CS3219 Project
AY2223 Semester 1

PeerPrep
Project Report

By Group 46

Group Member	Student No.
Hazel Hedmine Tan	A0204937B
Matthew Gani	A0204882A
Ng Han Poh Jacob	A0210640Y
Sherman Ho	A0201413A

Table of Contents

Introduction **3**
- Purpose 3
- Individual Contributions 3

Functional Requirements **4**
- User Service 4
- Matching Service 5
- Collaboration Service 5
- Chat Service 6
- Question Service 6
- History Service 6

Non-Functional Requirements **7**
- Security 7
- Performance 7
- Scalability 7
- Availability 8

Developer Documentation **8**
- Overall Application Architecture 8
- Design Patterns 9
  - MVC design pattern 9
  - Pub Sub Pattern 9
  - Observer pattern 10
- Design Decisions 10
- Tech stack 12
- Features 13
  - Frontend 13
  - User Service 14
    - Authentication 14
    - Authorization 15
  - Room Management Service 15
    - Matching Service 16
    - Collaboration Service 17
    - Chat Service 17
  - Question Service 18
  - History Service 20
- Deployment 21
- Development Process 22

Future Enhancements **24**

Reflection and Learning points **25**

Conclusion **25**

Introduction

Purpose

An increasingly popular style of technical interviews are whiteboard interviews where candidates have to write out their code to a technical question while explaining their thought process. It is a process that requires both technical and communication skills which requires practice to get good at. Having a platform that can help candidates hone their skills would aid them in their interviewing process.

PeerPrep is a web application that lets users prepare for technical interviews by conducting mock interviews with other users. The platform will match 2 users based on the difficulty level and topic chosen. Once matched, users can roleplay as the interviewer or interviewee by coding on a real-time collaborative editor and communicate via a chat box provided. Users can also view the details of their past matches to allow them to revise.

Individual Contributions

Group Member	**Technical Contributions**	**Non-Technical Contributions**
Sherman	Backend: ● Matching service ○ Collaboration service ○ Chat service	Report: ● Design Patterns ○ Pub Sub pattern ● Features ○ Room Management Service ● Design Decisions ○ Choice in cache technology for session info ● Future Enhancements ○ Improvements to room management service
Jacob	Frontend: ● Landing Page ● Matching Service ○ ProblemPage ○ TopicPage ○ MatchingPage ○ EditorPage ● Chat Service ○ Chat ● Collaboration Service ○ CodeEditor	Report: ● Architecture Diagram ● Features ○ Room management service sequence diagrams ● Design Decisions ○ Choice in frontend

Matthew	Backend: ● User service ● Question service ● History service	Report: ● Design Patterns ○ MVC, Observer ● Design Decisions ○ Architecture ○ Database choice ● Features ○ User service ○ Question service ○ History Service ● Reflection
Hazel	Frontend: ● Login and Sign up page ● Questions page ● Home page with history service ● Profile page	Report: ● Purpose ● NFR justifications ● Features ○ Frontend ○ User service ○ Question service ● Deployment ● Development process ● Reflection ● Conclusion

Functional Requirements

User Service

S/N	Functional requirement	Priority	Done
1.1	The system should allow users to create an account with username and password.	High	Yes
1.2	The system should ensure that every account created has a unique username.	High	Yes
1.3	The system should allow users to log into their accounts by entering their username and password.	High	Yes
1.4	The system should allow users to log out of their account.	High	Yes

1.5	The system should allow users to delete their account.	High	Yes
1.6	The system should allow users to change their password.	Medium	Yes
1.7	The system should store the the roles of the user (TA/ Teacher/ Students/ Admin) (Users start out as students)	Medium	Yes
1.8	The system should allow certain roles(Teacher) to change the roles of other users(Student -> TA)	Low	No

Matching Service

S/N	Functional Requirement	Priority	Done
2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High	Yes
2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High	Yes
2.3	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High	Yes
2.4	The system should provide a means for the user to leave a room once matched.	Medium	Yes
2.5	The system should allow users to rate each other after the session ends. (nice to have, maybe rating system)	Low	No

Collaboration Service

S/N	Functional Requirement	Priority	Done
3.1	The system should have a code editor.	High	Yes
3.2	The system should allow users to edit the code at the same time.	High	Yes
3.3	The user should be able to download the code created along with any relevant metadata (student paired, question presented) at the end of the session	Low	No
3.4	The code editor should support code syntax highlighting and formatting based on the user's chosen language.	Low	No

Chat Service

S/N	Functional Requirement	Priority	Done
4.1	The system should have a chat box.	High	Yes
4.2	The system should allow users to send messages at the same time	High	Yes
4.3	The system should allow users to select their temporary role for the question session (Eg: Examinar, student, Frontend_dev, backend_dev)	Low	No

Question Service

S/N	Functional Requirement	Priority	Done
5.1	The system should be able to store a question bank indexed by difficulty level and topic.	High	Yes
5.2	The system should retrieve a “random” question with a given difficulty level and topic when requested.	High	Yes
5.3	The system should be able to store new questions created by TAs or Teachers	Medium	Yes
5.4	The system should be able to receive and store feedback from users about the questions given	Low	No
5.5	The system should try to give questions that the users have not attempted previously.	Low	No

History Service

S/N	Functional Requirement	Priority	Done
6.1	The user should be able to see the question and its difficulty and topic level of all their previous matches.	High	Yes
6.2	The user should be able to see the last edited code of their past matches.	Medium	Yes
6.3	The user should be able to see the chat history of their past matches.	Low	No

Non-Functional Requirements

All of the requirements stated below were implemented.

Security

S/N	Non-Functional Requirements	Justification	Priority
1.1	Session management should be done using HttpOnly cookies.	HttpOnly cookies are more secure than storing on localStorage and can prevent attacks as it is only accessible server side.	Medium
1.2	Users' passwords should be hashed and salted before storing in the DB.	Using a one way hashed and salted password makes it very hard for attackers to brute force or guess. If the database is leaked, actual passwords still can't be retrieved as they are hashed.	Medium

Performance

S/N	Non-Functional Requirements	Justification	Priority
2.1	What both users see should be eventually consistent once both users stop editing (No desync)	Users need to see the same code editor output at every single time for coherent cooperative coding.	High
2.2	The users should be able to see messages sent by other users within 2 seconds.	The chat feature should be responsive enough so that the users can communicate easily without lag, and understand each other.	Medium
2.3	The users should be able to see code edits done by other users within 1 second.	The code edits should be as fast as possible so that the application feels responsive.	Medium
2.4	The user should be redirected to the code collaboration page within 5 seconds of a successful match.	The user should not have to wait too long after getting a match.	Medium

Scalability

S/N	Non-Functional Requirements	Justification	Priority
3.1	Up to 10 users should be able to	The application should be able to	High

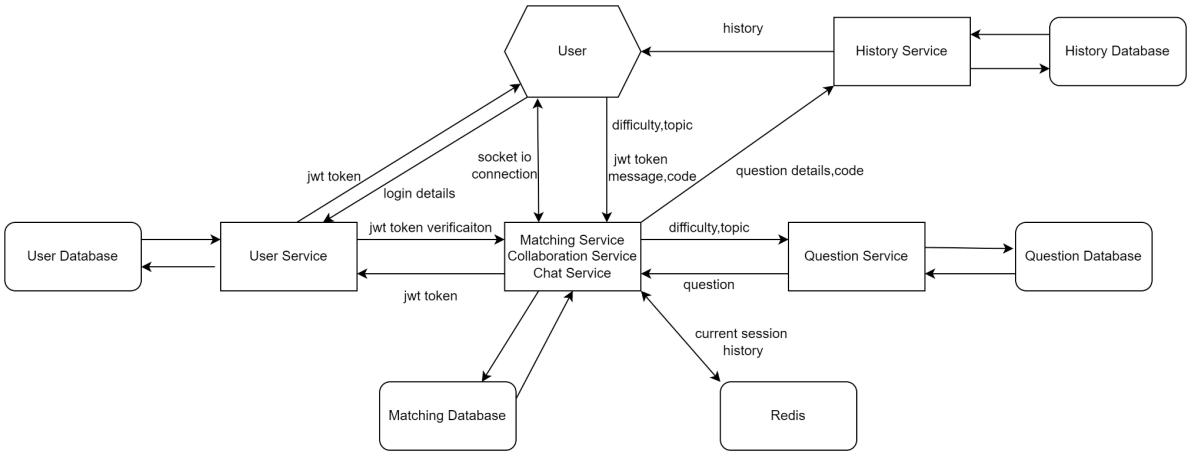
	use the matching service at the same time.	handle matching multiple users using it concurrently to maintain the user experience and remain usable.	

