CS3219 Group 47 Project report

1. Introduction **3**
- 1.1. Motivation 3
- 1.2. Target users and scope 3
- 1.3. The product and its features 3
- 1.4. Supported Platforms 3

2. Functional requirements **4**
- 2.1. User account management 4
- 2.2. Matching users 4
- 2.3. Communication 5
- 2.4. Let users program together 5
- 2.5. General application 6

3. Non-functional requirements **8**
- 3.1. Usability 8
- 3.2. Security 9
- 3.3. Portability 9

4. Architecture **11**
- 4.1. Architectural diagram 11
- 4.2. Microservices over monolithic 11
- 4.3. Authentication framework 12
- 4.4. Break down of microservices and their responsibilities 16

5. Frontend **18**
- 5.1. Features, FRs and NFRs 20
- 5.2. Key design choices 20
  - 5.2.1. Tech stack 20
  - 5.2.2. Authentication via JWT 21
  - 5.2.3. Global contexts to avoid prop drilling 21
- 5.3. The main workflow and how it works 21
  - 5.3.1. Sign up (/signup) and login (/login) 21
  - 5.3.2. The home/matching page (/home) 23
  - 5.3.3. The collaboration room page (/room?roomId={roomId}) 27
  - 5.3.4. The history page (/history) 29
  - 5.3.5. The settings page (/settings) 29
- 5.3.6. Development and deployment 30

6. User service **31**
- 6.1. Features, FRs and NFRs 31
- 6.2. Architecture 31
- 6.3. API 33
- 6.4. Examples (sequence diagrams) 36
- 6.5. Development and deployment 39

7. Question service **40**
7.1. Features, FRs and NFRs 40
7.2. Architecture 40
7.3. API 41
7.4. How it works 42
7.5. Development and deployment 42

8. Matching service **43**
8.1. Features, FRs and NFRs 43
8.2. Architecture 43
8.2.1. High level organization 43
8.2.2. Designing the queue service 44
8.3. Putting them together 45
8.4. Handling disconnections 48
8.5. API 49
8.6. Development and deployment 49

9. Collaboration service **50**
9.1. Features, FRs and NFRs 50
9.2. Architecture 50
9.3. External integrations 53
9.3.1. Y-WebRTC 53
9.3.2. Daily WebRTC Video 53
9.4. API 54
9.5. Sequence Diagrams 55
9.6. Testing 57
9.7. Development and deployment 58

10. Database: MongoDB **59**

11. How the services work together (under the hood) **59**
Create account, login, find match, enter room, collaborate 59

12. DevOps **61**
12.1. Sprint planning 61
12.2. Continuous Integration 61

13. Reflections **61**
13.1. Areas of improvement/features to implement 61
13.1.1. Ingress controllers 61
13.1.2. Code execution 63
13.1.3. Provide user feedback 64
13.1.4. More collaborative features 64
13.1.5. Telemetry 64
13.2. Contributions 64

1. Introduction

1.1. Motivation

Increasingly, students are facing challenging technical interviews when applying for computing-related roles. Even with the availability of platforms like LeetCode where one can get practice, their solitary and paywall-locked nature often mean that students are left with insufficient practice communicating their thought process or understanding where they went wrong. This has been worsened by the pandemic, where many students lost the opportunity to meet with peers in person and discuss solutions in class or more spontaneously.

The aim of our product, *PeerPrep*, is thus to provide students with an easy-to-use, collaborative platform that allows them to engage in pair-programming when working on technical interview questions.

1.2. Target users and scope

Our project is primarily focused on early-career individuals (where a good amount of technical interviews consist of algorithm problems, less so of other formats like system design). The selection of problems are taken from LeetCode or other open-source collections.

We scope our project as follows:

- Provide users with practice questions for technical interviews
- Provide users with a platform to do said questions with a similarly skilled peer with an IDE and appropriate communication/collaboration tools

1.3. The product and its features

PeerPrep is designed and built as a pair-programming interview preparation platform. It does so by **matching two users in real-time based on certain criteria** (e.g. problem difficulty), **directing them to a room** with a **technical interview question** and a **shared IDE** and **live chat functions** for collaboration.

1.4. Supported Platforms

PeerPrep is intended to operate within modern web browsers like Mozilla Firefox, Google Chrome and Microsoft Edge. *PeerPrep* is developed in a cross-platform (Windows 10/11, MacOS) environment.

2.Functional requirements

2.1. User account management

ID	Description	Priority	Function/element
Registration			
FR-ACC-1	User should be able to create an account using unique username and password	High	Sign up page
FR-ACC-2	The system should validate usernames (3-20 characters, no special characters).	High	Sign up page, backend validation
FR-ACC-3	The system should validate passwords (3-20 characters, no special characters, globally unique).	High	Sign up page, backend validation
Login			
FR-ACC-4	Users should be able to log in using the username and password they signed up with.	High	Login page
FR-ACC-5	A user should stay logged in if they have logged in within the last 24 hours and are using the same browser.	High	JWT authentication with expiry
Change credentials			
FR-ACC-6	Users should be able to modify their usernames and passwords if they are logged in. These usernames and passwords should conform to the validation conditions specified in FR-ACC-2/3.	Medium	User settings page
Logout			
FR-ACC-7	Users should be able to revoke the authorisation of the current session and log out.	High	Logout button

2.2. Matching users

ID	Description	Priority	Function/element
Pairing			

FR-MAT-1	Two logged in users who select the same difficulty should be paired and put in the same room	High	Matching page
FR-MAT-2	The user matching service must allow users to cancel their search for matches when they are not yet allocated a match by the system.	High	Matching page cancel button
FR-MAT-3	Users should be able to select at least three difficulties - easy, medium, hard for the criteria in matching to another user.	High	Matching page difficulty selection
Login			
FR-MAT-4	Users can send an invitation to a friend to work with them	Low	NA

2.3. Communication

ID	Description	Priority	Function/element
Chat			
FR-COM-1	Paired Users can communicate through text chat	Medium	Video/text chat via Daily API
Video			
FR-COM-2	Paired Users can communicate through video and voice	Medium	Video/text chat via Daily API

