NUS
National University of Singapore

School of Computing

CS3219 Software Engineering Principles and Patterns

AY22/23 Semester 1

Project Report – System Overview

Group 60

Team Members	Student No.	Email
Christian Drake Martin	A0200700H	christian.drake25@u.nus.edu
Woo Jian Zhe	A0199741N	e0406722@u.nus.edu
Florencia Martina	A0200760W	e0407741@u.nus.edu
Erin May Gunawan	A0200765L	erinmayg@u.nus.edu

Mentor: Foo Kai En

Table of Contents

Project Introduction

Background

Students often face challenging technical interviews when applying for jobs. There are many factors that cause this, ranging from the lack of communication skills to articulate their thought process to the inability to understand and solve the given problem. Furthermore, grinding practice questions can be tedious and monotonous.

Purpose

PeerPrep is an interview preparation platform and peer matching system where students can find peers to practice whiteboard-style interview questions together. The purpose of this application is to help students better prepare themselves for technical interviews. We will try to achieve this goal by providing questions with diverse algorithmic concepts and a peer learning system as a platform to revise concepts, learn from each other, and break the monotony of revising alone.

Functional and Non-Functional Requirements

User Service

S/N	Functional Requirement	Priority	Level
FR1.1	The system should allow users to create an account with username and password.	Must Have	High
FR1.2	The system should ensure that every account created has a unique username.	Must Have	High
FR1.3	The system should allow users to log into their accounts by entering their username and password.	Must Have	High
FR1.4	The system should allow users to log out of their account.	Must Have	High
FR1.5	The system should allow users to delete their account.	Must Have	High
FR1.6	The system should allow users to change their username.	Must Have	Medium
FR1.7	The system should allow users to change their password.	Must have	Medium
FR1.8	The system should not allow users to login or access API with the same token again	Must have	Medium

S/N	Non-Functional Requirement	Priority	Level
NFR1.1	Users' passwords should be hashed and salted before storing in the DB.	Must Have	Medium
NFR1.2	Users should not be required to manually login if the last time they logged in was within two hours.	Nice to Have	High
NFR1.3	The app should not allow user to view another user's data	Must Have	High

Matching Service

S/N	Functional Requirement	Priority	Level
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	Must Have	High
FR2.2	The system should be able to match two waiting users with the same difficulty levels and put them in the same room.	Must Have	High
FR2.3	If there is a valid match, the system should match the users within 30 seconds.	Must Have	High
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	Must Have	High
FR2.5	The system should alert the user when a peer has left/disconnected from the room and provide options for the user to remain or find another match	Must Have	High
FR2.6	The system should provide a means for the user to leave a room once matched.	Must Have	Medium
FR2.7	The system should provide an adjustable countdown timer.	Must Have	Medium

S/N	Non-Functional Requirement	Priority	Level
NFR2.1	The system should be able to find a match faster than 30 seconds.	Must Have	High

NFR2.2	The system should not allow other users to enter an occupied room (of two matched users)	Must Have	High

Question Service

S/N	Functional Requirement	Priority	Level
FR3.1	The system should provide options of different kinds of coding question types based on the difficulty	Must Have	High

S/N	Non-Functional Requirement	Priority	Level
NFR3.1	The question should load in < 3 seconds.	Must Have	High

Collaboration Service

S/N	Functional Requirement	Priority	Level
FR4.1	The system’s rooms should have at least 3 panels: one to view the question, one code editor, and the last for communication.	Must Have	High
FR4.2	The system should notify the users when the session time’s almost up.	Nice to Have	High
FR4.3	The system should provide shortcut functions like in IDE (e.g. to move lines up and down, to duplicate lines, etc.)	Nice to Have	Low

S/N	Non-Functional Requirement	Priority	Level
NFR5.1	The collaboration space should be updated in < 1 second.	Must Have	High
NFR5.2	The system should retain the typed code when the page is refreshed.	Nice to Have	High

Communication Service

S/N	Functional Requirement	Priority	Level
FR6.1	The system should have a communication channel (i.e. chat box or video chat).	Must Have	High

S/N	Non-Functional Requirement	Priority	Level
NFR6.1	The communication space should be updated in < 1 second.	Must Have	High
NFR6.2	The system should retain the chats when the page is refreshed.	Nice to Have	High

History Service (Nice to have)

S/N	Functional Requirement	Priority	Level
FR7.1	The system should allow the users to view their previous attempts' history.	Nice to Have	Low

S/N	Non-Functional Requirement	Priority	Level
NFR7.1	The system should not allow users to view another (unrelated) user’s attempts.	Must Have	High
NFR7.2	The system should not allow users to view their history attempt if not logged in.	Must Have	High

User Interface

S/N	Functional Requirement	Priority	Level
FR8.1	The system should have a basic UI to allow users to interact with the application.	Must have	High

S/N	Non-Functional Requirement	Priority	Level
NFR8.1	The UI should be optimized for desktop usage	Must Have	High

Quality Attribute Tradeoffs

Attribute	Usability	Security	Performance
Usability	▨	+	+
Security	▨	▨	+
Performance	▨	▨	▨

Priority of attributes in increasing order: Usability, Security, Performance

User Stories

Role: User

- As a user, I should be able to pick a certain difficulty level that I wish to attempt.
- As a user, I should be able to get feedback/solution after my attempt.
- As a user, I should be able to match with peers of the same difficulty level.
- As a user, I should be able to communicate with my peers while working on the question.
- As a user, I should be able to see what my peers are writing down in real time so that I can compare solutions and have meaningful discussions.
- As a user, I want to be able to login to save my preferences
- As a user, I want to be able to login to view my previous attempt
- As a user, I want to be able to change my password (if I forget it)
- As a user, I want to be able to delete my account should I no longer want to use the app

Role: Interviewee

- As an interviewee, I want to be able to practice coding questions with someone.
- As an interviewee, I want to be able to adjust the difficulty settings of the coding questions.
- As an interviewee, I want to be able to get live feedback and comments for my code.
- As an interviewee, I want a time limit to solve my code for more extreme difficulty.
- As an interviewee, I want to be able to ask for a time extension.
- As an interviewee, I want a shared whiteboard for sketching in case I need to draw diagrams or anything.

Role: Interviewer

Basic Functionality

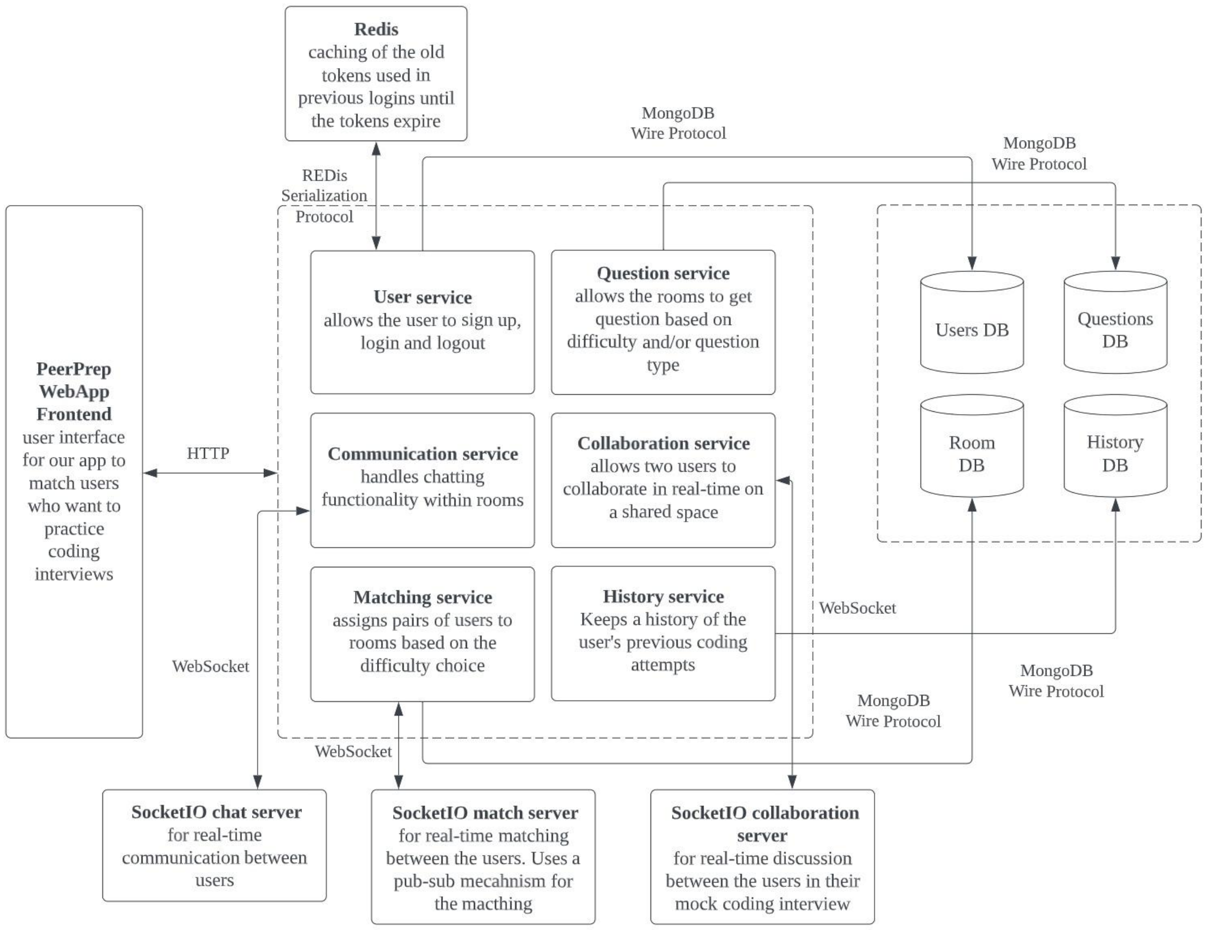
- I want to be able to practice with someone with the same difficulty preference
- I want to be able to to contact my match during the sessions (should I have any further questions)
- I want to be able to leave the room if I feel like my match isn't suitable
- I want to be able to find a match quickly (I don't want to wait a long time) and I want to be notified if there aren't any
- I want to adjust the font size when typing in my code.
- I want to be able to switch my role with my match

