Project Report: PeerPrep

CS3219 G35

Celeste Tigerlily Cheah Kae (A0245928R)
Sim Zhe Feng Kenneth (A0233254M)
Law Song En Timothy (A0233564E)
Tang Yuanyuan (A0240044X)
Kang Yue Hern (A0234795R)

1 Individual Contribution

Name	Contributions
Celeste Tigerlily Cheah Kae	**Technical Contribution** Frontend: ● History Service ○ Completed questions table, completed question modal ○ Progress counter and chart ● Matching Service ○ Finding a match page with spinner and facts ● Collab Service ○ Collab room page ○ Question description display, code editor, code results display ○ Modals for Change question, AI code generator, and leave room ● User Service ○ Login and sign up pages ○ Login and sign up forms Backend: ● Question Service ○ Set-up endpoints for server ○ Add get random question by difficulty, get random question by category, and get all questions
	Non-Technical Contribution Report: ● Section 4.2 ● Section 5 ● Section 6.1.1, 6.6, 6.7 UI and Design: ● Figma wireframing and designs for PeerPrep app
Sim Zhe Feng Kenneth	**Technical Contribution** Frontend: ● Messaging Chat Application Backend: ● Runtime Service ● Chat Service ○ Chat-room ● Deployment(Matching Service, Chat Service)

	● Matching Service ● User Service(Authentication)
	Non-Technical Contribution Report: ● Section 6.3.3, 6.6.4, 6.6.5, 6.6.6
Law Song En Timothy	**Technical Contribution** Frontend: ● Collab Service ○ Video-call ○ Change Question ○ Leave room Backend: ● User Service (Authentication) ● History Service
	Non-Technical Contribution Report: ● Section 6.2 ● Section 6.6.1, 6.6.2, 6.6.3 ● Section 6.7.1, 6.7.2
Tang Yuanyuan	**Technical Contribution** Frontend: ● Dashboard ○ Layout of dashboard ○ Navigation buttons ● Matching Service ○ Selection of difficulty level ○ Unsuccessful match page ○ Successful match page ● Collaboration Service ○ Change question modal ○ AI code generator modal ● User Service ○ Edit profile modal and feedback modal ○ Log out modal ● Chat Service ○ Video chat display
	Non-Technical Contribution Report: ● Section 2

	● Section 3 ● Section 4.1
Kang Yue Hern	**Technical Contribution** Frontend: ● Collaborative Code Editor ● AI Code Generator ● Display of Question in Collab page Backend: ● Collaborative Code Editor ○ Firebase Realtime Database synchronization ● Jenkins Devops Pipeline ○ Jenkinsfile ● Kubernetes Deployment of Backend Services ○ User Service, History Service, Questions Service ● Kubernetes setup and deployment of Backend Databases ○ MongoDB, PostgreSQL ● Kubernetes Deployment of Frontend ● Kubernetes Scaling
	Non-Technical Contribution Report: ● Section 6.8 ● Section 8.1, Section 8.3

2 Introduction

2.1 Background

With tech sector growth and expanding job opportunities, an increasing number of students are pursuing technical roles. Consequently, they frequently face technical interviews, which pose a challenge for many applicants. The challenge encompasses various complexities, including the imperative to adeptly handle posed questions and the difficulty in articulating their comprehension effectively.

2.2 Purpose

The purpose is to provide students with a user-friendly platform that enables real-time collaborative problem-solving. This platform provides an extensive collection of practice questions specifically curated for technical interviews, aiming to enhance students' ability to solve challenging problems and to articulate critical technical concepts with clarity. Additionally, it empowers students to explore alternative solutions with collaboration.

2.3 Scope

Since our project targets university students aiming to improve their technical interview skills, the scope of the interview questions will be a curated selection from the actual LeetCode questions. Students should be able to choose the difficulty level of the question and the platform should allow only two peers who have chosen the same difficulty level to be connected per session.

Upon a successful match, students should be able to see a randomly selected question of the chosen difficulty level, and they should be able to engage in a video call and a text-based chat. PeerPrep should facilitate a collaborative programming environment, allowing them to code together in real time. PeerPrep should be able to generate AI code to assist students during the coding process. Upon completion of a question, students should be able to review their attempted questions' history on the homepage.

3 Overall Description

3.1 Product Perspective

PeerPrep stands as a user-friendly platform for university students to practice technical interview questions with peers, bridging the gap between theoretical knowledge and practical application.

3.2 Product Features

PeerPrep's key features are:

1. User Service: Profile management
2. Question Service: Comprehensive question repository
3. Matching Service: User pairing based on difficulty level
4. Collaboration: Real-time code editing
5. User-friendly Interface
6. Deployment Flexibility: Containerization with Docker

Additional features included are:

1. Chat + Video: Real-time text-based chat and video streaming
2. Improved Code Editor: Code formatting, syntax highlighting
3. Runtime service: Code execution
4. Generative AI Integration: Generate AI Code
5. History Service: User activity log
6. Deployment + Horizontal Pod Autoscaling

3.3 Operating Environment

The server-side components can function in both Linux and Windows operating systems. The client-side components operate in web browser environments, including Google Chrome, Mozilla Firefox and Apple Safari.

4 Requirements Specification

4.1 Functional Requirements

4.1.1 User Service

ID	Functional Requirement	Priority
US-01	The system should allow users to **create an account** with username, password and email.	High
US-02	The system should ensure that every account created has a **unique username**.	High
US-03	The system should allow users to **sign in** with their registered username and password	High
US-04	The system should allow users to **log out** of their account from the homepage.	High
US-05	Users should be prompted to input correct login details upon unsuccessful login.	Medium
US-06	The system should allow users to **delete their account**.	Medium
US-07	The system should allow users to **change their email**.	Medium

4.1.2 Question Service

ID	Functional Requirement	Priority
QS-01	The system contains questions with 3 difficulty levels (easy, medium, hard).	High
QS-02	The system should allocate a random question of the selected difficulty level.	High
QS-03	The system should allow users to generate a new question while in the same room.	Medium
QS-04	The system should display questions with a title, description, difficulty and category as its body.	Medium

4.1.3 Matching Service

ID	Functional Requirement	Priority
MS-01	The system should allow users to **select practice questions** from 3 difficulty levels (easy, medium, hard).	High
MS-02	The system should **match users** with the same difficulty level when they request for a match.	High

MS-03	The system should **direct** matched users to the same room.	High
MS-04	The system should implement a 30 second **timer** to attempt to match.	High
MS-05	The system should inform users if a valid match cannot be found within 30 seconds.	High
MS-06	Users should be **redirected to the homepage** upon an unsuccessful match.	High
MS-07	The system should allow users to **cancel matching attempt** before the timer ends.	Medium

4.1.4 Collaboration Service

ID	Functional Requirement	Priority
CS-01	Users should be provided with a **code editor** to type his/her code.	High
CS-02	The system should allow two users in the same room to edit code **concurrently** in the code editor.	High
CS-03	The system should support **syntax highlighting** and **code formatting** for multiple programming languages.	Medium
CS-04	Users should be able to generate **AI code** to the provided question.	Low
CS-05	The system should allow users to **leave** the room anytime during the session.	Medium
CS-06	**Question** and **Code language** should be **synchronized** between both users in the room (a change in one user should cause a change in the other)	Medium
CS-07	All details in the Collaboration room (code, language, question) should **persist upon refresh**.	Medium

4.1.5 Communication Service (Chat + Video)

ID	Functional Requirement	Priority
CS-01	Users should be able to send and receive **text messages** to each other while in the same room.	Medium
CS-02	Chat history should **persist upon refresh**.	Medium
CS-03	Users should be able to **video-call** each other.	Medium

CS-04	Users should be able to **enable or disable** their camera and audio in the video-call.	Low
CS-05	Users should be able to see their **partner's username**.	Low

4.1.7 Runtime Service

ID	Functional Requirement	Priority
RS-01	The system should **run** the users' code.	High
RS-02	The system shall **display the output** in the collaborative workspace where users can view.	High
RS-03	The system should **display code errors** in the collaborative workspace if users make any mistakes in their code.	Medium

4.1.6 History Service

ID	Functional Requirement	Priority
HS-01	Users should be able to see his/her **previously attempted questions** in the homepage, including the question's title, difficulty level and the date attempted.	Medium
HS-02	Users should be able to view the **number** of questions that he/she has attempted for each difficulty level.	Low
HS-03	Users should be able to **save** their code progress for each question upon changing questions and leaving the room.	Medium

4.2 Non-Functional Requirements

4.2.1 User Service

ID	Non-Functional Requirement	Priority	Quality	How It's Achieved
US-01	Users' password on the database should be protected	H	Security	Passwords are encrypted using BCrypt hash function before storage
US-02	The database should handle storage of 500 unique users.	H	Scalability	Databases are deployed on GKE which can be replicated based on demand.
US-03	The database should be protected against unauthorized account deletion or editing of account information.	M	Security	Sensitive routes on frontend and backend are protected (requires verification of JWT)
US-04	Users should view a confirmation message when they request to delete their account.	L	Usability	A Delete modal appears for users to confirm deletion.

4.2.2 Matching Service

ID	Non-Functional Requirement	Priority	Quality
MS-01	Users should return to Dashboard when matching takes more than 30 seconds.	M	Usability

4.2.3 Question Service

ID	Non-Functional Requirement	Priority	Quality	How It's Achieved
QS-01	Questions and their descriptions should be displayed in a consistent and aesthetic manner.	M	Usability	Question bodies are stored as HTML strings to preserve formatting.
QS-02	The Question Service should have the ability to accommodate an increase in the number of questions.	M	Scalability	

4.2.4 Collaboration Service

ID	Non-Functional Requirement	Priority	Quality	How It's Achieved
CS-01	Users should enter the collab room within 5 seconds of a successful match.	M	Performance	
CS-02	The code editor should only experience a 5 second delay in response between the two Peers.	H	Performance	Firebase Realtime Database offers low latency in synchronization of code.
CS-03	The code editor should provide a range of programming languages for users to choose from.	H	Usability	CodeMirror supports multiple programming languages.
CS-04	Users should be reminded to save their work when they want to 'Change Question' or 'Leave Room'.	L	Usability	Reminder modal will appear to prompt user.

4.2.5 Chat Service

ID	Non-Functional Requirement	Priority	Quality
CHS-01	The chat should only experience at most a 3 second delay in response between the two Peers.	H	Performance
CHS-02	The video chat should not have significant lags.	H	Performance

4.2.6 History Service

ID	Non-Functional Requirement	Priority	Quality	How It's Achieved
HS-01	Users should be able to view previously completed questions along with details (eg. code solved with, programming language used, question attributes)	M	Usability	All relevant details are saved to the History table.

4.2.7 Runtime Service

ID	Non-Functional Requirement	Priority	Quality	How It's Achieved
RS-01	The code execution mechanism should provide a response within 5 seconds to ensure a smooth user experience.	M	Performance	Relevant docker images are pre-pulled into our GCE Virtual Machine, allowing for the prompt mounting of code onto the image for instant execution.

4.2.8 General Non-Functional Requirements

ID	Non-Functional Requirement	Priority	Quality	How It's Achieved
GE-01	The UI should be user-friendly and functions should be intuitive for users to navigate.	M	Usability	Wireframing using Figma with consistent color themes
GE-02	The UI should be standardized across all screens to ensure consistency and a more complete user experience.	L	Usability	
GE-03	The UI should include messages to confirm a user's action to prevent accidentally doing something.	L	Usability	Confirmation modals are displayed upon any sensitive action.
GE-04	The app should be able to run on different browsers.	H	Interoperability	

4.3 Availability

ID	Non-functional requirement	Priority
AV-NFR-01	The application should maintain an uptime of 99% during peak hours	High

Availability overview

Service	Failure detection	Replication
Frontend	Kubernetes	Deployment
User microservice	Kubernetes	Deployment
Question microservice	Kubernetes	Deployment
History microservice	Kubernetes	Deployment

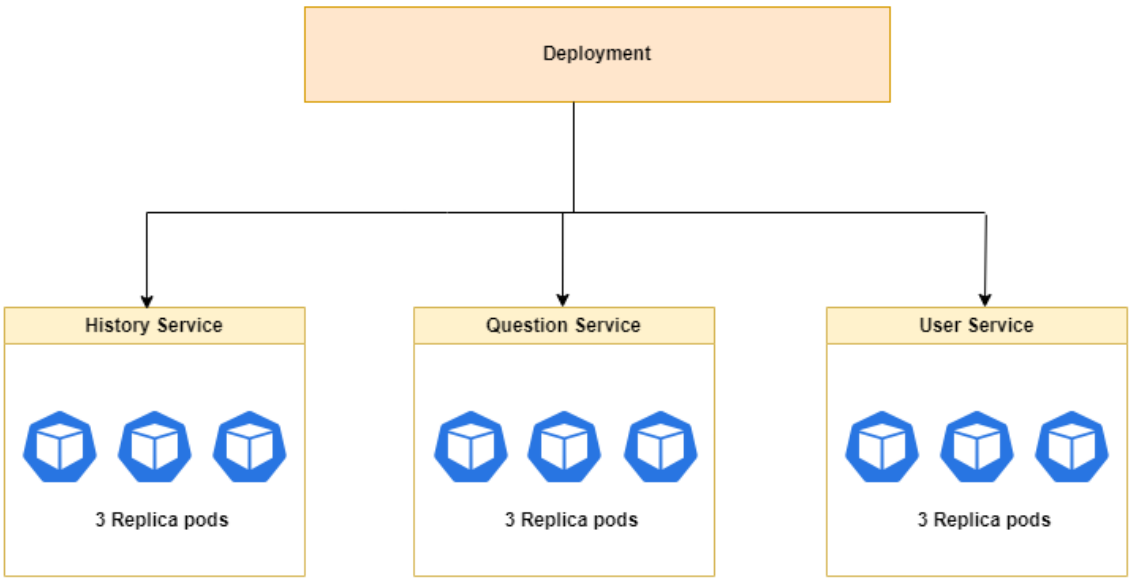


Figure 4.3.1 Microservices Deployment Architecture with Replicated Pods

Kubernetes uses Replica Sets (or Deployments, which manage Replica Sets) to ensure that a specified number of pod replicas are running at all times.
By declaring that we expect 3 replicas for each deployment, we lessen the chances of any downtime since the Replica Set automatically replaces it with a new pod to maintain the desired number of 3 replicas.

4.4 Scalability

Scalability will be elaborated in Section 6.7.5 Scaling below.

5 Software Development Process

5.1 Weekly Sprints

In our team, we decided to have check-ins where we shared our current progress, discussed any problems encountered, and what our next tasks would be. During this time, we also would discuss the current backlog we were facing and which aspects might be of higher focus. A lot of our tasks were interlinked so we also regularly kept each other up to date and worked together through our group chat.

5.2 Milestones

Milestone	Description	Weeks
1	• Design frontend • Set up Matching service • Set up User service	7, 8
2	• Set up Collab service • Set up Question service • Set up Chat service	9, 10
3	• Finalize all frontend pages • Dockerize application with Kubernetes • Deploy application	11, 12
4	• Testing of app and fixing any bugs • Documentation and report	13

5.3 Development Process

For this project, our team opted for a monorepo as opposed to a multi repo, for the microservices implementation. What appealed to us about monorepo is that since we're working as a team, it offers both code reusability and better collaboration. Namely, code is more easily accessed across different services and members can see other service's code without much trouble too. Additionally, there is a singular version of history for our monorepo. With a singular version history, our changes can be tracked more easily and through Git, we're able to roll back to previous versions, if needed. However, one thing to note was that by using a monorepo, our build and deployment took longer because a change to one service requires redoing the build and deployment again.

6 Application Design

6.1 Tech Stack

6.1.1 Frontend

Our frontend development tech stack included the following along with our rationale behind choosing them:

Component	Resource	Rationale
Frontend	Next.js with React	• Ease of setup with few configurations needed • Inbuilt file-based routing • Auto code-splitting reducing page load-time • Inbuilt CSS support (modules etc) • Component-based approach which helps reduce redundancies and complexities • Offers server-side rendering which enables web apps to run faster
UI Styling	Tailwind CSS	• Highly customizable design • Ease of use with its predefined set of utility classes • Less complicated work environment since separate stylesheets are not required • Offers design consistency by easily replicable styling across components or pages • Offers built-in styling which reduces time needed to write custom code, allowing more focus on design
	ChakraUI	• Component-based architecture which reduces the need for designing from scratch • Customisable components • Aesthetic components • Components are mobile-responsive out of the box
Code Editor	CodeMirror	• Specialized in code editors with collaboration features which satisfies desired specifications • Works with multiple programming languages and theme add-ons
Chat	SocketIO	• Enable real-time communication, allowing messages to be sent and received instantly without the need for constant polling. • Provide features like rooms or channels,

		allowing users to be organized under specific groups.
Video	PeerJS	• Easy configuration • Peer-to-peer direct connection to reduce server load

6.1.2 Backend

We mainly use Google Kubernetes Engine (GKE) for our microservices, as well as Google Compute Engine (GCE) for services that work better with virtual machines as compared to running on a container.

Microservice	Hosted On	Purpose
User Management	Google Kubernetes Engine	Manage user functions such as register/login/update profile
Real-Time Matching	Google Kubernetes Engine	Match users with each other
Question Service	Google Kubernetes Engine	Question Repository
History Service	Google Kubernetes Engine	History Repository
Chat Service	Google Kubernetes Engine	Facilitate chat between users in the room
Runtime Service	Google Compute Engine	Run user code
Jenkins	Google Compute Engine	Facilitate CI/CD

Microservice	Backend Language / Framework	Database
User Service	Express.js	PostgreSQL
Question Service	Express.js	MongoDb
Chat Service	SocketIO	-
Matching Service	SocketIO	-
Runtime Service	Express.js, Docker	-

History Service	Express.js	PostgreSQL

Justification for Frameworks and Databases

Express.js	for services which exposes REST apis due to its 1) support for routes / endpoints, 2) middleware, 3) simplicity.
PostgreSQL	Relational database for User and History services to 1) easily enforce Primary Keys, 2) support more complex queries (e.g. queries that do not query by primary keys).
MongoDb	Non-relational database for Question service to store question data, since no complex queries are required (only querying by Collection id).

6.2 Overall Architecture Design

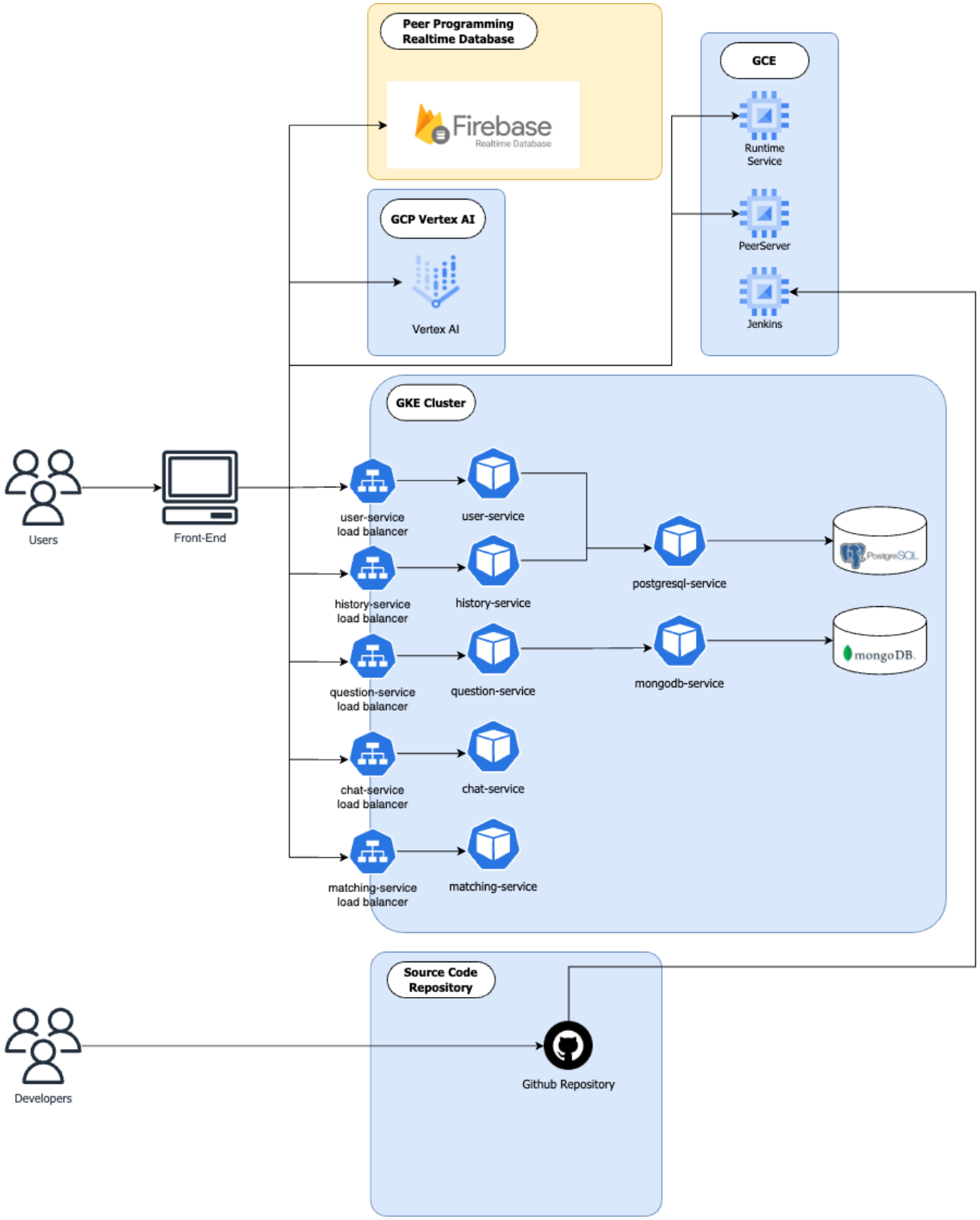


Figure 6.2.1 Overall Architecture Design of PeerPrep

6.3 Design Patterns

6.3.1 Data Transfer Object Pattern

```ts
type UserCreateRequestType = {
  username: string;
  password: string;
  email: string;
};

type UserEditRequestType = {
  email: string;
};

type UserResponseType = {
  id: string;
  username: string;
  role: string;
  email: string;
};
```

```ts
export const editUser = async (
  user: UserEditRequestType, id: string
): Promise<UserResponseType | undefined> => {

  return await prisma?.user.update({
    where: {
      id: id,
    },
    data: {
      email: user.email,
    },
    select: {
      id: true,
      username: true,
      role: true,
      email: true,
    },
  });
};
```

Figure 6.3.1 DTO Pattern Used in Editing of User

The DTO pattern allows us to specify and restrict the fields that are sent to and fro the User Service. For example, upon successful edit of a user, we return UserResponseType which contains all user fields but not the password for security.

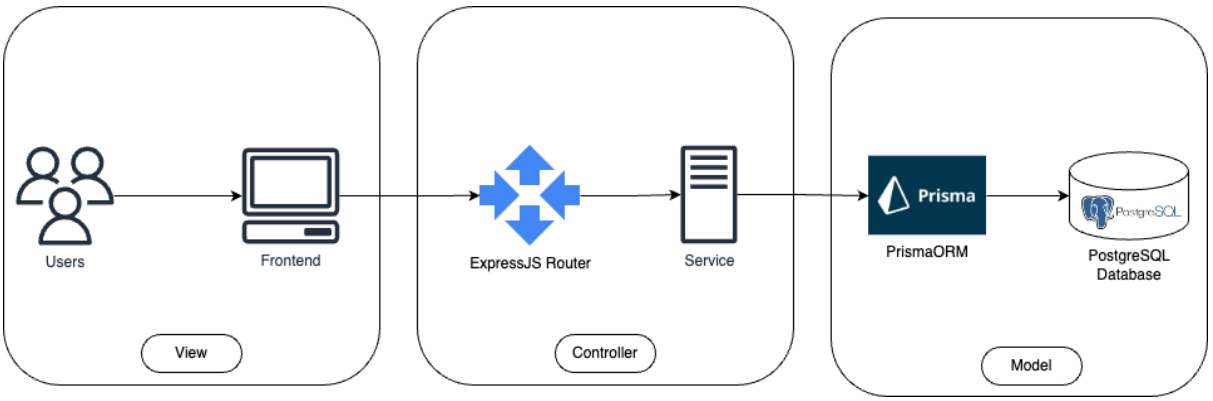
6.3.2 Layered architecture - Model View Controller Pattern (MVC)

We adopted the MVC pattern for services that require the storage, retrieval and display of information to end users (namely User, Question, History Services).

