CS3219 Project

AY22/23 Semester 1

Project Report

Team 32

Team Members	Student No.
Oei Yi Ping	A0202810Y
Chua Jun Jie Benedict	A0200412H
Tan Jing Yi Jace	A0205068J
Rachel Gina Abelarde	A0202181X

Content Page

Overall System

The following are features that we have implemented in our PeerPrep:

Must-haves: User service, Matching service, Question service, Collaboration service, basic UI

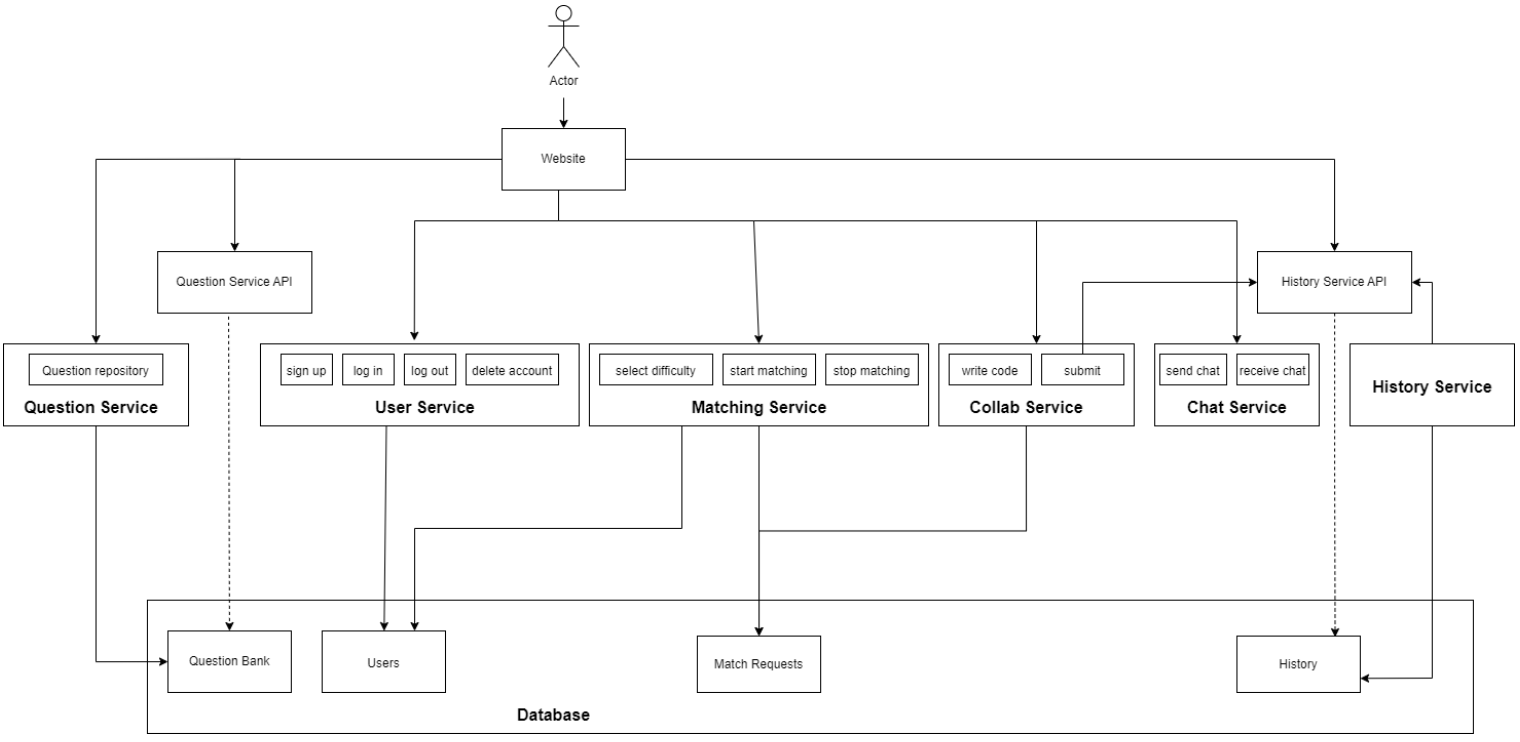
Nice-to-haves: History service, Chatting service, user-friendly UI, local docker deployment via docker compose.

Currently, our application is only deployed on local machines using native technology stack.

The system is implemented using the Microservices Architecture. The user interacts with the frontend, which in turn communicates with the various backend services to serve the desired site functionality. The backend services do not interact directly with one another. As such, the frontend acts as the orchestrator of the entire system, as well as serves the information to the view for the user.

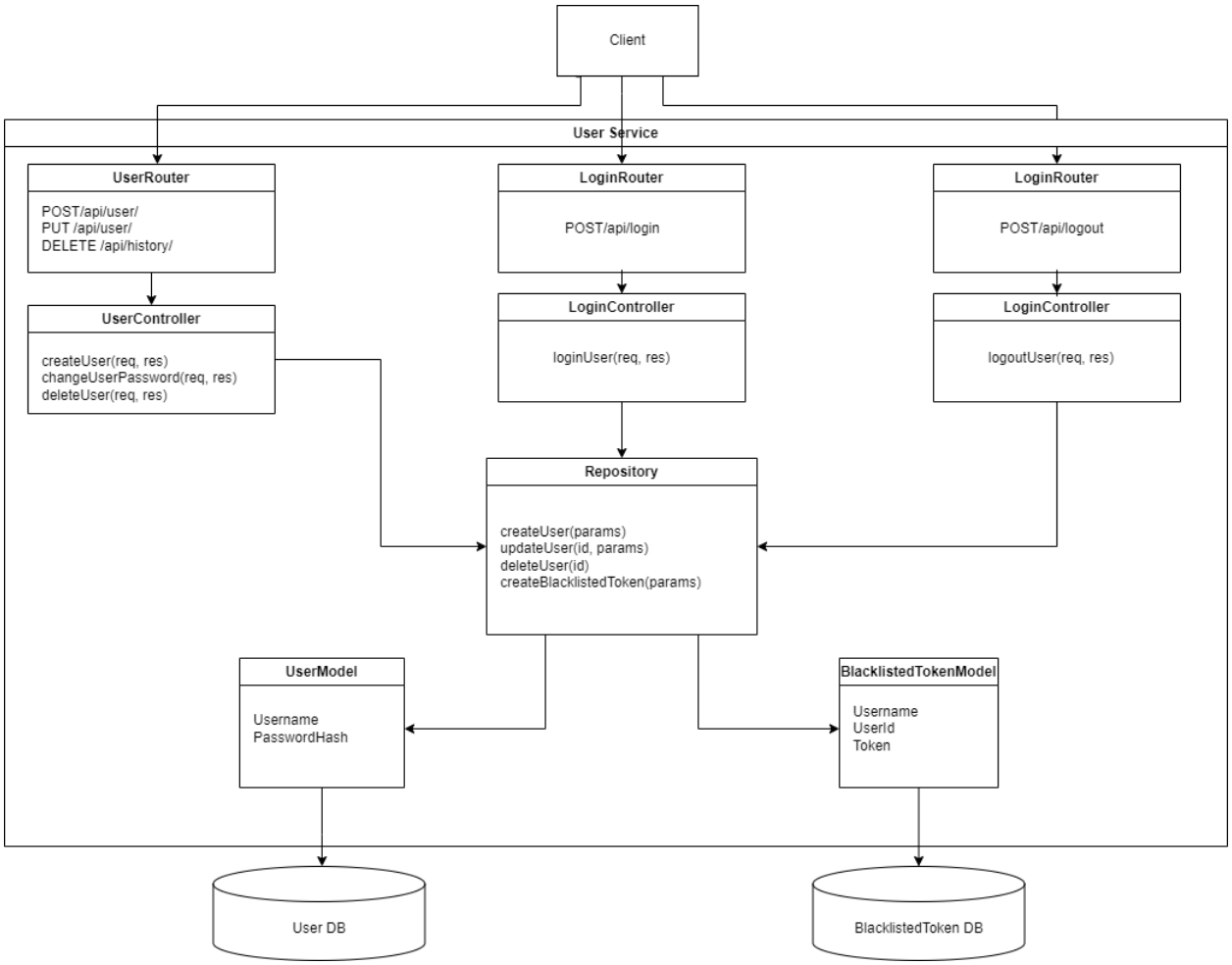
Minimal dependencies between the backend services allows each service to be implemented independently, and allow Single Responsibility Principle to be adhered as each service is solely responsible for their respective scope of features. The frontend fetches the necessary information from each service and organizes them to construct the desired response for the user.