2.4. Let users program together

ID	Description	Priority	Function/element
Live editor			
FR-PRO-1	Paired users in the same room should be able to edit the same textbox	High	Codemirror editor in room page, which users y-webrtc to sync changes in real time

FR-PRO-2	Users can view and edit their own whiteboard editor in real-time to solve practice questions.	High	Codemirror editor in room page
FR-PRO-3	The editor should have line numbers for users to communicate about the code.	High	Codemirror editor in room page
FR-PRO-4	Users can re-enter a room, so that they can reattempt the question.	Medium	Users can access history of rooms or last room
FR-PRO-5	Code from previous attempts should persist when a user refreshes, disconnects or re-enters a room, so that he can continue where he left off.	Medium	Room page state persists
FR-PRO-6	Paired users in the same room should be able to see where the other user is editing so that they know what is going on.	Medium	Codemirror editor uses y-webrtc to identify other editor's cursor position
FR-PRO-7	Paired users in the same room should be able to see where the other user is highlighting so that they can communicate about the code.	Medium	Codemirror editor uses y-webrtc to identify other editor's highlighted text

2.5. General application

ID	Description	Priority	Function/element
Reconnection			
FR-GEN-1	Room should stay alive as long as one user is still connected	High	Room page socket connections
FR-GEN-2	User who lost connection should be able to reconnect to his previous room if still alive	Medium	Room sessions are managed by backend and can handle reconnections
History			
FR-GEN-3	Users should be able to look at the questions they have attempted before	Medium	User history page

FR-GEN-4	Users should only be authorized to view their own history	Medium	JWT authentication in querying room history

3.Non-functional requirements

There are many non-functional requirements - availability, performance, security, portability, sability, scalability, security, etc.

We think of *PeerPrep* as a proof of concept potentially working as SaaS and in an early stage of development. As a result, we aim for high usability, security and portability.

Here, we illustrate our top 5 considerations and why they are important.

NFR	Rationale
Usability	As a prototype, the end-goal is to build a product that is easy-to-use and useful. The UI should be intuitive and the path-to-value for the average user should be short and low resistance to encourage uptake.
Security	Security is a consideration that is hard to integrate after a lot of code has been written, and should be built into the application as a design choice.
Portability	As a small team and young product, a focus on being able to develop in various environments is important for consistency.
Scalability	At some point, with more users, scale will be important. Note that we can always integrate scaling solutions if the application is written with appropriate abstractions (e.g. abstraction over the database, services), scaling implementation (e.g. queue servers, load balancers, containerization) can be manageable.
Performance	Similar to usability, the application should feel snappy and usable.
Explaining their ranking	
We rank the NFRs in order of usability > security > portability > scalability > performance. We justify this ranking based on the fact that *PeerPrep* is a product in an early stage of development, where finding product-market-fit, important must-haves that are hard to integrate later and development speed are the most important. As scalability and performance can always be optimized, they are ranked as the last two. Security choices should be built-in from the start. Usability brings product-market fit, and portability helps with development speed.	

3.1. Usability

ID	**Description**	**Priority**	**Element**
NFR-USA-1	UI should be easy to navigate for users	High	Low resistance path to value with only 3 pages

			of login/signup, match, room.
NFR-USA-2	UI should be optimized for desktop usage	High	UI dimensions/scaling are fit to typical desktop dimensions (1080p, 2K resolutions)
NFR-USA-3	App should work on Common web browsers (Chrome, Firefox, Edge)	High	Able to do so.
NFR-USA-4	App should show useful feedback information to user's actions	High	Snackbars (e.g. error prompts) are provided.

3.2. Security

ID	Description	Priority	Element
NFR-SER-1	Passwords should not be stored in plaintext (kept secure by salting and hashing)	High	User service hashes passwords
NFR-SER-2	Password complexity requirement (minimum length, mix of letters and numbers)	Medium	Enforced on the front end when registering and changing password and on the backend with validation.
NFR-SER-3	Users should only be able to view and edit information for their own account	High	Enforced by frontend allowing access only to one's account, and by JWT used as identifier for backend API calls.
NFR-SER-4	Communication channels between components should be authenticated	High	JWT's and symmetric keys are used for REST/socket connections between frontend and backend services, and backend services with each other.

3.3. Portability

ID	Description	Priority	Element

NFR-POR-1	Running the application on the local environment should be containerized.	High	Dockerised all application components with Docker and docker-compose.
NFR-POR-2	All dependencies should be logged and controlled.	High	Dependencies are logged into the project repository as requirement files or package lock files.

4.Architecture

4.1. Architectural diagram

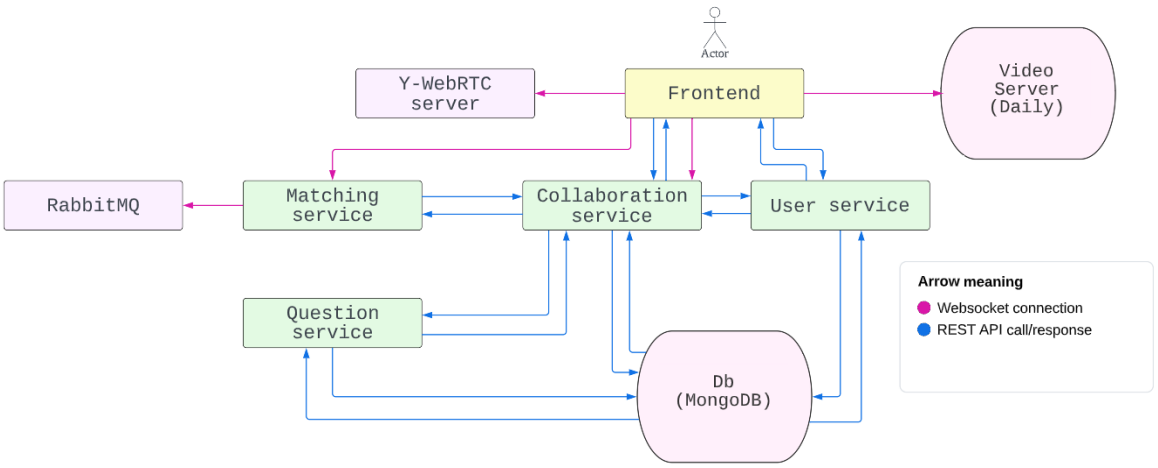


Figure 1: Architecture diagram of PeerPrep microservices and how they interact. External servers (non-self hosted) are indicated by round-edged boxes.