Nice To Haves

- I want to be able to use shortcuts when typing in my code (auto-tabs also helps :))
- I want to be able to adjust the theme of the website dark/light theme
- I want to choose the coding question types (i.e. Arrays, Graphs, etc.)
- I want to be able to add my match as a friend

Project Architecture

Component Architecture

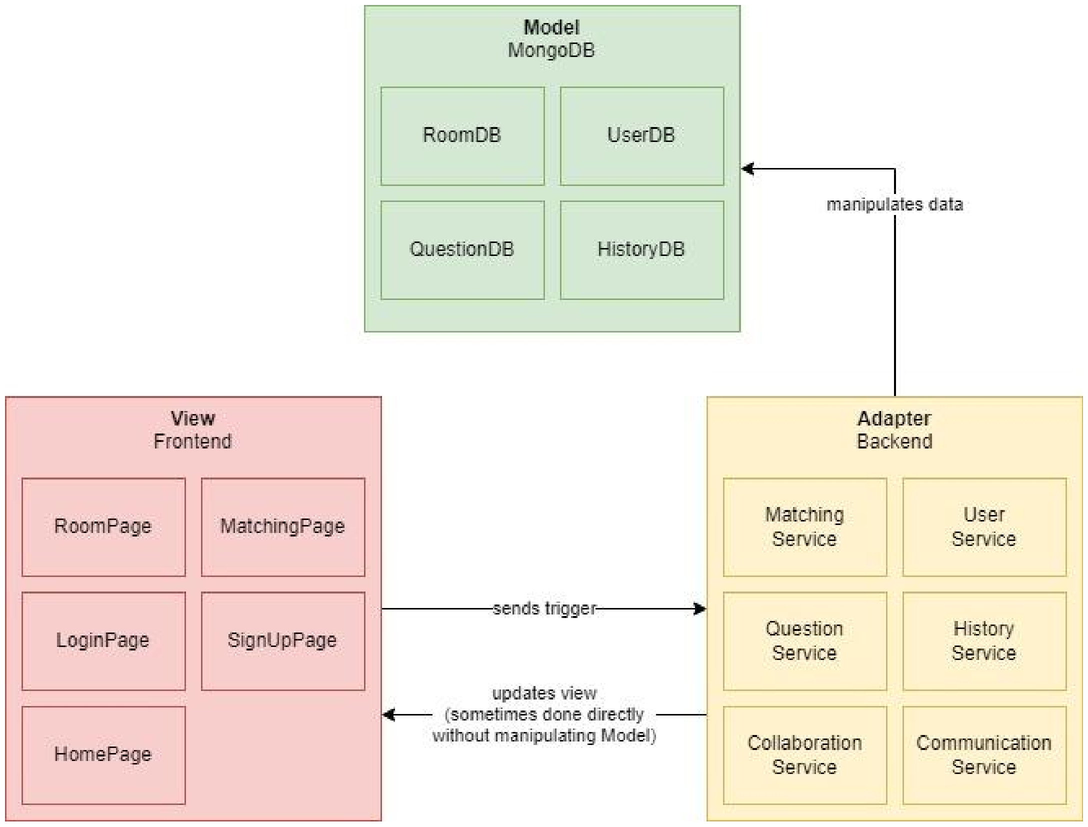

Redis
caching of the old tokens used in previous logins until the tokens expire
REDis Serialization Protocol
MongoDB Wire Protocol
MongoDB Wire Protocol
PeerPrep WebApp Frontend
user interface for our app to match users who want to practice coding interviews
HTTP
User service
allows the user to sign up, login and logout
Question service
allows the rooms to get question based on difficulty and/or question type
Communication service
handles chatting functionality within rooms
Collaboration service
allows two users to collaborate in real-time on a shared space
Matching service
assigns pairs of users to rooms based on the difficulty choice
History service
Keeps a history of the user's previous coding attempts
Users DB
Questions DB
Room DB
History DB
WebSocket
WebSocket
WebSocket
MongoDB Wire Protocol
MongoDB Wire Protocol
SocketIO chat server
for real-time communication between users
SocketIO match server
for real-time matching between the users. Uses a pub-sub mecahnism for the macthing
SocketIO collaboration server
for real-time discussion between the users in their mock coding interview

For our project, we decided to implement the ***Microservice Pattern***, which is implemented by splitting each task (i.e. user matching, code collaboration, chatting, retrieving questions, and saving user's attempts) into a service of their own (i.e. User service, Matching service, Collaboration Service, Communication service, Question service). This provides us with greater agility and easier management, since each service is modular and small, and also provides the opportunity for more horizontal scaling in the future. In addition, we think the microservice pattern fits the project due to the modularity trait of the pattern. By splitting the project into smaller and more specific services, each service is independent from each other and thus we can work on the project asynchronously without the need to wait for one another. We believe this architecture can reduce the complexity of distributing the task and increase the productivity of the team. And lastly, splitting them into microservices has the benefit of independent scaling for each service during deployment based on their traffic/demand.

The downside of this is that there may be repetition in the code (such as implementation of authentication and authorization). Furthermore, it also brings about an additional layer of complexity since there are so many services working with one another and it can be difficult to pinpoint the source of error when something goes wrong. Thus, it might be overkill for a small-scale application.

Each backend service has its own directory, each with their own `package.json`, server (`index.js` / `server.js`), and database. For the frontend to access each service, they have to make requests through different APIs that each service provides (refer to API Discovery for details on our public APIs). Each service also has its own independent CI, with their own tests and workflow.

Frontend



The ***Model-View Adapter(MVA) Pattern*** was used in the development of our user interface. There exists a component for each major functionality:

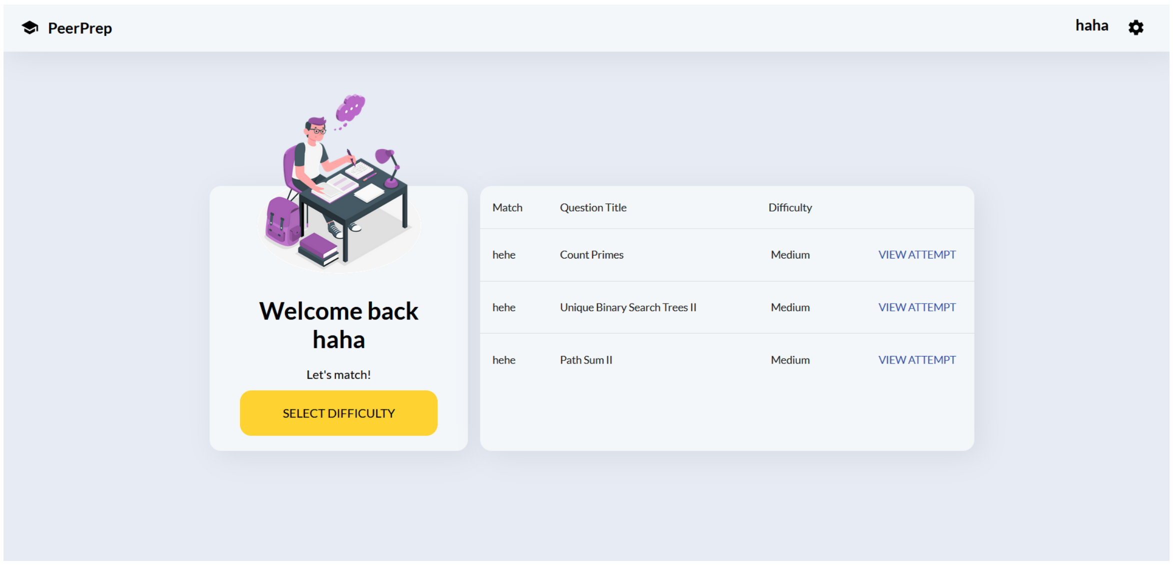
1. Model to store the data required for the app
2. View for users to see the data from the DB and interact with the DB through the Adapter
3. Adapter to manage the logic of the user interaction and modify the data from the Model.

