Update outdated http methods to interact with user-service.	[CI] - Node 16 build workflow - Linting workflow using SuperLinter - Local ESLint that highlights syntax error	[Matching Service] - Refactor matching service to allow async matching - Simplify matching service login [UI]

				- Add cancel match feature in matching page
8	[Question service] - Get a question based on difficulty level. - Randomize questions. - API documentation	[Ui] -Create collaboration page. -Add resizer and difficulty tag in collaboration page.	[CI] - Unit testing workflow - Setup Github Actions secrets for the microservices. [Matching Service] - UAT	[Collab Service] - Use socket io to place matched users in a room - Use socket io to sync text editing - Testing of data inconsistency for syncing - Look into sharedb and yjs to prevent data inconsistency
9	[History service] - History schema - Get history (with link to question details). - Update history. - API documentation	[Ui] -Add chat widget in collaboration page. -Integrate compilation functionality into frontend. [Compile-Service] - Setup compile-service to call HackerEarth API for code compilation.	[CI + Deployment] - Start MongoDB server in github actions for unit tests. - Start deployment of microservices as individual EC2 instances on AWS.	[Collab Service] - Switch from socket io to yjs-webRTC to sync chat and text editing - Use peerJs to add voice chat feature in the collab room [Ui] - Integrate collaboration page with collab service (voice chat , text editing , chat) [Matching Service] - Bug fixes and implement changes based on team member's comments
10	[User service] - Add maxAge for refresh token cookie. - Case-insensitive username. - Automatically delete expired refresh token in MongoDB - Hash refresh token (SHA512) instead of hash + salt for faster reads. [Question service] - Add questions with images encoded in base64.	[Deployment] -Help out in the deployment of microservices to AWS EC2. [Ui] - Fixed a few rendering bugs in the collaboration page. [Compile Service] - Implement code compilation feature.	[Cloud Database] - Setup cloud databases on MongoDB Cloud for all of the microservices, including frontend, under a MongoDB cluster on Atlas. - Migrate database connections of microservices [CI + Deployment] - Fix user service testing issues	[Matching Service] - Fetch question from Question service and return it to the matched users in the room [Matching UI] - Retrieve question from response and save it to redux store [Collab UI] - Retrieve question from redux store and if there is no question , the UI will use the default_question

11	[Question service] - Add optional id exclusion for getQuestion [Documentation] - Password strength - Hashed and salted password - Hashed refresh token - JWT and Refresh Token expiry	[Deployment] -Deploy all the services into GCP cloudrun. - Setup continuous deployment using cloudrun features. - Setup a bash script that automatically builds the frontend application and pushes it to the GCP cloud console.	[CI + Deployment] - Document how to connect to deployed instances for the different microservices. [All Services] - Setup cloud mongodb paths for all the services. - Modify env files to support toggling of environment between 'development' and 'production'.	[Matching Service & UI] - Remove fetch question from matching service [Collab Service] - Add new listener to trigger fetching of question and emit the question to all participants in room [Collab UI] - Add code for first participant in room to trigger the collab service to fetch question from question service - Add listener to listen to retrieve question from collab-service and update redux store.
12	[Question service] - Fix extra commas in the question constraints [Collab frontend] - Add STUN/TURN server to fix concurrent editing and voice. [Documentation] - Refresh token automatic cleanup - Protected APIs - Email case-insensitivity - Reset password - User service - Question service	[Collab Frontend] - Helped out in the implementation of bug fixes that prevent peers from detecting each other over a public network. [Documentation] - Requirements - Introduction	[Testing] - Helped out with UAT testing connectivity and functionality of features. [Documentation] - Write up README on github repository - Draw Design pattern diagrams and uml sequence diagrams on lucidchart - Write up on Design Patterns and Principles - CI pipeline - CD + reflection	[Collab Frontend] - Revert collaborative editing to use Socket IO - Allow syncing of code languages between room participants [Matching Service] - Fix bug whereby user can match with another user that has time out (30 secs) - Fix bug whereby user has to wait before submitting a new match request after timeout [Collab Service] - Add JWT authentication - Fix bug whereby user does not exit the room when tab is closed

