Documentation/Report Template

<Peer/Prep>

G16 Project Report

CS3219 Software Engineering Principles and Patterns

Name	Student Number
Aishwarya Nair	A0226586U
Jai Lulla	A0226609B
Lai Hui Qi	A0236474Y
Linda Tom	A0220190X
Huang Nanxi	A0206115W

Source Code:

GitHub - CS3219-AY2324S1/ay2324s1-course-assessment-g16: ay2324s1-course-assessment-g16 created by GitHub Classroom

Table of Contents

Introduction

Background

A technical interview is a common practice used by companies to evaluate an individual's capabilities for placement in technology-related roles. University students typically try to prepare themselves for these interviews by trying various types of coding-related questions available to them both online and offline. While this generally helps equip them with the technical skills required, it can sometimes feel boring and repetitive.

Project Scope

Our platform, PeerPrep aims to provide students with a collaborative space to work on the same question at the same time. When working with another user on a question, we let users communicate with them through chat and call to discuss their ideas. Additionally, we allow them to interact with an AI assistant if they require additional help or want to source more ideas. Overall, we seek to improve the learning journey for users by providing them with a collaborative experience that encourages both a conducive and interactive learning environment.

Product Overall

Product Perspective

By aptly matching users according to their specific requirements, our goal is to develop a platform that can help better prepare students for technical interviews. Prior to collaborating, users get to choose the topic and difficulty level of the questions they want to attempt. They can also indicate the language they want to use and their proficiency as a programmer. According to their specifications, we help match the user with someone else who has the same requirements. In doing so, we hope to help enhance users' programming skills by allowing them to work with someone else who is interested in working on a similar type of problem and is also of the same proficiency.

Additionally, we provide users with a feature that allows them to preview their previous attempts from the past. This could help aid the user with their revision if they wish to review questions that they have attempted before.

Operating System

The application is compatible with Linux (macOS) and Windows operating systems. The front end requires the use of a web browser (i.e. Google Chrome). As for the backend, we used Docker, RabbitMQ etc. which are functional on both systems.

Individual Contributions

Member	Technical Contributions	Non-Technical Contributions

Aishwarya Nair
 	
(Frontend)
	• Developed various Frontend Components including:
	  ◦ Login/Signup Page
	  ◦ Landing Page
	  ▪ Dashboard, user question
	  ◦ User Profile Page
	  ◦ Attempt History Page
	  ◦ Collaboration Page
	  ▪ Code Editor: design, syntax highlighting & syncing
	  ◦ Profile View Component
	  ◦ Admin question view - CRUD operation support
	• Error handling and input validation (questions and user authentication)
	• User experience enhancements: interactions with animations/transitions
	• Integrate Frontend and Backend for all the components
	• Testing and debugging matching and collaboration service
	• Unit testing
		• UI/UX design
		  ◦ Developing high fidelity Figma mockups to:
		  ▪ illustrate visual hierarchy of elements on the screen
		  ▪ multiple possible design alternatives for front end components
		• Project management
		  ◦ Reviewing PRs and maintaining code repository
		• Project report:
		  ◦ High Fidelity Prototypes:
		  ▪ Style guide
		  ▪ High fidelity mockups

Jai Lulla
 	
(Backend + DevOps)
	• Developed Backends for:
	  ◦ Authentication Service
	  ◦ User Service
	  ◦ History Service
	• Setup containerized MongoDB for User Service
	• Setup containerized MySQL Database for History Service
	• Dockerized the relevant services
	• Provided API files for relevant services to Frontend for easy access to Microservices
	• Performed Unit Testing and Integration Testing on relevant Services and API files
	• Performed Code and Service Integration (Shared)
	• DevOps:
	  ◦ Setup CI/CD
	  ▪ Add a workflow for each Service and Frontend
	  ◦ Create Master “docker-compose.yml”
		• Project Report:
		  ◦ Functional Requirements
		  ▪ Authentication Service
		  ▪ User Service
		  ▪ History Service
		  ◦ Non-Functional Requirements
		  ▪ Quality Attributes
		  • Usability
		  • Reliability
		  • Availability
		  • Performance
		  ◦ Features
		  ▪ User Management and Authentication
		  ▪ History
		  ◦ Application Design
		  ▪ Tech Stack
		  ▪ Overall Architecture
		  ▪ Backend Architecture
		  • RESTful API
		  ◦ Design Pattern: MVC
		  ◦ Authentication Service
		  ◦ User Service
		  ◦ History Service
		  ◦ Testing
		  ◦ DevOps
		  ◦ API Documentation
		  ▪ Authentication Service
		  ▪ User Service
		  ▪ History Service
		  ◦ Structuring and Formatting
		• Updated READMEs
		  ◦ Main README
		  ◦ Frontend
		  ◦ User Service
		  ◦ History Service

Lai Hui Qi
 	
(Backend)
	• Develop backends for:
	a. Matching Service
	b. Collaboration Service
	c. Communication Service
	d. GPTService
	• Initiate relevant microservices structures for the responsible services stated
	• Set up MongoDB model and relevant database operation API for relevant microservices
	• Dockerize relevant microservices
	• Write unit test cases for relevant microservices
	• Code refactoring and port reallocating (shared)
	  ◦ Code integration (shared)
		• Project Report:
		  ◦ RabbitMQ
		  ◦ ChatGPT API
		  ◦ Socket.IO
		  ▪ Introduction
		  ◦ WebRTC
		  ▪ Introduction
		  ◦ API Documentation for relevant services:
		  ▪ Matching Service
		  ▪ Collaboration Service
		  ▪ Communication Service
		  ▪ GPTService
		  ◦ Functional Requirement
		  ▪ Matching Service
		  ▪ Collaboration Service
		  ▪ Communication Service
		  ▪ GPTService
		  ◦ Improvement and Enhancement (shared)
		• README.md for relevant services.
		  ◦ Matching Service
		  ◦ Collaboration Service
		  ◦ Communication Service
		  ◦ GPTService

Linda Tom
 	
(Frontend and Backend)
	• Develop Frontend for:
	  ◦ Matching Service
	  ▪ Basic Layout
	  ◦ Collaboration Service
	  ▪ Question View
	  ◦ Question Service
	  ▪ Search bar, filter popup and button, language dropdown menu
	• Develop Backend for:
	  ◦ Question Service
	• Setup Containerized Mongodb for Question Service
	• Unit testing for Question Service
	• Code integration (shared)
		• Project Report:
		  ◦ Introduction
		  ◦ Product Overall
		  ◦ Functional Requirement
		  ▪ Question Service
		  ◦ Features
		  ◦ Sprint Planning
		  ◦ User Activity Flow
		  ◦ Question Service
		  ◦ Reflection
		  ◦ API Documentation
		  ▪ Question Service

Huang Nanxi
 	
(Frontend)
	• Develop Frontends for:
	  ◦ Matching Service
	  ◦ Collaboration Service
	  ▪ Basic Layout
	  ▪ Timer
	  ◦ Communication Service
	  ◦ Text-Based Communication
	  ◦ Audio-Based Communication
	  ◦ GPT Service
	  ◦ Code integration
		• Project Report:
		  ◦ Collaboration Room Timer Ticking Flow
		  ◦ Collaboration Room Timer Start/Stop Flow
		  ◦ Text-Bsed Messaging Flow (Receiving)
		  ◦ Text-Based Messaging Flow (Sending)
		  ◦ Audio-Based Calling Flow (Receiving & Accepting/Rejecting)
		  ◦ Audio-Based Calling Flow (Sending)
		  ◦ Audio-Based Calling Flow (Ending Call)
		  ◦ Feature Description (Collaboration Service, Communication Service, Generative Ai Service)

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.

F1: Authentication Service

ID	Requirement	Priority
FR1.1	The system should allow users to create an account with email ID and password	HIGH
FR1.2	The system should ensure that every account has a unique email ID	HIGH
FR1.3	The system should allow users to login into their accounts via email ID and password	HIGH
FR1.4	The system should allow users to logout of their accounts	HIGH
FR1.5	The Authentication State of a user should persist during refresh of tabs	HIGH
FR1.6	The Authentication State of a user should persist among various open tabs in the same browser	MEDIUM
FR1.7	The system should allow users to reset their password	HIGH
FR1.8	The system should allow users to delete their accounts	HIGH
FR1.9	The system should have not allow logins with invalid credentials	HIGH

F2: User Service

ID	Requirement	Priority
FR2.1	The system should store user related data	HIGH
FR2.2	The system should allow retrieval of stored user data	HIGH
FR2.3	The system should allow user to modify their data	HIGH
FR2.4	The system should allow user to delete their data	HIGH
FR2.5	The system to should have role based segregation of users	LOW
FR2.6	The system to should allow a privileged user to update a non-privileged user to one.	LOW

F3: Matching Service

ID	Requirement	Priority
FR3.1	The system should allow users to select desired criteria of questions including language, proficiency, difficulty, and topic to attempt.	HIGH
FR3.2	The system should match two waiting users who selected the same matching criteria.	HIGH
FR3.3	The system should show loading state to indicate the matching attempt initiated by the user is taking place.	MEDIUM
FR3.4	The system should generate a unique session ID for every successful matches.	HIGH
FR3.5	The system should auto-terminate a match finding attempt if the default time limit of 60 seconds has elapsed.	HIGH
FR3.6	The system should notify the user when a specific match is found.	HIGH
FR3.7	The system should notify the user that no match is found when a match cannot be found within 60 seconds.	MEDIUM
FR3.8	The system should allow a user to cancel a matching attempt during the match finding event.	MEDIUM

F4: Question Service

ID	Requirement	Priority
FR4.1	The system should provide a database for storing questions.	HIGH
FR4.2	The system should allow an admin to add a new question to the repository.	HIGH
FR4.3	The system should allow an admin to edit an existing question in the repository.	HIGH
FR4.5	The system should allow an admin to delete an existing question in the repository.	HIGH
FR4.6	The system should provide an admin with the option to categorise questions.	HIGH
FR4.7	The system should allow a admin to filter questions based on criteria(s).	LOW
FR4.8	The system should allow an admin to search for questions based on keywords.	LOW

F5: Collaboration Service

ID	Requirement	Priority
FR5.1	The system should put the matched users into the same, unique room.	HIGH
FR5.2	The system should launch the collaboration environment including the question panel, code editor, communication service, and GPT service for each of the users.	HIGH
FR5.3	The system should allow users to simultaneously edit on the code editor.	HIGH
FR5.4	The system should update and display every edited code or text in the code editor actively.	HIGH
FR5.5	The system should allow either user to terminate the session.	MEDIUM
FR5.6	The system should allow user rejoin event due to sudden disconnection.	LOW
FR5.7	The system should auto-terminate the collaboration session based on the default time limit set according to the difficulty level if time extension is not requested.	MEDIUM
FR5.8	The system should allow users to extend session duration when it is almost timeout until the default maximum time limit of 2 hours.	LOW
FR5.9	The system should notify the users in the same room for every event happened during the session including user join, time extension and session termination.	MEDIUM

F6: Communication Service

ID	Requirement	Priority
FR6.1	The system should allow text communication between users in the same room.	HIGH
FR6.2	The system should send and display the text message instantly to both users.	HIGH
FR6.3	The system should allow audio communication between users in the same room.	MEDIUM
FR6.4	The system should allow a user to initiate a call to the collaborator.	MEDIUM
FR6.5	The system should allow the called user to accept or reject the audio communication invitation.	MEDIUM
FR6.6	The system should allow either user to terminate the call.	MEDIUM
FR6.7	The system should restore the communication logs in a reconnection event.	LOW

F7: History Service

ID	Requirement	Priority
FR7.1	The system to should allow users to retrieve their past attempts	HIGH
FR7.2	The system should save a users attempt	HIGH

F8: GPT Service

ID	Requirement	Priority
FR8.1	The system should allow user to query the ChatGPT assistant when encountering difficulties in the coding session.	HIGH
FR8.2	The system should display the response from OpenAI server once the reply is captured.	HIGH
FR8.3	The system should restore the query logs for the specific user when the related session is active.	LOW

Non-Functional Requirements

Quality Attributes

The table below illustrates the prioritisation of Quality Attributes

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

Each attribute has been carefully considered against the others, recognizing that improvements in one area may affect another. For instance, enhancing security could impact usability with additional authentication steps, or boosting performance could decrease verifiability due to less comprehensive logging. Our current scores and tradeoffs reflect a balance suited for the academic environment of CS3219, with an eye towards future adjustments should PeerPrep evolve into a full-fledged commercial offering.

Based on our scores, the main attributes are Usability, Reliability, Availability, and Performance

Usability

Usability is a cornerstone of our application's design philosophy. To ensure an engaging and frictionless user experience, we have embraced the '3 Click Rule,' diligently crafting the user journey so that no feature or piece of information is ever more than three clicks away. From the initial sign-up process to the final step of matching with a peer, each interaction is streamlined for simplicity. Our commitment to intuitive navigation and straightforward processes minimizes the onboarding learning curve, facilitating quick user adoption and satisfaction.

Reliability

Reliability is a critical focus for our PeerPrep platform, as it underpins the trust our users place in our system. We strive to provide a robust service where each feature operates as expected, every time. From the accuracy and consistency of the resources provided to the synchronization of real-time activities between users, we leave no stone unturned in ensuring that the system behaves predictably during user interactions.

Availability

Availability stands as a fundamental pillar in the design of PeerPrep, acknowledging that our users – diligent learners and avid coders – often operate outside conventional hours, including late nights. To accommodate their commitment, we've architected our platform to be reliably at their service, adhering to the '5 9's' rule of 99.999% uptime. This level of availability translates to less than six minutes of downtime per year, ensuring that our users have access to PeerPrep's resources whenever they need them, without interruption.

Performance

Performance is paramount in PeerPrep, especially given its collaborative nature. We have tried to optimize our platform to eliminate any performance bottlenecks. With features like a live code editor, real-time messaging, and audio calls, we understand the critical need for responsiveness and speed. To meet this end, we've ensured that interactions within the platform are seamless and virtually instantaneous. This commitment to performance excellence is designed to support the fluidity and dynamism of live collaboration, providing users with a consistently snappy and reliable experience.