Views

For the Frontend, we also decided to implement a View for each major functionality.

Home Page

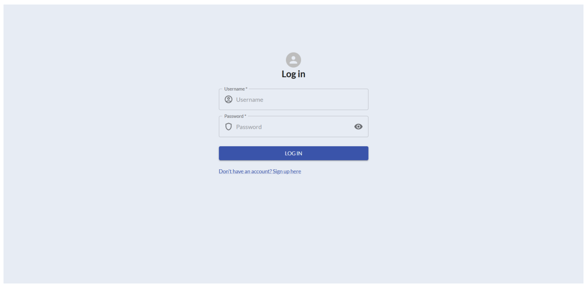
serves as a dashboard for logged-in users

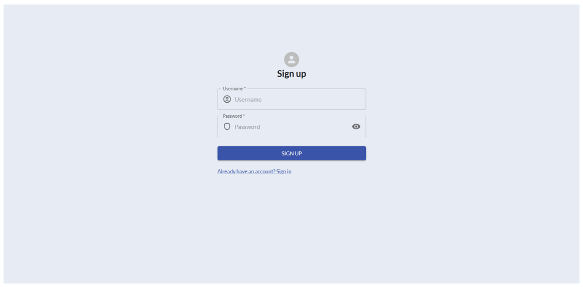


Match	Question Title	Difficulty	
hehe	Count Primes	Medium	VIEW ATTEMPT
hehe	Unique Binary Search Trees II	Medium	VIEW ATTEMPT
hehe	Path Sum II	Medium	VIEW ATTEMPT

The Sign Up and the Login Page

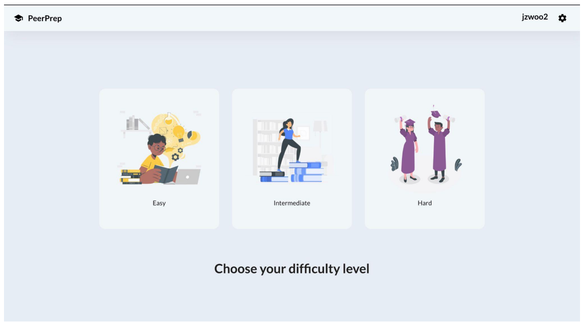
for new users to create accounts and for existing users to log in

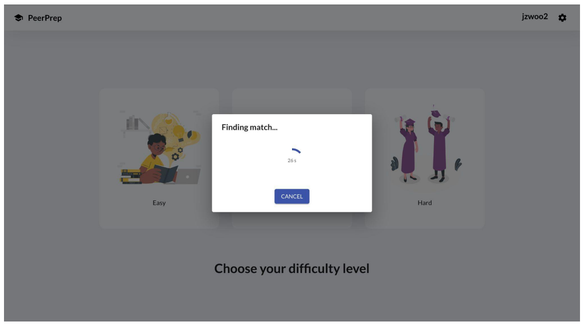




The Matching Page

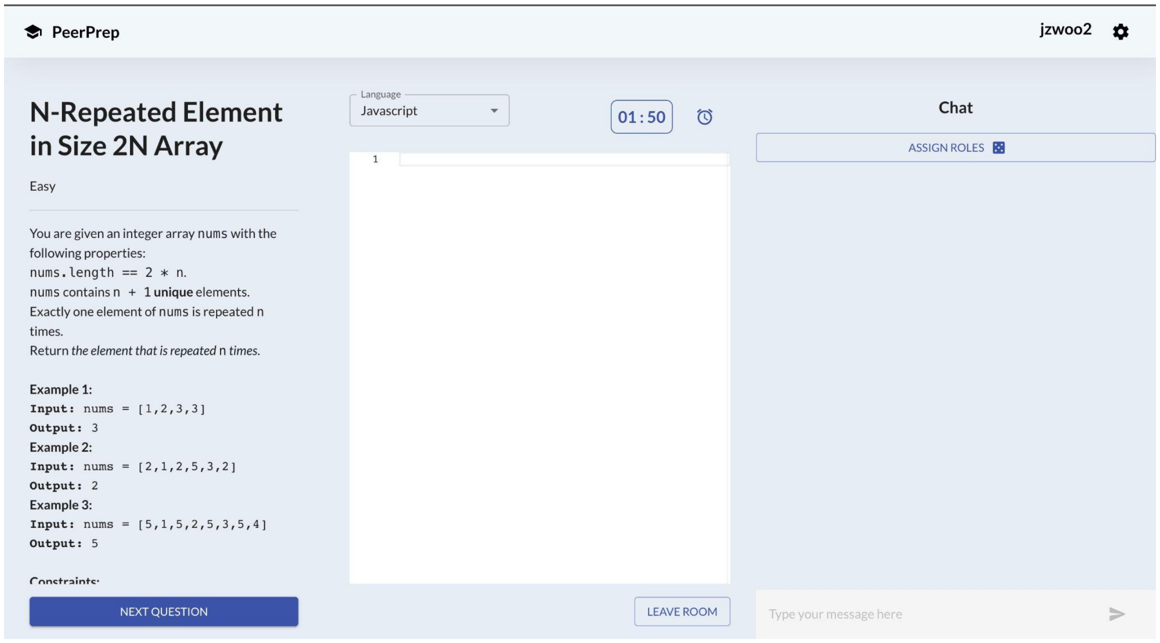
for existing users to find a random match and be joined in a room based on a certain criteria





The Room Page

which serves as the private space for matched-users to interact with



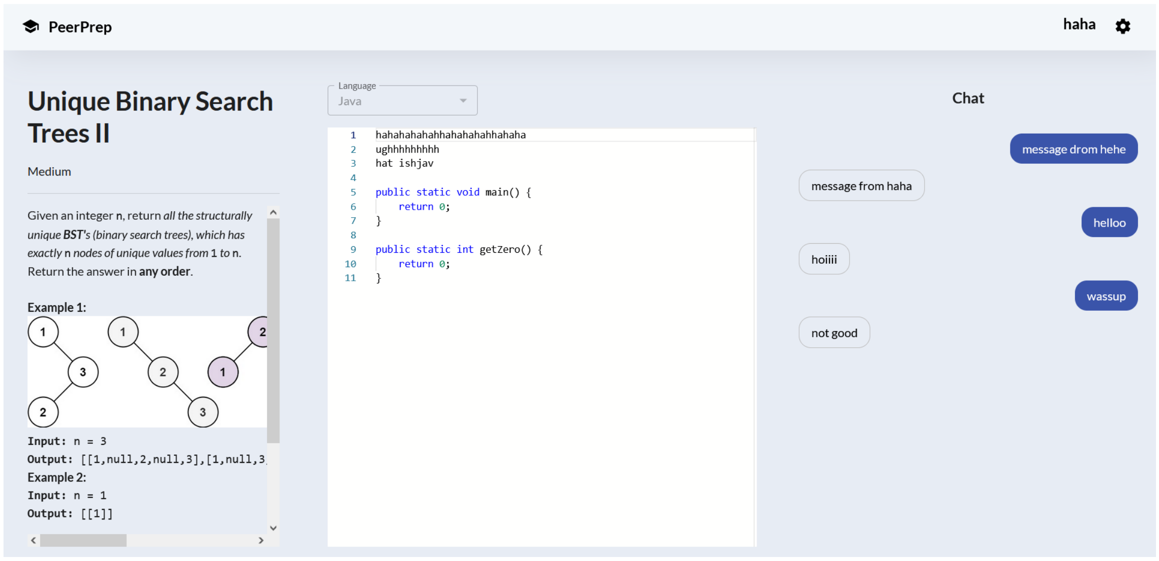
N-Repeated Element in Size 2N Array

Easy

You are given an integer array nums with the following properties:
nums.length == 2 * n.
nums contains n + 1 **unique** elements.
Exactly one element of nums is repeated n times.
Return *the element that is repeated* n *times*.

Example 1:
```
Input: nums = [1,2,3,3]
Output: 3
```
Example 2:
```
Input: nums = [2,1,2,5,3,2]
Output: 2
```
Example 3:
```
Input: nums = [5,1,5,2,5,3,5,4]
Output: 5
```
Constraints:

or as a page to view past attempt details (in which case everything is read-only)



Unique Binary Search Trees II

Medium

Given an integer n, return *all the structurally unique BST's (binary search trees), which has exactly* n *nodes of unique values from* 1 *to* n. Return the answer in **any order**.

Example 1:
```
Input: n = 3
Output: [[1,null,2,null,3],[1,null,3,
```
Example 2:
```
Input: n = 1
Output: [[1]]
```

```
hahahahahahhahahahahhahaha
ughhhhhhhhh
hat ishjav

public static void main() {
    return 0;
}

public static int getZero() {
    return 0;
}
```

message drom hehe

message from haha

helloo

hoiiii

wassup

not good

Components

Each view will also have a number of sub components based on their functionality. For example, the Room Page is separated into the CodingQuestion which is responsible for displaying the question; the CodeEditor which serves as the collaboration space for matched users; and the ChatView which is responsible for sending and displaying the private messages sent within the room.