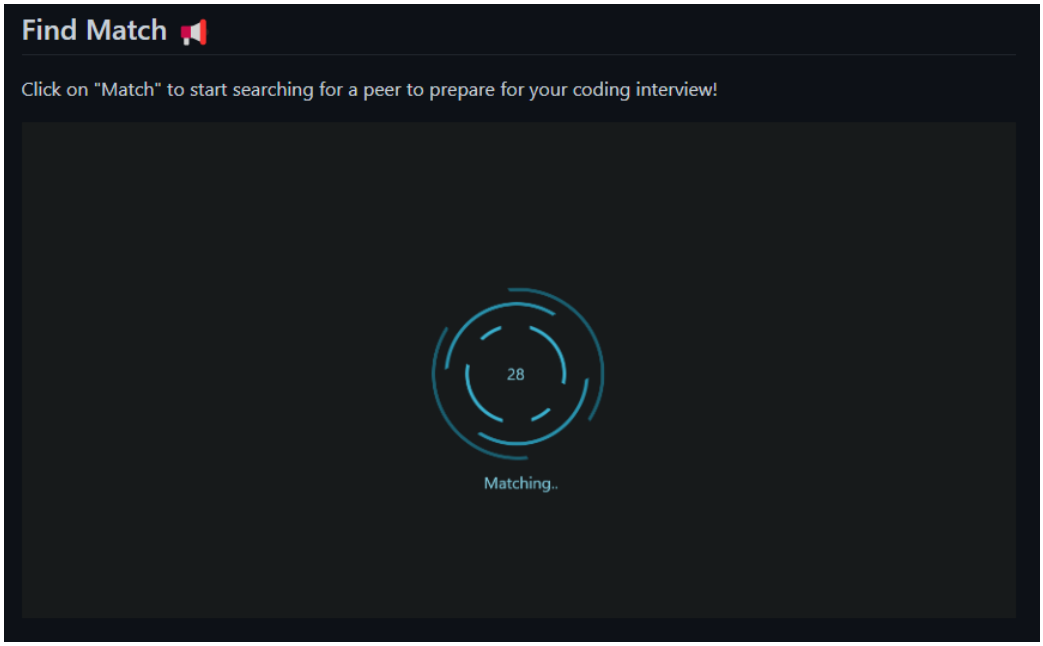
11. Product Screenshots



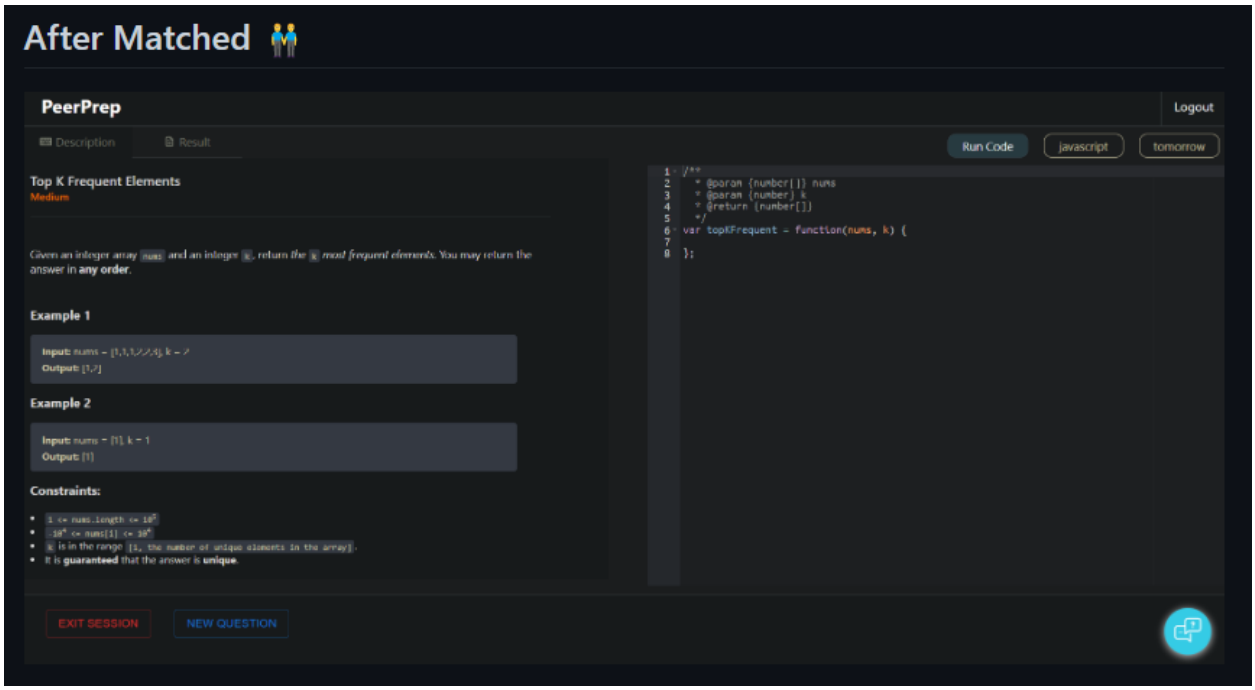


Find Match

Click on "Match" to start searching for a peer to prepare for your coding interview!



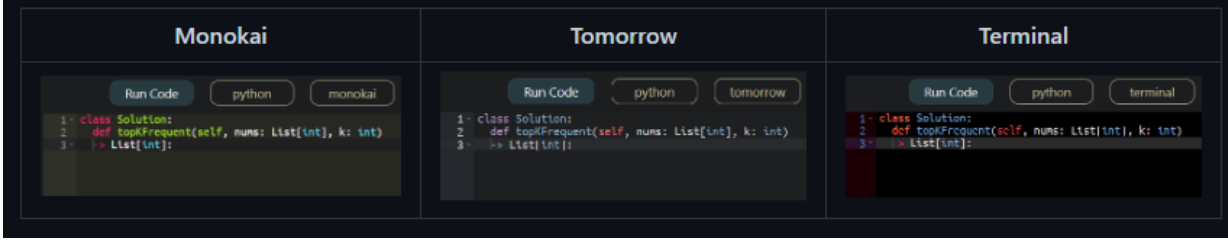
After Matched



Toggle between Programming Languages