Features

Service	Feature	Description
Matching	Difficulty, Language, Proficiency and Topic selection	The Matching Service in Peer-Prep application is designed to help match users according to four parameters. It provides users with the option to choose the level of difficulty, the topic for the question, the language they want to program in and their proficiency in programming. If they have no preference, we provide users with the flexibility to choose “No Preferences” as an option. According to their selected options, we match the user with someone else who has the same requirements as them.
Collaboration	Synchronised Timer, Code Editor, Customised Question Display, Attempt History Saving	The Collaboration Service in Peer-Prep application is designed to facilitate a synchronized coding environment, incorporating several features for an enhanced collaborative experience. The window displays a timer feature that manages session durations, allowing users to extend time as needed. Alongside the timer, a key component developed is the code editor, which offers real-time synchronization for matched users, complete with syntax highlighting and language options. Programming questions, tailored to users' choices from a prior Matching Service, are displayed to guide collaborative efforts. Additionally, the service integrates our Communication Service and Generative AI Service, providing seamless access to messaging, audio calls, and AI-driven assistance within the collaboration room, making it a comprehensive platform for interactive coding sessions.
Communication	Text-based Realtime Communication, Audio-Based Realtime Communication	The Communication Service in our *Peer-Prep* is a dynamic feature enabling users to interact in real-time through text-based messaging and audio-based calling. Leveraging WebSocket for instant message delivery and WebRTC for high-quality audio streams, it provides a seamless and integrated communication experience. This service is essential for effective collaboration within the application, allowing clients to effortlessly chat and conduct audio calls, enhancing the overall user engagement and interactivity.
GenerativeAI	GPT-3.5 services	The Generative AI Service in our Peer-Prep application integrates GPT-3.5, providing users with an interactive AI chat experience. Developed using React, it features a responsive chat interface where users can input queries and receive AI-generated responses in real-time. The backend, built on Node.js, communicates with OpenAI's GPT-3.5 model to process and return conversational responses, while also managing session caches for a continuous and contextual user experience.
User Management and Authentication	Authentication and State Management of a user	The User Management and Authentication service is a critical component of our platform, offering robust and secure mechanisms for user registration and login. Leveraging the well-established security infrastructure of Firebase, we ensure that our users' credentials are managed with the utmost care. We prioritize the confidentiality of user data, allowing only verified and registered individuals to access and utilize the platform's features. This approach reinforces our commitment to providing a secure environment where users can confidently engage with our services.
History	Storage and retrieval of a user’s question attempt history	The History Service is a key feature designed to enable users to reflect on their learning journey by accessing their previous question attempts. This service meticulously records each session, including details of the questions tackled and the peers they were paired with. It ensures that users can conveniently revisit their progress and peer interactions through a dedicated History tab on their Dashboard. This persistent access to past attempts not only fosters self-assessment but also encourages continuous learning and improvement.
Question	Storage and retrieval of questions	The Question Service is a core feature included to provide users with questions to attempt. For admins, it provides a way to view, add, edit and delete questions. To further support an admin’s use of the question service, the filtering and search functionality is provided for them to be able to easily find and identify questions that fall under their requirements. For regular users, we ensure that they are able to view questions with the associated tags (i.e. difficulty, topic) . This helps them better gauge and understand the nature of the questions that appear.

Sprint Planning

Confluence

We adopted Confluence as a project management tool. Confluence is well known for its abilities in documentation and collaboration. We chose to use it to be able to more easily create, share, and work on project-related documents.

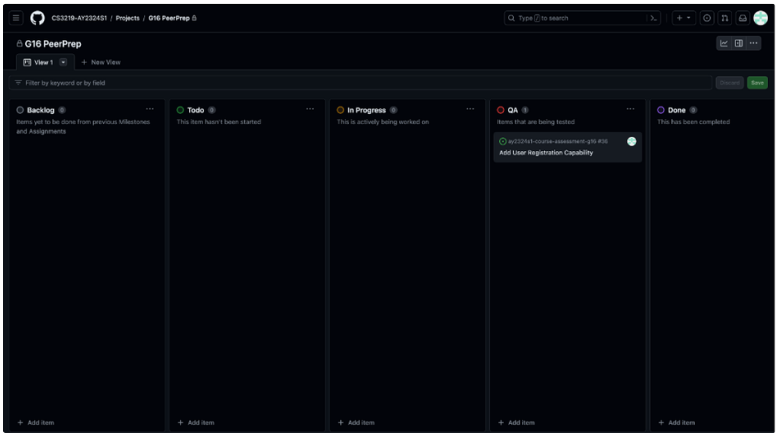
Sprint Review: Every week when we meet, we discuss the tasks we have done so that everyone is well informed about the progress we have made.

Sprint Planning: In the same meeting, we also discussed on what we plan to do for the next sprint. The team reviews the list of tasks we have left and allocates the tasks to different members.

Overall, by documenting our progress on Confluence, we were able to reflect on and improve our planning process for the next sprint.

Github Project

In our agile development process, we adopted GitHub Project issues to streamline our sprint planning. By being able to identify and track the tasks that we have, it was easier to understand the status of the tasks assigned to each member.



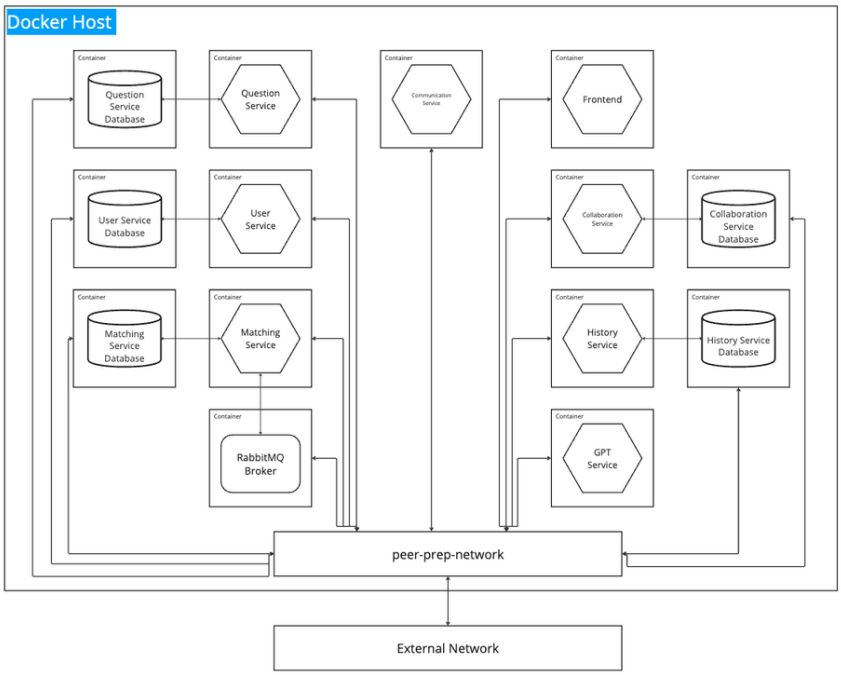
Application Design

Tech Stack

Technology	Used For	Rationale
React.js	Frontend	• Implements a virtual DOM to optimize rendering performance, crucial for dynamic user interfaces. • Offers a flexible development model that doesn't restrict architectural or design patterns, allowing for bespoke solutions. • Facilitates an organized codebase with modular components, improving code reuse and maintainability. • Robust tools like Redux for state management and React Router for navigation, enhance app functionality.
Express.js	All Microservices except Authentication Service	• Provides a lightweight framework for building efficient and scalable server-side applications in Node.js. • Supports a vast range of middleware. • Streamlines the creation of RESTful services with its routing and request handling capabilities.
MongoDB	User Service, Matching Service, Collaboration Service, Question Service	• Offers flexible data schema, making it suitable for the varying data types handled by different services. • Delivers high performance on large volumes of data.
mySQL	History Service	• Ensures data integrity and supports complex queries, which are essential for maintaining historical records and retrieving them efficiently.
Firebase	Authentication Service	• Provides a robust, out-of-the-box authentication solution that supports social logins and secure user management. • Integrates seamlessly with other Google Cloud services and supports cloud functions for extensible backend logic.
RabbitMQ	Matching Service	• Enables reliable message queuing with advanced message routing capabilities, vital for handling asynchronous matching tasks. • Offers a variety of exchange types and routing patterns to precisely control message flow according to the matching logic.
Socket.io	Collaboration and Communication Service	• Ensures real-time, bi-directional communication between clients and server, a necessity for live collaboration features. • Features automatic reconnection support, ensuring continuous service even in the face of transient network issues.
Docker	Deployment	• Offers an isolated environment for consistent deployment, crucial for microservices architecture to run smoothly across different environments. • Able to run frontend, all backend microservices, rabbitMQ and databases with a single command.
Jest	Testing	• Provides a zero-configuration testing framework with built-in mocking and assertion libraries. • Supports parallel test execution, reducing the time required for running extensive test suites.
GitHub Actions	Continuous Integration and Testing	• Easy to setup with GitHub being the code Repository. • Enables the creation of custom workflows for different microservices and frontend, providing granular control over the CI/CD process.
GitHub Issues	Project Management	• Quick and efficient tracking of issues, bugs and tasks within the GitHub repository, streamlining the workflow and collaboration among developers. • Provides a native experience for developers. • Integrates with GitHub's milestone tracking and Project Board allowing for better planning and deadline

		management.
GitHub Projects	Project Management	• Allows for automated project card movements based on triggers like issue closures or pull requests, keeping the project board up-to-date. • Provides a native experience for developers. • Perks of GitHub Integration
Confluence	Project Management	• Acts as a central repository for project documentation, aiding in knowledge sharing and collaboration across the team. • Easy to use. It is developer-friendly and has features like code snippet inclusion, making it ideal for developer documentation.

Overall Architecture



Overall System Architecture

Our application's infrastructure is built on a robust containerized environment using Docker, which encapsulates all our services, databases, and message brokers into discrete, self-contained units. These containers are interconnected through an internal Docker network, named 'peer-prep-network', which facilitates secure and efficient communication between services while isolating them from the external network.

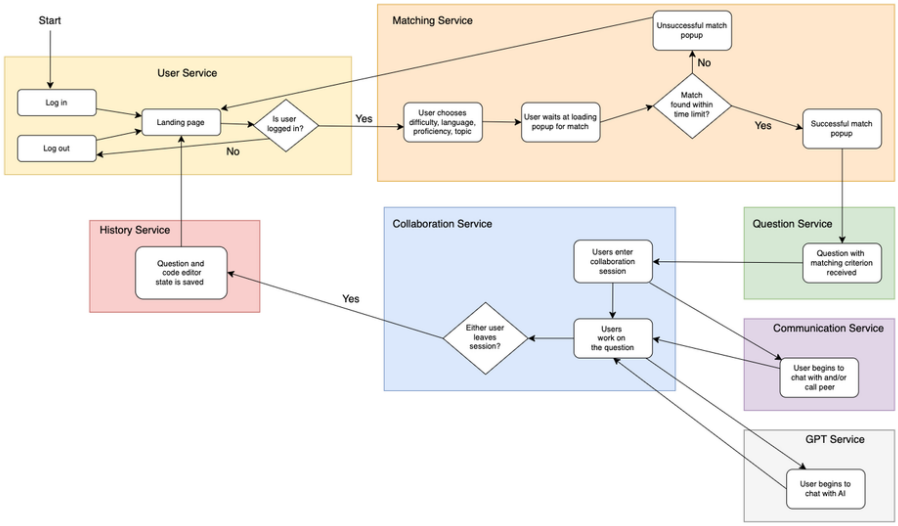
Users interact with the system via the frontend, which serves as the entry point to the entire application. The frontend, itself running within a container, communicates with backend services through well-defined endpoints, ensuring a seamless user experience.

Adhering to the Separation of Concerns principle, our architecture ensures minimal inter-service dependencies, enabling each service to operate independently. This design strategy enhances maintainability and provides a clear delineation of responsibilities across the system.

We've dedicated individual database instances to each microservice requiring persistent data storage. While MongoDB is the chosen database for most services due to its flexibility and performance with document-based data, MySQL powers the History Service. A detailed rationale for this choice is explored in the 'History Service' section of our documentation.

The architecture's adherence to the Separation of Concerns allows us to approach a 'plug and play' model, where services can be individually updated or replaced without disrupting the rest of the system. This not only simplifies updates and maintenance but also positions our platform for easy scaling and integration of new features in the future.

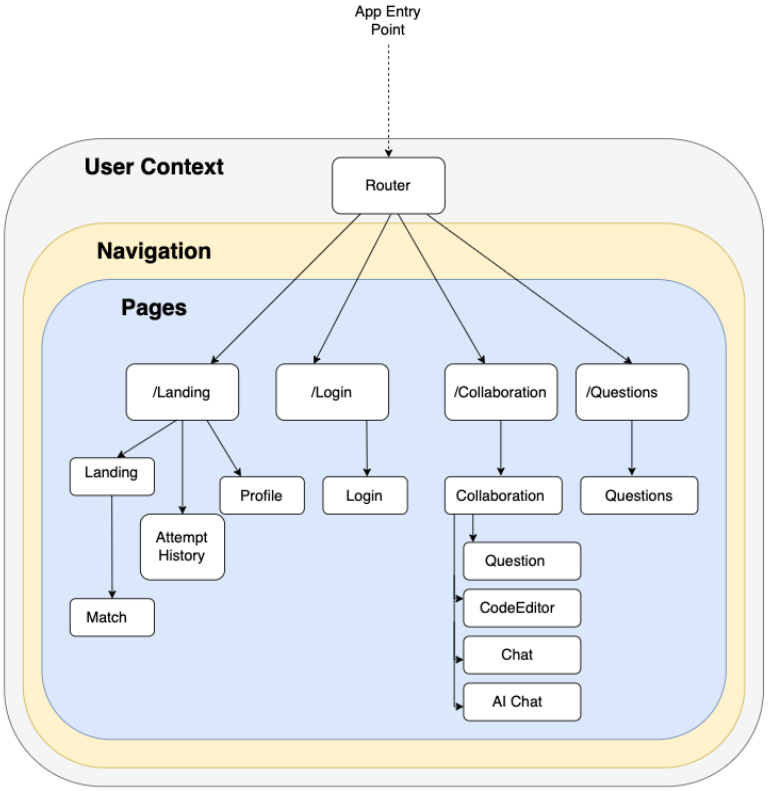
User Activity Flow



User Activity Flow

Frontend Architecture

Component Tree



There are four main pages as follows:

Landing

This is the page where you can find the landing page (dashboard), attempts history, profile, and a button to click on for matching. When a user enters the application, they would be mainly directed to this page.

Login

As the name suggests, this page is for enabling users to log in and sign up. Depending on whether the user keys in the right credentials, they should be redirected to the landing page upon successful login.

Collaboration

Collaboration is where the user works together with another peer on a question. This is where the code editor, question view, chat service, and AI assistant can be found.

Questions

This page is for administrators who wish to view, add, modify, and delete questions. When they arrive at this page, they will see all the questions retrieved from the database.

Design Patterns

Our application's user interface is crafted around familiar design patterns, optimizing for ease of use and intuitive navigation. We embraced a landing page that serves as a central hub, offering tabbed navigation to key areas of the platform. This design pattern is widely recognized and allows users to quickly orient themselves and access various functionalities with ease.

The login and signup pages are streamlined, providing a straightforward path for users to access their accounts or join the platform. We've kept the design minimalistic and focused, reducing cognitive load and guiding users through the authentication process with clear, concise prompts.

In the questions view, we adopted a list format to display available coding problems, drawing inspiration from established coding practice platforms like LeetCode. This approach not only provides familiarity for users but also allows for efficient browsing and selection of challenges.

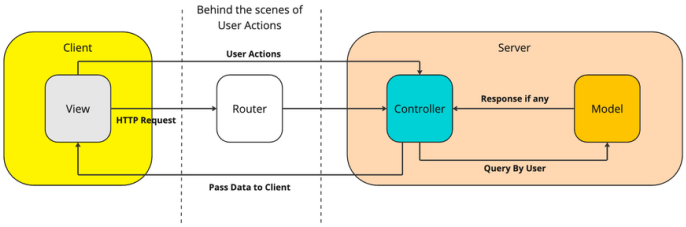
The collaboration window is the centerpiece of our real-time interaction experience. It features an integrated code editor, accompanied by utility buttons such as chat toggles and session controls. The adjacent display of the current question mimics the layout of popular coding interview platforms, fostering a focused environment for coding and discussion. This deliberate layout facilitates an uninterrupted coding flow, akin to an in-person technical interview, but with the advantages of digital convenience.

Overall, our design patterns are chosen to foster a seamless user journey, from initial landing to the collaborative coding sessions, ensuring that users can focus on the content and their learning experience without being hindered by the interface.

Backend Architecture

RESTful API

Design Pattern: Model-View-Controller (MVC)



MVC Pattern adhered to by our system

Overview

Our project is structured around the Model-View-Controller (MVC) design pattern, ensuring a clean separation of concerns. Each microservice is architected as a RESTful API, comprising both Controller and Model layers to handle business logic and data management, respectively.