6.3.2.1 Rationale for choosing the MVC Pattern

The following details the benefits that motivated our adoption of this pattern.

Benefits	Justifications
Better modularity and reusability	MVC enforces the separation of concerns principle.
Accelerates developmental processes	As coupling between the Model, View and Controller components are reduced, each team member will be able to work on each of the 3 components concurrently.
Potentially reduces code duplication	As web applications are constantly evolving, we may plan to create multiple Views for our model to serve different devices. In light of this possible project enhancement, MVC reduces the need for code duplication as it separates our display from our business logic and data components.
Improves our ability to conduct unit tests	Adopting the MVC pattern helps us reduce coupling and improve cohesion by separating the input, output and processing components.
Enhances UI modification capabilities	Since components are more loosely coupled, changes in the View components will less likely require changes in the Model component. This is essential as UI changes are usually frequent for web applications.



MVC Architecture in User Service

As seen above, we adopted a strict layered architecture, where each layer can only communicate with its adjacent layers.
Frontend will call REST api endpoints in ExpressJs router, which will call the functions in the services file.

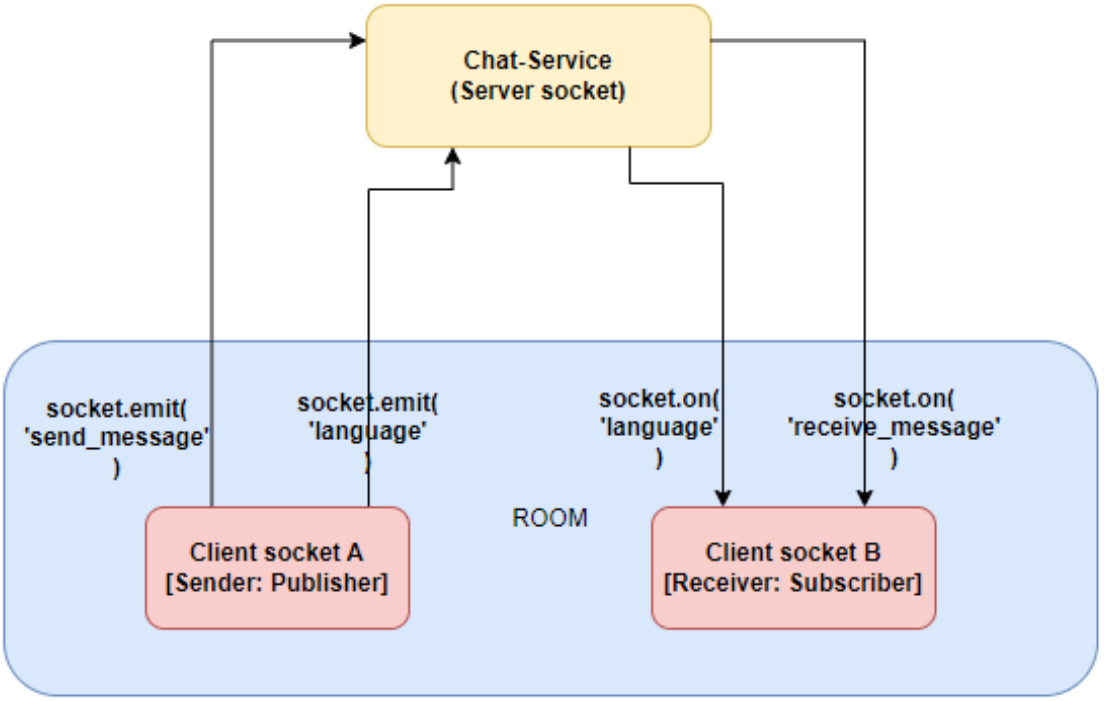
The services file calls upon Prisma commands to access our database.

6.3.3 Publisher-Subscriber Pattern - Chat Service

To facilitate real-time communication between matched users via our Chat Service, we have decided to utilize the Publisher-Subscriber Pattern in our application.

6.3.3.1 Rationale for choosing the Publisher-Subscriber Pattern

Benefits	Justifications
Decoupling of components	Pub-Sub allows for decoupling of the components involved in the communication process. Publishers and subscribers do not need to have direct knowledge of each other. This promotes a more modular and scalable system.
Extendability: Option to include more users in each chat room.	This can be implemented by having more user sockets joining the same room and listening on the "*receive_message*" event.
Asynchronous communication	Pub-Sub supports asynchronous communication, allowing subscribers to carry out their tasks without waiting for the publishers to acknowledge. This can enhance the overall responsiveness and efficiency of the system.



1. Publishers can be modeled as matched users who are sending messages.
2. Subscribers can be modeled as matched users who are receiving messages.
3. Events can be modeled as *'send_message'* and *'language'* to notify subscribers of any incoming message/change in coding language.

6.3.4 Exclusive-Pair pattern - Video-call

We applied the Exclusive-Pair pattern in the implementation of video-call through the use of WebRTC library PeerJS. PeerJS connects two users directly instead of having a server as the middleman.

It supports bidirectional communication between exclusively paired sockets and sockets within each pair can only connect to one other socket.

6.3.3.1 Rationale for choosing the Exclusive-Pair Pattern

Benefits	Justifications
Reduced server load	Having a server as the middleman between two video-calling clients will be very demanding on the server (compared to chat). Hence, we went with having peer-to-peer direct connection for video-call instead.

Also, allowing only two users in a video-call is sufficient for our requirements.

6.4 Frontend Design

We developed our Frontend using React.js, a component-based library. Hence, we opted for a component-based design structure for our architecture. The advantage of using a component-based design is that we could develop our Frontend faster and with better consistency since components can be reused across pages, without much hassle. The figure below is an architecture diagram for our Frontend.

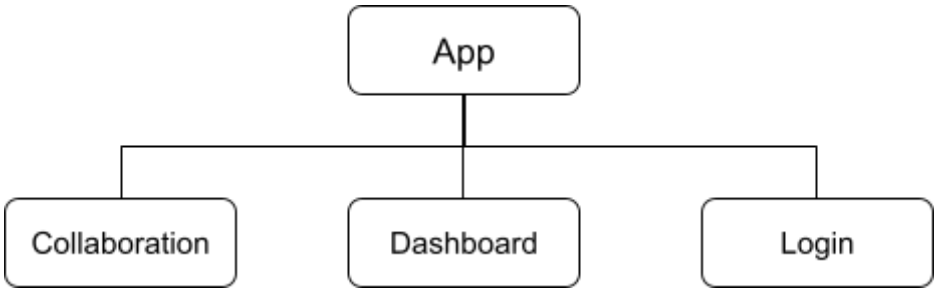


Figure 6.6.1 High-level View of Frontend Architecture

Our app is divided into the three main pages: the collaboration, dashboard, and login page. Each page has components associated with it.

Firstly, the collaboration page is where users work with a Peer on a coding question. They have the option to chat with the Peer, view the question, edit in the code editor simultaneously, test the results, and generate code solutions with AI. Our collaboration page features Chat, CodeEditor, CodeGen, QuestionDisplay, and CodeResults in order to provide the above mentioned functionalities.

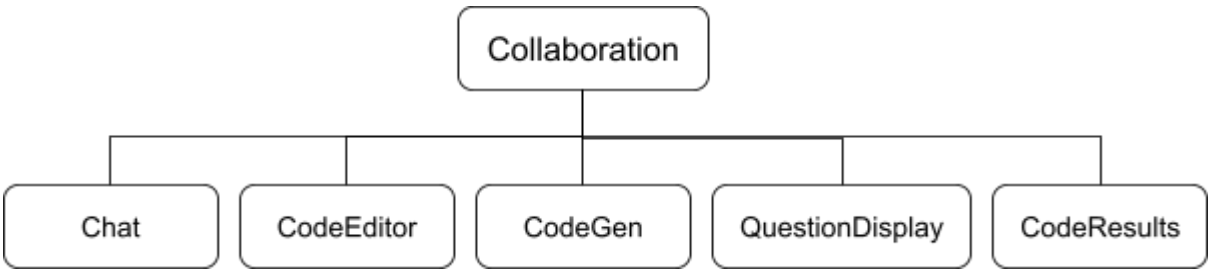


Figure 6.6.2 Collaboration Page Components

Following, the dashboard page is where users start upon logging into their accounts. On their dashboards, they can view previously completed questions, their progress with question completion, their profile and edit it, logout, or get matched to solve a coding question. To implement these functionalities, our dashboard page includes Questions, modals, and Activity. Questions refers to the table of previously completed questions and it's built with sub-components like QuestionRow (the display of the question's details in the table) and Categories (a group of tags related

to the question's topics). Activity refers to the question completed counter, based on difficulty levels, as well as the donut ring chart to visually represent it.

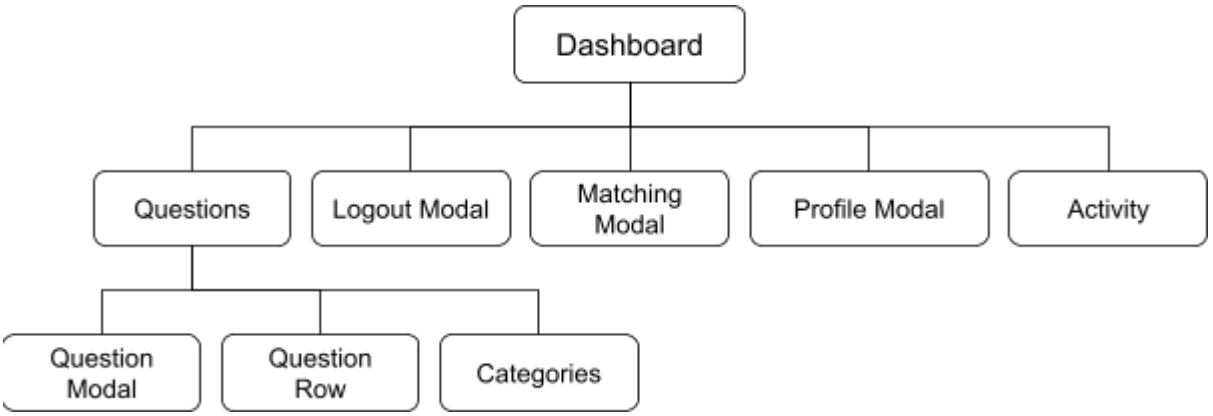


Figure 6.6.3 Dashboard Page Components

Finally, the login page is where existing users log into their accounts and new users create them. Previously, we had two separate pages for login and signup, but opted to combine them into one with just two different forms for accessibility.

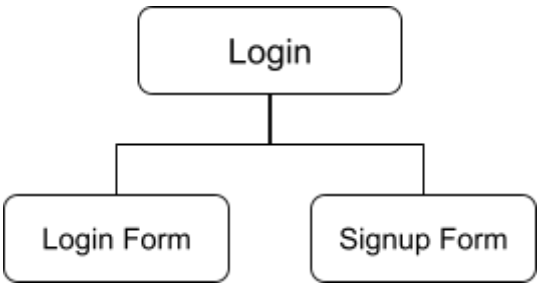


Figure 6.6.4 Login Page Components

6.5 User Flow

The figure below is an activity diagram which depicts the different actions users can take in PeerPrep, starting from their login or account creation, until they logout of their account.

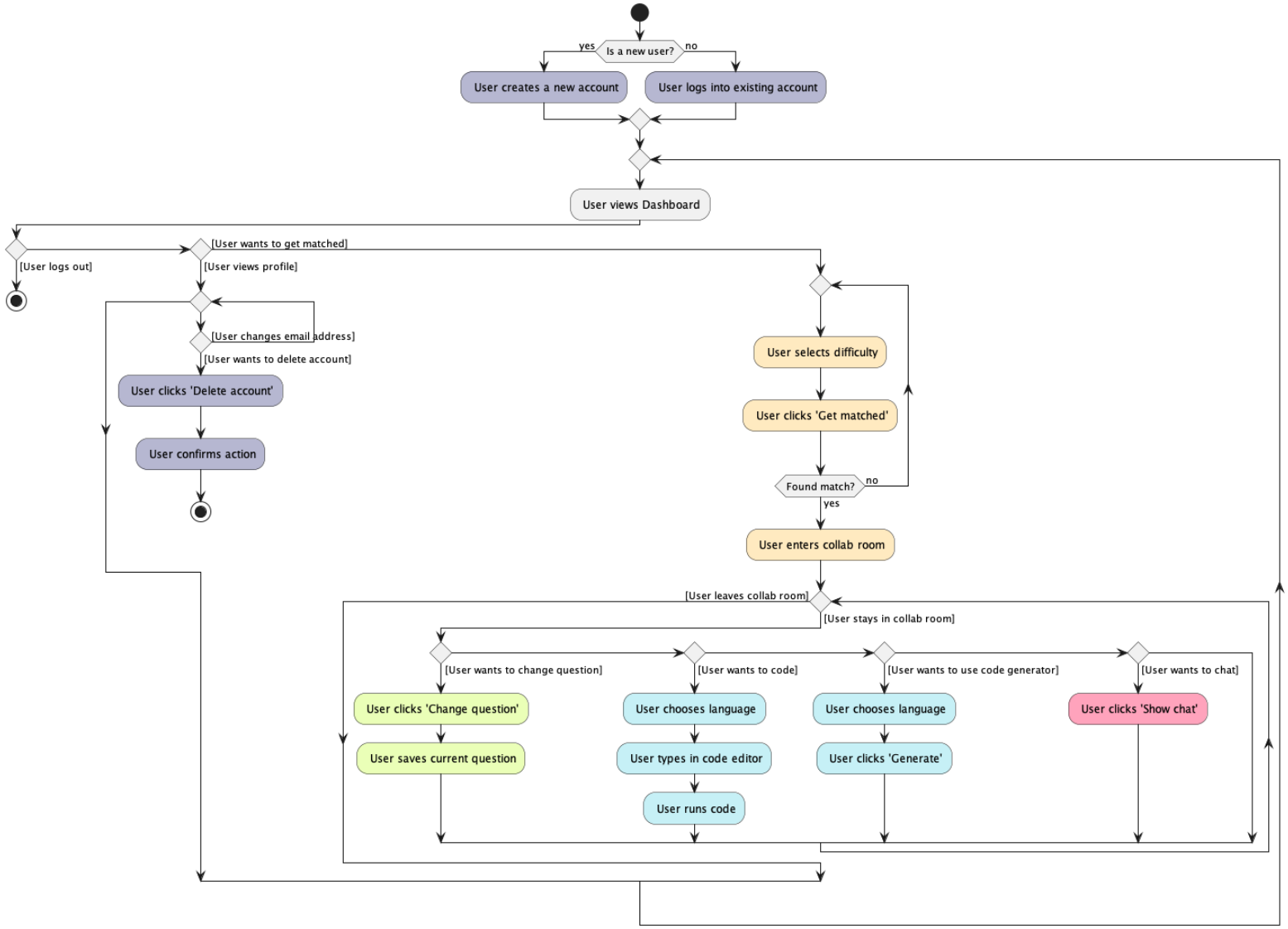


Figure 6.5.1 PeerPrep activity diagram

6.5.1 Signup and Login

If a user is new to PeerPrep, they can create an account by toggling to the Sign Up page.

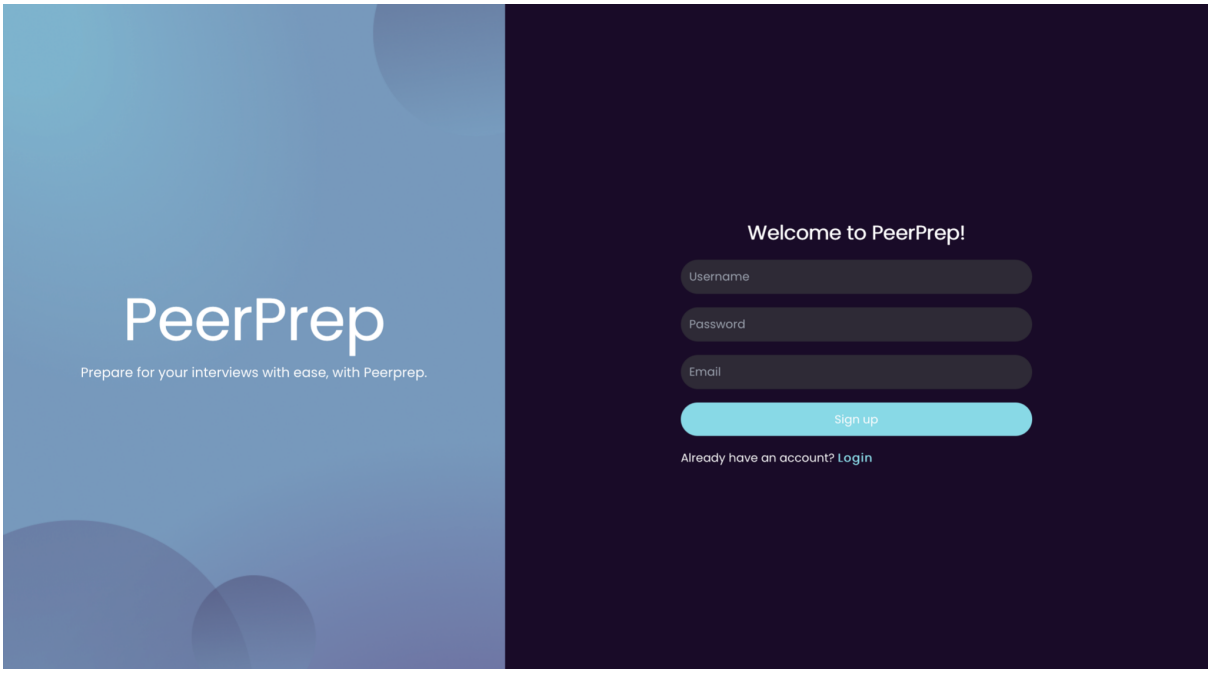


Figure 6.5.1.1 New user Sign Up page

Upon successfully creating a new account, the user will see a confirmation message.

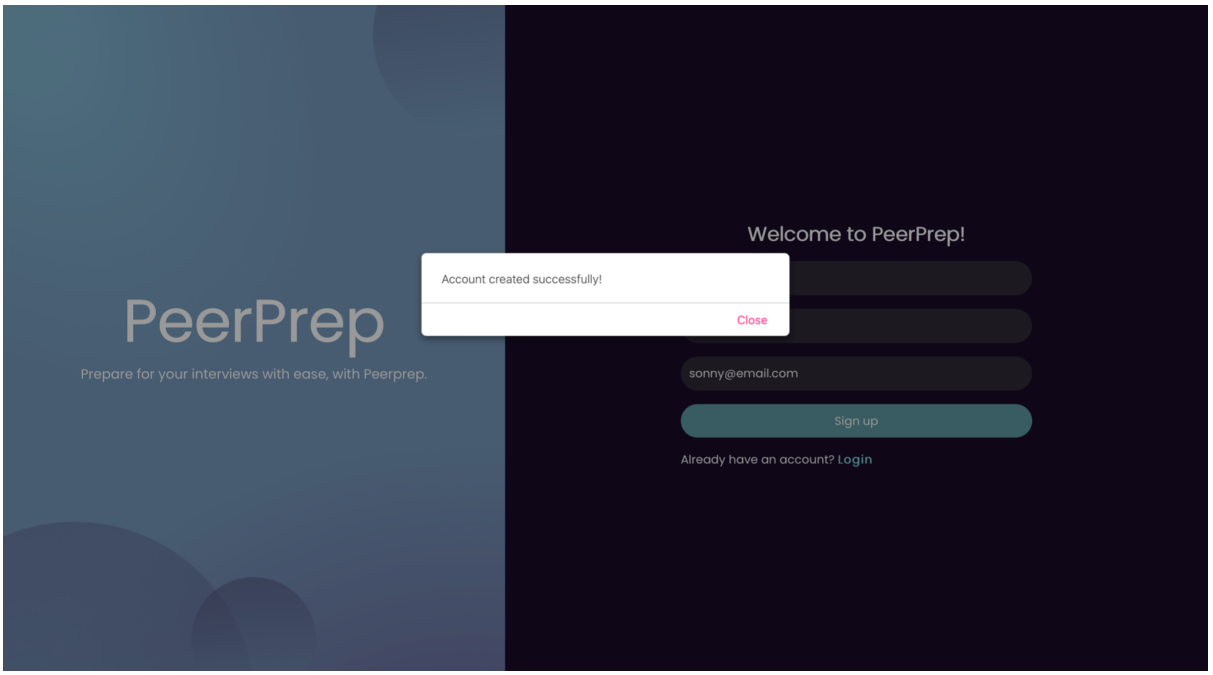


Figure 6.5.1.2 New account creation confirmation

Once the account is created, the user will return to the Login page. If a user already has an existing account with PeerPrep, they can simply login into their account using their username and password.

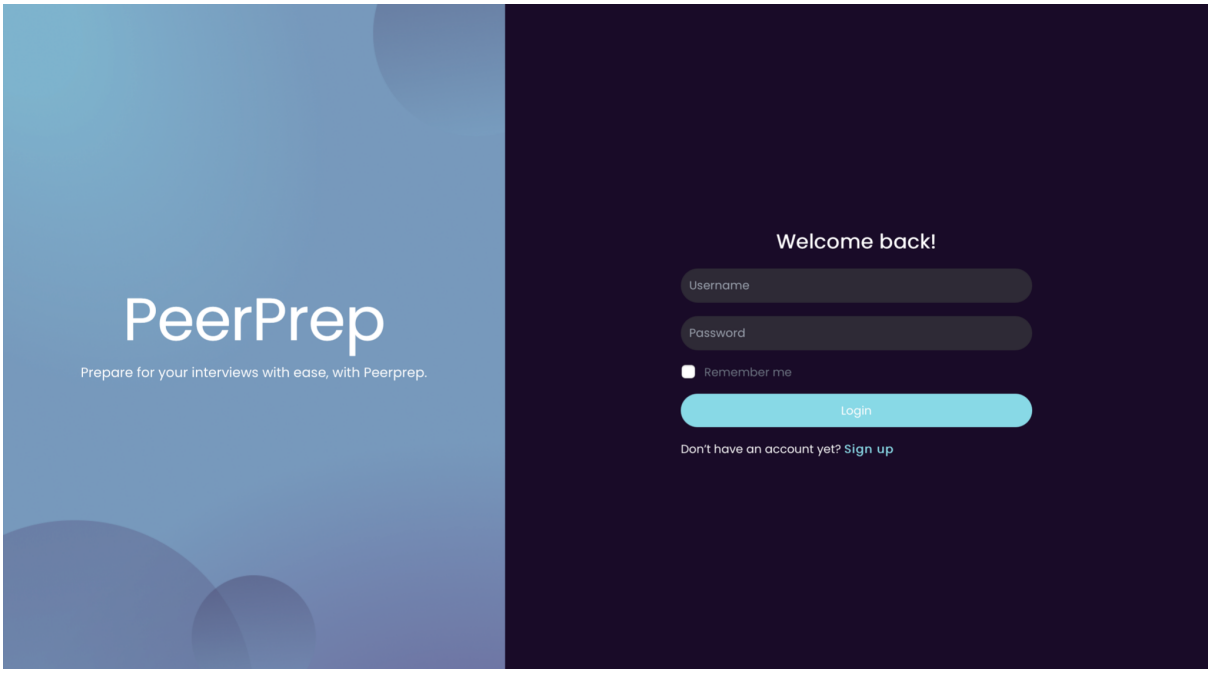


Figure 6.5.1.3 Existing user Login page

Our Login form handles errors such as missing username or password, wrong password, an invalid username which does not match with an existing user.