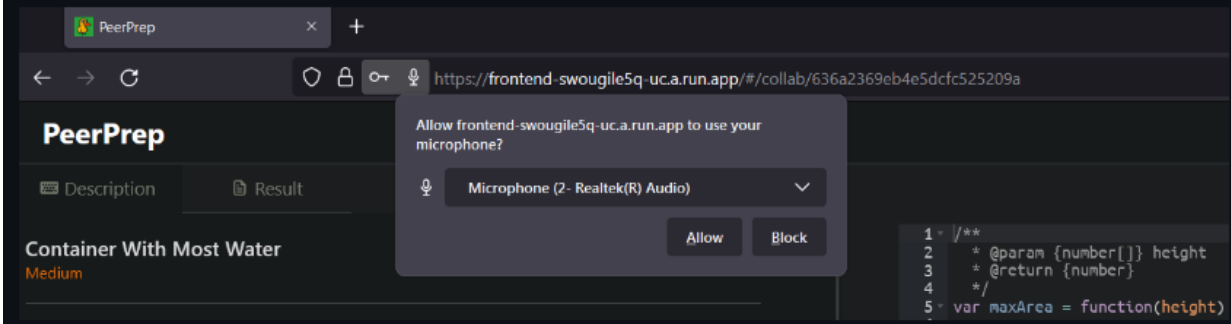
You can choose your desired language by toggling between `Java`, `JavaScript` and `Python`.

Toggle between Color Themes



Connect to voice

If you'd like to, you can allow PeerPrep to access your microphone for voice collaboration with your Peer.