Since there are a lot of subcomponents which might share the same resources, we also provided contexts for each functionality: the Socket Context, which stores all the sockets used in the app: the matching/room socket, the chat socket, and the collaboration socket; and the Room Context which stores all room-related data such as the chat messages, the current question displayed, the collaborated code, etc.

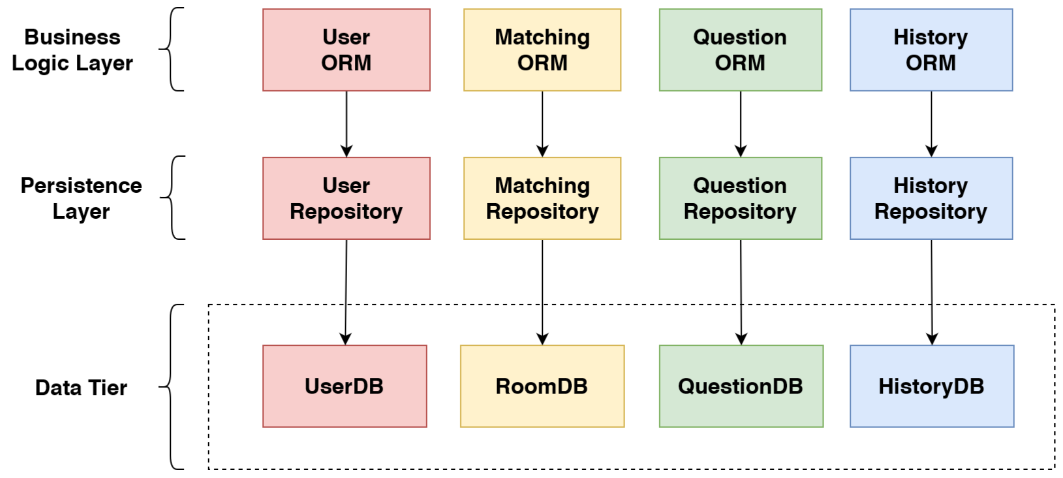
We also provided an API Request class which provides a template for every API request (i.e. GET, POST, PUT, DELETE), and for each API request to each service (e.g. Room API, History API, Authentication API, User API, etc.) we also created an API Class for them. Refer to the API Discovery section for a complete list of the public APIs.

Refer to the Appendix for the frontend directory structure.

Backend

Repository Pattern

For each backend service that has a database, we implemented the ***Repository Pattern***, which serves as an abstraction of the data layer and centralizes the handling of the domain objects. This is used in the backend of the services that utilizes the DB (i.e. User Service, Question Service, Matching Service, as well as History Service).



Pros of Repository Pattern

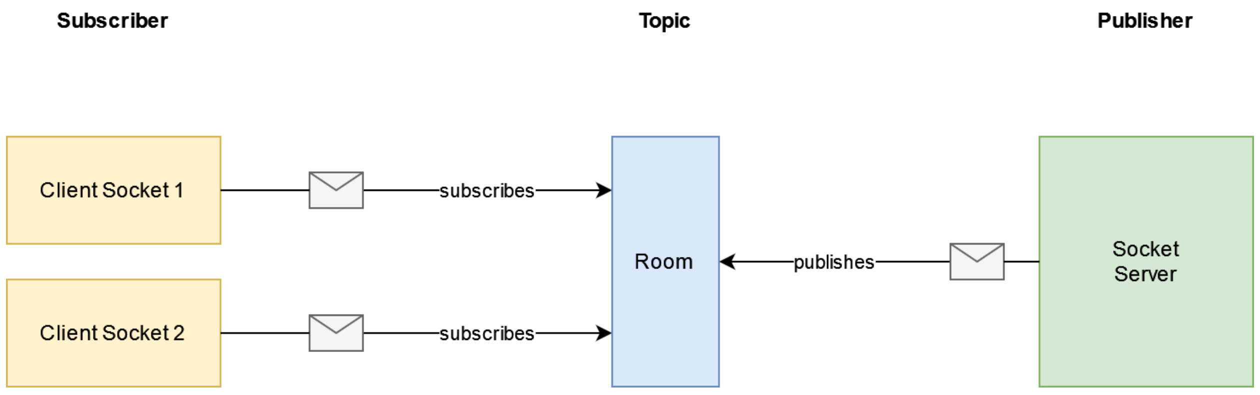
- **Separation of Concerns**: the business logic layer of the application does not need to know how data persistence works
- **Isolate** the application domain and data mapping layers → improves testability, code extensibility, and maintenance.
- **DRY (Don't Repeat Yourself)** : the code to query and fetch data from data source is not repeated.

Cons of Repository Pattern

- Add another **layer of abstraction** which increase the complexity and might seem useless in a small application

- The need to create a repository for each entity in the application → the repository will be very big as we include additional methods and complex search capabilities which gives us a repository that closely matches the database layer that we use.

PubSub Messaging System



PubSub messaging system is used for the implementation of Matching Service, Communication Service, and Collaboration Service. In Socket.io, the "topic" is equivalent to the "room" the sockets joined. The Clients (sockets) subscribed to the "topic" such that any event the socket server ("Publisher") emits to the room, only those subscribed/joined to it receives it.

Pros of PubSub

- **Asynchronous**
  PubSub is asynchronous → risk of performance degradation due to a process that takes a long time (e.g. long-running data exchange interaction) as each process doesn't need to wait for each other as opposed to synchronous process
- **Real-time communication**
  Most of our services require real-time communication (e.g. chat service, match service), thus it is ideal to use push-based PubSub, which allows instantaneous delivery of messages.

Cons of PubSub

- **Challenging to test**
  Testing PubSub architecture is challenging as interactions are asynchronous. In the usual testing scheme, we only need to make a request and analyze the result. However, in PubSub, we need to send a message to the system and the test needs to observe the behavior of the process to see when and how it handles the message. Thus, the testing can be difficult to manage. For asynchronous testing, we are currently implementing it by introducing a delay between each test case.

Developer Documentation

Tech Stack

- Frontend : ReactTS (Material UI)
- Backend : Node.js, Express.js, JWT, Redis, Selenium, Socket.io
- Database : MongoDB
- Project Management: GitHub Issues

Design Decisions

Why TypeScript?

We chose Typescript over Javascript because it supports static typing and interfaces.

- Static typing allows static detection of programming errors quickly and automatically.
  It helps us early on from the development process by potentially reducing the number of bugs and the amount of time spent on debugging.

- Interfaces define a contract for an entity to conform to.
  It helps us create an object with certain properties, validate the structure of an object, and give us the ability to identify and fix errors earlier. Also it helps us from repeating ourselves: instead of typing the same object structure over and over again, we use interfaces instead.

Why Material UI?

Material UI (MUI) is based on Google's Material Design, a unified system that combines theory, resources, and tools for crafting digital experiences.[1] We chose Material UI (MUI) components for React because MUI prioritizes accessibility and efficiency, while also allowing us independent styling.[2] It is also widely used by large companies such as Spotify, Amazon, Netflix, and NASA. By using MUI, we also have the advantage of developing a consistent look and feel of the UI across the components.

Why MongoDB?

- NoSQL Database vs SQL Database
  We decided to use a NoSQL database because we realized that the data are independent of each other. Moreover, NoSQL databases provide flexibility in modifying fields in a collection without having to modify fields in other collections.

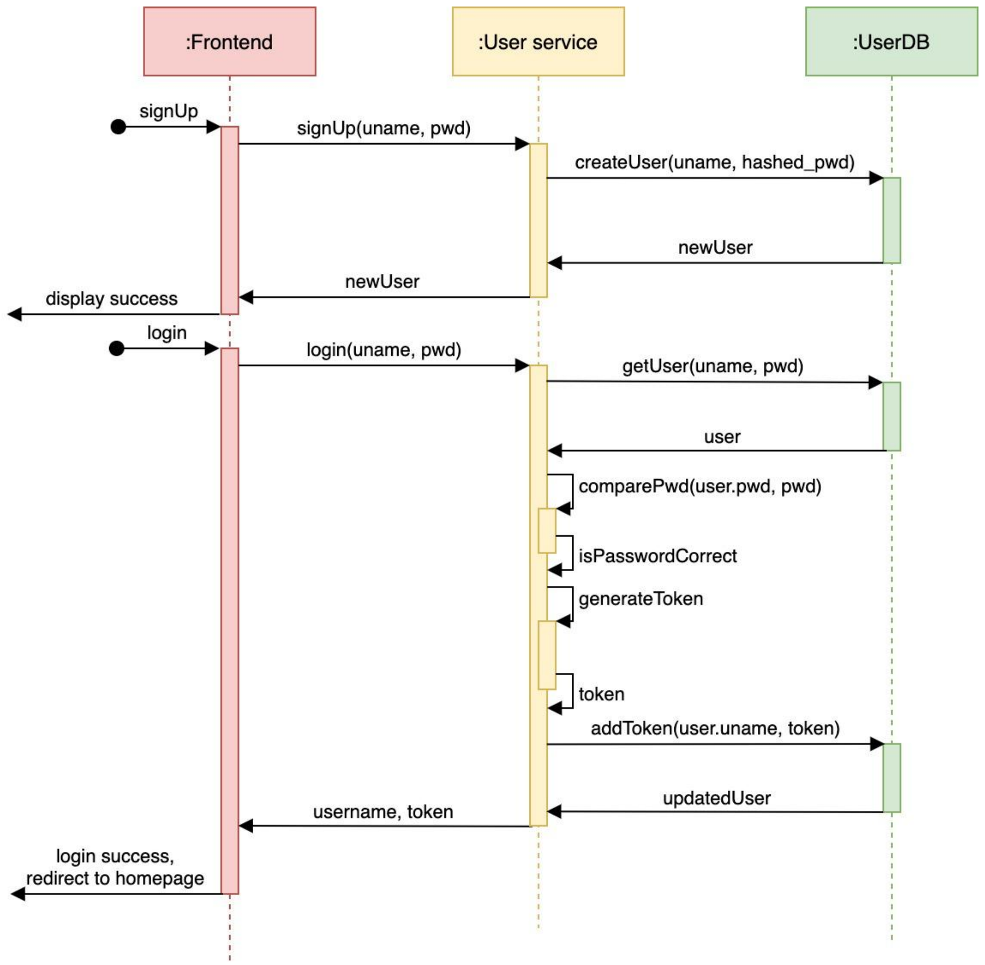
- MongoDB Atlas vs Self-hosted
  MongoDB Atlas provides strong authentication and encryption services. MongoDB Atlas also provides the ability to scale up / down based on demand, which would be a useful feature if we decided to deploy our application. (Related QAs: ***Security*** and ***Performance***).