Availability

S/N	Non-Functional Requirements	Justification	Priority
4.1	The web application should be able to be accessed via popular browsers such as Google Chrome, Firefox, Microsoft Edge and Safari.	Cross browser compatibility needs to be ensured since various users use different browsers.	High

Developer Documentation

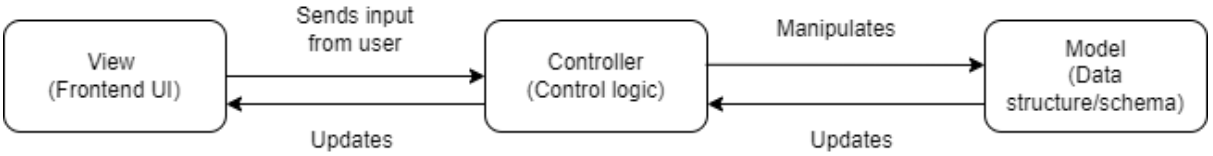
Overall Application Architecture



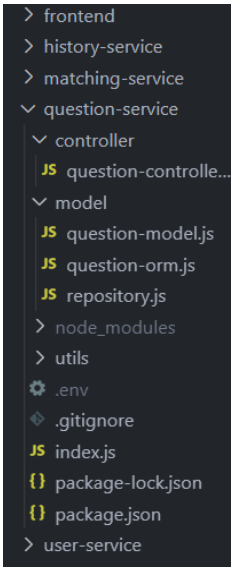
There are 4 main microservices used in our microservices architecture application. These microservices consist of the User service, Question Service, History service and Room management service (which is made up of matching, collaboration and chat). Users will use the frontend in order to interact with these microservices through REST APIs. Each microservice uses their own MongoDB database which is hosted on the cloud via MongoDB Atlas.The design decisions and design patterns used, and an explanation of each microservice will be discussed below.

Design Patterns

MVC design pattern



In the microservices architecture we have chosen, we utilise the Model View Controller pattern with respect to each individual microservice. However, our MVC pattern is not the same as the normal MVC pattern. In our design, the user sends inputs through the view (frontend UI), and the controller acts as a middleman or mediator between the views and the models. The controller will contain most of the backend logic and will change the state of the Model by adding or updating it. This means that the model cannot directly update the View and the View is only updated after getting responses from the Controller.
Here is a screenshot of how the question service code in our microservice architecture backend is structured:



Pub Sub Pattern

The chat, collaboration and matching microservices utilise Socket.IO for real time communication. In particular, it is used in order to implement a Pub-Sub pattern.
We first subscribe to our desired combination of difficulty and topic, to see if there are any existing sockets ready to pair up with. If not, we stop listening to our subscription and publish our own availability to that particular difficulty-topic partition, which we make available for 30 seconds before we indicate that our offer expired.
If we did see an advertised partner when we subscribed, we will use the published information to build a socket.io room.
Note that we used MongoDB as our place to store our published information for ease of development. MongoDBis needed to make the pub-sub pattern persistent.

Observer pattern

Implementing Socket.IO for the pub sub pattern also follows the observer pattern. In this case, the Socket.IO clients (on web browsers) act as subscribers and publishers themselves. The Socket.IO server facilitates this by connecting the clients using a room id. When there are new updates (like typing in code editor or new chat messages), the clients will publish to the server, which broadcasts the updates to the other clients. This keeps all the clients in sync. If a client leaves the room, they will not listen to any new updates for the chat or code editor. There is low coupling as there can be as many clients added to the same room if needed for future updates.

Design Decisions

Choice in frontend technology		
	React	Vue
Scalability	Template code written in JSX is more easily reused ,making it more scalable	Template code mainly written in HTML and reusing it can get more complex, making it less scalable
Development	React is mainly used for complex web applications as components are more reusable.	Vue is mainly used for lightweight and smaller web applications.
Tools and Libraries	React has a large amount of libraries and tools built by the community to help speed up development process	As Vue is a newer framework, it has a smaller community and hence less tools and libraries as compared to React.
Learning curve	Steep learning curve as React uses JSX syntax which is more complex	Developer friendly as Vue uses HTML-based template syntax
Final decision	We chose to use React as the PeerPrep project was a complex web application with multiple pages and has the need for reusable components. Furthermore, the team has experience with React hence the learning curve was not a big downside.	

Choice in overall architecture		
	Monolithic Architecture	Microservice Architecture
Scalability	Vertically scalable only, might lead to resource wastage.	Each individual microservice can be horizontally scaled depending on demand, less wastage of resources.
Development	Development may be simpler and faster as there is no need to	Easy to split up work between a team.

	manage communication between services. High coupling within a single codebase, may be hard to make changes.	Development may take longer as we need to build communication interfaces between individual microservices. Low coupling between microservices, easier to make changes.
Reliability	If service fails, the whole application will crash and be down.	If one service fails, it will not bring down the rest of the application.
Security	Single, larger attack surface	Multiple, smaller attack surfaces
Final decision	We chose the microservice architecture due to a combination of the above considerations. We felt the microservices architecture made more sense as it is more secure, reliable and easy to scale. Development being split up easily helps to reduce merge conflicts and makes it smoother due to the separation of concerns principle.	

Choice in Database		
	Relational Database (PostgreSQL)	NoSQL (MongoDB)
Scalability	Vertically scalable	Horizontally scalable
Development	Some members are familiar with PostgreSQL but it has a high learning curve.	More members are familiar with MongoDB and it is easy to use out of the box.
Schemas	Relational schemas are rigid, and it is very hard to make changes to predefined schemas.	Flexible, dynamic schema for unstructured data. MongoDB uses Document store.
Final decision	We decided to go with a noSQL database, MongoDB as it has a smaller learning curve versus a relational Database like PostgreSQL. MongoDB allows for more flexibility and data schema can be easily changed (if needed when introducing new microservices or requirements). There is also a limit to vertical scaling so we decided to use the horizontally scalable and easily implementable MongoDB. Furthermore, MongoDB Atlas which we use makes it very easy to use a cloud based database.	

Choice in cache technology for session info

	MongoDB	Redis
Scalability	Horizontally scalable	Horizontally scalable (Clusters only)
Development	More members are familiar with MongoDB and it is easy to use out of the box.	Not very familiar
Schemas	Flexible, dynamic schema for unstructured data. MongoDB uses Document store.	Flexible, dynamic schema for unstructured data. Redis user Key Value store
Storage	Traditional Storage	Fast In-Memory
Final decision	Redis fast In-Memory cache is much more important than the lack of familiarity with Redis, due to how frequently the cache is expected to be written to.	

