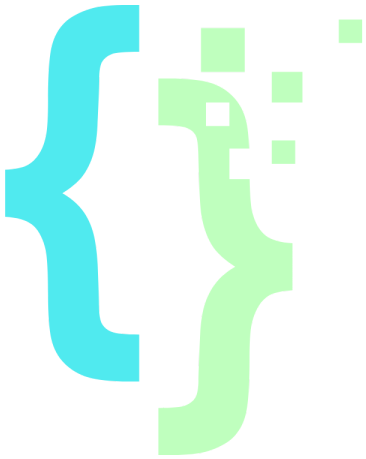


PeerPrep

G17 Project Report

CS3219 Software Engineering Principles and Patterns

Name	Student No.
N Vijay Narayanan	A0214584A
Lin Zhiwei	A0218495U
Dominic Lim Kai Jun	A0221012J
Nicole Joseph	A0226610R

Source Code:

https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g17

Deployed At:

http://alb-peerprep-2137662650.ap-southeast-1.elb.amazonaws.com/

Table of Contents

Table of Contents **2**

Introduction **5**
- Background 5
- Project Scope 5

Product Overall **6**
- Product Perspective 6
- Operating System 6

Individual Contributions **7**

Requirements Specification **8**
- Functional Requirements 8
  - User Service 8
  - Matching Service 9
  - Question Service 10
  - Collaboration Service 10
  - History Service 11
  - Messaging Service 11
- Non-Functional Requirements 12
  - Quality Attributes 12
    - Usability 12
    - Performance 18
    - Reliability 26

Features **30**

Sprint Planning **33**

Application Design **36**
- Tech Stack 36
- Overall Architecture 38
- User Activity Flow 40
- Frontend Architecture 41
  - Component Tree 41
  - Design Pattern: Context-Provider 46
- Backend Architecture 49
  - RESTful API 49
    - Design Pattern: Model-View-Controller (MVC) 49
      - Overview 49

Controller 49
Model 49
View 49
Benefits 50
Design Decision: Object-Oriented Programming 51
User Service 53
Login Flow 53
Question Service 54
History Service 55
Socket API 55
Introduction 55
Design Pattern: Publisher-Subscriber 56
Matching Service 58
Matchmaking Flow (Happy Path) 60
Matchmaking Flow (No Match) 61
Collaboration Service 62
Collaboration Room Code Editor Flow 66
Collaboration Room Timer Ticking Flow 67
Collaboration Room Timer Start / Stop Flow 68
Collaboration Room Timer Pause / Resume Flow 70
Collaboration Room Next Question Flow (Happy Path) 72
Collaboration Room Next Question Flow (Rejection) 73
Messaging Service 73

DevOps 75
Local Deployment/Development with Docker Compose 75
Deployment on AWS 78
Introduction to Elastic Container Service 78
Task definitions and Tasks 78
ECS Service 79
Elastic Container Registy (ECR) 79
Overview of deployment 79
Key deployment decisions 80
CI/CD with GitHub Actions 82

Improvements & Enhancements 84

Reflection & Learning Points 85

Appendix 87
UI/UX Hi-Fidelity Prototypes 87

- Style Guide 87
- Hi-Fi Mockups 87

API Documentation 88

- User Service 88
  - Health Check 88
  - Register User 88
  - Login 89
  - Logout 90
  - Change Password 91
  - Delete Account 92
  - Refresh 92
  - Change Language 93
- Question Service 95
  - Health Check 95
  - Get Question 95
  - Get Questions (Difficulty) 96
  - Get Questions (All) 98
  - Count Questions 99
  - Count Questions (Difficulty) 100
  - Seed Questions 101
- History Service 102
  - Health Check 102
  - Get History 102
  - Get History Records 104
  - Save History 106
  - Delete History 107
  - Get Statistics 108
  - Add Statistics 110

Milestone Timelines 111

- Milestone 1: MVP and Project Requirements 111
- Milestone 2: Project Progress Check 113
- Final Milestone: Submission, Demo & Presentation 115

Introduction

Background

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

Project Scope

PeerPrep is targeted at students who are actively looking for internships or new graduate positions. It can also be used by those who want to practise interviewing through mock interviews with their peers. This form of practice will help build the users' confidence and experience and give them a better sense of how a real technical interview is conducted.

The platform hosts a room for a pair of users - the interviewer and interviewee, which will provide a pair of questions based on the difficulty that both users have selected before matching is carried out. These questions are from a popular interview preparation website, Leetcode.

We have provided users with a dashboard to help motivate them on their practice and get a sense of inspiration from their peers by allowing cross viewing of their profile.

Product Overall

Product Perspective

We aim to develop a platform that helps students prepare for their technical interviews by conducting mock interviews with another peer that they do not know. This simulates the full environment of a real technical interview. Similar to the real one, each pair of users will receive a pair of questions based on the difficulty that they have selected e.g., Easy, Medium or Hard, before matching with the other participant. They can collaborate on the problem via a real time in-built browser code editor with a live chat to communicate. Once a session is completed, the state of it will be saved to each user and some statistics will be generated and previewed on the dashboard upon logging in.

Operating System

The team's development spans across Linux (macOS) and Windows. The back end tools that we have used e.g., Docker Desktop, Redis, etc., are operable across both operating systems. The front end requires us to use a web browser of which we have chosen Google Chrome v107.

Individual Contributions

Member	Contribution to Codebase	Contribution to Documentation

Vijay (Fullstack Engineer)	- UI for auth, match, and landing pages
	- UI for profile dashboard
	- UI for collaboration room
	  - Code editor
	  - Stopwatch
	  - Question display
	  - Chat box
	- UI for matching queue
	- User Service
	- Collaboration Service (timers)
		- Tech stack rationale
		- Frontend architecture diagrams
		- Overall architecture diagrams
		- Frontend design patterns
		- Quality Attributes (shared)
		- Application Design (shared)
		- Improvements & Enhancements (shared)

Zhiwei (DevOps & Backend Engineer)	- Matching Service - Collaboration Service - CI/CD	- DevOps - Matching Service - Overall Architecture - Reflection and Learning Points (shared)

Dominic (Backend Engineer)	- Question Service
	  - Data source of questions
	  - Question model and controller
	- History Service
	  - History model and controller
	  - Statistics model and controller
		- API Documentation
		- Requirements Specification (shared)
		- Quality Attributes (shared)
		- Application Design (shared)
		- Improvements & Enhancements (shared)
		- Reflection and Learning Points (shared)
		- Milestone Timelines
		- Introduction
		- Product Overall

Nicole (Frontend Engineer & UI/UX)	- UI/UX - Frontend - Messaging Service	- Quality Attributes (shared) - Features - Requirements Specification (shared) - Sprint Planning - UI/UX Hi-Fidelity Prototypes - Table of Contents

Requirements Specification

Functional Requirements

The Functional Requirements of our system are categorised based on the respective service they belong to, with a priority indicator for each.

User Service

S/N	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 ensure that users stay logged in when they refresh the browser.	High

FR1.7	The system should block logins when the wrong credentials are provided.	High
FR1.8	The system should allow users to change their password.	Medium

Matching Service

S/N	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 who selected the same difficulty level and put them in the same room.	High
FR2.3	The system should indicate to a waiting user how long that user has been waiting, counting down from 30 seconds.	High
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	Medium
FR2.5	The system should provide a means for the user to leave a room once matched.	Medium
FR2.6	The system should allow users to cancel the matching process.	Medium
FR2.7	The system should match the users instantly if there is a match.	High
FR2.8	The system should timeout a user if finding a match takes longer than 30 seconds.	High

Question Service

S/N	Requirement	Priority
FR3.1	The system should be able to be seeded with a list of questions.	Medium
FR3.2	The system should be able to provide a pair of random questions of a specific difficulty level.	High
FR3.3	The system should be able to provide a specific question based on its unique identifier.	High

Collaboration Service

S/N	Requirement	Priority
FR4.1	The system should allow both participants in a room to enter text into a text editor.	High
FR4.2	The system should display all text entered by the other participant and vice versa when both are in the same room.	High
FR4.3	The system should allow either participant to begin, pause, resume or stop the session timer.	Low
FR4.4	The system should allow either participant to request to move on to the next question if there are more questions remaining for that session.	Medium
FR4.5	The system should allow either participant to request to end the session if there are no more questions remaining for that session.	Medium

History Service

S/N	Requirement	Priority
FR5.1	The system should be able to display the state (question, code and chat) of a coding session for both participants.	High
FR5.2	The system should be able to display to users their past coding sessions in order of descending time.	High
FR5.3	The system should inform users that history will only be saved if the coding session is ended properly.	Medium
FR5.4	The system should allow users to retrieve their own history.	Medium
FR5.5	The system should allow users to retrieve their past pair's history.	Medium
FR5.6	The system should be able to churn out statistics on the past interviews done by a user.	Medium

Messaging Service

S/N	Requirement	Priority
FR6.1	The system should allow participants of a coding session to chat with each other.	High
FR6.2	The system should indicate the time that each message was sent.	Medium
FR6.3	The system should visually distinguish messages sent by each participant.	High

FR6.4	The system should allow chat to be hidden or brought into main focus.	High

Non-Functional Requirements

Quality Attributes

The table below illustrates the prioritisation of quality attributes.
Based on the scores, our 3 most important attributes are Usability, Performance & Reliability.

Attribute	Score	Availability	Integrity	Reliability	Performance	Robustness	Security	Usability	Verifiability
Availability	**4**		<	^	^	<	<	^	<
Integrity	**2**			^	^	^	<	^	<
Reliability	**5**				^	<	<	^	<
Performance	**5**					<	^	^	<
Robustness	**3**						<	^	<
Security	**1**							^	^
Usability	**7**								<
Verifiability	**1**								

We decided to categorise our Non-Functional Requirements at the system level instead of the microservice level.

Usability

Our website strongly focuses on usability to provide users with a great user experience by providing a modern user-friendly interface design with well thought out usability across different functions and pages. Users are able to interact with the

web application front end with low effort required on their inputs. For example, we use email and password for authentication. The sign up process is short and concise where we build it to request for only the relevant fields that suffice to use our system. Additionally, each page encompasses easy navigation across different pages and actions within the system.

S/N	Requirement	Priority
NFR1.1	The system should never expose users' password by any means of interactions either on the front end or back end.	High
NFR1.2	The system should display each incoming message almost instantly.	Mediu m
NFR1.3	The system should only move on to the next question or end the session if both participants consent to it.	High
NFR1.4	The system should minimally be able to support at least 1 waiting user per difficulty level at the same time.	High

NFR1.1 (High): The system should never expose users' password by any means of interactions either on the front end or back end.

We went with a high priority as users on any platform value that their account information be it personal information or credentials be stored safely. This sounds as though it is entirely in security, however, we are looking from a usability point of view.

To support this, on the front end, specifically the authentication form both registration and logging in, the password field serves as a visibility field where we provide a usable user interface for user's to enter their password with ease and a mind of comfort.

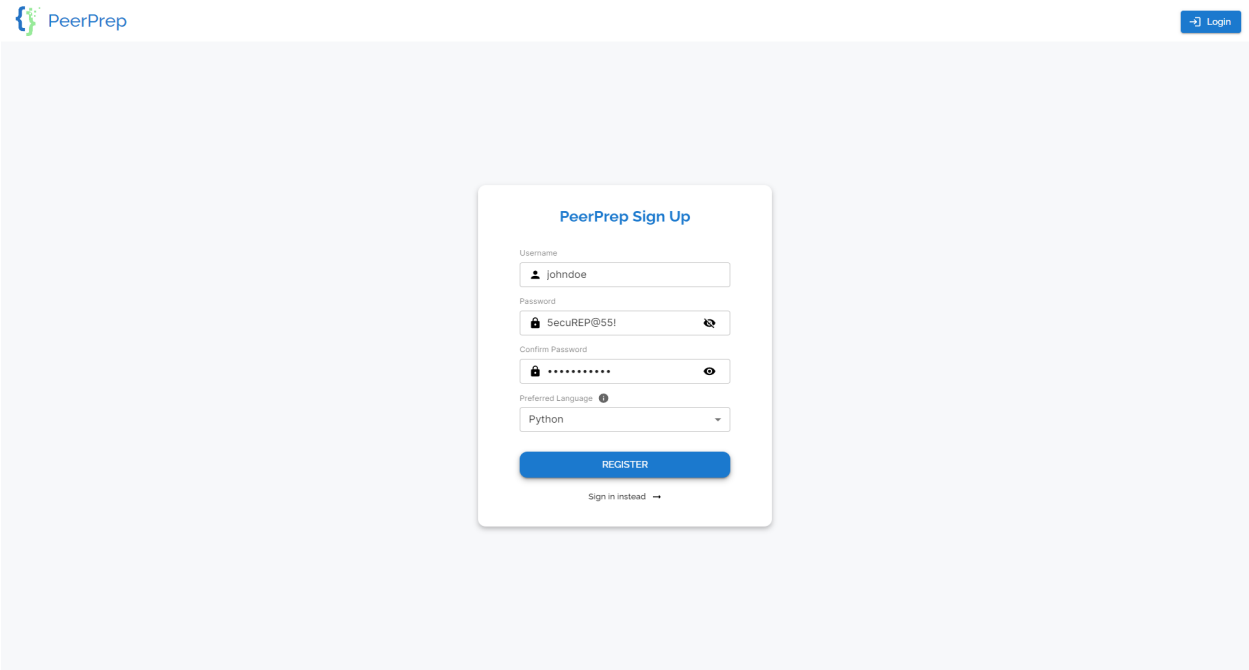


Figure 1: Sign up form with visibility fields (password)

On the back end, we have designed the user schema such that every request for data retrieval to the schema will never return the password (password: { ..., select: false }).

```
const userSchema = new mongoose.Schema<IUser, IUserModel>(
  {
    username: {
      type: String,
      unique: true,
      required: [true, 'Username is required'],
      minLength: [3, 'Username is too short'],
      maxLength: [12, 'Username is too long'],
    },
    password: {
      type: String,
      select: false,
      required: [true, 'Password is required'],
      minLength: [6, 'Password is too short'],
    },
    preferredLanguage: {
      type: String,
      enum: LANGUAGE,
      required: true,
    },
  },
  { timestamps: true }
);
```

Figure 2: A code snippet of User schema

NFR1.2 (Medium): The system should display each incoming message almost instantly.

We have selected medium for its priority as the end goal is to ensure that a message sent by a user to another e.g., Interviewee to interviewer, gets displayed on the receiver's chat section.

However, to substantiate this NFR, we have enhanced the way we handle generic peer-to-peer interaction (not limited to specifically communication) by using SocketIO with the Publisher-Subscriber (Pub-Sub) messaging pattern. This is in comparison with the method of using REST API to pass the message from the front end to messaging service and then back to front end where there are lots of overhead involved in the transportation of the single message.
The SocketIO server handles the communication between a pair of users in the same room. These users are connected to the same topic (room ID for this context) on the server. Hence, they are able to utilise this Pub-Sub messaging pattern as they are both consumers that have subscribed to the same topic. In this context, the event is instant messaging in the same topic between the two users. This implementation in turn reduces the overhead in interaction (be it communicating, collaborating, etc.) and allows for instant feedback.

NFR1.3 (High): The system should only move on to the next question or end the session if both participants consent to it.

We have selected high for the priority as we felt that for Usability to be our most important quality attribute, we will need to provide great user experiences for the actions or functionalities in our system as far as possible.

In each session between a pair of users, we provide prompts to them whenever either user wants to move onto the next question or end a session. This allows us to give both users a sense of usability by ensuring that they come to a mutual decision, else no one will proceed forward.

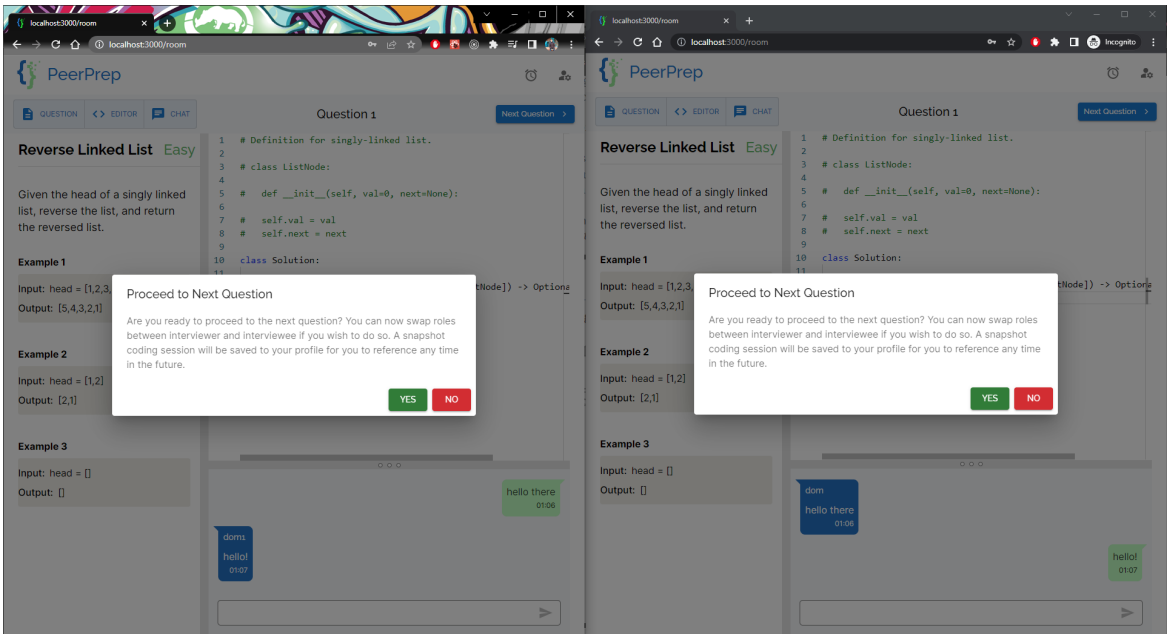


Figure 3: User wants to move onto the next question

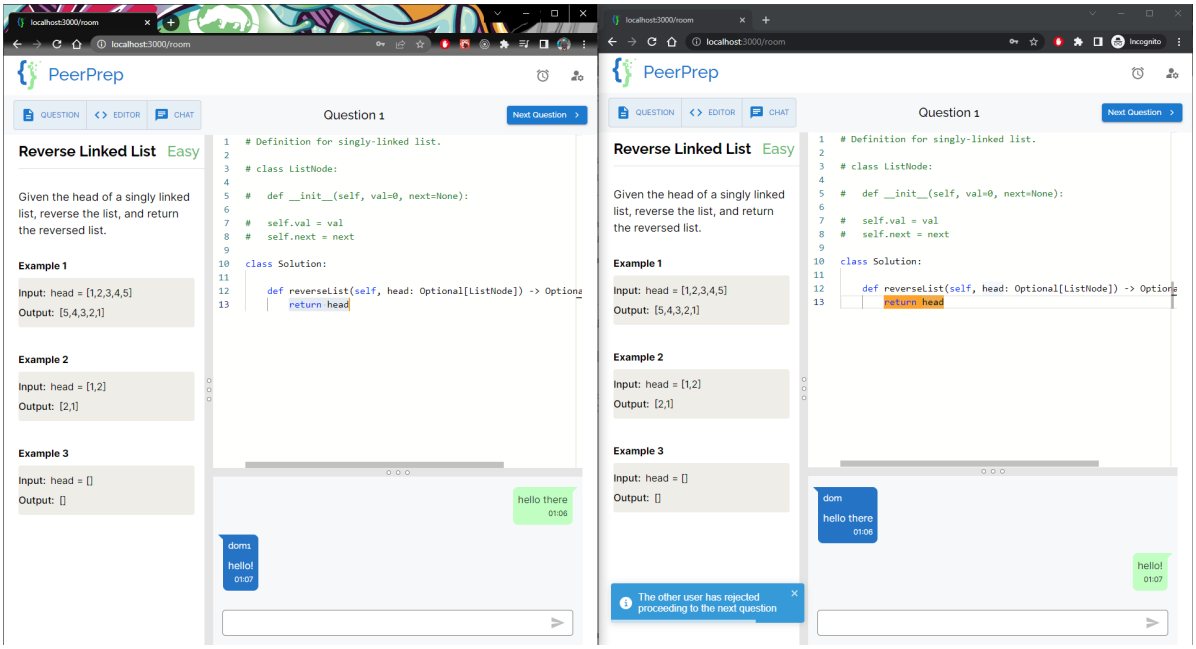


Figure 4: When a user selects Yes and the other selects No

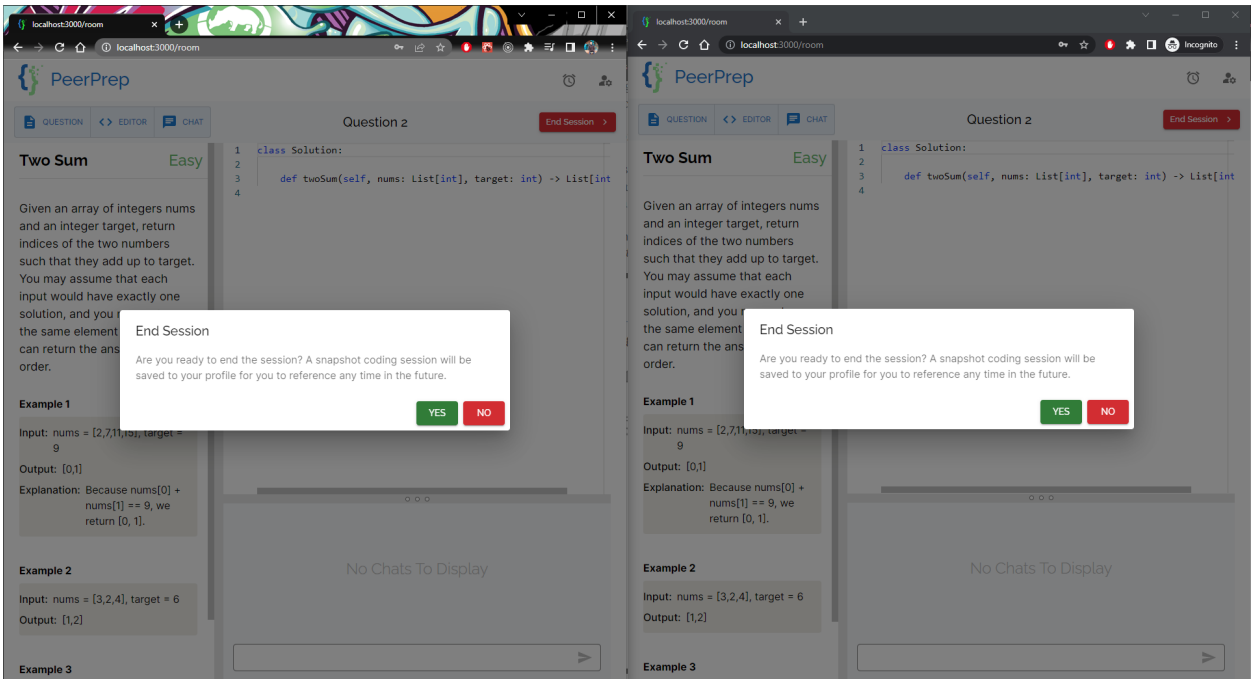


Figure 5: For when a user wants to end the current session

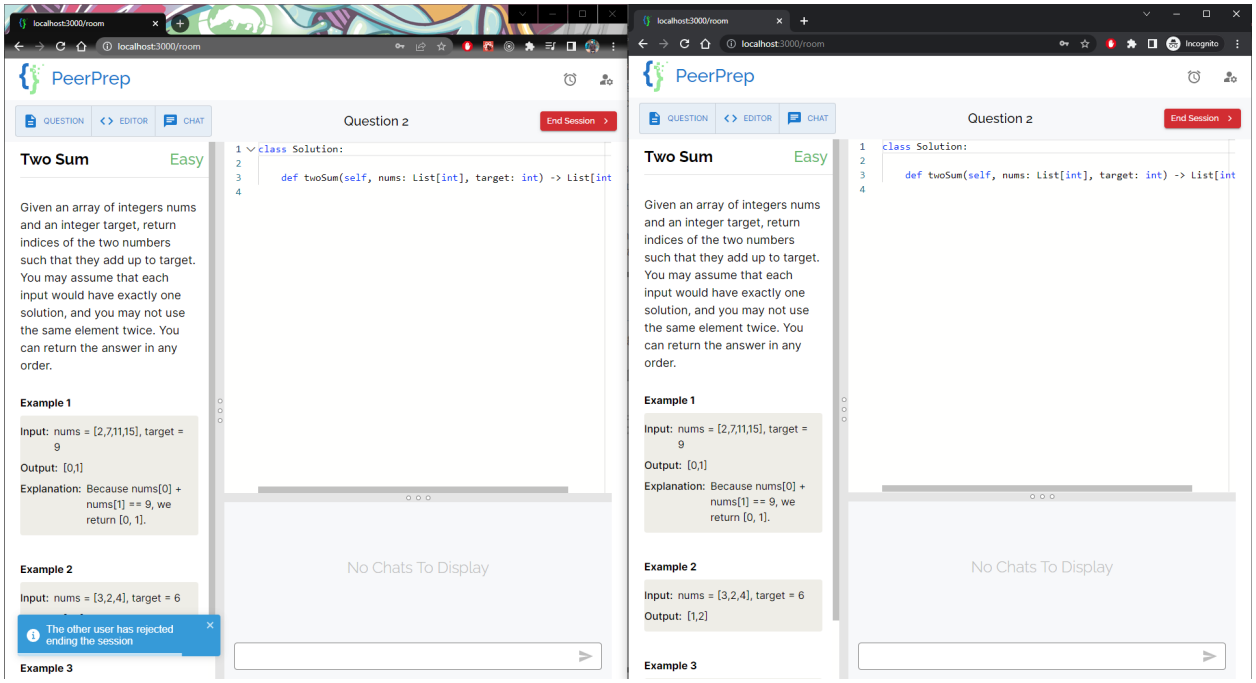


Figure 6: User (on the right) rejects the prompt to end the current session (bottom left)

Performance

Performance is another quality attribute that we have decided on as we are building a collaborative system between a pair of users. As our system provides real time collaboration and messaging, we have to try our best to ensure that our system is responsive and each action related to these areas i.e., Collaboration and messaging, has to be completed and feedback given to the users under a certain time frame.