Architecture Design

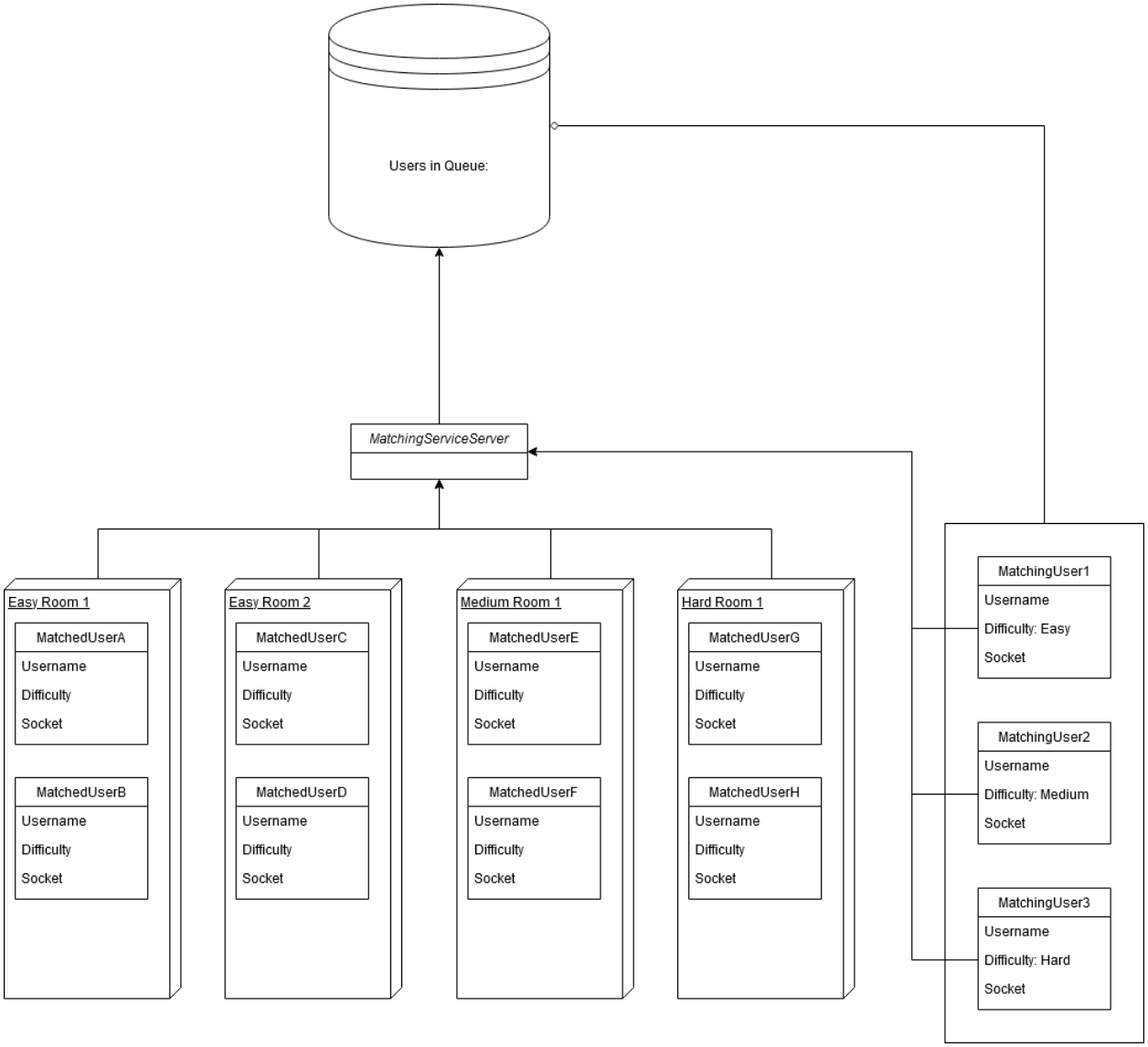


Class-level Design

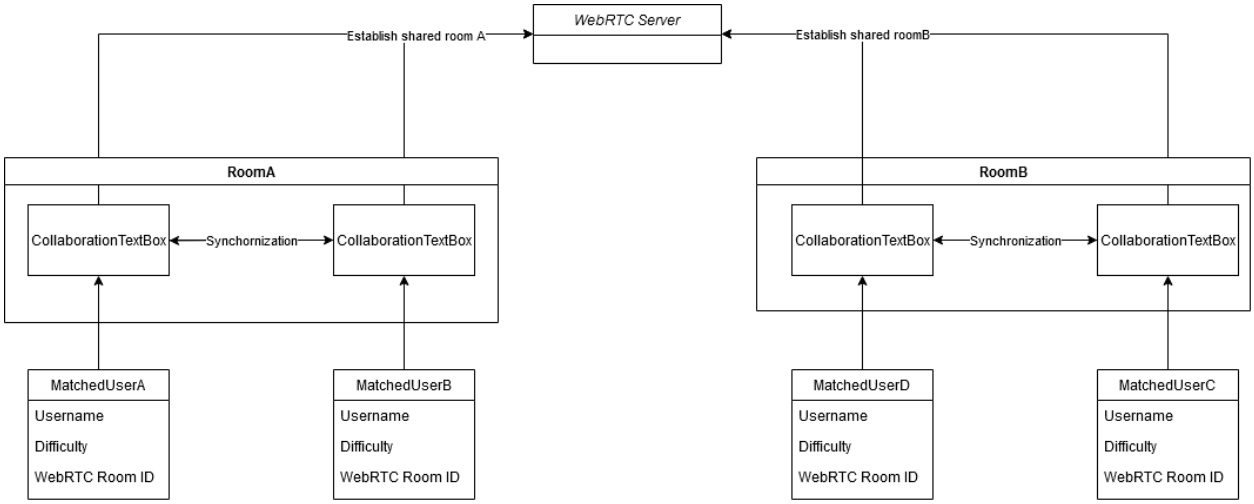
User Service



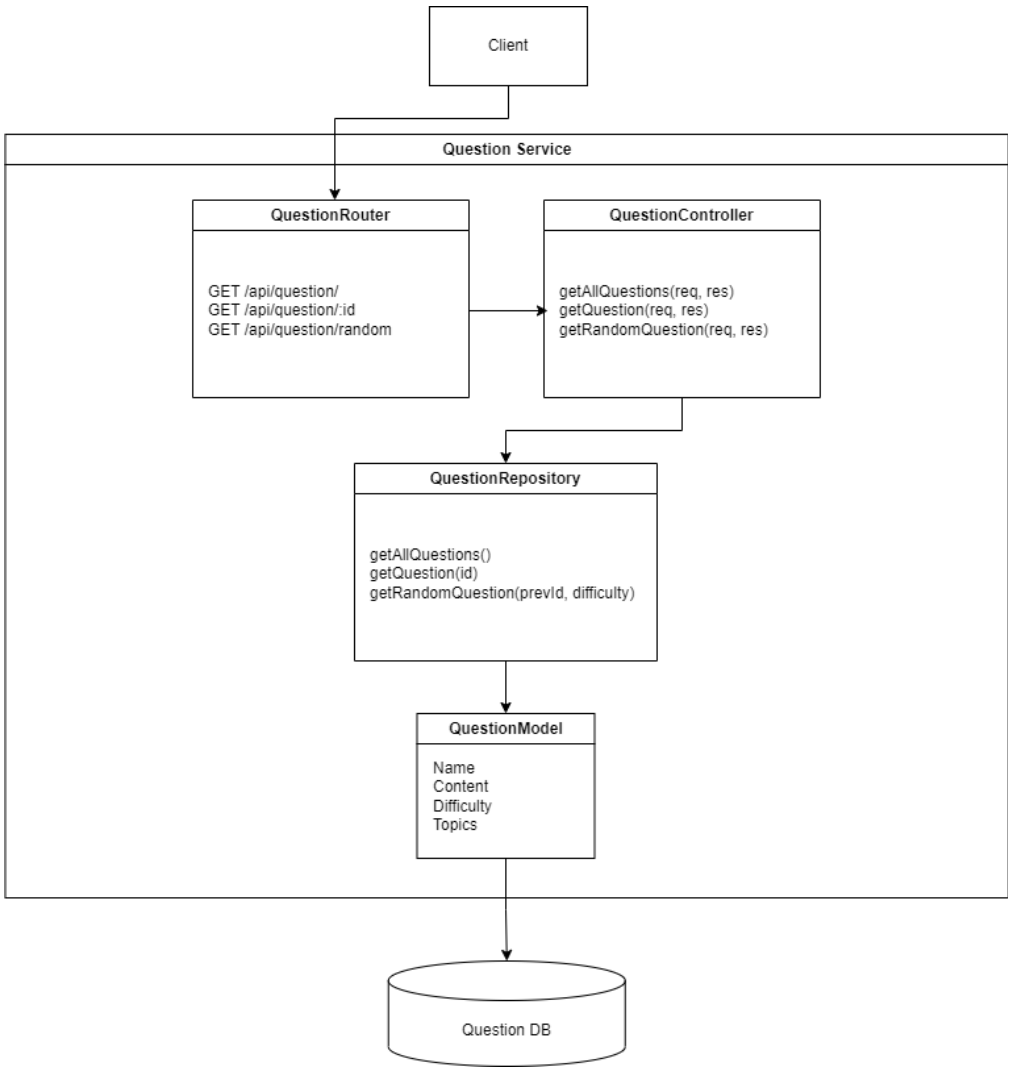