Choice in client to client data synchronisation (Collaboration + Chat)		
	Plain Socket.io	Yjs
Reliability	Very naive, bare-bones implementation	Robust implementation, tuned for exactly the problem task
Development	Very experienced at the time of decision point	Not very familiar at the time of decision point
Integration	Very easy to implement, with minimal implementation time	At least somewhat difficulty to implement and integrate, with uncertain time table
Final decision	Although Yjs would have been perfect for the task, it would have taken an uncertain amount of time to actually implement, and risk busting our looming deadline, so plain Socket.io was chosen for its fast implementation path.	

Tech stack

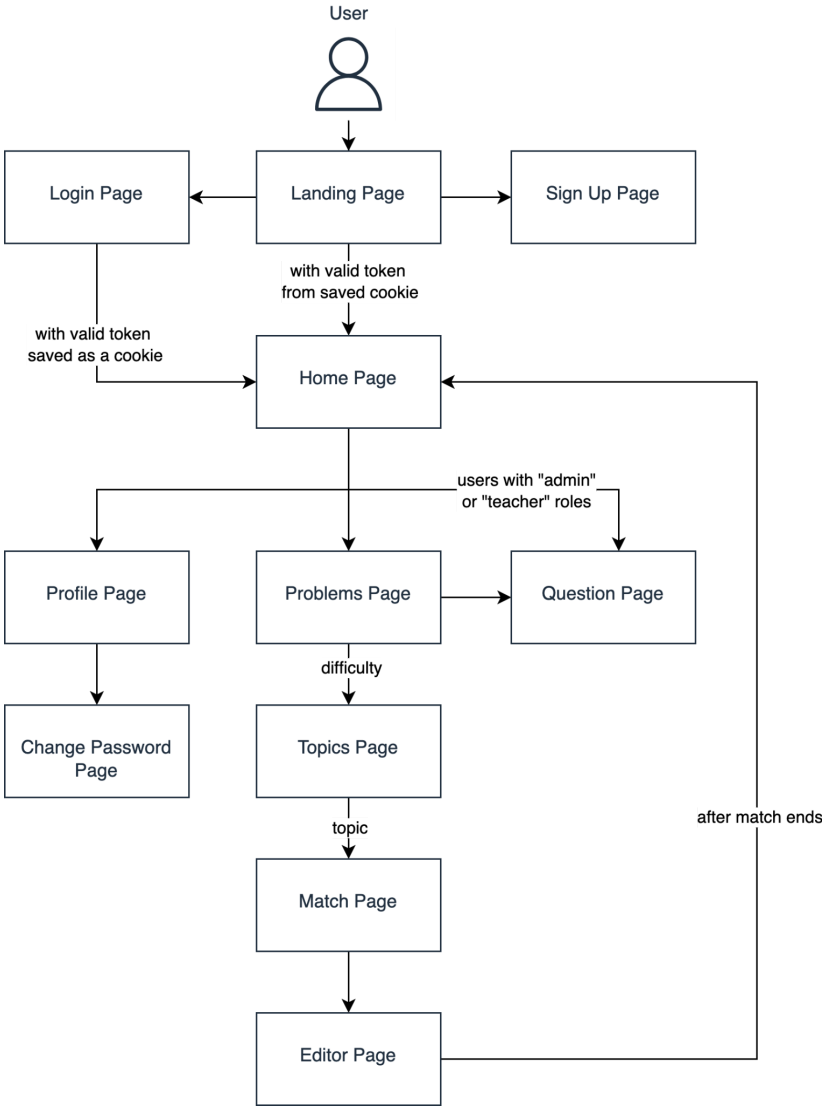
Component	**Technology**
Frontend	React
UI Component Library	Material UI
Backend	Express.JS, Node.JS
Pub-Sub Messaging	Socket IO

Caching	Redis
Database	MongoDB
Project Management Tool	Github Issues
Deployment	Docker

Features

Frontend

The framework used for our web application to develop the frontend is ReactJS. The styling library used was Material UI. The routing was implemented using the npm package called React Router Dom.

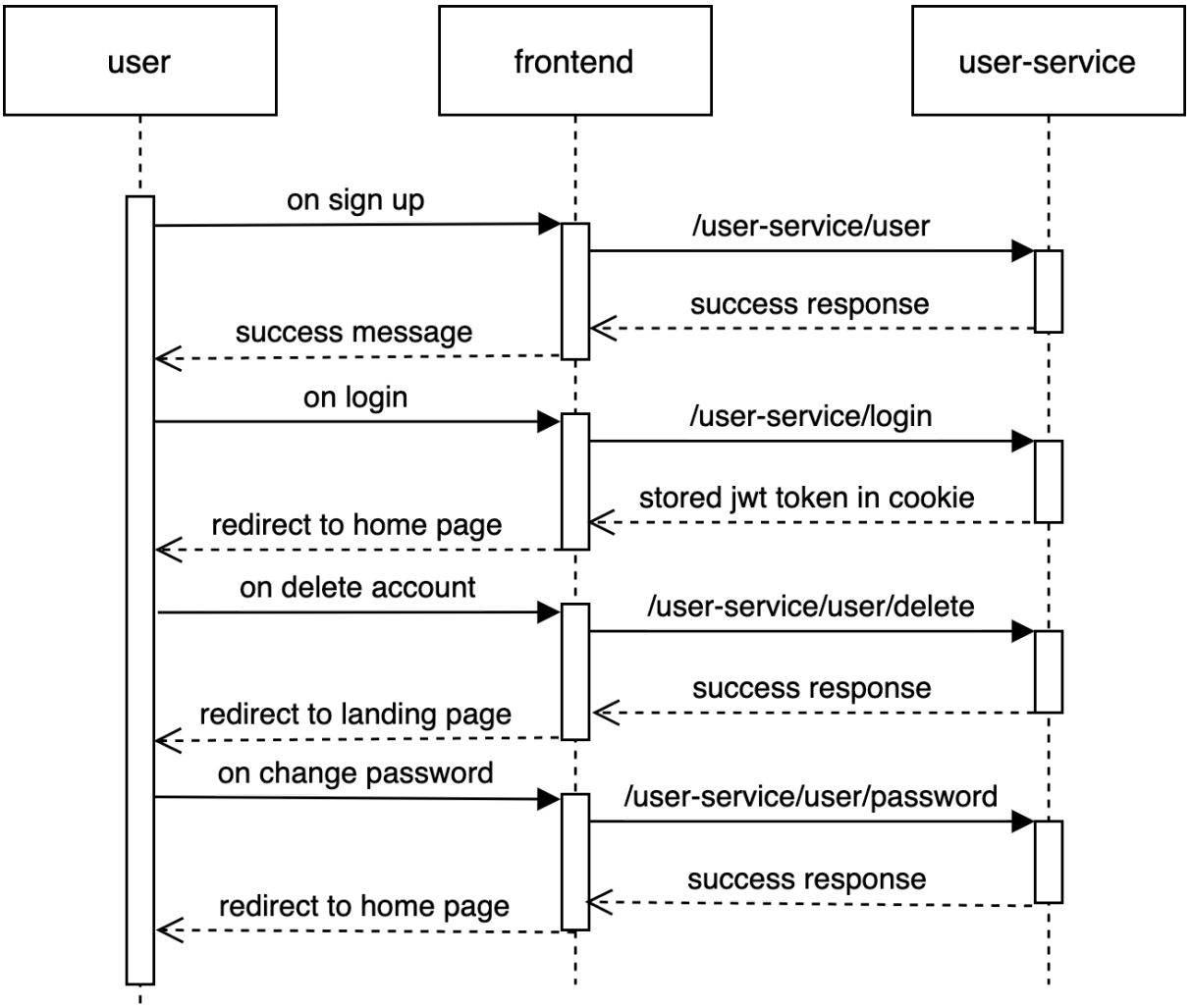


The diagram above demonstrates the flow of our application. If the user has a saved cookie with a valid token in the browser, the application will redirect them to the home page.