S/N	Requirement	Priority
NFR2.1	The security measures imposed on a user e.g., Authentication and authorization, within the system should take no longer than 1 second.	High
NFR2.2	The system should take no longer than 1 second to fetch a random pair of questions of a specific difficulty to a pair of users in the same room.	Medium
NFR2.3	The system should display each incoming message between users in the same room with no noticeable latency.	High
NFR2.4	The system should take no longer than 1 second to save a coding session.	Medium
NFR2.5	The system should take no longer than 3 seconds to fetch a user's history.	Medium
NFR2.6	The code editor should remain synchronised between both participants even if both are typing at the same time up to 70 words per minute (wpm).	High

NFR2.1 (High): The security measures imposed on of a user e.g., Authentication and authorization, within the system should take no longer than 1 second.

We set this NFR with high priority as such a process to the server and returning to the front end should not take too long. We are prioritising Performance too alongside with our most important quality attribute, Usability.

We will back this up during the final presentation with a demo. However, let's look at the network tab of Chrome v107 and Postman.

On Development Environment

94.19ms on Chrome v107

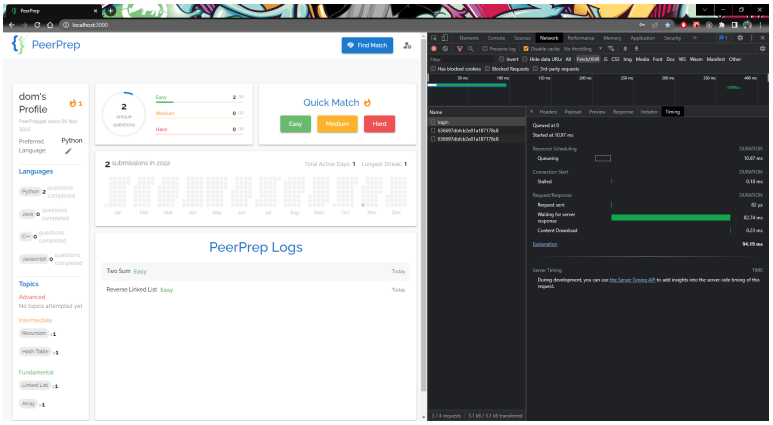


Figure 7: A screenshot of the time taken to login (Chrome)

77.6ms on Postman

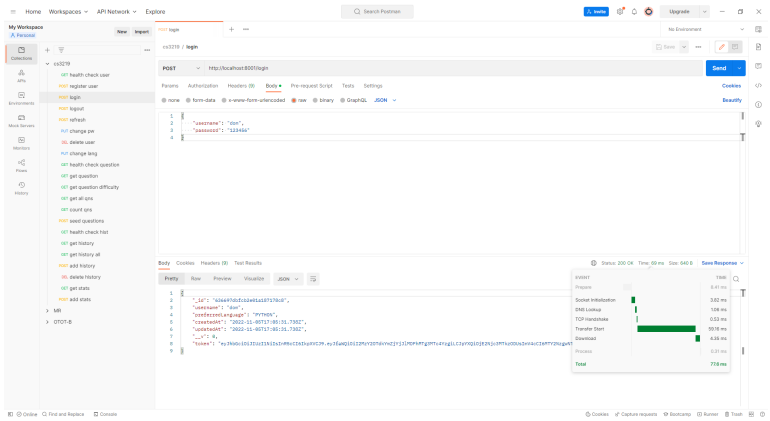


Figure 8: A screenshot of the time taken to login (Postman)

On Production Environment

175.09ms on Chrome v107

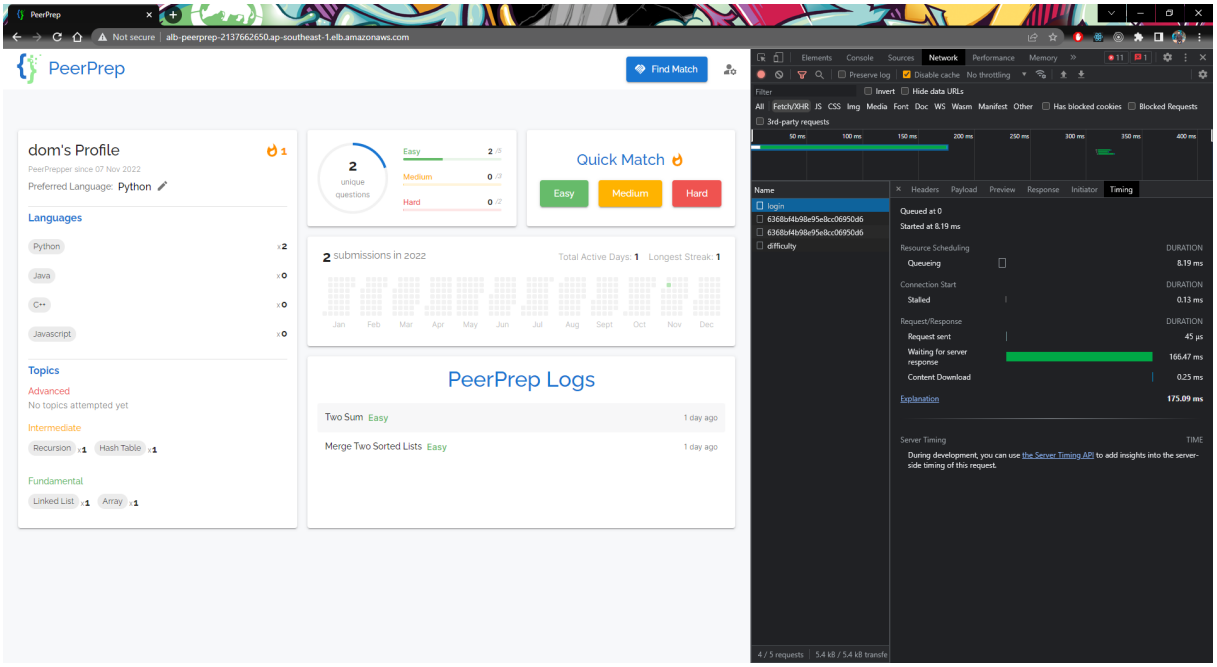


Figure 9: A screenshot of the time taken to login on prod (Chrome)

165ms on Postman

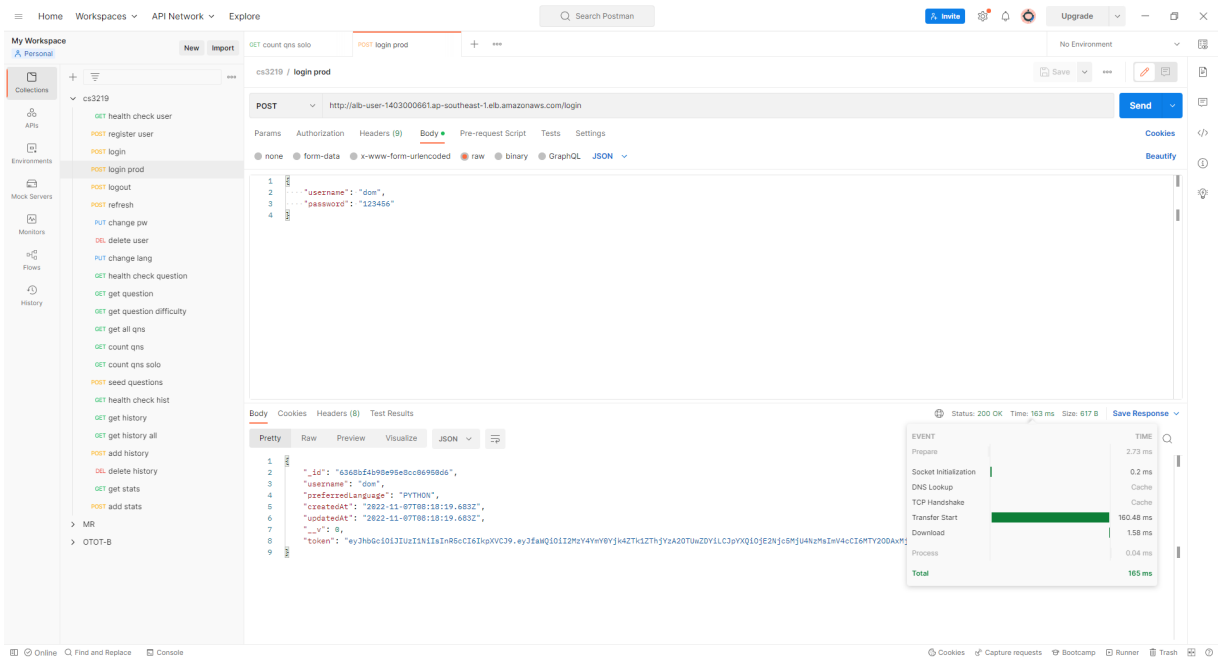


Figure 10: A screenshot of the time taken to login on prod (Postman)

NFR2.2 (Medium): The system should take no longer than 1 second to fetch a random pair of questions of a specific difficulty to a pair of users in the same room.

We chose a medium priority for this NFR as we thought that it is not as important as other Performance related NFRs for a high priority.

Let's look at the network tab of Chrome v107 and Postman.

On Development Environment

Chrome v107: Will be presented as we allow SocketIO server to handle the fetching.

22.34ms on Postman: Getting a set of Medium difficulty questions.

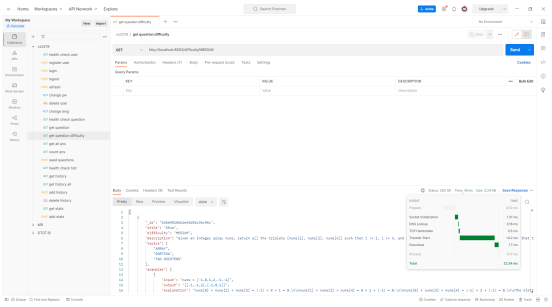


Figure 11: A screenshot of the time taken to fetch a pair of medium questions (Postman)

On Production Environment

Chrome v107: Will be presented as we allow SocketIO server to handle the fetching.

53.34ms on Postman

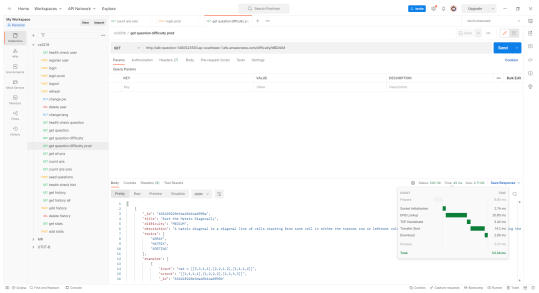


Figure 12: A screenshot of the time taken to fetch a pair of medium questions on prod (Postman)

NFR2.3 (High): The system should display each incoming message between users in the same room with no noticeable latency.

We have selected high priority for this NFR as messaging is the only form of communication we have added and we need to ensure we provide the highest performance for this.

Each message sent ought to be instantaneously and simultaneously displayed and received by sender and receiver respectively. To implement this, instead of doing it through a RESTful API, similar to NFR 1.2 in Usability, we make use of the Publisher-Subscriber messaging pattern with the help of SocketIO. This pattern helps us create low latency between both users or clients for this matter as they both listen to the same topic (room ID for this context) on the SocketIO server. As soon as a message is published at either end, it will be received by other clients subscribed or connected to this socket.

NFR2.4 (Medium): The system should take no longer than 1 second to save a coding session.

This NFR has a medium priority as saving the state of a coding session between a pair of users can take awhile with the amount of data. It is also dependent on the performance of the server to some extent.

Similar to NFR 2.2, this action is handled by the SocketIO server.

On Development Environment

Chrome v107: Will be presented as we allow SocketIO server to handle saving.

55.13s on Postman

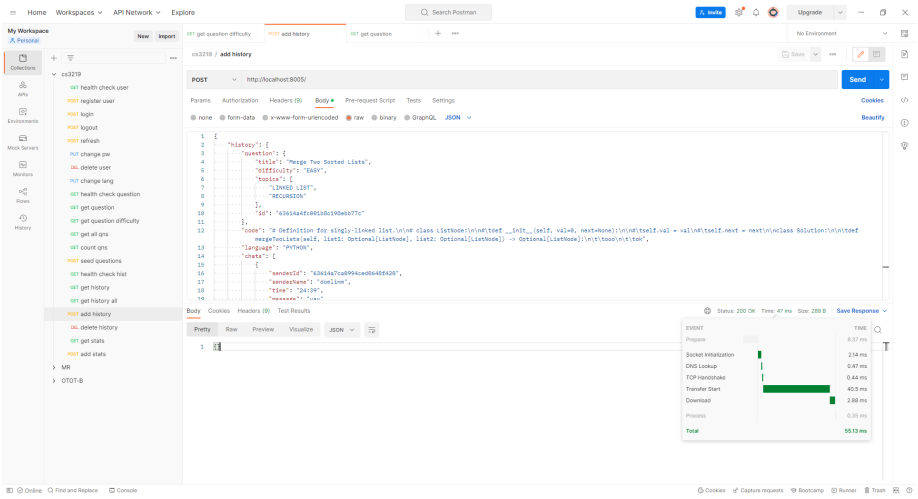


Figure 13: A screenshot of the time taken to save state of a session (Postman)

On Production Environment

Chrome v107: Will be presented as we allow SocketIO server to handle saving.

55.64ms on Postman

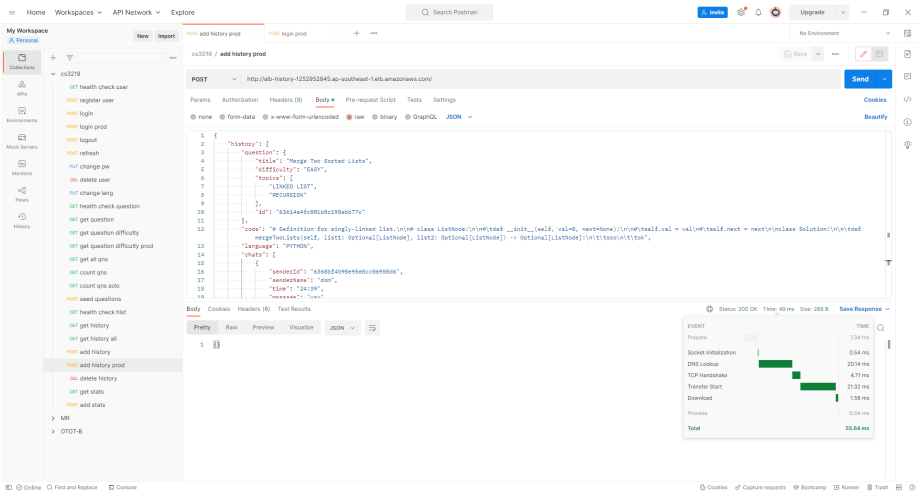


Figure 14: A screenshot of the time taken to save state of a session on prod (Postman)

NFR2.5 (Medium): The system should take no longer than 3 seconds to fetch a user's history.

NFR2.5 has a priority of medium as the response depends on various factors. Examples, how much history data a user has, the performance of the server to some extent.

On Development Environment

12.45ms on Chrome v107: For userId 636697dbfcb2e01a187178c8

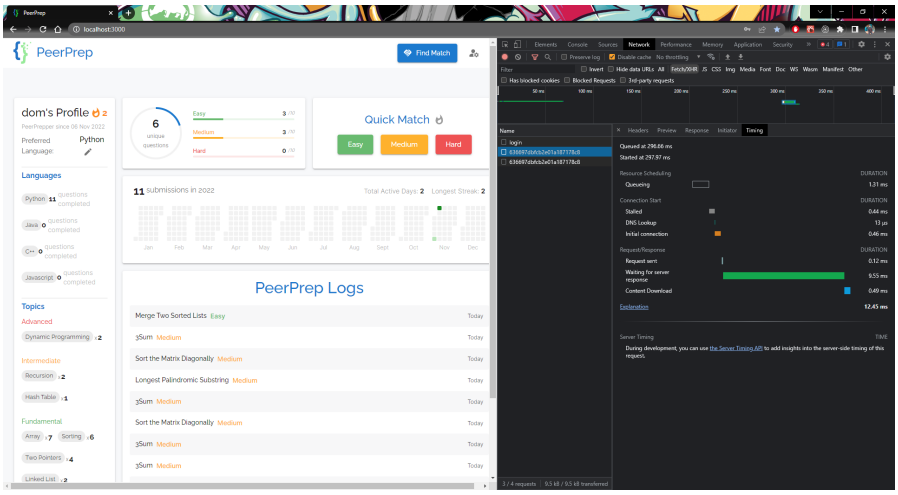


Figure 15: A screenshot of the time taken to fetch history (Chrome)

21.22ms on Postman: With the same user

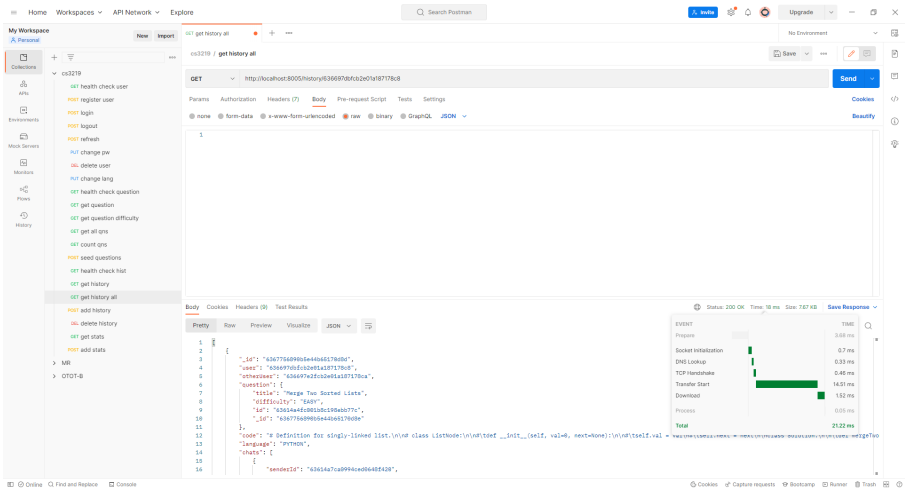


Figure 16: A screenshot of the time taken to fetch history (Postman)

On Production Environment

13.64ms on Chrome v107

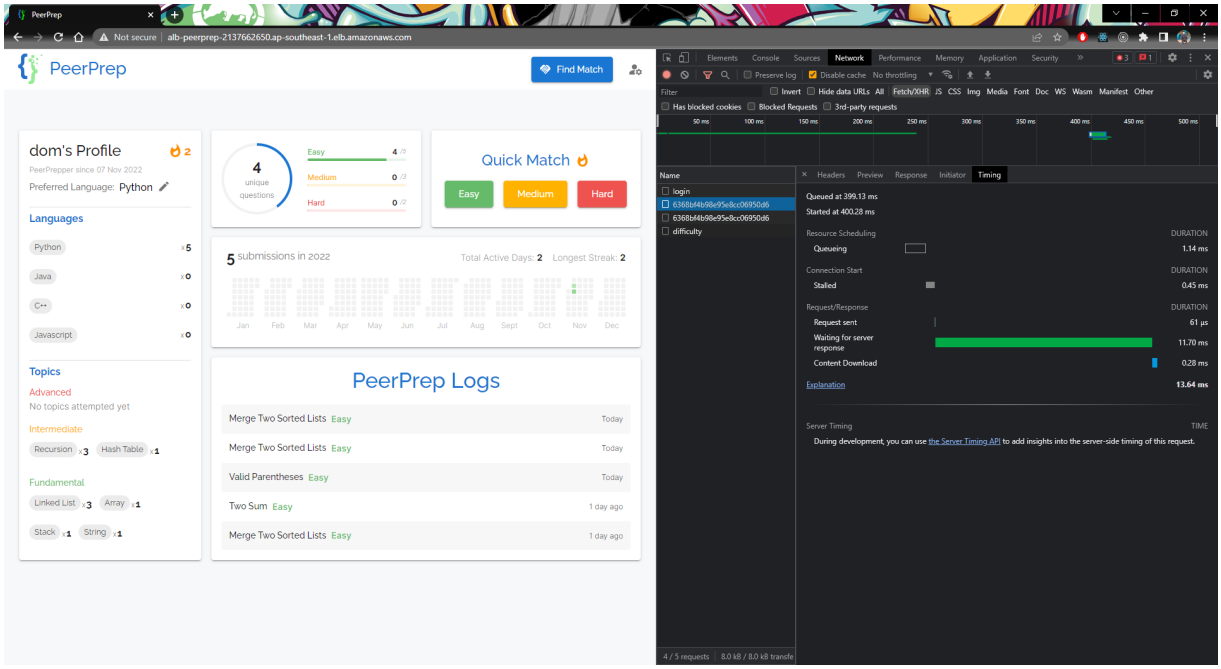


Figure 17: A screenshot of the time taken to fetch history on prod (Chrome)

19.7ms on Postman

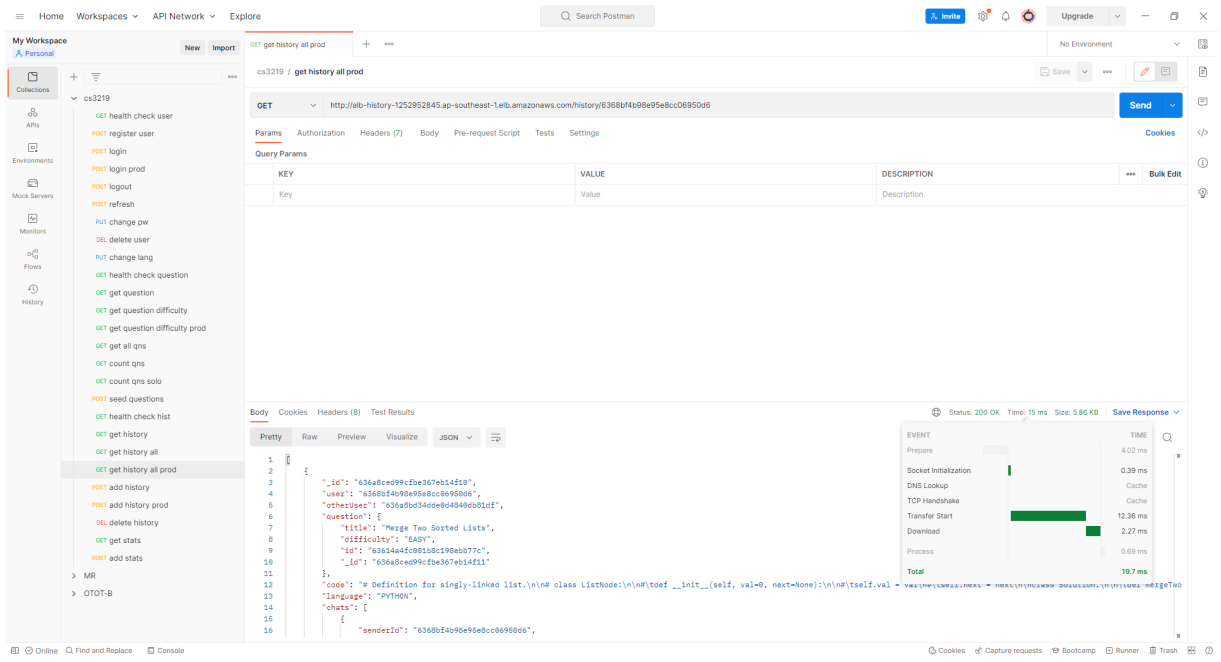


Figure 18: A screenshot of the time taken to fetch history on prod (Postman)

NFR2.6 (High): The code editor should remain synchronised between both participants even if both are typing at the same time up to 70 words per minute (wpm).

We set a high priority on this as we feel that this will serve as a baseline to the performance of the system's code editor. Also, the main perspective of our system is to provide users with a great environment to conduct their mock interviews. Similar to NFRs implemented with SocketIO server, this will be substantiated through a demo.

Reliability

Our website also prioritises reliability to ensure that users are given a reliable experience with the functionalities of the system and provides the user with the correct resources consistently and to ensure that any form of real time activity is synchronised between a pair of users. Aside from real time, we look to guarantee users that the system will not fail on them at any point in time as far as possible.