Edit together, in real time

Compile and run your code 🔨 🏃

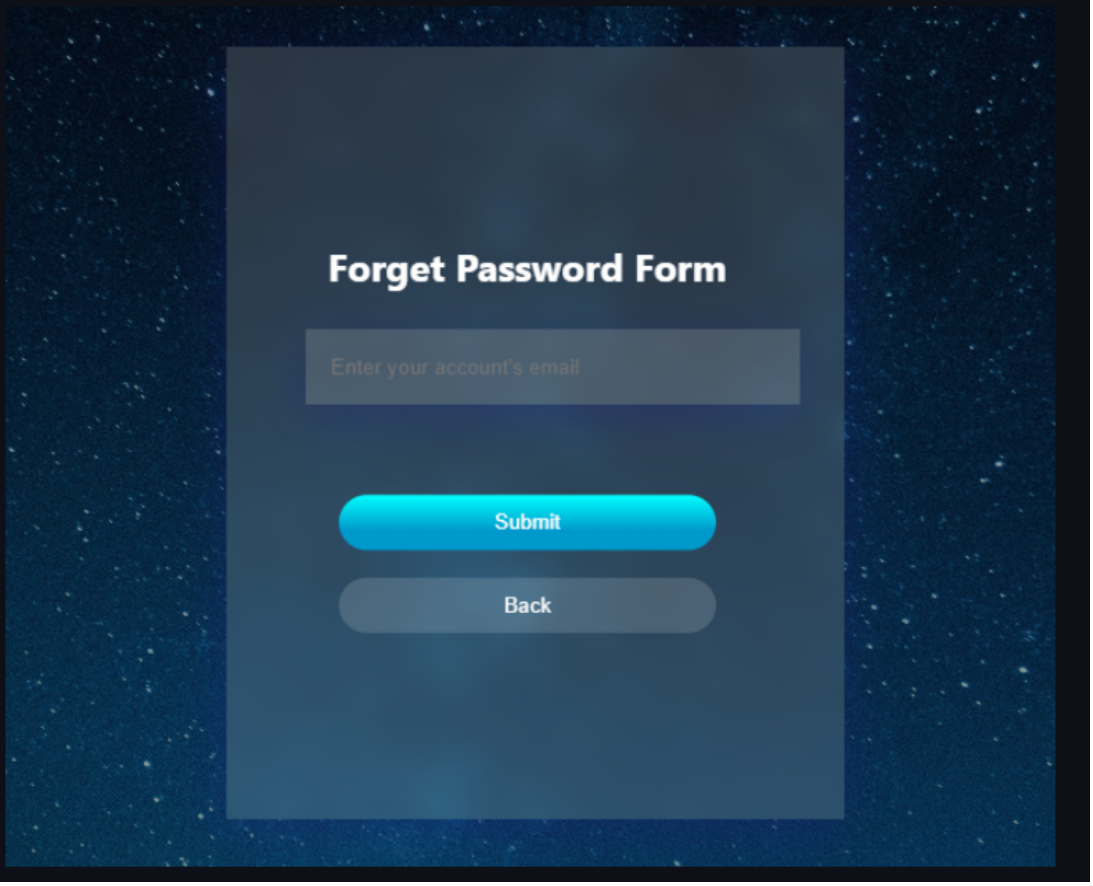
Click on "Run Code" to compile your current code.