Otherwise, they will need to sign up and/or login. Upon login, a token created via JWT will be saved at the browser as a cookie.

User Service

This microservice implements user authentication and authorization. Each user will be assigned a role of either “Student”, “TA”, “Teacher” or “Admin” with increasing authority. The default role for all new users will be “Student”, and in order to get their account upgraded they will need to send an email to an account we have set up where manual checking of their credentials are done for now. Only “Teacher” or “Admin” users will be allowed to access the questions page where they can view all existing questions in the database and also add new questions. Once the user creates an account, we use the bcrypt library to hash the password and store it in our database’s user database.



Authentication

We have implemented authentication through the use of JWT tokens stored in HttpOnly cookies. After a user logs in, a JWT token containing the user’s username and role is encrypted by signing with a secret. We attach a HttpOnly cookie containing the encrypted JWT token of the user to the response. This is so that we do not need to store JWTs in local

storage. HttpOnly cookies prevent client side scripts from accessing the cookie, so that attacks like cross-site scripting will be less effective. It also reduces the complexity of configuring headers from the frontend as the cookies will automatically be attached to the requests made by the web browser after logging in.

For security reasons, once the user logs out, we blacklist the token to ensure that the user is not able to log in with the same token again. This is done by storing past tokens and comparing against the database when a user sends a new request.

Authorization

We have implemented authorization using Roles for the user service. As mentioned above, the roles include “Student”, “TA”, “Teacher” or “Admin”. In the question service and history service, only users with the roles ‘Admin’ or ‘Teacher’ will have the ability to create questions and history records. This is so that only verified and authorised users can have the ability to affect other users’ data or what they can access.

Room Management Service

Note that these services were grouped together under one ‘microservice’ for ease of implementation. This service can be split into two as a future improvement; Matching service, and Chat + Collaboration Service.

Room connection flow:

1. Clients connect to matching service after choosing desired difficulty and topic (Socket.io)
2. Once matching service finds a match, it does the following
    a. Clients chosen difficulty and topic will be sent to Question service which queries a random question that fits the criteria from the question database
    b. Matching Service receives a question from the question service and passes the question to each of the clients
        i. It also stores the question in cache for later storage in History service
    c. Matching services starts socket.io listeners that implement Code collaboration service and Chat service
        i. These listener also stores the chat history and current code for later saving in History service
3. Matching Service starts listening for “Leave Room” events or disconnects.

At this stage, each client has a single socket.io connection, that implements the following:

- Code Collaboration Service: For code
- Chat Service: For chat
- Matching Service: Listens for clients leaving the logical session.

Disconnect occurs as follows:

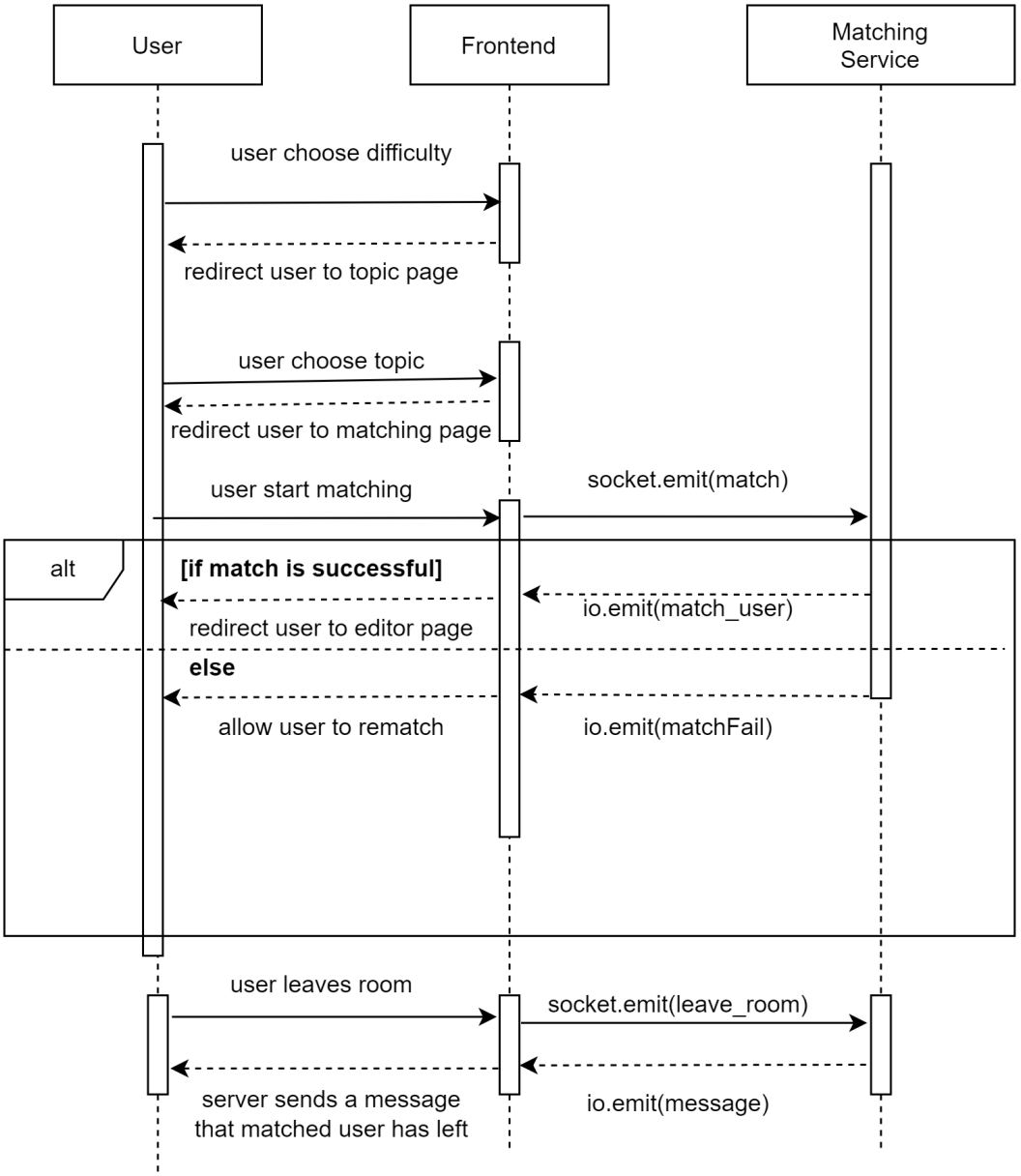
4. Either a client disconnects from Matching Service, or sends out a “Leave Room” event. The matching service then indicates to the other client to close down the connection
5. Furthermore, the Matching service takes the cached question, code and chat, and saves it with History service, alongside some other metadata

6. Clients either realise that the Matching service connection is gone, or receive feedback that the other client has disconnect