S/N	Requirement	Priority
NFR3.1	The system should not miss any messages that were sent at the same time.	High
NFR3.2	The chat logs between two participants should always be synchronised.	Low
NFR3.3	The system should ensure that the session timer should be synchronised between both participants at all times.	High
NFR3.4	The system should ensure that users stay logged in even if they refresh the browser.	High

NFR3.1 (High): The system should not miss any messages that were sent at the same time.

We set a high priority on this as messaging is the system's only form of communication. Hence, it deserves to be placed at the top of the priority list.

Note: This NFR solely means that the system should capture the messages sent but in no means imply that the intended recipient should receive the sender's message instantly.

To ensure we provide reliable communication between a pair of users, we utilise the Publisher-Subscriber messaging pattern (similar to other NFRs) through a SocketIO server where each message sent by a user to sent to this server and verified. As all users, or for this case a pair of users, have subscribed to the same topic, the messages will be synchronised as a best-effort service by the SocketIO server to ensure the messages are not missed from the clients' perspective. This pattern substantiates reliability on this NFR. Please refer to our Messaging Service section for more an in depth outlook. A demo during the presentation will help further substantiate this NFR.

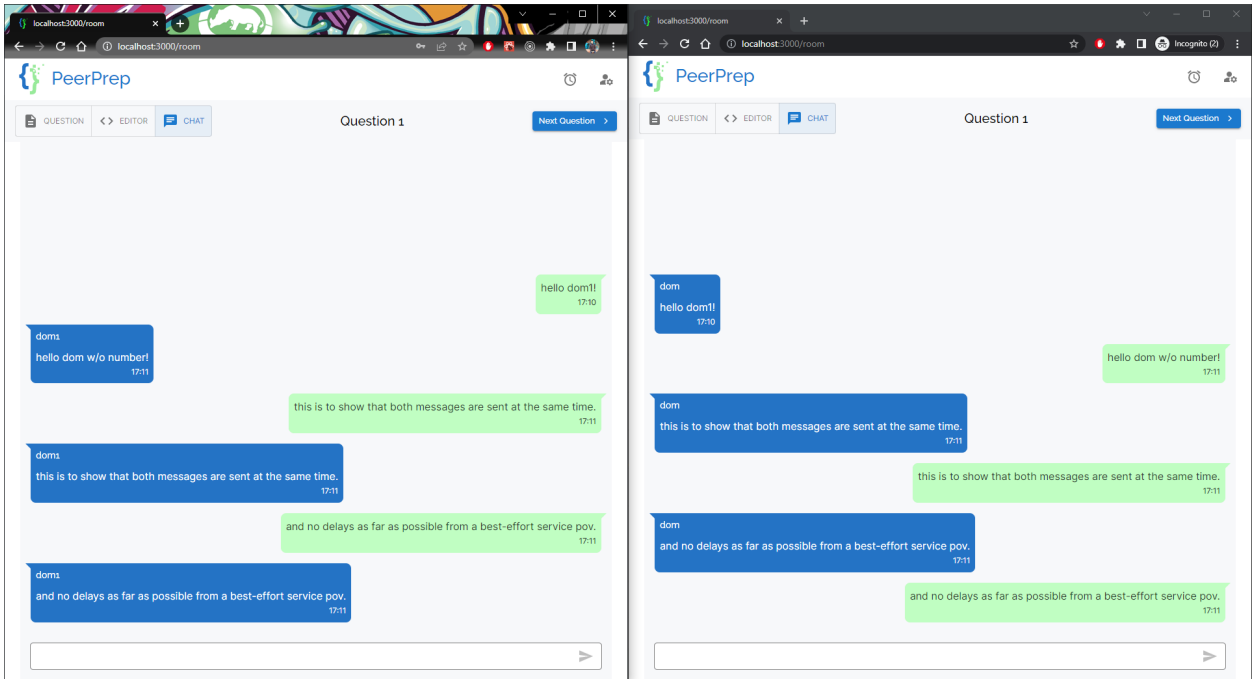


Figure 19: A pair of users and their current chat log

NFR3.2 (Low): The chat logs between two participants should always be synchronised.

This NFR is given a priority of low as this is a NFR that we have chosen to try and enforce. It is also based on the implementation of the system that guarantees the synchronisation. Also, comparing with NFR3.1, this is secondary.

The evidence to uphold such a NFR can be viewed from the same perspective as NFR3.1. The Publisher-Subscriber messaging pattern helps to ensure synchronisation on a best-effort delivery basis between a pair of users reliably.

NFR3.3 (High): The system should ensure that the session timer should be synchronised between both participants at all times.

This NFR has a high priority as it is important to the main purpose of the system to ensure a great environment for mock interviews. Also, similar to NFR2.6 and synchronisation related ones.

We implemented this by making use of SocketIO server. The timer value is entirely handled by the server i.e., Updating the value, sending this value to the pair of users, etc. To further ensure it is synchronised, whenever a user presses on either the pause or stop button, a state, timerLoading, captures this action and disables the buttons on the timer to prevent any race conditions. These minute details of implementing the timer help to ensure that the session timer should be synchronised between the pair of users at any given point of time on a best-effort basis by the server reliably.

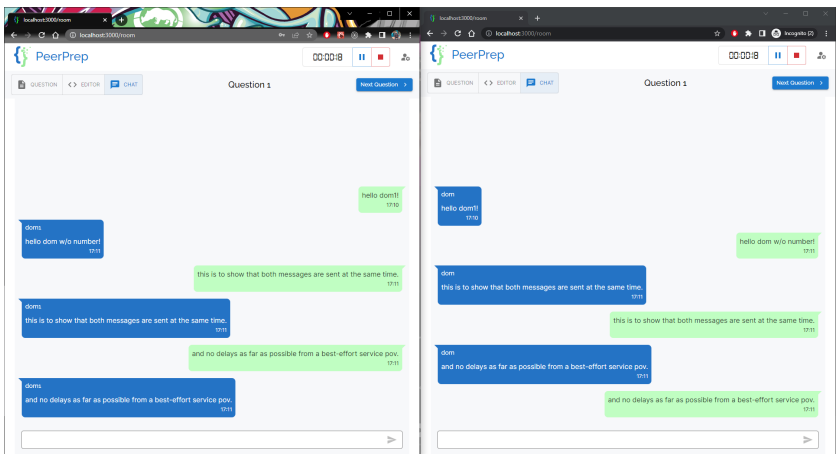


Figure 20: Image to show timer on both users (top right)

NFR3.4 (High): The system should ensure that users stay logged in even if they refresh the browser.

We set this NFR as high as we want to ensure that the system does not break due to authentication or authorisation after a user has successfully logged in.

To substantiate this, we have implemented the authentication system using a JSON Web Token (JWT) stored in the local storage on the user's device and this JWT persists till it expires. On the front end, we make use of React's hook for context and this context handles the authentication related functionalities e.g., Login, Register, Logout. The context aids in the requests triggered by the user's actions on the system to inform the server that this user is authenticated on every request. Additionally, since the JWT is stored on the local storage, the user will only be logged out when the token expires i.e., Relogging in the next time will automatically authenticate the user if the token has not expired. Hence, the system provides reliability to when a user is authenticated and he/she should not be logged out even when the page is refreshed, or even so coming back again in the future.

Features

Feature Domain	
Feature	**Description**
Matching Service	
Difficulty Selection	• Users have the option to choose from 3 levels of difficulty of - Easy, Medium and Hard. • Upon selecting a level of difficulty the system attempts to match them with another user who also selected the same difficulty level and has the same preferred programming language.
Countdown Timer	• During the matching process a 30 second countdown timer is initiated which is displayed to the user. • If a match is not found for the user within this time period they would be removed from the waiting room after informing the user of the unsuccessful matching attempt. • If a valid match is found for the user - they would be informed of the successful match and would proceed to the code editor with their matched user. • Users can cancel the matchmaking process at any point while they are in queue.
Collaboration Service	
Questions	• Two question of the chosen difficulty are selected and displayed to the two users one at a time. They can navigate to the next question when they are done with the first, and both parties must agree to proceed to the next question.

	• When a user attempts to navigate to the next question - a prompt is displayed to the other user informing them that their partner would like to change the question and the navigation only occurs after the user agrees to the prompt. • Having two questions allows the two users to alternate the roles of interviewer and interviewee between the two questions • The questions are displayed with the difficulty, question description and basic example of input and expected outputs. • The question window can be resized or hidden as required by the users
Code Editor	• The code editor is shared between the two users • Any code entered into the editor by one user would be reflected and visible to the other user with negligible latency. • Any code in the editor that is selected by one user should be highlighted to the other user to indicate what one user may be pointing or referring to. • The code for one question would be saved if the users were to navigate to the next question. • The code editor window can be resized or hidden as required by the users
Timer	• A session timer is present to mimic a real interview environment. • The timer is synced between the users and either of the users can start, pause, resume and stop the timer as required.

	• If the timer is still active when the users proceed to the next question, it will automatically be stopped.
Messaging Service	
Chat	• Both users can communicate with each other via the messaging functionality which is located below the code editor. • The messages in chat are distinguished as sent messages and received message by the use of colour and positioning of the chat bubble • The messages and sent and received almost instantaneously. • The chat window can be resized or hidden as required by the users.
History Service	
Save Coding Session	• All the information from a given collaborative coding session is saved. This includes the question, specified difficulty, the code as well as the chat logs. • Users can view past coding sessions through their profile at any time. • The coding session is not auto-saved intermittently but only when proceeding to the next question or at the end of the session upon confirmation, this is done as it is not critical to save logs intermittently and is beneficial for the system performance.

Sprint Planning

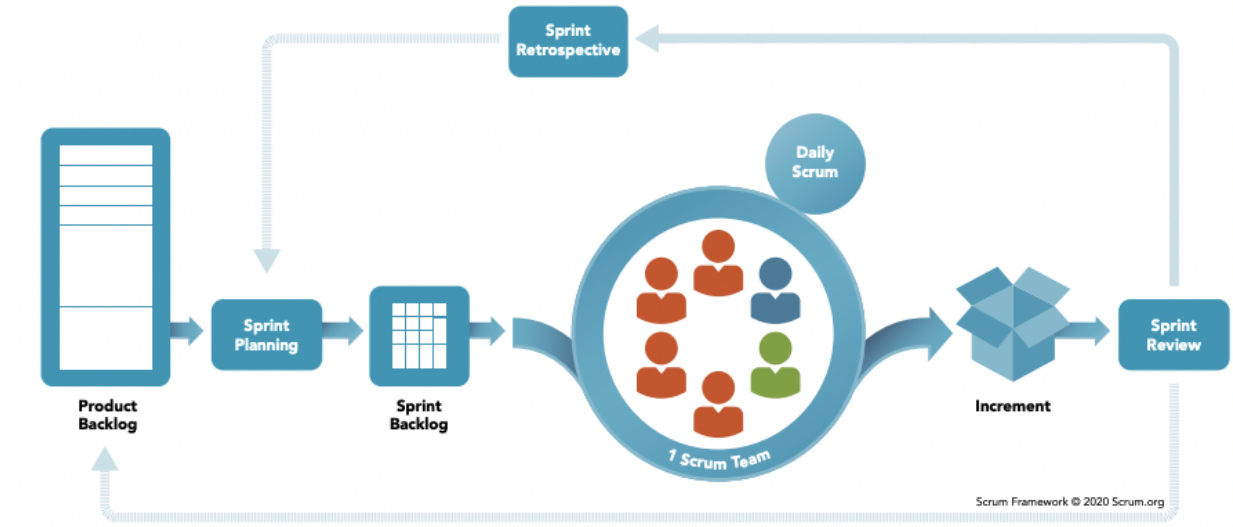


Figure 21: Scrum Framework

For this project, our group adopted the agile principles and practiced the Scrum method in particular (a typical diagram of Scrum is shown above for reference).

At the very beginning of the project, we first pinned down all the functional requirements of each service or component, followed by creating the product backlog categorized by which service or component it belongs to.

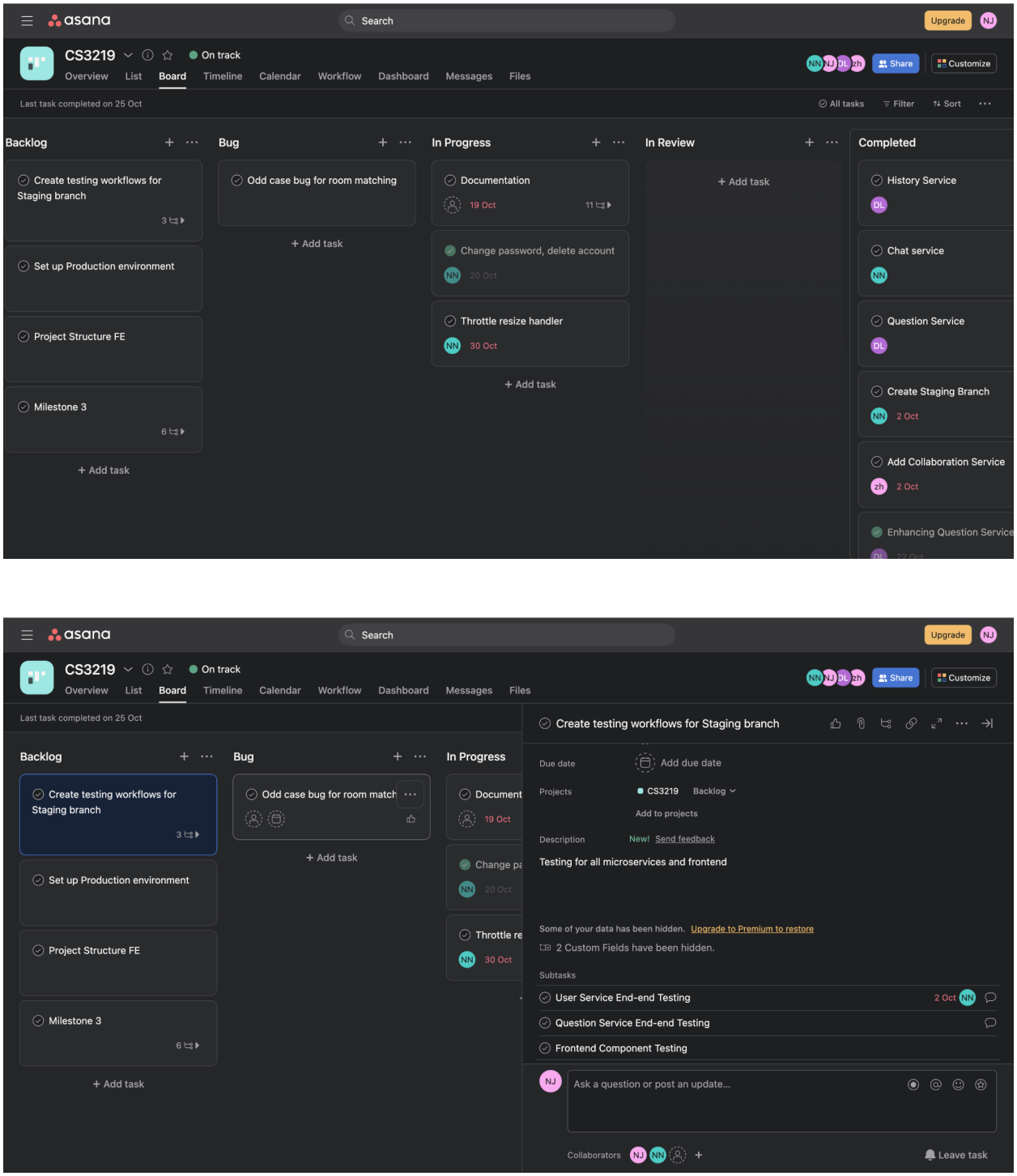


Figure 22: Kanban Board

The product management tool that we chose to use is Asana's project development kanban board, which is free and easy to use. We carried out weekly sprints instead of sprints lasting 2 or more weeks because it allows us to assess our progress and make

changes more frequently. Every Friday, we conducted a sprint review for the previous sprint and also the sprint planning for the upcoming sprint.

The 4-person team is also further split into teams of two with each pair being responsible for developing a particular component of the application. The purpose of splitting into pairs is twofold. Firstly, since we are writing much of the code from scratch, the smaller the number of people in a team, the better it would be for development in terms of speed and coordination. Secondly, this allows us to do pair programming which is proven to reduce bugs and mistakes, and improve code quality.

The 2 person teams also allowed us to carry out pull request reviews in a more systematic manner. As pull requests to the staging branch would be reviewed by another person within the two person team and pull requests made to the production branch would be reviewed by someone from the other team.

The splitting and allocation of major components are as follow:

Pair	**Component**
Zhiwei & Dominic	• Collaboration Service (real-time code editing) • Question Service • Matching Service • History Service
Vijay & Nicole	• User Service • Messaging Service • Collaboration Service (timers & synchronizing questions) • Frontend

However, this allocation is not completely strict. Occasionally, a member would still work on tasks for components not assigned to them, to ease another pair's workload or to integrate a particular service with the front end for example.

Application Design

Tech Stack

Technology	Rationale
Next.js (Frontend)	• Highly performant thanks to React's implementation of vDOM • Not strongly opinionated, allows for freedom in development process • Page and component based architecture helps us to enforce organized project structure • Able to make use of hot reload even within docker development environment, which helps to speed up development
Express.js (All backend microservices)	• Extremely widely used framework for Node.js that is supported by many libraries • Low barrier to entry due to wide availability of resources and ease of setup

MongoDB with Mongoose (User and History microservices)	• Helps us to easily enforce MVC design pattern
Socket.io (Matching, collaboration and messaging microservices)	• Reliable, low-latency one-to-many communication between server and clients in real-time • Easy to use and set up
Docker (Deployment for development)	• Allows us to containerize our entire development environment • Able to spin up frontend, all backend microservices and databases with a single command • Docker image helps to standardize development environment (especially version of installed dependencies) across all developers regardless of OS
GitHub Issues (Project management)	• Quick and direct creation/closure of issues within GitHub itself • Allows us to create branches directly from issues, hence enforcing branch naming conventions by default

Asana (Project management)	• Easy to use kanban board tool that allows us to quickly split up the project workload • Allows us to set deadlines and priority levels for each task to remain on-task

Overall Architecture

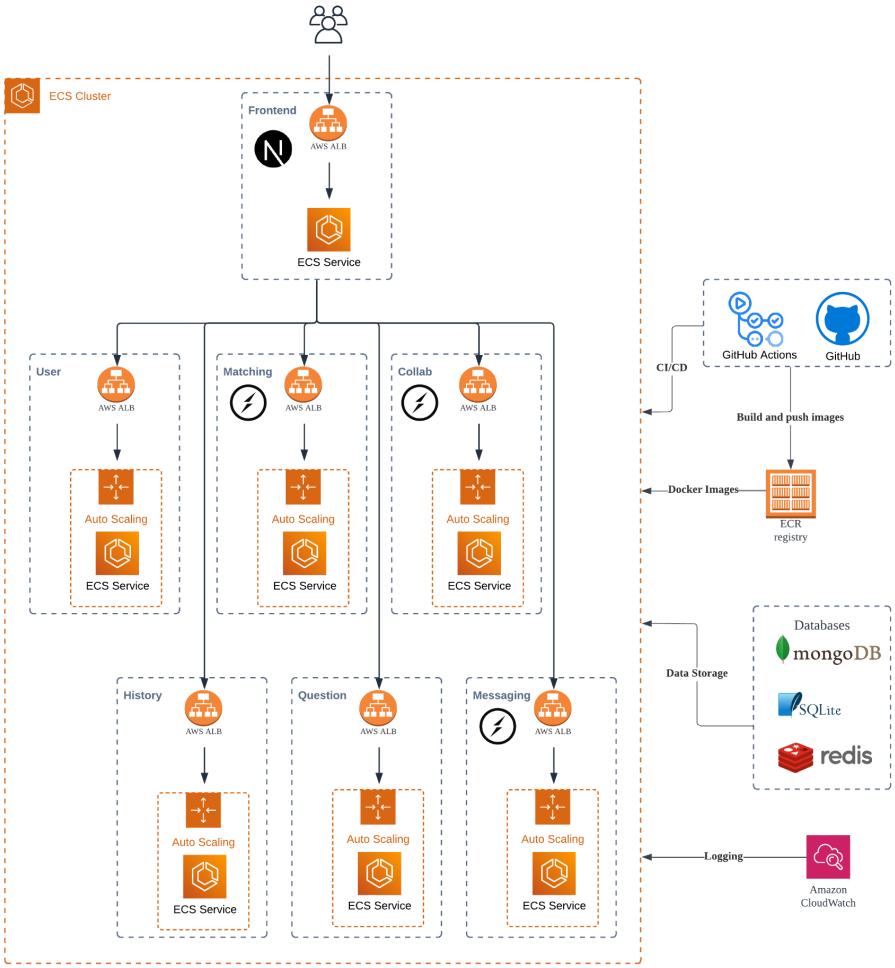


Figure 23: Overall System Architecture

In short, users interact with the frontend, while the frontend communicates with all the microservices. The architecture is designed in such a way that there is little to no dependency between each microservice due to a strict adherence to the separation of concerns principle through clearly defined bounded contexts.

The microservices that require databases each have their own database instances. For services where persistence is vital (e.g. user, history, question service), mongoDB database is used because its NoSQL cloud database is robust, secure, and highly available. The remaining services use either SQLite if they require relatively more complex data queries (e.g. Matching service), or redis otherwise (e.g. collaboration service).

The strength of this design, as mentioned previously, lies in the fact that there is a very low level of coupling between the numerous microservices. This means any changes in one service would cause little to no change in other services.

Each service (microservice or frontend) is deployed in a separate ECS service so they could be provisioned and scaled independent. Furthermore, each service sits behind a load balancer and is provisioned with auto-scaling capabilities. Details about deployment in AWS is written under devops.