Frontend

Our frontend architecture boasts a diverse array of views and components, which seamlessly integrate with the microservices' endpoints. This facilitates a smooth, responsive user interface that is adept at executing application functionality.

Controller

The Controller acts as an intermediary between the frontend's requests and the Model's data processing. It is responsible for extracting data from incoming requests and orchestrating the subsequent business logic execution within the Model.

Model

The Model layer contains essential processing logic and database operations. It is designed to encapsulate data manipulation, ensuring that business rules are applied before data is sent back through the Controller.

View

Independently managed, the View layer interacts with the backend via router directives. User actions on the frontend trigger HTTP requests to specific endpoints. These requests are routed through Controllers, which in turn engage the appropriate Model functions to fulfill the request. The resulting data flows back to the View, enabling users to see the outcome of their interactions.

Benefits

Embracing the MVC pattern has brought modularity to the forefront of our project. It allows for the addition of new views, controllers, and services with minimal impact on existing components, fostering an environment conducive to scalability and ease of maintenance. This modularity not only streamlines development but also enhances our ability to extend and improve the application over time.

Authentication Service

In considering our authentication strategy, we opted for the Firebase Authentication JS SDK and client-side route protections. Our decision was driven by scalability and maintenance considerations, as implementing token verification middleware in each microservice would introduce additional complexity and potential latency. Moreover, Firebase Authentication provides the flexibility to seamlessly integrate various authentication providers, such as GitHub, Google, and Apple, enabling us to easily extend our login capabilities and offer users multiple authentication options. Such an approach, while secure, ensures we maintain the responsiveness of our platform, which is integral to the user experience of PeerPrep.

Moreover, the decentralized enforcement of Auth Tokens poses a heightened security risk; any lapse in middleware implementation could lead to vulnerabilities. As PeerPrep evolves, ensuring the consistent application of security checks across an expanding suite of services would become increasingly challenging and error-prone. By leveraging client-side SDK for authentication, we establish a streamlined, consistent security protocol that enhances development efficiency and positions us well for future scalability without compromising on security.

Some alternatives we considered included Firebase Admin SDK and Custom Authentication Setup using MongoDB.

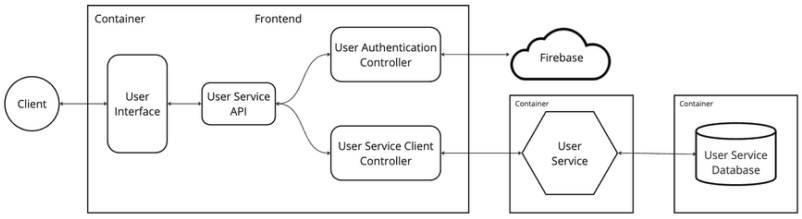
Why are we calling it Authentication Service?

In the context of our application, the use of Firebase's client-side SDK for user authentication is aptly characterized as leveraging an Authentication Service. This designation is rooted in the service-oriented architecture of our application, particularly the integration of a dedicated User service. Firebase, in this capacity, acts as a cloud-based provider, handling the intricacies of user authentication processes, including the secure storage and management of user credentials, and session management. This clear functional delineation not only enhances the understanding and management of our system's components but also brings scalability and ease of maintenance. As our application's user base expands, Firebase's robust infrastructure can accommodate growing authentication demands without necessitating substantial modifications from our end. Additionally, Firebase's managed service nature ensures adherence to security standards and compliance requirements, critical in authentication services. Thus, referring to Firebase's role within our system as an Authentication Service is not only accurate but also aligns with modern application architecture principles, ensuring scalability, security, and efficient system maintenance.

User Service

The User Service is responsible for managing user data and interacts with a MongoDB database. It offers RESTful endpoints for the essential CRUD operations, allowing for the creation, retrieval, updating, and deletion of user profiles. Role-based access control is a cornerstone of the system, clearly differentiating between non-admin users and admins. Admins possess higher privileges, which include the ability to modify user roles and access admin-specific routes on the frontend. New users are automatically designated as non-admins, with administrative rights being assigned through a controlled elevation process.

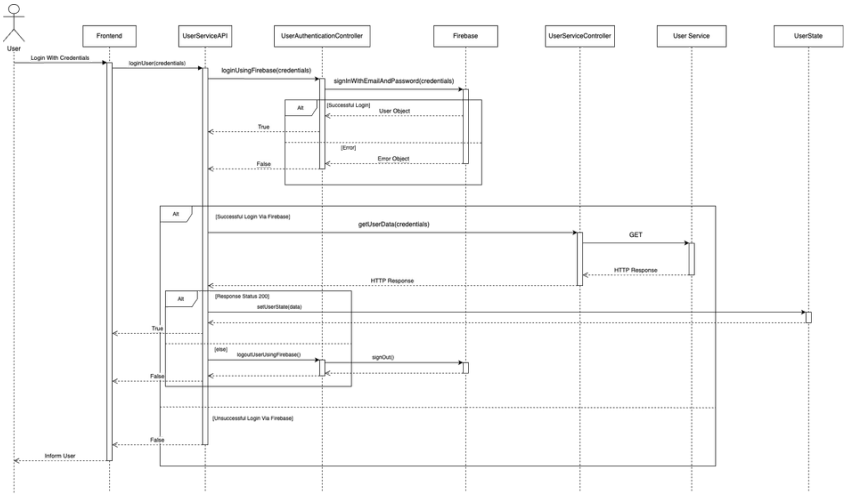
To ensure secure interactions with the User Service, client-side authentication is mandated, thereby permitting only authenticated users to execute operations. This security measure is crucial for preventing unauthorized access and operations within the platform. Authenticated user state is preserved on the frontend, leveraging Firebase Authentication's robust security features. This preservation of state enhances system efficiency by negating the need for repeated user data queries, provided the user remains authenticated.



Component Architecture Diagram of User Service: Containerized Deployment with Firebase Integration

User Login Flow

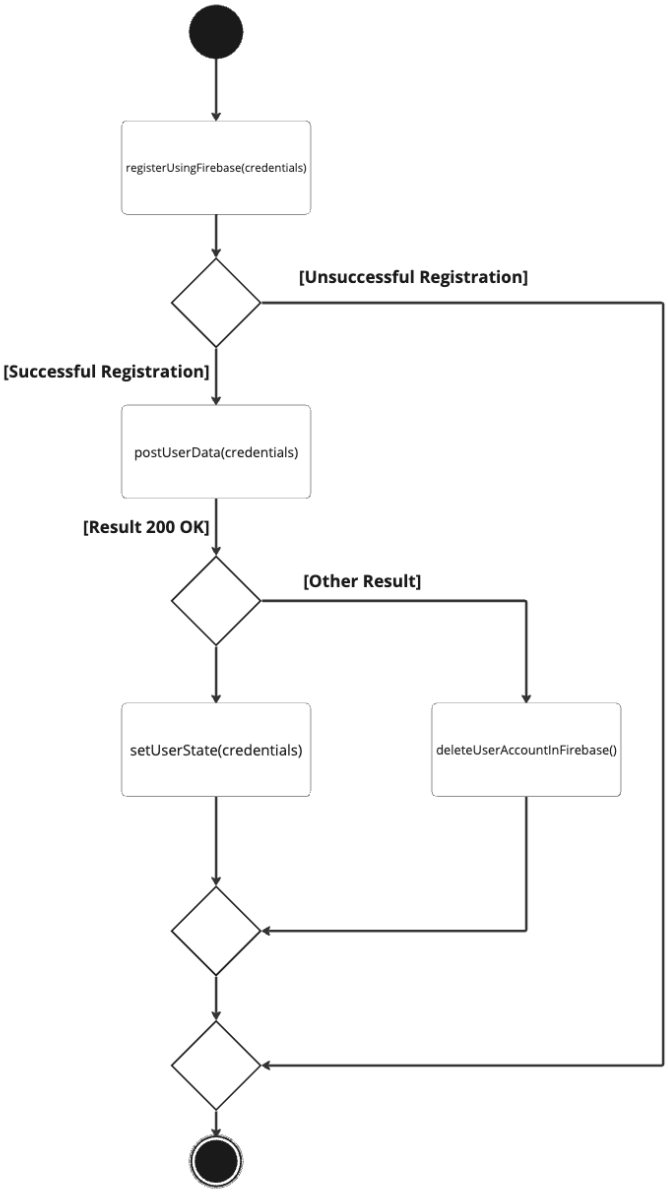
The detailed sequence diagram below shows the events involved in logging in a user in our system.



User Login Flow

User Registration Flow

The high level activity diagram below shows the events involved in the registration of a user in our system.



High Level Activity Diagram: Showing a User's Registration Flow in our System

Question Service

The Question Service primarily manages adding, modifying, and deleting questions. Additionally, it supports filtering and searching based on a set of criteria.

In the front end, the react components are modular since they each manage specific aspects of the UI. As for the backend, the structure adheres to the Model View Controller pattern as it separates the routes controllers and models. The frontend communicates with the RESTful backend API to execute CRUD operations on questions.

To encourage separation of concerns, the front end and back end of the application interact via HTTP requests. Questions are added, fetched, edited, and deleted using these requests. Filtering is implemented by making use of backend API, which provides support for query parameters. Clients can send GET requests to `/questions` with parameters like `?complexity=Easy` allowing them to dynamically tailor their queries. On the other hand, searching is performed locally on the fetched data in the front end. The decision to keep the search on the front end is to enhance responsiveness by providing users with a quick and interactive search functionality where they don't need to click on a search button to dynamically search for questions.

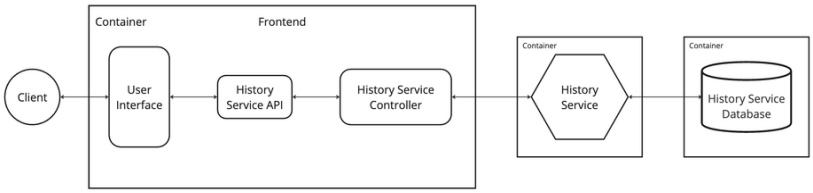
History Service

The History Service plays a crucial role in recording the attempt of question between a pair of users matched by the Matching Service. It stores essential identifiers, such as user IDs of both participants, session ID, question ID, question related data, and the timestamp of the attempt. The choice to store only necessary IDs is deliberate to reduce data redundancy; this is particularly important as the History Service is frequently engaged for both storing and retrieving records, where computational efficiency is paramount.

Our design choice to store question data within the History Service stems from the need for data integrity over time. Questions may undergo edits or deletions within the Question Database; therefore, relying solely on question IDs could result in the inability to retrieve the original content of a question as it was during an attempt. By storing the question data, we ensure that users can always access the exact question they attempted, preserving the context of their experience.

Opting for a SQL database, specifically MySQL, is a strategic decision underpinned by several factors. MySQL excels in read-heavy environments, which aligns with our service's operational profile where read operations are more common than writes. Additionally, the data structure involved is uncomplicated, contrasting with the more complex data handled by services like the Collaboration Service. SQL databases also offer efficient querying capabilities, which we leverage for operations such as retrieving a user's history based on user ID, where either of the user IDs involved may fulfill the query condition.

This service is not only functional but also enhances the user experience. Through the History tab on the frontend, users can review their past question attempts. This retrospective insight serves as motivation and encourages continuous engagement with the platform.



Component Architecture Diagram of History Service: Containerized Deployment

```
const createTableResult = await pool.query(
  `
  CREATE TABLE IF NOT EXISTS ${TABLE_NAME}
  (historyId integer PRIMARY KEY AUTO_INCREMENT,
  userId1 VARCHAR(255) NOT NULL,
  userId2 VARCHAR(255) NOT NULL,
  sessionId VARCHAR(255) NOT NULL,
  questionId VARCHAR(255) NOT NULL,
  questionTitle TEXT NOT NULL,
  questionDescription TEXT NOT NULL,
  questionCategory TEXT NOT NULL,
  questionComplexity TEXT NOT NULL,
  attempt_time TIMESTAMP NOT NULL DEFAULT NOW());
  `
);
```

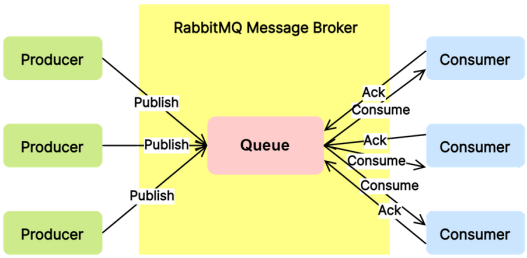
SQL Schema Definition for the History Service: Capturing User Attempt Records

RabbitMQ

Introduction

RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol (AMQP). The RabbitMQ API is used as a support to the Matching Service backend to perform a formal matching queue logic that handle different matching requests.

Design Pattern: Produce - Consume



Simple model for RabbitMQ Message Broker

The producer-consumer pattern in RabbitMQ with single queue is used to accept the incoming matching request (producer publish) and broadcast the new incoming requests to the existing consumers. Since the matching event between users should be reliability and scalability, RabbitMQ is a better choice to fulfill these requirement with a good failure handle mechanism and high system performance on high workload. Furthermore, the message queue and message acknowledgment feature in RabbitMQ also ease the usage of enqueue and dequeue matching requests.

The key usage of RabbitMQ in matching service is the matching queue. The matching request is published into the matching queue first once received, after checking it does not have any current matching pair. Then, it waits for 5 seconds to be consumed by existing consumers. If matched pair is found instantly, the paired users will be directed to the `collaboration window`. Hence, the minimum matching time is around 5 seconds. After that, if it does not match with current consumers, it will play the role as a consumer, and starts to consume new incoming requests. When a matched requests is consumed, the consumer will acknowledge the queue on receiving the request and add the relevant matched pair data into the database. The maximum time for the matching is set to 60 seconds currently, the requests in queue are set to timeout if they are not consumed after the time limit.

Matching Service

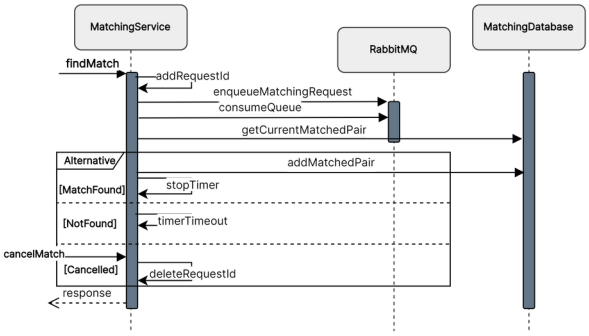
The matching service initiates a temporary pairing between two users. It provides access to the services using RESTful API while the matching logic is governed by RabbitMQ, and the matched pair data is stored using MongoDB.

The `findMatch` function can be called by POST-ing a request with the user Id and the preferred matching criteria (language, proficiency, difficulty, topic) **(FR3.1)** in the request body. Then, the user will then be enqueued into the RabbitMQ channel with its matching criteria, until a collaborator with same preferences is found or arrived to the queue and paired with the user that matches the criteria exactly **(FR3.2)**.

A timer is set to complete matching within 60 seconds. In addition, an interval is set to check the user's request against the database records and the cancel event periodically to provide a more accurate matching outcome.

At the end of a successful matching, the matched pair information including the unique session Id **(FR3.4)**, collaborator Id, question Id, and original request will be sent back as the response **(FR3.6)**. Otherwise, when the matching timeout **(FR3.5)** or encounter error **(FR3.7)**, it will response with match-failed or error-handling message.

In the meantime, we provide a `cancelMatch` DELETE-method to enable user to cancel the matching request **(FR3.8)** immediately only before a collaborator is found. The user should execute the cancel match at least 5 seconds from requesting the match. The user Id will be removed from the `availabilityCache` and the matching ID (created for each matching event) is added into the `isCancelled` array to prevent error matching.