4.2. Microservices over monolithic

PeerPrep is designed with five main steps in mind - signup/login, find a match, join a room with a question, work on a problem collaboratively.

When we consider the actions within each of these workflows, it becomes apparent that we can segment the space into user-related, question-related, matching-related and collaboration/editing-related contexts within the subdomain of the *PeerPrep* space.

These spaces are relatively independent, and for that reason a natural microservices architecture design arises.

Aside from the intuitive formulation, the following issues and their resolutions are considered below:

Issue	Decision	Justification
Coupling in development, given the tight timeline	Microservices	A monolithic approach means that development, testing and integration is challenging with multiple developers due to merge conflicts and dependencies. Also, a monolithic approach means that different developers with different degrees of familiarity must agree to the same tech stack for the whole application. A microservices approach provides much more flexibility given the tight timeline and

		need for working in parallel.
Different loads expected at different parts of the application	Microservices	Matching-related services, collaboration-related services are likely to have high traffic, and less so for question- and user-related services. As a result, they may have different degrees of scaling needs (e.g. load balancing, number of machines) Having a microservice architecture allows different components to be treated differently and independently.
Need for communication (and thus authentication) channels between services		This is a drawback of microservices. An orchestrator was considered to reduce this need (reduces the need for up to N^2 channels to 2N), but because the graph of dependencies is closer to a sparse graph, this is not necessary.

4.3. Authentication framework

The use of JWTs and secret keys

Security is one of the required NFRs. There are three main types of communication in *PeerPrep* - synchronous REST API calls between microservices, synchronous REST API calls from the frontend to the microservices and websocket connections between the frontend and microservices.

With these in mind, we have designed our authentication framework to be such that a user (or user proxy - the frontend) interacting with backend services must have a JWT, and a backend service requesting some resource from another backend service must have a secret key for authentication.

Here is an overview of roles in this authentication framework:

Role	**Responsibility/actions**	**Components**
User making REST API call	Makes API calls to the backend services. For example, logging in is a POST request to the user service. Is expected to have a JWT as a cookie, provided by a JWT authenticator.	Frontend
User connecting to a	Connects to a backend	Frontend

backend service via a websocket	service via a websocket. The connection is expected to have a JWT provided by the JWT authenticator as a query parameter.	
JWT authenticator	Provides JWTs via cookies or in the body of the response to the requester with the correct credentials. For example, a login request with correct credentials would have an instruction to set the JWT as a HTTP-only cookie. Alternatively, a backend service with the correct secret key in the request would have the JWT returned in the response body.	User service
JWT requester (backend)	Requests for a JWT in the response body from the JWT authenticator. Needs to provide a secret key for authentication/authorization on request.	Any backend service. E.g. collaboration service.

JWT authentication for REST calls from frontend to backend services

1. The Frontend should have a HTTP-only JWT in the request
2. A microservice reads the JWT and asks the User Service to validate the JWT. On success, the microservice goes ahead.

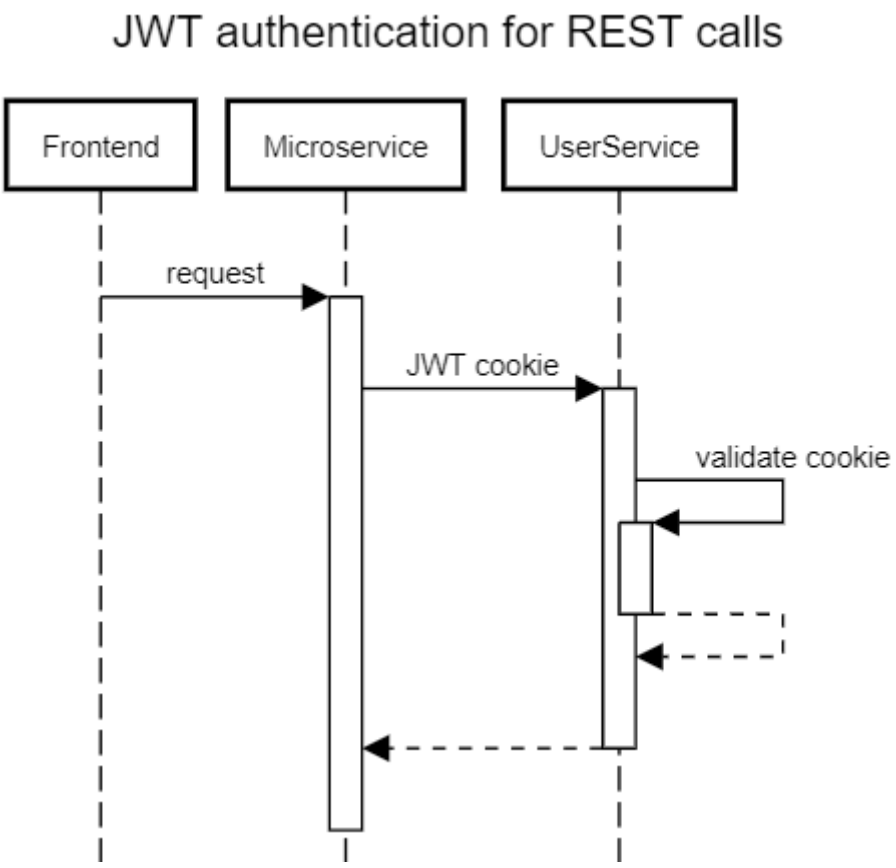


Figure 2: Sequence diagram illustrating how a microservice, when receiving a REST call from the frontend, authenticates it.