Matching Service

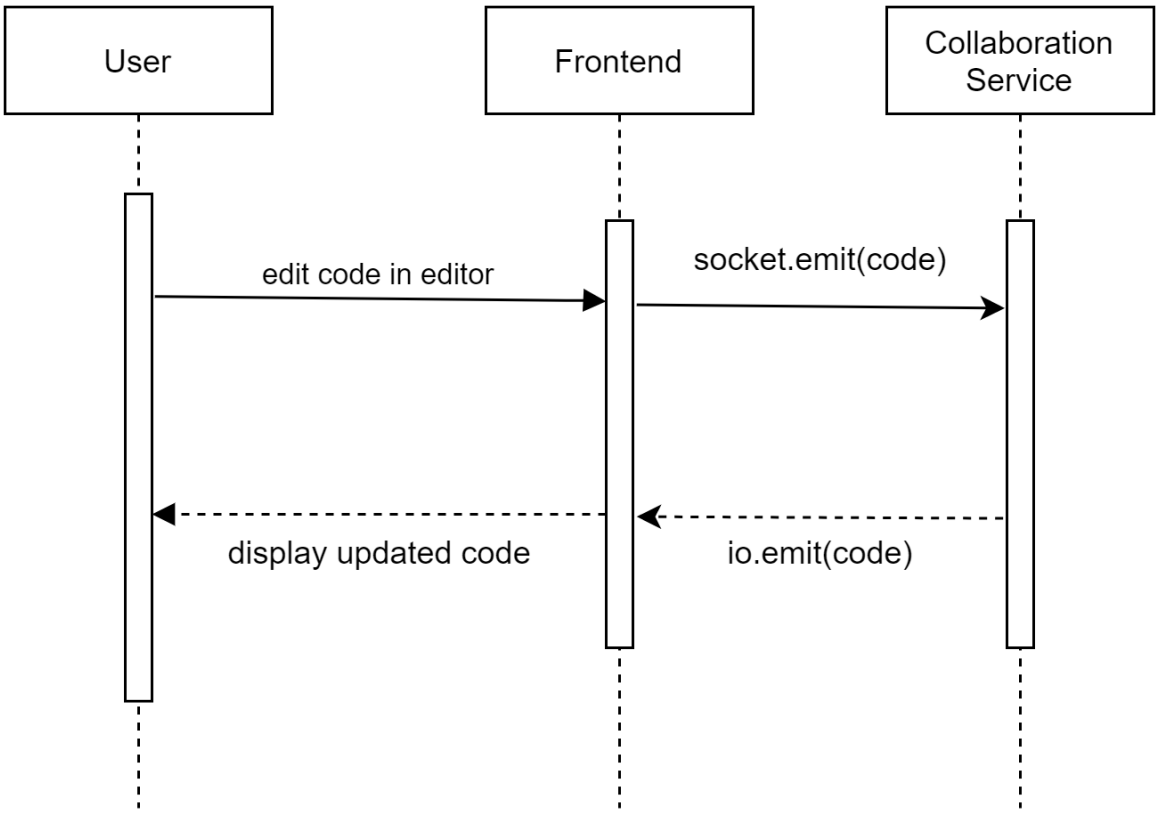
This microservice is responsible for matching users using the application. After selecting the difficulty and topic, a match event will be emitted by the frontend to the matching service. The matching service will then match users who have selected the same question topic and difficulty. It is also the socket.io server which is responsible for communicating with the users during the session.

After getting a match, it talks to the Question service to retrieve a random question, with the matching difficulty and topic, saves it to a localhost Redis cache, and serves the question to the Frontend via a socket.io listener.



Collaboration Service

Provides the mechanism for real-time collaboration (e.g., concurrent code editing) between the participants in the room. Implemented using basic socket.io technologies. Codemirror library is used on the frontend to provide users with an interactive code editor. Once completed, the code (which is stored in a localhost Redis cache) will be sent to the History service for archival.



Chat Service

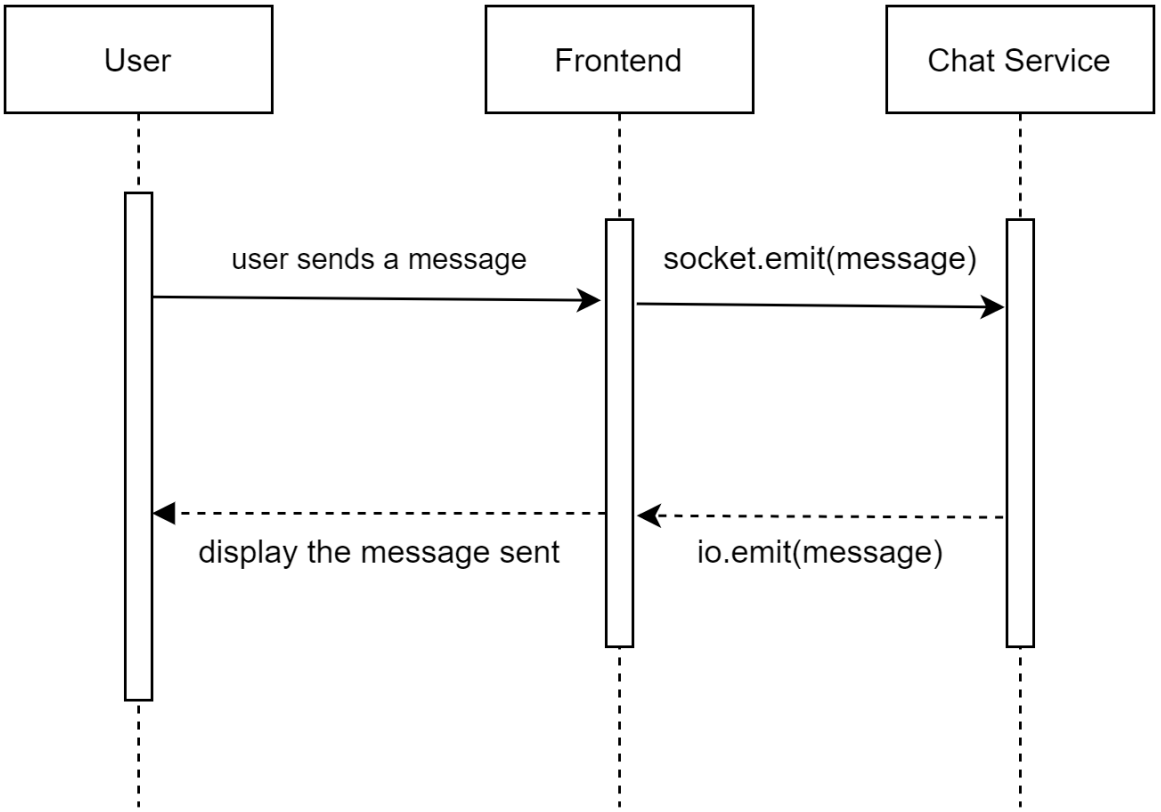
Our chat service provides, as in the name, basic chat services. Socket.io is ideal for this as it can fulfil the role of a transient point-to-point pub-sub architecture. For real time chat between 2 clients, this is just fine.

Furthermore, the ‘transient’ nature of the pub-sub system matters a lot less with socket.io’s automatic reconnection, which communicates over what is essentially TCP, which makes any messages very reliable, with messages only dropping because either because of serious outage of one of the clients, or deliberate misconnection.

Lastly, the bidirectional nature of socket.io means that it is suitable for the equally bidirectional nature of chat.