Matching Service



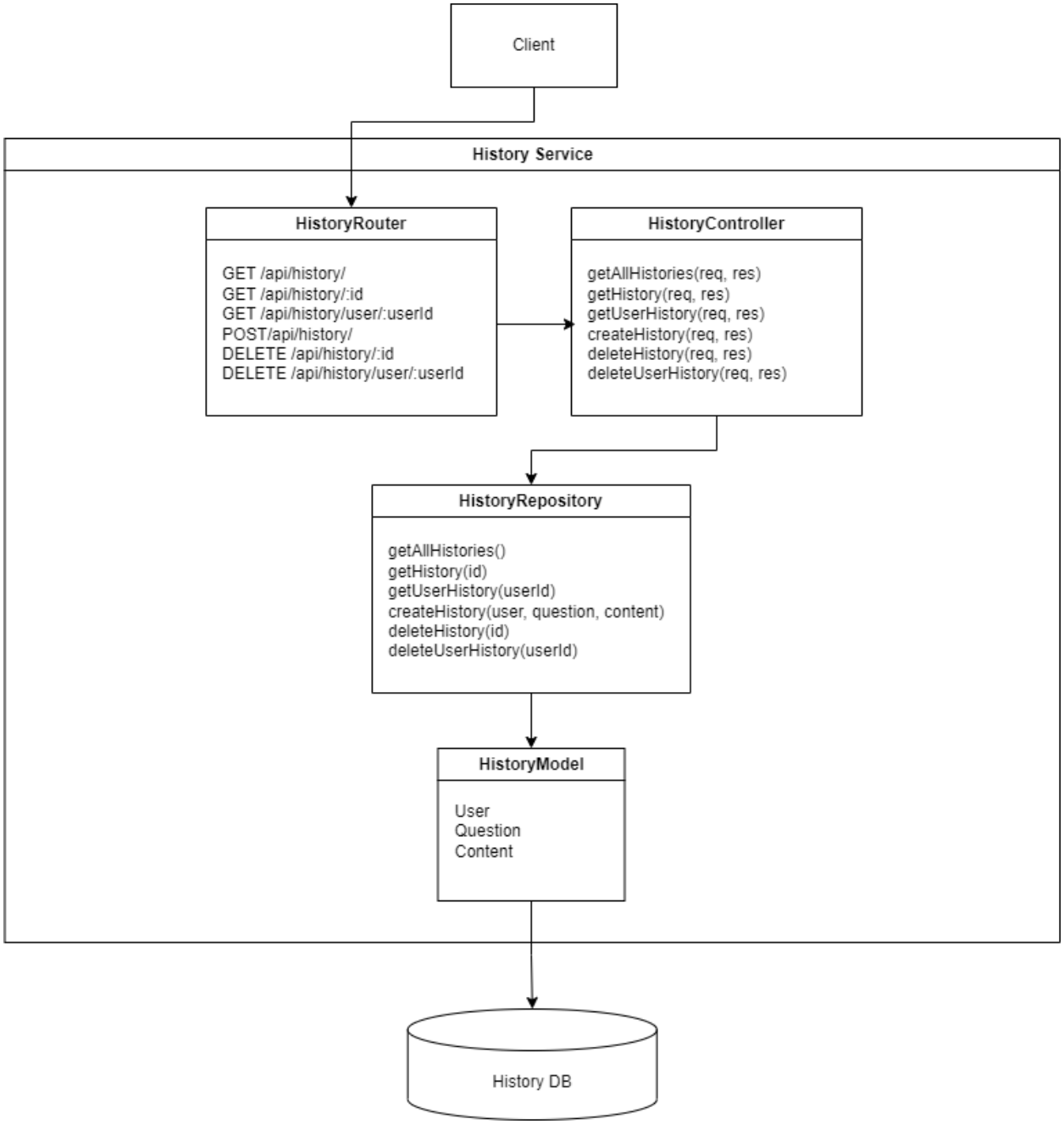
Collaboration Service



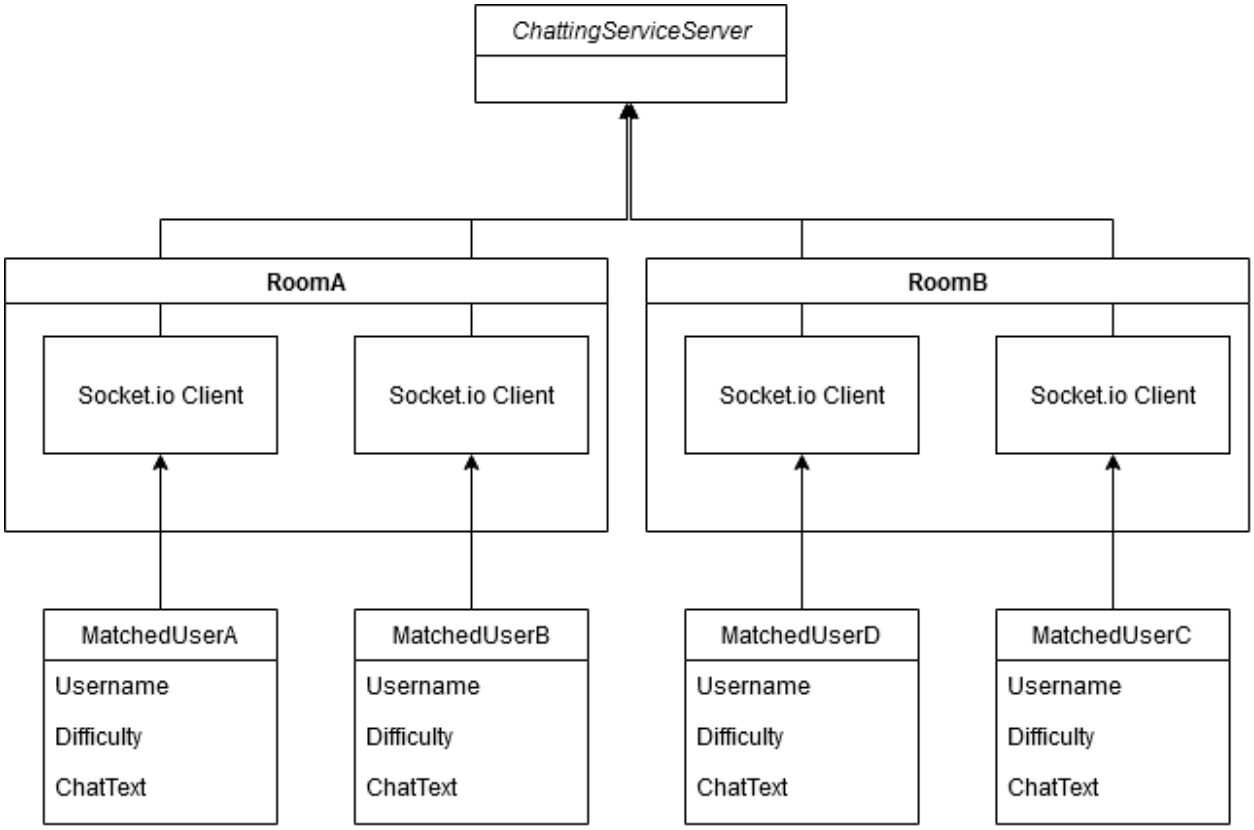
Question Service



History Service

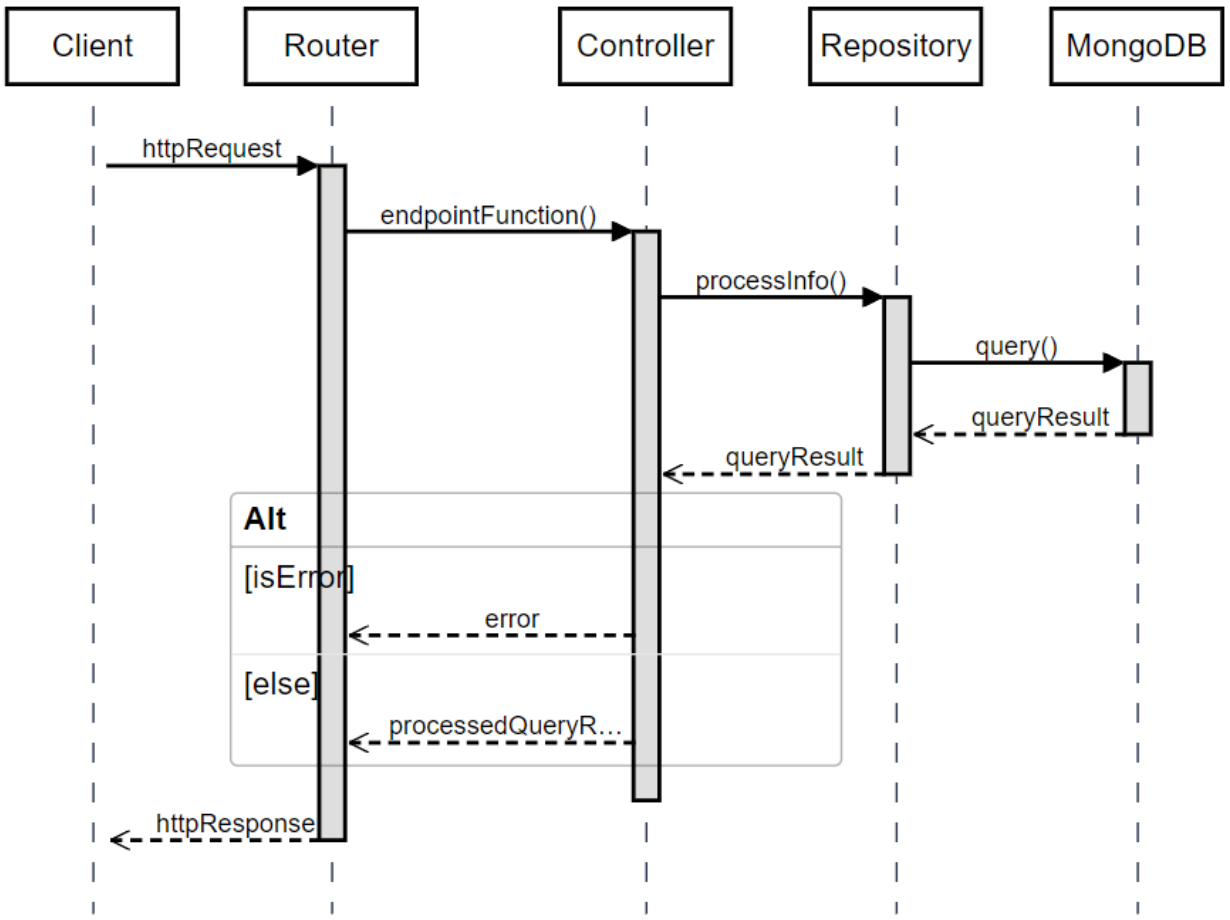