JWT authentication for websocket connections from frontend to backend

1. The Frontend makes a request to User Service for a new JWT.
2. In its websocket connection with a microservice, the frontend sends its JWT. The microservice reads the JWT and asks the User Service to validate the JWT. On success, the microservice goes ahead.

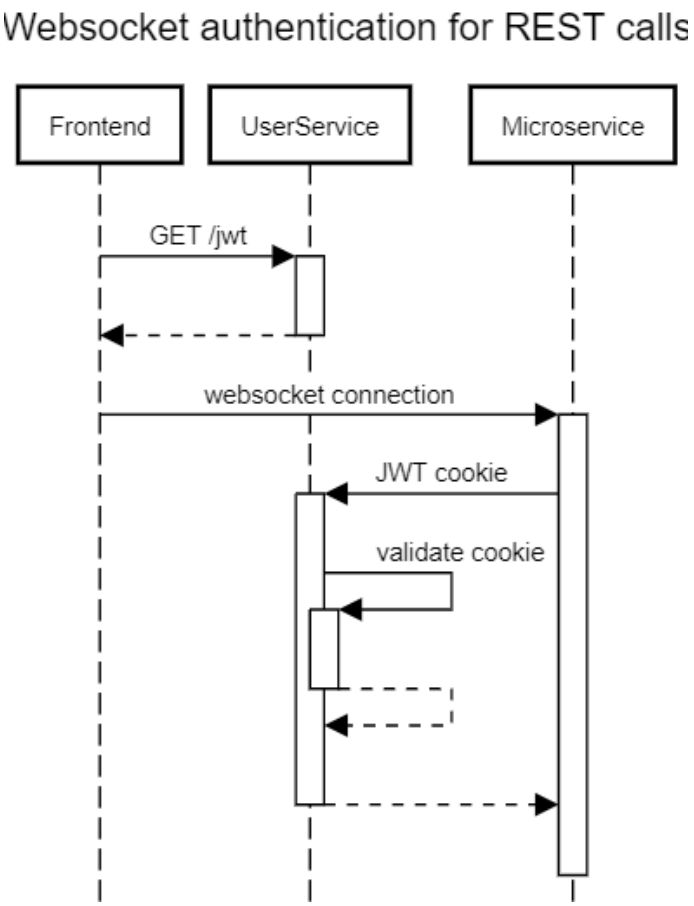


Figure 3: Sequence diagram illustrating how websockets are authenticated by a microservice - the frontend is expected to get a new JWT that is then used, and verified by the microservice.

Secret key authentication for REST calls between services

1. Caller calls service
2. Service checks “service-secret-key” cookie, seeing if it matches any in a list of secret keys.
3. If yes, it returns a 200 OK, else a 401 unauthorized.

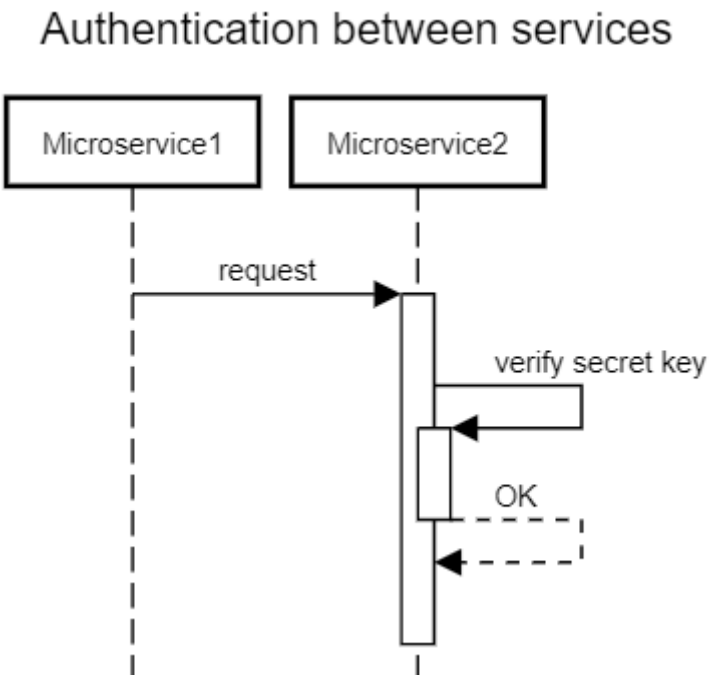


Figure 4: Sequence diagram illustrating how communication between microservices work.

4.4. Break down of microservices and their responsibilities

Service	Responsibilities	Corresponding FRs
Implemented in house		
Frontend	User interface for users to trigger different workflows. Specifically: Signup, login, find a match, enter a room/program, leave a room, see history of rooms, modify user settings, join the most recent room.	FR-ACC-1, FR-ACC-2, FR-ACC-3, FR-ACC-4, FR-ACC-5, FR-ACC-6, FR-ACC-7, FR-MAT-1, FR-MAT-2, FR-MAT-3 FR-COM-1, FR-COM-2 FR-PRO-4 FR-GEN-2
User service	Handles all things user related. Signup, login, JWT authentication (for frontend accessing REST APIs), JWT authentication (routed from other backend	FR-ACC-1, FR-ACC-2, FR-ACC-3, FR-ACC-4, FR-ACC-5,

	microservices being called by the frontend), changing user settings, user deletion, logout. Communication channel via REST API calls.	FR-ACC-6, FR-ACC-7, FR-GEN-2 FR-GEN-3
Question service	Handles all things related to questions. Get questions by difficulty. Communication channel via REST API calls.	FR-MAT-3
Matching service	Handles matching of two users based on a queue for each difficulty. Communication channel via REST API calls and web sockets.	FR-MAT-1 FR-MAT-2 FR-MAT-3
Collaboration service	Handles CRUD operations on collaboration rooms and real-time updating of room states for persistence reasons. Communication channel via REST API calls and websocket connections. Includes y-WebRTC and DailyVideo.	FR-COM-1, FR-COM-2 FR-PRO-1, FR-PRO-2, FR-PRO-3 FR-PRO-4 FR-PRO-5 FR-PRO-6 FR-PRO-7 FR-GEN-1 FR-GEN-2 FR-GEN-3 FR-GEN-4
External services		
MongoDB	NoSQL database shared by all microservices. Handles storage, with a unique database for each microservice.	
y-WebRTC	Open source WebRTC provider that handles peer-to-peer document updates.	FR-PRO-1, FR-PRO-2, FR-PRO-3 FR-PRO-6 FR-PRO-7
Daily Video	Open source WebRTC video API provider that handles peer-to-peer video calls.	FR-COM-1, FR-COM-2

5.Frontend

We designed our frontend/user interface to serve the needs of a specific target audience: computer science students looking to prepare for technical interviews by collaborating on practice technical questions with peers. We created a user persona named “Bob” to represent our intended users and imagined a user journey to contextualize our UI design.