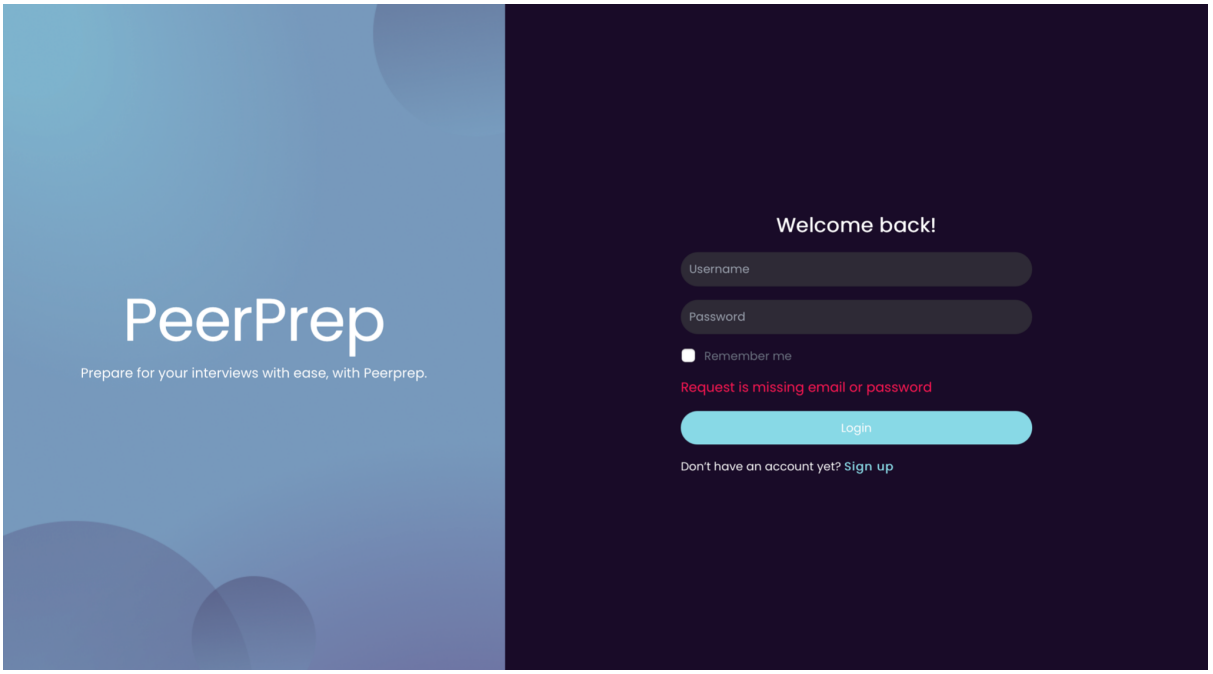


Figure 6.5.1.4 Missing username or password login attempt

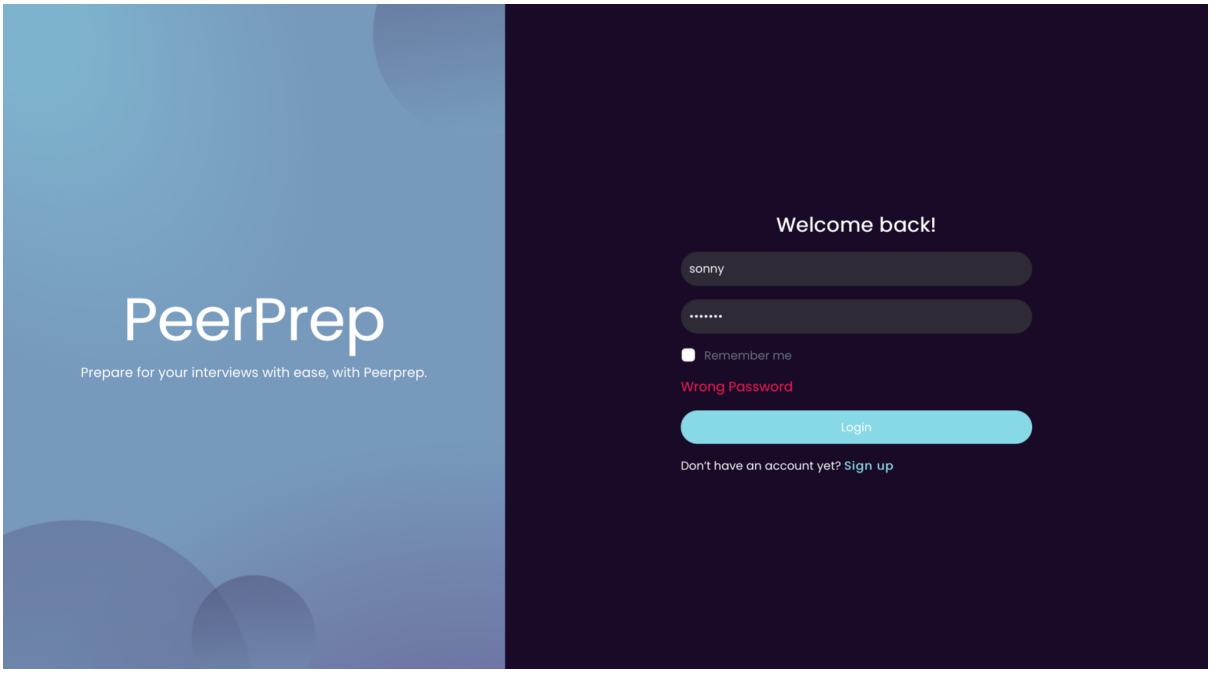


Figure 6.5.1.5 Incorrect password login attempt

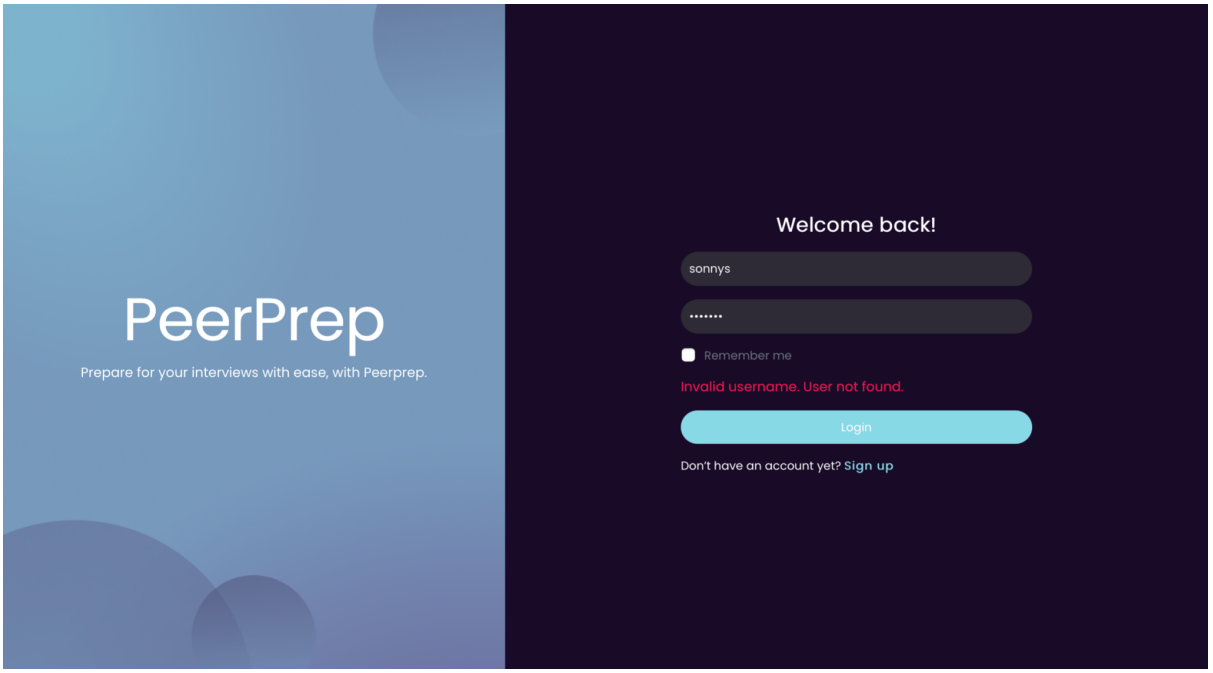


Figure 6.5.1.6 Invalid username login attempt

6.5.2 Dashboard

After the user successfully logs into PeerPrep, they will enter the Dashboard view. On their Dashboard, users can view and edit their profile, logout, view previously completed questions, and view their current progress.

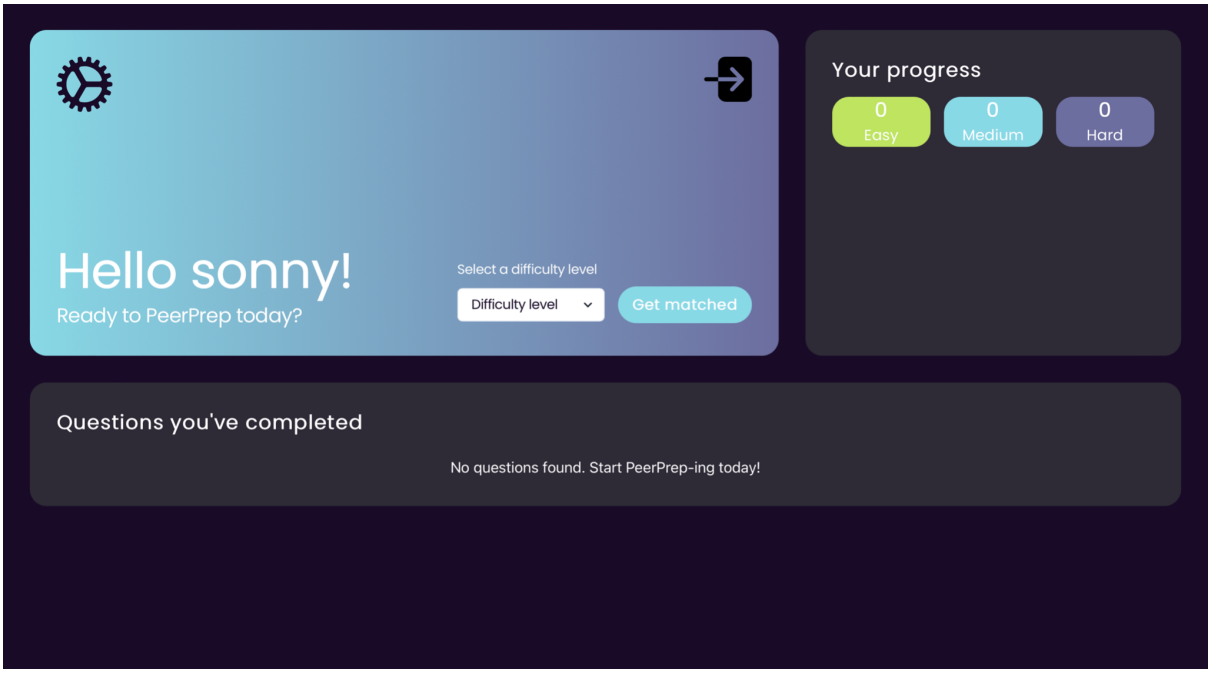


Figure 6.5.2.1 Dashboard view of new user

Profile

Clicking on the top-left gear icon opens the user's profile where they can view and edit their profile details.

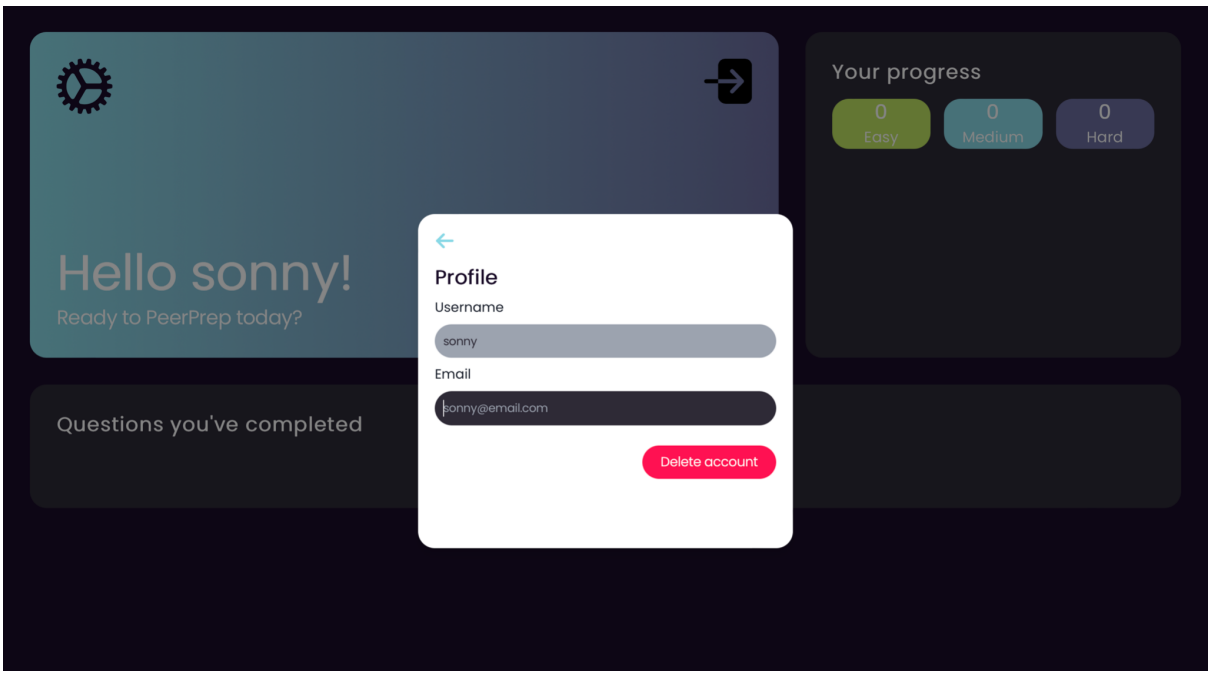


Figure 6.5.2.2 User profile view

Users can update their email address by typing into the field which will bring up the 'Update profile' button.

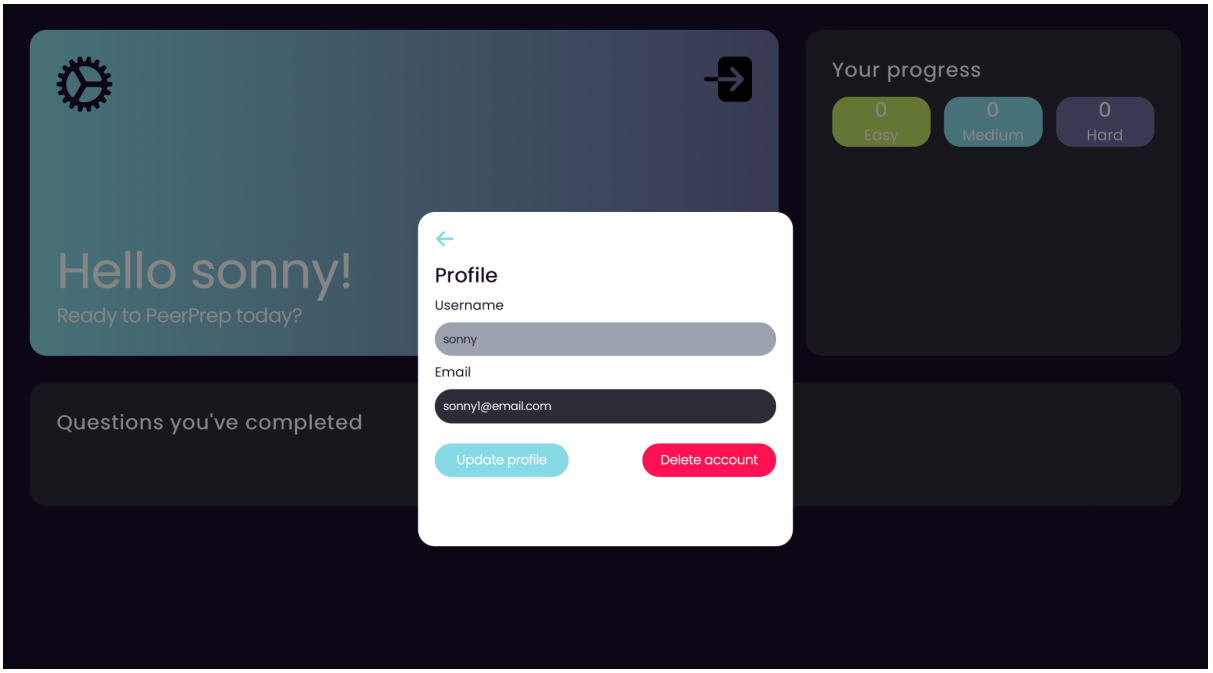


Figure 6.5.2.3 User updating email address

Users can choose to delete their PeerPrep accounts by clicking on the 'Delete account' button and will be prompted to confirm their decision.

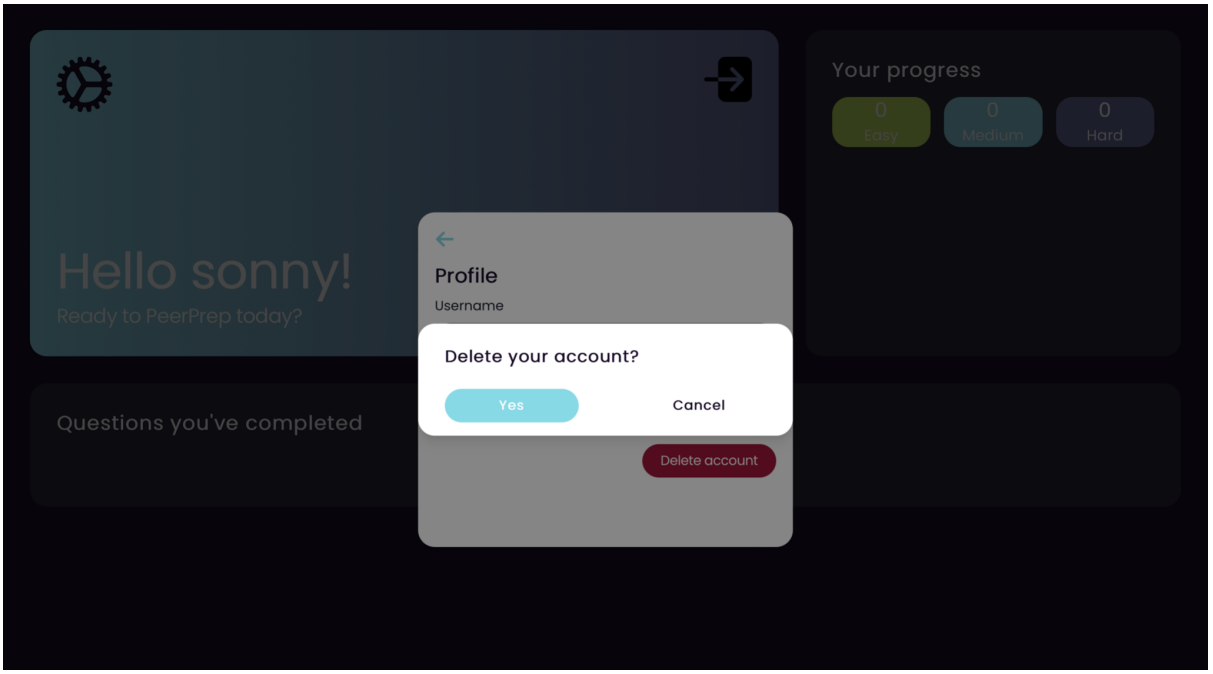


Figure 6.5.2.4 User confirming whether to delete account

Logout

Clicking on the top-right arrow icon allows the users to logout from their account at the end of their PeerPrep session. They will be asked to confirm their logout.

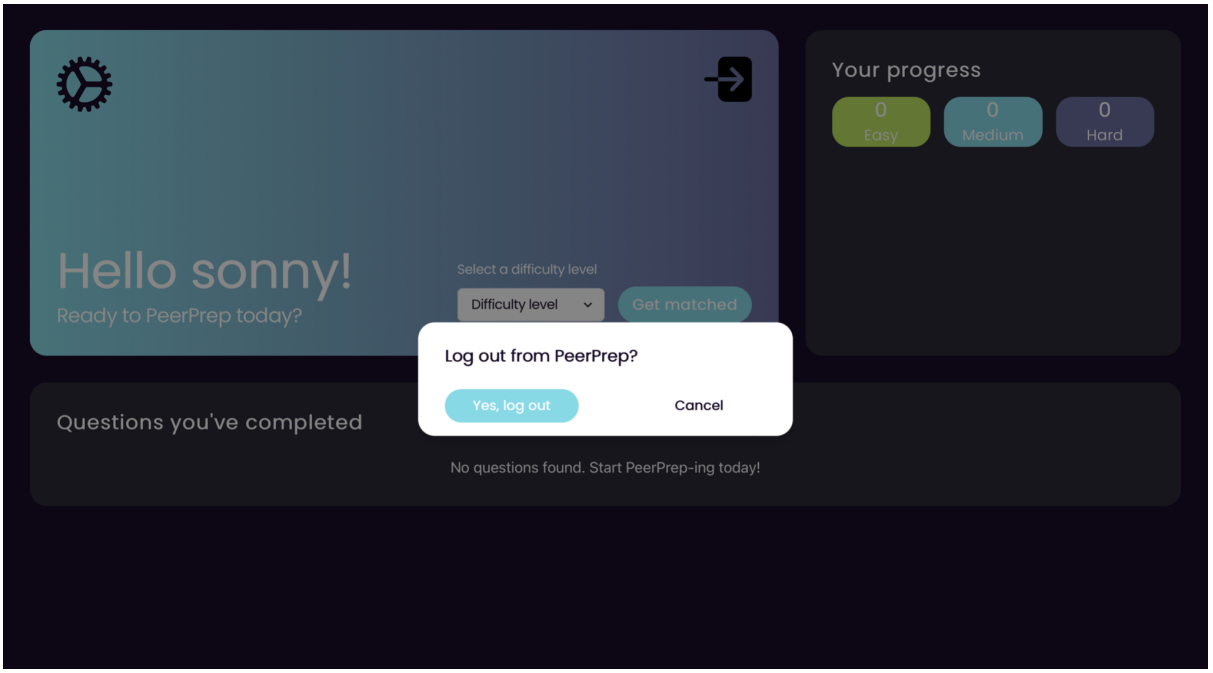


Figure 6.5.2.5 User confirming whether to logout from account

Progress and Completed Questions

Users can view their progress through how many Easy, Medium, or Hard questions they have completed through the counter and donut ring chart.

Under the 'Questions you've completed' section, users can view the list of previously completed questions. Each question is listed with the title, difficulty, categories, Peer completed with, and completion date.