Chatting Service

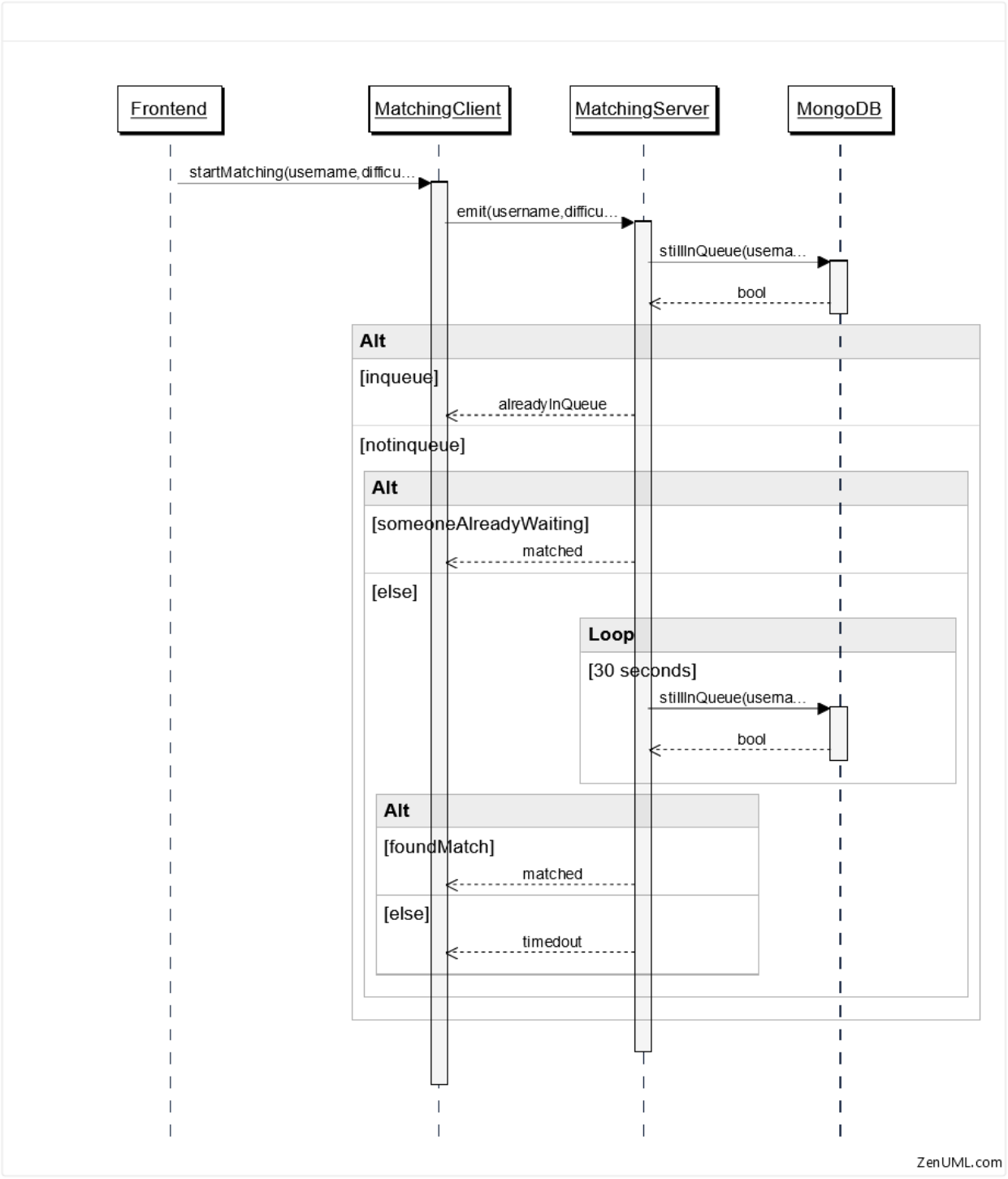


Behavioural Design

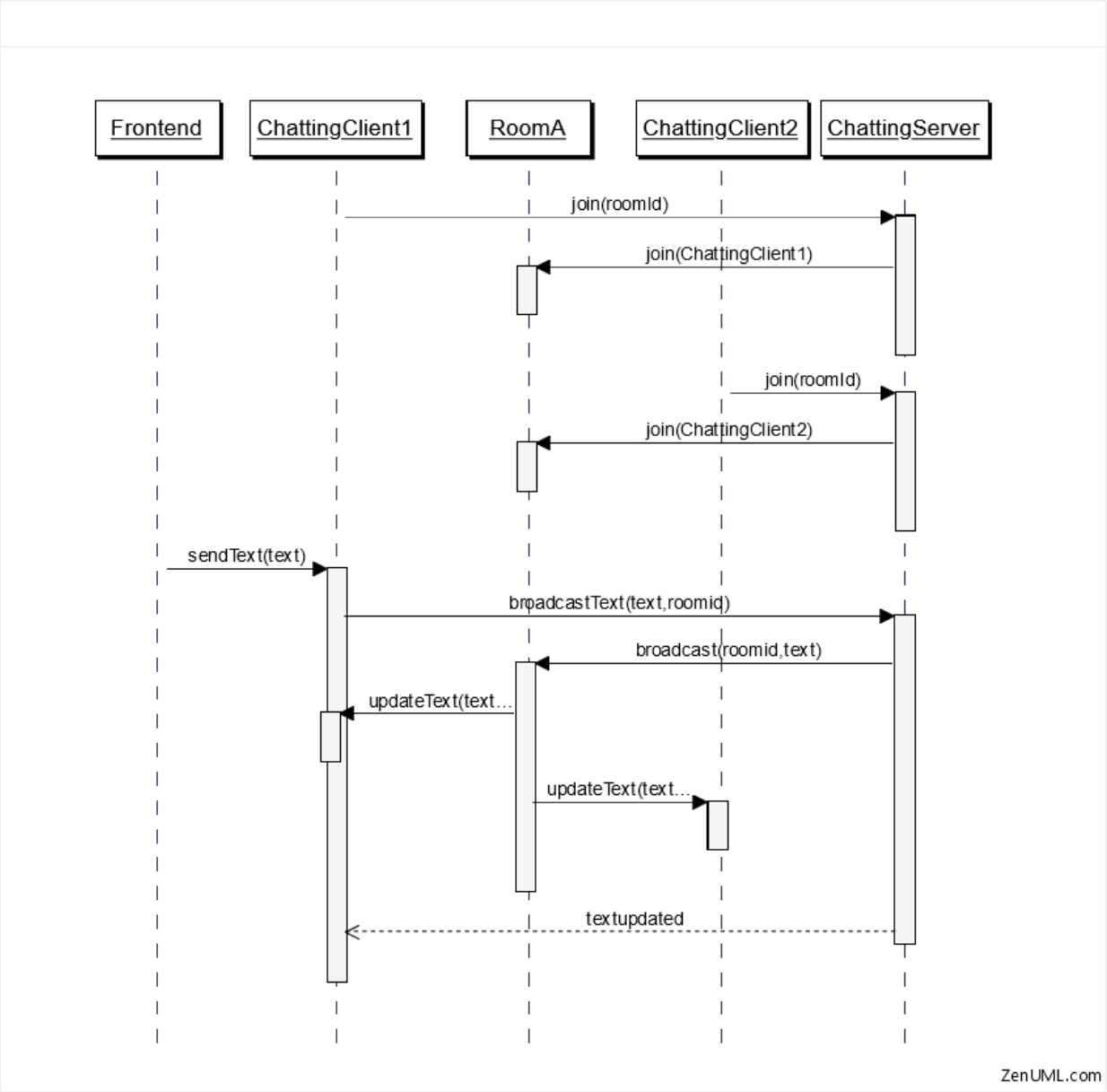
User, Question, History Service (Generalised)