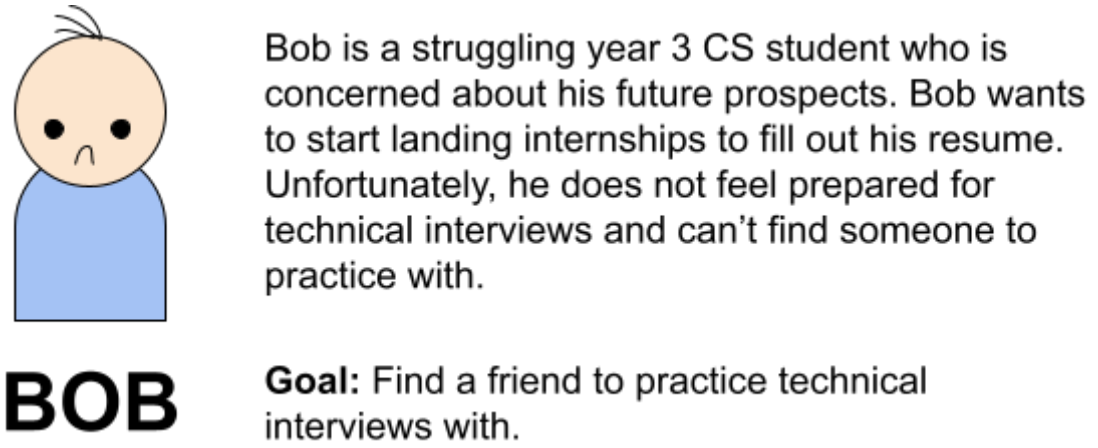


Figure 5a: Bob.

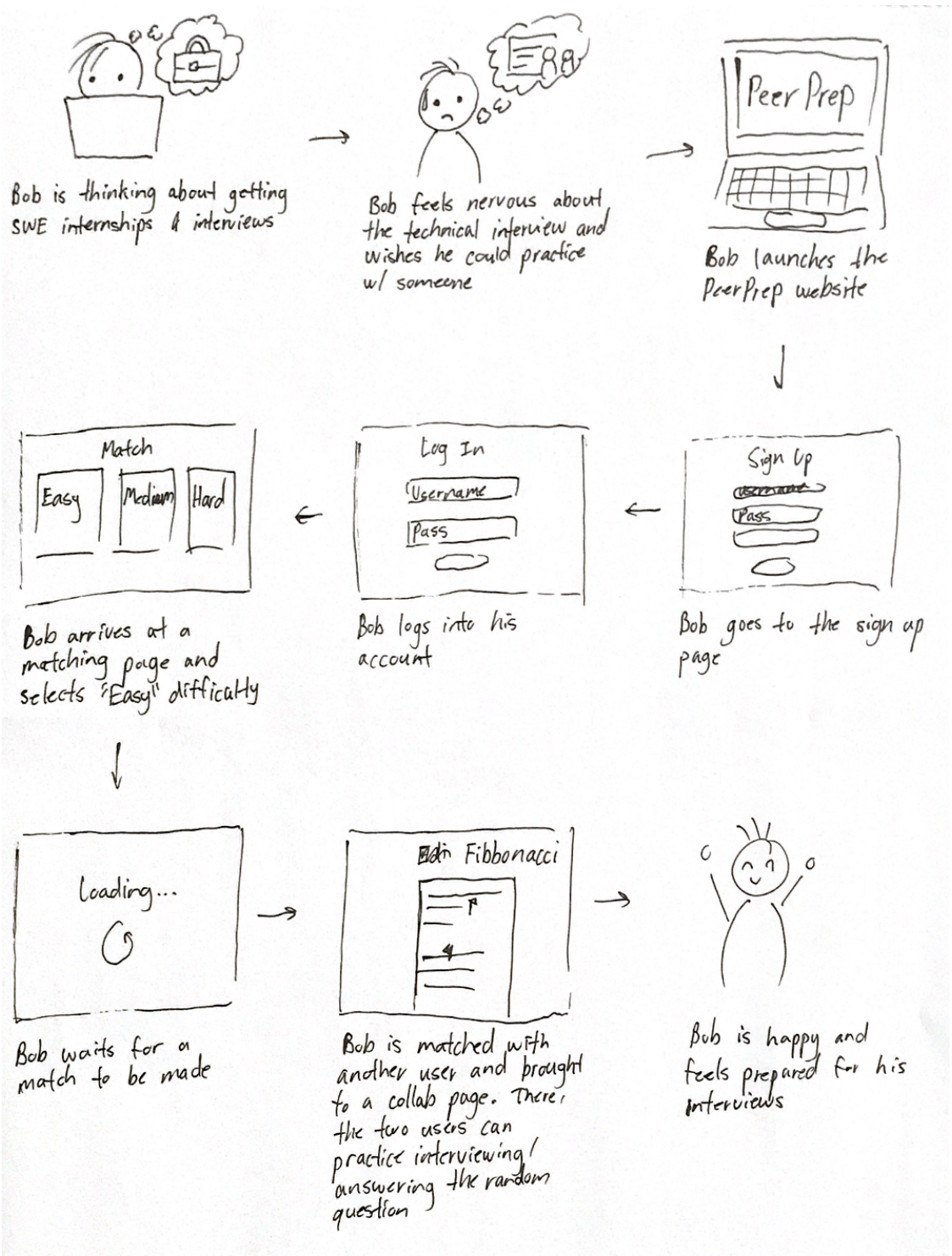


Figure 5b: An illustration of the main path to value of the typical user and how we expect them to use the application.