User Activity Flow

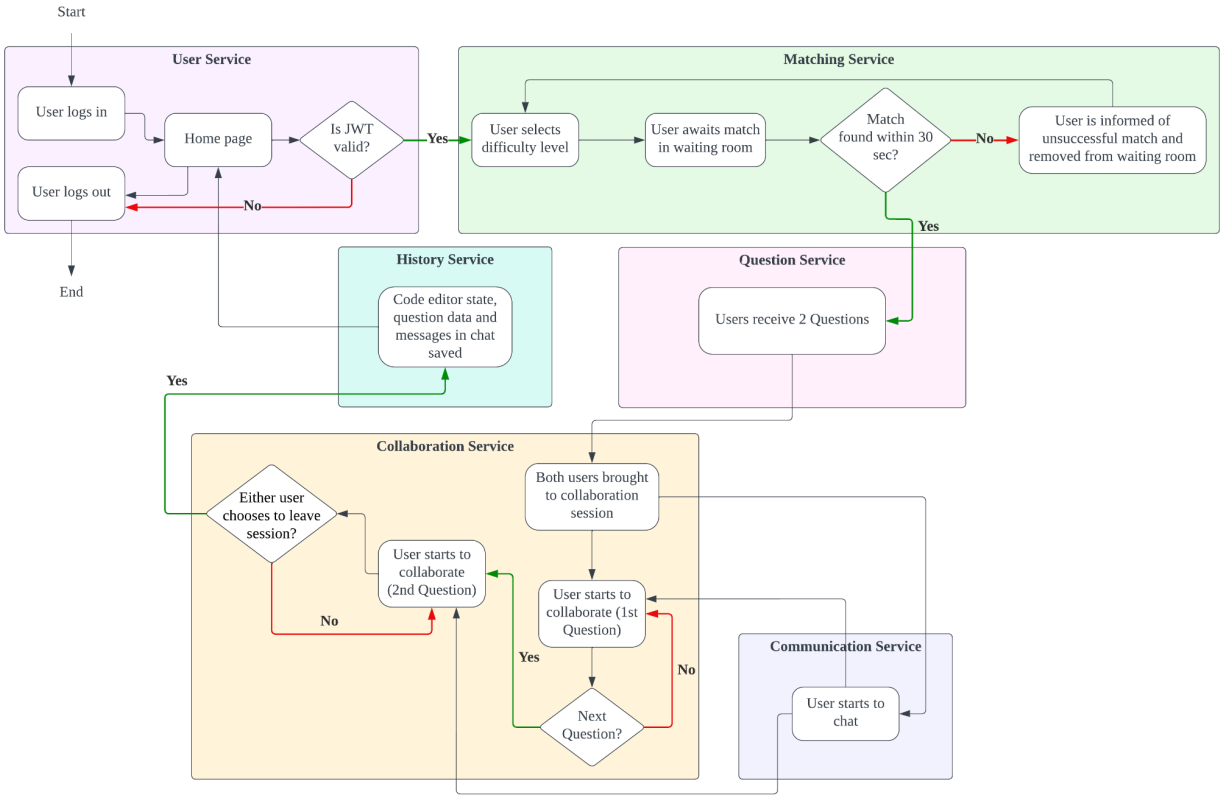


Figure 24: User Activity Flow

Frontend Architecture

Component Tree

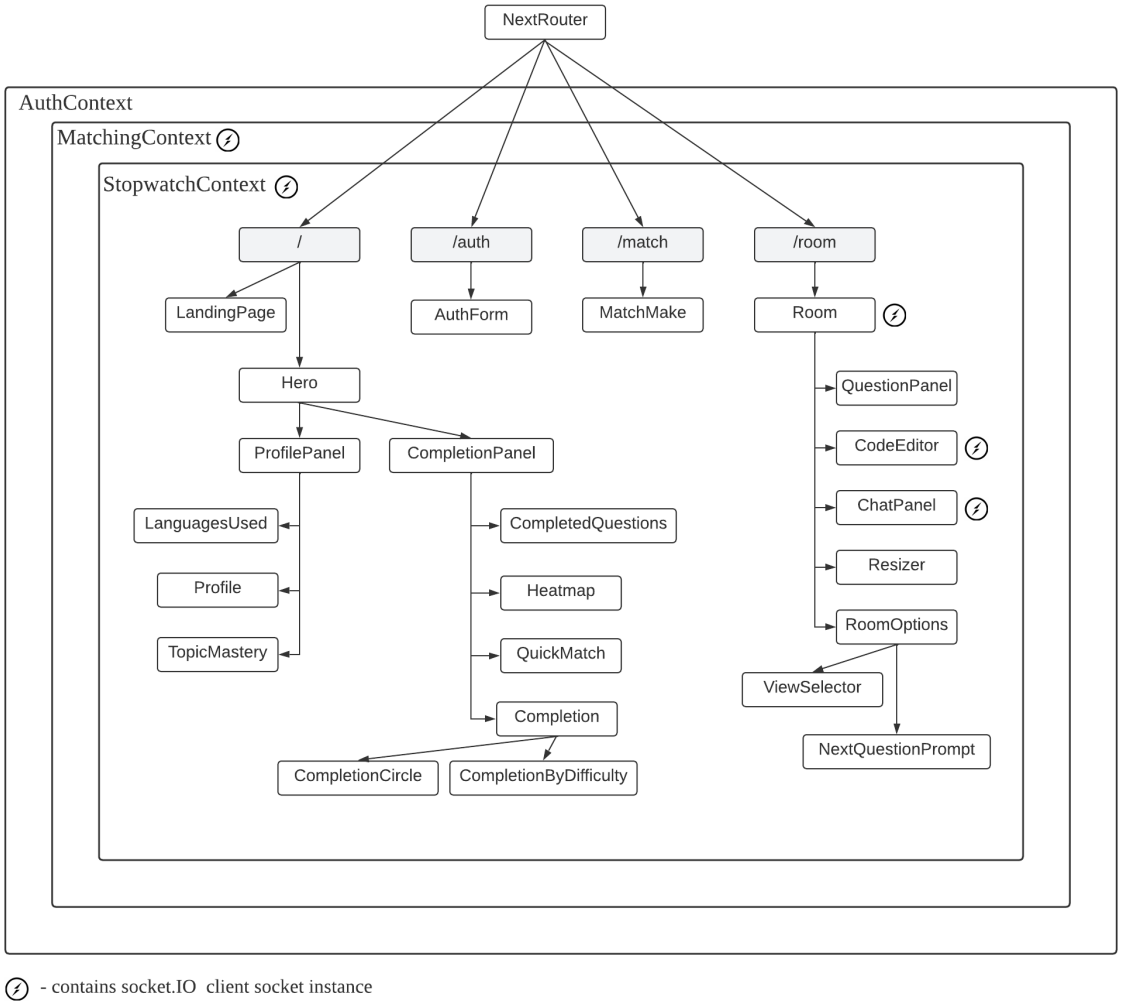


Figure 25: Frontend React Component Tree

The high level structure of our frontend, built with NextJs, is shown in the diagram above. There are a total of 4 pages, namely homepage, authentication page, matching page and the collaboration room page. A brief description of what each page does is presented below:

1. Homepage
    a. When the user is not logged in, they see a landing page that invites them to register for a PeerPrep account or log in if they already have one.

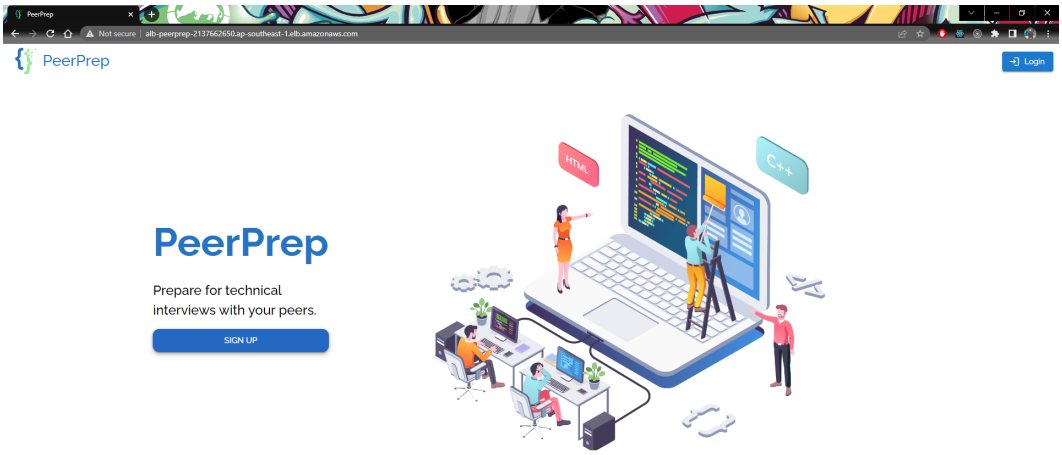


Figure 26.1: Landing

    b. When the user is logged in, they see an elaborate dashboard that details their PeerPrep statistics including daily streak, programming language usage, topic mastery and more.

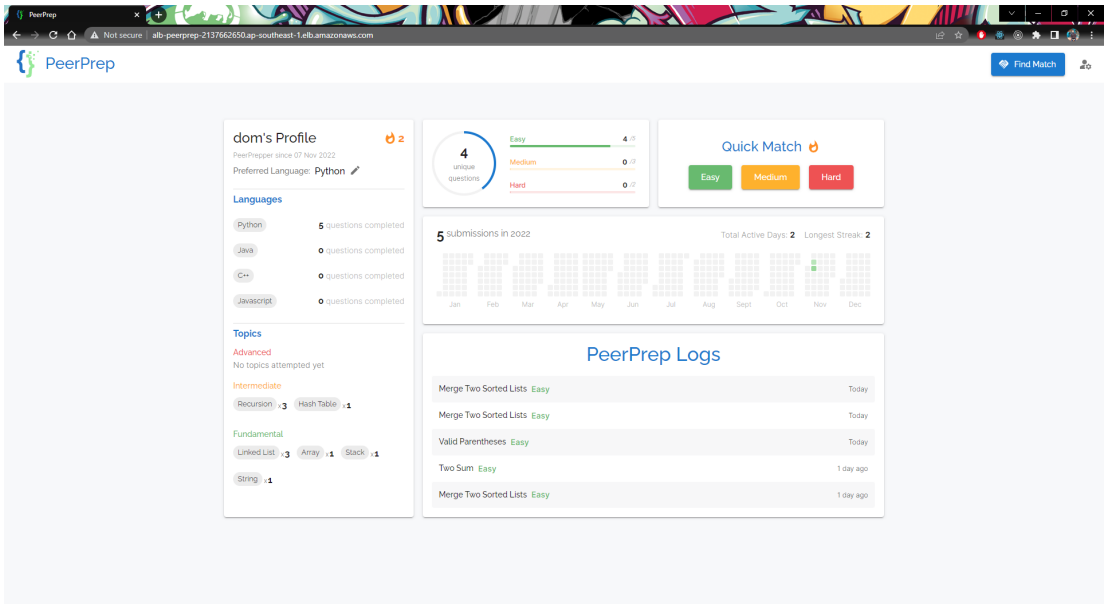


Figure 26.2: Dashboard

2. Authentication page
    a. When a user is not logged in, they see a form through which they can either register for a PeerPrep account or log in if they already have one.

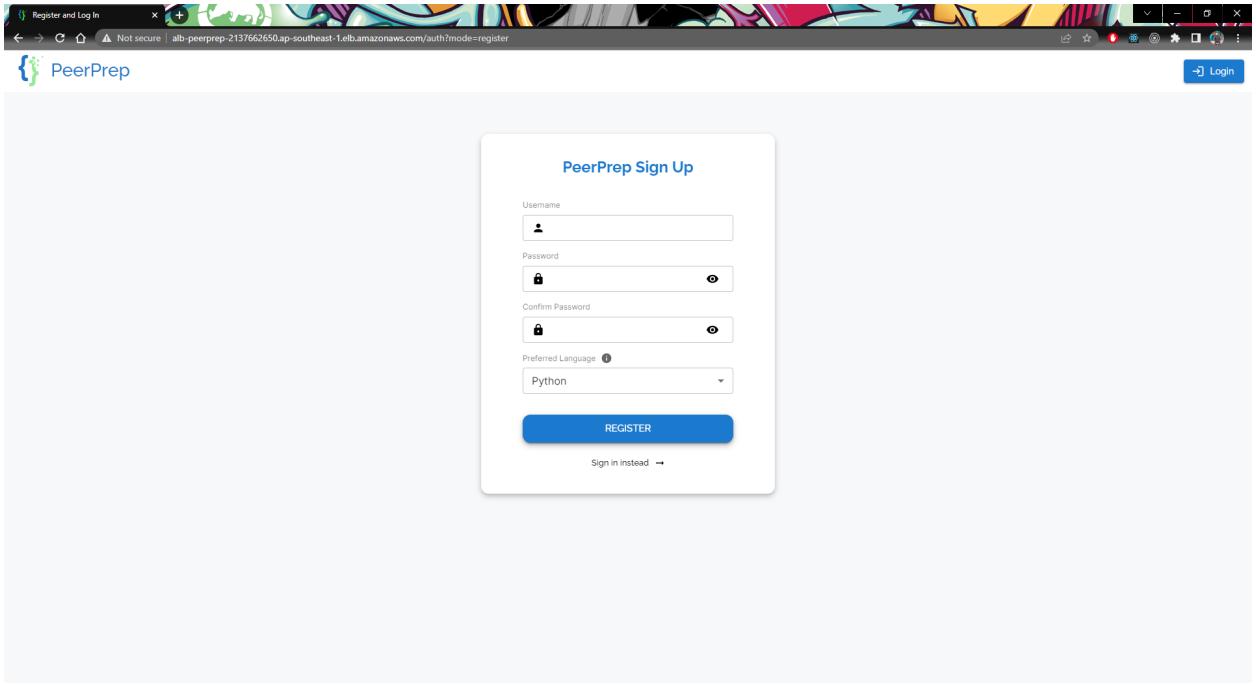


Figure 26.3: Sign Up

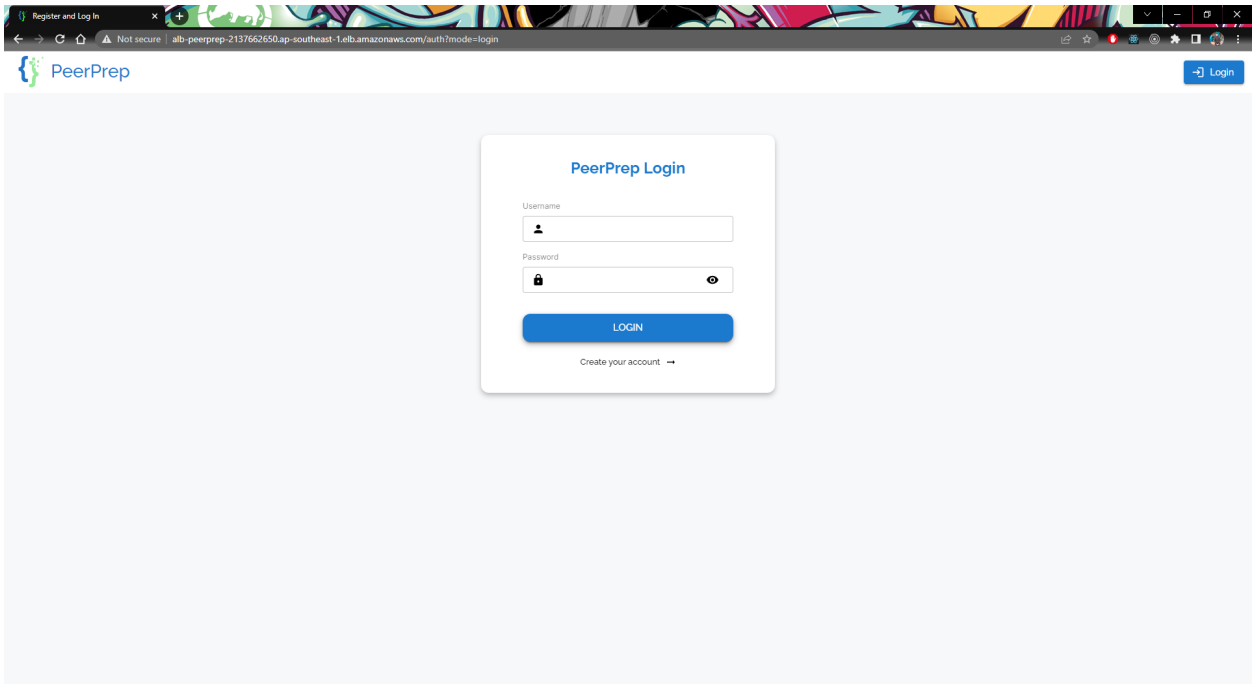


Figure 26.4: Log In

b. When a user is logged in, they see a different form through which they can either update their preferred programming language or change their account password.

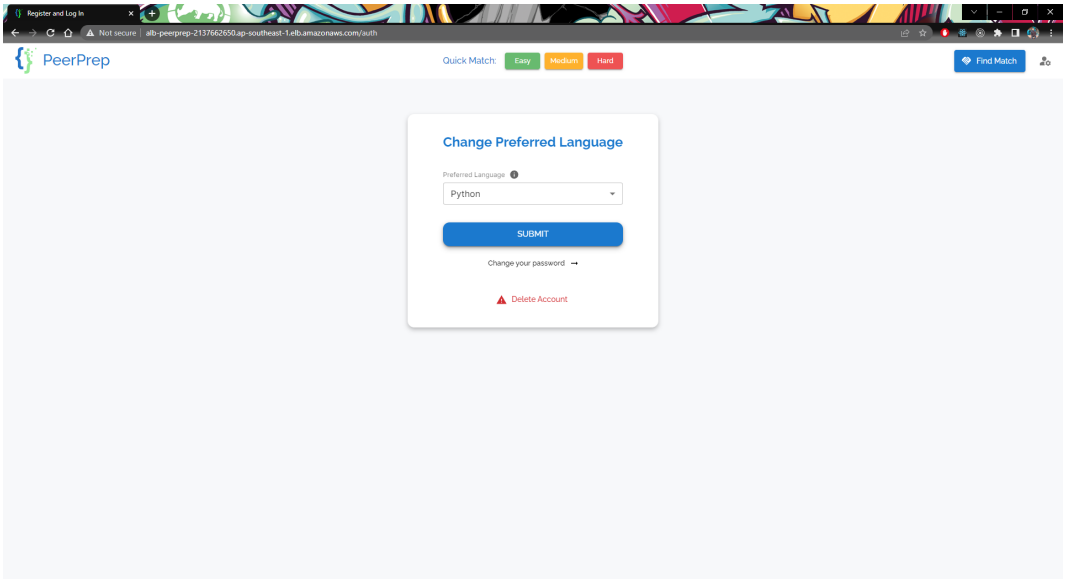


Figure 26.5: Change Preferred Language form

3. Matching page
    a. This page prompts logged-in users to select a difficulty for matchmaking. It also lays out all the potential topics that questions of each difficulty may consist of.

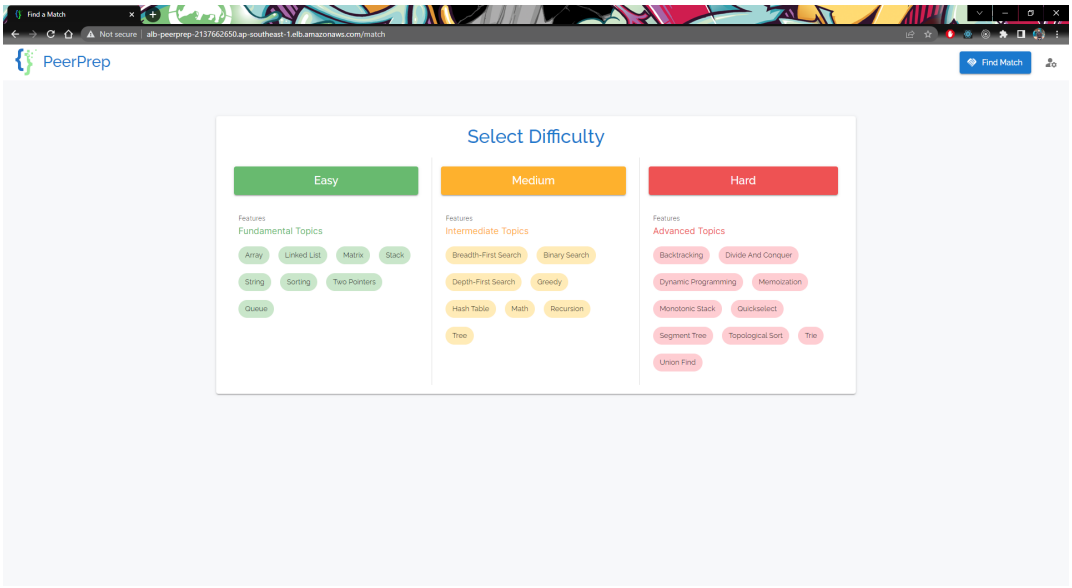


Figure 26.6: Matching

4. Room Page
    a. When two users are matched by our matchmaking system, both users are redirected to this collaboration room page where they see a code editor, a chat panel as well as a pair of randomly chosen questions of their selected difficulty. Users can move on to the next question after completing the first, and every panel in this page can be resized as well as hidden or brought in to focus as the user pleases.

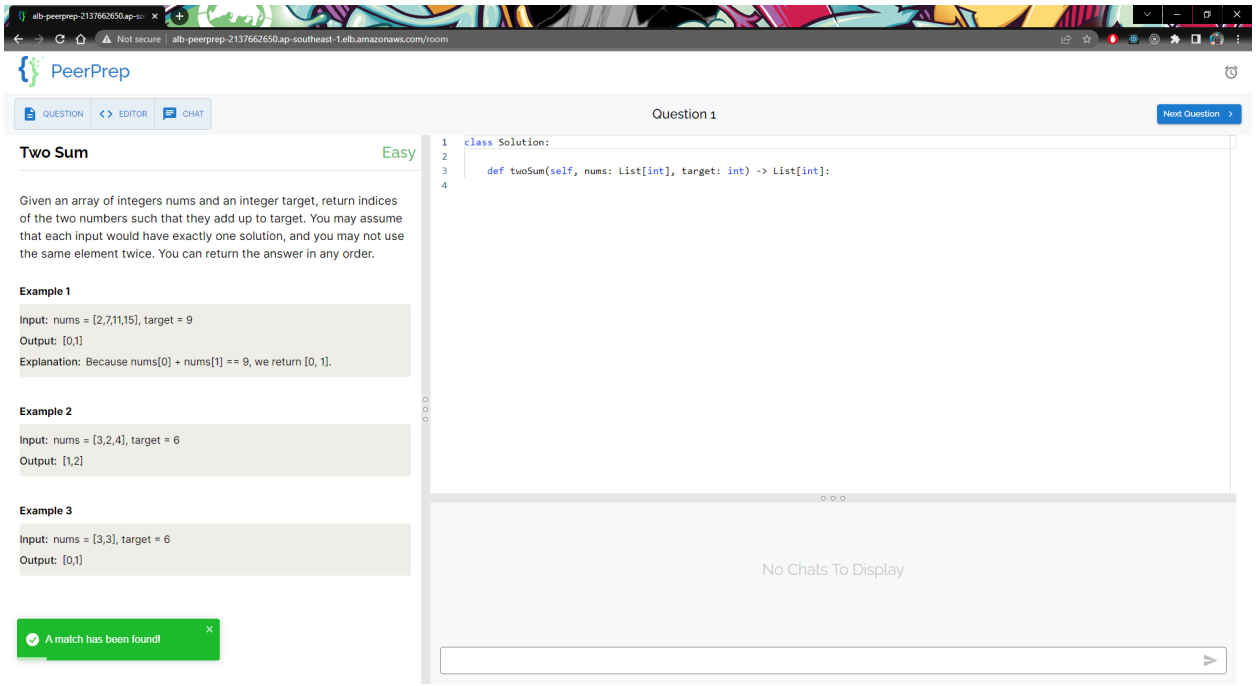


Figure 26.7: Room with Code Editor, Question, Chat sections

Design Pattern: Context-Provider

React provides *Context* object which is designed for sharing global data across components. Every *Context* object is paired with a *Provider* component through which consuming components can subscribe to changes in said *Context*. As such, it can be seen that making use of *Context* and *Provider* is a way for us to implement the **Singleton** GOF pattern. In fact, not only do these *Context-Provider* pairs provide global data, we also make use of them as **Facades** for frontend components to interact with the backend without being exposed to any complexity.

In our project, we have three *Context-Provider* pairs.

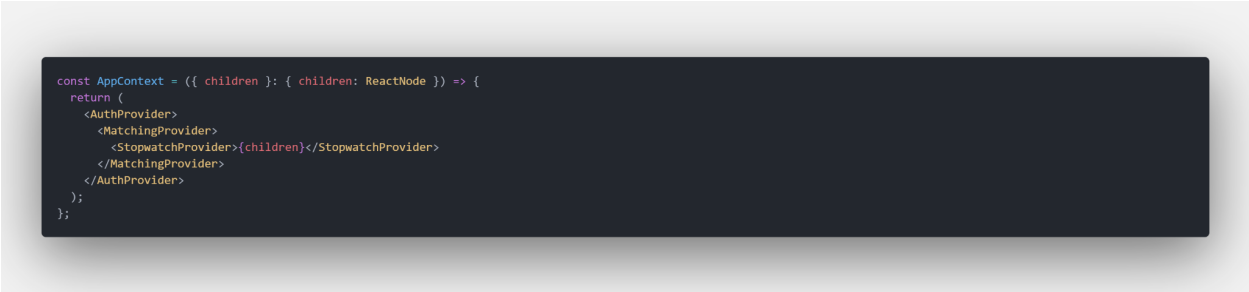


Figure 27: A code snippet of AppContext

First, we have a single instance of *AuthContext which* contains information about the currently logged in user (if there is one). It is paired with a single instance of *AuthProvider* that wraps around our frontend application. Many of our frontend components conditionally render different content depending on whether or not the user is logged in. *AuthProvider* is the single entrypoint for these components to access user information before deciding what content to render. It also provides functions such as *register()* which our registration form on the frontend can make use of to interact with the user microservice on the backend without needing to know any details about it.

```
interface IAuthContext {
  user: User | undefined;
  isLoading: boolean;
  refreshToken: (onSuccess?: () => void) => Promise<void>;
  register: (
    username: string,
    password: string,
    preferredLanguage: LANGUAGE,
    onSuccess: () => void
  ) => Promise<void>;
  login: (username: string, password: string) => Promise<void>;
  logout: () => Promise<void>;
  changePassword: (
    currentPassword: string,
    newPassword: string,
    onSuccess: () => void
  ) => Promise<void>;
  changePreferredLanguage: (
    preferredLanguage: LANGUAGE,
    onSuccess: () => void
  ) => Promise<void>;
  deleteAccount: (onSuccess: () => void) => Promise<void>;
}
```

Figure 28: A code snippet of IAuthContext (I represents Interface)

Next, we have a single instance of *MatchingContext* which contains information about the matching state of the currently logged in user (ie; whether the current user is currently looking for a match or is currently matched, etc). It is paired with a single instance of *MatchingProvider* that exists one level below *AuthProvider*. Some of our frontend components behave differently depending on the matching status of the currently logged in user. For example, the matching queue only appears when the user is looking for a match, and users are not allowed to edit their profiles while looking for a match as well. *MatchingProvider* is the single entrypoint for these components to access matching information before deciding what behaviour to exhibit. It also provided functions such as *startMatch()* for our matchmaking components on the frontend to queue the user up without directly interacting with the matching microservice on the backend.