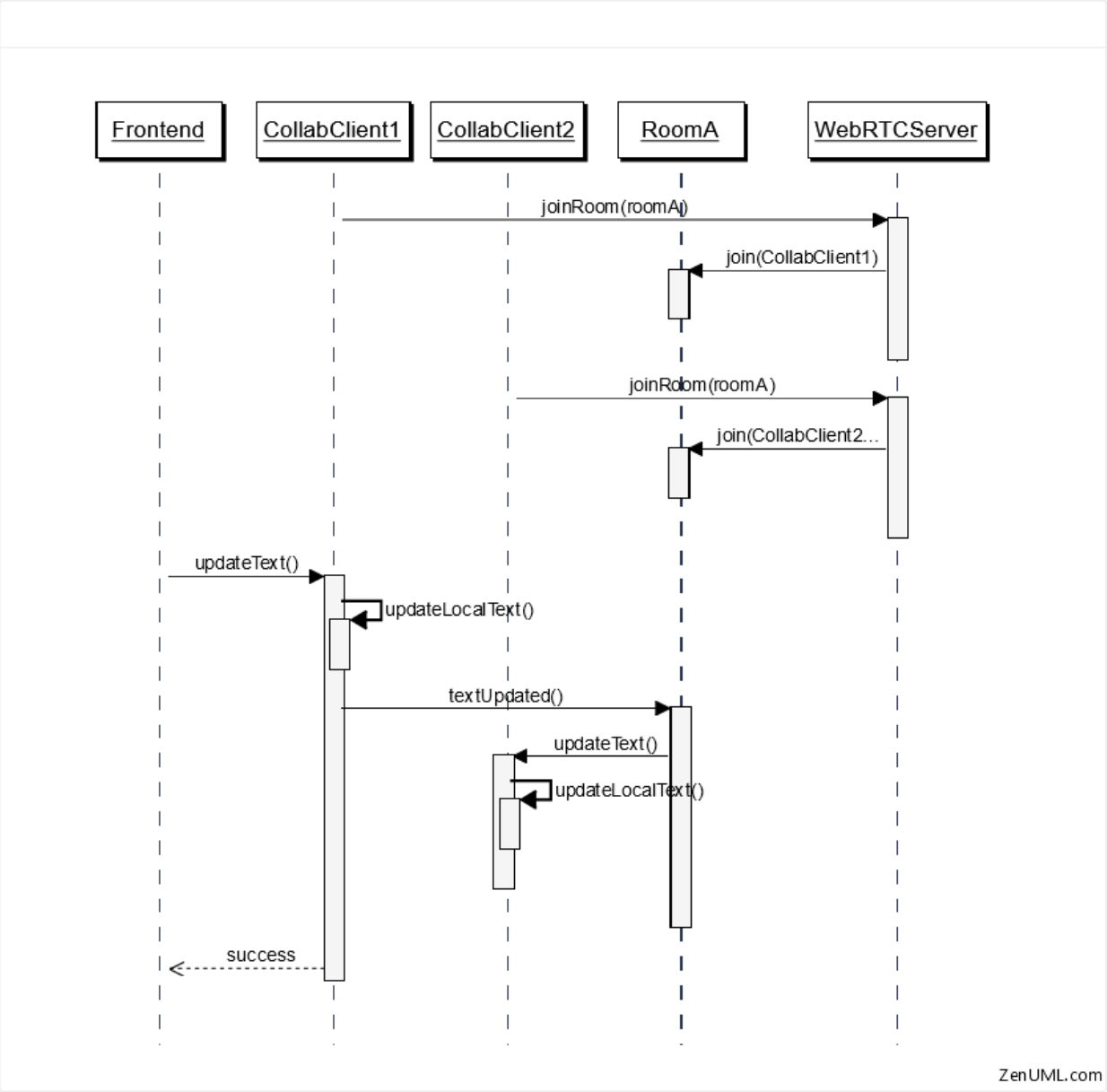
Matching Service



Chatting Service



Collaboration Service



Local Deployment

All services and frontend are deployed locally using docker compose, where their respective images are generated based on the individual dockerfiles. The local machine ports are then mapped to the exposed ports of each of the containers, thus allowing the all services and frontend to be accessible from the local machine.

Development Process

In each milestone, the team will discuss the features to be developed for the milestone, as well as their respective requirements. These requirements are then written and stored as a form of requirements documentation. Each team member would then pick a feature to own, and develop the feature independently. This development process works since the overall system follows the microservices architecture, where individual services can be independently developed and provides a well-defined set of features within their corresponding scope of responsibility. The frontend is often the last feature to be developed per milestone, since it acts as the orchestrator and is therefore dependent on the development of the backend services it will use.

Frontend

Overview

All services are integrated using the frontend. Using React and material UI components, the frontend is displayed to users. All pages are consolidated in the App.js file for navigation purposes. Order of page functional navigation is:

Sign up -> Login -> Select Difficulty -> Matching -> Match Success -> Code Collaboration and Chat

From any screen, the user can also select “HOME” on the top left corner of the page to go back to selection of difficulty of their questions. Users may also access their account settings at any point, which consists of 3 sections/components: Change Password, Delete Account, and Question History.

Notable API calls from the frontend are to the question service, which retrieves questions for the user from the database, and to history service, which retrieves past questions and answers that the user has submitted from the database. More information described in the respective sections below.

User Authentication utilising User Service API

The user service API allows us to easily using POST, PUT/PATCH as well as DELETE to login, modify user information (i.e. password) and allow users to delete their account respectively. Initially, we had used axios for all of these user functions to communicate with the API. However, axios can also be used intercept HTTP requests and responses and enables client-side protection against XSRF. As such, due to CORS protection of authentications, we faced a restriction in communication with our API. There seemed to be no official solution for this “cors blockage” of communication between the frontend and backend API for sending Bearer Tokens (which is what we use for our user authentication; both login and logout). Due to this protective but restrictive issue, we could not do API calls easily.

We decided to shift over to the Fetch API which is less protective but less restrictive and more lightweight natively supported by more web browsers. This resolved the issue of our bearer tokens not passing through the the API successfully.

On top of this, we utilise cookies on the client-side to store usernames as well as bearer tokens generated upon successful log ins. This process is as follows: the moment the user tries to log in, we send a POST request to our user service API to authenticate our user's submitted information. Upon successful log in, the server will send back a response that includes the bearer token generated personally for the user using jwt, along with a success status code; else, it will return a failure status code with an error message. The successful response contains the bearer token which will be stored in a document cookie on our client-side. We decided to use cookies as there is the HTTPOnly cookie flag available to us which can be used to restrict the cookie access in JavaScript to mitigate a few security issues such as cross-site scripting. On top of that, we knew that we were not storing a large amount of information in the cookie, hence we did not see much use for browser storage methods like local and session storages. Cookies were also useful as whenever we send a request to the server, all the cookies are also sent along; so they are also especially useful for tasks related to authentication.

