CS3219: Software Engineering Principles and Patterns

AY22/23 Semester 1

Project Report

Team 13

Name	Student Number
Bryan Ong WeiXin	A0227253J
Tiew Wei Jian	A0218321R
Sourabh Raj Jaiswal	A0226598M
Yong Zi Xin	A0221778X

Table of Contents

Background

This project is an adaptation of the project PeerPrep that has been completed in the previous semesters of CS3219.

There is an increasing trend of companies conducting technical interviews as part of the interview process. Websites like Leetcode, Geekforgeeks and Kattis are increasingly popular among job seekers to prepare for interviews. While the resources are available for practice, it is a steep learning curve for many who are not well versed in algorithms. Many would find it difficult, and eventually give up. How can we help job seekers, and even undergraduate students tap on the resources available online to train for technical interviews more effectively?

In recent years, pair programming is also gaining popularity, and proved to be effective for many reasons. The collaborative nature of pair programming enables programmers to tap on each other's strengths and learn from one another.

To solve this issue, we embarked on this project to create **CodeUs**, a collaborative technical interview platform where one can receive a coding question and collaborate with someone to answer the question together.

The general idea of the application is as follows:

An user can sign up for an account on the platform with basic authentication. The services will allow them to manage their account, like changing their passwords, or eventually deleting the account if they no longer require the account.

The user will be able to use the application by choosing the level of difficulty they will like to attempt. The application will attempt to find a match for the user within 30 seconds. If a match is found, the current user and their matched user will be brought to a page where two different technical questions will be displayed to them. The two users will be able to choose either question, or both, to attempt together in real time with the help of a real-time collaborative text editor and a chat widget.

Once one of the users decides to end the session, both of the users will be brought out of the page, and they can reselect their difficulty level and find a new match.