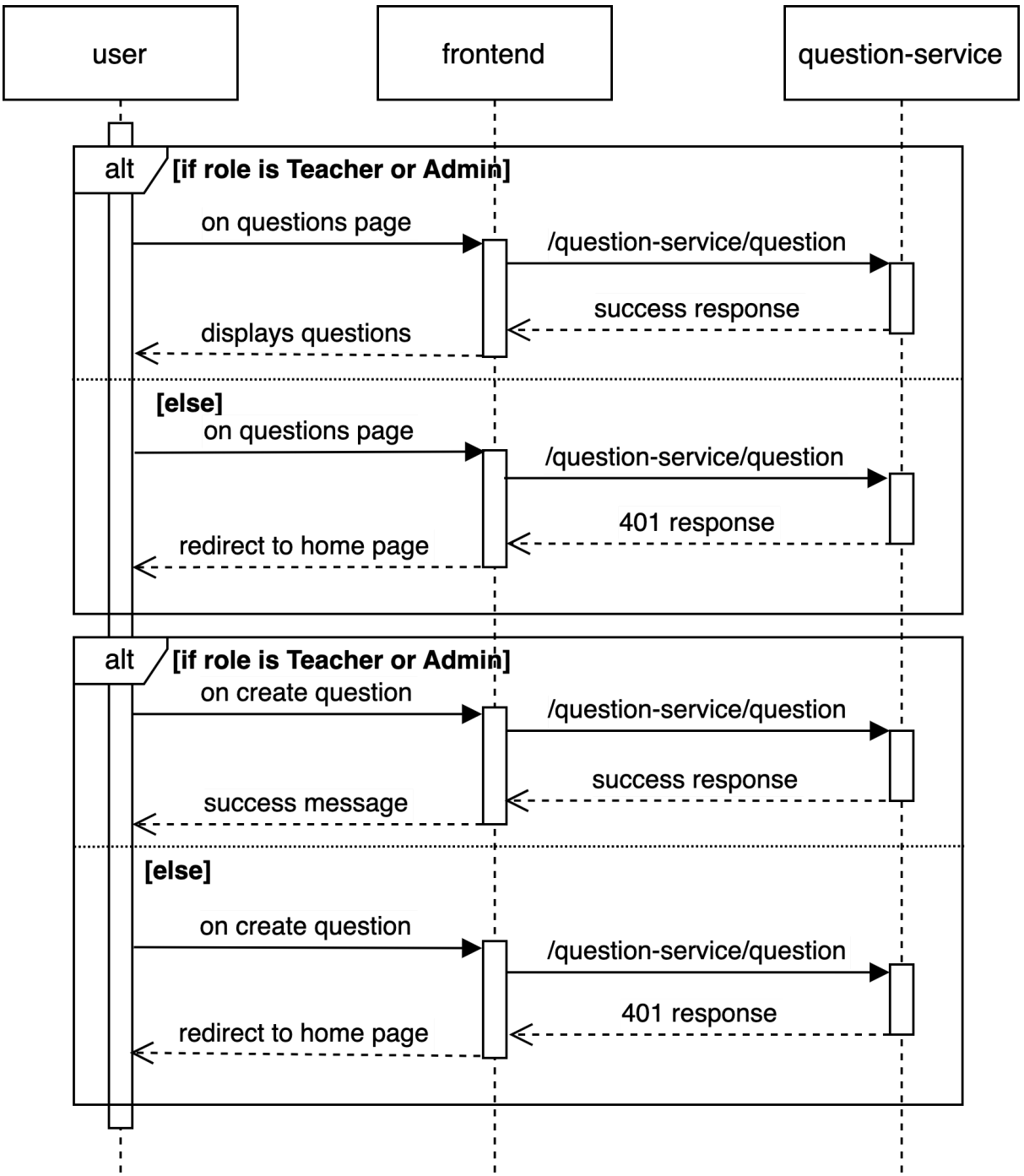
Much of the benefits of socket.io outlined for chat service applies to collaboration service.

Note that the chat service is first logged onto a localhost Redis cache, which is flushed to the History service at the end of the match session.

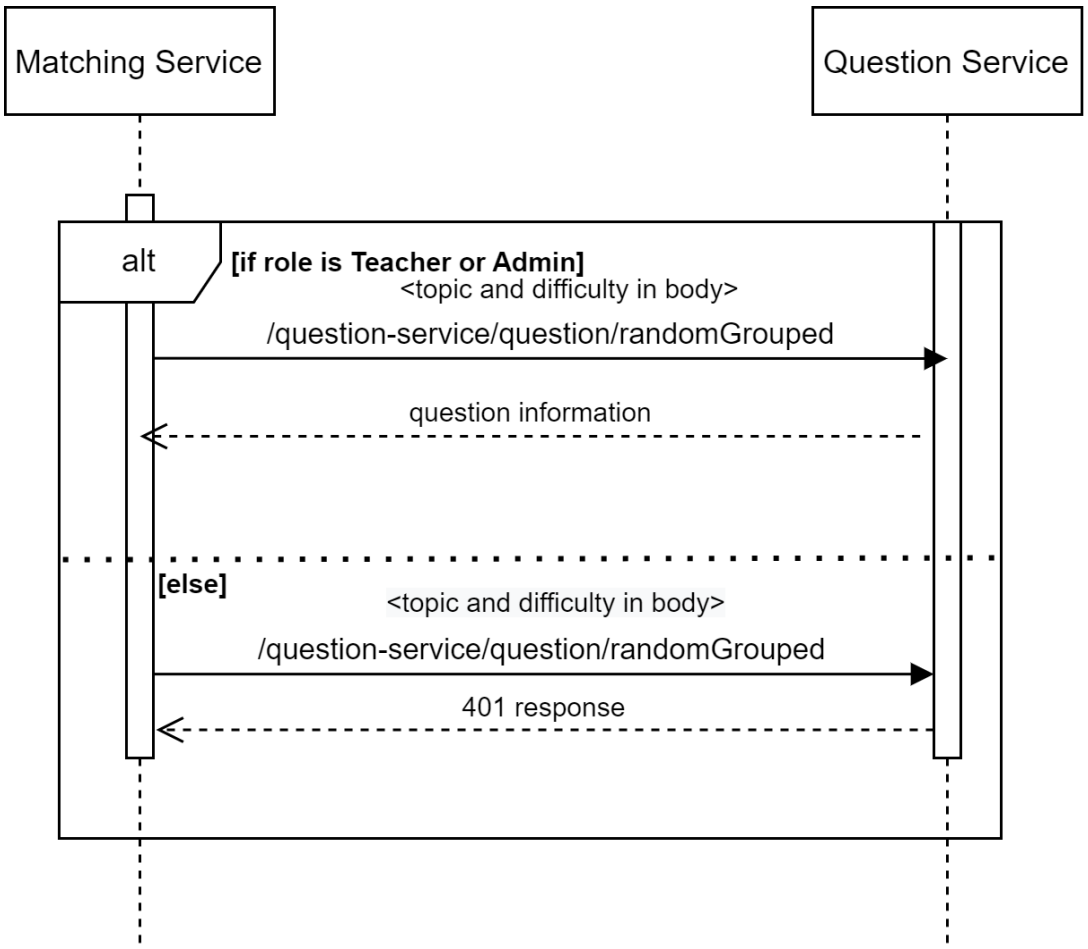


Question Service

This microservice is responsible for storing a question bank categorised by difficulty level and topic. It will also retrieve a question based on the chosen categories to the rooms created by Room Management Service once the participants are matched.