Currently, our cookies are persistent. We have specifed the Max-Age attribute as 24 hours. This means that our cookies do not expire on closing the browser but will expire after 24 hours have passed.

Real-Time Updates using Socket.io

Matching service, Collaboration service and Chatting service rely on socket.io to ensure users are able to communicate with one another and have speedy, real-time updates. The messages emitted from the service side are captured on the frontend, likewise, if a message is emitted from the frontend, it can be captured by the service. More details are discussed further in the sections below.

Usage of Components and CSS

Multiple screens also utilise the components personalised for them that we have utilised to provide some structure to our client side codebase within the 'Components' folder. Simple CSS style sheets have also been created for these components to ensure that there is consistency through the frontend.

User Service

Overview

User Service is a service that handles users by accounts. A user is able to login, logout, change passwords and delete their account. Non-users will not be able to access internal functions of our application without authentication.

Requirements

Functional Requirement	Justification	Priority	Status

The system should allow users to create an account with username and password.	To allow the user to register an account via a conventional and familiar registration process.	High	Completed
The system should ensure that every account created has a unique username.	To allow the user to identify other users via a more reader-friendly manner (as opposed to identifying via user id, which is a string of random characters).	High	Completed
The system should allow users to log into their accounts by entering their username and password.	To allow the user to log into an account via a conventional and familiar login process.	High	Completed
The system should allow users to log out of their account.	To allow the user to safely log out of the website on the device, so that another user cannot continue using the previous user's account.	High	Completed
The system should allow users to delete their account.	To allow the user to delete any stored account information.	High	Completed
The system should allow users to change their password.	To allow the user to change to a new password for security or convenience reasons.	High	Completed
The system should allow only authenticated users to make modifications to their own account.	To prevent other users from modifying information or executing critical operations pertaining to another user's account for security purposes. Examples include preventing changing of another user's password, and deleting	High	Completed

	another user's account.		

User process and functional requirements

Upon loading of the site, users are first shown a sign up page. If the user does not have an account, they are able to sign up with a username and password. The username and password is then stored in our database, mongoDB, as an object with username and userID in plaintext, and password hashed with SHA256. Afterwhich, the user may use the same authentication page and choose the login option and login with the same username and password. If either the username or password is incorrect, the user will be shown the same error message that says "Incorrect username or password".

On login, the user will be brought to the select difficulty page immediately. On the top right corner, through all the possible authenticated routes, the user will be able to look at their account settings. The user is able to logout from their account at any page in the application, from the icon in the top right corner. Upon clicking logout, the user will be logged out and brought to the default "user authentication" page of our application where they can choose to log in or sign up.

In the Account Settings page, firstly, the user is able to update/change their password. By simply entering their new password and clicking "change password", the user is able to change their password successfully. Next, the user is given an option to delete their account. To delete their account, they have to enter their username as confirmation, and then press confirm. Their account will then be deleted from the database.

Should the user want to create an account again with the same username, they will be able to do so. However, the userID associated with the new account will be different and hence, the user will not be able to access any past user history that the previous account associate with the same username may have anymore.

Non-Functional Requirement	**Justification**	**Priority**
System should store the users' passwords that are hashed and salted.	To not reveal the users' password in plaintext for security purposes.	Completed

Technology Stack	
Frontend	React
Backend	Express.js/Node.js
Backend dependencies	Mongoose dotenv cors bcrypt
Database	MongoDB
Project Management Tools	GitHub Issues

Development Tech Stack

The selection of React and Express.js/Node.js is inline with the technologies used for the overall structure of the development project. MongoDB, similar to the rationale behind its use throughout the different services, is the database of choice due to its scalability and availability on the cloud.

Mongoose is used to facilitate and ease the interactions with MongoDB. This is so as Mongoose provides ready-to-use core functionalities such as data validation and abstracts away much of the logic that should otherwise be implemented in order to query or operate on the data stored in MongoDB. Mongoose also requires the need of defining a schema for a User model. Since the attributes of a Question are fixed to begin with, the need of a defined schema is not a limitation, and is advantageous due its facilitation of data validation. As such, Mongoose's functionality saves much development effort that is otherwise required to interact with MongoDB.

dotenv is used to facilitate the reading of environment variables stored in the .env file. These environment variables often contain sensitive information such as the credentials for database access, and therefore should not be committed to the repository. As such, reading directly from the

environment is necessary for the sake of security, as opposed to directly including such sensitive information in the source files.

cors is used to allow communications between the backend server and the frontend, by resolving the security issues resulting from Cross-Origin Resource Sharing (CORS), since the frontend and backend are of different origins.

bcrypt is used to facilitate the salting and hashing of the user password, according to the industry standard, prior to storing the processed password in the database.

Using External Database as Datastore

User tokens are blacklisted upon logout to prevent the ability to use the now invalid tokens to carry out operations that require authentication. MongoDB is used as the datastore to keep track of the blacklisted tokens. Another alternative datastore was considered - Redis, an in-memory solution. However, since MongoDB is already being used to store the User information, using Redis would mean that the number of dependencies would increase. Furthermore, the use of Redis would mean that an additional server would have to be instantiated for Redis, thereby increasing the complexity of the program.

Matching Service

Overview

Matching service matches users according to their preferred level of difficulty when choosing questions.

Requirements

Functional Requirement	Priority	Status
The system should allow users to select a difficulty.	High	Completed
The system should ensure users can proceed to the right difficulty matching page.	High	Completed
The system should allow users to change difficulties.	High	Completed
The system should allow users to start the matching process.	High	Completed
The system should allow users to stop the matching process.	High	Completed
The system should allow users to match with another user of the same difficulty level.	High	Completed
The system should allow users to leave the room after finding a match.	High	Completed

User process and functional requirements

Users are brought to a difficulty selection page which they must confirm their selected difficulty before they start matching. Users can change their difficulty before moving to the matching page, users could also move back to the difficulty selection page from the matching page.

Afterwards, users would be at the start matching page and can start their matching based on the selected difficulty. The user would then be put on queue to wait for a matching user with the same difficulty. At any point in the queue, the user can cancel their request to be matched.

If no user with matching difficulty is found, the user is timed out after 30 seconds and removed from the queue.

If a same user with matching difficulty is found, they are brought to the matched page where the user can proceed to the question solving page. If the user decides to not proceed, the user can return back to the matching page or select difficulty page after being matched.

Non-Functional Requirement	**Priority**	**Status**
Matching should start within 0.5s when the user clicks on “Start Matching”.	High	Completed
Matching should stop within 0.5s when the user clicks on “Stop Matching”.	High	Completed
After 30s without matching another user, the user should be removed from the matching queue.	High	Completed

<u>User process and non-functional requirements</u>

When the user selects the option to start matching as mentioned in the functional requirements section, the server receives the request and processes the request immediately. All matching requests happen instantaneously without delay and should not take more than 0.5 seconds.

Similar to the start matching requirement, when the user selects the option to stop matching, the server receives the request and processes it immediately. This request is also instantaneous and should not take more than 0.5 seconds.

After a user enters the queue to be matched, the server would asynchronously check the status of each user in the queue. If they are matched, they would be removed from the queue and proceeding information would be shared for the setup of the question solving page. However, after 30 seconds, if the server detects that the status of the user has not been updated to matched, the user would be notified that they have not found a match and have been removed from the search.

Technology Stack	
Frontend	React

Backend	Express.js/Node.js
Library	Socket.io
Database	MongoDB
Project Management Tools	GitHub Issues

Development Tech Stack

The selection of React and Express.js/Node.js is inline with the technologies used for the overall structure of the development project. However, there might be deviations from the library used for handling the matching service.

Socket.io was used based on the recommendation of the tutorial from the document provided in CS3219. The reason for using it was for different browsers (client) to be able to communicate with each other via a central server hosted in the backend. This is a requirement as each client needs to update their local statuses when they are either in a matching, matched or not matching status. Therefore, as Socket.io has the capability for browsers to communicate with each other at low-latency, we decided to use this library for handling the matching capabilities of our development project.

MongoDB was used instead of SQLite that was suggested in the tutorial. There are both functional and non-functional reasons for this decision.

The non-functional reason was that setting up an additional database software for future and current developers was thought to be wasteful. MongoDB also has cloud features which can allow future developers to easily setup the project without installing any additional software for databases. MongoDB is also able to handle the functionalities required for matching services to be handled.

The functional reason was that there might be a need for scaling if the project is being expanded upon. Extensions of the matching algorithm might happen which include, more options to filter matching user are added (age, tutor/student, language) and rating based or other complex algorithms to match users. The extension might result in queues extending past the current capacity resulting in needing an easily scalable database to handle this extension. MongoDB is easily horizontally scaling which can handle the extra load and there are no relational database functionalities required in the matching service. Therefore, we made a decision to choose MongoDB over SQLite.