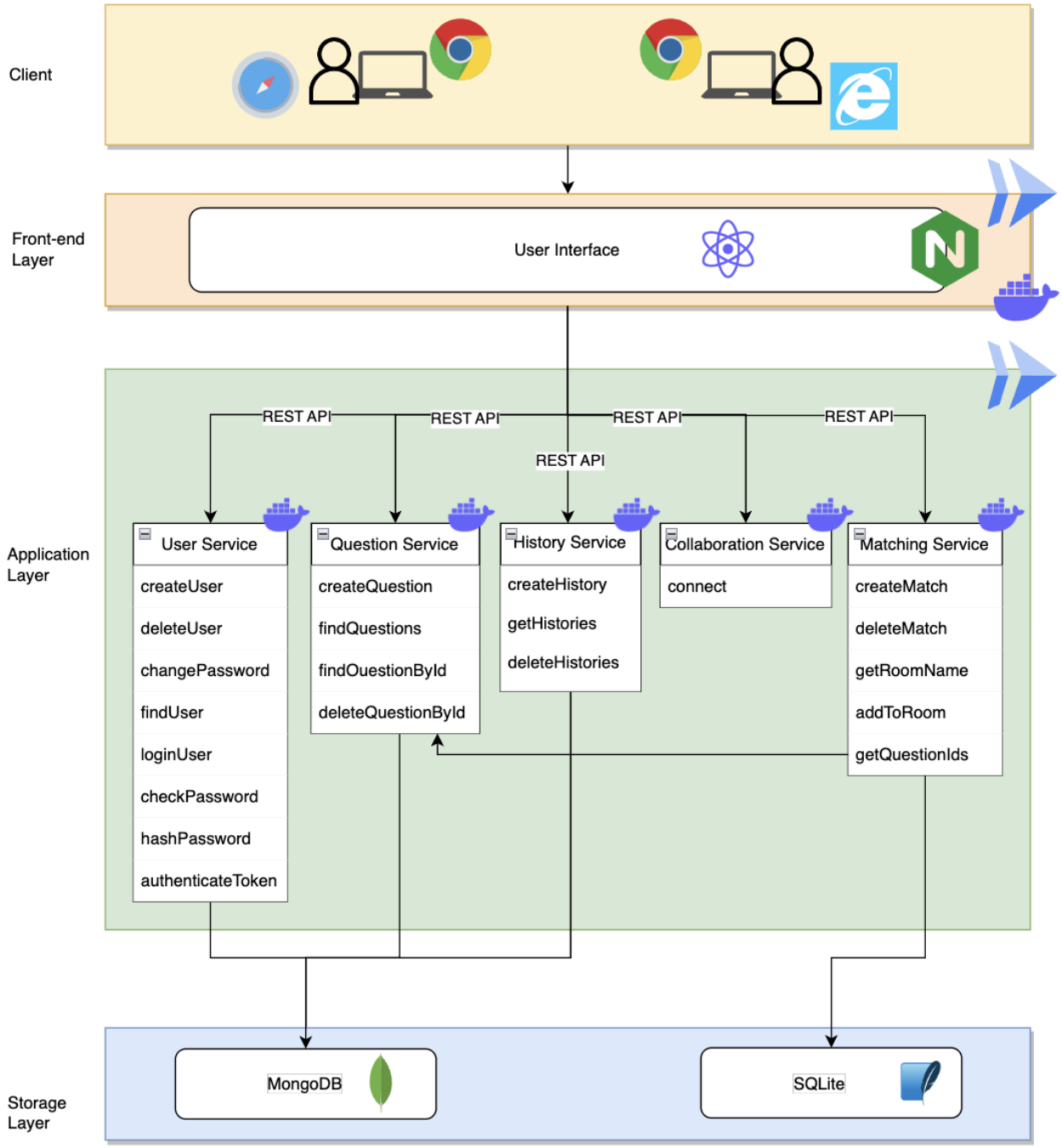
Purpose

The purpose of this project is to create a web application, **CodeUs**, to help job seekers, undergraduate students to prepare for technical interviews through collaborating with people in real time.

This can be achieved by providing a bank of algorithmic questions, and a collaborative text editor for matched users to work together. On top of that, we would also implement a user system where users are required to sign up for an account to verify their identity and keep track of questions attempted.

Developer Documentation

System Architecture Overview



All the docker built services are run on separate Cloud Run instances and the frontend is serviced via nginx reverse proxy to fetch the frontend from port 3000.

Design Principles

The first step of designing the system was to decompose the entire system's requirements into different subcomponents, and group them together into different microservices. We decided to perform vertical slicing by features, where we identify requirements that are functionally related and group them together while keeping other requirements away. Our group focused on the functional cohesion of the services, where every service should have a clear and unique responsibility. For example, requirements related to users' are grouped together and are separated from requirements related to answering the question collaboratively. The grouping of requirements can be seen across the different system components, which are elaborated below.

The functional grouping of requirements helps us to reduce coupling between the different services as much as possible as they do not need to interact with each other. The reduced coupling provides greater flexibility and reusability for our code.

When it comes to the implementation, our group made use of different design patterns. The Facade pattern was one of the main patterns used by our team while scaling up our project as there were many subcomponents involved. In each subsystem, we provided REST APIs for each subsystem to be used, without the implementation known to the end user. The User Service uses such a design pattern, as the database query is complicated, but the end user will only need to access the REST API in order to utilise the User Service.

There is also the use of the Observer pattern. One such example is In the Matching Service. The Matching Service acts as the publisher, awaiting for users request's to find a match. The users in this case are the subscribers, and once a match is found, it will notify the subscribers, and also allocate them a virtual room to begin collaboration. Another example would be the Collaboration Service. In the Collaboration Service, an user acts as both the publisher and subscriber, where they subscribe to listen to events from each other, such as sending and receiving a chat message.

System Components

For this project, we have implemented 5 different independent services — User, Matching, Question, Collaboration, and History each with their own purpose. All these services work together in the Frontend of the CodeUs web application, which is the main platform for users to use the services in a meaningful way. Each component is described below along with their individual requirements, design, technical stack and other information relevant for the component.

User Service

The user service is responsible for creating and maintaining information about users of the application. It allows persistent storage of user information in the cloud.