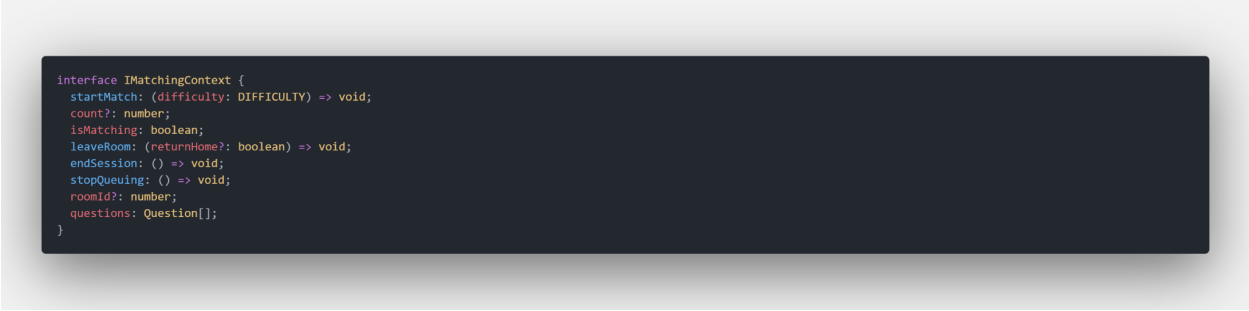


Figure 29: A code snippet of IMatchingContext (I represents Interface)

Finally, we have a single instance of *StopwatchContext* which provides information about the stopwatch for the currently active coding session (if there is one). This refers to the status of the stopwatch such as the time that it is currently displaying as well as whether or not it is active, paused or loading. *StopwatchContext* is paired with a single instance of *StopwatchProvider* that exists one level below *MatchingProvider*. There are exactly two components in our frontend that require knowledge of and access to the state of the stopwatch: *Stopwatch* itself (clearly) as well as *NextQuestionPrompt*. *StopwatchProvider* is the single entrypoint that provides this information to both of the aforementioned components. It also provides functions such as *handleStop()* which, during an active coding session, allows *NextQuestionPrompt* to stop the stopwatch (if it is active) upon proceeding to the next question or ending the coding session to prevent race conditions and enhance user experience.

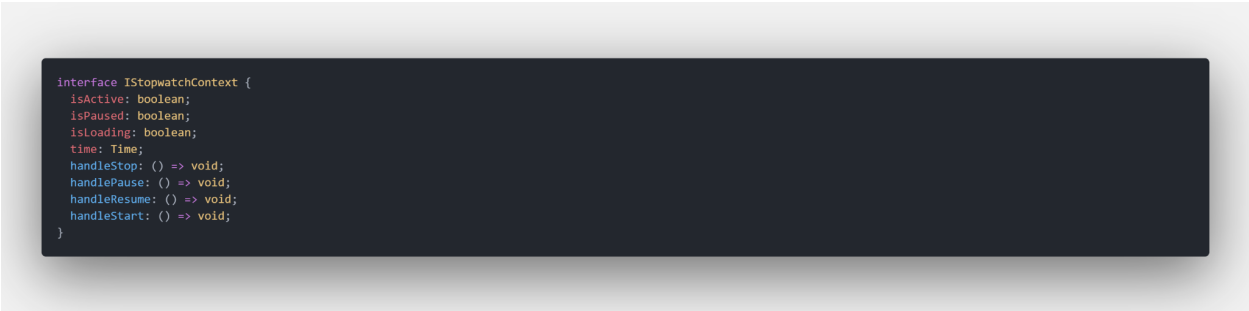


Figure 30: A code snippet of IStopwatchContext (I represents Interface)

Backend Architecture

RESTful API

Design Pattern: Model-View-Controller (MVC)

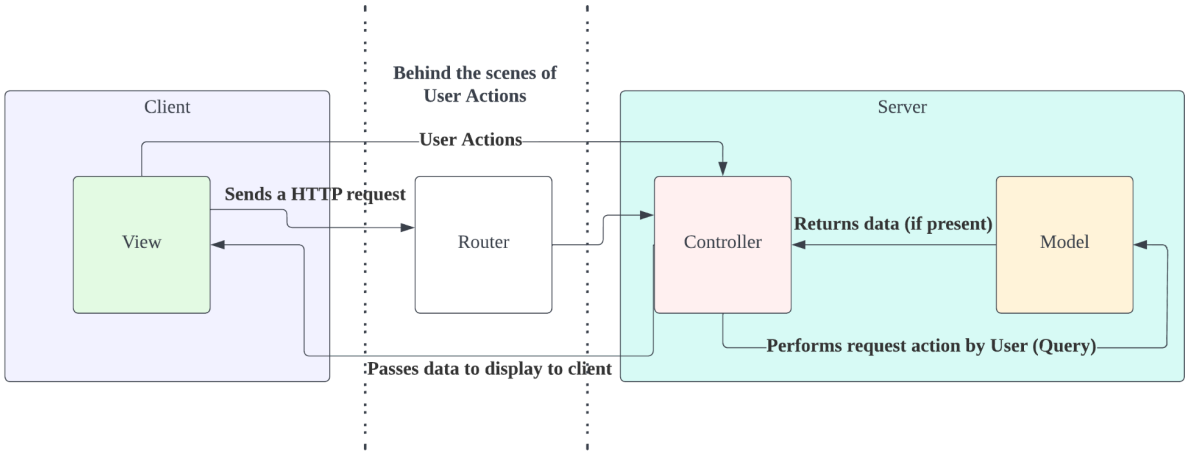


Figure 31: A drawing of our system's MVC pattern

Overview

We use a structure that follows the MVC pattern. In each of our microservice that serves as a RESTful API, it has the controller and model layers.

Controller

The controller serves as a connector between the router (contains the endpoint routes) and *model*. Its sole purpose is to read the data from either the request body or query parameter and passes it onto the *model* for processing. Hence, it makes use of the *model* and the API it provides for the business logic.

Model

Files in the model folder deal directly with mongoose's schema i.e., These files only have database related methods written. This way, model encapsulates the data and has methods for manipulating the state before sending it back to the controller. Additionally, it contains the type definitions for each schema.

View

The view is not found in the back end but rather in the front end where it's an independent, separate entity. It communicates with the router (as seen in Figure 30) to be able to execute any backend related functionalities. Firstly, the client performs an action on the front end by sending a HTTP request to the specified endpoint. The bridge, i.e., Router, pings the controller and invokes that client's action and execute the function that handles the requested transaction. Wherever necessary, the controller updates the model as per the client's action. The data, if any, is sent back from the model to the controller and back to the view to display to the client.

Benefits

Some benefits of adopting the MVC pattern include better modularity as the Separation of Concerns principle is strictly followed. Some examples are the output from a user's action is separated from handling the user's input, the output and user input are separated from the application's state and the transaction being processed at each time. Such a pattern facilitates extensibility where a new view and controller pair can be added for a new interface medium or even a new functionality can be easily added to the model independently of other components in this pattern.

Design Decision: Object-Oriented Programming

For all of our RESTful API microservices, we designed them in an OOP fashion. Let's take for instance in Java, we would define a class with its attributes, methods (both static and non-static, if applicable) and others necessary. In the model of each service, we first define the class with its attributes (in TypeScript's context, the interface with attributes). Following that, we declare the static and non-static methods of this service's class. These are written in the *.types.ts file.
Let's look at the User service for a clearer view:

```
export interface IUser {
  username: string;
  password: string;
  preferredLanguage: LANGUAGE;
}

export type UserDocument = Document<unknown, any, IUser> &
  IUser & {
    _id: Types.ObjectId;
  } & IUserMethods;

export interface IUserMethods {
  generateJwtToken(): string;
}

export interface IUserModel extends Model<IUser, {}, IUserMethods> {
  createUser(
    username: string,
    password: string,
    preferredLanguage: LANGUAGE
  ): Promise<void>;
  findVerifiedUser(username: string, password: string): Promise<UserDocument>;
  findUserByUsername(username: string): Promise<UserDocument>;
  findUserById(id: string): Promise<UserDocument>;
  updateUserPasswordById(
    id: string,
    currentPassword: string,
    newPassword: string
  ): Promise<void>;
  updateUserPreferredLanguageById(
    id: string,
    preferredLanguage: LANGUAGE
  ): Promise<void>;
  deleteUserById(id: string): Promise<void>;
}
```

Figure 32: A code snippet of user.type.ts

The static and non-static methods are further declared in *.model.ts file. A recap, the model files contain only mongoose related functionalities. Each of the static and non-static methods runs on the database functionalities to act on the requests by the user on the front end. However, not directly from front end to model, but through a HTTP request to the router, the controller - passes the transaction onto the model to interact with the database.

```
questionSchema.static(
  'findQuestionById',
  /**
   * Attempts to find a Question by ID
   *
   * @param id id of a Question
   */
  async function findQuestionById(id: string) {
    if (!id) throw new Error('Question id is required');

    const count = await getQuestionsCount();

    if (count === 0) throw new Error('No question found, seed questions');

    const question = await Question.findById(id).lean();

    if (!question) throw new Error(`No question with id ${id} found`);

    return formatQuestion(question);
  }
);

questionSchema.static(
  'findAllQuestions',
  /**
   * Attempts to find all questions
   */
  async function findAllQuestions() {
    const count = await getQuestionsCount();

    if (count === 0) throw new Error('No question found, seed questions');

    const questions = await Question.find({});

    return questions;
  }
);
```

Figure 33: A code snippet of question.model.ts showing static methods

```
userSchema.method(
  'generateJwtToken',
  /**
   * Generates a jwt token for a user
   *
   * @returns User's jwt token
   */
  function generateJwtToken() {
    const user: UserDocument = this;

    return jwt.sign({ _id: user._id }, String(process.env.JWT_SECRET), {
      expiresIn: '24h',
    });
  }
);

const User = mongoose.model<IUser, IUserModel>('User', userSchema);

export default User;
```

Figure 34: A code snippet of user.model.ts showing a non-static method

This design decision to structure our RESTful API microservices highly supports future extensions if we need to introduce more functionalities related to this service easily. It also helps us to keep the codebase clean, readable and understandable.

User Service

The user service maintains data about all users and provides authentication. When a user logs in, we generate a JWT token for that particular login session which lasts for 1 day. We also have a middleware for authentication which checks for a valid, non-expired JWT token in the ‘Authorization’ header of an incoming request to the user service. If the JWT token does not exist or is invalid, the request is rejected with HTTP status code 403 as it is unauthorized. If the token is expired, the request is rejected with HTTP status code 440, which is an informal code for login timeout. Upon receiving the 440 response, the frontend will log the user out and prompt them to log back in to be re-authenticated.

Login Flow

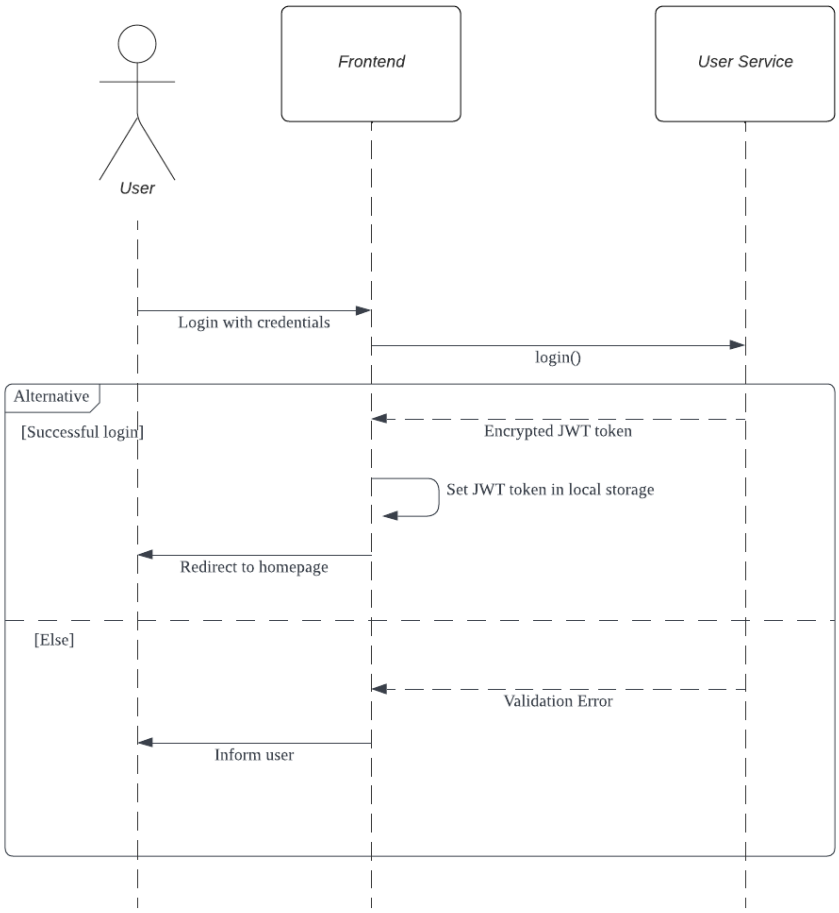


Figure 35: Login Flow Sequence Diagram

Question Service

The Question service provides a pair of matched users with two random questions of the difficulty that they have selected. The questions will be presented to them upon being matched and after both have entered a room. Each question is for each user where they get to independently decide who goes first. The frontend holds both questions in a state and will display each accordingly based on the users' action. The motivation behind the second question is so that the users get to switch their role from an interviewee to an interviewer and vice versa.

The source for the questions seed file comes directly from Leetcode and the construction of the schema was based on what we have selectively handpicked from each question on Leetcode.

```
const questionSchema = new mongoose.Schema<IQuestion, IQuestionModel>({
  title: {
    type: String,
    required: true,
  },
  difficulty: {
    type: String,
    enum: DIFFICULTY,
    required: true,
  },
  description: {
    type: String,
    required: true,
  },
  topics: {
    type: [String],
    enum: TOPIC,
  },
  examples: {
    type: [
      {
        input: {
          type: String,
          required: true,
        },
        output: {
          type: String,
          required: true,
        },
        explanation: {
          type: String,
        },
      },
    ],
  },
  templates: {
    type: [
      {
        language: {
          type: String,
          required: true,
        },
        starterCode: {
          type: String,
          required: true,
        },
      },
    ],
  },
  link: {
    type: String,
    required: true,
  },
});
```

Figure 36: A code snippet of Question Schema

History Service

The History service is in charge of saving the mock interview session's state between a pair of users. It records the questions attempted, codes written in the code editor and chat messages between the users. Additionally, statistics on a user is generated and processed as soon as a history record is received. Some examples of statistics include questions completed grouped by difficulty, languages used for the questions, the date when a question is completed, topics completed, and both daily and longest consecutive streak. This service serves as an important role to help power the user's dashboard, the page that the user sees as soon as he/she logs in. These information help to motivate and inspire the users in their preparation and practice.

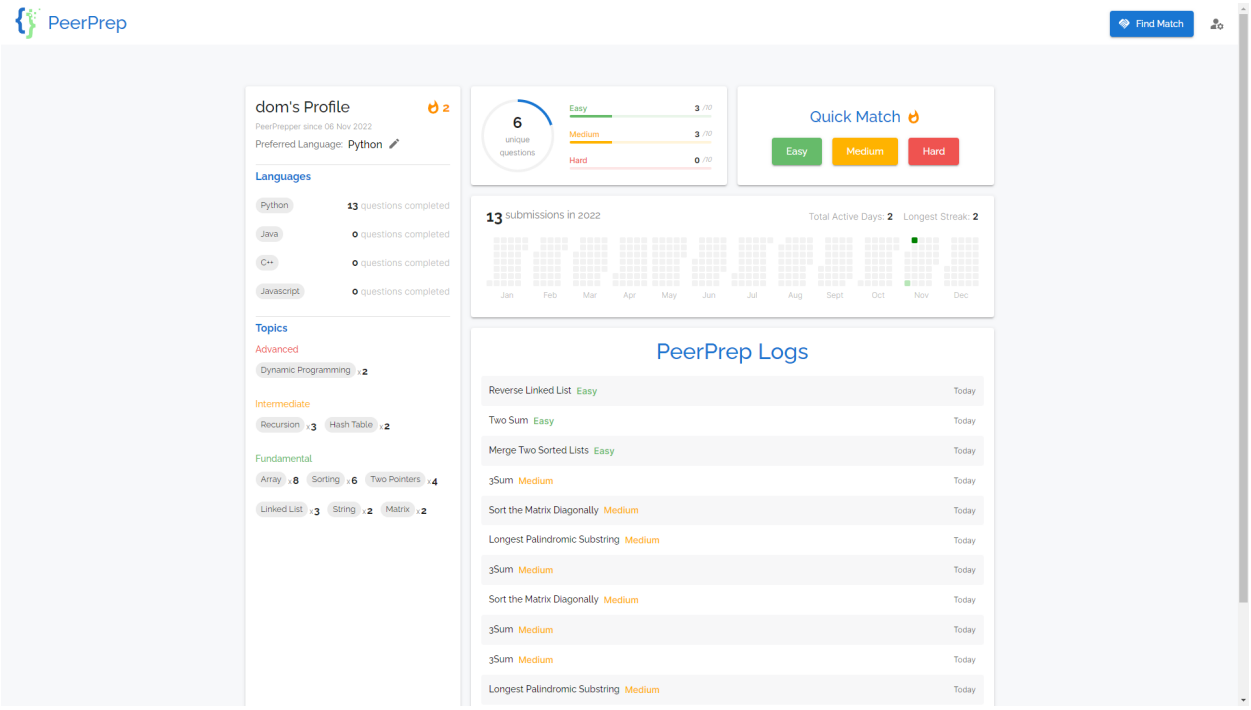


Figure 37: Dashboard

Socket API

Introduction

Socket.IO is a library used by both the frontend and backend for real-time bi-directional communication. It is essential for a large part of our application

including the frontend itself, and a number of microservices namely matching, collaboration and messaging.
Socket.IO at its core an event-based communication and knowing the list of Socket.IO events is essential in understanding the application's design. The socket.io events are listed below.

Design Pattern: Publisher-Subscriber

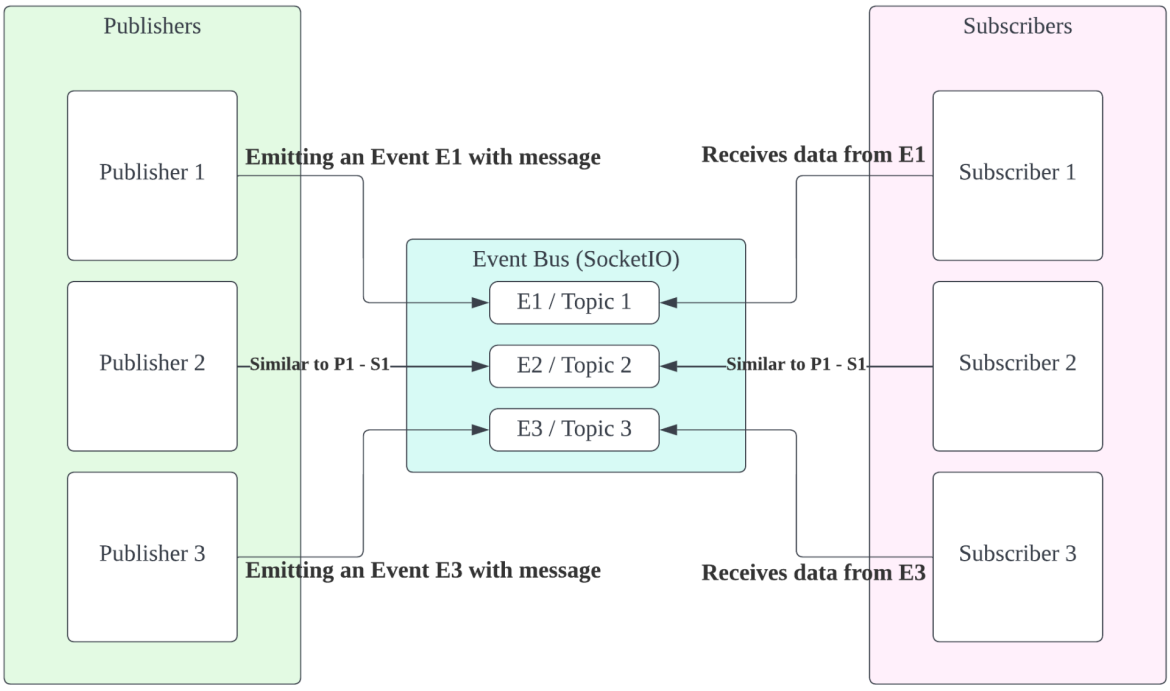


Figure 38: High-level overview of the pub-sub implemented

The Publisher-Subscriber (pub-sub) messaging pattern is mainly used for these microservices - Matching, Collaboration and Messaging. Our system makes use of SocketIO to enforce this messaging pattern. As communication (generally across all microservices) needs to be reliable and performant, this pattern and choice of implementation have helped to uphold these quality attributes.

The overall key idea is that a pair of users will first connect to a socket opened by the SocketIO server. Once a connection has been established, they will both join the

same Publish-Subscribe channel and subscribe to the same topic. From there on, when either user (or sender) publishes messages on that topic, these messages will be broadcasted to all the consumers that have subscribed to the topic. However, in our context, there will be at most two subscribers. But using this pattern allows for scalability where it can be used in the future should there be more than two subscribers involved.

We will explore how each socket API microservice implements this pattern below.

Matching Service

The matching service initiates and maintains a connection between two users. It uses socket.io to enable real-time communication. The matching service uses in-memory SQLite. We chose SQLite because of its speed which is in the context of real time match making, and because which is extremely fast and serves its purpose well.

There are three entities in the matching database, namely User, WaitRoom, and Room. When a user's web client first connects to the matching service through socket.io, a user entity is created. When the user initiates match finding from the frontend, a "matchStart" socket.io event is transmitted to the matching service, triggering a sequence of events. The "matchStart" event also contains a difficulty variable which the match service would use to satisfy FR 2.1.

The details of the sequence are shown in the following sequence diagram:

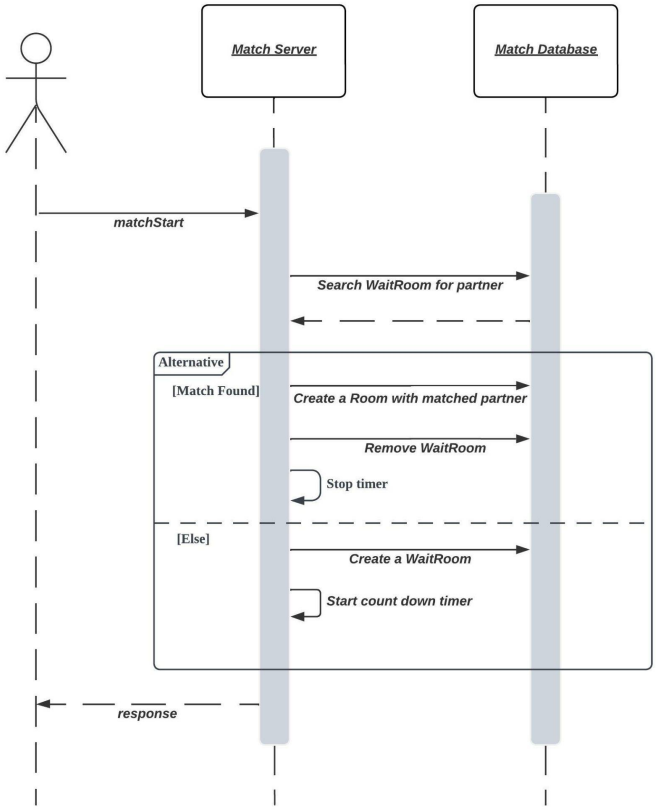


Figure 39: Interaction between client to Match Server and DB Sequence Diagram

When a matchStart event is received by the server, this signals a new user is looking for a partner. The match server queries the database for the wait room that is created the earliest and is of the correct difficulty. Once found, matchmaking is successful and a matched room is finally created. The WaitRoom table essentially serves as a match queue. The above describes how FR 2.2 is implemented.

A timer is created and held by the server when a wait room is created. The server emits a countdown socket.io event to the frontend every second which triggers the frontend timer to update (FR 2.3). When the timer times out, the server does not send a separate socket.io event to inform the frontend. However, it sends a countdown event with time equal to zero. The frontend is designed to interpret this as a timeout event and notifies the user of it (FR2.4).