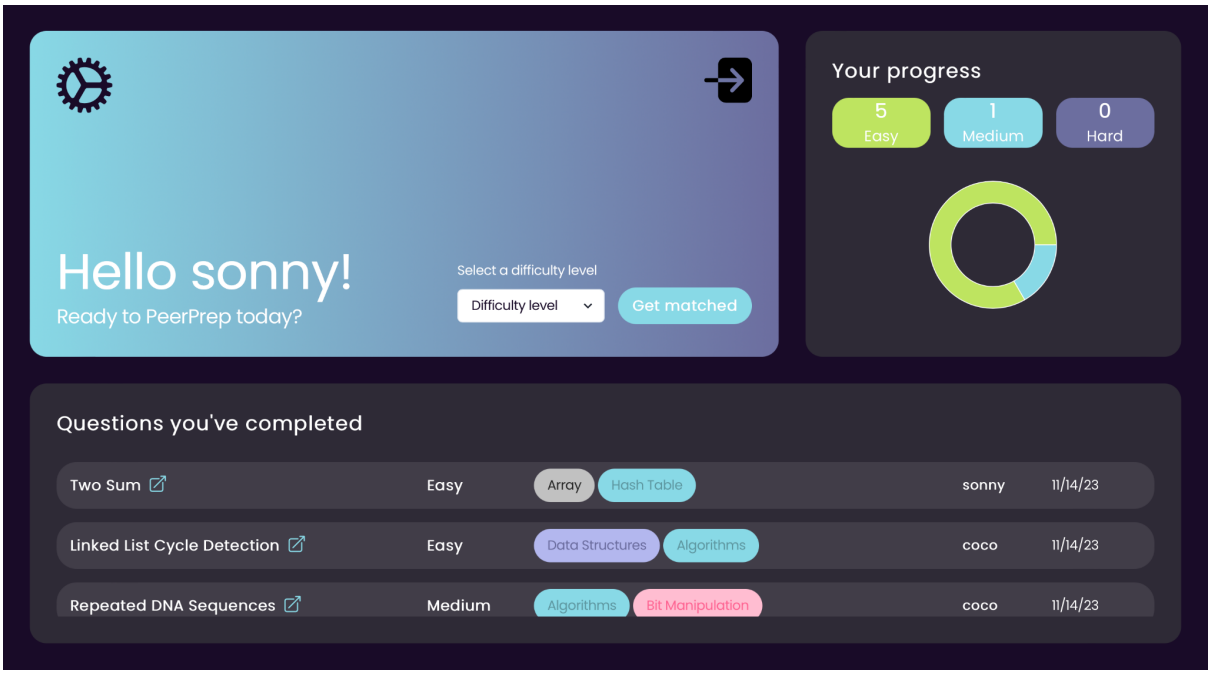


Figure 6.5.2.6 Dashboard view of existing user with existing activity

Clicking on the open icon on any question, users can view the question description, programming language they used, and the code written.

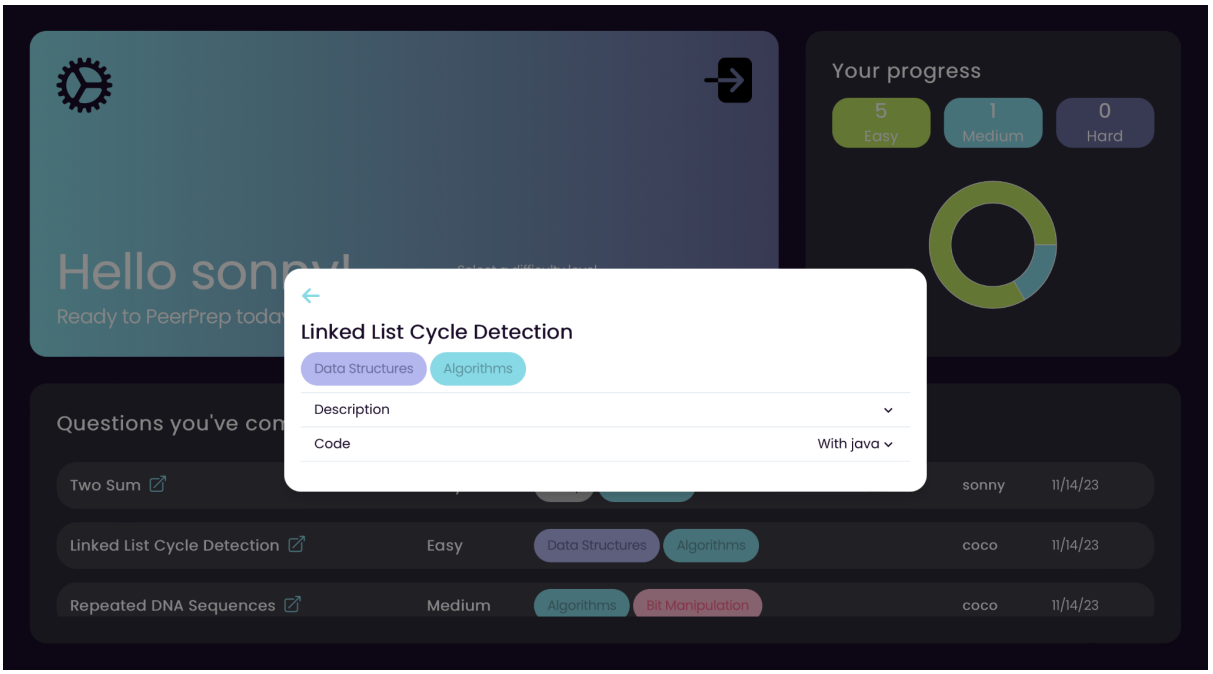


Figure 6.5.2.7 Completed question detail view

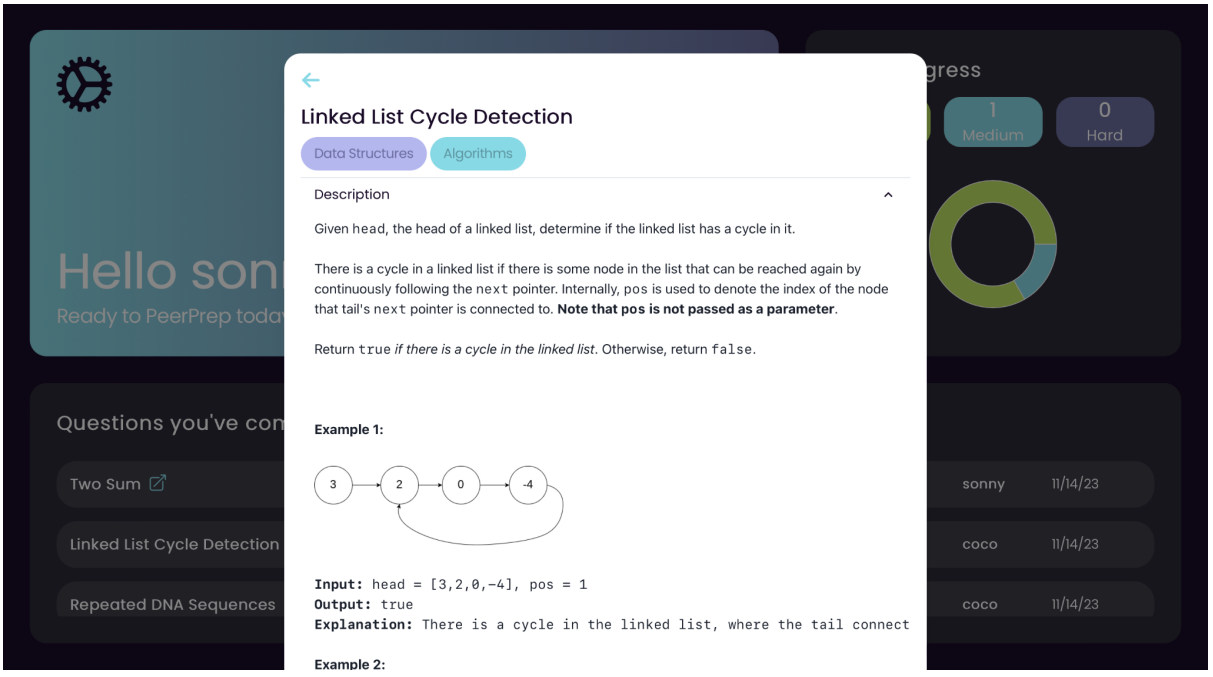


Figure 6.5.2.8 Completed question detail view with description

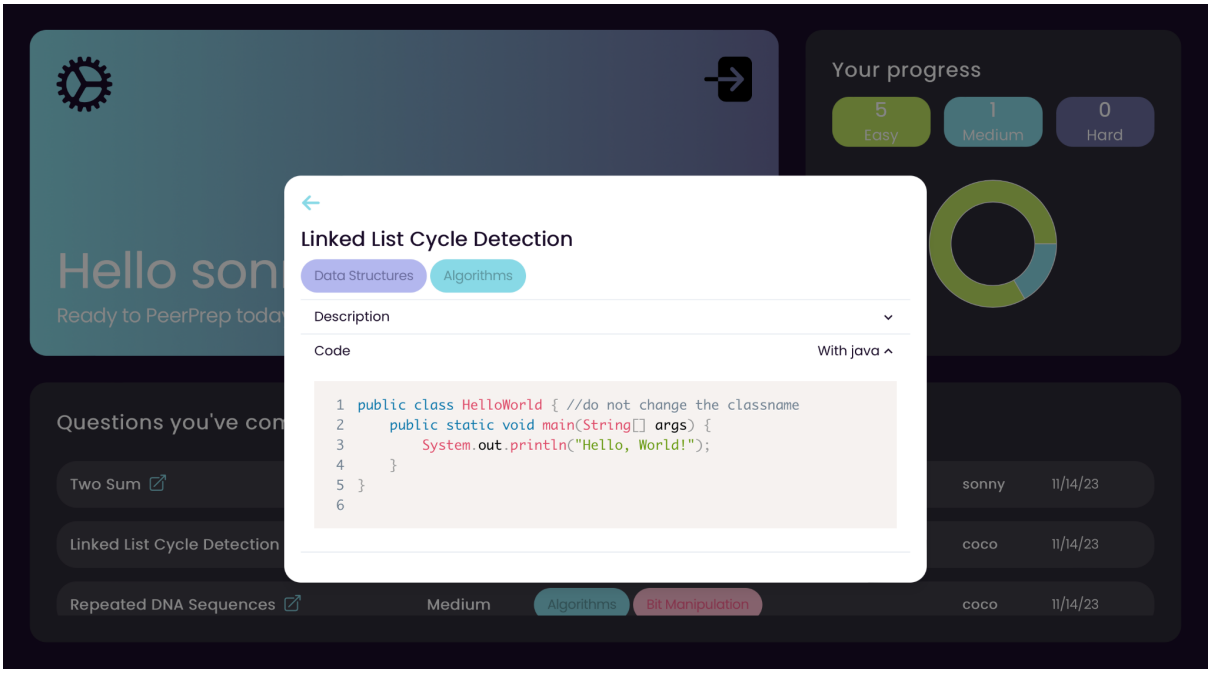


Figure 6.5.2.8 Completed question detail view with code written

Matching

In order to get matched, users first select their desired difficulty level from the dropdown and then, click the ‘Get matched’ button.

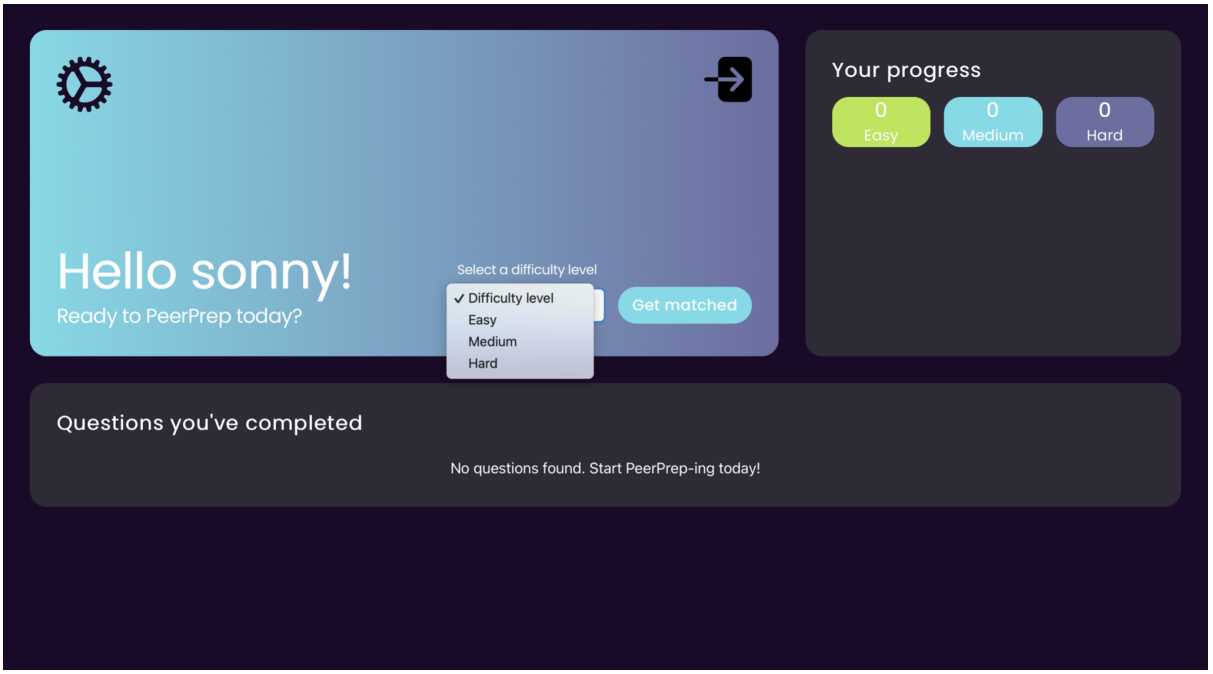


Figure 6.5.2.9 User selects a difficulty

As the user waits for a match to be found, they see the ‘Matching’ screen which displays a countdown timer of 30 seconds since they requested a match, as well as changing random facts. They also have the option to return to the dashboard and stop matching midway with the top-left button, ‘Stop matching’.

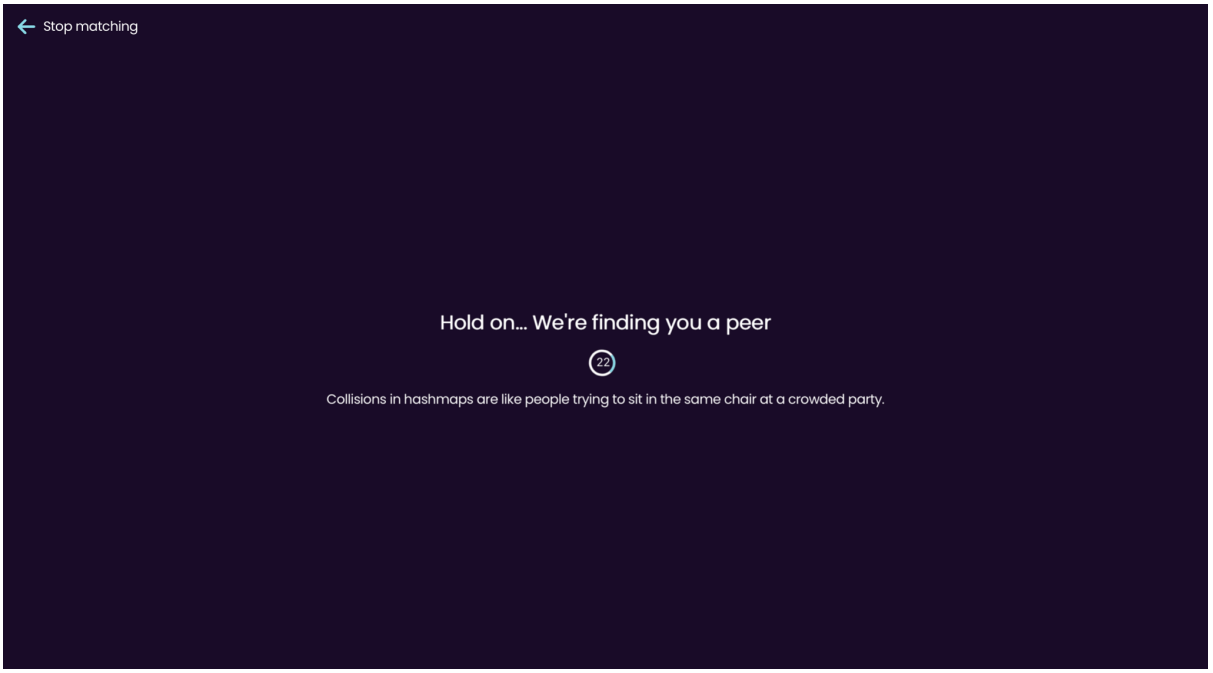


Figure 6.5.2.10 User waiting for match to be found

If a match is found within 30 seconds, a successful screen is shown and the user is brought to the collab room.

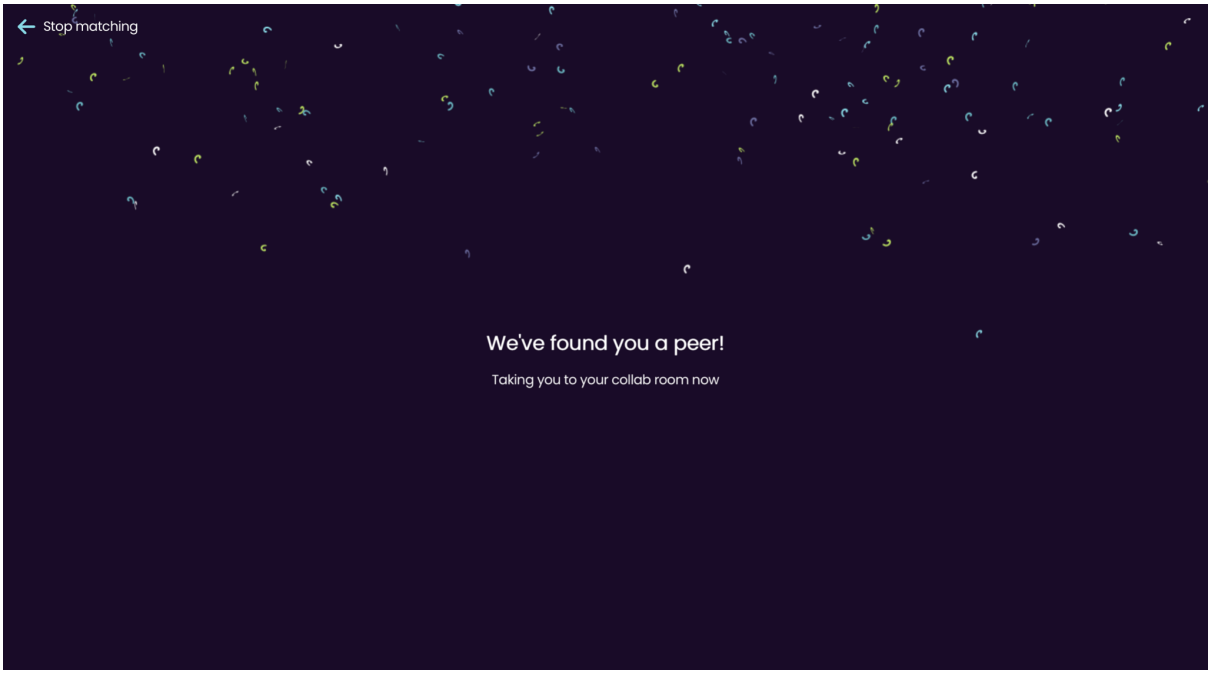


Figure 6.5.2.11 Successful match

If no match is found within 30 seconds, an unsuccessful screen is shown and the user is brought back to the dashboard where they can try to match again.

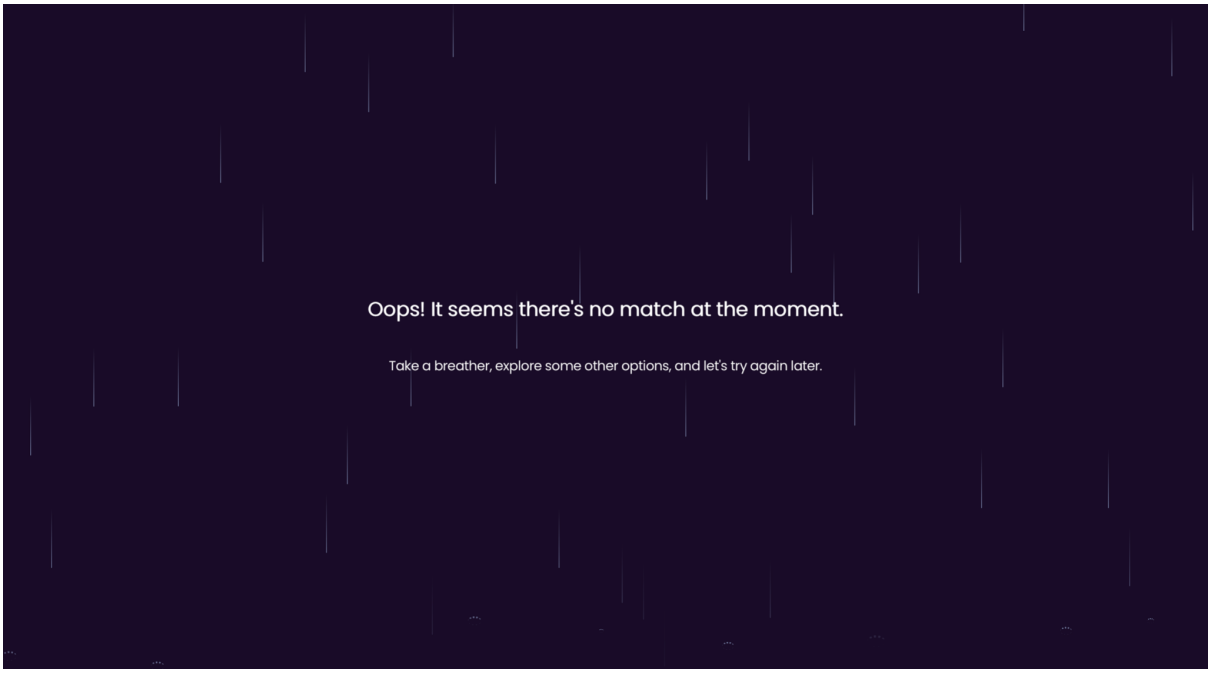


Figure 6.5.2.12 Unsuccessful match

6.5.3 Collab Room

Upon a successful match, the user is brought to the collab room with their matched Peer. In the Collab Room, they can change the programming language, collaboratively edit in the code editor, run the code, generate AI code solutions, chat, and change questions.

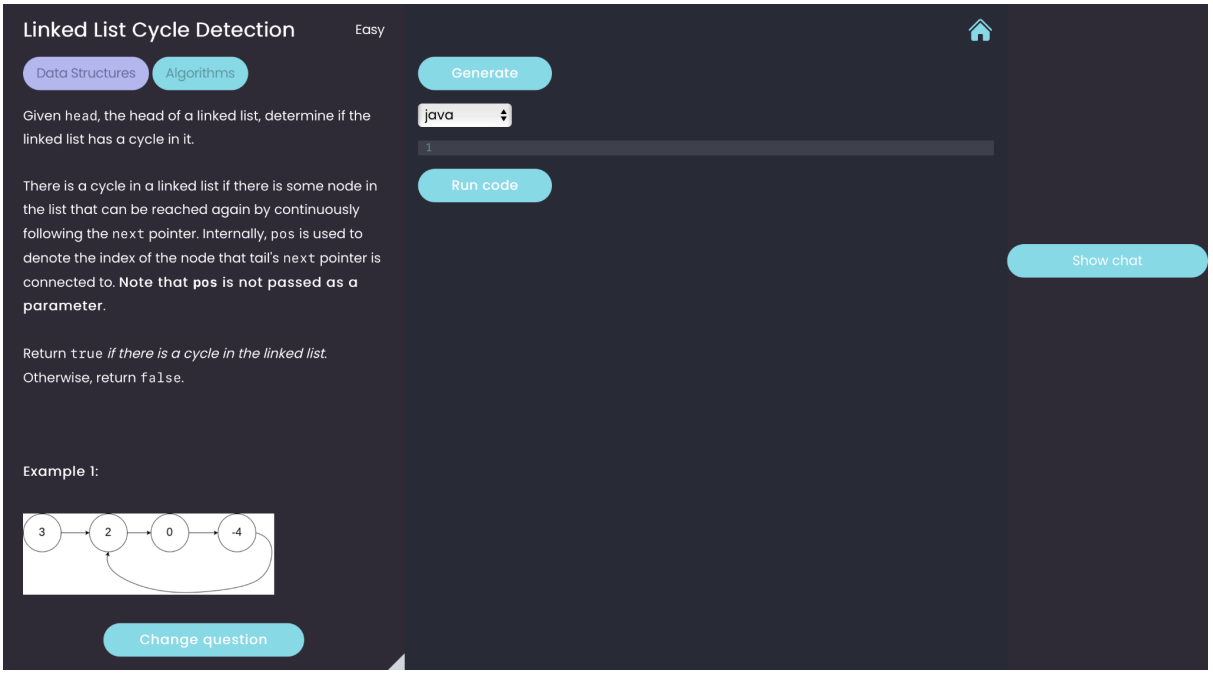


Figure 6.5.3.1 Collab room view

Users and their Peer can chat with each other by clicking on the ‘Show chat’ button and typing and sending messages.