The requirements identified for the User Service are as follows:

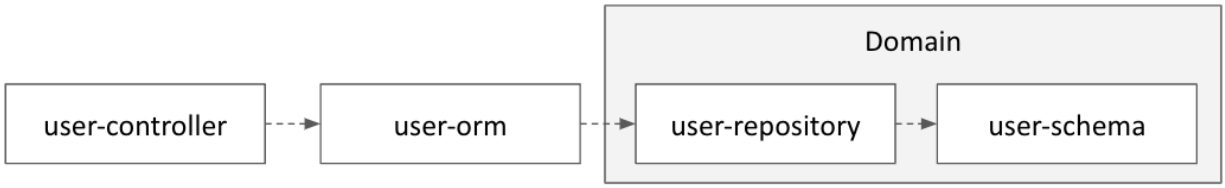
	Functional Requirements	**Priority**
1	The system should allow users to create an account with username and password.	High
2	The system should ensure that every account created has a unique username.	High
3	The system should allow users to log into their accounts by entering their username and password.	High
4	The system should generate a JWT after verifying the user's credentials.	High
5	The system should be able to verify the validity of a JWT.	High
6	The system should allow users to log out of their account.	High
7	The system should allow users to delete their account.	High
8	The system should allow users to change their password.	Medium

	Non-Functional Requirements	**Priority**
1	User accounts should be persistent and accessible across all devices.	High
2	Users' passwords should be hashed and salted before storing in the database.	Medium
3	Users' passwords should be caps-sensitive, and usernames should not be.	Medium

Design of User Service

The User Service makes use of a Domain Driven Design, where the domain is the Users represented as documents in MongoDB using the Mongoose Object Document Mapper (ODM). It utilises an Object Relational Mapper (ORM) to translate between an user object and the relational representation in the database.

The overall design of the User Service is illustrated as follows:



user-controller: Provides the different endpoints for clients of the API to access.
user-orm: Translates between an user object, consisting of username and password, and the relational representation (model) of user data in the database.
user-repository: Provides different methods to interact with the users database.
user-schema: Defines the schema of an user document in the database.

Technical Stack

ExpressJS: Framework to provide the different API endpoints.
MongoDB + *MongooseODM*: To store the user data in the cloud and verify validity of the data.
bcrypt: To encrypt the user's password before storing in the database.
jsonwebtoken: To generate and verify JWT tokens for logged in users.

API Endpoints

The different API endpoints provided by the User Service are summarised below:

`POST /`	Body: `{ username, password }` Creates a new user in the database if the username does not exist
`DELETE /`	Body: `{ username, password }` Deletes the user if the username and password matches
`POST /login`	Body: `{ username, password }` Verifies the user credentials and return a JWT if valid
`POST /check`	Body: `{ token }` Checks if the specified JWT token is valid
`POST /change_password`	Body: `{ username, oldPassword, newPassword }` Changes the password of the user if the old password is correct

Matching Service

The matching service enables users to find another user of the same selected difficulty level, and match them together.

The requirements identified for the matching service are as follows:

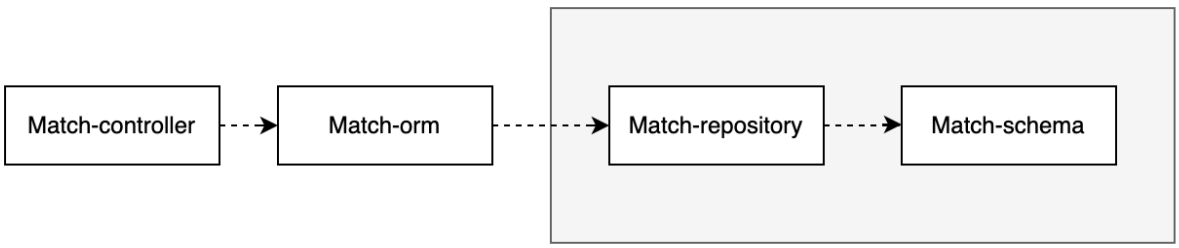
	Functional Requirements	**Priority**
1	The system should allow users to select the difficulty level they want	High
2	The system match two users with the same selected difficulty level	High
3	The system should attempt to find a match for a user within 30 seconds	High
4	The system should allow the two matched users to enter the same room	High
5	The system should allow user to leave and remove the match	High