5.1. Features, FRs and NFRs

FR	Description	Design Element
FR-ACC-1	Users can create an account on this page	Account creation page
FR-ACC-2	Validation rejects invalid and existing usernames	Account creation page
FR-ACC-3	Validation rejects invalid passwords	Account creation page
FR-ACC-4	User can login from this page	Login page
FR-ACC-5	Front end stores the JWT in browser storage for future authentication and authorization	Browser storage
FR-ACC-6	User can change their username and password on this page	Account setting page
FR-ACC-7	User can logout, which clears the JWT from the browser storage	Logout button
FR-MAT-1	Front end redirects from waiting from match page to collaboration page room with the partner	Matching page
FR-MAT-2	Matching page cancel button	Matching page
FR-MAT-3	Matching page offers buttons to select difficulty	Matching page
FR-COM-1	Collaboration page has provides area for text communication	Collaboration page
FR-COM-2	Collaboration page has provides area for video and voice communication	Collaboration page
FR-PRO-4	Users can view previous rooms	History Page
FR-GEN-2	Matching page offers a popup to rejoin last session	Matching Page

5.2. Key design choices

The main workflows and user-paths considered in our application are:

1. User management - create account/login/change credentials
2. Using the core service - finding a match, collaborating with someone
3. Reflections - looking at history, going back to previous rooms

5.2.1. Tech stack

We choose React as our main frontend UI library for its functional-style components, ease of styling and intuitive state management.

5.2.2. Authentication via JWT

With security as an important NFR, every connection to other services needs to be authenticated. With the frontend server as a proxy for the user, every user thus needs to have an authentication token to access backend services.

We use JWT's for that. See Authentication Framework for details of how the frontend fits into our authentication framework.

5.2.3. Global contexts to avoid prop drilling

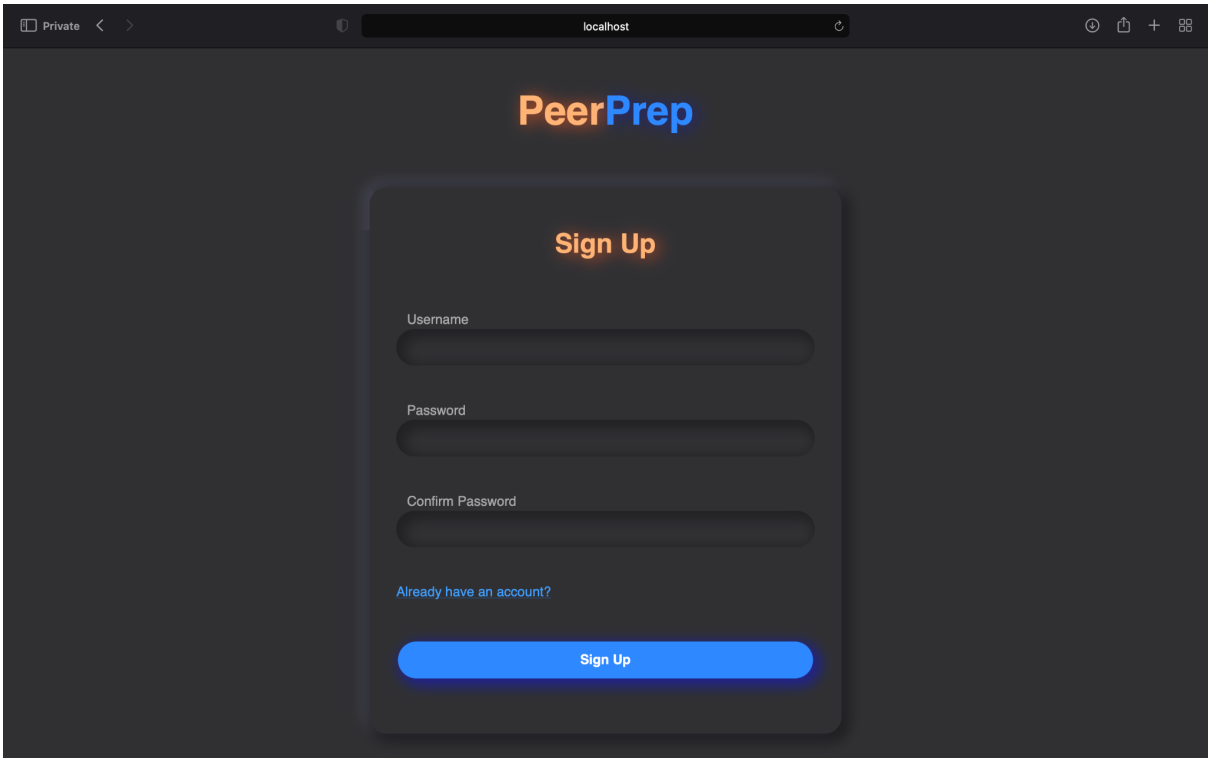
React's conceptual model is a tree of documents. In multiple pages, we need to maintain tokens (for authentication), a user state (for authentication or stateful displays e.g. username).

Prop drilling is a problem that arises as a result. For that reason, we actively use React Contexts, which functionally serve as a global state in the conceptual model.

5.3. The main workflow and how it works

In this section, we document the pages, how they work, and key design choices.