Development Process and Design

Using External Database as Datastore

The socket.io server uses MongoDB as a datastore for the users in queue. The alternatives we considered is keeping a local copy for the server instance (as a data structure in the server) or not using any kind of datastore to keep track of users.

Keeping a local data structure for keeping track of users in a queue was not implemented due to the asynchronous processes in the server. As the can be handling multiple requests simultaneously, the read and write to the data structure must be able to handle asynchronous read and write situations. This means implementing a data structure that can perform bug-free async read and writes. However, MongoDB already supports asynchronous operations, we can just make use of MongoDB instead of spending time implementing a less robust data structure than MongoDB.

The other alternative was to make user of socket.io rooms to add users and use rooms as a stateless data store for matching users. For example, 3 rooms will be created with the names “Easy”, “Medium” and “Hard”, once there is 2 clients they would be evicted and set in a separate room with a shared room id for sharing question page data. However, we found that the issue was there was difficulty scaling as rooms are hardcoded server side and should there be a huge number of matching requests, there would be a slow down as a server has to handle all the extra logic. The order of users being entered in the queue was also not maintained in this alternative. Therefore, we decided to use MongoDB as the processing of storing user data is not put on MongoDB and the server only has to handle the requests.

Handling matching process server side

There is a consideration for peer-to-peer connection for matching services. A peer-to-peer service entails that each user would have both a server and a client for sending and handling requests to every other user. This way the matching is done independently by each user and does not use a centralised server. The advantage is that there would only be one client server pair per user for all cross user services and would not require multiple servers for different services.

However, peer-to-peer connection has its disadvantages, performance of each user is reliant on themselves and the connected user, security might be compromised should there be a bad actor and would break the microservices architecture. Therefore, we decided that a centralised server would be a better alternative.

Question Service

Overview

Question Service is a server that provides services pertaining to programming questions. In particular, it provides several endpoints to serve the entire catalog of questions, a specified question, or a random question. Question data can be fetched from the appropriate endpoints via HTTP GET requests.

Requirements

Functional Requirement	Justification	Priority
The system should provide the user with a list of questions upon request.	To allow the user to browse through the entire catalog of questions and potentially pick one to attempt.	High
The system should allow users to select a specific question upon request.	To allow the user to pick the desired question to attempt.	High
The system should allow the user to get a question based on the selected difficulty.	To allow the user to pick a question of the specified difficulty, without having to browse through the entire catalog of questions or pick a specific question.	High
The system should provide an option to get a random question.	To allow the user to pick a question of the specified difficulty, without having to browse through the entire catalog of questions or pick a specific question.	Medium
The system should provide an option to get a different question than before.	To allow the user to pick a random question without receiving the same question as the current question.	Medium

Non-Functional Requirement	Justification	Priority
The system should provide a question within 1 second.	To provide a fast and pleasant experience for the user.	High
The system should provide at least a total of 10 questions.	So users have more options to choose from, and are less likely to receive repeated questions too frequently.	Medium

Development Tech Stack

Technology Stack	
Frontend	React
Backend	Express.js/Node.js
Backend dependencies	Mongoose dotenv cors
Database	MongoDB
Project Management Tools	GitHub Issues

The selection of React and Express.js/Node.js is inline with the technologies used for the overall structure of the development project. MongoDB, similar to the rationale behind its use throughout the different services, is the database of choice due to its scalability and availability on the cloud.

The elaborations for the use of the below dependencies are as described in User Service. The summary of the rationale as follows:

Mongoose is used to facilitate and ease the interactions with MongoDB.

dotenv is used to facilitate the reading of environment variables stored in the .env file.

cors is used to allow communications between the backend server and the frontend.

Implementation Decisions

There are several major components that make up Question Service: Model, Repository, Controller, and the Router.

The Model component contains the Mongoose schema definition for the Question model, which contains the definitions of Question such as name and difficulty, as well as their corresponding data restrictions.

The Repository component contains the implementations of different functions that interact with the Question database. The functions provide core data-access functionalities such as fetching the desired Question records, and do so by interacting with the Question database via Mongoose. Since these functions heavily depend on the Question model, the Repository component is highly coupled to the Model component. As such, the Model and Repository makes up the Data access layer.

The Controller component contains the implementations of business logic. The Controller functions extract relevant information from the HTTP requests, and calls the relevant Repository functions to fetch the requested Question data. Afterwhich, the Controller returns the fetched data as response. If an error occurs, the response would contain the error message instead. The Controller interacts with the Repository through the Repository functions, and does not depend on their implementation. As such, a new database and Repository implementations can be readily used without affecting the Controller implementation, as long as the interface provided by the Repository remains the same, thus adhering to Dependency Inversion Principle. Separation of Repository and the Controller also separates object persistence from the business logic, thus promoting separation of concerns.

The Router component defines the paths of the API endpoints, and calls the relevant Controller function containing the desired business logic for each endpoint. Since the Router merely calls the Controller functions, it does not depend on the implementation of these functions. As such, the internal implementation of the Controller can be modified without affecting the Router, as long as the interface provided by the Controller remains the same.

Middlewares are also implemented to handle errors. For instance, accessing an unimplemented endpoint would return an HTTP 404 unknown endpoint message.

History Service

Overview

History Service is a server that provides services pertaining to history records of question attempts. In particular, it provides several endpoints to serve all history records, a specified history record, and the history records of a specified user. History Service also provides endpoints to create new history records, as well as delete existing ones.

Requirements

Functional Requirement	Justification	Priority	Status
The system should provide all history records pertaining to the specified user.	To allow the viewing of the entire history of attempted questions by the specified user.	High	Completed
The system should allow users to select a specific history record.	To allow the user to pick the desired history record to inspect.	High	Completed
The system should allow the creation of a new history record.	To allow new question attempts, as well as the participating users and submitted answers to be recorded.	High	Completed
The system should allow the deletion of history records pertaining to the specified user.	To allow records to be deleted from the database when they are no longer needed (eg. when the user account is deleted).	High	Completed
The system should store the following information in a history record: the user ID of interest, the collaborator name, the name of the attempted question, and the submitted answer, time of submission.	To allow the user to view the core information of the history record, without storing additional unnecessary information.	High	Completed

If a list of history records is requested, the system shall return the records in descending order according to the timestamps.	To allow the user to view the list of history records starting from the most recent (and thus likely more relevant) record, instead of records sorted in an arbitrary order.	Medium	Completed
The history record of a user should not be deleted even if the collaborator account has been deleted.	To allow the user to not lose any history record regardless of external factors (eg. the collaborator account being deleted)	Medium	Completed

Non-Functional Requirement	**Justification**	**Priority**
The system should provide history records within 1 second.	To provide a fast and pleasant experience for the user.	High

Technology Stack	
Frontend	React
Backend	Express.js/Node.js
Backend dependencies	Mongoose dotenv cors

Database	MongoDB
Project Management Tools	GitHub Issues

Development Tech Stack

The selection of React and Express.js/Node.js is inline with the technologies used for the overall structure of the development project. MongoDB, similar to the rationale behind its use throughout the different services, is the database of choice due to its scalability and availability on the cloud.

The elaborations for the use of the below dependencies are as described in User Service. The summary of the rationale as follows:

Mongoose is used to facilitate and ease the interactions with MongoDB.

dotenv is used to facilitate the reading of environment variables stored in the .env file.

cors is used to allow communications between the backend server and the frontend.

Implementation Decisions

Similar to the Question Service, there are several major components that make up History Service: Model, Repository, Controller, and the Router. The implementation and rationale of each component is similar to its counterpart in the Question Service.

The Model component contains the Mongoose schema definition for the history model, which contains the definitions of History such as user, collaborator, question name, submitted content, and timestamp, as well as their corresponding data restrictions.

The Repository component contains the implementations of different functions that interact with the History database and provide core data-access functionalities such as fetching the desired history records. The Model and Repository makes up the Data access layer.

The Controller component contains the implementations of business logic. The Controller functions extract relevant information from HTTP requests, and calls the relevant Repository functions to process the requested History data, and return the fetched data as response. If an error occurs, the response would contain the error message instead. The Controller interacts with the Repository through the Repository functions, and does not depend on their implementation thus adhering to Dependency Inversion Principle. Separation of Repository and the Controller also separates object persistence from the business logic, thus promoting separation of concerns.

The Router component defines the paths of the API endpoints, and calls the relevant Controller function containing the desired business logic for each endpoint. Internal implementation of the Controller can be

modified without affecting the Router, as long as the interface provided by the Controller remains the same.

Middlewares are also implemented to handle errors. For instance, accessing an unimplemented endpoint would return an HTTP 404 unknown endpoint message.

Chatting Service

Overview

Chatting service allows users to chat with one another while collaborating on code. Users are connected to one another based on the room ID they are matched into.