	Non Functional Requirement	**Priority**
1	The matching data stored should be persistent	High
2	The matching data should be removed from the database after the user cancel or exit the room	Medium
3	The two users should enter and exit the room page at the same time	Medium

<u>Design of Matching Service</u>

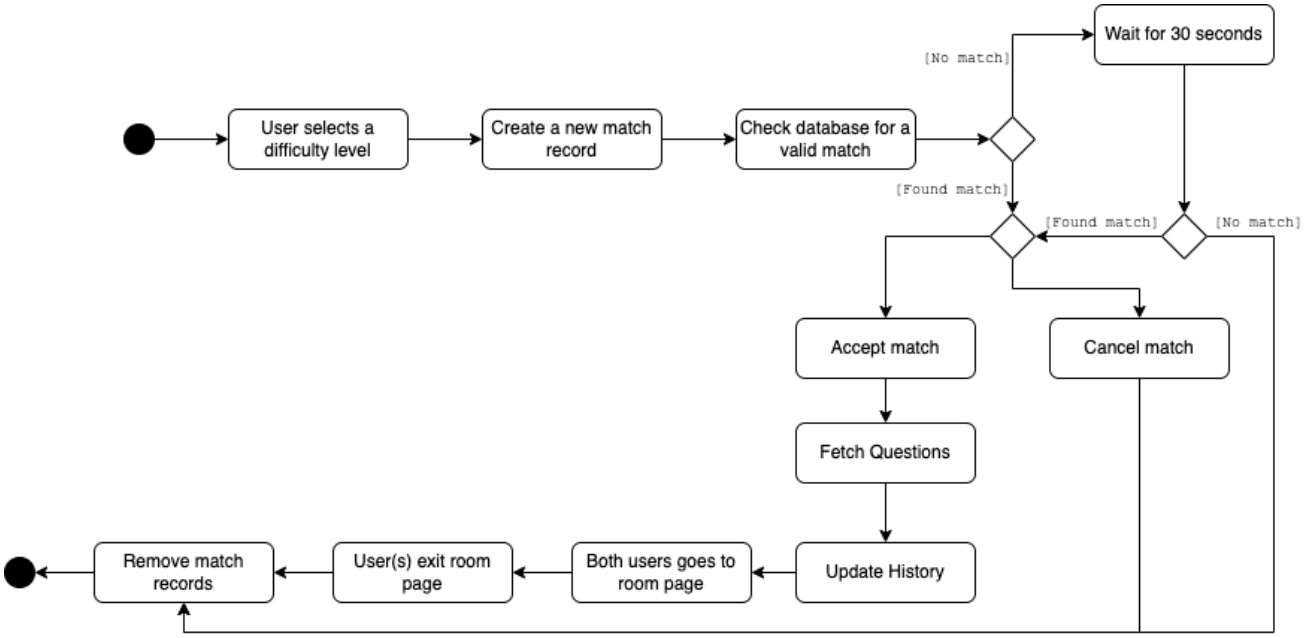
The Matching Service makes use of a Domain Driven Design, where the domain is the Match records represented as documents in SQLite. It utilises Sequelize, an Object Relational Mapper (ORM) to translate between a match object and the relational representation in the database.

The overall design of the Matching Service is illustrated as follows:



match-controller: Provides the different endpoints for clients of the API to access.
match-orm: Translates between a match object, consisting of a userOne, userTwo, difficulty, userOneId, userTwoId and the relational representation (model) of match data in the database.
match-repository: Provides different methods to interact with the match database.
match-schema: Defines the schema of a match document in the database.

The overall flow from creating a Match object to deleting the Match object is illustrated as follows:



Technical Stack

ExpressJS: Web-application framework for Node.js
Sequelize: Node.js based Object Relational Mapper (ORM) handles the database records by representing the data as objects
SQLite3: Relational database engine to store the match objects
Socket.io: A library to enable low-latency, bidirectional and event-based communication between client and server

Events

Matching service listens to the following events and provides the corresponding services:

`match`	Parameters: `{ userOne, difficulty, socketIdOne }` Creates a new match record for `userOne` and finds any valid match. If a valid match if found, add both users to a room
`removematch`	Parameters: `{ user }` Deletes the match record for `user`
`start`	Emit '`partner start`' event to the client socket which triggers the navigation of users to room

Question Service

The Question Service provides different functionalities to add questions to the question bank and retrieve questions .