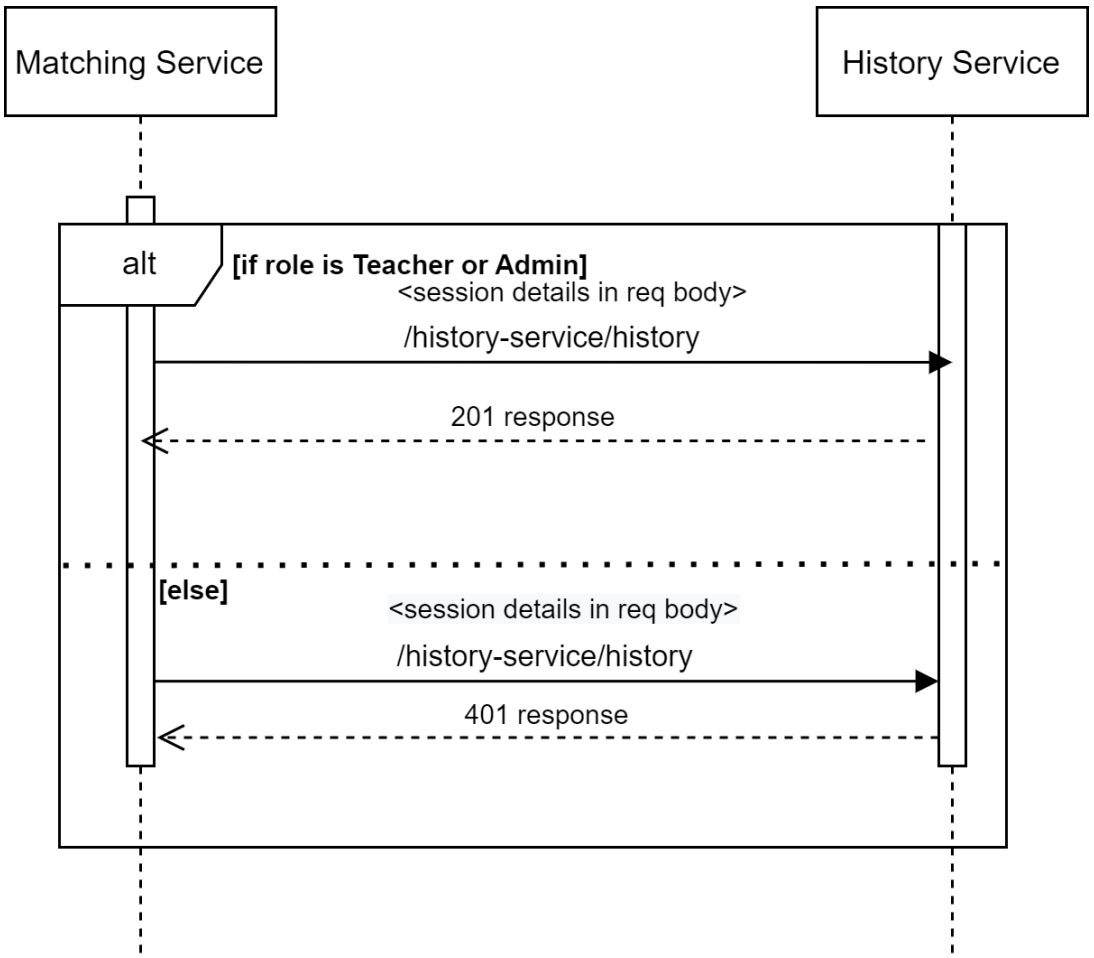
The users with roles ‘Teacher’ or ‘Admin’ can also send a POST request from the frontend to insert questions into the database, in order to view all questions, they can send a GET request from the frontend.



The question service also interacts with the matching service. For a given topic and difficulty, the question service looks up questions in its database that match the topic and difficulty, and returns a random one to the matching service. The matching service then handles sending the question chosen for the frontend. The sequence diagram above explains the interaction. The matching service is permanently logged into an admin role so it is able to get questions from the question service.

History Service

The history service is responsible for storing history about coding sessions created by the matching service. After one of the users finishes the session by leaving, the matching service interacts with the history service by sending a POST request containing the two usernames of the participants, the question ID, the code history and the finish date. Below is a sequence diagram demonstrating how the matching service interacts with the history service. The matching service is logged in with an admin role so it can send POST requests to add new histories.



As long as a user has a valid JWT token in their cookies, they will be able to send a GET request to the History service in order to get a list of their past coding sessions.

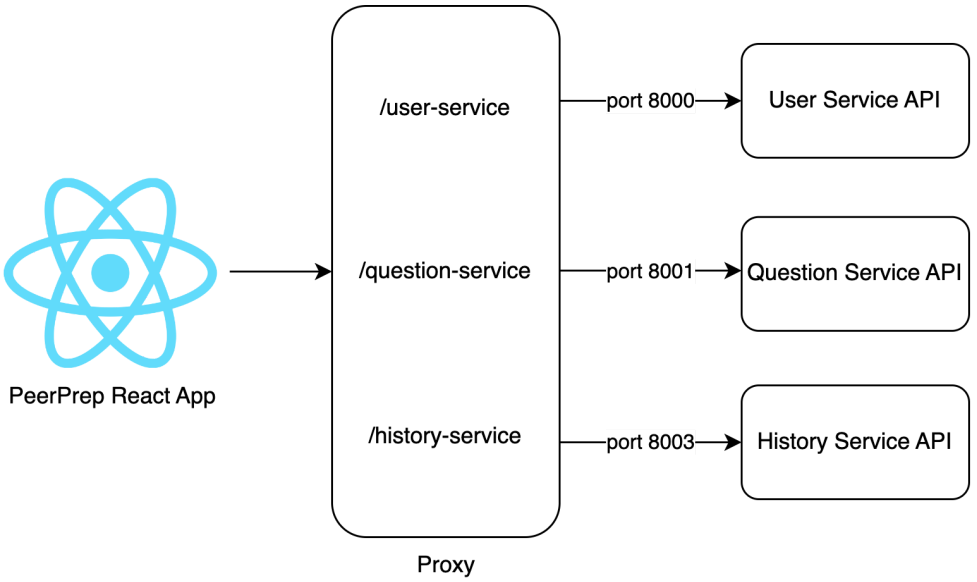
Deployment

Our application can be deployed locally with the following steps:

Step	**Instructions**
Clone the repository	Go the the folder on your local machine that you wish to clone the repository to and run this following command: 1. `git clone https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g46.git` *NOTE: If you wish to clone the repository in other ways the repository can be accessed from here.*
Run redis cache	1. Install Docker Desktop 2. Run the following command: `docker run --name=redis --detach=true`

	`--publish=6379:6379 redis`
Run the microservices	Run the following commands: 1. `npm install` 2. `npm start` In the different folders in this order: 1. user-service 2. question-service 3. history-service 4. matching-service
Run the frontend	In the frontend folder, run the following commands: 1. `npm install` 2. `npm start`

Since we have implemented a microservice architecture, the frontend needs to proxy into multiple APIs.