Requirements

Functional Requirement	Priority	Status
The system should allow both users to communicate with each other on the same page.	High	Completed
The system should show messages sent from each user.	High	Completed
The system should allow users to easily distinguish sent message and received message	High	Completed
The system should allow users to send the text written in the message box to the other user.	High	Completed
The system should show the name of the other user that is being matched with.	High	Completed
The system should allow both users to easily distinguish each message sent or received individually	High	Completed

User process and functional requirements

When the user enters the collaboration page, the chat app should appear on the collaboration page. The chat app would display the username of the other user on the top of the chat app.

The user can use the chat app to communicate with the other user that he is collaborating with. The user can send a message by typing in the text box in the chat app and clicking the send button or pressing enter.

Sent message and received message would be displayed in different coloured text boxes with sent messages aligned to the right while received messages are aligned to the left. Each sent message are contained in their own separate boxes for the user to easily distinguish where each message starts and ends.

Non-Functional Requirement	**Priority**	**Status**
The system should reflect sent and received messages in 1 second.	High	Completed
The system should auto scroll to the bottom so that new messages are visible.	Medium	Completed
The system should be able to handle sending of messages written in the English language	High	Completed
The system should not be resource intensive and affect the core functionalities of the web app	High	Completed

User process and non-functional requirements

Users communicate with each other via socket.io client and server services. As the server processes each message asynchronously and has low-latency, messages are sent and received instantaneously and should take no more than 1 second. The chat app also automatically detects if a new sent/received message is not displayed, to scroll the window to the bottom so new messages are displayed.

As the development project has all components in English, we expect the majority of users to communicate in English. As such that chat app should be able to handle English language-based communication. Since the text is transmitted from user1 to server to user2, if the underlying programming language (JavaScript) is able to handle scripts from the English language, then the chat app should not have a problem in handling English scripts.

The overall messages sent and received are stored locally for each user. Which means that after a message is sent, each local storage will be updated with the new message. This ensures that there is no delay from accessing a database for CRUD operations. This prevents excessive processing from being used on the local program, preventing resources from being hogged by the chat app.

Technology Stack	
Frontend	React

Backend	Express.js/Node.js
Library	Socket.io
Project Management Tools	GitHub Issues

Development Tech Stack

The selection of React and Express.js/Node.js is in line with the technologies used for the overall structure of the development project. However, there might be deviations from the library used for handling the chatting service.

Similar to the matching services, Socket.io library was used for different browser (client) can communicate with their matched partner. We make use of Socket.io implementation of client, server, rooms and async capabilities to implement the chat application.

Each client is assigned a shared room id given by matching-service and passes it to the chatting-service server so that the correct partners are joined in the same room on the server. After which, each client sends messages to the server which would update information for the room and reflect the messages sent/received. Lastly, async capabilities means that the server can handle multiple requests from different rooms and update them without delay.

Development Process and Design

Using a datastore for chat messages

There was consideration to keep the messages in a centralised location like MongoDB and have the server update it whenever there are messages sent and update the values of local copies afterwards. However, after some experimentation, we found that there were some issues arising from doing so.

Firstly, multiple processes before users can see an update being reflected. The server now has an additional step of updating the database. Seeing that the chat app might have a high number of requests, this might cause a bottleneck as numerous requests would be sent to the database to get and update the data. Resulting in a slow down for the local users due to the load on the system.

Second, if the database reaches capacity, what eviction method should we decide on. As deleting messages after they reach a certain capacity for each room would be ideal, it would be process heavy and might interfere with the chatting capacity. Also, when a room closes, there needs to be processes to clean up the DB to make sure there is no stale data.

Lastly, how to handle errors from the database. Should the data retrieved from the database be corrupted, how can we recover from this situation as local copies are reliant on the database copy.

As we were not able to resolve these issues with a satisfactory solution, we decided to remove the complexity of adding a data store for the chatting service.

Loss recovery for unexpected disconnections

The current implementation does not provide loss recovery. The situation happens when a user disconnects or loses connection to the current page and a message is sent, only the user who sent the message would see it.

The method proposed to solve loss recovery is to implement a Redis cache for storing local copies that can be quickly and easily checked against the local copy and updated. However, as discussed in the datastore segment above, we would then be saddled with a bunch of processes that need to be handled.

The problems mentioned in the previous segment can be easily solved by having multiple Redis instances handling an appropriate number of rooms and cleaning up after each room leaves. However, the complexity of the solution makes it difficult to test and implement and thus should be left to future extension should there be a scale needing Redis to be implemented.

Collaboration Service

Overview

Collaboration service allows users to collaborate while writing code, within a collaborative code editor that both users are able to edit concurrently and instantaneously.

Requirements

Functional Requirement	Justification	Priority
The system should show changes made by the other user	To allow the other user to type in his/her input to contribute to the solution for the question	High
The system should show changes made by the current user	To allow the current user to type in his/her input to contribute to the solution for the question	High

The system should be able to update the changes made to the text box in real time	To allow both users to collaborate and understand what is being typed by each other.	High
The system should show where the other user cursor is currently at	To allow both users to understand where the other user is typing currently to prevent them from typing over each other	Low
The system should be able to resolve conflict in changes and show the correct changes made by both users	To allow users to collaborate in real time without their typed answers from disappearing from the text box	High
The system should be able to highlight code text properly	To allow users to read the typed code easily as syntax is highlighted correctly	Medium

Non-Functional Requirement	Justification	Priority
The system should update the text box with 0.5 seconds.	To allow both users from typing simultaneously instead of waiting for the text box from updating the new input	High
The system should be able to extend the text box such that long lines of code can be written.	To not restrict the user from typing out long solutions or other text in the text box without seeing the new or old texts	Medium
The system should keep the text shared between each collaborator only and nobody should have access to the typed text	To allow privacy for both the collaborators so that their solution would not be exposed to outsiders who do not have the privilege	High

Technology Stack	
Frontend	React
Backend	Express.js/Node.js
Library	yjs codemirror y-webrtc
Project Management Tools	GitHub Issues

Development Tech Stack

The selection of React and Express.js/Node.js is in line with the technologies used for the overall structure of the development project. However, there might be deviations from the library used for handling the collaboration service.

Codemirror was used as it provides code highlighting services, user cursor indicator that would be helpful in our web application.

Originally the prototype of the collaboration service was done with Socket.io with a centralised server to provide updates based on the inputs of each user. However, the problem we realised was that users would receive updates slower than expected and the state of the text box might be rewritten to a previous state that was not updated yet. Simultaneous updates from both users also result in a loop where an update causes a new update and repeats infinitely. The problem was how conflict in text box content was resolved which results in the usage of yjs.

Yjs uses a CRDT algorithm (Conflict-free replicated data type) which resolves conflict based on mathematical models instead of index based or a main document as a central source of truth. Thus by establishing a webrtc room to connect to and linking the text box to the room, users can collaborate with each other through the algorithm and receive minimal conflict errors.

Development Process and Design

Using Socket.io as a collaboration library

As mentioned in the Tech Stack segment, we intended to use socket.io for the collaboration textbox. This is due to not being reliant on yjs webrtc servers and having full control of the features and parts of the text box. However due to the errant results and the need to implement a resolving mechanic for conflicts, we decided against using Socket.io to implement the collaboration text box.

Reflections

To develop our PeerPrep, there were many learning points along the way.

It was imperative to take a step beyond what was learnt in class, utilizing those learning points and doing our own research in order to apply that knowledge into practical use. For example, socket.io was merely suggested as a possible technology for pub-sub messaging, but we heavily applied it within our application, such as in matching service, collaboration service, and chatting service. Through trial and error and extensive online research, the services are working, and we were able to take that knowledge beyond the classroom and put it to actual use.

Time management was also important in the process. It was hard to juggle developing a full web application while working on other projects and modules. Planning and managing what services to implement next and how long it is expected to take was key to developing the project in time for demos. Meetings were short, effective, and to the point, and task delegation was done to ensure each member had a part to play in the project. Thankfully, each member was able to complete their designated task before the specified deadlines, thus allowing parallel development of the various services to be completed smoothly, and eventually be pieced together to form a coherent system.

Lastly, the code written was very readable so other developers would be able to work on it in the future without difficulties. Moreover, documentation in the root folder in the form of a README gives clear instructions on how to run the service in each folder, or to run the entire application using docker.

Team Contributions

Team Members	Contributions
Oei Yi Ping	User service Question service History Service Frontend test and debug Local docker deployment
Chua Jun Jie Benedict	Matching service Collaboration service Chatting service Frontend test and debug
Tan Jing Yi Jace	Frontend UI Frontend-backend integration Frontend test and debug
Rachel Gina Abelarde	Frontend UI Frontend-backend integration Frontend test and debug