Sequence Diagram : Interactions happen within findMatch

The matched pair data will be stored in MongoDB to allow data usage for other services. For example, the collaboration service make use of `getMatchSession` or `getSession` which are GET-methods provided to retrieve a live session for a user, so that the system can verify if the session exists before establishing it. Also, a DELETE-request of `end` is provided to mark the matched pair as end.

Socket IO

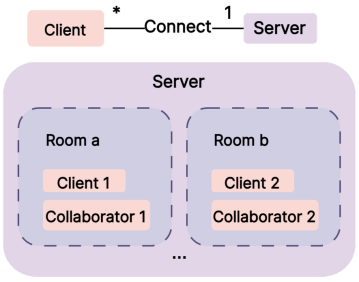
Introduction

Socket.IO is a bi-directional and low latency communication used in our text-based collaboration and communication microservices. It involves both frontend and backend utilizations to establish the communication between clients and server. The event-based communication is implemented using several complement event emitters and event listeners from both sides. It is an important API that is involved in a large portion of our project design due to its efficiency and convenience on implementation.

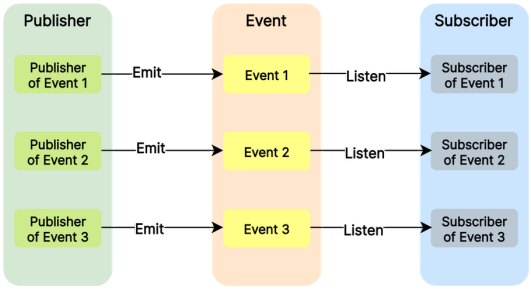
Design Pattern: Publish - Subscribe

Both client side socket and server side socket are set up to facilitate the publishing (emitting) and subscribing (listening) events which in turns are the publisher and subscriber. There are two Socket.IO usage in our project, which include the code editor syncing in collaboration service and the text-based communication in communication service. Due to the implementation logic of our microservices, there will only be at most two users that connect to the same room at the same time. We however appreciate the reliability and high performance that Socket.IO brings to us.

The design pattern of Socket.IO is as shown. The publisher is the event emitter, while the subscriber(s) will have a complement event listener to handle the captured event. Both client and server can act as the publisher and subscriber.



A single server serves all the connected clients by assigning them into their respective rooms.



Relation: Publishers emit events while subscribers listen to emitted events.

We integrate Socket.IO into the collaboration service to sync the code editor for both collaborating users. The publish-subscribe pattern provided is suitable to transfer code input and session information between the clients through the services provided by the server. By the same concept, we also use Socket.IO API in our text-based communication channel to transfer the text messages through the communication server. To illustrate our main idea, the client emit a change (event) to the server, the server listen to the emitted event and then response by emit a new event to the selected subscriber. The server will also emit event to clients for some management purpose, such as timer initiation and session termination. Overall, Socket.IO provides reliable and high quality service that meet our expectation on the speedy communication.

Collaboration Service

Collaboration Room Code Editor Flow

Collaboration Room Timer Ticking Flow

The timer ticking flow is handled through frontend logic and backend coordination. This process ensures the countdown of the collaboration session's timer is displayed and updated in real-time. The core of this functionality resides within a `useEffect` hook in the frontend, where an interval is set to decrement the `timeRemaining` state by one second at each interval tick.

This countdown mechanism activates once a session starts, indicated by the `sessionStarted` state turning *true*. The interval, set using *setInterval*, calls a function every 1000 milliseconds (1 second) that updates the `timeRemaining` state. This state reduction reflects the passage of time in the session. To ensure the countdown's accuracy, the initial time is fetched from the backend when the session begins, aligning the frontend timer with the server's.

If the `timeRemaining` reaches zero, the frontend triggers a popup to inform clients that the session time has elapsed, offering options like extending the session or ending it. This is crucial for maintaining user engagement and providing a clear indication of the session's progress.

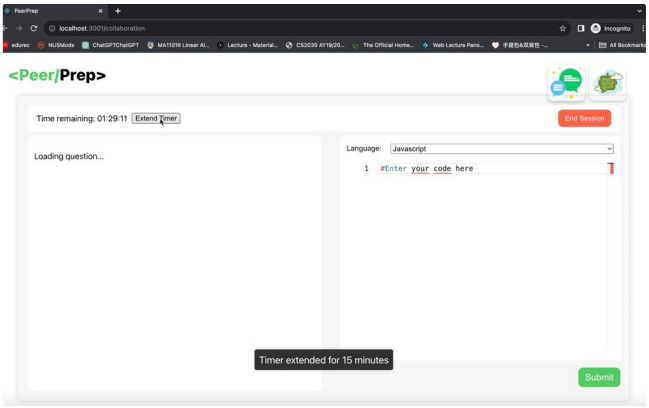
Moreover, the `useEffect` hook includes cleanup logic to clear the interval when the component unmounts or the session ends, preventing any memory leaks and ensuring the timer stops when no longer needed. This ticking flow of the timer is a blend of React's state management and JavaScript's time functions, offering a real-time, responsive, and user-friendly countdown experience in the collaborative session.

Collaboration Room Timer Start / Stop Flow

The timer's start/stop flow is managed primarily through the frontend with real-time backend interactions. The timer initiates when a user successfully joins a collaboration session, as indicated by the *sessionStarted* state variable being set to `true` within the `useEffect` hook upon receiving a `join` event from the `socket.io-client`. This initiation is closely tied to the session's unique ID (`sessionId`) and the client's ID (`userId`), ensuring that the timer is specific to each session. The duration of the timer is dynamically set based on the session's difficulty level, retrieved from the backend and adjusted in real-time for accuracy.

To stop the timer, the frontend listens for specific socket events like `system-terminate` or `user-terminate`. On receiving these events, the application triggers navigation away from the session, effectively stopping the timer and concluding the session. The `handleEndSession` function encapsulates this logic, sending a termination signal to the backend and then redirecting users, ensuring a seamless transition out of the collaborative environment.

Collaboration Room Timer Extension Flow



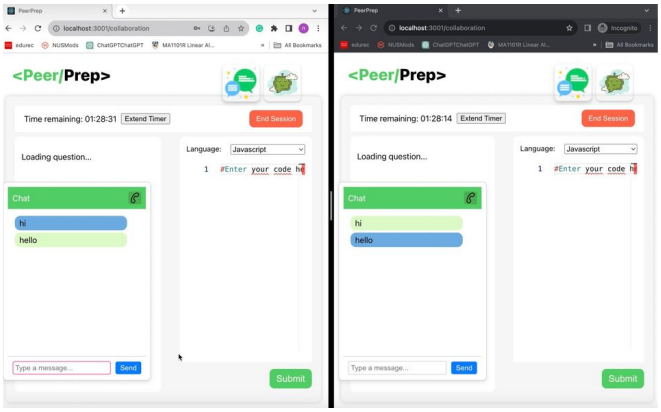
Timer Extension for Collaboration Session

The timer extension flow is managed through frontend interactions and backend synchronization. This feature activates when one of the participants elects to extend the session by clicking the `Extend Timer` button. The frontend, upon this interaction, triggers the `handleExtendTimer` function, which emits an `extend-time` event to the backend via `socket.io-client`. This event signals the backend to augment the current session's timer by a pre-set duration of 15 minutes.

Once the backend processes this extension request, it communicates the new total time remaining back to all session participants through a `time-extended` event. The frontend, upon receiving this update, utilizes the `setTimeRemaining` function to update the session's timer display, ensuring that both the participants see the adjusted session duration in real-time. This mechanism, embedded within the React component's `useEffect` hook, guarantees that the timer reflects the new duration promptly and accurately across all client interfaces.

Communication Service

Text-Based Messaging Flow (Receiving)



Text-Based Communication during Collaboration Session

The text-based messaging receiving process is handled through a combination of React's frontend capabilities and WebSocket communication via `socket.io-client`. This flow allows users to receive messages in real-time during a collaboration session.

When a user is part of a collaboration session, the frontend establishes a WebSocket connection to the server using `socket.io-client`. This connection is maintained throughout the session, listening for incoming messages. The `useEffect` hook in the CommunicationWindow component initializes this connection, setting up event listeners for various message types, including text-based messages.

Upon receiving a new text message, identified by the `new-message` event from the server, the frontend updates the `messages` state array. This update is performed using the `setMessages` function, which appends the new message to the existing chat history. Each message is rendered in the chat window, differentiated by whether it originated from the client themselves (`fromSelf` property) or another collaborator.

The chat interface provides real-time updates with React's efficient state management and the real-time nature of WebSocket communication.

Text-Based Messaging Flow (Sending)

The text-based messaging sending process is implemented, combining React's frontend development with WebSocket communication facilitated by `socket.io-client`. This flow enables users to send messages to other participants in real-time during a collaborative session.

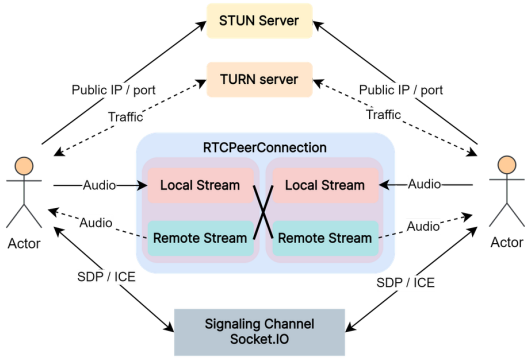
When a user types a message into the chat input field and initiates the send action (either by clicking the send button or pressing Enter), the `handleSendMessage` function is triggered. This function first checks for the presence of a valid WebSocket connection and ensures the message is not just whitespace. If these conditions are met, it constructs a message object containing the text and a flag `fromSelf` to indicate that the message was sent by the user. This object is then emitted to the server with the `message` event.

Concurrently, the local state of the `messages` array is updated using `setMessages`. This update adds the new message to the chat history, which then triggers a re-render of the chat window to display the latest message. This state management ensures the user's chat interface remains synchronized with the actual conversation.

WebRTC

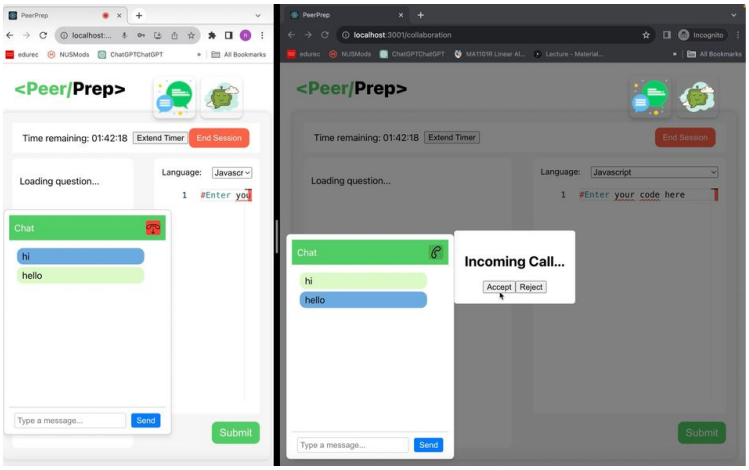
Introduction

WebRTC supports real-time communication for the web, in transferring streams of generic data, audio and video. The Session Traversal Utilities for NAT (STUN) server establishes direct peer-to-peer connection when possible, if public IP and port are stated in clear, while Traversal Using Relays around NAT (TURN) serves as a relay when direct connection is not possible due to network configurations. However, these setups are not mandatory. The WebRTC server facilitates the data flows between connected peers. The local data streams are sent out as remote streams to the receiver. In addition, a signaling channel is set up to confirm the peers' engagement in a communication (start, stop, request and reject).

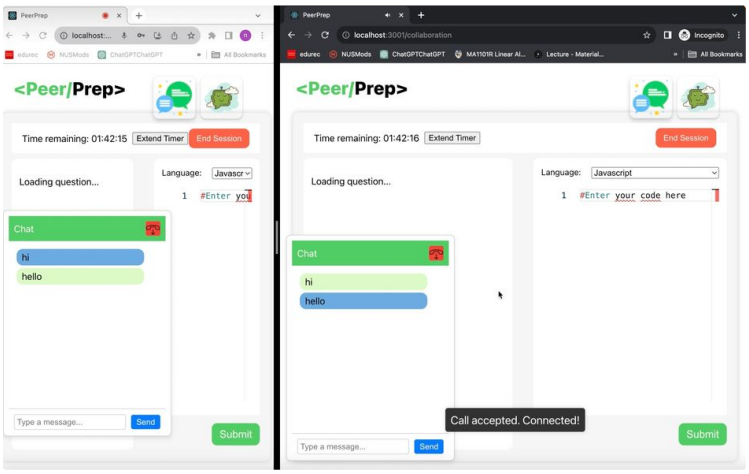


Brief illustration of Audio-based WebRTC communication

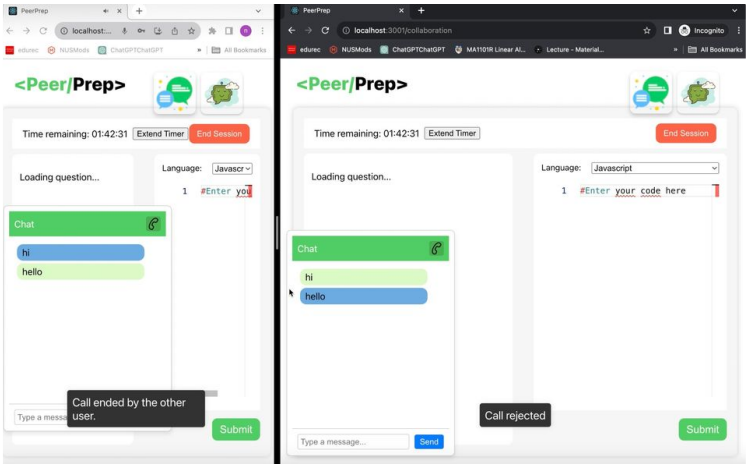
Audio-Based Calling Flow (Receiving & Accepting/Rejecting)



Receiving a Call



Accepting a Call



Rejecting a Call

The audio-based calling flow for receiving, accepting, and rejecting calls is implemented using a combination of WebRTC for real-time audio communication and WebSocket via `socket.io-client` for signaling. This setup ensures users can manage incoming audio calls effectively within the collaborative environment.

Receiving a Call:

When an incoming call is initiated by another user, the frontend's WebSocket connection, established in the *CommunicationWindow* component, listens for a `called` event. This event is triggered when the server receives a call request from another collaborator in the session. Upon receiving this event, the frontend displays an incoming call modal, giving the user the option to accept or reject the call. The modal is controlled by the `showIncomingCallModal` state, which is set to true to make the modal visible.

Accepting a Call:

If the user chooses to accept the call, the `acceptCall` function is executed. This function interacts with the WebRTC `RTCPeerConnection` to create an answer to the received offer. It sets the local description of the peer connection and sends this answer back to the caller through the WebSocket server using the *answer* event. Subsequently, the frontend establishes the audio stream, which is rendered to the user through an audio element in the DOM, represented by `remoteAudioRef`.