[1] https://m3.material.io/

[2] https://mui.com/

User Service

Sequence Diagram of Sign up and Login



User service manages all user-related operations such as sign up, login, change username, change password, logout, and delete user. Users can sign up to create an account by providing a unique username and a password. The password will be salted and hashed before stored into the User DB.

After signing up, users will be able to login to their accounts if they provide the correct credentials. The user provides their username and password and the service checks if the password matches up with the hashed password in the database for the corresponding username. If the password matches, then a JWT token will be generated and passed to be added to the user record in the UserDB. At the same time, the same token will also be returned to the user in the frontend for the purpose of authentication in future API calls.

Whenever users log out of their account, we store the blacklisted tokens in a Redis server and set the TTL according to the expiry date of the tokens. After that, each time the user logs in, a check will be made to ensure that the token is not one of the blacklisted tokens that have been recently used.

Design Decisions

Why did we use salting and hashing for password management?

We decided to implement salting and hashing before storing user's passwords in our database instead of just storing it in plaintext. In addition to maintaining the privacy of our users, hashing and salting adds a layer of security.

Hashing is an algorithm used to transform plaintext passwords into digests. It makes data storage more secure as it is difficult to reverse-engineer a hashed password/digest to obtain the original password. However, since different users might pick the same passwords, there is a possibility of having multiple similar digests in the database. Therefore, salting is used to add random characters before or after the passwords prior to hashing. Thus, even if the attacker manages to reverse-engineer one of the hashed passwords in the database, it would be difficult for an attacker to retrieve the rest of the passwords in the database since he/she would have to recompute every hashed password.

Why JWT for authentication?

We decided to use JSON Web Token (JWT) as our form of API authentication, because it is a secure way to exchange information online. Moreover, it is an authentication process that does not have to rely on a database to save a session's information. Its token-based stateless mechanism is well suited for our microservice architecture where API calls are made constantly.

Why did we use Redis to keep track of blacklisted tokens?

There are several reasons why we choose Redis over storing it into MongoDB:

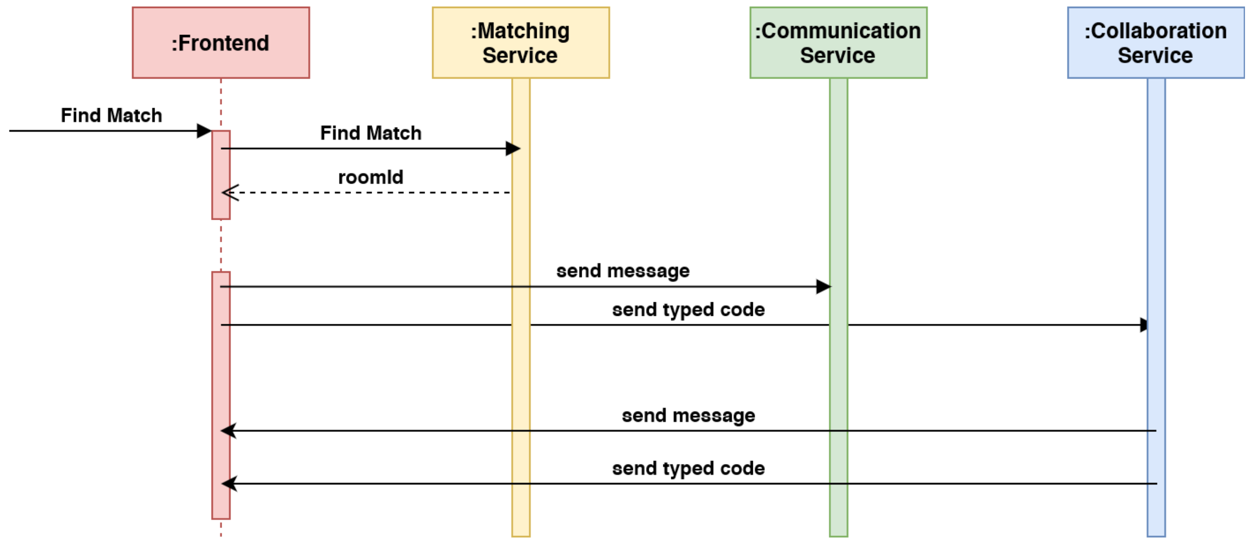
1. It is an in-memory (uses RAM) database that provides faster database response.
   This is in contrast with MongoDB where data is stored in a server/cluster in the cloud. Redis provides a much faster response due to the usage of an in-memory database, which allows faster reads and writes. (Related QAs: ***Performance***)

2. It is based on a key-value model, which allows for extremely efficient access time.
   Two of these characteristics make it possible to check efficiently whether the token is blacklisted everytime user is trying to login. Consequently, this provides a faster login and logout when the blacklisted tokens are retrieved and compared, giving the users a better user experience.

3. It provides control over the lifetime of a data (TTL).
   Since we set the TTL according to its expiry date, we don't have to remove it from the database via an API call or manual removal. We removed the token from Redis after its expiry time because it is redundant to store an expired token as it would never pass verification. (Related QAs: ***Security***)

Matching and Room Services

Sequence Diagram of Finding Match and Joining a Room



Finding Match

When a user requests to be matched with another user, the client socket first establishes a connection to the matching server. When the connection is established, the client socket emits a ‘find-match’ event to the server, indicating the client’s intention for a match. A timer of 30s will also start on the client side.

The matching server keeps track of a ‘waiting room’ of clients that are finding a match. When it receives the ‘find-match’ event, it looks for another client in the ‘waiting room’. If the clients’ criteria matches, a ‘room’ is created (a room entry is created in the RoomDB) and a unique `roomId` will be emitted to the users. On the client side, the users will then be redirected to the room page. The `roomId` will also be used for future room-related requests (e.g. saving attempt, sending private messages, collaborating, etc.).

However if the server is unable to find a suitable match, the requesting client will then be placed in the ‘waiting room’ until the timer on the client side runs out, which will then disconnect the client socket with the server.

Joining a room

Every room has a unique `roomId` and can be accessed through the URL suffix ‘/room/`<roomId>`’. Upon matching, users should be automatically redirected to the room page in the browser. Only the users who are matched to the room can access the URL. This is done by a creation of a specific JWT token for the room itself, when a user attempts to access the room, the user is first authenticated using their own JWT tokens, and then authorized by the room when it checks the user’s identity with its own JWT token.

In the frontend, the room contains a question panel, a code editor and a chat. When the users are redirected to the room, a collaboration and communication socket are automatically connected to the collaboration and communication server respectively. They will also be subscribed to any events that occur for that particular room in the collaboration and communication channel. This means that the users are able to see real time changes when they make any changes in the code editor or send a message through the chat. There will be a more detailed explanation about how the chat and collaboration services work in the following sections.

Design Decisions

Why did we separate matching and room services?

We decided to separate the matching service and each service in the room (i.e. communication, collaboration, question) because we decided to implement a Microservices design pattern. It also helps during deployment that each service is able to scale independently from each other according to their usage.

Why did we use socket.io for matching and room services?

We chose socket.io to implement our matching-service, communication-service, and collaboration-service because the way each works (i.e. which is triggered via a specific set of events) is suitable with how socket.io is designed to use. Socket.io is also low-latency, which is suitable since we want a low delay during communication and collaboration since we want it to be (as close to) real-time (refer to our Communication Service and Collaboration Service NFRs for more detail).