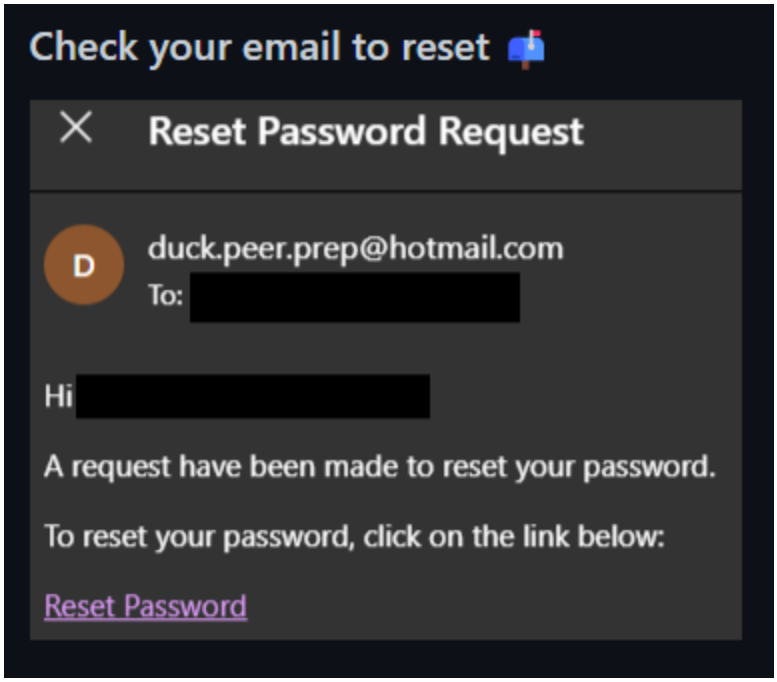
Run Code

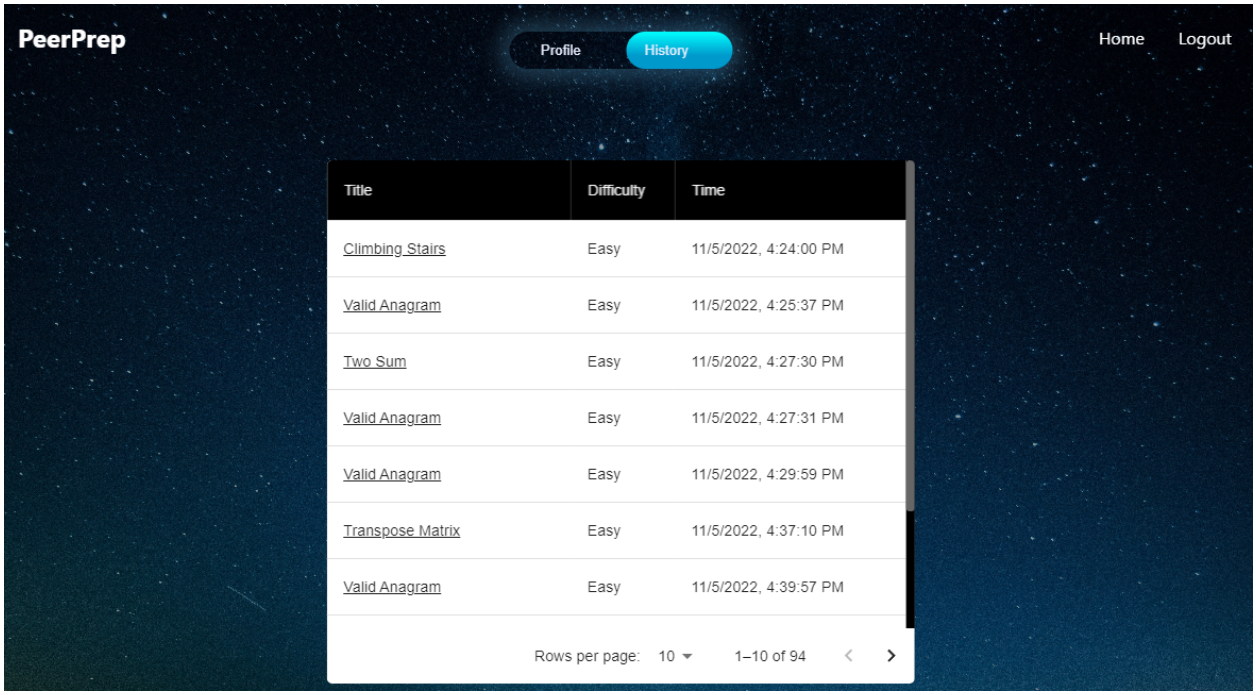
Example: Compiling and running in Java

What if you forgot your passcode 😨

Fill up the Forget Password Form 😁




Check your email to reset
Reset Password Request
D
duck.peer.prep@hotmail.com
To:
Hi
A request have been made to reset your password.
To reset your password, click on the link below:
Reset Password


PeerPrep
Profile
History
Home
Logout
Title
Difficulty
Time
Climbing Stairs
Easy
11/5/2022, 4:24:00 PM
Valid Anagram
Easy
11/5/2022, 4:25:37 PM
Two Sum
Easy
11/5/2022, 4:27:30 PM
Valid Anagram
Easy
11/5/2022, 4:27:31 PM
Valid Anagram
Easy
11/5/2022, 4:29:59 PM
Transpose Matrix
Easy
11/5/2022, 4:37:10 PM
Valid Anagram
Easy
11/5/2022, 4:39:57 PM
Rows per page: 10
1–10 of 94

PeerPrep

Profile About FAQs Logout

AT A GLANCE, PEERPREP IS

An interview preparation tool that helps you ace that job interview

With a user-friendly interface and a quick and efficient matching system, PeerPrep helps you find peers to practice technical coding interviews with.

PeerPrep

Profile About FAQs Logout

Frequently Asked Questions

- What is the purpose of PeerPrep? +
- How do I use PeerPrep? +
- Who created PeerPrep? +
- What kind of features does PeerPrep provides? +
- The voice chat doesn't seem to be working. What should I do? +
- Is there a fast way for me to compile code written in PeerPrep? +