Rejecting a Call:

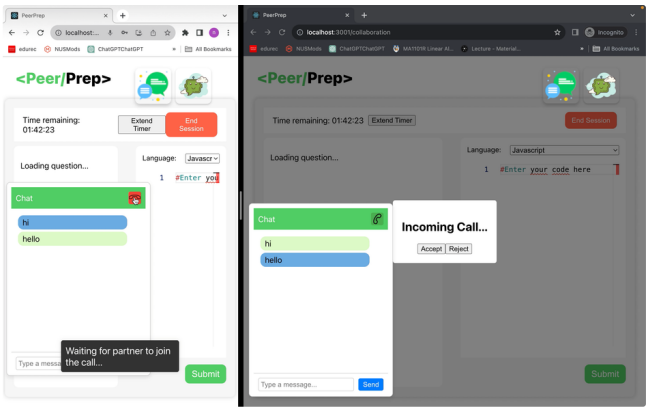
If the user decides to reject the incoming call, the `rejectCall` function is triggered. This function simply informs the server of the rejection by emitting an `end-call` event through the WebSocket connection. The server then notifies the caller that the call has been rejected. The frontend updates its UI accordingly, dismissing the incoming call modal and resetting relevant states.

This implementation demonstrates a sophisticated use of WebRTC for handling real-time audio streams, along with WebSocket for reliable signaling in call management. The integration of these technologies allows for a seamless and interactive user experience, where users can receive, accept, or reject audio calls effectively within the collaborative session.

Audio-Based Calling Flow (Sending)

The audio-based calling flow for sending call requests is implemented, leveraging WebRTC for real-time audio communication along with WebSocket communication for signaling and coordination. This flow allows users to initiate audio calls within the collaborative session environment.

Initiating a Call:



Initiating a Call

When a user decides to start an audio call, the `startCall` function is triggered. This function first sets up a local audio stream using the WebRTC API. It accesses the user's audio input device (like a microphone) and adds this local stream to the `RTCPeerConnection` instance. The function then creates an offer for the audio call, a process involving the negotiation of media capabilities and session parameters.

Creating and Sending the Offer:

After creating the offer, `startCall` sets this offer as the local description of the peer connection. This step is crucial as it defines the local end of the call setup. Once the local description is set, the offer is sent to the server using the WebSocket connection with a `call` event. This action informs the server that the user is attempting to start a call, and the server then relays this offer to the intended recipient in the session.

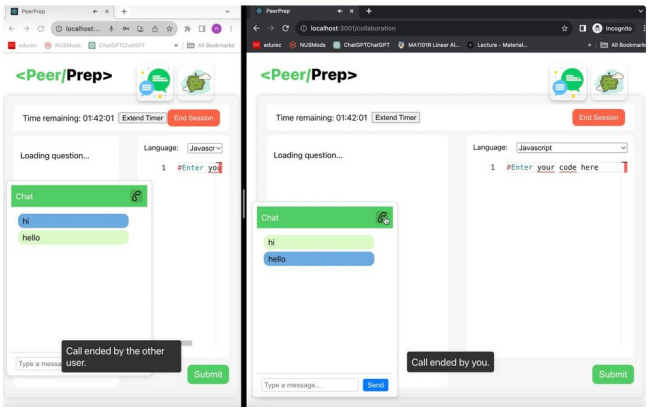
Call Establishment and UI Updates:

During this process, the UI provides feedback to the user, such as indicating the call status (waiting for the partner to join, call in progress, etc.). Additionally, the local audio stream is attached to a DOM element (usually hidden) represented by `localAudioRef` to play the local audio.

Handling Responses:

The `startCall` function also prepares the application to handle responses to the call offer. This involves listening for events like `answered`, which indicates that the call has been accepted by the recipient, and `ice-candidate`, which facilitates the establishment of the peer-to-peer connection necessary for the audio stream.

Audio-Based Calling Flow (End Call)



Ending a Call

The audio-based calling flow for ending a call is designed, utilizing WebRTC for managing the real-time audio stream and WebSocket communication for signaling. This feature allows users to terminate an ongoing audio call within the collaborative session, ensuring control and flexibility in communication.

Initiating Call Termination:

The process of ending a call is initiated by the user through an interactive UI element, typically a `hang-up button`. When this button is clicked, the `endCall` function is executed. This function is responsible for cleanly closing the audio call, ensuring that all resources are properly released and that both collaborators are notified of the call's termination.

Emitting End Call Signal:

Within the `endCall` function, the application emits an `end-call` event via the WebSocket connection. This event signals to the server that the user intends to terminate the call. The server then relays this information to the other collaborator in the call, ensuring both parties are aware that the call is ending.

Handling Media Streams and Peer Connection:

The function also includes logic to stop all media tracks associated with the local audio stream. This is accomplished by iterating over each track in the local stream and invoking the *stop* method on each one. Additionally, the function handles the closing of the `RTCPeerConnection` instance, which is essential for terminating the peer-to-peer connection established during the call. This step is crucial to free up system resources and prevent any potential memory leaks.

UI and State Updates:

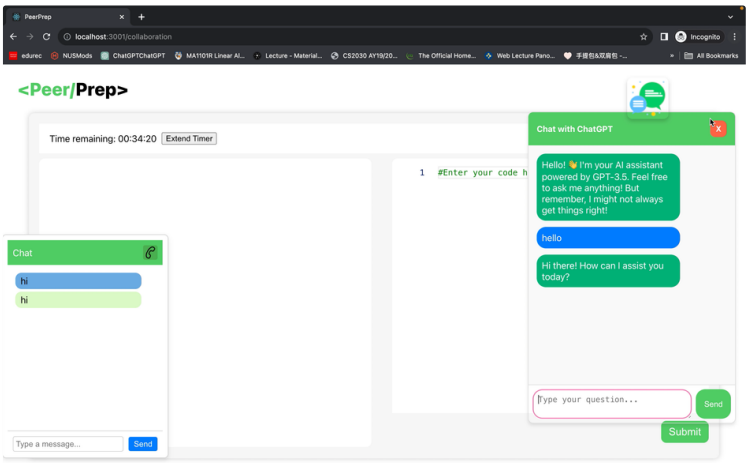
Upon successfully ending the call, the application updates the UI to reflect the change in call status. This typically involves disabling call-related controls and possibly switching the UI back to a pre-call state. The application also updates relevant state variables, such as resetting flags that indicate an ongoing call, to maintain consistency in the application's state.

ChatGPT API

Introduction

ChatGPT API is a portion of the OpenAI API, which is a versatile tool for building interactive and engaging chatbot or virtual assistant. The API provides an interface to communicate with the AI language model, which a question prompt is responded by the AI generated reply. It is an interesting API to be integrated into our generative AI - GPT Service to provide virtual assistance to our user.

GPT Service



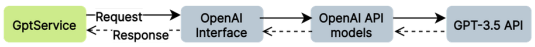
Chatting with GPT-3.5

Communication With OpenAI (ChatGPT) Server

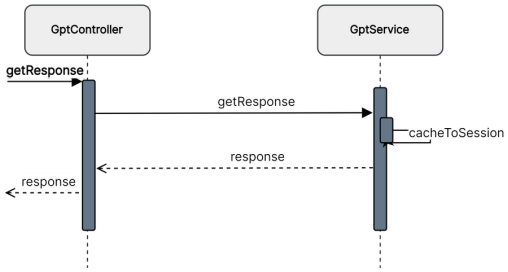
The GPT Service aims to provide an AI chat communication service to user. This service is attached to the collaboration service, which user can make use of it when they face some difficulties on certain concepts or code implementation during collaboration **(FR7.1)**.

There are three entry point to access the GPT microservice using RESTful API. A POST-method with `/generate` is provided to communicate with the GPT-3.5-turbo language model, by having the user Id and question prompt in request body. The `getResponse` function creates a connection to the OpenAI server interface and sends out the user request to be responded by the ChatGPT-3.5 model. The response is then cached in the mapping under the user Id key for current session only and transferred back to the user **(FR7.2)**.

Session chat restoration can be done by using the GET-method `/getCache` by user Id as the parameter **(FR7.3)**. The respective cache will be freed using DELETE-method `exitGpt` when the session ends.



Overview: Request flow from GPT Service to GPT-3.5 API



Sequence diagram: Normal execution of GPT Service

Testing

In the development of Peer Prep, a comprehensive testing strategy was implemented to ensure the quality and reliability of each service. Here's an overview of the testing measures we conducted:

1. **Coverage Metrics**: Each microservice within our application achieved approximately 80% test coverage. This high level of coverage indicates a substantial breadth of our test suites, encompassing a wide range of possible scenarios and edge cases.
2. **Testing Types**: Our testing methodology included both unit tests, which validate individual components in isolation, and integration tests, which ensure that different parts of the application work together as expected. This combination allowed us to verify not only the functionality of discrete units but also the seamless operation of those units within the larger system.
3. **CI/CD Integration**: The integration of our testing suite with the CI/CD pipeline facilitated smoother iterations and deployments. With each push or pull request, our automated tests are executed, enabling us to identify and address issues early in the development cycle.
4. **Primary Testing Framework**: Jest, a widely-adopted JavaScript testing framework, was utilized as the primary tool for writing and running our tests. Its rich feature set and active community support made it an ideal choice for our project.
5. **Auxiliary Libraries**: To enhance our testing capabilities, particularly for database operations, we employed additional libraries such as Mockgoose. These tools allowed us to simulate a MongoDB environment, enabling us to perform database operations in a controlled, mock setup that reflects the live database's behavior without impacting actual data.

DevOps

Development with Docker

Docker provides a consistent development environment across all team members' machines. This means that everyone works with the same OS, libraries, and dependencies, reducing the "it works on my machine" conundrum.

The isolation provided by Docker is a boon for microservices development. Each service runs in its own container, isolated from others. This isolation reduces conflicts between services and simplifies debugging and testing.

For the above reasons, we believe that it is well-suited for microservices because it allows each service to be developed and deployed independently in separate containers, which can then be orchestrated to work in tandem without stepping on each other's toes.

```
FROM node:18-alpine

WORKDIR /app
COPY package.json .
RUN npm install
COPY . .

EXPOSE 3002

CMD ["npm", "start"]
```

Dockerfile for User Service

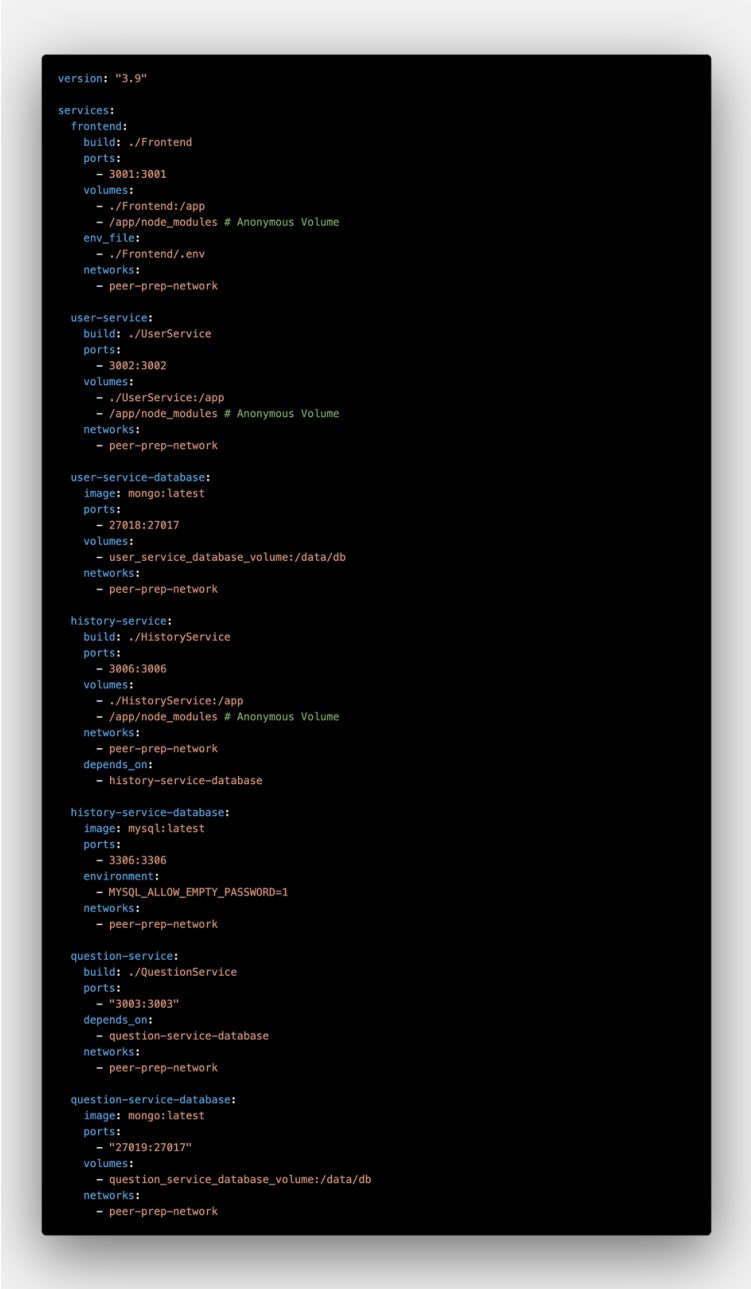
Local Deployment with Docker Compose

Docker Compose streamlines the management of applications composed of multiple microservices, which would otherwise be cumbersome to initiate and run individually. By using a single YAML file to define the various services, networks, and volumes, developers can ensure that their complex multi-container setups mirror production environments closely. With the simple execution of `docker-compose up`, the entire suite of services can be activated simultaneously.

Networking, often a challenge with multiple services, is elegantly managed by Docker Compose, allowing containers to communicate through service names rather than dealing with IP addresses directly. This not only simplifies connectivity but also supports service discovery within the local development network.

As these Docker Compose files are version-controlled, teams can track and roll back environment changes with the same precision as application code, maintaining a robust and consistent development workflow.