5.3.1.Sign up (/signup) and login (/login)



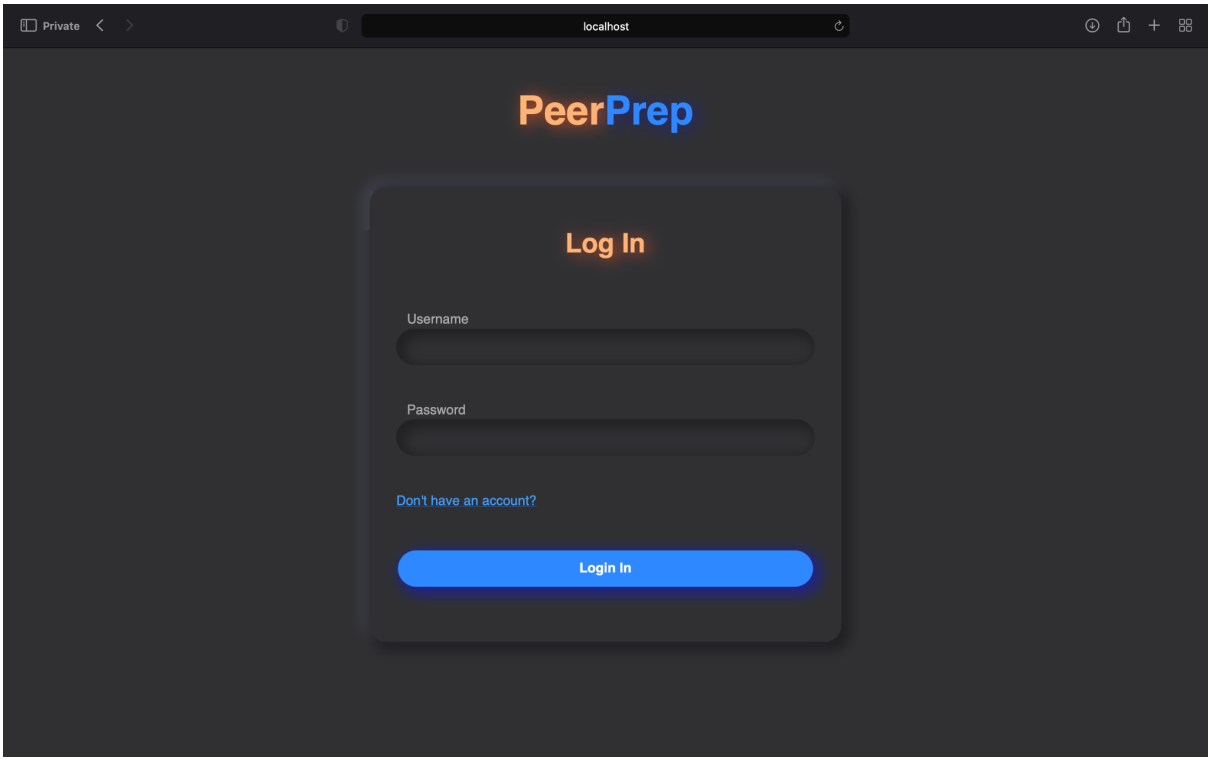


Figure 6: How the signup (top) and login (bottom) pages look. Upon successful signup, a user is directed to the login page to log in.

What it does: Allows users to sign up (creates a new user) and log in.

How it works:

1. User keys in credentials. Invalid details will prompt a user.
2. Valid credentials will call the User Service to create an account and return a 200 OK. A collision (e.g. existing username) will throw an error (e.g. 409)
3. On success, the user will be redirected to the login page.
4. Upon submission of valid user details, the login page sends the data to user service via API call.
5. This API call sets a http-only cookie containing a JWT token and returns a response containing the logged in user. The user is then stored in a global context.
6. The user is now logged in and redirected to the home page.

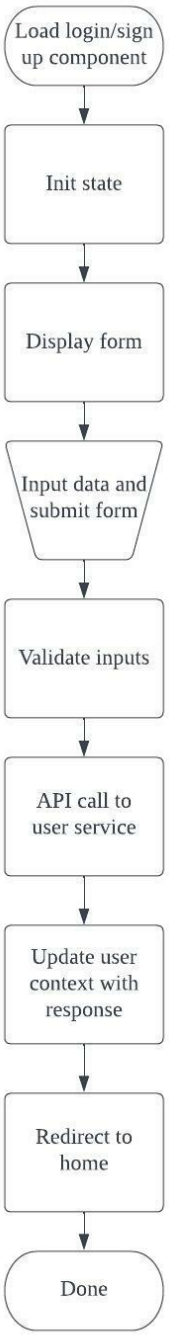
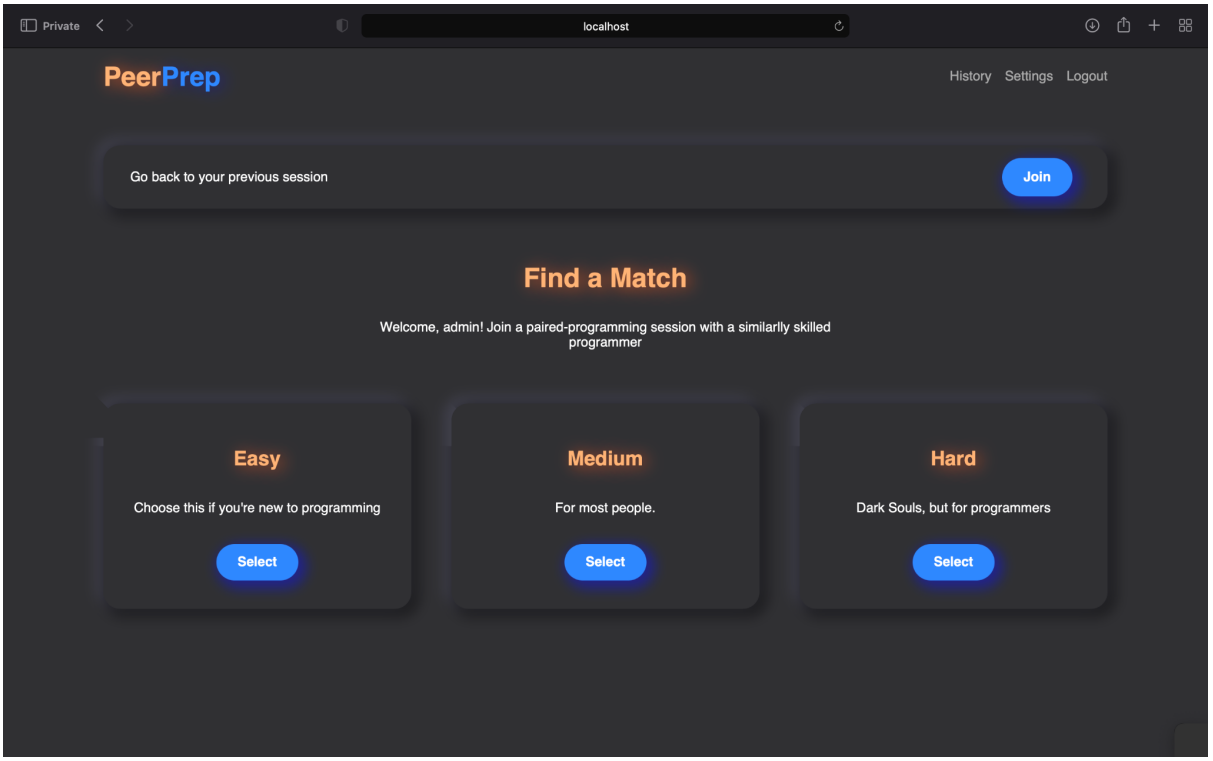