The requirements of the question service is as follows:

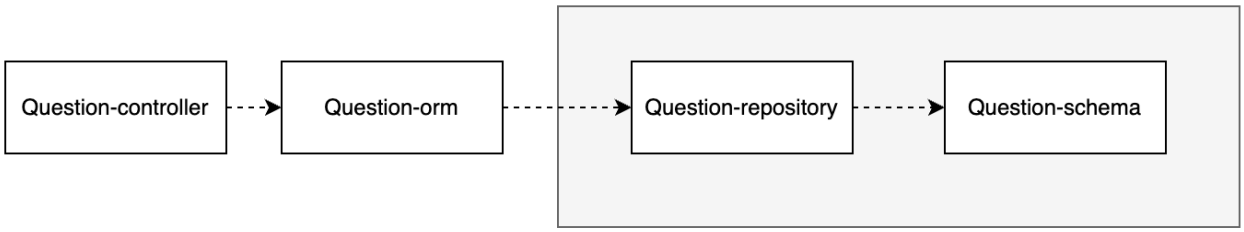
	Functional Requirement	**Priority**
1	The system should be add a question which has a title, body and difficulty	High
2	The system should return two questions of the given difficulty for two users	High
3	The system should fetch a question with the question id	Medium
4	The system should be able to delete a question	Low

	Non Functional Requirement	**Priority**
1	The question should be same after accidentally reloading the page	High
2	The pair of users should be receive the set of questions within 1s	High

Design of Question Service

The Question Service makes use of a Domain Driven Design, where the domain is the Questions represented as documents in MongoDB using the Mongoose Object Document Mapper (ODM). It utilises an Object Relational Mapper (ORM) to translate between a question object and the relational representation in the database.

The overall design of the Question Service is illustrated as follows:



question-controller: Provides the different endpoints for clients of the API to access.
question-orm: Translates between a question object, consisting of title, body and difficulty, and the relational representation (model) of question data in the database.
question-repository: Provides different methods to interact with the question database.
question-schema: Defines the schema of a question document in the database.

Technical Stack

ExpressJS: Web-application framework for Node.js
MongoDB + *MongooseODM*: To store the question data in the cloud and verify validity of the data.

API Endpoints

The different API endpoints provided by the Question Service are summarised below:

`POST /`	Body: `{ title, body, difficulty }` Creates a new question in the database if the title does not exist
`GET /id`	Param: `{ id }` Returns the question document with the given id
`GET /`	Param: `{ difficulty }` Returns two questions with the given difficulty
`DELETE /id`	Param: `{ id }` Deletes the question with the given id

Testing

The different API endpoints are tested using *Mocha* and *Chai* which cover:

- ☐ *Creating* valid and invalid questions
- ☐ *Retrieving* an existing question with id
- ☐ *Retrieving* a pair of questions of the given difficulty
- ☐ *Delete* a created question
- ☐ *Delete* a non existing question

Collaboration Service

The Collaboration Service provides different functionalities to enable matched users to collaborate and work on the question together. It is responsible for sharing information between the matched users such that the users can get the same question to do, share their code using the real-time collaborative text editor, and also send messages to each other.

The requirements of the collaboration service is as follows:

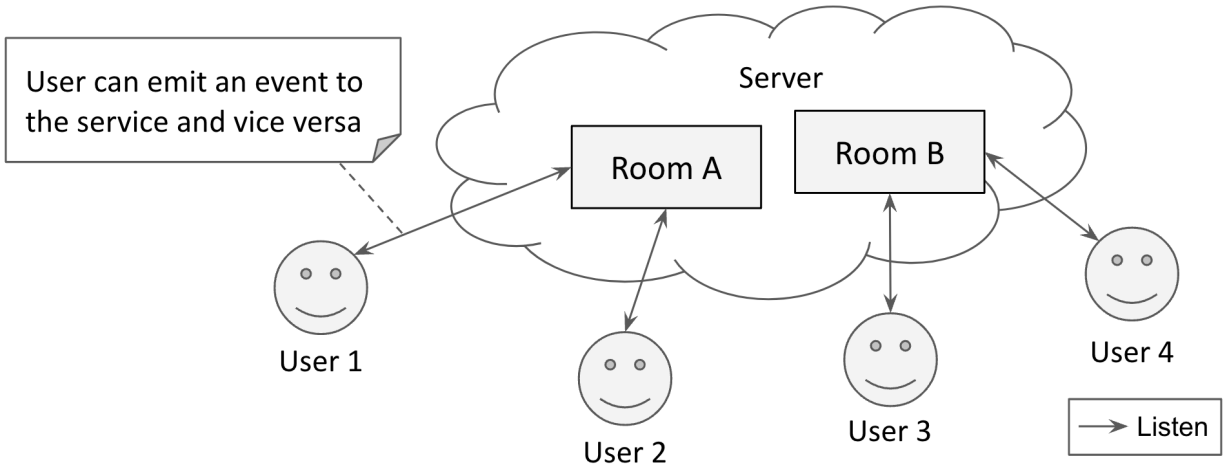
	Functional Requirement	**Priority**
1	The system should be able to bring matched users into the same room	High
2	The system should allow both users to get the same question	High
3	The system should allow user to view the content that the other party entered into the text editor	High
4	The system should allow users to send messages to their partner	Medium
5	The system should be able to notify the user if their partner has exited	Medium

	Non Functional Requirement	**Priority**
1	The collaborative text editor should not have delays that exceeds 1 second	High
2	The users should be able to type into the text editor simultaneously	High
3	The users should not be disconnected before they exit	High

Design of Collaboration Service

The Collaboration Service provides a server consisting of different rooms. An user can connect to the service by providing the name of the room. Matched users will join the same room.

The overall design of the Collaboration Service is illustrated as follows:



After joining a room, the server will be listening to events from the user, and the user can listen to events from the server, thus both the users and server will be able to emit an event to each other. This allows users to share information with their partner who is in the same room. The user first emits an event to the server, which is listening to the event. After receiving the event, the server will emit a corresponding event to the other user(s) in the same room as the sender.

Technical Stack

For the Collaboration Service, we used *ExpressJS* as the web-application framework. We used *Socket.io* to emit and listen to the different events within the different rooms.

Events

Collaboration Service listens to the following events:

`join room`	Parameters: `{ roomId }` Adds the incoming socket to the room specified
`exchange question`	Parameters: `{ roomId, question }` Emits `receive other question` to the room specified
`send message`	Parameters: `{ roomId, sender, message }` Emits `receive message` to the room specified
`exit`	Parameters: `{ roomId }` Emits `partner exit` to the room specified

History Service

The History Service provides different functionalities to track the previous questions and attempts made by the user

The requirements of the question service is as follows:

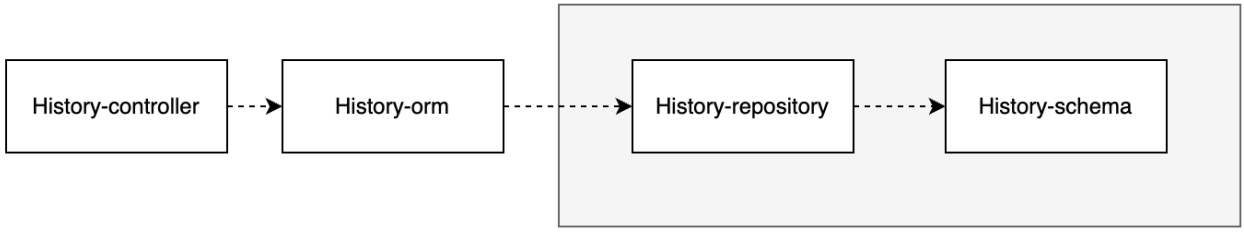
	Functional Requirement	**Priority**
1	The system should add a history which has both usernames, difficulty and question name and question id	High
2	The system should allow users to view their history	High
3	The system should delete the history for the user once the account is deleted	Medium

	Non Functional Requirement	**Priority**
1	The history should be loaded within 1s of opening the home page	High

<u>Design of History Service</u>

The History Service makes use of a Domain Driven Design, where the domain is the History represented as documents in MongoDB using the Mongoose Object Document Mapper (ODM). It utilises an Object Relational Mapper (ORM) to translate between a question object and the relational representation in the database.