Matching Service		
Event	**Description**	**Emitted by**
matchStart	Triggered when user starts matchmaking in the frontend. The matching service will find a match for the user depending on the difficulty selected.	Frontend
matchLeave	Triggered when user leaves a room. The matching service informs the other user and cleans up.	Frontend
matchSuccess	Emitted to the frontend when a match is found	Matching Service

matchCountdown	Emitted to the frontend when the waiting timer ticks down	Matching Service

Matchmaking Flow (Happy Path)

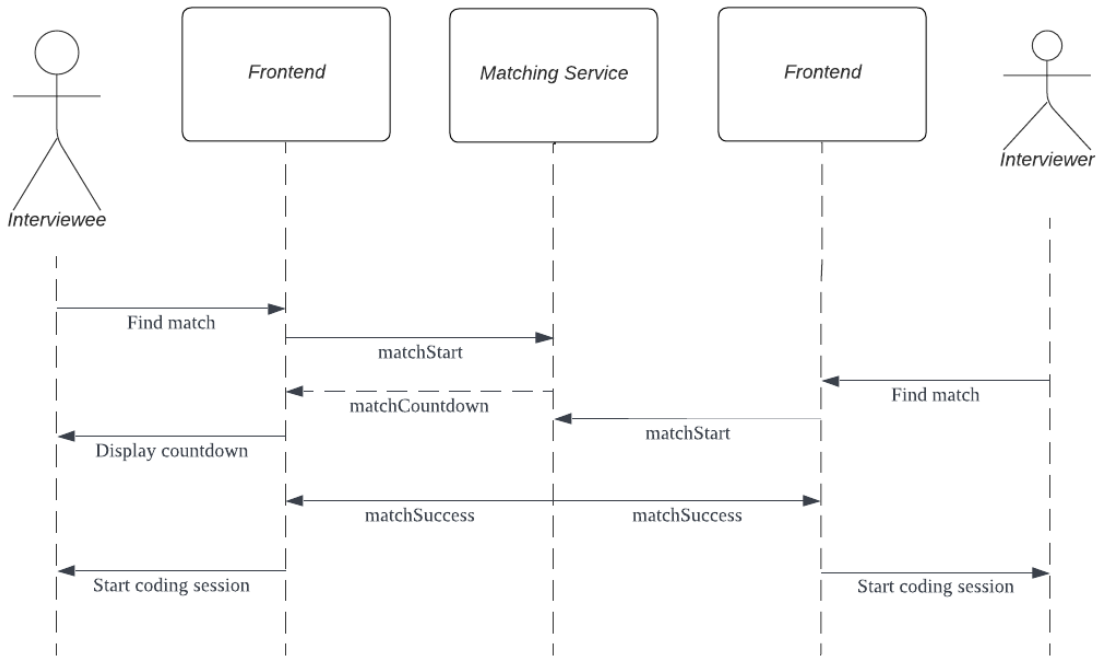


Figure 40: Matchmaking Flow (Happy Path) Sequence Diagram

Matchmaking Flow (No Match)

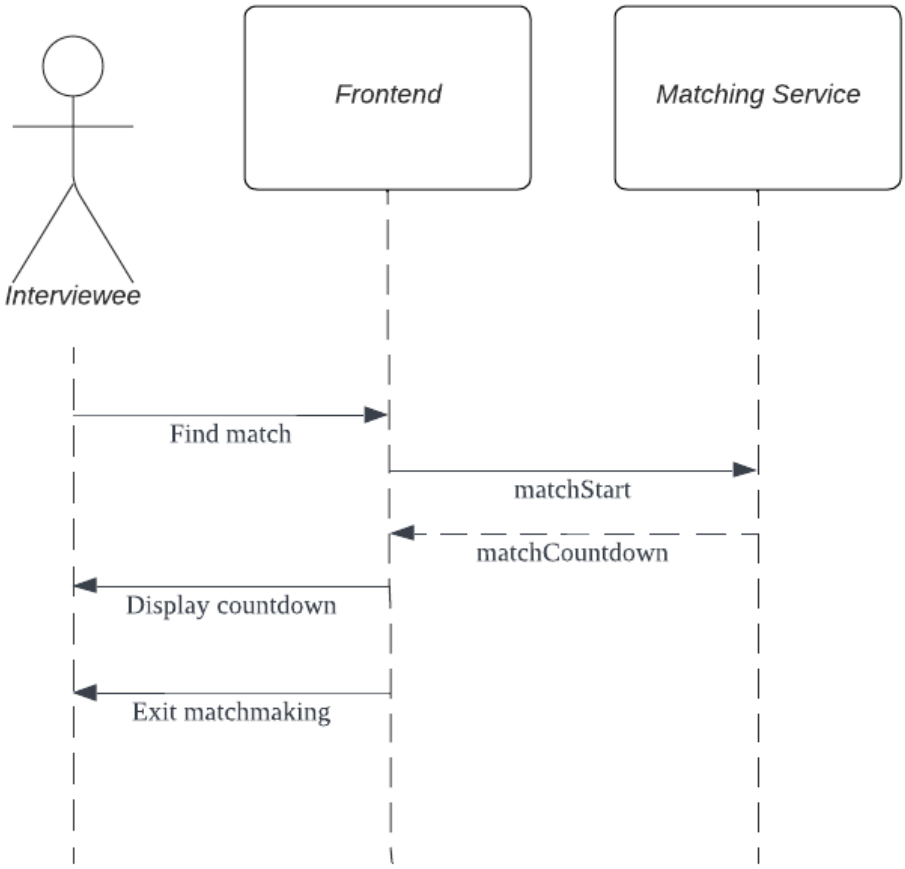


Figure 41: Matchmaking Flow (No Match) Sequence Diagram

Collaboration Service

Matched users will subscribe to the same topic in the collaboration service through the Socket.IO API. Whenever a user makes any change to the code editor, the user's code editor displays the input instantly on the frontend and the changes to the code editor are published to the topic. The other user who is also a subscriber of that topic will receive the editor changes and his or her code editor will update accordingly and display the changes.

The pub-sub pattern along with SocketIO ensures little to no latency for the user making changes to the code editor. High latency for such a platform will cause annoyance, hence providing a bad user experience. Performance for this service is more important as compared to the chat as messaging is usually done on a best-effort delivery basis. Additionally, updating the state of the code editor is more complex under the hood than merely pushing a new message into the chat logs array for the Messaging Service.

Redis also supports the Collaboration Service. Each room with a pair of matched users are given two questions. The purpose is to have the two users swap roles, interviewer to interviewee vice versa, so as to allow them to have the practice they need for each session. However, both users have to mutually accept before moving onto the next question, or ending the session after finishing both questions. We use Redis as an in-memory store to identify which user has selected yes for both scenarios and the state will be updated accordingly.

Collaboration Service Socket Events		
Event	**Description**	**Emitted by**
editorChange	Triggered by the frontend when a user edits the code in the text editor.	Frontend

	The collaboration service then broadcasts the event.	
selection	Triggered by the frontend when one user's cursor position and selection changes in the frontend. The collaboration service then broadcast the event.	Frontend
openNextQuestion	Triggered by the frontend when one of the users presses the next question button or end session button.	Frontend
openNextQuestionPrompt	Broadcasted by the collaboration service in response to receiving the *openNextQuestion* event. This indicates to the frontend of both users that the next question prompt should be opened.	Collaboration Service
acceptNextQuestion	Triggered by the frontend when a user confirms that they would like to proceed to the next question in the next	Frontend

	question prompt.	
rejectNextQuestion	Triggered by the frontend when a user rejects proceeding to the next question in the next question prompt.	Frontend
otherUserRejectedNext-Question	Emitted by the collaboration service to the second user when it receives the *rejectNextQuestion* event from the first user.	Collaboration Service
proceedToNextQuestion	Broadcasted by the collaboration service when both users agree to proceed to the next question. Indicates to the frontend of both users to change to the next question and reset the timer.	Collaboration Service
error	Broadcasted by the collaboration service when any server error occurs	Collaboration Service
timerStart	Triggered by the frontend when the stopwatch is started.	Frontend

timerStop	Triggered by the frontend when the stopwatch is stopped.	Frontend
timerPause	Triggered by the frontend when the stopwatch is paused.	Frontend
timerResume	Triggered by the frontend when the stopwatch is resumed.	Frontend

Collaboration Room Code Editor Flow

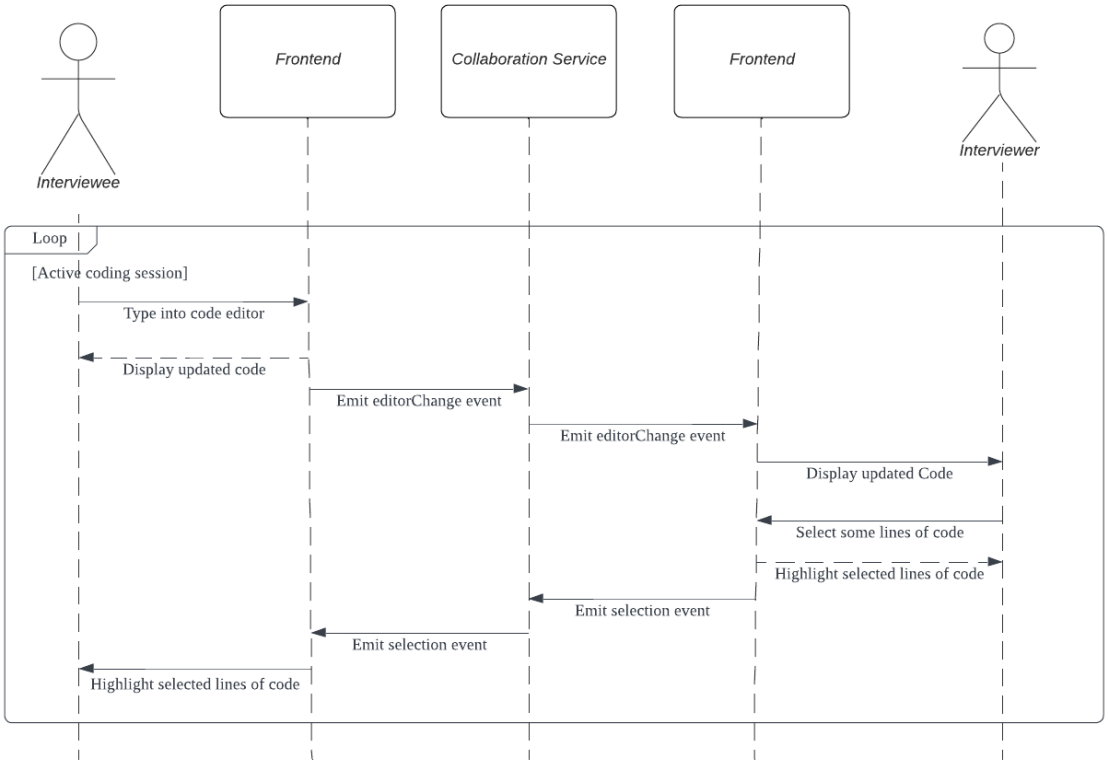


Figure 42: Collaboration Room - Code Editor Sequence Diagram

Collaboration Room Timer Ticking Flow

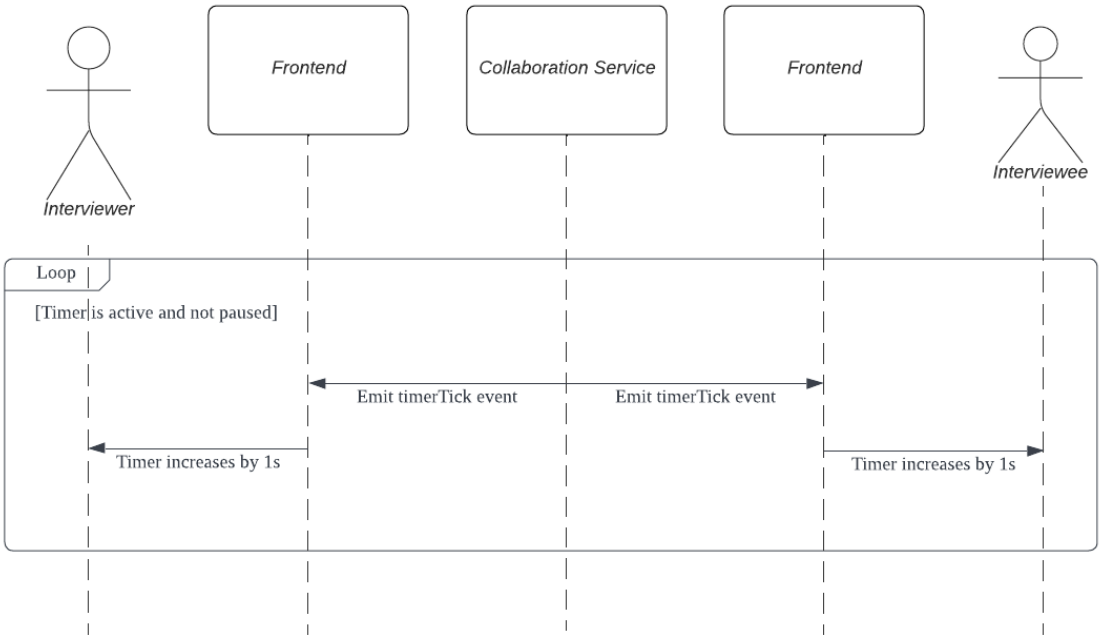


Figure 43: Collaboration Room - Timer Ticking Sequence Diagram

Collaboration Room Timer Start / Stop Flow

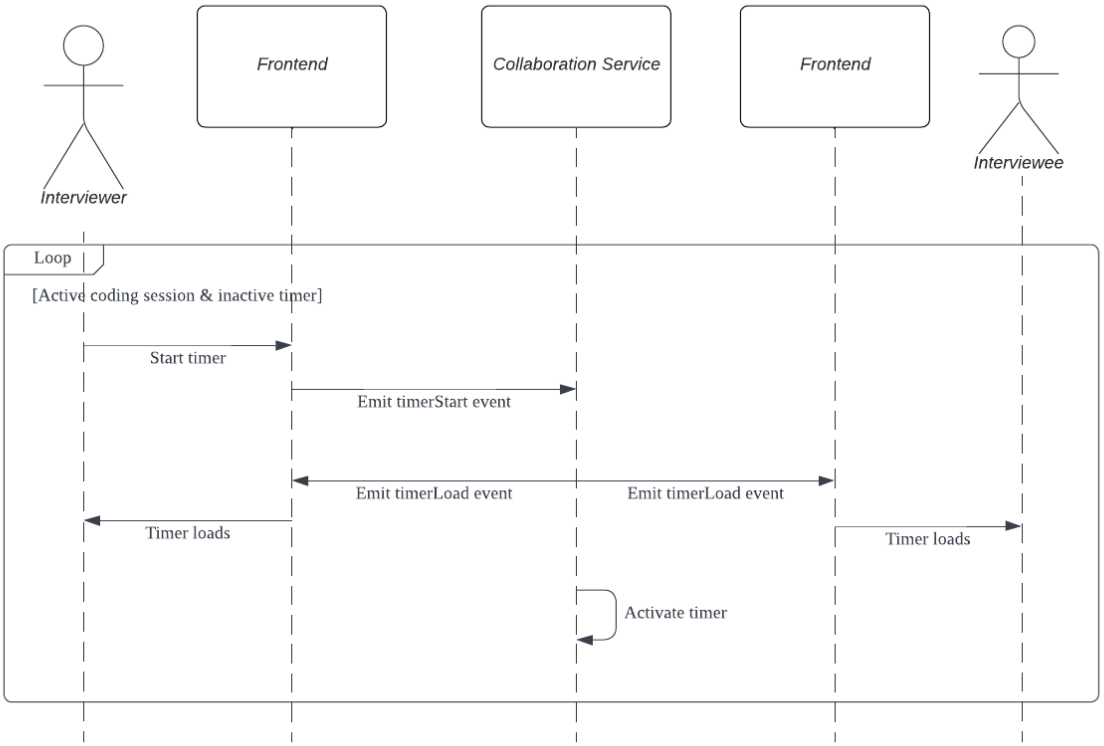


Figure 44: Collaboration Room - Timer Start Sequence Diagram

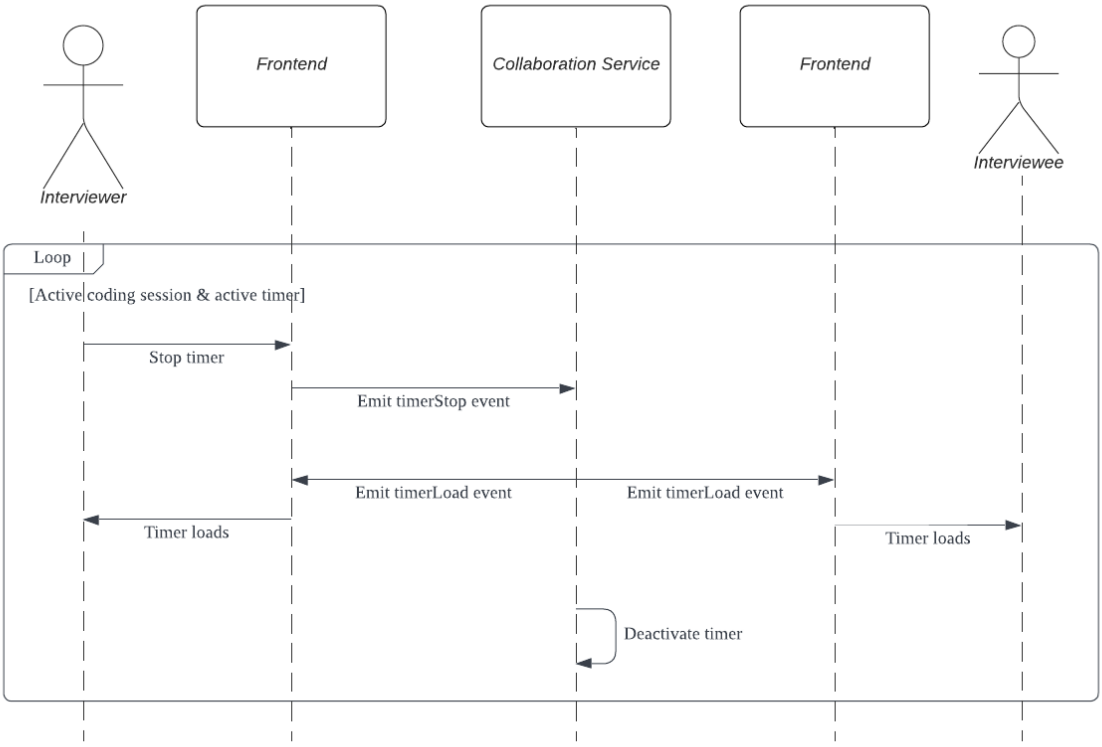


Figure 45: Collaboration Room - Timer Stop Sequence Diagram

Collaboration Room Timer Pause / Resume Flow

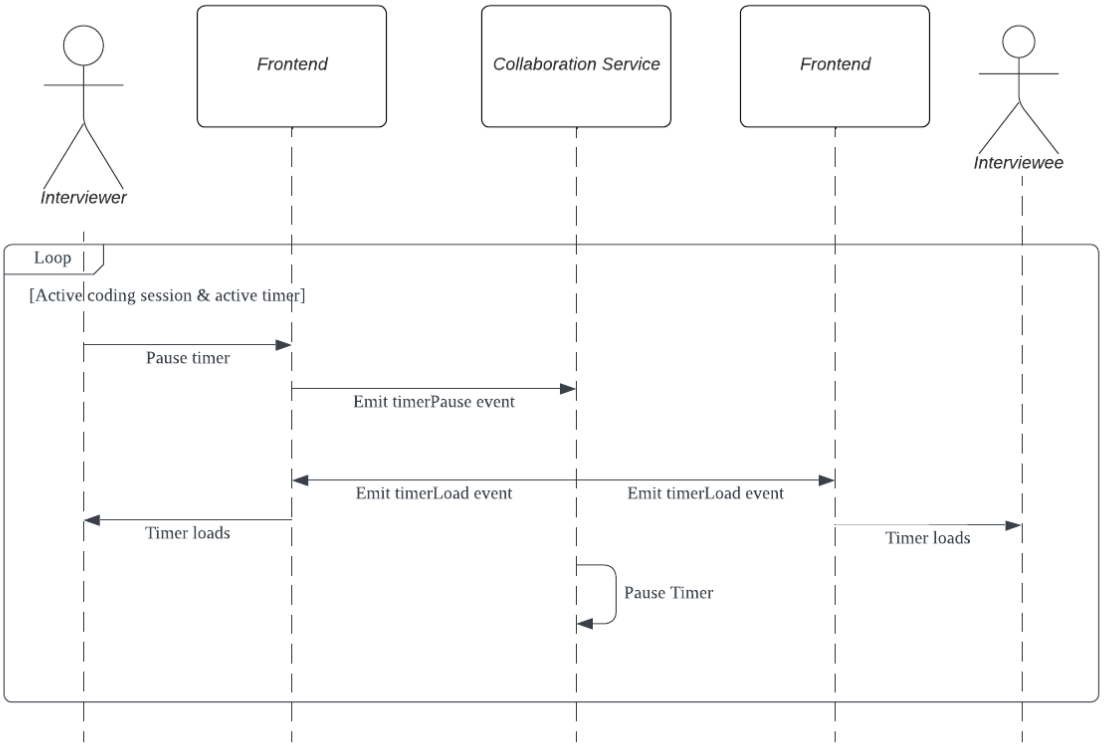


Figure 46: Collaboration Room - Timer Pause Sequence Diagram

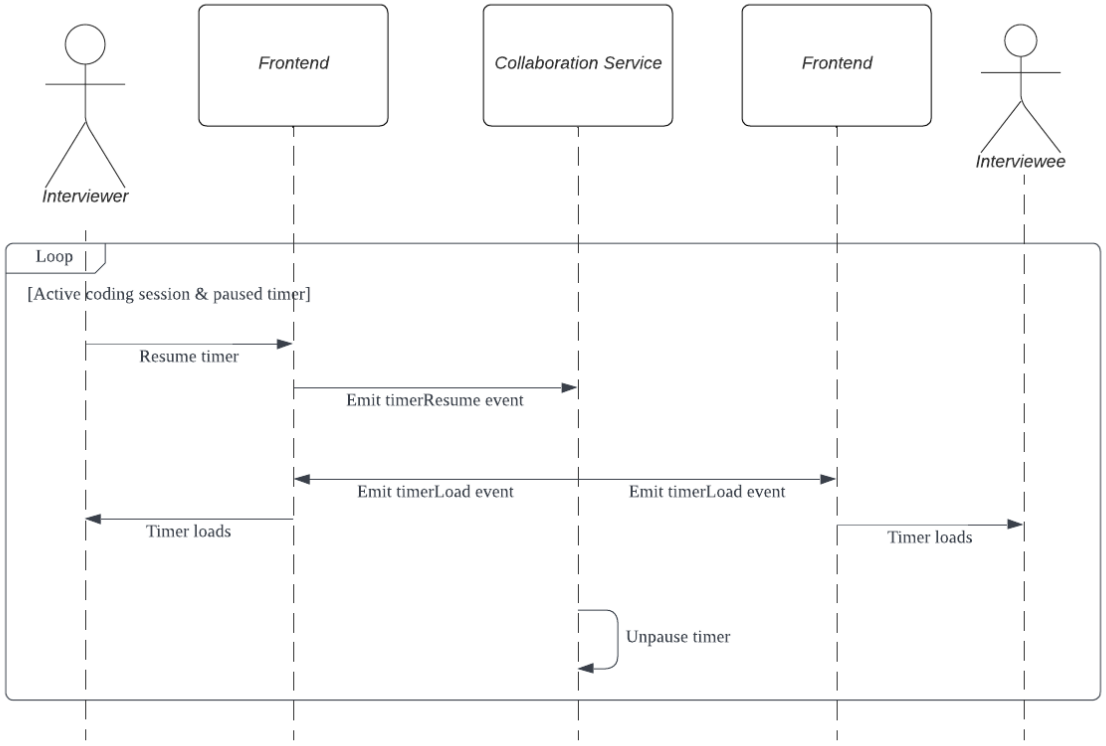


Figure 47: Collaboration Room - Timer Resume Sequence Diagram

Collaboration Room Next Question Flow (Happy Path)

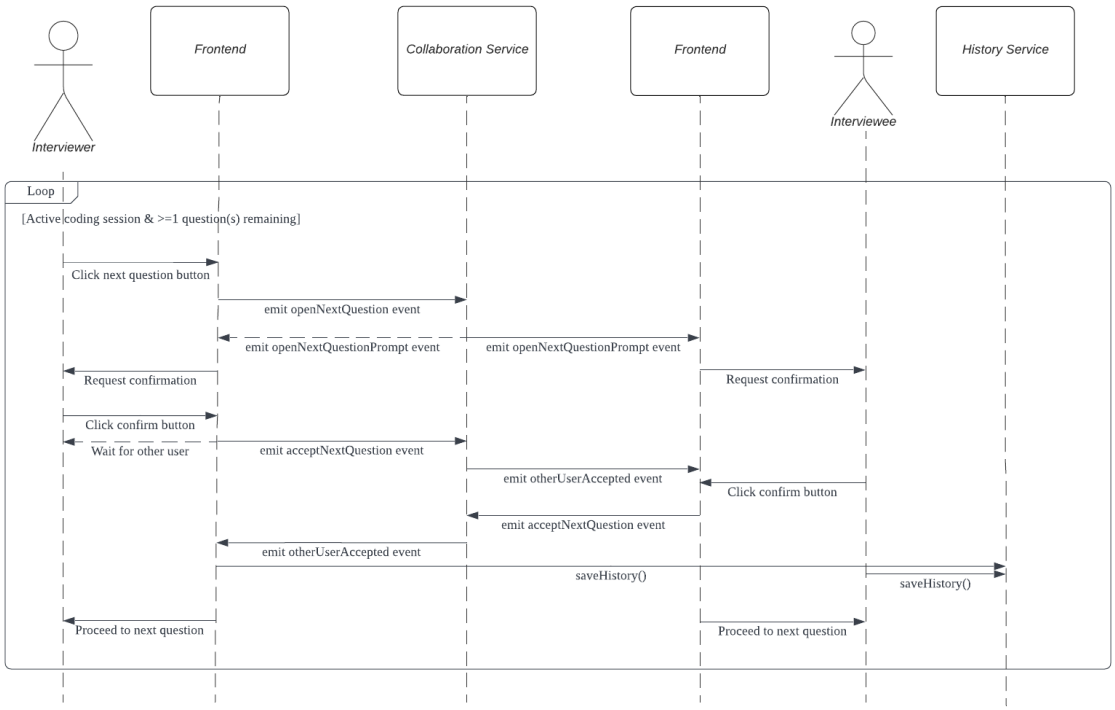


Figure 48: Collaboration Room - Next Question (Happy Path) Start Sequence Diagram

Collaboration Room Next Question Flow (Rejection)

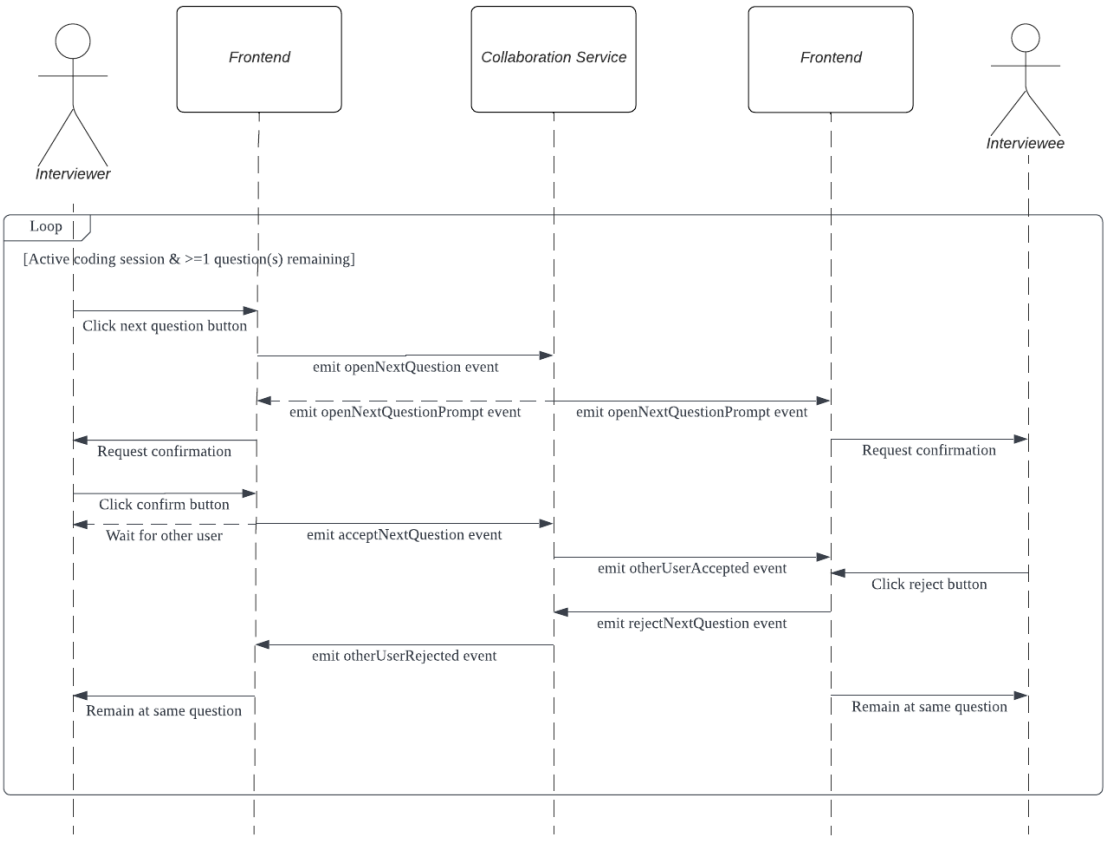


Figure 49: Collaboration Room - Next Question (Rejection) Start Sequence Diagram

Messaging Service

The messaging service handles the chats in a session/room between a pair of matched users. It utilises the server provided by SocketIO API.