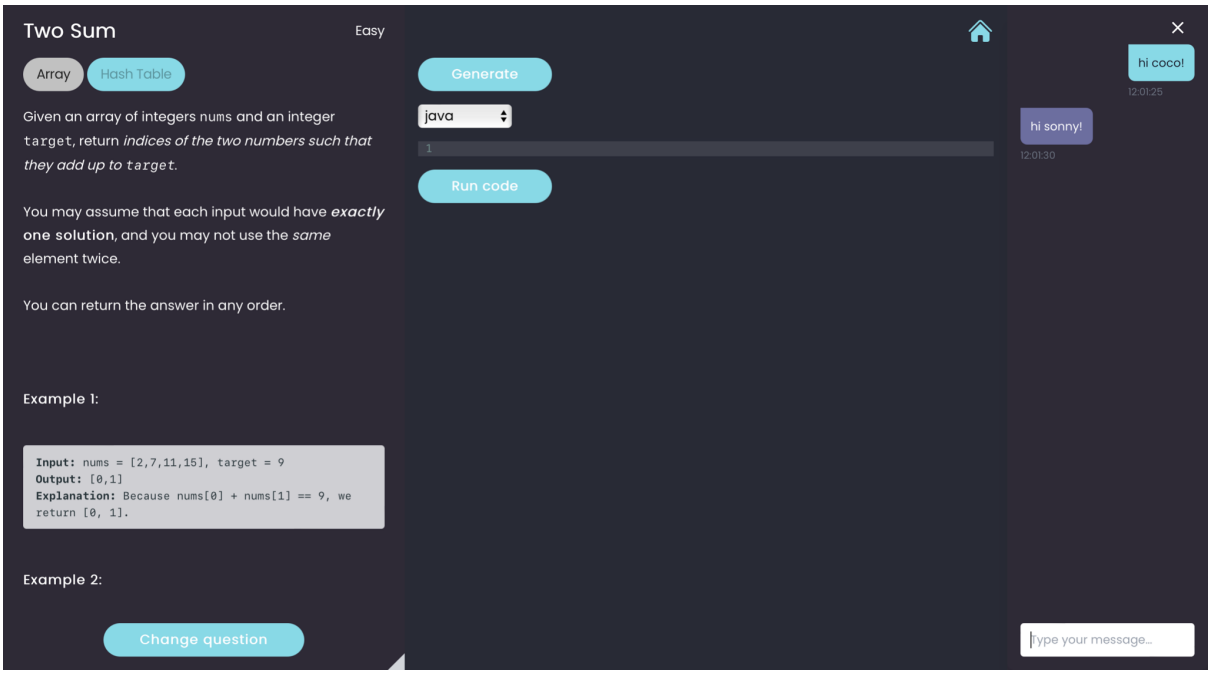


Figure 6.5.3.2 Chat between Peers

Users can select a programming language from the dropdown and edit collaboratively with their Peer, in the code editor. When they’re ready, they can click on the ‘Run code’ button in order to see the results of their code.

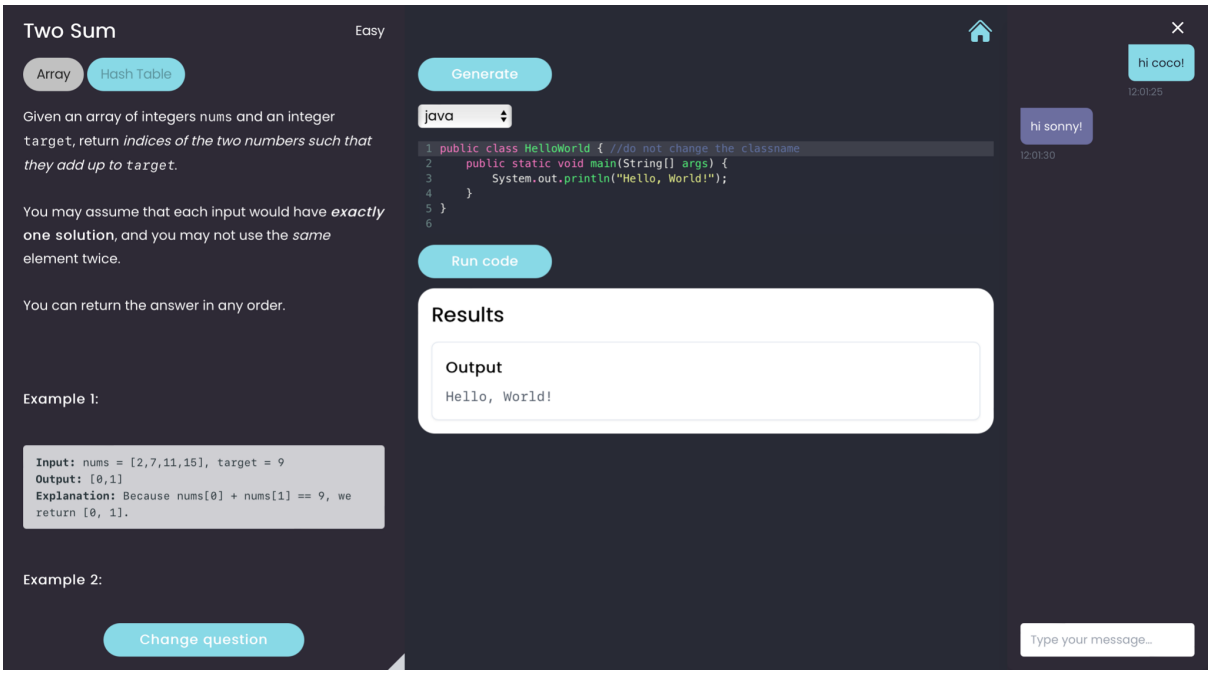


Figure 6.5.3.3 Code editor and results

If users are stuck on a question and want help, they can click on the ‘Generate’ button, select a programming language, and receive an AI generated code solution to their problem.

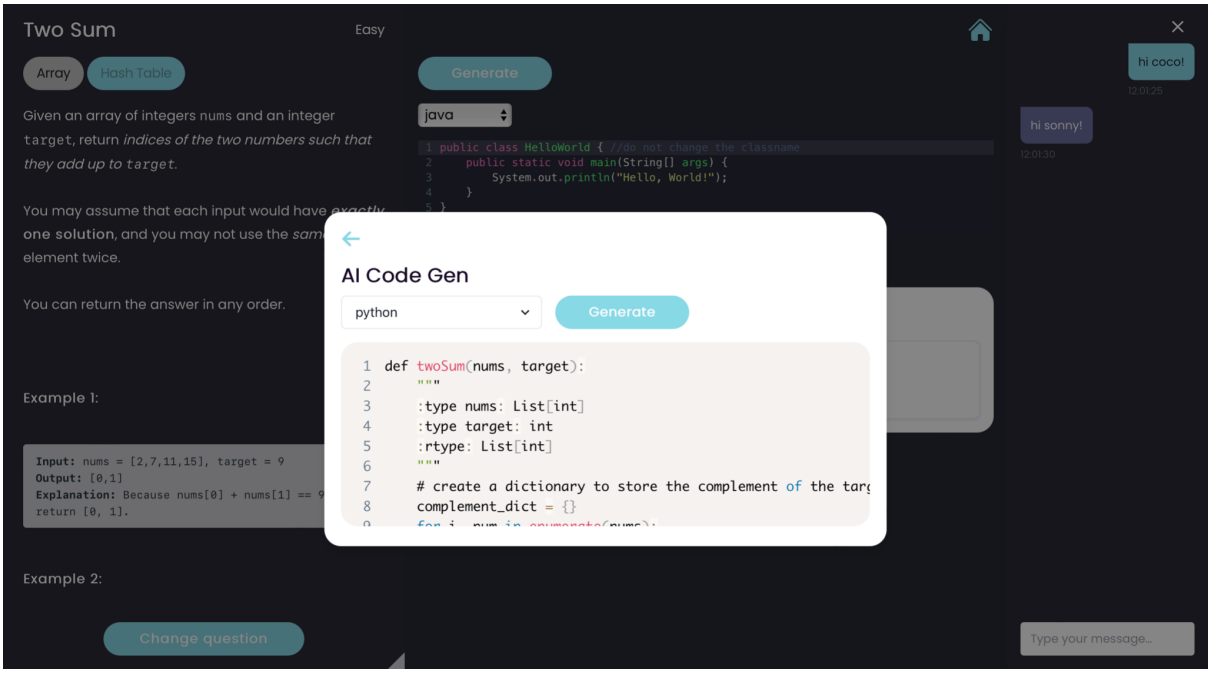


Figure 6.5.3.4 AI generated code solution

If a user wants to change the current question, they can click on ‘Change question’ which prompts them to first save their current progress, before switching to a new question. Both the user and their Peer will see the same, new question.

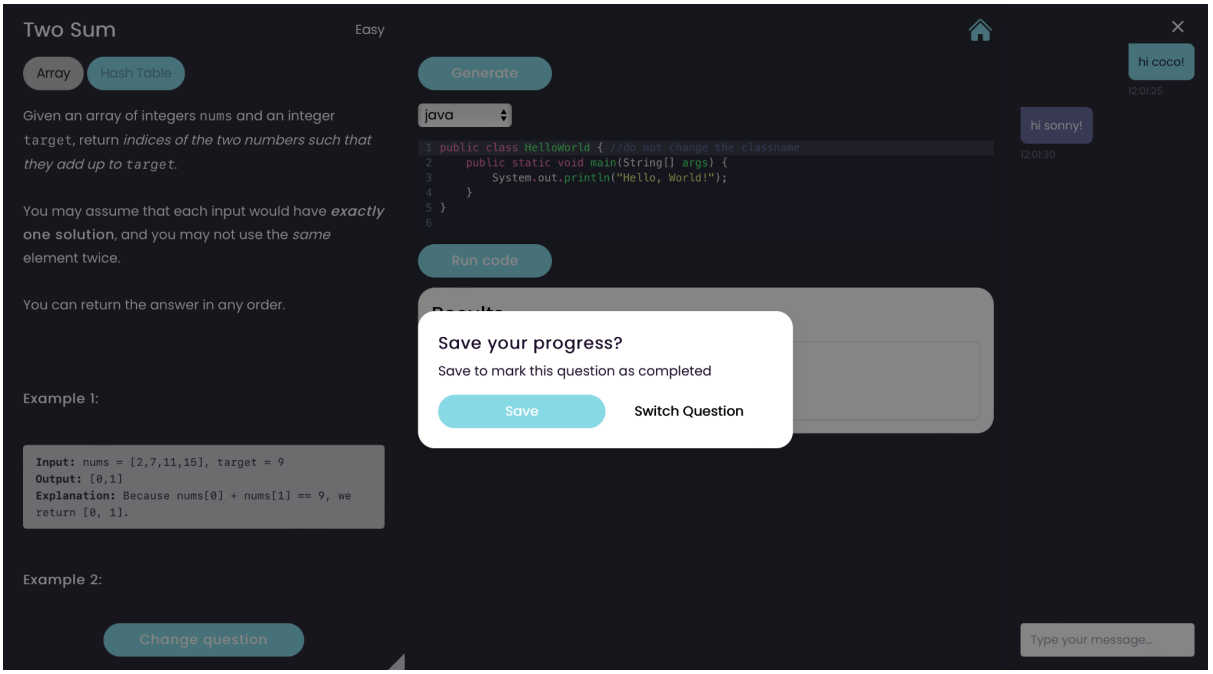


Figure 6.5.3.5 User changes question

When the user is done solving the problem, they can leave the room by clicking the top-right home button and save their work. They will be brought back to the Dashboard.

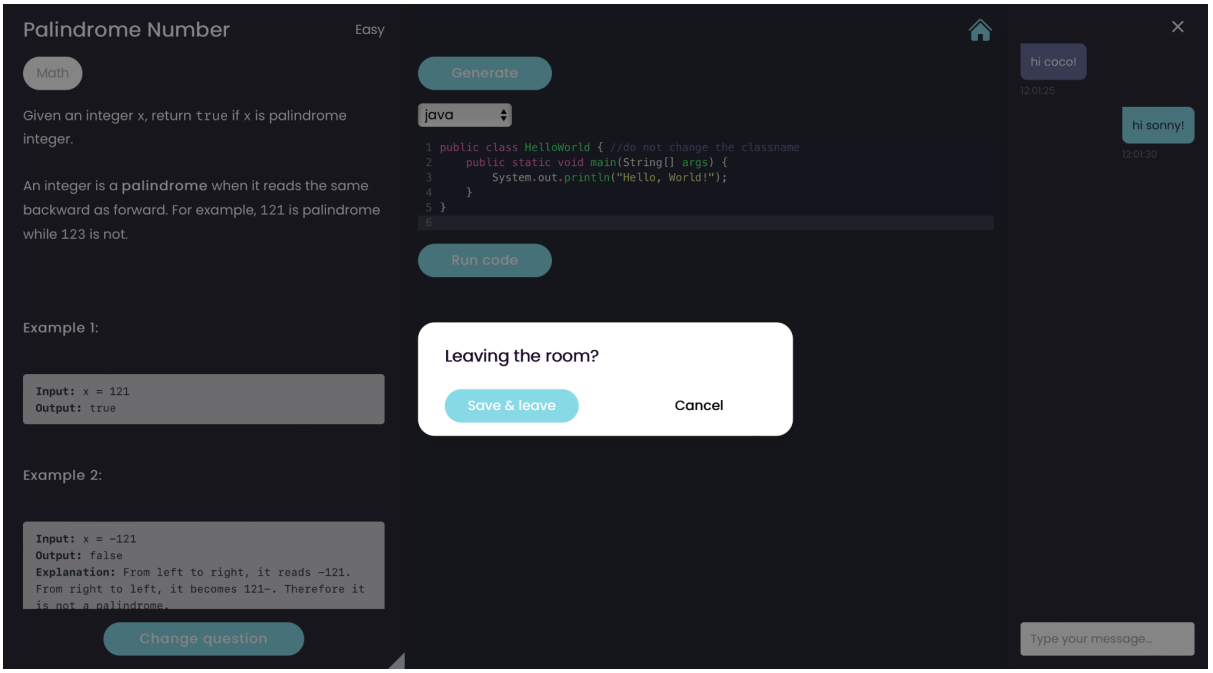


Figure 6.5.3.6 User leaves room

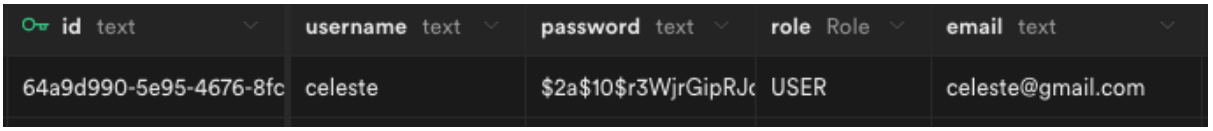
6.6 Services

6.6.1 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. Delete Account
3. Login
4. Verify token: client will call to verify validity of token upon page change

Schema - User Model



id text	username text	password text	role Role	email text
64a9d990-5e95-4676-8fc	celeste	$2a$10$r3WjrGipRJ	USER	celeste@gmail.com

Name	**Type**	**Description**
id	text	Unique string identifying the row
username	text	Unique string identifying the user
password	text	hashed and salted password string to authenticate user
role	String	‘USER’ or ‘ADMIN’
email	text	Email address of the user

Design: MVC, Separation of Concerns, DRY

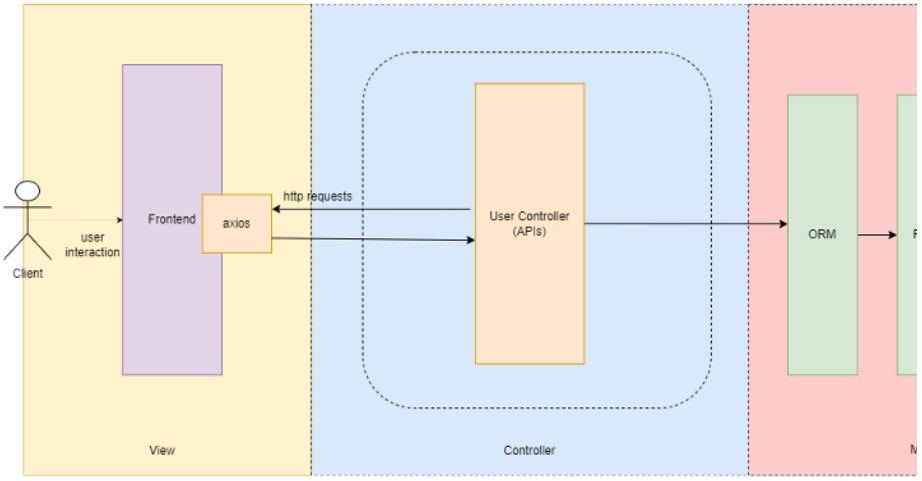


Figure 6.6.1.1 Architecture diagram of user-service, showing MVC pattern

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

As explained in Section 6.3.2, 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 PostgreSQL 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 and DRY principle, utility-related methods, such as JWT verification function, is also abstracted into a separate middleware file.

6.6.1.1 REST APIs

GET /retrieve	
Path	/retrieve
Usage	Get user's data based on id. To be displayed (e.g. in Profile modal)
Request Payload	id of the user
Returns	That specific user's details containing all fields except password

POST /	
Usage	Create new user. Used in Sign-ups

Request Body	UserCreateRequestType = {
	  username: string;
	  password: string;
	  email: string;
	};

Returns	UserResponseType = {
	  id: string;
	  username: string;
	  role: string;
	  email: string;
	};

POST /edit	
Usage	Edit user's fields. Used in Profile modal
Request Payload	Decoded JWT
Returns	Edited UserResponseType
DELETE /	
Usage	Delete User
Request Payload	Decoded JWT
POST /login	
Usage	Login
Request Body	Username, Password
Returns	JWT token
GET /verify	
Usage	Verify validity of JWT. Used by Frontend upon switching of pages (to prevent unauthenticated access to protected pages)

Request Body	JWT
Returns	204 Status upon verification of JWT

Implementation - Authentication

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

We utilize 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 localStorage.

Procedure - Authentication

User flow in Authentication:

1) When a user logs in, the server generates a JWT and sends it to the client.
2) The client stores the JWT, usually in local storage or a cookie.
3) For subsequent requests, the client includes the JWT in the request headers.
4) The server validates the JWT and, if valid, processes the request. Else, returns 401 Unauthorized error.

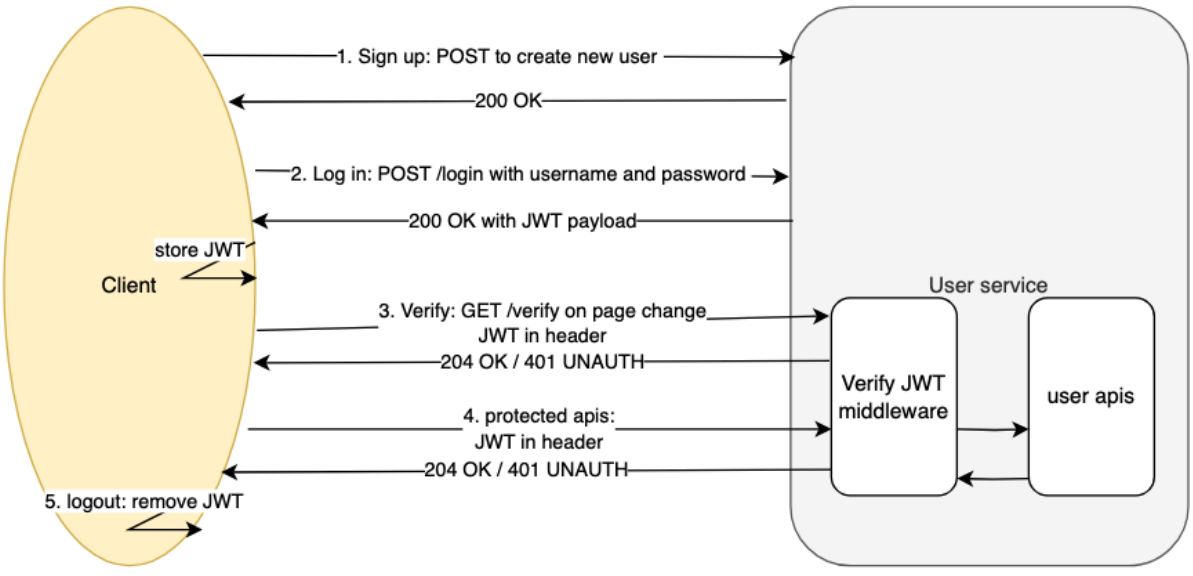


Fig 6.6.1.2 Authentication procedure for Signup, Login, Verify, Logout

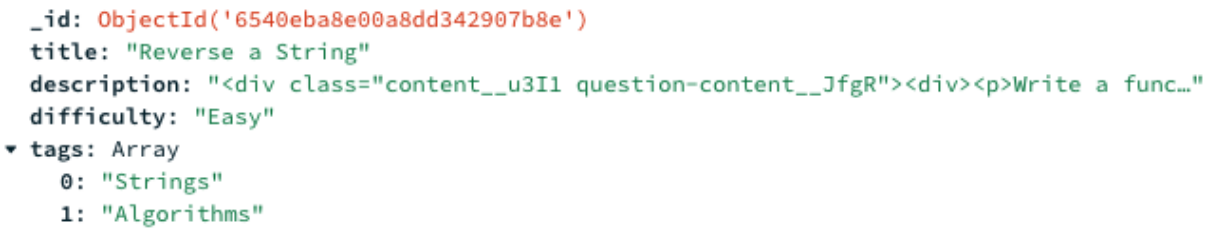
Justification - JWT in localStorage

Pros	
Statelessness	JWTs are stateless, meaning the server doesn't need to keep track of the user's session. All necessary information is contained within the token, making it scalable and easy to deploy in distributed systems.
Reduced Database Queries	Since JWTs contain user information, there's no need to query the database for each request to authenticate the user. This can lead to improved performance, especially in applications with a large number of users.
Ease of Implementation	Well-established libraries such as jsonwebtoken and passportJs available in NodeJS.
Flexibility	JWTs can carry custom claims and additional information about the user, allowing for flexibility in implementing various features like authorisation.
Cons	
Security Risks	Storing JWTs in local storage exposes them to potential security risks, such as Cross-Site Scripting (XSS) attacks. If an attacker can inject malicious scripts into the application, they may access and steal the JWT.

6.6.2 Question Service

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

Schema - Question Model



Field	Type	Description
id	ID	Unique auto-generated id to identify the question
title	string	Question title
description	string	- In the form of HTML string to preserve formatting - Images are stored as urls within img tags - Includes question description, example inputs and outputs, and constraints.
tags	array	String representing the difficulty of a question (i.e. “Easy”, “Medium”, “Hard”).

Design: MVC

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.

6.6.2.1 REST APIs

question-service provides API for retrieving questions:

GET /id/:id	
Usage	Get the question's details based on id. To be displayed (e.g., in Questions Display table and modal).
Request Param	question id

Returns	The details of that specific question with the following structure:
	{
	_id: string;
	title: string;
	description: string;
	difficulty: string;
	tags: string[];
	}

GET /random	
Usage	Retrieve a random question based on difficulty. Used to provide a new random question for users.
Request Param	difficulty (string): The difficulty level of the question. questionId (optional, string): The ID of the current question to ensure a different question is returned.

Returns	The details of a random question with the following structure:
	{
	_id: string;
	title: string;
	description: string;
	difficulty: string;
	tags: string[];
	}

Implementation - Question Synchronization

To ensure both users in the same room get access to the same question, Frontend will 1) get random question from Question Service, 2) client socket will share the question to the other user with socket.emit() where the server socket will broadcast the question to all client sockets in the same room.