Additionally, Docker Compose's volume management capabilities address the persistent storage needs and state management across container restarts. By mapping volumes to our development environment, we ensure persistence of source code changes and dependencies. This setup, in conjunction with tools like Webpack for our React frontend and Nodemon for our Express backend, allows us to utilize hot reloading. Changes in the codebase are immediately reflected in the running containers without the need for a restart, which significantly accelerates our development cycle. The integration of these tools with Docker's volume management creates a seamless and efficient developer experience, streamlining the transition from code modification to testing and validation.



```yaml
version: "3.9"

services:
  frontend:
    build: ./Frontend
    ports:
      - 3001:3001
    volumes:
      - ./Frontend:/app
      - /app/node_modules # Anonymous Volume
    env_file:
      - ./Frontend/.env
    networks:
      - peer-prep-network

  user-service:
    build: ./UserService
    ports:
      - 3002:3002
    volumes:
      - ./UserService:/app
      - /app/node_modules # Anonymous Volume
    networks:
      - peer-prep-network

  user-service-database:
    image: mongo:latest
    ports:
      - 27018:27017
    volumes:
      - user_service_database_volume:/data/db
    networks:
      - peer-prep-network

  history-service:
    build: ./HistoryService
    ports:
      - 3006:3006
    volumes:
      - ./HistoryService:/app
      - /app/node_modules # Anonymous Volume
    networks:
      - peer-prep-network
    depends_on:
      - history-service-database

  history-service-database:
    image: mysql:latest
    ports:
      - 3306:3306
    environment:
      - MYSQL_ALLOW_EMPTY_PASSWORD=1
    networks:
      - peer-prep-network

  question-service:
    build: ./QuestionService
    ports:
      - "3003:3003"
    depends_on:
      - question-service-database
    networks:
      - peer-prep-network

  question-service-database:
    image: mongo:latest
    ports:
      - "27019:27017"
    volumes:
      - question_service_database_volume:/data/db
    networks:
      - peer-prep-network
```

A Code snippet of docker-compose.yml (First Half)

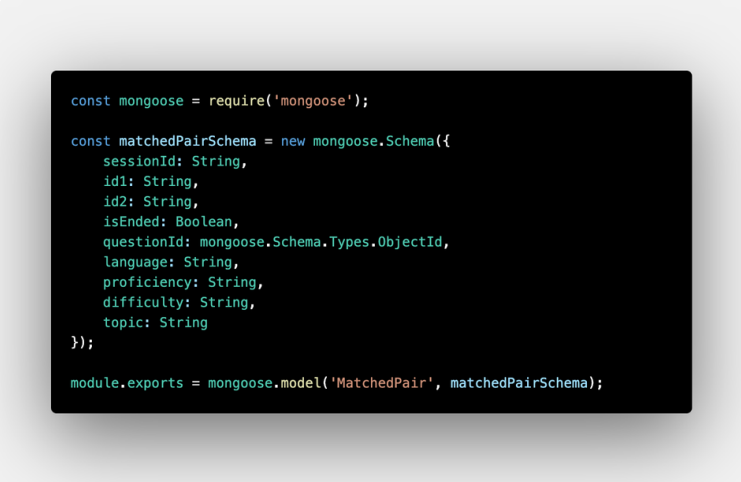
```yaml
matching-service:
    build: ./MatchingService
    ports:
      - "3004:3004"
    depends_on:
      - matching-service-database
      - rabbitmq
    networks:
      - peer-prep-network

  matching-service-database:
    image: mongo:latest
    ports:
      - "27020:27017"
    volumes:
      - matching_service_database_volume:/data/db
    networks:
      - peer-prep-network

  # Rabbitmq service
  rabbitmq:
    image: rabbitmq:latest
    ports:
      # AMQP protocol port
      - "5672:5672"
      # HTTP management UI
      - "15672:15672"
    networks:
      - peer-prep-network

  collaboration-service:
    build: ./CollaborationService
    ports:
      - "3005:3005"
    depends_on:
      - collaboration-service-database
    networks:
      - peer-prep-network

  collaboration-service-database:
    image: mongo:latest
    ports:
      - "27021:27017"
    volumes:
      - collaboration_service_database_volume:/data/db
    networks:
      - peer-prep-network

  communication-service:
    build: ./CommunicationService
    ports:
      - "3007:3007"
    networks:
      - peer-prep-network

  gpt-service:
    build: ./GptService
    ports:
      - "3008:3008"
    env_file:
      - ./GptService/.env
    networks:
      - peer-prep-network

networks:
  peer-prep-network:
    driver: bridge

volumes:
  matching_service_database_volume:
    driver: local

  collaboration_service_database_volume:
    driver: local

  question_service_database_volume:
    driver: local

  user_service_database_volume:
    driver: local
```

A Code snippet of docker-compose.yml (Second Half)

Database Management

Our approach to database management involves the use of locally containerized databases, providing each service with a dedicated data store. We have empowered our services with the flexibility to utilize either MongoDB or MySQL depending on their specific data handling requirements. Leveraging official Docker images, which are both verified and supported, enables us to expedite the development process by quickly setting up consistent and secure database environments.

The architecture wherein each service is coupled with its own database container not only enhances flexibility but also ensures resource allocation is optimized—preventing services from competing for database resources. This design philosophy aligns with our forward-looking infrastructure strategy, which anticipates eventual deployment on orchestration platforms like Kubernetes. Such deployment would fully harness the inherent scalability of containerized databases, efficiently managing user load and dynamically adapting to the demands of the system.



```
const mongoose = require('mongoose');

const matchedPairSchema = new mongoose.Schema({
    sessionId: String,
    id1: String,
    id2: String,
    isEnded: Boolean,
    questionId: mongoose.Schema.Types.ObjectId,
    language: String,
    proficiency: String,
    difficulty: String,
    topic: String
});

module.exports = mongoose.model('MatchedPair', matchedPairSchema);
```

Schema for storing a Matched Pair

CI/CD with GitHub Actions

Our project harnesses GitHub Actions for a comprehensive CI/CD pipeline, tailored individually for each microservice and the frontend. To maintain system integrity, we've established a global workflow that builds all components collectively, guarding against configuration discrepancies.

Resource efficiency is a cornerstone of our CI/CD process. Workflows are conditionally triggered to conserve computational resources, activating only when changes are detected in the service-specific file set.

The workflow sequence across services follows a standardized pattern:

1. **Environment Preparation**: Initialize a Docker environment to ensure a clean, reproducible build context.
2. **Service Assembly**: Employ `docker-compose build --no-cache service-name service-database-name` to construct the latest iterations of a service and its associated database.
3. **Service Activation**: Launch the services with `docker-compose up service-name service-database-name`, establishing a live test environment.
4. **Testing**: Execute `docker exec container-name npm test` within service containers, validating the latest changes.
5. **Artifact Handling**: Collate build artifacts and upload them for further analysis.
6. **Clean Up**: Gracefully terminate the service environment using `docker-compose down`, maintaining a clean slate for subsequent operations.

By streamlining these steps, our CI/CD pipeline ensures rapid, reliable delivery of updates with minimal manual intervention, propelling our development velocity while upholding high-quality standards.

```yaml
name: Build And Test GPT Service

on:
  push:
    paths:
      - "GptService/**"
      - ".github/workflows/build_and_test_master_gpt_service.yaml"

  pull_request:
    paths:
      - "GptService/**"
      - ".github/workflows/build_and_test_master_gpt_service.yaml"

jobs:
  build-and-test-gpt-service:
    runs-on: ubuntu-latest

    env:
      CONTAINER_NAME: "ay2324s1-course-assessment-g16_gpt-service_1"

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: Set up Docker Environment
        uses: docker/setup-buildx-action@v3

      - name: Create New Environment File With GPT API Key
        run: echo GPT_API_KEY=${{ secrets.GPT_API_KEY }} > GptService/.env

      - name: Build GPT Service with Docker Compose
        run: docker-compose build --no-cache gpt-service

      - name: Run GPT Service with Docker Compose
        run: docker-compose up -d gpt-service

      - name: List Docker Containers
        run: docker ps

      - name: Logs [For Checking Status Only]
        run: docker-compose logs

      - name: Run Test Cases in GPT Service
        run: docker exec $CONTAINER_NAME npm test

      - name: Retrieve Artifacts [JEST Test Reports]
        run: docker cp $CONTAINER_NAME:/usr/src/app/coverage ./gpt-service-test-jest-reports

      - name: Upload Artifacts
        uses: actions/upload-artifact@v3
        with:
          name: gpt-service-test-jest-report-artifacts
          path: ./gpt-service-test-jest-reports/*

      - name: Shut Down Containers
        run: docker-compose down
```

Code snippet: GitHub Workflow of GPT Service

Improvements & Enhancements

1. We envision Peer Prep evolving to support a larger user base, accommodating thousands of simultaneous users and expanding the capacity for participants in collaborative sessions. This scalability will be achieved through backend optimization and potential cloud-based solutions to ensure that our platform remains responsive and efficient, regardless of user volume.

2. Our goal is to enhance the matching mechanism to reduce the wait time currently experienced by users. By refining our algorithm and infrastructure, we aim to shorten the matching process from 7 seconds to a more immediate response, ensuring a more fluid user experience.

3. We plan to address the latency in recording match cancellations, which occasionally leads to re-matching issues. An optimized synchronization process will be developed to instantly reflect cancellation requests, thus preventing any overlap with new match requests.

4. Future developments will consider integrating video communication capabilities. Adding to our existing text and audio channels, video communication will offer users a more comprehensive and interactive collaborative experience, closely simulating in-person sessions.

5. Recognizing the delays in socket communication between the frontend and backend, we're researching more efficient protocols to minimize latency. By reducing the current 2-second buffer on session timeouts, we aim to make real-time interactions more seamless and responsive.

6. Initially, security considerations were modest, reflecting the project's academic context. However, for PeerPrep to thrive in a commercial environment, bolstering security is imperative. Future enhancements will focus on implementing advanced security protocols, employing rigorous testing, and ensuring compliance with industry standards to safeguard user data and interactions comprehensively.

7. Inspired by the accessibility in gaming, we plan to introduce room codes for collaborative sessions. This feature will enable users to invite peers into a coding session, fostering collaboration and enhancing the educational experience by leveraging familiar social interactions.

8. To augment the coding experience, we would like to integrate live runtime environments. This will empower users to execute code and observe real-time results. Additionally, custom input fields for test cases will be provided, offering users the ability to validate their solutions against various scenarios, further simulating a real-world software development environment.

9. The History Service has the potential to become a more comprehensive resource. By capturing and storing detailed session artifacts, such as chat logs and call recordings, users will be able to revisit their collaborative experiences with greater context, enhancing their learning curve and allowing for deeper post-session analysis.

Reflection & Learning Points

This project has been an enlightening journey, deepening our understanding of software engineering principles and design patterns. Our collaborative efforts across various services have illuminated the value of best practices in a real-world setting.

From the outset, our approach lacked a unified methodology, leading to disparate coding styles and structures. Recognizing the need for harmony, we adopted standardized practices such as consistent file directories, which significantly streamlined our development workflow and minimized merge conflicts during integration.

The project also underscored the significance of strategic planning. Regular weekly meetings and clear task delegation fostered a collective responsibility and ensured alignment on the project's direction. Prior to implementing new features, we engaged in thorough discussions, reaching a consensus that guided our development choices. In retrospect, a more evenly spread task schedule throughout the semester could have alleviated end-term pressures, a lesson we will carry forward.

Our gratitude extends to the teaching team for their dedication to this module. Your guidance has been invaluable, and for that, we are profoundly thankful.

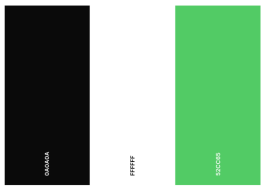
. Appendix

UI/UX Hi-Fidelity Prototypes

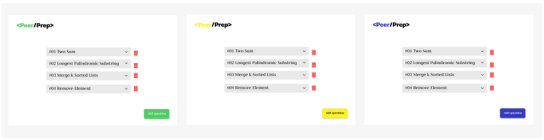
Style Guide

Font face: **Roboto**

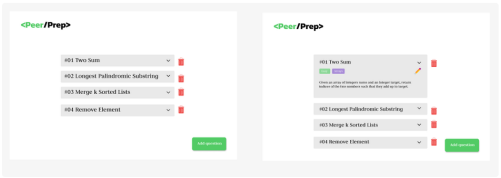
Main colour palette:



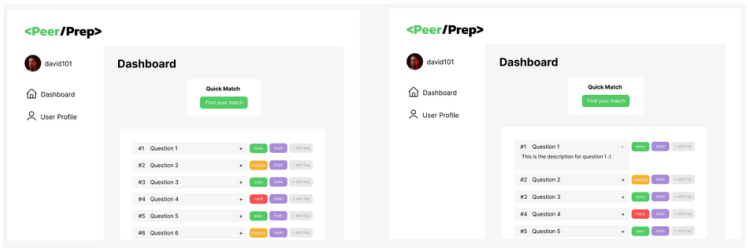
Hi-Fi Mockups



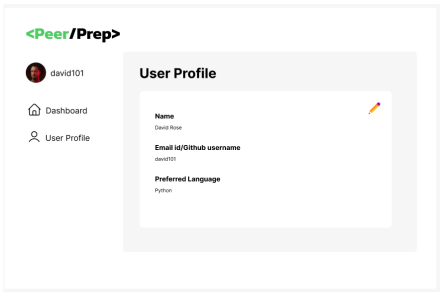
Admin Question View (colour variations)



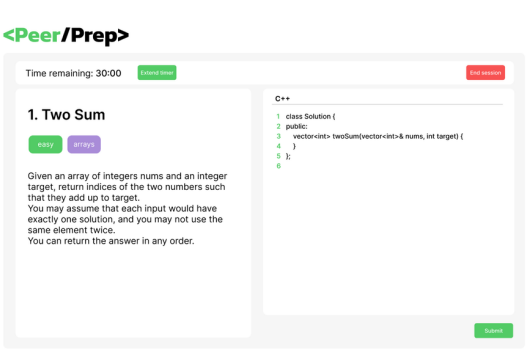
Admin Question View (chosen colour)



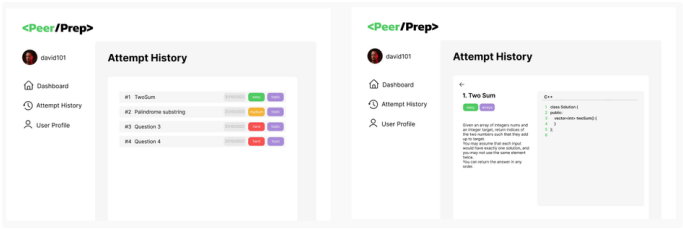
User dashboard and question view



User profile page



Collaboration window



Attempt history list and detailed views

API Documentation

Authentication Service

We utilised Firebase for Authentication, and referred to this for Firebase Authentication APIs and Documentation.

Thanks to our extensive testing, during the development process, we found a bug in Firebase Auth API. We identified an unexpected behavior in the Firebase Auth API, specifically within the `updateEmail()` function. Initially, the issue presented as a potential flaw in our implementation. However, after thorough review and validation against our codebase, the problem persisted, leading us to seek external confirmation.

A thorough search revealed a relevant discussion on StackOverflow, where other developers had encountered similar issues, corroborating our findings.

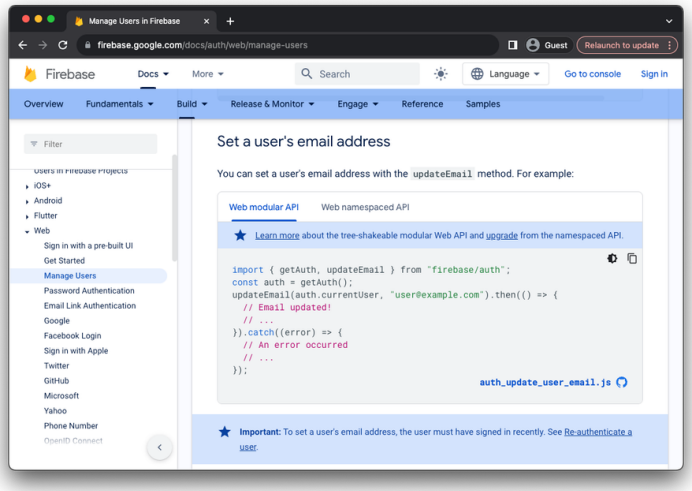
Link to StackOverFlow Post talking about the bug

Bug Report Synopsis:

- **Expected Behavior**: The Firebase API documentation suggests that users should be able to update their email addresses without requiring email verification.
- **Actual Behavior**: Attempts to update a user's email address without email verification result in an error message: `[FirebaseError: Firebase: Please verify the new email before changing email. (auth/operation-not-allowed).]`

This discrepancy between the actual behavior and the official Firebase documentation indicates a potential oversight in the Firebase Auth API. The issue has been echoed by several developers in the community, as seen in the StackOverflow post, and has led to multiple bug reports being submitted for further investigation by the Firebase team.

As of the time of submitting this report, the issue with Firebase Auth's `updateEmail()` method remains unresolved and is acknowledged as a valid bug by the developer community.



Screenshot: Firebase `updateUser()` API Documentation

User Service

Register User

This endpoint allows one to add a user and their related data into the database. The userId is specified as a parameter in the endpoint.