Pub-sub pattern ensures that the Messaging Service provides an updated and synchronised chat log between the subscribers to that topic. It is a design decision to use this pattern as there is possibly some latency introduced for the message sender if implemented with another approach e.g., RESTful API to subscribe to that schema and listen for changes. However, it is unlikely to be noticeable.

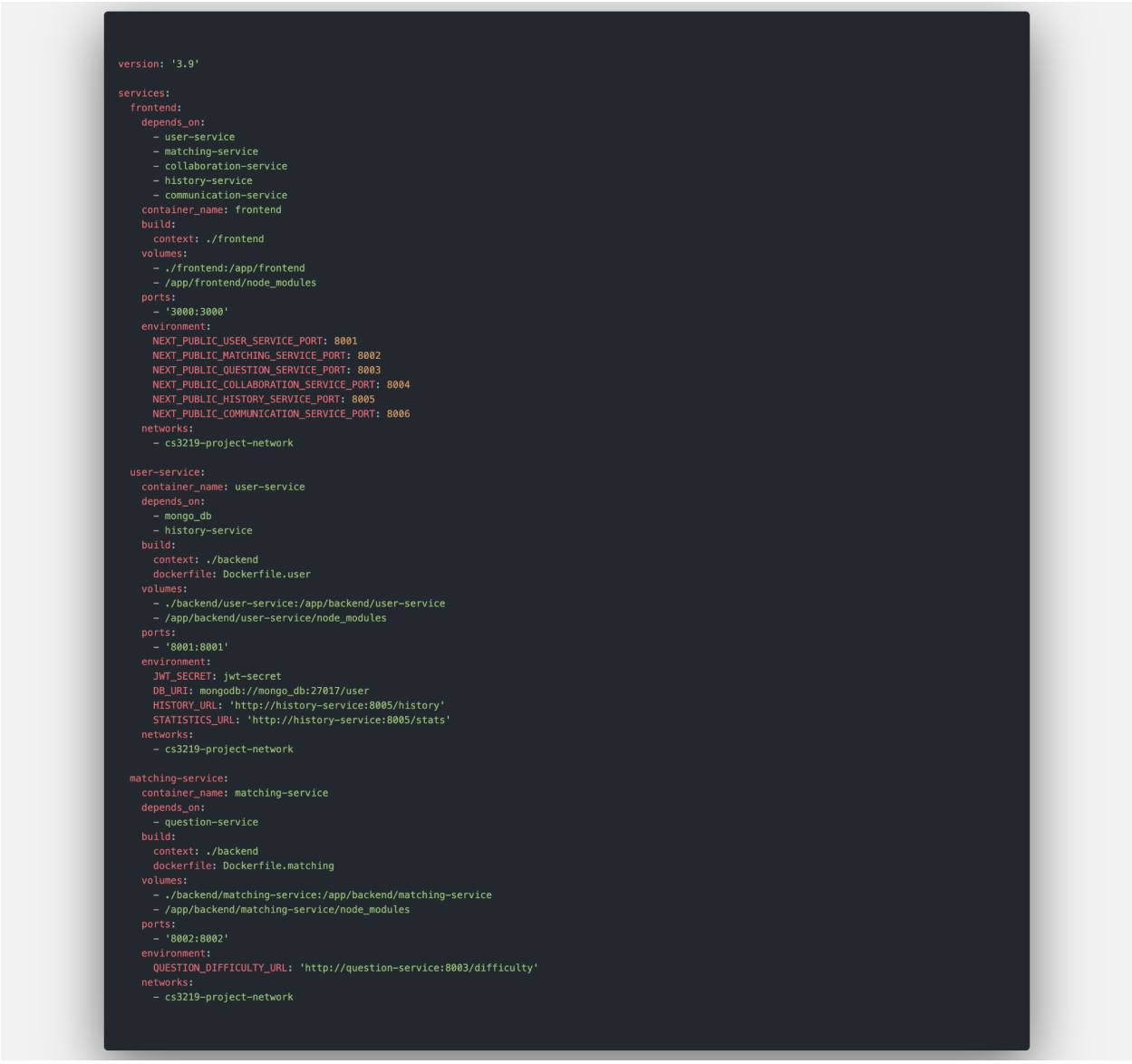
The flow as such is a pair of users matched will subscribe to the same topic in the messaging service through the SocketIO API. When a user sends a message, the message gets published to the topic and both users will receive the message. The front end Chat box component receives the latest messages and displays it to both users.

Messaging Service Socket Events		
Event	**Description**	**Emitted by**
message	Triggered when an user sends a chat message. The Chat service then broadcasts the message, upon which the chat panel of each user's frontend updates to include the new message.	Frontend

DevOps

Local Deployment/Development with Docker Compose

Local deployment is done using Docker Compose. Since this is an application with multiple microservices, developing without Docker Compose would be highly inconvenient as a developer needs to start up each of the seven processes, including frontend, separately. With Docker Compose however, all a developer needs to do to start up the entire application is to execute "*Docker Compose Up*".
The following is our docker-compose.yml file that sets up the local development environment:



```yaml
version: '3.9'

services:
  frontend:
    depends_on:
      - user-service
      - matching-service
      - collaboration-service
      - history-service
      - communication-service
    container_name: frontend
    build:
      context: ./frontend
    volumes:
      - ./frontend:/app/frontend
      - /app/frontend/node_modules
    ports:
      - '3000:3000'
    environment:
      NEXT_PUBLIC_USER_SERVICE_PORT: 8001
      NEXT_PUBLIC_MATCHING_SERVICE_PORT: 8002
      NEXT_PUBLIC_QUESTION_SERVICE_PORT: 8003
      NEXT_PUBLIC_COLLABORATION_SERVICE_PORT: 8004
      NEXT_PUBLIC_HISTORY_SERVICE_PORT: 8005
      NEXT_PUBLIC_COMMUNICATION_SERVICE_PORT: 8006
    networks:
      - cs3219-project-network

  user-service:
    container_name: user-service
    depends_on:
      - mongo_db
      - history-service
    build:
      context: ./backend
      dockerfile: Dockerfile.user
    volumes:
      - ./backend/user-service:/app/backend/user-service
      - /app/backend/user-service/node_modules
    ports:
      - '8001:8001'
    environment:
      JWT_SECRET: jwt-secret
      DB_URI: mongodb://mongo_db:27017/user
      HISTORY_URL: 'http://history-service:8005/history'
      STATISTICS_URL: 'http://history-service:8005/stats'
    networks:
      - cs3219-project-network

  matching-service:
    container_name: matching-service
    depends_on:
      - question-service
    build:
      context: ./backend
      dockerfile: Dockerfile.matching
    volumes:
      - ./backend/matching-service:/app/backend/matching-service
      - /app/backend/matching-service/node_modules
    ports:
      - '8002:8002'
    environment:
      QUESTION_DIFFICULTY_URL: 'http://question-service:8003/difficulty'
    networks:
      - cs3219-project-network
```

Figure 50: A code snippet of docker-compose.yml (1st half)

```yaml
question-service:
    container_name: question-service
    depends_on:
      - mongo_db
    build:
      context: ./backend
      dockerfile: Dockerfile.question
    volumes:
      - ./backend/question-service:/app/backend/question-service
      - /app/backend/question-service/node_modules
    ports:
      - '8003:8003'
    environment:
      DB_URI: mongodb://mongo_db:27017/question
    networks:
      - cs3219-project-network

  collaboration-service:
    container_name: collaboration-service
    depends_on:
      - redis
      - history-service
    build:
      context: ./backend
      dockerfile: Dockerfile.collaboration
    volumes:
      - ./backend/collaboration-service:/app/backend/collaboration-service
      - /app/backend/collaboration-service/node_modules
    ports:
      - '8004:8004'
    environment:
      REDIS_HOST: 'redis'
      HISTORY_URL: 'http://history-service:8005'
    networks:
      - cs3219-project-network

  history-service:
    container_name: history-service
    depends_on:
      - mongo_db
    build:
      context: ./backend
      dockerfile: Dockerfile.history
    volumes:
      - ./backend/history-service:/app/backend/history-service
      - /app/backend/history-service/node_modules
    ports:
      - '8005:8005'
    environment:
      DB_URI: mongodb://mongo_db:27017/history
      QUESTION_URL: http://question-service:8003
    networks:
      - cs3219-project-network

  communication-service:
    container_name: communication-service
    build:
      context: ./backend
      dockerfile: Dockerfile.communication
    volumes:
      - ./backend/communication-service:/app/backend/communication-service
      - /app/backend/communication-service/node_modules
    ports:
      - '8006:8006'
    networks:
      - cs3219-project-network

  mongo_db:
    container_name: mongo_db
    image: mongo:latest
    restart: always
    ports:
      - '27017:27017'
    volumes:
      - mongo_db:/data/db
    networks:
      - cs3219-project-network
    logging:
      driver: 'none'

  redis:
    container_name: redis
    image: redis:latest
    restart: always
    ports:
      - '6379:6379'
    volumes:
      - redisdata:/data
    networks:
      - cs3219-project-network

networks:
  cs3219-project-network:
    driver: bridge

volumes:
  mongo_db: {}
  redisdata: {}
```

Figure 51: A code snippet of docker-compose.yml (2nd half)

Here we elaborate on some notable notes associated with this Docker Compose setup.

- Docker Compose helps us create a virtual network. Within this virtual network, the various services can easily access each other through the service name and designated port number.
- Instead of using Mongodb Atlas cloud database for local development, a container running mongodb is configured. This is so that each developer will access to his or her own local database instead of sharing a development cloud database which would negatively affect developer experience.
- For each backend microservice, we mount two volumes in their respective docker containers. Docker volumes are file systems that can be mounted on Docker containers to preserve data generated by the running container. As such, these volumes help us to persist changes to the source code and packages that we make during development. This means that with supporting tools such as webpack for our React frontend and nodemon for our Express backend microservices, we are able to hot reload the containers that we are working on to see the effects of code changes that we make without having to restart the container, hence greatly speeding up our development process.

Deployment on AWS

Introduction to Elastic Container Service

Deployment of the application is done using AWS Elastic Container Service (ECS). ECS is a highly scalable container management service. Here we go through a few key concepts related to our ECS.

Task definitions and Tasks

To run containers in ECS, we need to first define task definitions which contains key information about the containers we want to run such as Docker image used, CPU and memory resource allocated, network settings, the launch type of the container

and more. Each task definition can run one or more containers. Tasks are the container processes and the environment that are being maintained and run by AWS according the task definition associated with it.

ECS Service

ECS Services are task management processes associated with a task. Services provide features such as load balancing, health checks, automatic scaling, automatic redeployment, and numerous other key management functionalities. Typically

Elastic Container Regisity (ECR)

The ECR is a Docker image repository stored in AWS cloud. Like github, we will push the latest production build to ECR which would then be used by the ECS tasks and services.

Overview of deployment

Our application has six microservices and one frontend website. For each of these we run one ECS service for it. With ECS service, we also configured load balancing by using AWS application load balancers and automatic scaling. Diagram 52 below represents the deployment of one particular microservice or frontend. For each of the microservices and frontend, the following steps are taken to create a robust and scalable service:

1. Create a task definition that would create container(s) pulling image from the ECR
2. Create a ECS service using the previously created task definition
3. Create an application load balancer and a target group in EC2 dashboard and add the the load balancer to the service
4. Configure auto-scaling by using CPU utilization as the scaling metric

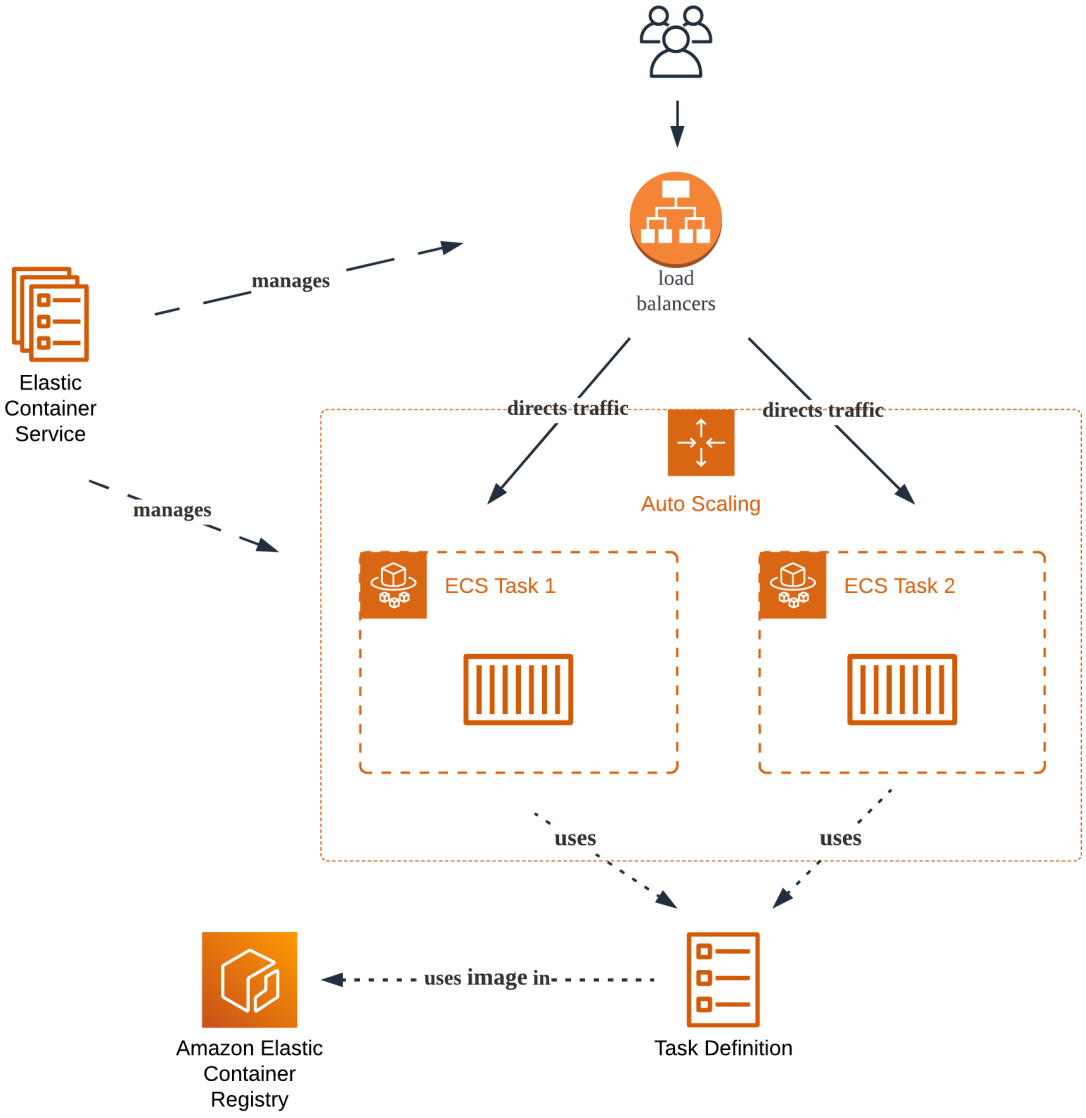


Figure 52: ECS deployment architecture:

Key deployment decisions

1. Multiple services in one task definition VS single service in single task definition

   For deployment on ECS, we can choose to deploy multiple microservices in one task definition or deploy them separately in different task definitions. We made the decision to deploy each of the microservices and frontend in

separate task definitions. By doing so, we can scale, provision and deprovision each of them independently. This is how microservices should be deployed.

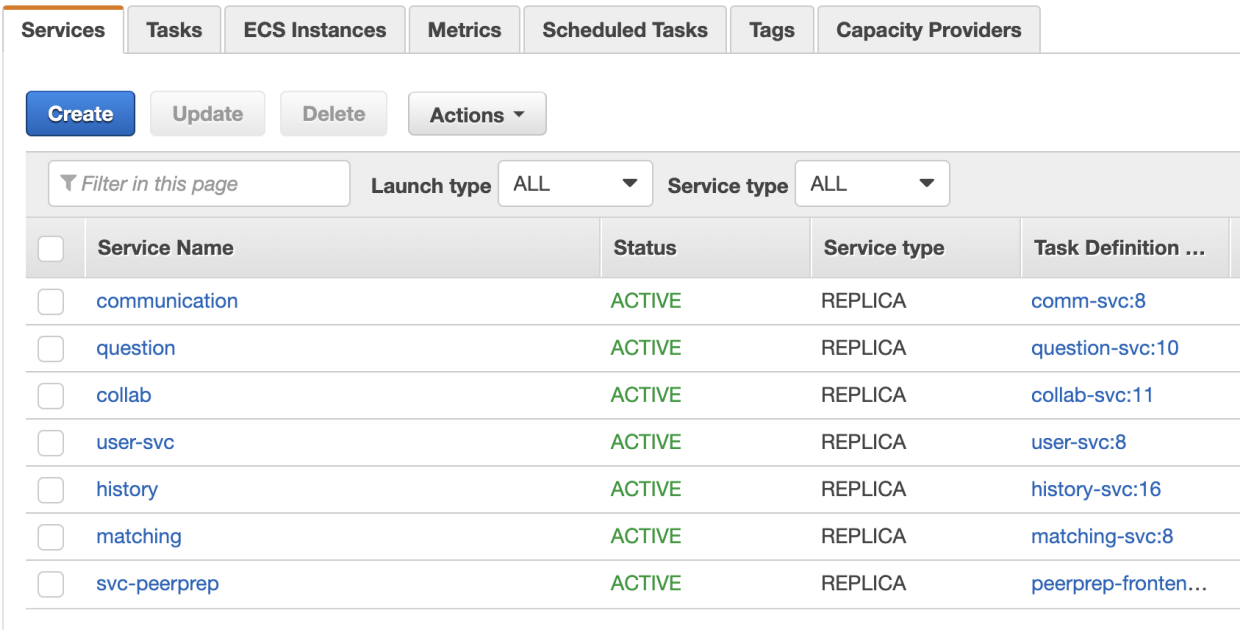


Figure 52: ECS services

2. Auto-scaling using ECS auto-scaler.
   There are several metrics that could be used for auto-scaling, we decided to go with average CPU utilization as it is general measure of how much client load is handling at a particular time.

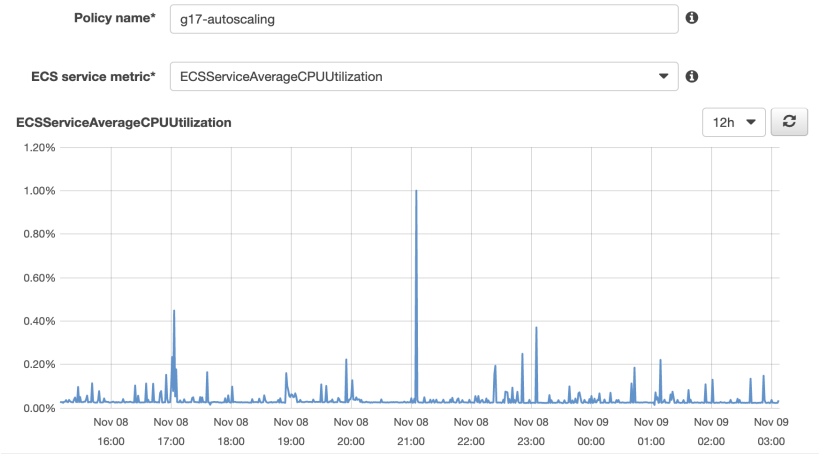


Figure 53: ECS CPU Monitoring

3. Load balancer

   We also added load balancer to each of the ECS services. This is a must-have since we have configured auto-scaling. If there is no balancer, then running multiple tasks of the same task definition serves no purpose since clients would only be routed to a single task. With load balancer, different clients would be automatically distributed across the various tasks of the same task definition running in an ECS service.

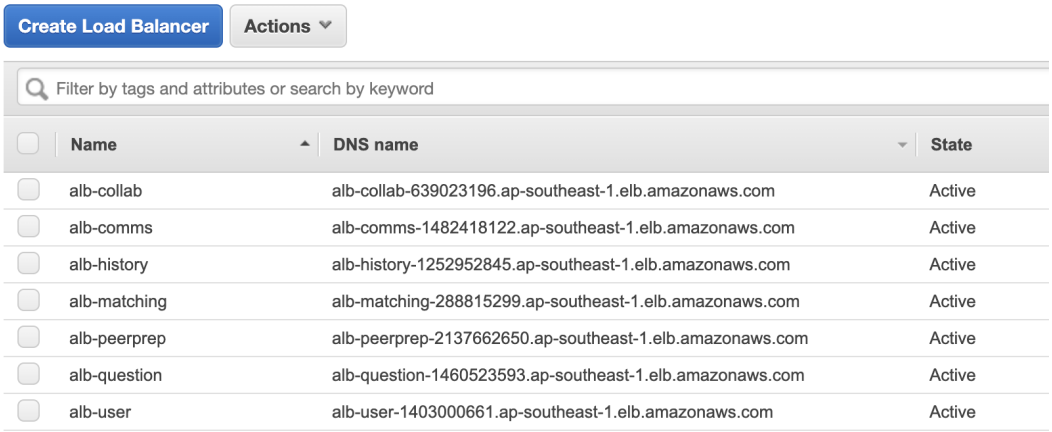


Figure 54: Application Load Balancers

CI/CD with GitHub Actions

CI/CD is done using GitHub Actions. One GitHub workflow is set up for each of the microservices and frontend. Each workflow consists of the following steps:

1. Configure AWS credentials and login to AWS ECR
2. Build the latest Docker image using the new code pushed to GitHub repository
3. Push the new image to ECR
4. Create new task definition based on the new image
5. Deploy the new task definition over the old task currently running in the ECS service

For example, below is the GitHub workflow for Matching Service:

```yaml
name: Deploy Matching Service

on:
  push:
    branches: ['production']

env:
  AWS_REGION: ap-southeast-1 # set this to your preferred AWS region, e.g. us-west-1
  ECR_REPOSITORY: matching-svc # set this to your Amazon ECR repository name
  ECS_SERVICE: matching # set this to your Amazon ECS service name
  ECS_CLUSTER: peerprep-g17 # set this to your Amazon ECS cluster name
  ECS_TASK_DEFINITION:
    .aws/matching-td.json # set this to the path to your Amazon ECS task definition
    # file, e.g. .aws/task-definition.json
  CONTAINER_NAME:
    matching-svc # set this to the name of the container in the
    # containerDefinitions section of your task definition

permissions:
  contents: read

jobs:
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    environment: production

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ env.AWS_REGION }}

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build, tag, and push image to Amazon ECR
        id: build-image
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        run: |
          docker build -f backend/matching-service/Dockerfile.prod.matching \
              --build-arg "QUESTION_DIFFICULTY_URL=${{ secrets.QUESTION_DIFFICULTY_URL }}" \
              -t $ECR_REPOSITORY backend/matching-service/
          docker tag $ECR_REPOSITORY:latest $ECR_REGISTRY/$ECR_REPOSITORY:latest;
          docker push $ECR_REGISTRY/$ECR_REPOSITORY:latest
          echo "::set-output name=image::$ECR_REGISTRY/$ECR_REPOSITORY:latest"

      - name: Fill in the new image ID in the Amazon ECS task definition
        id: task-def
        uses: aws-actions/amazon-ecs-render-task-definition@v1
        with:
          task-definition: ${{ env.ECS_TASK_DEFINITION }}
          container-name: ${{ env.CONTAINER_NAME }}
          image: ${{ steps.build-image.outputs.image }}

      - name: Deploy Amazon ECS task definition
        uses: aws-actions/amazon-ecs-deploy-task-definition@v1
        with:
          task-definition: ${{ steps.task-def.outputs.task-definition }}
          service: ${{ env.ECS_SERVICE }}
          cluster: ${{ env.ECS_CLUSTER }}
          wait-for-service-stability: true
```

Figure 55: Code snippet of matching-service.yml (github workflow)

The workflow is set to run when changes are pushed to the production branch. Once the workflow is complete, the new services will be deployed over the old services, achieving continuous deployment.