The overall design of the Question Service is illustrated as follows:



history-controller: Provides the different endpoints for clients of the API to access.
history-orm: Translates between a history object, consisting of username, matchedUsername, history, questionName and questionId, and the relational representation (model) of history data in the database.
history-repository: Provides different methods to interact with the history database.
history-schema: Defines the schema of a history document in the database.

Technical Stack

ExpressJS: Web-application framework for Node.js

MongoDB + *MongooseODM*: To store the history data in the cloud and verify validity of the data.

API Endpoints

The different API endpoints provided by the History Service are summarised below:

`POST /`	Body: `{ username, matched, difficulty, questonName, questionId }` Creates a new history in the database
`GET /`	Param: `{ username }` Returns the history for the given username
`DELETE /`	Param: `{ username }` Deletes the history for the given username

Testing

The different API endpoints are tested using *Mocha* and *Chai* which cover:

- ☐ *Creating* valid and invalid history
- ☐ *Retrieving* existing histories for a username
- ☐ *Retrieving* non existing histories
- ☐ *Delete* history for a username
- ☐ *Delete* history for a non existing username

Frontend Service

The frontend service provides the user interface for users to make use of the different services.

The requirements of the frontend service is as follows:

	Functional Requirement	**Priority**
1	The system should allow users to sign up for an account	High
2	The system should allow users to log in with their account credentials	High
3	The system should allow users to change their password	High
4	The system should allow users to delete their accounts	High
5	The system should allow users to select the level of difficulty they want	High
6	The system should show the amount of time left when searching for a match	High
7	The system should inform that user if a match is found or not found	High
8	The system should show two interview question based on difficulty selected	High
9	The system should show both matched users the same question	High
10	The system should allow both users to type code and collaborate together	High
11	The system should allow users to leave the matched session	High
12	The system should automatically start an user if their partner has started	Medium
13	The system should inform the user if their partner has left the session	Medium
14	The system should allow users to send messages to their partner	Medium
15	The system should display the number of solved problems for each difficulty	Medium
16	The system should allow user to change text style in the code editor	Low
17	The system should have a landing page to provide information about the app	Low

	Non Functional Requirement	**Priority**
1	Interface should be responsive and respond within 1s after user interacts with it	High
2	Textfield should hide password when entering password	Medium

Design of Frontend Service

Each page in the frontend is a component by itself. It is handled by its respective .js file. Certain components that are used in several pages, such as the AppBar are separated out as a component by itself.

For our frontend, we have the following pages:

- *Landing Page*: Show app information when the user first access the application
- *Sign Up Page*: For users to create an account with a new username and password
- *Sign In Page*: For users to log in with their username and password
- *Home Page*: Main page of the application with different functionalities:
    - Allow users to change their password
    - Allow users to delete their account
    - Allow users to log out of the current session
    - Show users their match history (individual questions attempted and summary)
    - Allow users to select a difficulty level to look for a match
- *Room Page*: Allow users to collaborate on the questions
    - Users can type code together using a real-time collaborative code editor
    - Users can send messages to each other using a chat widget

Technical Stack

We used *ReactJS* to build our frontend user interface, with components from *Material UI* and some *Cascading Style Sheets (CSS)* elements. We used *Axios* to call our backend API, and *Socket.io* to communicate with our *Socket.io* server. For the real-time collaborative text editor, we used *Quill Editor* for the editor interface, and *YJS* with *y-webrtc* provider to connect the matched users. For the chat widget, we used *react-chat-widget* as the interface.