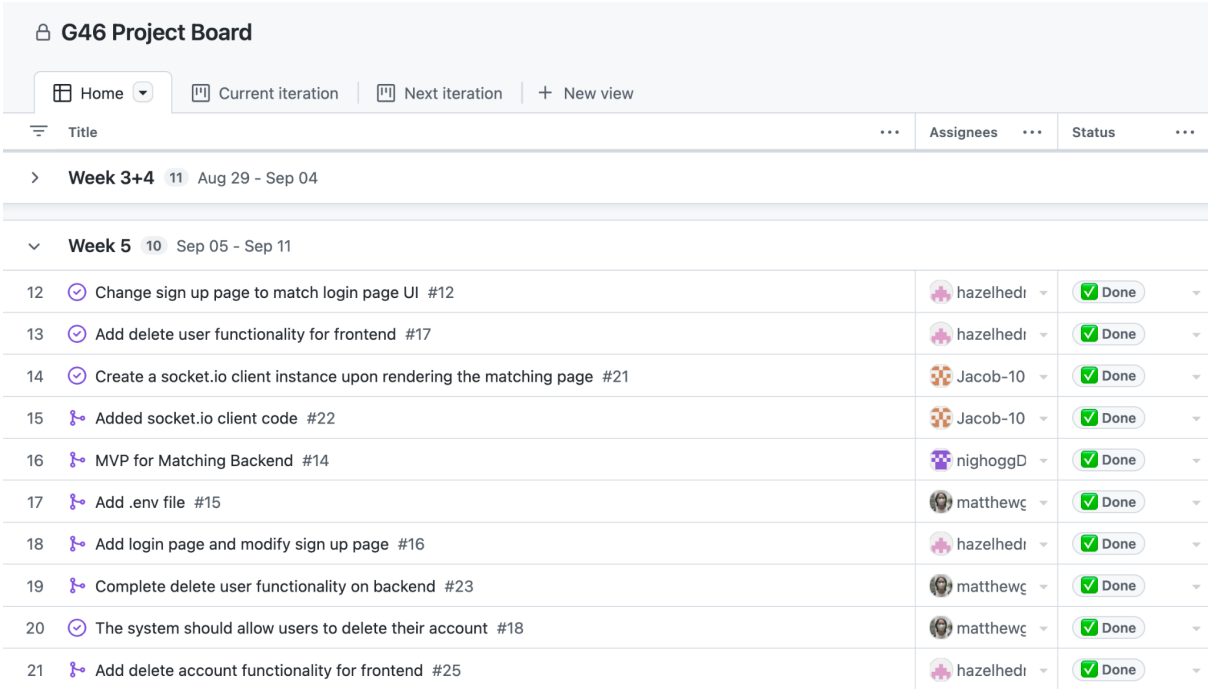
The [http-proxy-middleware] npm package was used to create a manual setup to allow the frontend to access multiple endpoints depending on the URL handle.

Development Process

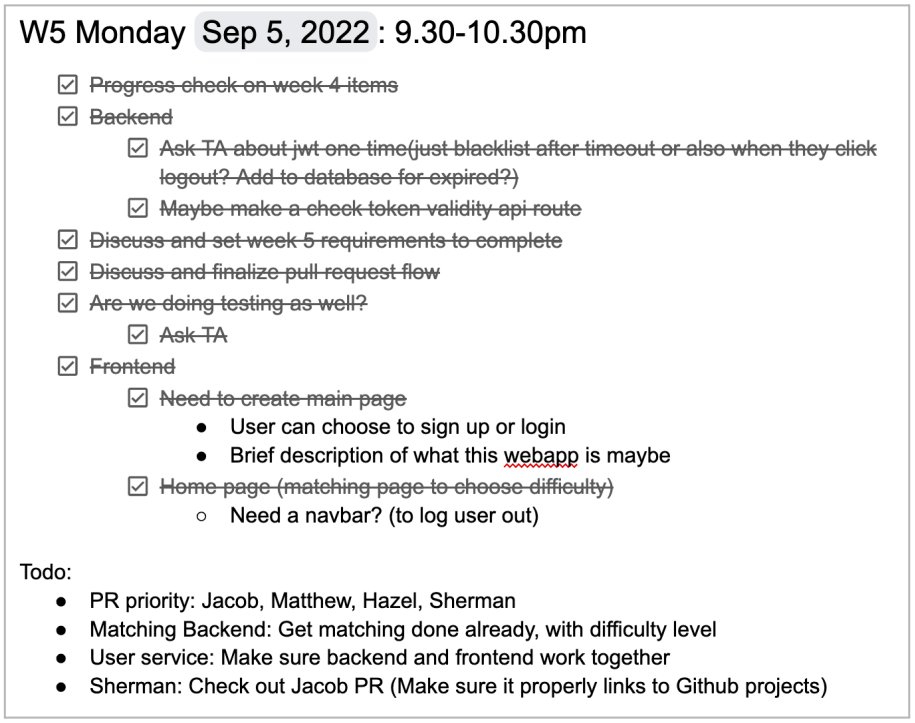
The development of the application was split to 3 milestones.

Milestone 1	1. User Service 2. Matching Service 3. Basic UI
Milestone 2	1. Collaboration Service 2. Question Service

Milestone 3	1. Chat Service 2. History Service 3. Fancy UI



Project management was done using github issues and we also practised code reviews to ensure that at least one other teammate has approved a pull request before integrating it. We also organised weekly meetings on Mondays and Wednesdays to update each other on our individual progress and participate in discussions to decide key architectural decisions.



W5 Monday Sep 5, 2022 : 9.30-10.30pm

- [x] ~~Progress check on week 4 items~~
- [x] ~~Backend~~
  - [x] ~~Ask TA about jwt one time(just blacklist after timeout or also when they click logout? Add to database for expired?)~~
  - [x] ~~Maybe make a check token validity api route~~
- [x] ~~Discuss and set week 5 requirements to complete~~
- [x] ~~Discuss and finalize pull request flow~~
- [x] ~~Are we doing testing as well?~~
  - [x] ~~Ask TA~~
- [x] ~~Frontend~~
  - [x] ~~Need to create main page~~
    - User can choose to sign up or login
    - Brief description of what this webapp is maybe
  - [x] ~~Home page (matching page to choose difficulty)~~
    - Need a navbar? (to log user out)

Todo:

- PR priority: Jacob, Matthew, Hazel, Sherman
- Matching Backend: Get matching done already, with difficulty level
- User service: Make sure backend and frontend work together
- Sherman: Check out Jacob PR (Make sure it properly links to Github projects)

The picture above shows an example of the agenda set for a meeting. Having regular meetings and clearly defined agendas for them helped us to consistently develop our application to meet our milestone goals.

Future Enhancements

Component	**Enhancement(s)**
Pure Matching Service	Match the users together and pass on to the next service using a generated ‘room key’. Possibly also registering the ‘room key’ with the next service as valid.
Collaboration + Chat Service	This can use a Yjs, a CRDT implementation, as the backbone for both Collaboration and Chat service as one unified data object, which simplifies implementation and integration with the history service considerably. Frontend presentation of Collaboration service can be done using CodeMirror, which can bind with Yjs as the underlying data structure. ● Collaboration would be much improved due to CRDTs being largely motivated by this exact task of collaborative real-time editing, with explicit solutions for edit merging. Coordination of the Yjs data type can be done over websocket using the first party npm package y-websocket. ● However, to stick with the same technology as the pure matching service, we can instead coordinate over the more fully featured socket.io using the npm package y-socket.io.
Deployment to the Internet	As our main focus is to provide users with an interactive and high performance web application, we decided to deploy only to our local machine using native tech stack due to time constraints. The team hopes to create a Docker image using Docker Compose of the web application and use CICD tools like TravisCI to deploy to a cloud service in the future, as well as using Kubernetes in order to make the microservices scale when needed.

Reflection and Learning points

1. Communication between team members is essential in software development as we need to convey our thought processes clearly to each other so as to better work towards the common goal.
2. The usage of socket.io for bidirectional communication between server and client and how to integrate the socket.io calls into React frontend components for the room management service.
3. It was very rewarding learning how to separate up a product like PeerPrep into the microservices architecture and following design patterns to improve the development process.
4. It's very important to think about the Functional Requirements, Non-Functional Requirements and design choices before starting the project so that development can go smoothly.
5. The use of scheduling tools like Github Projects, as well as weekly meetings helped us keep on schedule and keep up with the milestones.

Conclusion

The module and project has taught us how to properly develop a full stack web application. It taught us how to consider qualities like security, scalability, availability as well as planning. We did not have the time to put our skills from the OTOT tasks to use for CICD and online deployment but the experience was still fruitful as we enjoyed learning and going through the project together.