The same procedure follows for the 'Change Question' feature.

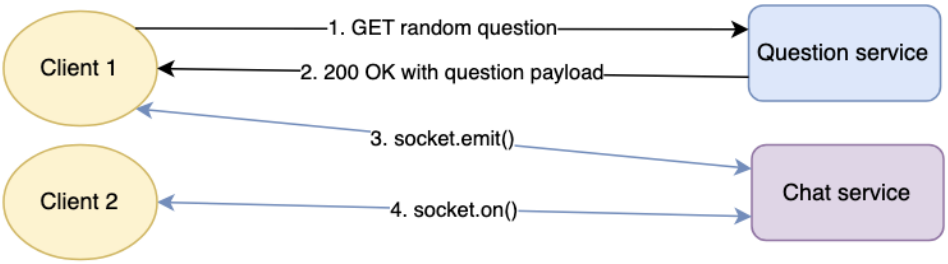


Figure 6.6.2.1 Diagram showing flow when displaying a question on match success

Implementation - Question Formatting and Display

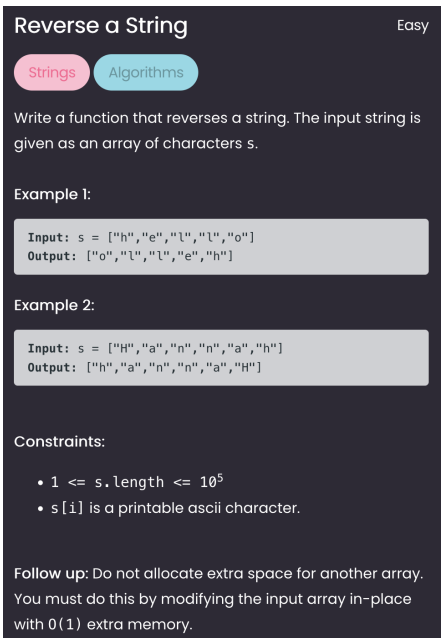


Figure 6.6.2.2 Screenshot Showing Question Display

The retrieved question is then rendered in a QuestionDisplay component on the frontend.

```
<div class="content__u3I1 question-content__JfgR">
  <div>
    <p>
      Write a function that reverses a string. The input string is given as an
      array of characters <code>s</code>.
    </p>
    <p> </p>
    <p><strong>Example 1:</strong></p>
    <pre><strong>Input:</strong> s = ["h","e","l","l","o"]
        <strong>Output:</strong> ["o","l","l","e","h"]</pre>
    <p><strong>Example 2:</strong></p>
    <pre><strong>Input:</strong> s = ["H","a","n","n","a","h"]
        <strong>Output:</strong> ["h","a","n","n","a","H"]</pre>
    <p> </p>
    <p><strong>Constraints:</strong></p>
    <ul>
      <li>
        <code>1 &lt;= s.length &lt;= 10<sup>5</sup></code>
      </li>
      <li>
        <code>s[i]</code> is a
        <a
          href="https://en.wikipedia.org/wiki/ASCII#Printable_characters"
          target="_blank"
          >printable ASCII character</a
        >.
      </li>
    </ul>
    <p> </p>
    <p>
      <strong>Follow up:</strong> Do not allocate extra space for another array.
      You must do this by modifying the input array
      <a href="https://en.wikipedia.org/wiki/In-place_algorithm" target="_blank"
        >in-place</a
      >
      with <code>O(1)</code> extra memory.
    </p>
  </div>
</div>

```

Question body is stored as an HTML string in the database to preserve its format (e.g. bolding, lists). It is being rendered on the frontend using dangerouslySetInnerHTML, a React attribute that directly sets HTML content on a React element.

There is potential for security vulnerabilities, such as cross-site scripting (XSS) attacks. To prevent this, we ensure that the HTML content we inject is safe and doesn't contain malicious scripts, and do not allow users to add questions.

Implementation - Populating Question Bank

We achieve the population of the question bank by
running a script which gets HTML representation of questions from a third-party source [https://bishalsarang.github.io/Leetcode-Questions/out.html] and cleans it by removing unnecessary characters (e.g. '/n', '[Object]'), before inserting into MongoDB.

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.

6.6.3 Matching Service

The Matching Service is responsible for matching two users together based on question difficulty level.

6.6.3.1 How It Works

Storing Users in Queue

The matching service utilizes a temporary data structure to store socket Ids of users who have yet to be matched. This data structure follows FIFO and is used to help us match users based on their difficulty(Fig). We can interact with this data structure via the matching-service, which we communicate with via SocketIO.

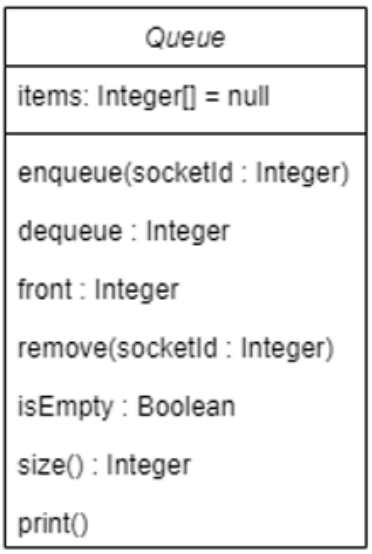


Figure 6.6.3.1.1 Queue implementation

Our implementation utilizes the idea of temporary storage where the role is to store pending match users of each difficulty level. There will be a queue for each difficulty level(Easy, Medium, Hard) and at any point in time, there will only be a maximum of 1 user waiting in each queue. Once there is a match for a specific difficulty level, the user of that respective difficulty level would be removed from the queue.

Frontend communication with matching service

We make use of SocketIO to communicate with the matching service via the "*queue*" event. From the frontend, after clicking on the start matching button, the frontend will emit the difficulty level to the Matching-Service via SocketIO. The Matching-Service will check the corresponding queue(Based on difficulty level) if there is already a user waiting to be matched. If there is no user already waiting, then the Matching-Service will enqueue this socketId into the queue. In the case that there is already a user waiting, the Matching-Service will dequeue the waiting

user, generate a UUID(RoomId), and emit a json object(roomId, myId, otherId, difficulty) via the "*match*" event to notify both users of a match.

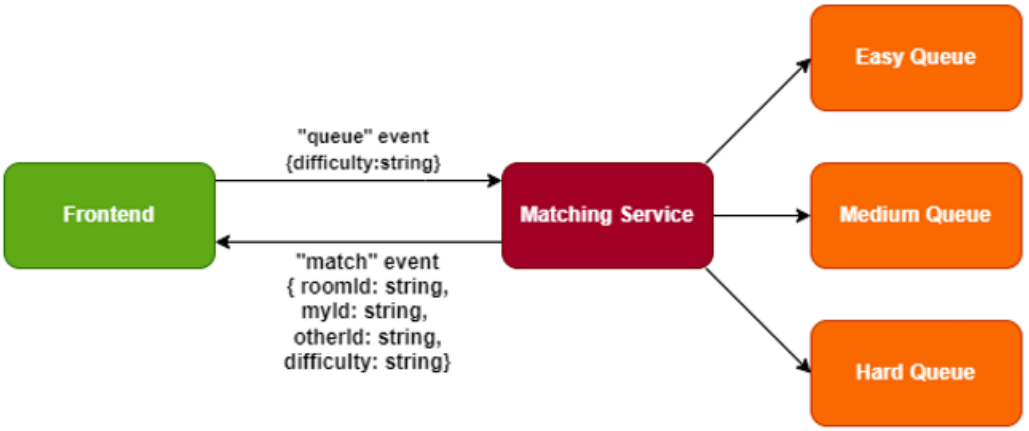


Figure 6.6.3.1.2 Frontend interaction with Matching Service

In the case that the user disconnects or clicks leave queue in the frontend side, it will send a "*disconnect*" or "*leave*" via the socket, which will automatically dequeue the socket from the corresponding queue.

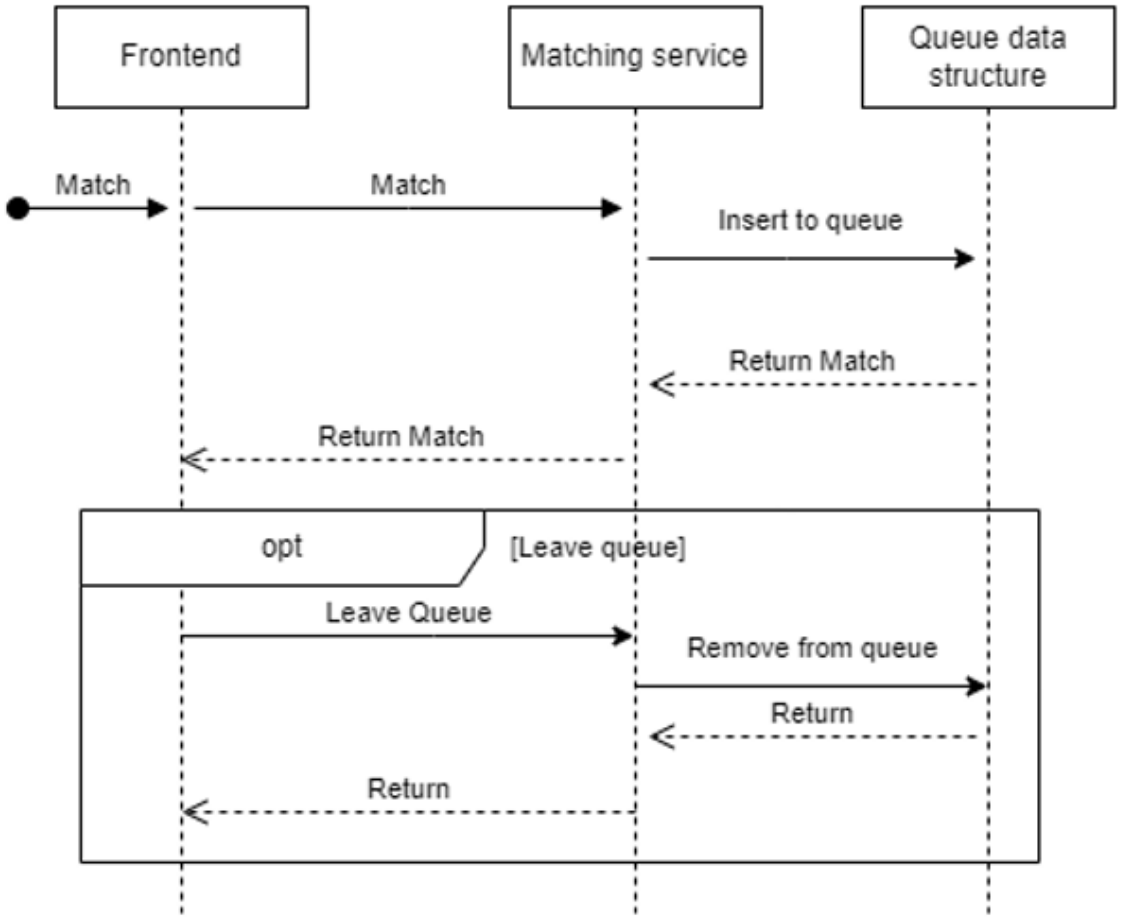


Figure 6.6.3.1.3 Sequence diagram for matching)

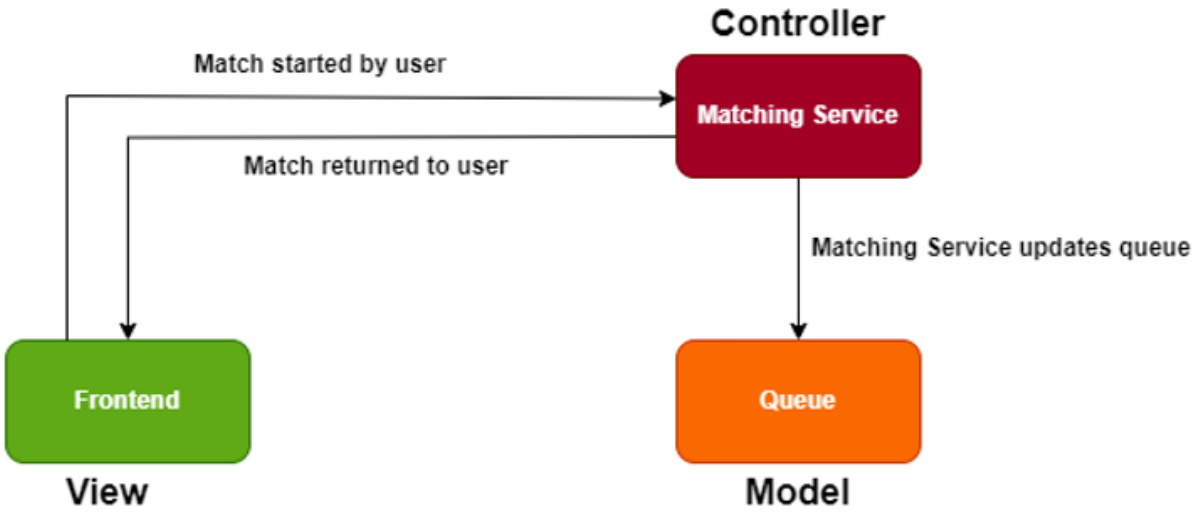
6.6.3.2 Design - MVC

Our implementation utilizes MVC design patterns.

Model (Queue Data Structure): In our implementation, the temporary data structure storing socket IDs serves as our model. Separating the data-related operations into a distinct model class allows for clear organization and maintenance.

View (Frontend User Interface): The frontend user interface is responsible for collecting user input and passing information to the matching service. The view is separate from the controller logic, allowing for the user interface to be updated independently of the underlying data and logic

Controller (Matching-Service): The Matching-Service functions act as the controller. It receives user matches from the frontend, interacts with the model (temporary data structure) by enqueuing/dequeuing, and updates the view by emitting events via SocketIO.



(Figure 6.6.3.3.2.1 MVC overview)

6.6.3.3 Pros and Cons of implementation

Pros	
Real-time communication	SocketIO enables real-time bidirectional communication between the frontend and the Matching-Service, allowing for efficient and instantaneous updates on match status.
Scalability	The use of a queue data structure for each difficulty level facilitates scalability. The FIFO approach ensures a fair matching system, and the design allows for easy scaling to accommodate more difficulty levels without significant modifications.
Clear separation of concerns	Our implementation utilized the MVC design pattern, promoting a clear separation of concerns. This makes the codebase modular and easier to maintain, as each component (Model, View, Controller) has a distinct role.
Unique room identifiers	The use of a UUID (RoomId) for room identification enhances security and uniqueness in the matching process. This helps to avoid potential conflicts and ensures a reliable identifier for matched users.
Cons	
Potential queue congestions	In scenarios with high demand for a particular difficulty level, the queue might become congested, leading to longer

	wait times for users. This could impact the overall user experience.

6.6.4 Chat Service

The Chat Service is responsible for allowing real-time seamless communication between users who are matched. Users, having successfully joined the room, can now send and receive messages using the same socket room. Chat service uses socket.IO for real-time bi-directional communication.

6.6.4.1 How It Works

Upon successful match by the matching-service, users will be taken to a Collab page with a UUID as roomId(generated by the matching service).
Upon rendering the page, we will fetch the roomId from the query parameters of the URL.

A new socket will be created, which will emit a *join_room* event with the roomId, which will allow the socket to join the room in the chat-service backend(Fig). In short, both users will have a socket upon joining the collab room, which will eventually join the same socketIO room, which they will use for communication with each other.

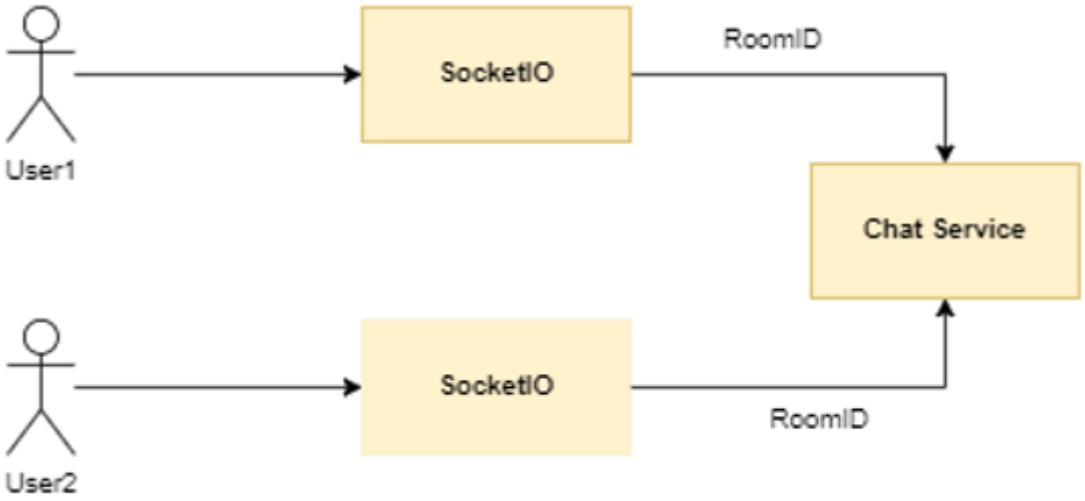


Figure 6.6.4.1.1 socket connection

Upon mounting of the chat component, the socket will also listen to the event("*receive_message*"). This is to listen for any messages sent by the other user in the chat room.

Users are able to send messages to the chat room, which will be broadcasted to every other user in the chat room(1 other). Upon clicking the send button, a json object consisting of {roomId, author(userId), message and time} is sent to the backend via the "*send_message*" event. The chat-service will receive the json object sent by the client socket, retrieve the roomId, and forward the json object to the chat room by emitting the "*receive_message"* event. Therefore, the json object will be broadcasted to other users in the same chat room, who are already listening on the event("receive_message"). The activity diagram of this interaction is seen below.

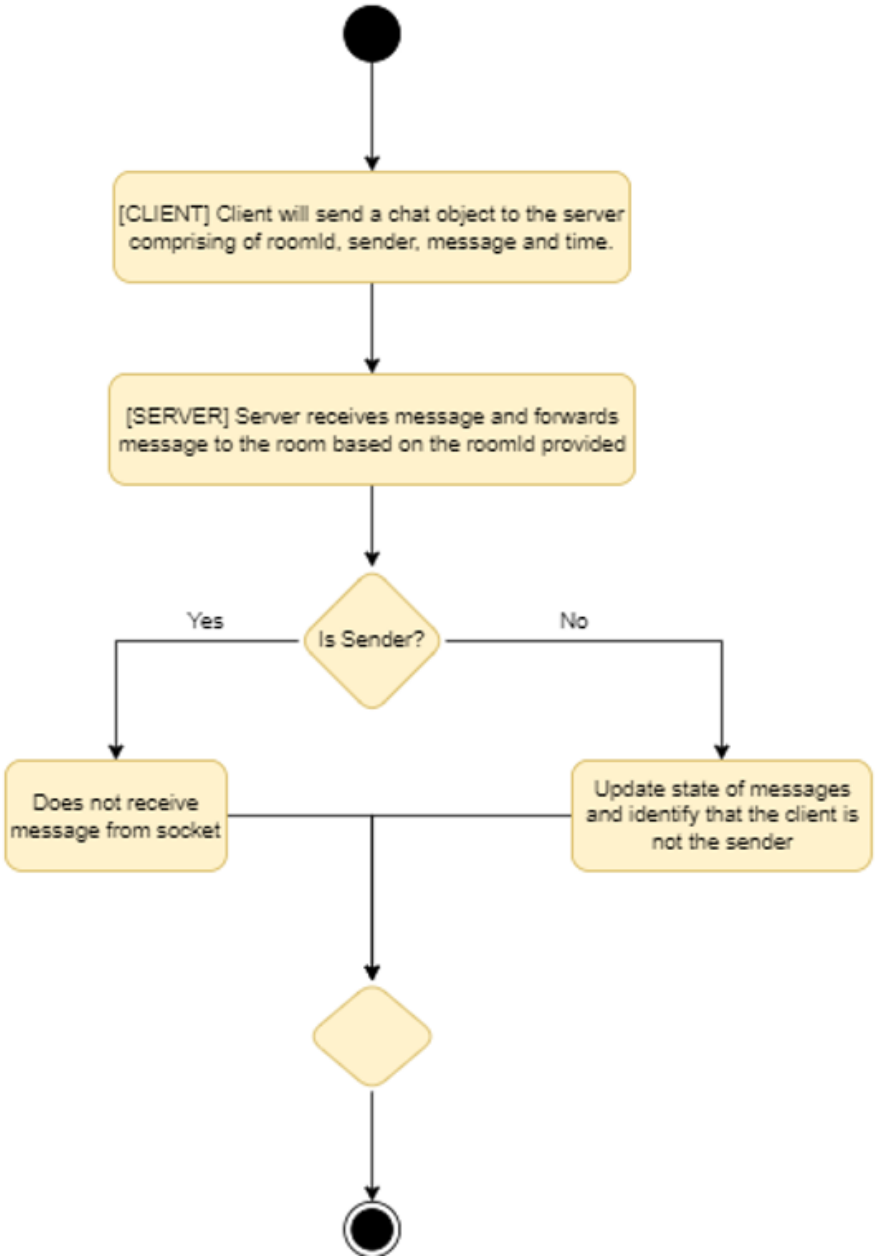


Figure 6.6.4.1.2 Activity diagram for sending message

6.6.4.2 Design

We made use of the Pub-Sub messaging pattern to implement our chat service. In our implementation of chat-service, each client socket acts as both a publisher and subscriber to the socket room. Each client socket can publish messages to the socker room via the "*send_message*" event, and subscribe to messages published in the socket room via the "receive_message" event.

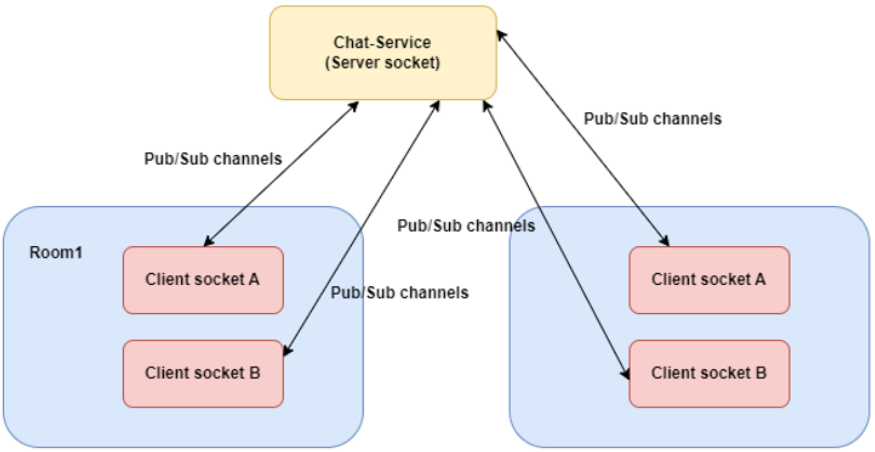


Figure 6.6.4.2.1 Pub-Sub design

Change Question and Language

Since two users are subscribed to the socket room, we can leverage the socket room to ensure synchronization of the question and language settings between both users. This is done through the "*language*" and "*question*" events, which both client socket will listen to. Upon a change in language/question, the "*language*"/ "*question*" event will be emitted to the Chat-Service, which will then be forwarded to the appropriate room(based on RoomId), thus notifying the other user of the change, rendering the change in the frontend.