Routes

The frontend service provides the following routes:

`/`	The landing page of the application
`/signin`	The sign up page of the application
`/signup`	The sign in page of the application
`/home`	The home page of the application Users will be redirected to `/signin` if accessed without logging in
`/room`	The room page of the application Users will be redirected to `/signin` if accessed without logging in

Deployment

We have utilised continuous deployment for the microservices in this project. Upon a successful pull request to the main branch, the Docker image will be rebuilt based on the updated codes, and pushed to the container registry in the Google Cloud Project. The image will then be deployed to Google Cloud Run. Cloud Run allows for easy deployment of microservices. The deployed microservices scale automatically based on the number of incoming requests without the need to configure or manage Kubernetes cluster.

We have utilised continuous deployment for the frontend in this project. The React app is run with Nginx and deployed to Cloud Run.

Suggested Improvements

Other methods of authentication

One improvement we could potentially make is to allow the user to sign up for an account in our application through other social media accounts, such as Google, Facebook, or even link their Github account. Through this, users will be able to log in with ease, without remembering their password.

Friends and Scoreboard

From the previous feature improvement, we could expand it further by including a feature where users can find friends through our platform, and potentially have a scoring system. (Eg. finishing a question successfully and the user gets awarded a certain amount of points). This could motivate users to finish more questions, and create a healthy competition among their peers.

Online Compiler

The collaborative text editor on our platform currently is a plain text editor. We could extend it further to include a programming language compiler (eg. Java), and allow the users to test the codes from the website itself on the go. Users would be able to verify their answers, without copying their codes to their integrated development environment to verify their answers.

Individual Contributions

Name	Contribution
Bryan	- Implement deletion of users in User Service - Implement YJS for code collaboration in frontend - Implement Collaboration service - Implement components in the room page, including code editor using Quill Editor and chat widget to send messages inside the room page - Add timer dialog when user is looking for a match - Clean up code base and fix warnings - Minor UI improvements - Created slides for milestone 2 and 3 presentation
Wei Jian	- Frontend service for login and signup pages - Frontend service to implement AppBar for some pages - Set up of MongoDB Cluster for Team Project - Collaboration service research and implementation - Managed and ensure there are no outstanding pull requests on our team repo - Set up team meetings and soft deadlines for team - Manage team report for final submission - Update documentation of project on repo README to provide instructions to run project locally/deployed
Sourabh	- Implement logging in, finding an existing user and creation of access token in User Service - Implement Question Service - Implement History Service - Set up automatic testing for question service and history service - Implement common components Formatter(getting HTML equivalent for a question) and Section - Implement Question design. - Design and build the Landing Page
Zi Xin	- Implement password hashing and storing in User Service - Implement change password in User Service - Implement Match design - Set up of SQLite database for storing of Match objects - Implement Matching Service - Implement adding of matched users to room in Matching Service - Frontend service for rendering of question - Set up continuous deployment for all backend services and frontend to Google Cloud Run

Team Reflection

As a team, when we were given the project brief, it seemed daunting at first. While there were many technical stacks that were very new to us, as a team, the biggest challenge was breaking the ice in the team and getting to know one another. Along the weeks, we slowly got to know one another better, and understood each other's strengths and weaknesses. We realised that the ability to work together as a team could not be underestimated, as we had to be in constant communication with one another as the project scales up.

One of the important lessons that we learnt was the ability to let go of our existing codes, and to be open to new ideas and technical stacks. One such example was the Collaboration Service. We had an existing implementation of the collaborative text editor at the earlier days of our project, but it proved to be not as reliable.

Working towards the final milestone, we had a short meeting to decide whether we should go ahead with the "not as ideal" implementation, or to start from scratch again. It was a difficult decision, but we chose the latter and it was the right decision. We redid the implementation of the text editor, and it was much better than the previous implementation. That required the diligence and perseverance of the team, as we were working towards a tight deadline.

Working together as a relatively new team proved to be a joyful experience, and an experience that we will carry forward in our internships and future careers. Through that, we learn to work with people with different backgrounds and skills, we learn to work together with an Agile software lifecycle, and we learn to enjoy the process of building from scratch. We learnt many new technical stacks that are used in the industry, and would be very helpful for us