Socket.io also provides a Pub-Sub type of messaging system via the use of rooms, which instead of broadcasting the whole message for all to see (and the end-users would need to implement a filtering-system), it allows the end-users to subscribe to a room, and only see the events transmitted within said room, without it ever affecting users subscribed to a different room. This is suitable with our app design, which allows users to collaborate and communicate with each other within the confined space of the “Coding Room”.

Why did we store the Room object in a database?

We decided to implement the Room DB in order to make use of MongoDB’s unique ObjectID to use as the room’s ID, as well as to implement syncing of the questions (such that for users within a room, the questions displayed are the same).

We do not update the chats or the code typed (within the code editor) in the DB because we realize that communication and collaboration needs to be real-time (i.e. small delay) and updating the model in MongoDB everytime there’s a new chat or a change in the code would incur a significant amount of delay. Thus for communication and collaboration, only sockets are used and state saving only occurs when one of the users.

Why did we implement authorization for the room service?

When two users are matched, a room entry is created in the Room DB, with a roomId. The users are then redirected to the room page, which is unique to each pairing, identified by the `roomId` suffix in the URL. This URL should only be accessible to the two matched users, meaning that even if other users obtained the roomId, they should not be allowed to access the room. This is mainly due to privacy reasons and also because it can lead to disruption if the third party is malicious.

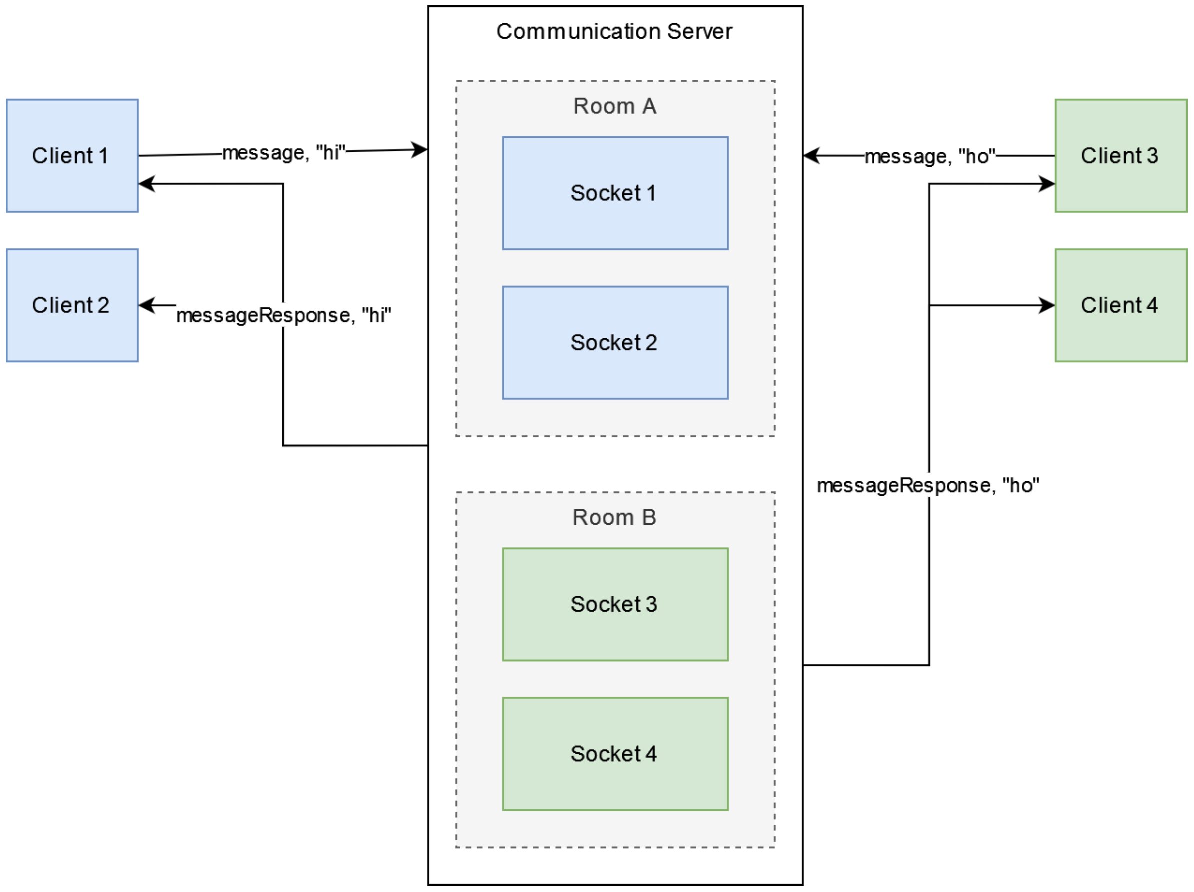
Event List

Event	Argument	Event Response	Description
find-match	{ user: { user_id: string, username: string, } difficulty: int }	No response in the event no suitable match found ——— join-room in the event a match is found	Notifies the matching server to look for a match, if there is no suitable match found, the user is put in the server's waiting room.
join-room	{ room: string }	joined-room	Triggers the socket to join the room specified in the argument, and an internal timer to be created within the server.
cancel-req	{ user: { user_id: string, username: string } }		Removes the user from the waiting room.
get-question	{ room: string }	question	Retrieves a random question based on the room criteria from the DB, and updates the room DB's question field.
		timer	Emitted by the server at regular intervals (every second) to the room specified. Allows clients within a room to have a synced timer. Triggered upon room joining.
extend-time	{ room: string, seconds: string }		Extends the timer for the specified room by the specified number of seconds.
delete-room	{ room: string }	match-left	Removes the room from the Room DB, removes the timer for the room within the server, and notifies the clients within the room.

Communication Service

The Communication Service is in charge of handling private messages sent between users within the same room, it handles multiple events such as sending messages, assigning roles, as well as sending typing notifications. It accomplishes so by using the PubSub messaging system to emit private events between users in the same room. Users who are not in the room won't receive the events (i.e. all the events are sent within the confined space of the room).

Messages are sent to all users within the room (instead of just to the recipient) and only displayed in the frontend once the message is received (the "message" event).



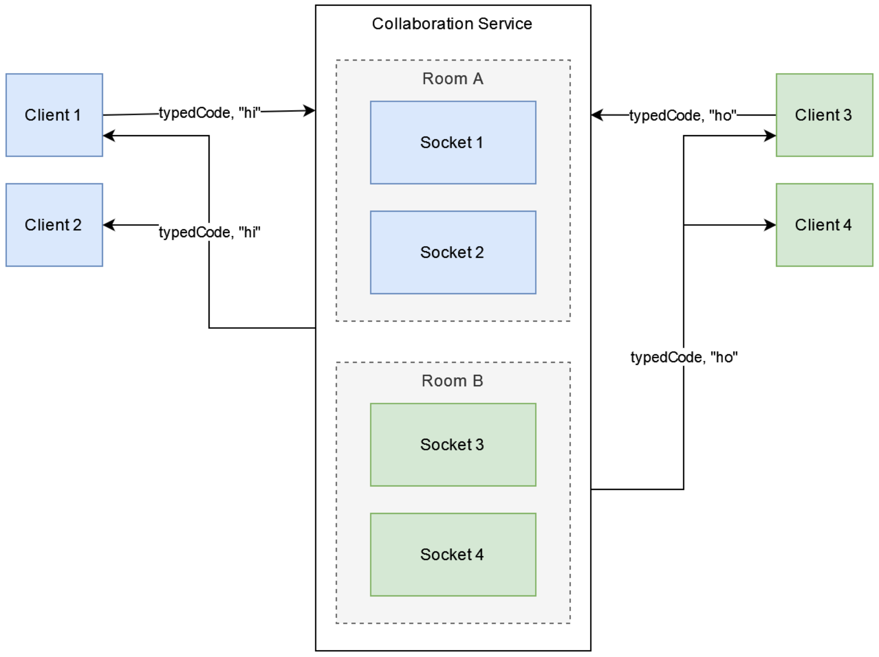
For Communication Service's design decisions, refer to Matching Service's section on socket.io.

Event List

Event	Argument	Event Response	Description
join-room	{ room: string }	joined-room	Triggers the socket to join the room specified in the argument
message	{ text: string, socketId: string, name: string, room: string, }	message	Sends the message to clients within the room specified in the argument.
typing	{ socketId: string, name: string, room: string }	typing	Notifies the users within the specified room that the user (with the name specified) is typing.
stop-typing	{ socketId: string, name: string, room: string }	stop-typing	Notifies the users within the specified room that the user (with the name specified) has stopped typing.
delete-room	{ room: string }		Deletes any data related to the room within the socket server (e.g. role assignments)
get-role	{ username: string, room: string }		Assigns a random role to the user within the room. It is triggered when the client socket is connected to the socket server.
get-roles	{ room: string }	assign-role	Sends the role assignment to the client sockets within the specified room.

Collaboration Service

Similar to Communication Service, the Collaboration Service also utilizes the PubSub Messaging System via Socket.io. In this case, the code typed within the code editor, and the language the user selects are the events the socket transmits. When one of those changes (i.e. when the user selects a different language, or they type something within the code editor), it will trigger an event where the changes are transmitted to the other user within the same room.