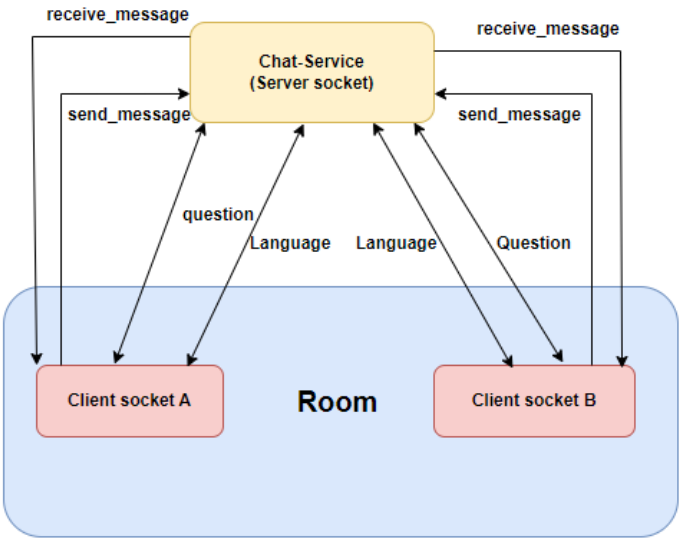


Figure 6.6.4.2.2 overall view

6.6.4.3 Pros and Cons of implementation

Pros	
Real-time communication	SocketIO facilitates real-time bidirectional communication between two matched users in a room, providing a seamless and instantaneous exchange of messages between users.
Synchronization of settings	Leveraging the socket room for synchronization of language and question settings ensures that both users in a room are kept in sync, enhancing the collaborative experience.
Cons	
Scalability challenges	If the application experiences a substantial increase in users, managing the synchronization and communication between numerous chat rooms may become challenging.

6.6.5 Runtime Service

The Runtime Service is responsible for allowing code execution on the code editor. The languages that Runtime Service supports are Java, Python, C and Javascript.

6.6.5.1 How It Works

The Frontend service will communicate with the Runtime Service via a GET request on the “/code” endpoint.

GET /code	
Usage	Gets the output of the code execution given the code and selected language.

Request Body	{
	selectedLanguage: string;
	code: string;
	}

Returns	The output of the code execution:
	{
	stdout: string;
	stderr: string;
	error: string;
	}

Our Runtime Service is provided with the code as well as the selected language by the frontend. Based on the selected language, it will write the code to a designated file(Test.js for javascript). Subsequently, the service mounts this file onto the respective Docker image (refer to Fig) and initiates the execution using the 'docker run' command. The docker run command will return error, stdout and stderr, which we will return to the frontend.

Language	Docker Image
Javascript	node:18
C	gcc:latest
Python	python:3
Java	openjdk:latest

Figure 6.6.5.1.1 *Relevant Docker Images*

We deploy the Runtime Service on a Google Cloud Engine virtual machine. We install docker on our VM and pull the relevant images into our VM. Therefore, when sending the GET request to our VM, the VM does not need to waste time pulling the docker images from docker hub and can immediately call the docker run command to return the output. An overall view of the runtime service is provided.

Figure 6.6.5.1.2 Overall View

6.6.5.2 Design

The design follows a client-server architecture by using REST api. This means that the Frontend service sends a request to the Runtime Service, and the Runtime Service responds with the output of the code execution. This synchronous communication ensures a clear flow of control between the Frontend and Runtime Service.

We chose client-server architecture so as to achieve separation of concerns. The client should not need to be concerned with how the code is executed, the server does not need to be concerned with how the code is keep tracked of. Therefore, we make use of REST, where each request is stateless full information(code, selected language) is provided to the Runtime Service by the frontend.

6.6.5.3 Pros and Cons of implementation

Pros	
Separation Of Concerns	Utilizing REST API to communicate between the frontend and Runtime Service effectively decouples the Frontend service from the complexities of code execution and Docker management.
Isolation with Docker	Utilizing Docker for code execution provides isolation, ensuring that each execution runs in its own environment without affecting the overall system.
Cons	

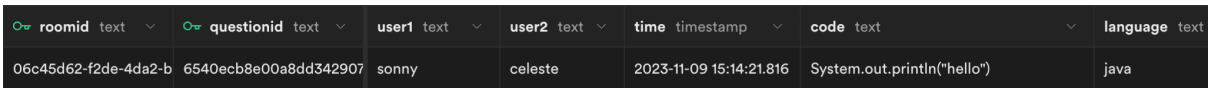
Resource Utilization	Docker containers consume additional resources, and running multiple containers simultaneously can affect the VM overall performance.

6.6.6 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

Schema - History Model



roomid text	questionid text	user1 text	user2 text	time timestamp	code text	language text
06c45d62-f2de-4da2-b	6540ecb8e00a8dd342907	sonny	celeste	2023-11-09 15:14:21.816	System.out.println("hello")	java

Field	Type	Description
roomid	ID	Unique auto-generated id to identify the room
questionid	ID	Unique auto-generated id to identify the question
user1	string	Name or identifier of the first user
user2	string	Name or identifier of the second user
time	string	Timestamp of the attempt, logged when user ends the session
code	string	Code attempted during session with formatting preserved
language	string	Programming language of the code

Design: MVC

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.

REST APIs

GET /getUserHistory	
Usage	Retrieve user history based on the provided username. Records are displayed in Questions Table on Index page.
Request Param	userName (Query Parameter): The username for which to retrieve the history.
Returns	Status Code: 200 if successful. User history containing {user1, user2, roomId, time, code, questionId}
POST /create	
Usage	Create or update a history record. Called in Collab page when saving a History record upon change question / end session
Request Body	roomid: The ID of the room. user1: User1's information. user2: User2's information. time: end time of session code: Code for that question questionid: The ID of the question. language: The programming language used.
	If the [roomid, questionid] record does not exist, it will create a new record in the History table. Else, it will update the existing record, since it refers to the same room and question. This prevents duplicate entries when two users in the same session attempt to save the same question.

6.7 Features

6.7.1 Collaborative Code Editor

```
1 public class HelloWorld { //do not change the classna
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
6
```

The Editor Service can be divided into the two components listed below

Frontend	CodeMirror powers our collaborative code editor.
Backend	Firebase Real-time Database powers our backend synchronization

We have chosen **CodeMirror** as our code editor and have justified the decision in the table below.

Alternative	Justification
Ace Editor	Includes more features such as multiple cursors, code folding and has a larger set of supported languages. However, it is more complex and requires more configuration out of the box to use.

Advantages of using CodeMirror
Support for multiple languages (i.e. C, Java, Python…)
Many third-party add-ons to improve user experience (i.e. automatic inclusion of brackets)
Extensive Documentation available

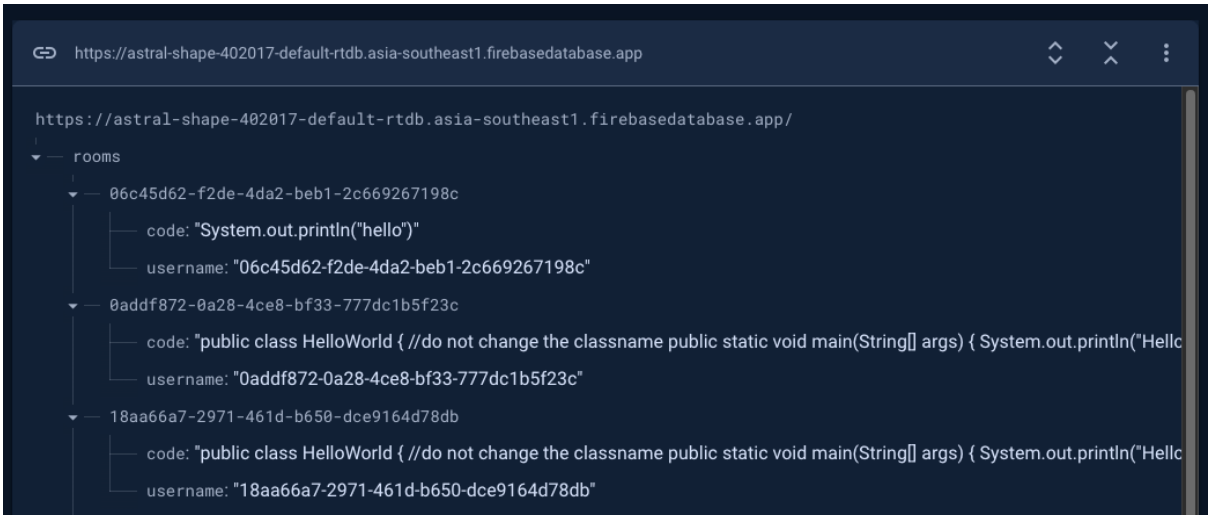
Firebase Real-time Database

Alternative	Justification
Custom designed ActiveMQ Message Broker	Message brokers need to have a backend configured, while for Firebase Realtime Database there are frontend libraries available which require less configuration.
Pubsub mechanism (i.e. Kafka / Redis)	Have to manually do real-time implementation using socket as compared to using Firebase Realtime Database frontend libraries.

Advantages of using Firebase Realtime database

- Persistent, data will be stored in the Firebase Realtime database will be persisted
- Real-time support in browsers with extensive documentation and examples available. Given that Firebase Realtime Database has been around since 2012, there are many examples and documentation which will make development a swift process.

All data stored in the Firebase Realtime Database is stored as JSON objects. The ‘rooms’ key will contain a list of all the rooms that are currently available.



The code is uploaded in real-time into the Firebase Realtime Database as the user types into the code editor and the update of the code editor of users in the same room will be synchronized.

```
function writeUserData(code: string) {
  const db = getDatabase();
  set(ref(db, "rooms/" + roomId), {
    username: roomId,
    code: code,
  });
}
```

Figure 6.7.1.1 Uploading code in real-time into Firebase Realtime Database

The Firebase Realtime Database of the specified room will be updated with the code currently in the editor.

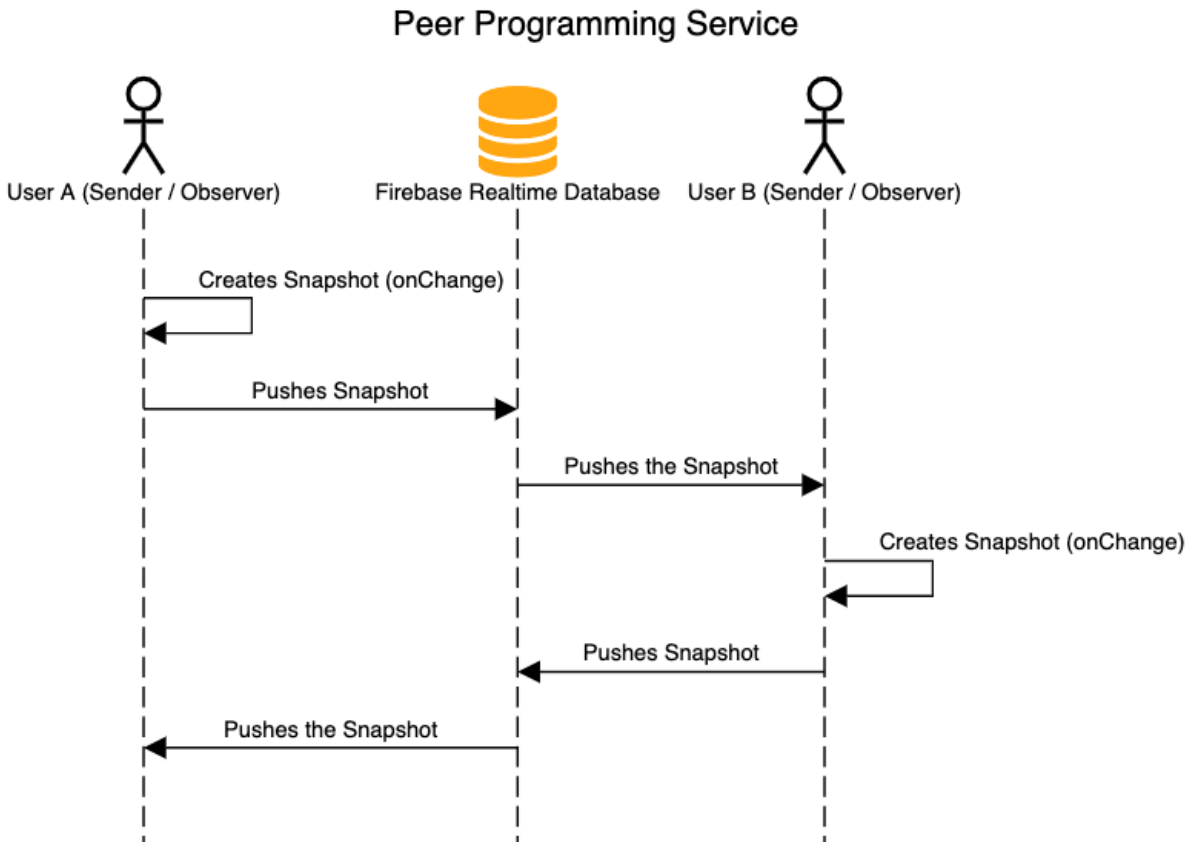


Figure 6.7.1.2 Peer Programming

Here, we can see how an update is propagated to all peers in real-time. Whenever there is a change in the contents of the code editor, the client's code editor will capture a snapshot of its current state, before pushing it to the Firebase Realtime Database. The same snapshot will be pushed to observers that are subscribed to the same "room" (i.e. the other matched user) and replace the contents of the code editor. The same thing happens when the other user now makes a change to the code editor. This results in both the selected modes, as well as the contents of the code editor being synced in real-time.

We support the following four languages in the CodeMirror editor (C, Python, JavaScript, Java). The default mode when users first begin their session is JavaScript. Users can toggle to change the language at any point in time and the code editor will be reset (i.e. all written code will be lost).

6.7.2 Video

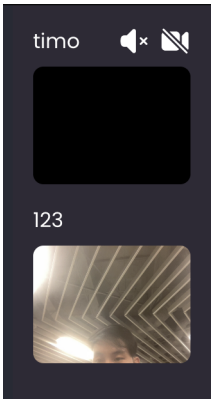


Figure 6.7.2.1 Video Service example between two users

Video Service provides video-call capabilities to users during their collaboration session.
It is implemented using a JavaScript library PeerJS, which wraps the browser's WebRTC (Web Real-Time Communication) implementation to provide a complete, configurable, and easy-to-use peer-to-peer connection API. A peer can create a media stream connection to a remote peer just by calling the remote peer's assigned peerId.
It builds on the WebRTC technology to enable direct communication between web browsers, instead of relying on a central server.

6.7.2.1 PeerServer

To broker connections, PeerJS (client) connects to a PeerServer. Note that no peer-to-peer data goes through the server, which lightens the load on the PeerServer. The server acts only as a connection broker.

Once peers are aware of each other's IDs, one peer initiates the connection by calling the connect method on the other peer's ID.

```
1  const connection = peer.connect(otherPeerId);
```

Figure 6.7.2.1.1 Initiating connection to 'otherPeerId'

6.7.2.2 Justification - PeerJS

Pros		Cons	
Simplicity and Abstraction	PeerJS abstracts away many of the complexities of WebRTC, making it easier for developers to implement real-time communication features. Pro: The API is designed to be straightforward, reducing the learning curve for integrating video call functionality.	**Limited Customization**	While PeerJS provides a convenient abstraction, it might limit customization options for developers who require more control over the underlying WebRTC components. However, for our simple use case of only requiring audio and video transfer, PeerJS provides sufficient customisation.
Cross-Browser Compatibility	PeerJS handles browser inconsistencies and provides a consistent interface for communication, ensuring cross-browser compatibility.	**Security Considerations**	PeerJS doesn't provide built-in security features, such as end-to-end encryption. Developers need to implement additional security measures if encryption is a crucial requirement.
Minimal Server-side code	Pro: PeerJS includes a server (PeerServer) that helps with NAT traversal, making it simpler to establish connections between peers behind different network configurations. No Server-Side Code Required: Pro: Minimal server-side code is needed to get started. PeerJS provides a hosted signaling server, eliminating the need for developers to set up their own.		
Lighter load on server	Relying on peer-to-peer connection for audio and		

	video transfer instead of server as the middleman		
Easily scalable	For large-scale applications, we can leverage container-orchestration technologies like Kubernetes to create more instances of PeerServer to handle higher connection demands.		

Flow of events from matching to setting up of Collab page (chat, video, collaborative editor)

Upon being matched, the matching service will send the following information to each user in the pair.

```
const roomDetails = {
  roomId: roomId,
  myId: firstUsername,
  otherId: secondUsername,
  difficulty: msg.difficulty,
};
```

Figure 6.7.2.2.1 URL parameters for Collab page

User will then be routed to the Collab page with the following URL parameters:

Parameter Name	Description

roomId	- unique roomId which will be used on
	  - Firebase real-time database to synchronize code
	  - chatroom id for chat

myId	The id here will be the user's username, which will be unique for all users, so there will not be collisions when PeerServer is brokering connections.
otherId	id which PeerJS will send to PeerServer to connect both users for video-call.
difficulty	difficulty of question

for Collab page to specify when getting question from Question Service

```
router.push(
  `/Collab?roomId=${msg.roomId}&myId=${msg.myUsername}&otherId=${msg.otherUsername}&difficulty=${msg.difficulty}`
);
```

Figure 6.7.2.2.2 Routing to Collab Page

The choice of placing these fields in the url is to
1) pass these essential data from the Index page to the Collab page,
2) persist them upon refresh.

6.7.3 AI Code Generation

We have incorporated an AI Code Generation service to allow users to have a reference of the ideal answer from an AI on what the answer should look like. We are using the Vertex AI service from Google Cloud Platform to generate code solutions for the problem.

To get started, users can click on the ‘Generate’ button on the IDE when they are in a room.

Figure 6.7.3.1 Generate button on IDE

A modal will pop up and users can select the language that they want to generate the code in.

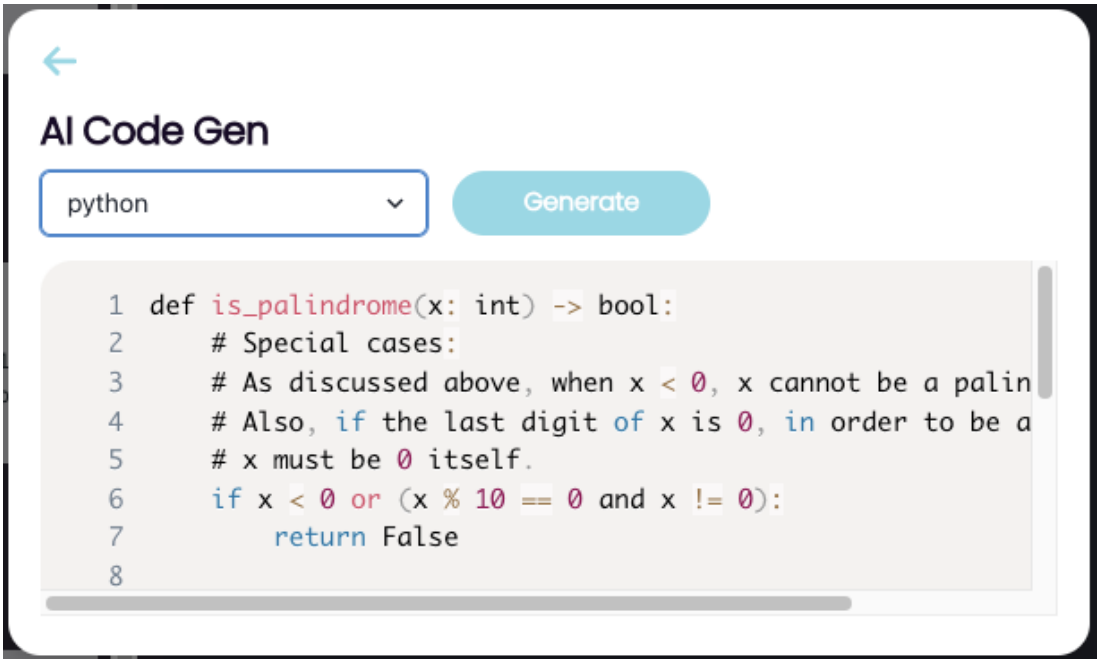


Figure 6.7.3.2 AI Code Generation feature

Users can click on the ‘Generate’ button and the Vertex AI service will generate a solution where the user can reference.

We send a prompt to the Vertex AI code generation REST API service that includes the question and language selected.

6.7.4 Summary of Collaboration Page interaction with Back-end services

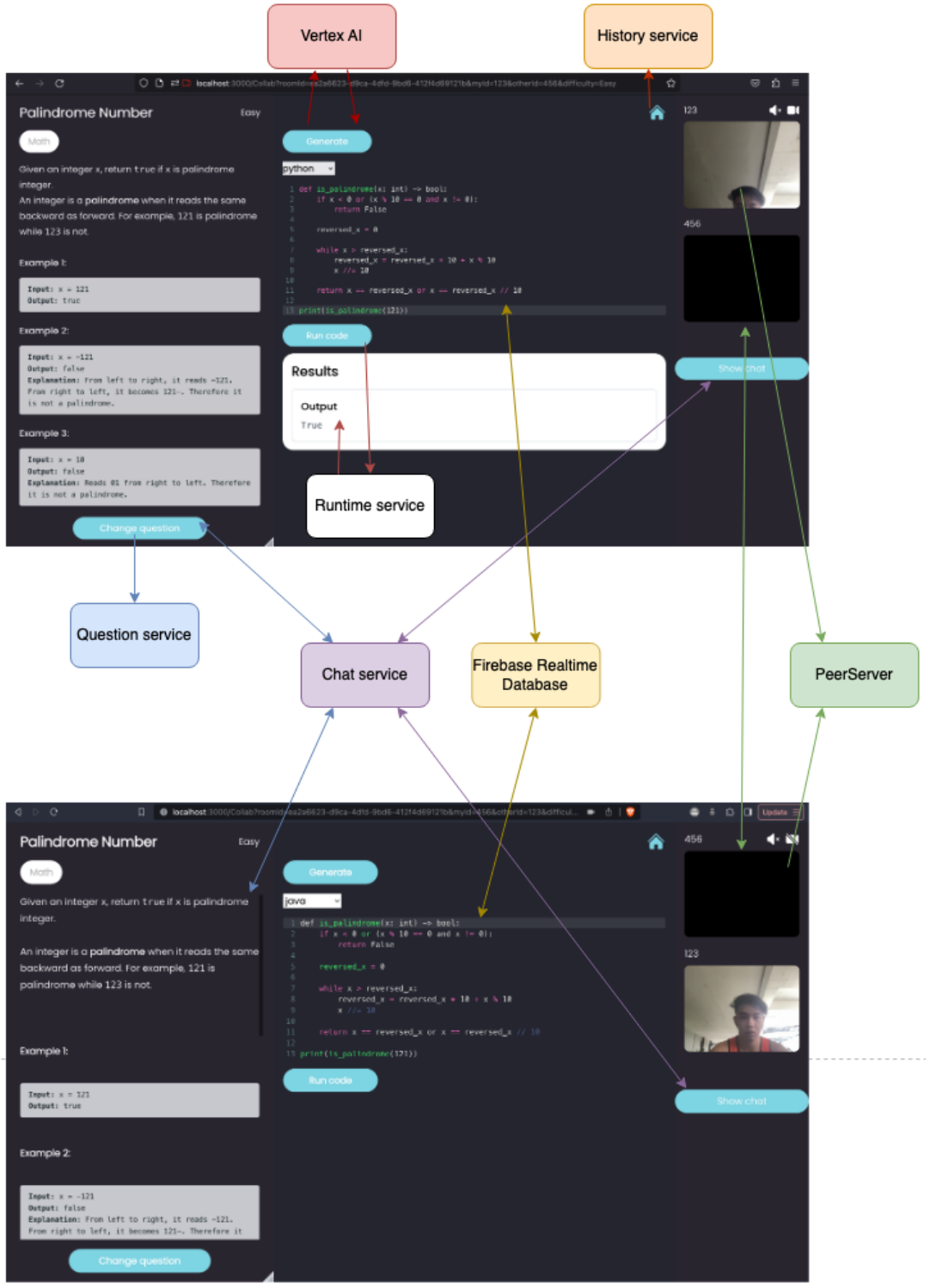
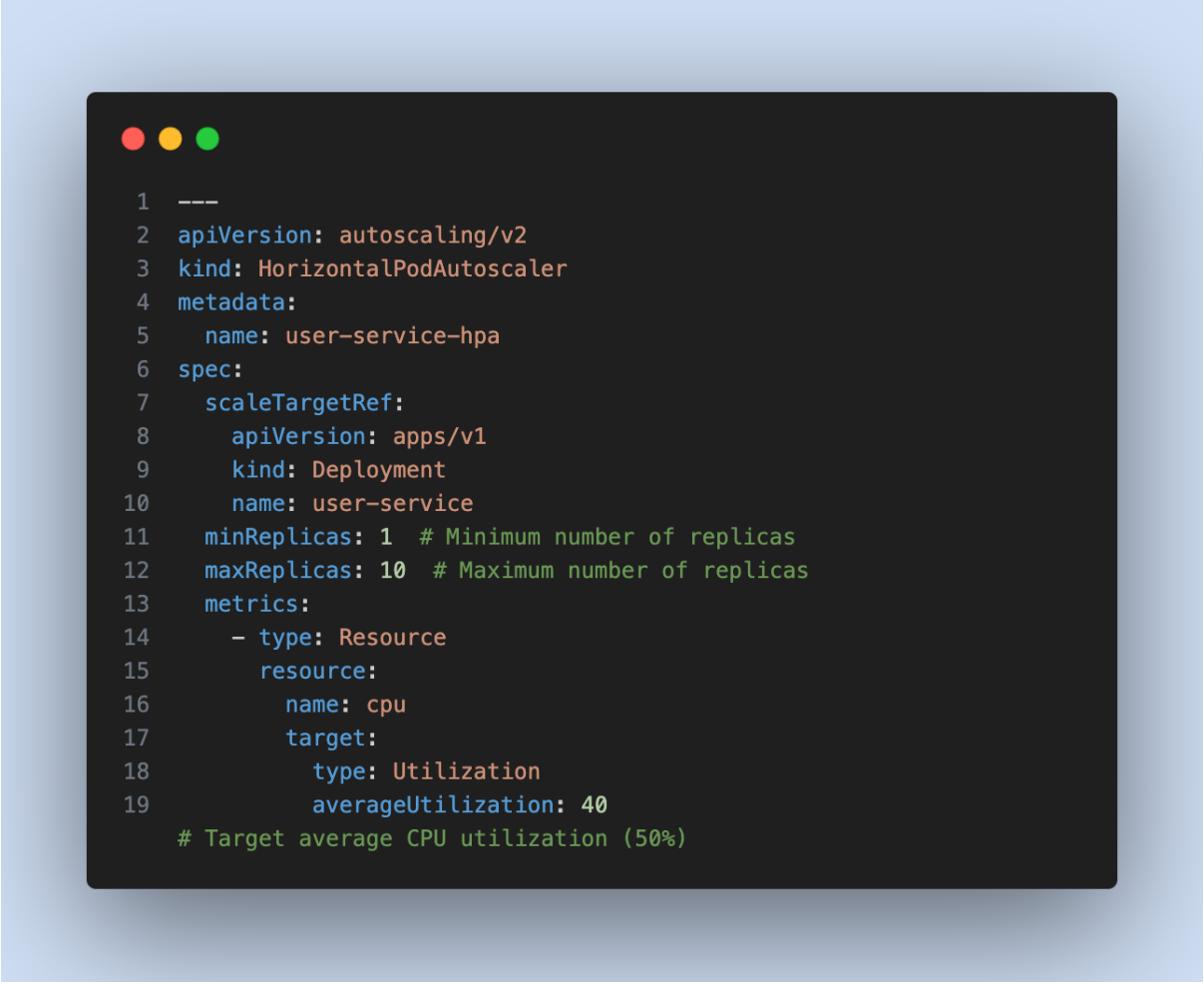