HTTP Method: `POST`

Endpoint: http://localhost:3002/users/:userId

Required (all): name (string), email (string), githubId (required), preferredLanguage (required)

```
{
  "name": "Jai",
  "email": "sample@gmail.com",
  "githubId": "Jai2501",
  "preferredLanguage": "C++"
}
```

Responses

Status	Parameters / Request Body	Response

201 (Created)	Parameter:
	`userId:`
	`SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", "githubId": "Jai2501", "preferredLanguage": "C++" }`
		`{ "message": "POST USER", "user": { "mongo_id": "6553c3f70b0549fdd1f96e6e", "_id": "SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1", "name": "Jai", "email": "sample@gmail.com", "githubId": "Jai2501", "preferredLanguage": "C++", "isAdmin": false, "__v": 0 } }`

404 (Not Found)	Parameter:
	`userId:`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", "githubId": "Jai2501", "preferredLanguage": "C++" }`
		`{ "error": { "message": "Route Not Found" } }`

500 (Internal Server Error)	Parameter:
	`userId:`
	`SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", }`
		`{ "message": "POST USER", "error": { "message": "User validation failed: gitl } }`

Get User Data

This endpoint allows one to retrieve user related data from the database. The userId is specified as a parameter in the endpoint.

HTTP Method: `GET`

Endpoint: http://localhost:3002/users/:userId

Required: none

Responses

Status	Parameters / Request Body	Response

200 (OK)	Parameter:
	`userId: SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
		<pre>{
		  "message": "GET USER",
		  "user": {
		  "_id": "SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1",
		  "mongo_id": "65537d9967d11e855029d797",
		  "name": "Jai",
		  "email": "test_sample@gmail.com",
		  "githubId": "GitHub ID Placeholder",
		  "preferredLanguage": "Java",
		  "isAdmin": true,
		  "__v": 0
		  }
		}</pre>

404 (Not Found)	Parameter:
	`userId: non-existent`
		<pre>{
		  "message": "GET USER: NOT FOUND",
		  "user": null
		}</pre>

500 (Internal Server Error)	Parameter:
	`userId: SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
		<pre>{
		  "message": "GET USER",
		  "error": {
		  "reason": "Internal Error"
		  }
		}</pre>

Patch User Data

This endpoint allows one to update user and their related data in the database. The userId is specified as a parameter in the endpoint.

HTTP Method: `PATCH`

Endpoint: http://localhost:3002/users/:userId

Required (one or more): name (string), email (string), githubId (required), preferredLanguage (required)

```
{
  "name": "Jai",
  "email": "sample@gmail.com",
  "githubId": "Jai2501",
  "preferredLanguage": "C++"
}
```

Responses

Status	Parameters / Request Body	Response

200 (OK)	Parameter:
	`userId: SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", "githubId": "NewGitHubId", "preferredLanguage": "Python" }`
		`{ "message": "PATCH USER", "status": { "acknowledged": true, "modifiedCount": 1, "upsertedId": null, "upsertedCount": 0, "matchedCount": 1 } }`

404 (Not Found)	Parameter:
	`userId: non-existent`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", "githubId": "NewGitHubId", "preferredLanguage": "Python" }`
		`{ "message": "PATCH USER: NOT FOUND", "status": { "acknowledged": true, "modifiedCount": 0, "upsertedId": null, "upsertedCount": 0, "matchedCount": 0 } }`

404 (Not Found)	Parameter:
	`userId:`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", "githubId": "NewGitHubId", "preferredLanguage": "Python" }`
		`{ "error": { "message": "Route Not Found" } }`

500 (Internal Server Error)	Parameter:
	`userId: SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
	
	Body:
	`{ "name": "Jai", "email": "sample@gmail.com", "githubId": "NewGitHubId", "preferredLanguage": "Python" }`
		`{ "message": "PATCH USER", "error": { "reason": "Internal Error" } }`

Update User Privilege

This endpoint allows one to update a user's privilege, by making them an admin too.

HTTP Method: `PATCH`

Endpoint: http://localhost:3002/users/update-privilege

Required (at least email): email (string), isAdmin (boolean)

Responses

Status	Parameters / Request Body	Response
200 (OK)	Body: `{ "email": "sample@gmail.com", "isAdmin": true }`	`{ "message": "PRIVILEGE UPDATE USER", "status": { "acknowledged": true, "modifiedCount": 1, "upsertedId": null, "upsertedCount": 0, "matchedCount": 1 } }`
404 (Not Found)	Body: `{ "email": "unknown@gmail.com", "isAdmin": true }`	`{ "message": "PRIVILEGE UPDATE: USER NOT FOU "status": { "acknowledged": true, "modifiedCount": 0, "upsertedId": null, "upsertedCount": 0, "matchedCount": 0 } }`
404 (Not Found)	Body: `{ "isAdmin": true }`	`{ "message": "PRIVILEGE UPDATE: USER NOT FOU "status": { "acknowledged": true, "modifiedCount": 0, "upsertedId": null, "upsertedCount": 0, "matchedCount": 0 } }`
500 (Internal Server Error)	Body: `{ "email": "sample@gmail.com", "isAdmin": true }`	`{ "message": "PRIVILEGE UPDATE USER", "error": { "reason": "Internal Error" } }`

Delete User Data

This endpoint allows one to delete a user and their related data in the database. The userId is specified as a parameter in the endpoint.

HTTP Method: `DELETE`

Endpoint: http://localhost:3002/users/:userId

Required: none

Responses

Status	Parameters / Request Body	Response

200 (OK)	Parameter:
	`userId: SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
		<pre>{
		  "message": "DELETED USER",
		  "userId": "SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1",
		  "docs": {
		  "acknowledged": true,
		  "deletedCount": 1
		  }
		}</pre>

404 (Not Found)	Parameter:
	`userId: non-existent`
		<pre>{
		  "message": "DELETE USER: NOT FOUND",
		  "status": {
		  "acknowledged": true,
		  "deletedCount": 0
		  }
		}</pre>

404 (Not Found)	Parameter:
	`userId:`
		<pre>{
		  "error": {
		  "message": "Route Not Found"
		  }
		}</pre>

500 (Internal Server Error)	Parameter:
	`userId: SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
		<pre>{
		  "message": "DELETE USER",
		  "error": {
		  "reason": "Internal Error"
		  }
		}</pre>

Get All Users Data

This endpoint allows one to retrieve the data of all the users from the database.

HTTP Method: `GET`

Endpoint: http://localhost:3002/users/

Required: none

Responses

Status	Parameters / Request Body	Response

200 (OK)	-	{
		"message": "GET ALL USERS",
		"users": [
		{
		"_id": "1QWlzOLhckSHwoaHMIAr2StV4u
		"mongo_id": "654e27ed808a52febf94a
		"name": "Jai",
		"email": "jai@u.nus.edu",
		"githubId": "Jai2501",
		"preferredLanguage": "Java",
		"isAdmin": true,
		"__v": 0
		}
		]
		}

500 (Internal Server Error)	-	{
		"message": "GET ALL USER",
		"error": {
		"reason": "Internal Error"
		}
		}

Matching Service

Health Check

Verify Matching Service is accessible at localhost:3004

HTTP method: `GET`

Endpoint: http://localhost:3004/

Response

Status	Parameters / Request Body	Response
200 (OK)	-	`<h1>Matching Service is up!</h1>`

Find Match

Send a matching request to the server with user Id and matching criteria.

HTTP method: `POST`

Endpoint: http://localhost:3004/home/:userId

Response

Status	Parameters / Request Body	Response
200 (OK) [Matched]	http://localhost:3004/home/Qa5Xb8Rv2KpL Request Body (JSON): `{ "id":"Qa5Xb8Rv2KpL", "language":"Python", "proficiency":"Basic", "difficulty":"Intermediate", "topic":"Strings" }`	`{ "status": "success", "isMatched": true, "sessionId": "45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd7 "questionId": "65378371752185e6e1b5b68c", "collaboratorId": "Zu8YkQ3mBvLx", "request": { "id": "Qa5Xb8Rv2KpL", "language": "Python", "proficiency": "Basic", "difficulty": "Intermediate", "topic": "Strings" } }`
	http://localhost:3004/home/Zu8YkQ3mBvLx Request Body (JSON): `{ "id":"Zu8YkQ3mBvLx", "language":"Python", "proficiency":"Basic", "difficulty":"Intermediate", "topic":"Strings" }`	`{ "status": "success", "isMatched": true, "sessionId": "45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd7 "questionId": "65378371752185e6e1b5b68c", "collaboratorId": "Qa5Xb8Rv2KpL", "request": { "id": "Zu8YkQ3mBvLx", "language": "Python", "proficiency": "Basic", "difficulty": "Intermediate", "topic": "Strings" } }`
500 (Internal Server Error) [No Match]	http://localhost:3004/home/Abctest123 Request Body (JSON): `{ "id":"Abctest123", "language":"Python", "proficiency":"Basic", "difficulty":"Intermediate", "topic":"Arrays" }`	`{ "status": "error", "isMatched": false, "sessionId": null, "questionId": null, "collaboratorId": null, "request": { "id": "Abctest123", "language": "Python", "proficiency": "Basic", "difficulty": "Intermediate", "topic": "Arrays" } }`

Cancel Match

Cancel a matching request.

HTTP method: `DELETE`

Endpoint: http://localhost:3004/home/:userId/matching

Response

Status	Parameters / Request Body	Response

200 (OK)	**Match request:**
	http://localhost:3004/home/Abctest123
	Request Body (JSON):
	{ “id”:”Abctest123”,
	”language”:”Python”,
	”proficiency”:”Basic”,
	”difficulty”:”Intermediate”,
	”topic”:”Strings” }
		<pre>{
		  "status": "cancel",
		  "isMatched": false,
		  "sessionId": null,
		  "questionId": null,
		  "collaboratorId": null,
		  "request": {
		  "id": "Abctest123",
		  "language": "Python",
		  "proficiency": "Basic",
		  "difficulty": "Intermediate",
		  "topic": "Arrays"
		  }
		}</pre>

	Cancel request:
	http://localhost:3004/home/Abctest123/matching
		<pre>{
		"message": "Match cancelled successfull
		}</pre>

500 (Internal Server Error)	-	<pre>{
		"message": "Failed to cancel match"
		}</pre>

Get Active Session Id By User Id

Get the session Id of an active session for a user.

HTTP method: `GET`

Endpoint: http://localhost:3004/getMatchSession/:userId

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3004/getMatchSession/Qa5Xb8Rv2KpL	`{ "sessionId": "45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd }`
200 (OK) [No active session]	http://localhost:3004/getMatchSession/abcTest123	`{ "sessionId": null }`
500 (Internal Server Error)	-	`{ "sessionId": null }`

Get Active Session By Session Id

Get the details of an active session for a session.

HTTP method: `GET`

Endpoint: http://localhost:3004/getSession/:sessionId

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3004/getSession/45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd7	`{ "sessionId": "45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd7", "session": { "_id": "65520801586d616457827485", "sessionId": "45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd7 "id1": "Qa5Xb8Rv2KpL", "id2": "Zu8YkQ3mBvLx", "isEnded": false, "questionId": "65378371752185e6e1b5b68c", "language": "Python", "proficiency": "Basic", "difficulty": "Intermediate", "topic": "Strings", "__v": 0 } }`
200 (OK) [No active session]	http://localhost:3004/getMatchSession/abcTest123	`{ "sessionId": null, "session": null }`
500 (Internal Server Error)	-	`{ "sessionId": null, "session": null }`

End Session

Set `isEnded` to true for a matched pair.

HTTP method: DELETE

Endpoint: http://localhost:3004/end/:sessionId

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3004/end/45e15e6a-16a7-40c4-9fdf-8b0cc4ca5fd7	`{ "status": "success", "message": "Session ended successfully" }`
500 (Internal Server Error)	-	`{ "status": "error", "message": "Failed to end session" }`

Question Service

Get Question

Gets question by id

HTTP method: `GET`

Endpoint: http://localhost:3003/api/questions/:id

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3003/api/questions/6554b4749798ff792c2c0d6f	`{ "questionId": "6554b4749798ff792c2c0d6f", "question": { "_id": "655471569b66b6250ce79c3e", "title": "Test Question 1", "description": "This is test question 1.", "complexity": "Easy", "category": "Data Structures", "language": "Other Languages", "__v": 0 } }`
500 (Internal Server Error)	http://localhost:3003/api/questions/123	`{ "error": "Internal server error" }`

Get Questions (All)

Gets all questions from the database

HTTP method: `GET`

Endpoint: http://localhost:3003/api/questions

Response

Status	Parameters / Request Body	Response

200 (OK)	http://localhost:3003/api/questions	[
		  {
		  "_id": "6554b4749798ff792c2c0d6f",
		  "title": "Test Question 1",
		  "description": "This is test question 1.",
		  "complexity": "Easy",
		  "category": "Data Structures",
		  "language": "Other Languages",
		  "__v": 0
		  },
		  {
		  "_id": "655471569b66b6250ce79c3f",
		  "title": "Test Question 2",
		  "description": "This is for Test Question 2",
		  "complexity": "Medium",
		  "category": "Sorting",
		  "language": "SQL",
		  "__v": 0
		  }
		]

500 (Internal Server Error)	http://localhost:3003/api/questions	{
		  "error": "Error retrieving questions"
		}

Create Question

Adds new questions to database

HTTP method: `POST`

Endpoint: http://localhost:3003/api/questions/

Response

Status	Parameters / Request Body	Response

200 (OK)	<pre>{
	  "title": "Test Question 1a",
	  "description": "This is qn.",
	  "complexity": "Easy",
	  "category": "Data Structure",
	  "language": "Other Languages"
	}</pre>
		<pre>{
		  "errors": {},
		  "question": {
		  "_id": "6554a9b661731bcc457d05d7",
		  "title": "Test Question 1a",
		  "description": "This is qn.",
		  "complexity": "Easy",
		  "category": "Data Structure",
		  "language": "Other Languages",
		  "__v": 0
		  }
		}</pre>

400 (Bad Request)	<pre>{
	  "title": "Test Question 1",
	  "description": "This is qn 1.",
	  "complexity": "Easy",
	  "category": "Data Structure",
	  "language": "Other Languages"
	}</pre>
		<pre>{
		  "errors": {
		  "duplicateTitle": "Question title already exists. Please enter new title!"
		  },
		  "question": null
		}</pre>

400 (Bad Request)	<pre>{
	  "title": "Test Question 1b",
	  "description": "This is qn.",
	  "complexity": "Easy",
	  "category": "Data Structure",
	  "language": "Other Languages"
	}</pre>
		<pre>{
		  "errors": {
		  "duplicateDescription": "Question description already exists. Please enter new description!"
		  },
		  "question": null
		}</pre>

Update Question

Edit existing questions in database

HTTP method: `PATCH`

Endpoint: http://localhost:3003/api/questions/:id

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3003/api/questions/6554b4749798ff792c2c0d6f `{ "title": "Question A", "description": "This is qn.", "complexity": "Easy", "category": "Data Structure", "language": "Other Languages" }`	`{ "_id": "6554b4749798ff792c2c0d6f", "title": "Question A", "description": "This is qn.", "complexity": "Easy", "category": "Data Structure", "language": "Other Languages", "__v": 0 }`
400 (Bad Request)	http://localhost:3003/api/questions/6554b4749798ff792c2c0d6f `{ "title": "Test Question 1a", "description": "This is qn1a.", "complexity": "Easy", "category": "Data Structure", "language": "Other Languages" }`	`{ "error": "Question with an identical title already exists." }`
400 (Bad Request)	http://localhost:3003/api/questions/6554b4749798ff792c2c0d6f `{ "title": "Test Question 1b", "description": "This is qn.", "complexity": "Easy", "category": "Data Structure", "language": "Other Languages" }`	`{ "error": "Question with an identical description already exists." }`
500 (Internal server error)	http://localhost:3003/api/questions/1	`{ "error": "Invalid question ID." }`

Get Match Question

Gets questions for matching according to difficulty, topic, and language specified by the user

HTTP method: `GET`

Endpoint: http://localhost:3003/api/questions/match

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3003/api/questions/match `{ "complexity": "Easy", "category": "Data Structure", "language": "Other Languages" }`	`{ "question": "6554bf9ac3cfb372e94df4b2" }`
200 (OK)	http://localhost:3003/api/questions/match `{ "complexity": "Hard", "category": "Data Structure", "language": "Other Languages" }`	`{ "question": null, "request": { "complexity": "Hard", "category": "Optimisation", "language": "Other Languages" } }`

Delete Question

Delete questions from database

HTTP method: `DELETE`

Endpoint: http://localhost:3003/api/questions/:id

Response

Status	Parameters / Request Body	Response

200 (OK)	http://localhost:3003/api/questions/6554b4749798ff792c2c0d70	{
		"_id": "6554b4749798ff792c2c0d70",
		"title": "Test Question 2",
		"description": "This is for Test Question 2",
		"complexity": "Medium",
		"category": "Sorting",
		"language": "SQL",
		"__v": 0
		}

500 (Internal server error)	http://localhost:3003/api/questions/1	{
		"error": "Invalid question ID."
		}

Collaboration Service

Health Check

Verify Collaboration Service is accessible at localhost:3005

HTTP method: `GET`

Endpoint: http://localhost:3005/

Response

Status	Parameters / Request Body	Response
200 (OK)	-	`<h1>Collaboration Service is up!</h1>`

Get Collaboration History

Get the details for a collaboration session.