For Collaboration Service's design decisions, refer to Matching Service's section on socket.io.

Event List

Event	Argument	Event Response	Description
join-room	{ room: string }	joined-room	Triggers the socket to join the room specified in the argument
typedCode	{ text: string, socketId: string, name: string, room: string, }	typedCode	Sends the typedCode to clients within the room specified in the argument.
set-language	{ language: string, room: string, }	set-language	Sets the programming language within the room specified in the argument, triggered when a user within the room changes the programming language.

Question Service

Question service manages the questions that are shown to each room. We scraped around 800 questions from LeetCode API using Selenium and stored them in our Question DB. Our Question DB is separated into two different collections with different schemas: QuestionModel and QuestionTypeModel. QuestionModel stores all the questions while QuestionTypeModel stores the question difficulty, type, and question ID which links the collection to QuestionModel.

QuestionTypeModel was created because we planned to implement a feature that allows users to choose the question based on difficulty and/or type. With the existence of this collection, we can easily choose a random question based on the criterias given.

Currently, the only API that is exposed to the frontend side is the GET random question, which accepts a difficulty level in its body. We didn't create any POST, PUT, or DELETE APIs because users are not supposed to modify the questions DB in any way. We also didn't create an API to GET a question using its ID or name because the question is randomly assigned to a room.

Everytime a room is created, it will invoke an API call to GET a random question, which will be shown in the room page. When the user clicks the "Next Question" button, it will call the API endpoint once again to get a new question with the same level of difficulty.

Design Decisions

How did we implement the QuestionTypeModel and why?

The scraper works by looping all questions in the API. For each question, we insert it to our QuestionDB and also scrape all the types related to that particular question. This information will be then used to insert the question to QuestionTypeDB.

We chose to separate QuestionTypeDB into a different collection rather than creating new fields in the QuestionModel in a form of array that stores all the types to increase efficiency in querying a random question. Had we chosen to use an array to store all the question types, getting a random question with only the difficulty level wouldn't be affected. However, querying a random question given the difficulty level AND question type (e.g. Array, Math, Dynamic Programming) would be a major problem because we have to go through all questions in our QuestionsDB, fetch the *type* field that contains all the question types related to it, which might not even contain the type that the user desire. This method is very inefficient because the worst case would be going through all 800 questions in the DB just to fetch one.

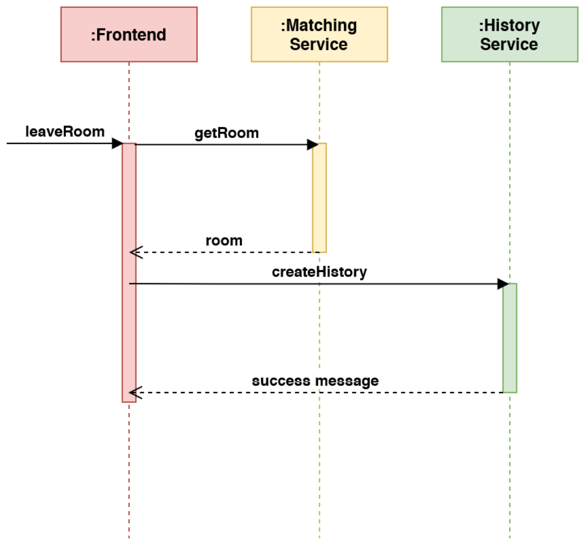
Separating the QuestionTypeModel eliminates this issue, because given the difficulty level and question type, we can simply find all question IDs that match the difficulty level and question type, pick random IDs from the result, and fetch the question with matching ID from QuestionDB.

Why no authorization is used for the question service?

We chose not to implement authorization since the question service is only providing an API to retrieve a question from the database. Hence, users will not be able to do any harm to the question database because users will not be able to modify/add questions to the question database since the Question DB is prepopulated.

History Service

Sequence Diagram of Saving Attempt History



The History service manages histories of all user attempts. Every time one of the users leaves a room, a history entry will be created and stored in the History DB. The history entry has a field for both matched users' username as well as their user ID (obtained from the User's ObjectID in MongoDB) in the same room. Users possessing the user ID specified in the history entry will be associated with said entry. Since the frontend doesn't store any related room info (such as the matched user, the matching criteria, etc.), every history creation will invoke getting the room using the roomId in order to get the matching user's username.

Design Decisions

Why did we separate User DB and History DB?

We separated the user Info and the user's attempt histories into two different databases to avoid unnecessary updates to User DB. If we combine them into one DB, we would be required to update it every time a user joins a room. In addition, suppose we decided to combine the User DB and the History DB, each matched user would have the same record in their history attempt. Splitting them into separate databases has the advantage of avoiding duplicate records, finding out a user's history attempts would only require querying for their username in our History DB.

Why store the User ID and the Username in the History DB?

It might seem redundant at first to store both user-related info in the History DB. However, we have valid reasons for doing so:

1. The Home page dashboard

   Currently, our homepage serves as a dashboard which displays the logged-in user's previous attempts, this includes their match's username. Just opting to store the user ID would require multiple API calls from the frontend to fetch the match's username from the User DB, which may be a bit of an overkill since it is such a small functionality and yet it serves an important function for the UI/UX.

2. Changing usernames

   We might also opt to just store the username in the History DB. However this presents another problem as currently the app allows users to change their usernames, and when they do so, they might no longer be linked to their old history entries as the username (which serves as the "foreign key") has changed.

   It can also be proposed that we update the history entries related to the user in the event of a username change, however this incurs a significantly large cost as we would have to iterate through all available history entries (which may or may not be related to the user) and update the ones related to the user (which incurs a large amount of read/write operations).

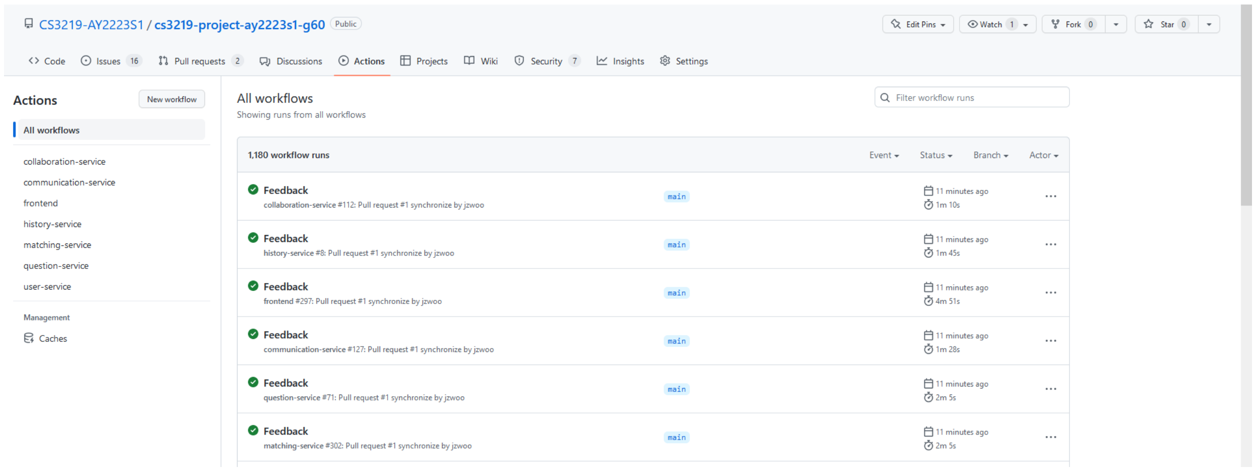
Development Process

Project Methodology

We adopt Scrum methodology to facilitate the project. We implement this methodology by having weekly standups to update each other's progress and impediments. We also have sprint planning before the start of each new sprint to specify the scope of the work that needs to be done.

Continuous Integration

We have also implemented Continuous Integration for our project. The main function of the CI is running *npm ci* which checks for consistency of the modules and dependencies; and *npm run test* which runs unit tests for each service. It is currently running every pull request on GitHub Actions.



All workflows
Showing runs from all workflows

1,180 workflow runs

Feedback collaboration-service #112: Pull request #1 synchronize by jzwoo	main	11 minutes ago 1m 10s	...
Feedback history-service #8: Pull request #1 synchronize by jzwoo	main	11 minutes ago 1m 45s	...
Feedback frontend #297: Pull request #1 synchronize by jzwoo	main	11 minutes ago 4m 51s	...
Feedback communication-service #127: Pull request #1 synchronize by jzwoo	main	11 minutes ago 1m 28s	...
Feedback question-service #71: Pull request #1 synchronize by jzwoo	main	11 minutes ago 2m 5s	...
Feedback matching-service #302: Pull request #1 synchronize by jzwoo	main	11 minutes ago 2m 5s	...

Testing

As mentioned in Continuous Integration, we have employed automated testing for each of the services listed. Tests are done using mocha and chai, for services utilizing the socket.io, we ensured that all the custom events are tested; we also included connection tests for services that connect with MongoDB using a collection specifically created for testing.