Improvements & Enhancements

There are many areas for improvements and enhancements. Some possibilities are adding more features/functionalities to our existing services, logging on the server and re-prioritising the quality attributes.

Let us first start from the top and go back to the drawing board and look at the quality attributes. Security has a score of 1 and we would like to improve on that for starters.

Given the context of this being a school project, we have undervalued security. Generally, we would not expect abusive or malicious users, or even Cross Site Request Forgery (CSRF) attacks. If we were to release PeerPrep as a commercial product, security will have to shift up the scoreboard as an important consideration. To improve on this area, we could work towards adding authorization checks aside from the baseline layer of security that we have which is authentication. We could opt for a secure domain with a signed SSL certificate to ensure that the connection between client and server is encrypted. This would mean that our deployed url will be HTTPS. It also allows us to make use of HTTP-only cookies to help prevent JSON Web Tokens being exposed to the browser or client site scripting as this leaves the application opened to CSRF attacks. We would also secure our routes with an authorisation middleware so that a client's private information can only be accessed by themselves.

Next, for the User Service, we would look into allowing users to add other users i.e., A friends list, and allow peers to interview one another directly instead of having the system perform a random match. Aside from that, we would introduce real-time statuses of each user to allow them to set their status to Online, Busy, a custom status, etc.

As for the Messaging Service, we aspire to integrate voice and video communication to turn it into a *Communication Service* instead as we only have a chat space for

users. This enhances the users' experience as it will simulate a technical interview in its entirety. We will always want to look into allowing more users to join a room, to be able to spectate the mock interview sessions. Perhaps even have a page to show all the active mock interview sessions to allow users to be an audience for any of their choice.

Further enhancement to the Question Service will be to allow users to select their pair of questions from the system's question bank before entering a collaborative room. This will further improve on our Usability quality attribute.

The Collaboration Service can be improved by integrating a compiler to allow users to compile and run their codes for testing, even more so providing an input area for adding custom test cases.

The above are some of the more crucial ones, there might be some others which we might have missed. There will always be room for improvement and we believe that these improvements/enhancements once added, will definitely help to polish up our system and elevate the entire experience.

Reflection & Learning Points

As a team, it has been a really exciting and challenging journey throughout the past 13 weeks of working on PeerPrep. The four of us had a lot of fun with the entire development process, including all of the non-technical work that we have done. We have learnt a lot, and we believe that what we have taken away will help shape and mould us into better Software Engineers. We would like to thank the teaching team for this module and all of their hardwork! We are grateful and appreciative!

Some key learning points that we want to elaborate on:

1. Git hygiene and CI/CD pipeline
   We lack testing and our GitHub issues and pull requests could be better structured. At times, we have taken a back seat for our code reviews (when

timelines are tight, and working around our busy schedules). To improve, we could setup rubric or evaluation metrics to aid us in the project management workflow i.e., What is expected for creating tickets (we could add story points), how to assess when a pull request can be successfully merged in. A note on the CD pipeline would be to implement it earlier, or even at the start before we started out on development. On the other side, as for the CI, we could implement it with automated tests to ensure happy paths are always working, or just in general testing of the entire system.

2. Planning with milestones
   For a project like this, even though we have written out all our backlogs at the very beginning, it is very difficult to gauge what should be the exact deadlines for each set of backlogs. We could have done a better job in defining clear milestones for when to finish what.

Appendix

UI/UX Hi-Fidelity Prototypes

Style Guide

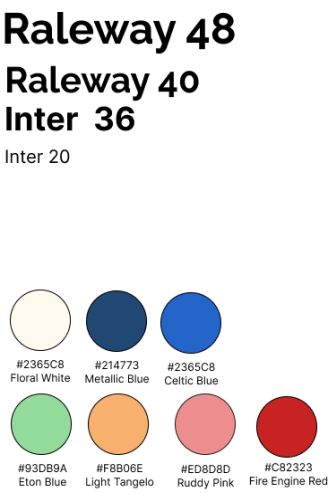


Figure 56: Style Guide

Hi-Fi Mockups

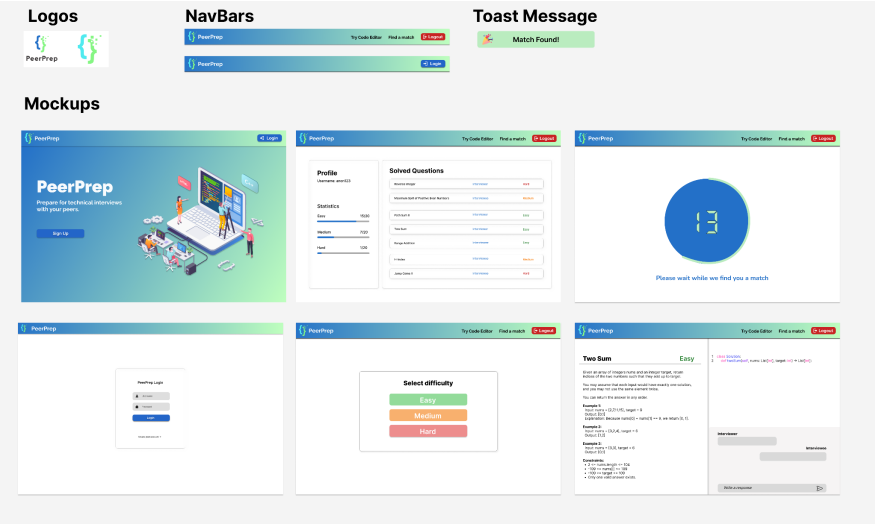


Figure 57: Hi-Fi Mockup

API Documentation

User Service

Health Check

Endpoint: http://localhost:8001/

Brief: This endpoint verifies that the User Service is accessible at localhost:8001.

HTTP Method: **GET**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	A HTML page with a h1 text.	User Service

Register User

Endpoint: http://localhost:8001/register

Brief: This endpoint registers and adds a User and his/her account information into the system.

HTTP Method: **POST**

Required: username (string), password (string)

```
{
  "username": "johndoe",
  "password": "password123!",
  "preferredLanguage": "JAVA"
}
```

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**

200	OK	{
		"username": "johndoe",
		"password": "password123!",
		"preferredLanguage": "JAVA"
		}
			-	{}

400	BAD REQUEST	{
		"username": "johndoe"
		}
			Either required param is missing.	{
				"error": [
				"Password is required"
				]
				}

400	BAD REQUEST	{
		"username": "johndoe",
		"password": "123!"
		}
			Either required param does not meet validator.	{
				"error": [
				"Password is too short"
				]
				}

Login

Endpoint: http://localhost:8001/login

Brief: This endpoint logs a user in based on his/her credentials and returns a JWT stored in local storage and future authenticated routes will extract the JWT and store it in the bearer authorisation header for authentication.

HTTP Method: **POST**

Required: username (string), password (string)

```
{
  "username": "johndoe",
  "password": "password123!"
}
```

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**

200	OK	{
		"username": "johndoe",
		"password": "password123!"
		}
			Returns an object of the user's id and username.	{
				"_id": "63575d7287f136ba410dc9f8",
				"username": "johndoe",
				"__v": 0
				}

400	BAD REQUEST	{
		"username": "johndoe"
		}
			Required param, password, is missing.	{
				"error": "data and hash arguments required"
				}

400	BAD REQUEST	{
		"password": "password123!"
		}
			Required param, username, is missing.	{
				"error": "User undefined not found"
				}

Logout

Endpoint: http://localhost:8001/logout

Brief: This endpoint logs a user out and clears the JWT.

HTTP Method: **POST**

Required: None

Optional: None

Requires Authentication: True

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	Returns an empty object with user	{}

			logged out.	

401	UNAUTHORISED	-	User is not authenticated to be logged out.	{
				"error": "Authentication Required"
				}

Change Password

Endpoint: http://localhost:8001/

Brief: This endpoint changes a user's password.

HTTP Method: **PUT**

Required: userId (string; Found in JWT), currentPassword (string), newPassword (string)

```
{
  "currentPassword": "password123!",
  "newPassword": "123456"
}
```

Optional: None

Requires Authentication: True

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**

200	OK	{
		"userId": "636489217bb6be2f51740127",
		
		"currentPassword": "password123!",
		"newPassword": "123456"
		}
			-	{}

401	UNAUTHORISED	-	User is not	{

			authenticated to change his/her password.	"error": "Authentication Required" }

Delete Account

Endpoint: http://localhost:8001/

Brief: This endpoint deletes a user account and clears the JWT from the local storage.

HTTP Method: **DELETE**

Required: userId (string; Found in JWT)

Optional: None

Requires Authentication: True

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	-	{}

401	UNAUTHORISED	-	User is not authenticated.	{
				"error": "Authentication Required"
				}

Refresh

Endpoint: http://localhost:8001/refresh

Brief: This endpoint generates a new JWT for the current authenticated user based on the existing JWT.

HTTP Method: **POST**

Required: None

Optional: None

Requires Authentication: True

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**

200	OK	-	Returns an object of the user's id and username.	{
				"_id": "63575d7287f136ba410dc9f8",
				"username": "johndoe",
				"__v": 0
				}

401	UNAUTHORISED	-	User is not authenticated.	{
				"error": "Authentication Required"
				}

Change Language

Endpoint: http://localhost:8001/language

Brief: This endpoint updates a user's choice of his/her preferred programming language.

HTTP Method: **PUT**

Required: userId (string), preferredLanguage (string)

```
{
  "userId": "63612d39f54fdd11e59c9c9c",
  "preferredLanguage": "PYTHON"
}
```

Optional: None

Requires Authentication: True

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	{ "userId":	-	{}

		"63612d39f54fdd11e59c9c9c", "preferredLanguage": "PYTHON" }		

401	UNAUTHORISED	-	User is not authenticated.	{
				"error": "Authentication Required"
				}

Question Service

Health Check

Endpoint: http://localhost:8003/

Brief: This endpoint verifies that the Question Service is accessible at localhost:8003/question.

HTTP Method: **GET**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	A HTML page with a h1 text.	Question Service

Get Question

Endpoint: http://localhost:8003/:id

Brief: This endpoint gets a question based on specified id.

HTTP Method: **GET**

Required: questionId (string)

http://localhost:8003/6357724ea9bdebe04481af38

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**

200	OK	http://localhost:8003/6357724ea9bdebe04481af38	The question object with its predefined key value pairs.	{
				... (truncated)
				"_doc": {

				"_id": "6357724ea9bdebe04481af38", "title": "Two Sum", "difficulty": "EASY", "description": "Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.\n\nYou may assume that each input would have exactly one solution, and you may not use the same element twice.\n\nYou can return the answer in any order.", ... (truncated) }
400	BAD REQUEST	http://localhost:8003/question/6357724ea9bdebe04481af30	No question with such an id found.	{ "error": "No question with id 6357724ea9bdebe04481af30 found" }
400	BAD REQUEST	-	No questions found.	{ "error": "No question found, seed questions" }

Get Questions (Difficulty)

Endpoint: http://localhost:8003/difficulty/:difficulty

Brief: This endpoint gets a pair of questions of the difficulty specified.

HTTP Method: **GET**

Required: difficulty (string; EASY | MEDIUM | HARD)

http://localhost:8003/difficulty/EASY

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**

200	OK	http://localhost:8003/difficulty/EASY	A pair of questions of the difficulty specified for.	[
				{
				"_id": "6357724ea9bdebe04481af47",
				"title": "Valid Parentheses",
				"difficulty": "EASY", ... (truncated)
				},
				{
				"_id": "6357724ea9bdebe04481af4c",
				"title": "Climbing Stairs",
				"difficulty": "EASY",
				"description": "You are climbing a staircase. It takes n steps to reach the top.\n\nEach time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?", ... (truncated)
				}
				]

400	BAD REQUEST	http://localhost:8003/difficulty/intermediate	No questions returned as difficulty does not match.	{ "error": "No questions with this difficulty 'intermediate' found"

				}
400	BAD REQUEST	-	No questions found.	{ "error": "No question found, seed questions" }

Get Questions (All)

Endpoint: http://localhost:8003/all

Brief: This endpoint gets all the questions available.

HTTP Method: **GET**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	All questions in the database.	[{ "_id": "6357724ea9bdebe04481af47", "title": "Valid Parentheses", "difficulty": "EASY", ... (truncated) }, { "_id": "6357724ea9bdebe04481af4c", "title": "Climbing Stairs", "difficulty": "EASY", "description": "You are

				climbing a staircase. It takes n steps to reach the top.\n\nEach time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?", ... (truncated) }]
400	BAD REQUEST	-	No questions found.	{ "error": "No question found, seed questions" }

Count Questions

Endpoint: http://localhost:8003/count

Brief: This endpoint gets the number of questions available in the database.

HTTP Method: **GET**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	Number of questions in the database.	10
400	BAD REQUEST	-	No questions found.	{ "error": "No question found, seed questions" }

Count Questions (Difficulty)

Endpoint: http://localhost:8003/count/difficulty

Brief: This endpoint gets the number of questions categorised by difficulty that is in the database.

HTTP Method: **GET**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	Number of questions by difficulty in the database.	{ "EASY": 5, "MEDIUM": 3, "HARD": 2 }
400	BAD REQUEST	-	No questions found.	{ "error": "No question found, seed questions" }

Seed Questions

Endpoint: http://localhost:8003/seed

Brief: This endpoint adds pre-defined questions' data stored as an array of JSON objects in the project directory into the database.

HTTP Method: **POST**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	Returns an array of the question objects.	[{ "title": "Two Sum", "difficulty": "EASY", "description": "Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.\n\nYou may assume that each input would have exactly one solution, and you may not use the same element twice.\n\nYou can return the answer in any order.", }, .. (truncated)]

History Service

Health Check

Endpoint: http://localhost:8005/

Brief: This endpoint verifies that the History Service is accessible at localhost:8005.

HTTP Method: **GET**

Required: None

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	-	A HTML page with a h1 text.	History Service

Get History

Endpoint: http://localhost:8005/:id

Brief: This endpoint gets a history based on specified id.

HTTP Method: **GET**

Required: historyId (string)

http://localhost:8005/63614e44d7e78bc165fafb78

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	http://localhost:8005/63614e44d7e78bc165fafb78	The history object requested for based on id.	{ "_id": "63614e44d7e78bc165fafb78",

				"user": "63614a7ca0994ced0648f420",
				"otherUser": "63614a7ca0994ced0648f420",
				"question": {
				"title": "Merge Two Sorted Lists",
				"difficulty": "EASY",
				"id": "63614a4fc001b8c198ebb77c",
				"_id": "63614e44d7e78bc165fafb79"
				},
				"code": "# Definition for singly-linked list.\n\n# class ListNode:\n\n#\tdef __init__(self, val=0, next=None):\n\n#\t ... (truncated)",
				"language": "PYTHON",
				"chats": [
				{
				"senderId": "63614a7ca0994ced0648f420",
				"senderName": "domlimm",
				"message": "yay",
				"time": "24:39",
				"_id": "63614e44d7e78bc165fafb7a"
				}, ... (truncated)
				],

				"createdAt": "2022-11-01T16:50:12.503Z", "updatedAt": "2022-11-01T16:50:12.503Z", "__v": 0 }

Get History Records

Endpoint: http://localhost:8005/history/:userId

Brief: This endpoint gets all history records of a user based on specified id.

HTTP Method: **GET**

Required: userId (string)

http://localhost:8005/history/6363fe3b405c5cad35addd60

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	http://localhost:8005/history/6363fe3b405c5cad35addd60	All history records of user.	[{ "_id": "6363e6554b1ccfdf94180125", "user": "6363fe3b405c5cad35addd60", "otherUser": "6363fe41405c5cad35addd62", "question": { "title": "Merge Two Sorted Lists", "difficulty": "EASY",

```
        "id":
"63614a4fc001b8c198ebb77
c",
        "_id":
"63614e44d7e78bc165fafb7
9"
      },
      "code": "# Definition for
singly-linked list.\n\n# class
ListNode:\n\n#\tdef
...(truncated)",
      "language": "PYTHON",
      "chats": [
        {
          "senderId":
"6363fe3b405c5cad35addd
60",
          "senderName":
"dom",
          "message": "yay",
          "time": "24:39",
          "_id":
"63614e44d7e78bc165fafb7
a"
        }, ...(truncated)
      ],
      "createdAt":
"2022-11-01T16:50:12.503Z",
      "updatedAt":
"2022-11-01T16:50:12.503Z",
      "__v": 0
    }, ...(truncated)
]
```

Save History

Endpoint: http://localhost:8005/

Brief: This endpoint saves a history record.

HTTP Method: **POST**

Required: history (object)


```
{
  "history": {
    "question": {
      "title": "Merge Two Sorted Lists",
      "difficulty": "EASY",
      "topics": [
        "LINKED LIST",
        "RECURSION"
      ],
      "id": "63614a4fc001b8c198ebb77c"
    },
    "code": "# Definition for singly-linked list.\n\n# class ListNode:\n\n#\tdef __init__(self, val=0, next=None):\n\n#\tself.val = val\n#\tself.next = next\n\nclass Solution:\n\n\tdef mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:\n\t\tooo\n\t\tok",
    "language": "PYTHON",
    "chats": [
      {
        "senderId": "63614a7ca0994ced0648f420",
        "senderName": "domlimm",
        "time": "24:39",
        "message": "yay",
        "isConsecutive": false
      },
      {
        "senderId": "63614a7ca0994ced0648f420",
        "senderName": "domlimm",
        "time": "24:39",
        "message": "lit",
        "isConsecutive": true
      }
    ],
    "users": [
      "63614a7ca0994ced0648f420",
```

```
        "63614a7ca0994ced0648f420"
      ]
    }
}
```


Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	As per the required params example above.	-	{}
400	BAD REQUEST	As per the required params example above but without the question object.	Error is thrown as the question done between interviewer and interviewee cannot be found.	{ "error": "Cannot read properties of undefined (reading 'id')" }

Delete History

Endpoint: http://localhost:8005/history/:userId

Brief: This endpoint deletes all history records of the specified user id.

HTTP Method: **DELETE**

Required: userId (string)

http://localhost:8005/history/6363fe3b405c5cad35addd60

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	http://localhost:8005/history/6363fe3b405c5cad35addd60	-	{}

Get Statistics

Endpoint: http://localhost:8005/stats/:userId

Brief: This endpoint adds a statistic record.

HTTP Method: **GET**

Required: userId (string)

http://localhost:8005/stats/636489217bb6be2f51740127

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	http://localhost:8005/stats/636489217bb6be2f51740127	Statistics record of the specified user id.	{ "_id": "6364892131fee05198eb33c3", "user": "636489217bb6be2f51740127", "completedQuestions": ["6361386be37801ddb2846fc5", "6361386be37801ddb2846fd5"

				]
				
				"completedQuestionsByDifficulty": {
				"EASY": 2
				},
				"languagesUsed": {
				"PYTHON": 0,
				"JAVA": 2,
				"C++": 0,
				"JAVASCRIPT": 0
				},
				
				"completedQuestionsByDay": {
				"307": 2
				},
				"completedTopics": {
				"LINKED LIST": 1,
				"RECURSION": 1,
				"DYNAMIC PROGRAMMING": 1,
				"MATH": 1,
				"MEMOIZATION": 1
				},
				"dailyStreak": 1,
				"longestStreak": 1,
				"__v": 0
				}

Add Statistics

Endpoint: http://localhost:8005/stats/:userId

Brief: This endpoint adds a statistic record.

HTTP Method: **POST**

Required: userId (string)

http://localhost:8005/stats/6364e1542be31303617804bb

Optional: None

Requires Authentication: False

Responses				
Status Code	**Status Name**	**Input**	**Output Desc.**	**Actual Response**
200	OK	http://localhost:8005/stats/6364e1542be31303617804bb	Statistics object created for the user specified by id.	{ "user": "6364e1542be31303617804bb", "completedQuestions": [], "completedQuestionsByDifficulty": {}, "languagesUsed": { "PYTHON": 0, "JAVA": 0, "C++": 0, "JAVASCRIPT": 0 }, "completedQuestionsByDay": {}, "completedTopics": {}, "dailyStreak": 0, "longestStreak": 0, "_id": "6364e28b5d6b3bc5b49d9e4d", "__v": 0 }

Milestone Timelines

Milestone 1: MVP and Project Requirements

Spanning Week 3, 22/08 to Week 6, 18/09

Agenda	Timeline
Miscellaneous	
- Project Planning, Discussion, Management, Requirements	Week 3, 22/08 - 28/08
- Migrated base repository to TypeScript - Set up project configuration e.g., Coding standard, linting, etc.	Week 4, 29/08 - 04/09
- Refactored commonly used utilities out of each microservice	Week 6, 12/09 - 18/09
Frontend	
- Converted React application to use Next.js	Week 4, 29/08 - 04/09
- Created authentication page	Week 5, 05/09 - 11/09
- Created matching page with question difficulty selection - Integrated matching service into frontend - Cleaned up UI before Milestone 1 check	Week 6, 12/09 - 18/09
User Service	
- Initialised with relevant features e.g., Login, register	Week 4, 29/08 - 04/09

- Set up database - Added authentication and authorisation	
Matching Service	
- Set up socket.io on frontend and backend - Added in-memory database	Week 5, 05/09 - 11/09
DevOps	
- Dockerised frontend and microservices	Week 6, 12/09 - 18/09

Milestone 2: Project Progress Check

Spanning Week 7, 19/09 to Week 10, 23/10

Agenda	Timeline
Miscellaneous	
- Project Report	Recess - Week 10
Frontend	
- Created matching context to enhance integration with matching service - Added chat UI - Improved room, collaboration editor UI - Integrated with history service to save interview sessions	Recess, 19/09 - 25/09 Week 10, 17/10 - 23/10
Matching Service	
- Created rooms with socket.io	Recess, 19/09 - 25/09
Collaboration Service	
- Set up service and integrate with frontend	Week 7, 26/09 - 03/10
Question Service	
- Initialised with relevant features e.g., Seeding questions, retrieving questions on different filters - Set up database	Recess, 19/09 - 25/09
History Service	
- Initialised with relevant features e.g., Saving history	Week 9, 10/10 - 16/10

of a session, retrieving a specific history, etc. - Set up database	
Messaging Service	
- Set up socket.io on frontend and backend to allow messaging between interviewer and interviewee	Week 10, 17/10 - 23/10
DevOps	
- Dockerised new microservices i.e., Collaboration, Question, History and Messaging	Recess - Week 10

Final Milestone: Submission, Demo & Presentation

Spanning Week 11, 24/10 to Week 13, 9/11

Agenda	Timeline
Miscellaneous	
- Project Report - Tidy up routes - Bug fixes across all services and necessary refactors - Update README	Week 11 - Week 13 Week 12, 31/10 - 6/11
Frontend	
- Add hero page - Add user dashboard with history and statistics shown - Enhance timer in room - Add language preference for user (front end)	Week 12, 31/10 - 6/11
User Service	
- Add language preference for user - Enforce authentication - Change authentication method to use JWT bearer authorisation header instead of http only cookie	Week 12, 31/10 - 6/11
Question Service	
- Add topics and skills for each Question and update Question schema - Add more language templates for questions - Add endpoint to get count of questions by	Week 11, 24/10 - 30/10 Week 13, 7/11 - 9/11

difficulty	
DevOps	
- Setup production deployment	Week 12, 31/10 - 6/11