HTTP method: `GET`

Endpoint: http://localhost:3005/getCollaborationHistory/:sessionId

Response

<table>
<tr><th>Status</th><th>Parameters / Request Body</th><th>Response</th></tr>
<tr><td>200 (OK)</td><td>http://localhost:3005/getCollaborationHistory/123c44c9-9bc3-402f-ba56-689eb0d2774d</td><td><pre>{
  "status": "success",
  "initTime": 1699876344556,
  "language": "Java",
  "codes": [
    {
      "line": 2,
      "code": "console.log(\"java is good\");",
      "lastModifier": "Gc2Bz9Nl8Wx4",
      "_id": "65520e0c9436297d49bc96c4"
    }
  ]
}</pre></td></tr>
<tr><td rowspan="2">500 (Internal Server Error)</td><td>http://localhost:3005/getCollaborationHistory/abcTest123</td><td>Data not exists:<pre>{
  "status": "error",
  "message": "Collaboration data does not exist",
}</pre></td></tr>
<tr><td>-</td><td>Error:<pre>{
  "status": "error",
  "message": <error message>,
}</pre></td></tr>
</table>

Delete Collaboration History

Delete the record of a collaboration history.

HTTP method: `DELETE`

Endpoint: http://localhost:3005/getSessionHistory/:sessionId

Response

Status	Parameters / Request Body	Response

200 (OK)	http://localhost:3005/deleteSessionHistory/123c44c9-9bc3-402f-ba56-689eb0d2774d	<pre>{
		"status": "success",
		"message": "Collaboration data deleted"
		}</pre>

500 (Internal Server Error)	http://localhost:3005/deleteSessionHistory/abcTest123	Data not exists:
		<pre>{
		  "status": "error",
		  "message": "Collaboration data does not e
		}</pre>

	-	Error:
		<pre>{
		  "status": "error",
		  "message": <error message>,
		}</pre>

History Service

Add Attempt Details

This endpoint allows one to add a user's attempt into the database.

HTTP Method: `POST`

Endpoint: http://localhost:3006/history/add-attempt

Required (all): userId1 (string), userId2 (string), sessionId (required), questionId (required)

```
{
  "userId1": "abcd1234",
  "userId2": "wxyz7890",
  "sessionId": "4321dcba",
  "questionId": "C0987zyxw",
  "questionTitle": "Sample Titile",
  "questionDescription": "Sample Description",
  "questionCategory": "Sample Category",
  "questionComplexity": "Sample Complexity"
}
```

Responses

Status	Parameters / Request Body	Response
201 (Created)	Body: `{ "userId1": "abcd1234", "userId2": "wxyz7890", "sessionId": "4321dcba", "questionId": "C0987zyxw", "questionTitle": "Sample Tit… "questionDescription": "Samp… "questionCategory": "Sample … "questionComplexity": "Sampl… }`	`{ "message": "SUCCESSFUL: ADD USER ATTEMPT H… "result": { "fieldCount": 0, "affectedRows": 1, "insertId": 1, "info": "", "serverStatus": 2, "warningStatus": 0, "changedRows": 0 } }`
500 (Internal Server Error)	Body: `{ "userId2": "wxyz7890", "sessionId": "4321dcba", "questionId": "C0987zyxw" }`	`{ "message": "HISTORY DATA COULD NOT BE ADDED… "error": "Missing Fields or Internal Error" }`

Get Attempt Details

This endpoint allows one to retrieve a user's attempt related data from the database. The userId is specified as a parameter in the endpoint.

HTTP Method: `GET`

Endpoint: http://localhost:3006/history/:userId

Required: none

Responses

Status	Parameters / Request Body	Response

200 (OK)	Parameter:
	`userId:`
	`SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
		<pre>{
		  "message": "SUCCESSFUL: GET USER ATTEMPT H
		  "result": [
		  {
		  "historyId": 1,
		  "userId1": "SyKrnq4q0Uf1nm3g9w9x5B
		  "userId2": "1QWlzOLhckSHwoaHMIAr2S
		  "sessionId": "654e27ed808a52febf94
		  "questionId": "1001",
		  "questionTitle": "Sample Titile",
		  "questionDescription": "Sample Des
		  "questionCategory": "Sample Catego
		  "questionComplexity": "Sample Comp
		  "attempt_time": "2023-11-14T20:19:
		  }
		  ]
		}</pre>

404 (Not Found)	Parameter:
	`userId: non-existent`
		<pre>{
		  "message": "NO HISTORY DATA FOUND: GET USER
		  "result": []
		}</pre>

404 (Not Found)	Parameter:
	`userId:`
		<pre>{
		  "error": {
		  "message": "Route Not Found"
		  }
		}</pre>

500 (Internal Server Error)	Parameter:
	`userId:`
	`SyKrnq4q0Uf1nm3g9w9x5BM5ZDq1`
		<pre>{
		  "message": "NO HISTORY DATA FOUND",
		  "error": "Internal Error",
		}</pre>

Communication Service

Health Check

Verify Communication Service is accessible at localhost:3007

HTTP method: `GET`

Endpoint: http://localhost:3007/

Response

Status	Parameters / Request Body	Response
200 (OK)	-	`<h1>Communication service is up!</h1>`

GPT Service

Health Check

Verify GPT Service is accessible at localhost:3008

HTTP method: `GET`

Endpoint: http://localhost:3008/

Response

Status	Parameters / Request Body	Response
200 (OK)	-	`<h1>GPT Service is up!</h1>`

Get GPT Response

Request reply from ChatGPT on a question prompt.

HTTP method: `POST`

Endpoint: http://localhost:3008/generate

Response

Status	Parameters / Request Body	Response
200 (OK)	http://localhost:3008/generate Request Body (JSON): `{ "userId":"Gc2Bz9Nl8Wx4", "prompt":"Generate a short story in 40 words" }`	`{ "status": "success", "reply": "In a quaint village, a young girl found an old key abandoned near a dilapidated house. Intrigued, she ventured inside, only to discover a hidden room filled with dusty books. As she opened one, she was transported to a magical realm, where she would embark on extraordinary adventures." }`

500 (Internal Server Error)	-	Function error:
		
		`{ "status": "error", "reply": <any> }`
		
		`{ "message": "Unknown error. Please try again!"" }`
		
		Server error:
		
		`{ "message": "Failed to get response from gpt. Please try again!" }`

Get Cached GPT Replies

Get the cached replies for a user in an active session.

HTTP method: `GET`

Endpoint: http://localhost:3008/getCache/:userId

Response

Status	Parameters / Request Body	Response

200 (OK)	http://localhost:3008/getCache/Gc2Bz9Nl8Wx4	{
		"status": "success",
		"record": [
		{
		"prompt": "What are stack and queue?",
		"reply": "A stack and a queue are both common da
		structures in computer science used to store an
		organize data.\n\n1. Stack: A stack is an order
		collection of elements where the insertion and
		removal of elements follow the Last-In-First-Ou
		(LIFO) principle. This means that the last
		element added to the stack will be the first on
		to be removed. For example, imagine a stack of
		plates, where you can only add or remove plates
		from the top.\n\nKey operations:\n- Push: Adds
		element to the top of the stack.\n- Pop: Remove
		and returns the top element of the stack.\n-
		Peek/Top: Returns (without removing) the top
		element of the stack.\n- IsEmpty: Checks if the
		stack is empty.\n\nApplications:\n- Function
		calls and recursion.\n- Undo/Redo operations.\n
		Evaluating expressions (e.g., infix to postfix
		conversions).\n\n2. Queue: A queue is an ordere
		collection of elements where the insertion
		happens at the rear (also called \""
		},
		{
		"prompt": "Generate a short story",
		"reply": "Once upon a time in a small village,
		there lived a young girl named Lily. She was kno
		for her kindness, cheerfulness, and her love for
		flowers. Lily spent most of her days tending to
		her beautiful garden, which was filled with
		vibrant roses, lilies, and daisies.\n\nOne brigh
		morning, as Lily was watering her flowers, she
		discovered a tiny wounded bird in the bushes. Sh
		carefully picked it up and cradled it in her han
		, feeling an immediate bond with the fragile
		creature. Determined to help the bird, Lily gent
		placed it in a small nest, lined with soft
		feathers, inside her garden shed.\n\nDay after
		day, Lily diligently cared for the bird, feeding
		it and nursing it back to health. Slowly but
		surely, the bird’s injured wing healed, and it
		became stronger with each passing day. The bird,
		who Lily named Sunny, was immensely grateful to
		Lily for her love and care.\n\nAs the weeks went
		by, Sunny and Lily"
		},
		{
		"prompt": "Generate a short story in 40 words",
		"reply": "In a quaint village, a young girl foun
		an old key abandoned near a dilapidated house.
		Intrigued, she ventured inside, only to discover
		a hidden room filled with dusty books. As she
		opened one, she was transported to a magical
		realm, where she would embark on extraordinary
		adventures."
		}
		]
		}

500 (Internal Server Error)	-	Function error:
		{
		"status": "error",
		"reply": <any>
		}
		Server error:

		<pre>{
		  "status": "error",
		  "message": <error message>
		}</pre>

Exit GPT Session

Clear the cached replies for a user when the session ended.

HTTP method: `DELETE`

Endpoint: http://localhost:3008/exitGpt/:userId

Response

Status	Parameters / Request Body	Response

200 (OK)	http://localhost:3008/exitGpt/Gc2Bz9Nl8Wx4	<pre>{
		  "status": "success"
		}</pre>

500 (Internal Server Error)	-	Function error:
		<pre>{
		  "status": "error",
		  "reply": "Failed to exit session."
		}</pre>Server error:
		<pre>{
		  "status": "error",
		  "message": "Failed to get response from
		  Please try again!"
		}</pre>

Milestone Timelines

Milestone 1: MVP and Project Requirements

Time: Week 3 to Week 7

TIMELINE	TASK
PROJECT PLANNING	
Week 3 - Week 5	• Brainstorming the user requirements • Classify Functional Requirements and Non-Functional Requirements
ASSIGNMENT	
Week 6 - Week 7	• Started the development of Frontend with Question View in mind for Assignment 1 • Implemented local storage for storing questions and its related data • Started the development of Matching Service with Assignment 5 in mind
FRONTEND	
Week 6 - Week 7	• Started the development of Login and Signup Pages • Development of UI for questions • Developed UI for Matching Window and Button
USER SERVICE	
Week 6 - Week 7	• Started the development of User Service, by setting up MongoDB • Set up Express Server and Routes • Support Create and Read Operations on User Data
MATCHING SERVICE	
Week 6 - Week 7	• Start the implementation of Matching Microservice • Implement a matching algorithm using RabbitMQ to match the users • Set up server, routes and controller for Matching Service • Set up MongoDB database model for matched pair
QUESTION SERVICE	
Week 6 - Week 7	• Create APIs to perform CRUD operations • Link frontend to backend
DOCUMENTATION	
Week 7	• Publish README.md for related services

Milestone 2: Project Progress Check

Time: Week 8 to Week 10

TIMELINE	TASK
PROJECT PLANNING	
Week 8 - Week 10	• Brainstormed ideas for collaboration and finalised nice to haves that we wished to implement
ASSIGNMENT	
Week 8 - Week 9	• Started working on Assignment 2 and 3
FRONTEND	
Week 8 - Week 9	• Development of UI for questions • Development of UI for popups in matching • Development of User Profile
AUTHENTICATION SERVICE	
Week 8 - Week 9	• Implemented a trial version using Firebase Admin SDK
USER SERVICE	
Week 8 - Week 9	• Support All CRUD operations on User Data • Add Route to get all Users • Add Route and support for updating a Users Privilege • Dockerized User Service and User Service Database
MATCHING SERVICE	
Week 8	• Add simple unit test cases • Dockerized Matching service
COLLABORATION SERVICE	
Week 8 - Week 9	• Add server socket using Socket.IO • Implement server-side socket algorithm • Set up server, routes and controller for Collaboration Service • Set up MongoDB database model for collaborative input • Dockerized collaboration service
QUESTION SERVICE	
Week 8 - Week 9	• Created APIs for deleting and adding tags (removed feature) • Added unit tests for question service • Added additional fields and modified schema • Dockerized question service
DOCUMENTATION	
Week 9	• Update README.md for available services

Final Milestone: Submission, Demo & Presentation

Time: Week 11 to Week 13

TIMELINE	TASK
ASSIGNMENT	
Week 12	• Started working on Assignment 4, and created a “master” `docker-compose.yml` file
FRONTEND	
Week 11-13	• Set up view for collaboration service • Developed UI for history service, communication service, GPT service • Improved UI for question service
Week 12	• Added Route Protections ◦ Segregate based on Authentication State and User Privilege • Added Unit and Integration Tests for User Service API • Dockerized Frontend • Update README.md
AUTHENTICATION SERVICE	
Week 11	• Implement Authentication using Firebase Client SDK • Added Unit and Integration Tests
USER SERVICE	
Week 12	• Added Unit and Integration Tests • Update README.md
QUESTION SERVICE	
Week 12-13	• Add filter and search functionality
HISTORY SERVICE	
Week 11	• Started the development of History Service, by setting up MySQL • Set up Express Server and Routes • Support Create and Read Operations on History Attempt Data • Dockerized History Service and History Service Database
Week 12	• Added Unit and Integration Tests • Update README.md
COMMUNICATION SERVICE	
Week 12 - Week 13	• Implement server socket using Socket.IO to handle text communication • Set up WebRTC Connection event listeners using Socket.IO in server to support audio communication • Implement server logic for the communication service • Dockerized Communication Service
GPT SERVICE	
Week 12 - Week 13	• Create connection to ChatGPT-3.5 using OpenAI API • Implement server logic for the generative AI service (GPT Service) • Write simple unit test cases • Dockerized GPT Service
DEVOPS	
Week 12 - Week 13	• Created a “master” `docker-compose.yml` file for running all the services in parallel using one command. • Setup CI/CD using GitHub Actions • Added a workflow for each Service • Added a workflow for Frontend • Added a workflow to build all Services
DOCUMENTATION	
Week 12 - Week 13	• Update README.md for relevant services • Project Final Report