Tests are run every Pull Request within the GitHub repo and PRs are only merged when all tests for each service pass.

Improvements and Enhancements

The user's token is not updated after being generated until the user logs in again because we haven't implemented a **refresh token** mechanism, which allows users to stay logged in even after the expiry date (because the token is refreshed automatically in the background).

The user's token is stored in the database, however since the token is unique to the user and hard to be obtained, we actually do not need to store it in the database, we can just decrypt the token and get the username from the token. Moreover, it would be good to implement a more robust and extensible authentication/authorization mechanism (e.g. using RBAC for authorization).

A possible improvement to the project is to implement the matching service using **kubernetes cluster**, because kubernetes provides the ability to automatically scale up the service if the usage increases. This is very possible since matching service is probably one of the services that are more frequently used.

In terms of functionality, it would be great if we can implement an "Add match as a friend" in the future. Thus, users can practice not only with a random match but also with a "friend" that they've matched with before.

Reflection and Learning Points

We felt that there were alot of missed opportunities in the process of our project development since all of us began working on the project first and before working on the individual OTOT tasks. We should have done our OTOT tasks first before attempting to do this project because we felt like there were a lot of useful (and important) implementations that could be applied to our project. Additionally, it would have saved us a lot of research time when we were figuring out how things were supposed to work (even though some of the tech stacks used were different).

We also encountered a difficulty towards the end of this project, as we decided to implement a microservice pattern, we had to run all services in parallel, which takes up quite a lot of RAM in our PCs, this makes developing and debugging quite hard as the PC slows down.

However this project has taught us a lot of things, from project management to software architectures and design patterns. We usually didn't give much thought into these things when doing other (non-software-engineering) school projects, as long as things worked the way they should we were satisfied with it. However, actually implementing the design patterns makes our code more organized and makes for easier debugging and developing. In the future we will decide to learn more about various design patterns and where best to apply them.

This project took quite a lot of time to develop and even though we are familiar with some of the tech stacks used, it still took us quite a lot of time. It also didn't help that all members within the group also took another project-based software engineering module (CS3203). We really exercised our time-management skills this semester and we learned that when doing project-based modules, we also shouldn't take another heavy project-based module at the same time.

Individual Contributions

Member	Technical Contribution	Non-Technical Contribution
Christian Drake Martin	**Frontend**: Sign-In functionality, sign-Up functionality **Backend**: User service, user authorization, question service (scraper), history service	- Script running - Developer documentation
Erin May Gunawan	**Frontend**: Room page, home page (attempt history), matching service functionality, socket context, room context **Backend**: Matching service, communication service, collaboration service **Testing**: Matching service, communication service, collaboration service, history service	- Setup CI - Code formatting - Script running - Update Github Issues - Developer documentation
Florencia Martina	**Frontend**: Room frontend (display question), history service (frontend), room context **Backend**: User service, user authorization, room service authorization, question service, **Testing**: Question service, history service, user service	- Setup CI - Update GitHub Issues - Developer documentation
Woo Jian Zhe	**Frontend**: UI, sign up page, login page, home page, API request **Backend**: Matching service, User service	- Developer documentation

Appendix

API Discovery

User service

	API	Request body & header	Response

POST	/api/user/
	signup
	
	Allows creating new users with corresponding username and password. Return error if username exists in the userModel
		body : {
		  username : string,
		  password : string
		}
			Success :
			200
			--- 	
			Fail :
			500 - Username already exist

POST	/api/user/
	loginWithUsername
	
	Allows the user to login using username and password
		body : {
		  username : string,
		  password : string
		}
			Success :
			200 : token
			--- 	
			Fail :
			404 - User not found
			400 - Wrong username/password

POST	/api/user/
	loginWithToken
	
	Allows the user to automatically login without using username and password.
		header : JWT Bearer Token
		body : {
		  username : string
		}
			Success :
			200
			--- 	
			Fail :
			404 - Username not found
			401 - Unauthorized

POST	/api/user/
	logout
	
	Allows the user to logout. The current token will be blacklisted until it expires.
		header : JWT Bearer Token
		body: {
		  username : string
		}
			Success :
			200
			--- 	
			Fail :
			404 - Username not found
			401 - Unauthorized token

Room service

	API	Request body	Response

GET	/api/room	header : JWT Bearer Token
		body : {
		roomId : string,
		}
			{
			roomId : string,
			user1: {
			user_id: string,
			username: string,
			}
			user2: {
			user_id: string,
			username: string,
			}
			difficulty: string,
			question: QuestionObj
			}

PUT	/api/room	header : JWT Bearer Token
		body : {
		roomId : string,
		question: Object
		}
			Updated room: {
			roomId : string,
			user1: {
			user_id: string,
			username: string,
			}
			user2: {
			user_id: string,
			username: string,
			}
			difficulty: string,
			question: QuestionObj
			}

History service

	API	Request body	Response

GET	/api/history/
	historyList/:user_id
		header : JWT Bearer Token
		Params: {
		user_id: string
		}
			Status:
			201: Success
			400: Missing parameter
			401: Unauthorized
			500: DB error
			
			data: {
			histories: History[];
			}
			
			History: {
			user1: {
			user_id: string,

			username: string,
			}
			user2: {
			user_id: string,
			username: string,
			}
			roomId: string;
			question: QuestionObj;
			chats: ChatObj[];
			code: {
			language: string,
			codeStr: string
			}
			}

GET	/api/history
	/:user_id/:roomId
		header : JWT Bearer Token
		Params: {
		user_id: string;
		roomId: string;
		}
			Status:
			201: Success
			400: Missing parameter
			401: Unauthorized
			500: DB error
			
			data: {
			history: History
			}
			
			History : {
			user1: {
			user_id: string,
			username: string,
			}
			user2: {
			user_id: string,
			username: string,
			}
			roomId: string;
			question: QuestionObj;
			}

POST	/api/history
	/:user_id
		header : JWT Bearer Token
		history: {
		user1: {
		user_id: string,
		username: string,
		}
		user2: {
		user_id: string,
			Status:
			201: Success
			400: Missing parameter
			401: Unauthorized
			500: DB error

		username: string,
		}
		roomId: string;
		question: QuestionObj;
		chats: ChatObj[];
		code: {
		language: string,
		codeStr: string}
		}
		}
		
		ChatObj : {
		name: string;
		text: string;
		}	

Question service

	API	Request body	Response

GET	/api/questions
	/getRandom
		body : {
		difficulty : int,
		type : string,
		}
			Success:
			{
			res_code : 200,
			question: QuestionObj
			}
			
			QuestionObj: {
			title : string,
			question : string
			difficulty : int
			type : string[],
			examples : example[],
			constraints : string[],
			solution_link : string[],
			}
			
			Fail :
			401 - Unauthorized

Frontend Directory Structure

A truncated structure of the project's `frontend/src`

```
├── App.tsx
├── components
│   ├── HistoryTable.tsx
│   ├── Navbar.tsx
│   ├── modal
│   │   ├── ChangePasswordDialog.tsx
│   │   ├── ChangeUsernameDialog.tsx
│   │   ├── ConfirmationDialog.tsx
│   │   ├── LoadingModal.tsx
│   │   ├── MatchLeftDialog.tsx
│   │   └── TimerModal.tsx
│   └── room
│       ├── CodeEditor.tsx
│       ├── CodingQuestion.tsx
│       ├── HistoryModel.d.ts
│       ├── QuestionModel.d.ts
│       └── chat
│           ├── ChatBox.tsx
│           ├── ChatBubble.tsx
│           ├── ChatInput.tsx
│           ├── ChatModel.d.ts
│           └── ChatView.tsx
├── context
│   ├── RoomContext.tsx
│   ├── SnackbarContext.tsx
│   ├── SocketContext.tsx
│   ├── Sockets.ts
│   └── UserContext.tsx
├── pages
│   ├── HomePage.tsx
│   ├── LoginPage.tsx
│   ├── MatchingPage.tsx
│   ├── RoomPage.tsx
│   └── SignupPage.tsx
└── utils
    ├── api-history.ts
    ├── api-request.ts
    ├── api-room.ts
    └── auth-client.ts
```

Which section should be more emphasized?

- ~~Having to prioritized the NFR (high, medium, low shd be enough)~~
- ~~Milestone 2 : Check if we have **tech stack** listed somewhere~~
- ~~Cross-referencing, labelling, linking parts of the document is expected throughout the report. (link different parts together?? Hyperlinks and etc)~~
- ~~Testing can be a plus point~~
- NFR → explain abt the tradeoff (ranking → e.g. why prioritize one NFR over the other?)
- Development process (Agile, iteration, sprints, CI/CD) can screenshot and put in, state that we do weekly meetings, etc
- ~~Include some screenshots for UI~~
- Listed out all of the test cases (backend)
- ~~Future enhancement~~

What are we missing?

- Milestone 2 is ok ?
- More NFRs
- ~~Design patterns, design decisions~~
- Product Requirement
- ~~Not required to include User Stories~~
- I~~nclude database schema and justification why we use the schema~~
    - ~~Why do we need RoomDB?~~
    - ~~Why we separate User and History DB~~
-