Figure 6.7.4.1 Summary of Collaboration Page

6.7.5 Scaling

Scaling is done in Kubernetes where it automatically adjusts the number of running instances (pods) of a particular service based on demand. We primarily use the Horizontal Pod Autoscaler (HPA).



```yaml
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: user-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: user-service
  minReplicas: 1  # Minimum number of replicas
  maxReplicas: 10  # Maximum number of replicas
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 40
# Target average CPU utilization (50%)
```

Figure 6.7.5.1 Horizontal Pod Autoscaler

In the example above is the Horizontal Pod Autoscaler for the User Service. We set a minimum of 1 replica and a maximum of 10 replicas. The average utilization is set at 40% which refers to the target average utilization of the specific resource and the HPA aims to keep the average CPU utilization of all pods in the specified deployment at 40%. When the average CPU utilization of all pods exceeds the specified target of 40% in this case, the HPA will automatically scale up the number of replicas to handle the increased demand. Similarly, if the utilization drops below the target, the HPA will automatically scale down the number of pods to ensure optimal performance and resource efficiency.

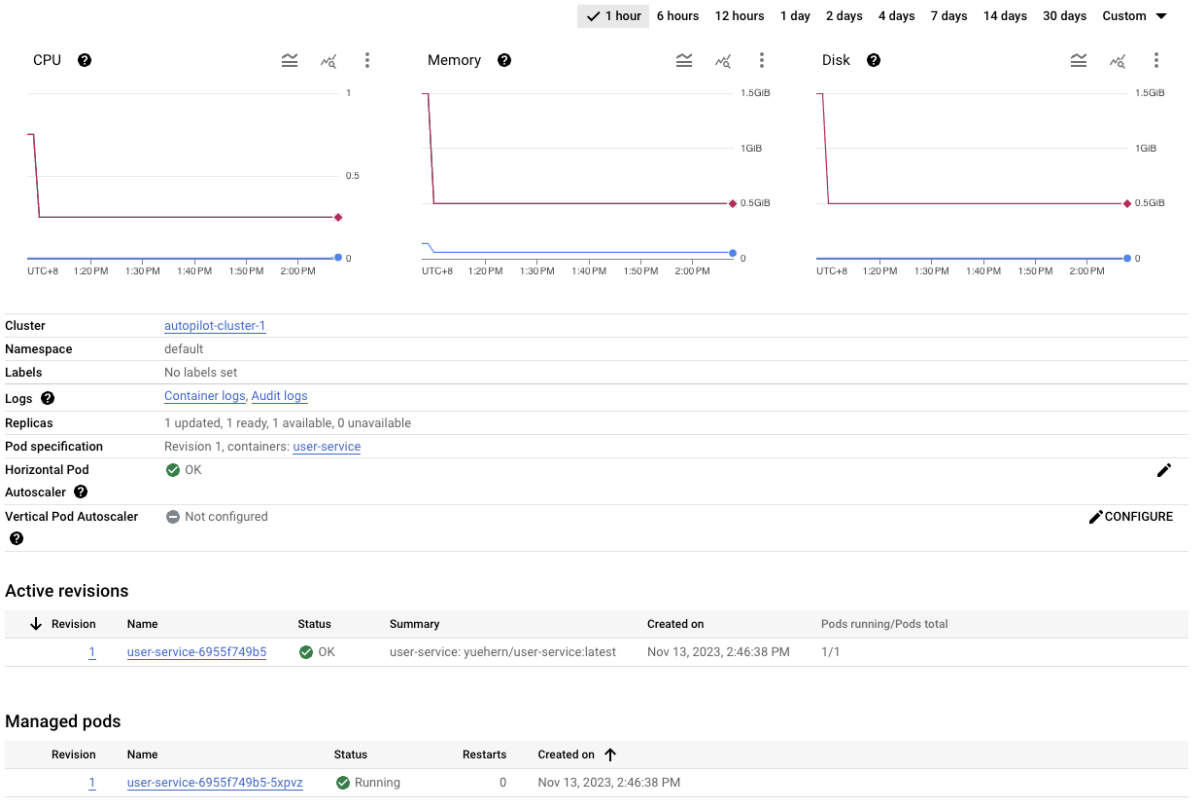


Figure 6.7.5.2 Initial state of Kubernetes Deployment

The image above shows the initial state of the Kubernetes Deployment where there is low utilization and only 1 pod running as stated for our minimum replicas.

```
(base) yuehernkang@Yues-MacBook-Pro-2 ay2324s1-course-assessment-g35 % ab -n 1000 -c 10 http://34.126.107.52:8000/
This is ApacheBench, Version 2.3 <$Revision: 1903618 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.126.107.52 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:
Server Hostname:        34.126.107.52
Server Port:            8000

Document Path:          /
Document Length:        0 bytes

Concurrency Level:      10
Time taken for tests:   40.520 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      177000 bytes
HTML transferred:       0 bytes
Requests per second:    24.68 [#/sec] (mean)
Time per request:       405.203 [ms] (mean)
Time per request:       40.520 [ms] (mean, across all concurrent requests)
Transfer rate:          4.27 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        5  389 2662.4      9   19035
Processing:     7   11   6.6     10      97
Waiting:        5   11   6.5     10      97
Total:         12  401 2662.1     19   19044

Percentage of the requests served within a certain time (ms)
  50%     19
  66%     20
  75%     21
  80%     22
  90%     26
  95%     37
  98%  19021
  99%  19025
 100%  19044 (longest request)
```

Figure 6.7.5.3 Stress Testing user-service

We use Apache Benchmark to stress test our user-service in this example where we sent a total of 1000 requests to the user-service and we specify to send 10 concurrent (simultaneous) request to the service.

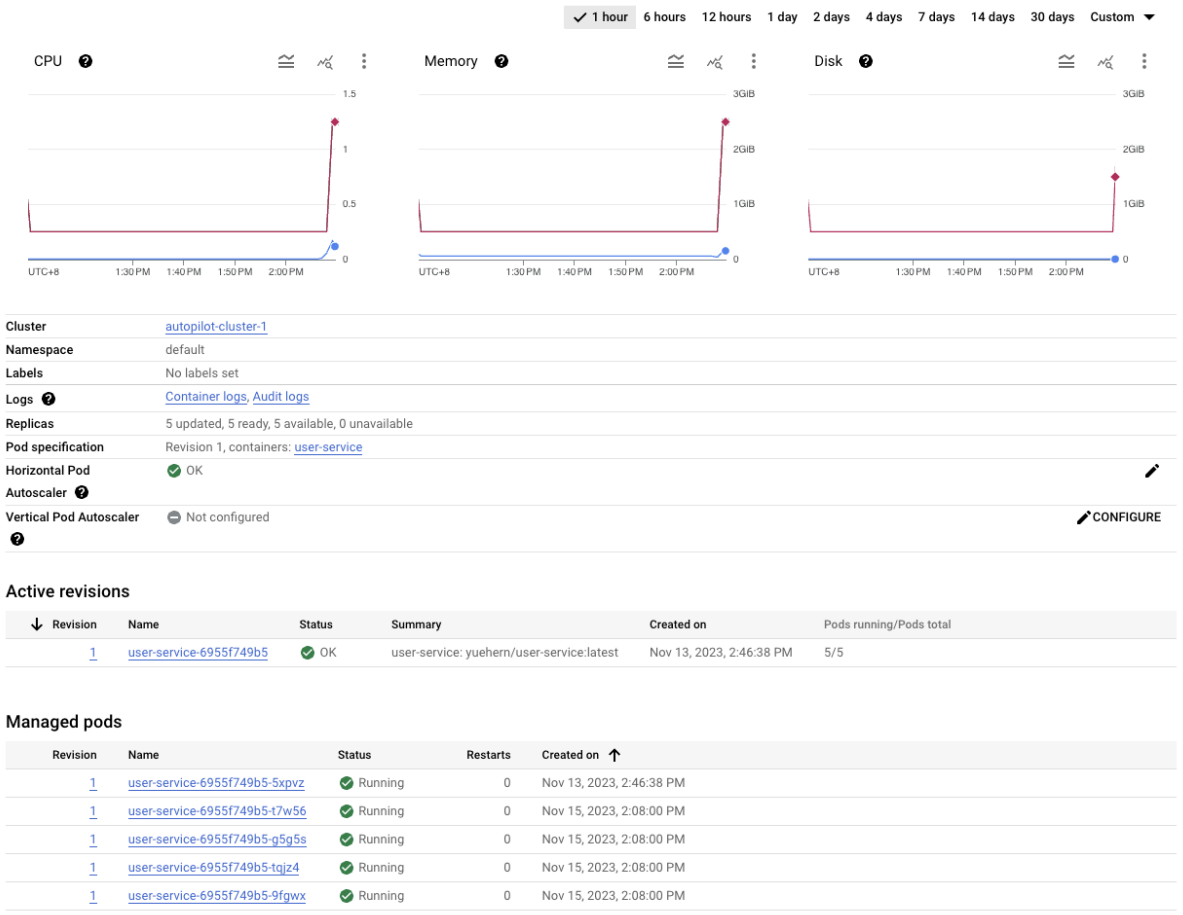


Figure 6.7.5.4 Results of Stress Testing user-service

We can see here the result of the stress testing will have 4 additional pods due to the Horizontal Pod Autoscaler (HPA), giving a total of 5 pods as there was an increased load on the user-service and automatic scaling has taken place. This allows the service to take on additional workload.

6.8 DevOps

6.8.1 Jenkins

We use Jenkins build our code to ensure that there are no errors. Everytime there is a push or pull request to the GitHub repository, the pipeline will be triggered to run. This is to ensure that all code can be built and Dockerize without any issues before merging into the master branch.

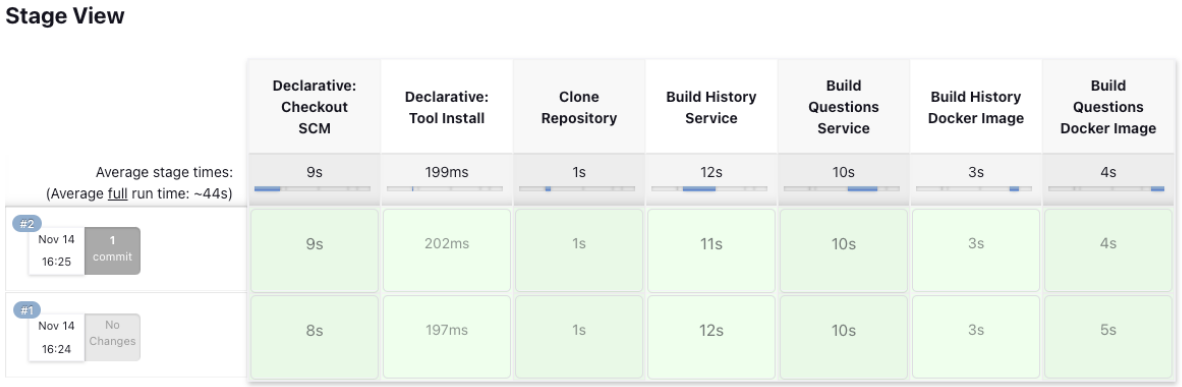


Figure 6.8.1.1 Pipeline stages

Pipeline stages include:

1. Building History service
2. Building Question service
3. Building History service Docker image
4. Building Question service Docker image

```
stages {
        stage('Clone Repository') {
            steps {
                checkout scm
            }
        }

        stage('Build History Service') {
            steps {
                sh '''
                    # Navigate to your Node.js app directory
                    cd history-service

                    # Install dependencies
                    npm install

                    # Build your Node.js application
                    npm run build
                '''
            }
        }
        stage('Build Questions Service') {
            steps {
                sh '''
                    # Navigate to your Node.js app directory
                    cd questions-service

                    # Install dependencies
                    npm install

                    # Build your Node.js application
                    npm run build
                '''
            }
        }
```

Figure 6.8.1.2 Part of the Jenkinsfile Pipeline Steps

We use a Jenkinsfile at the root of the repository to specify the actions to be taken such as build a service or Docker image.

S	W	Name ↓	Last Success	Last Failure	Last Duration
✓	☀	change-url	7 hr 32 min #2	N/A	44 sec
✓	⛈	jenkins-setup	9 days 20 hr #50	9 days 21 hr #49	19 sec
✓	⛈	jenkins-test	7 hr 45 min #11	7 hr 49 min #10	1 min 40 sec
✗	⛈	kubernetes	N/A	7 days 8 hr #12	7.1 sec
✓	☀	master	5 hr 36 min #2	N/A	51 sec
✗	⛈	questions-service	N/A	12 days #6	7 sec

Figure 6.8.1.3 Jenkinsfile Multibranch Pipeline

For each branch that has a Jenkinsfile at the root directory, a pipeline will be created for that branch and each push to that branch will trigger the pipeline to run.

6.9 Deployment

Service/Application Name	Deployment Type	Exposed Using	Exposed To
MongoDB	StatefulSet	ClusterIP	Internal Only
PostgresQL	StatefulSet	ClusterIP	Internal Only
User Service	Deployment	LoadBalancer	External
Real-Time Matching	Deployment	LoadBalancer	External
Question Service	Deployment	LoadBalancer	External
History Service	Deployment	LoadBalancer	External
Chat Service	Deployment	LoadBalancer	External

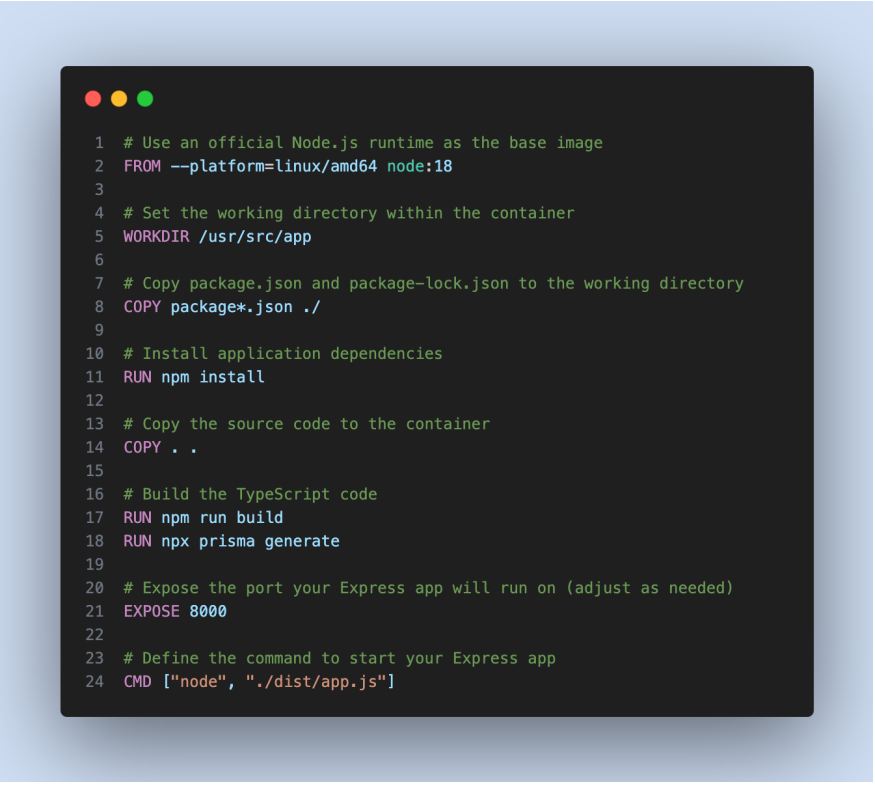
6.9.1 Stateful Deployment

Stateful application deployment is done using StatefulSet. StatefulSets will be used for applications such as the databases in our application. Our MongoDB and the PostgresQL databases are deployed using StatefulSet along with Persistent Volume Claim so that data will be persisted.

6.9.2 Stateless Deployment

Stateless applications such as the User Service and History Service will be deployed as Deployments in our Kubernetes Cluster.

The frontend and Services such as User Service, History Service and Question Service will be first built and pushed to Docker Hub, then they will be deployed on Google Kubernetes Engine (GKE).



```dockerfile
# Use an official Node.js runtime as the base image
FROM --platform=linux/amd64 node:18

# Set the working directory within the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install application dependencies
RUN npm install

# Copy the source code to the container
COPY . .

# Build the TypeScript code
RUN npm run build
RUN npx prisma generate

# Expose the port your Express app will run on (adjust as needed)
EXPOSE 8000

# Define the command to start your Express app
CMD ["node", "./dist/app.js"]
```

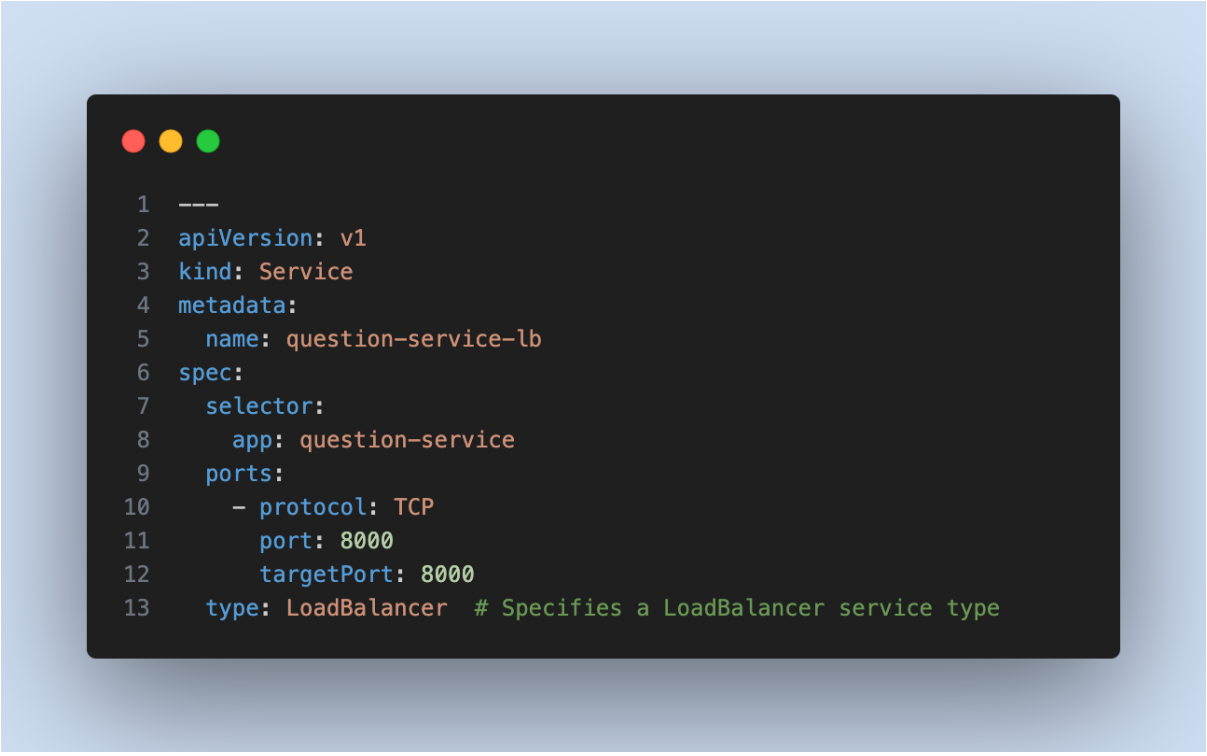
Figure 6.9.2.1 Dockerfile for services

This is the Dockerfile used to build our services into a container image. The image is then pushed to Docker Hub.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: question-service
spec:
  selector:
    matchLabels:
      app: question-service
  template:
    metadata:
      labels:
        app: question-service
    spec:
      containers:
        - name: question-service
          image: yuehern/question-service:latest
          env:
            - name: QUESTIONS_DB_URL
              value: "34.118.229.101"
# Set the database connection URL
            - name: PORT
              value: "8000" # Set the database connection URL
          ports:
            - containerPort: 8000  # Adjust the port as needed
          resources:
            limits:
              cpu: "0.5"  # Maximum CPU limit (0.5 CPU cores)
            requests:
              cpu: "0.2"  # CPU requested (0.2 CPU cores)
```

Figure 6.9.2.2 History Service Kubernetes yaml file

The Kubernetes YAML file shown in Figure 6.9.2.2 will be used to deploy the service as a Deployment and a LoadBalancer to expose the service will also be created.



```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: question-service-lb
spec:
  selector:
    app: question-service
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
  type: LoadBalancer  # Specifies a LoadBalancer service type
```

Figure 6.9.2.3 Question Load Balancer

For external facing services such as the User Service, History Service and Question Service, we use an External Load Balancer to expose that service. The example above shows the Question Service using a Load Balancer to expose port 8000.