Figure 7: An activity diagram illustrating how a user navigates through the frontend and how the frontend server interacts with the other services to login/signup.

5.3.2.The home/matching page (/home)

The matching page is both the home page and will be the matching page. This is to reduce the length of the path to value (consider the alternative, where the user instead goes to a home, clicks to a matching page, then clicks to find a match).



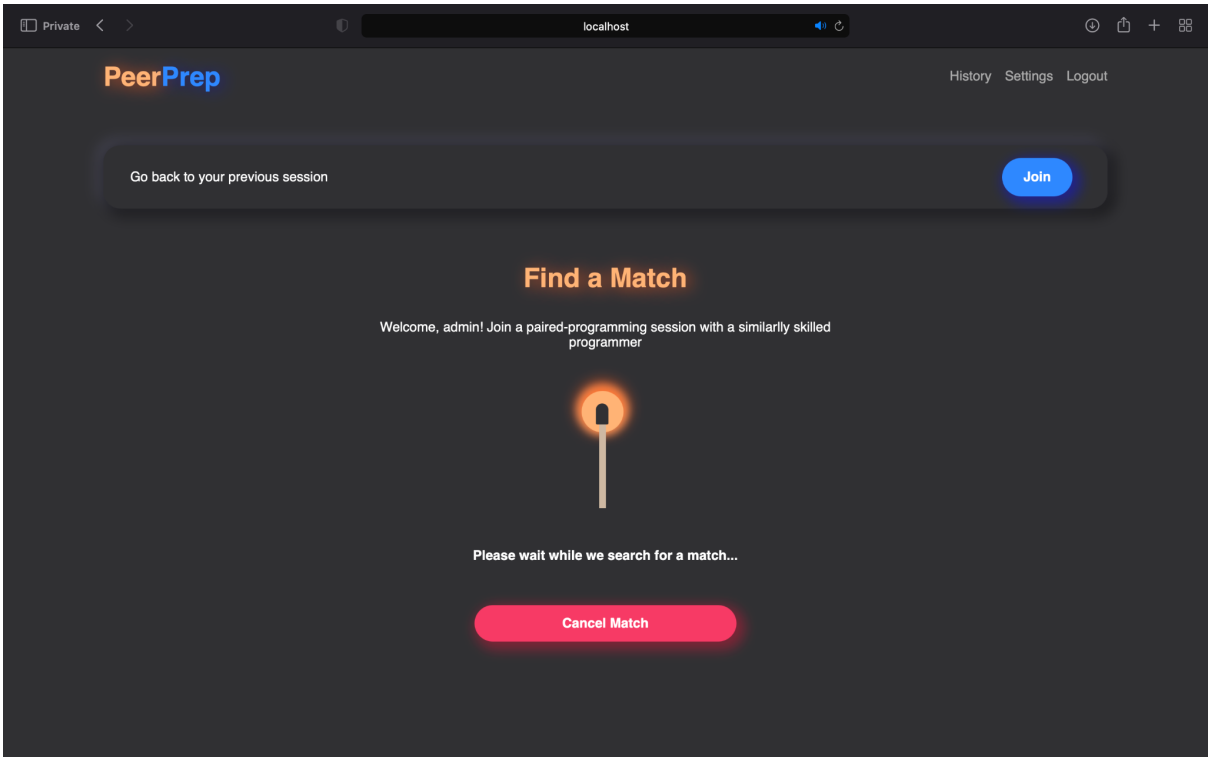


Figure 8: (Top) The home page (Bottom) The page while a match is being found. The home page doubles as the matching page to reduce the length of the path to value, and is intended to be intuitive and easy to use. Observe that we allow users two ways to enter a room - find a match, or go to the previous (most recent) session. On clicking to find a match, we will see the bottom page.

What it does: Allows the user to rejoin an active room or create a new match.

How it works:

1. User enters the page.
    a. Under the hood, we make an API call to the User Service to get a JWT token in preparation for authenticating the socket connection needed for interfacing with the Matching service. This step is necessary to authenticate a socket connection since our http-only cookie set in logging in cannot be passed over socket connections. For more details on authentication in this way, see Authentication Framework.
    b. We also make an API call to the collaboration service to get a history of rooms. The access key to the most recent room is retrieved (if it exists), to allow a user to join his most recent room.
2. A user clicks to find a match. He waits for 60s to find a match. While waiting, he can cancel the search or join his previous room.
    a. When a user selects a match difficulty level, frontend creates a new SocketIO socket, opens a connection, and emits a “match” event.
    b. While the frontend listens for a “matchSuccess” event, the frontend displays a loading component and a “Cancel Match” button. When the user selects the “Cancel Match” button, the socket emits a disconnect event and displays the initial difficulty level selection.
    c. If the frontend receives a “matchSuccess” event, the user is successfully matched with another user. Both users are redirected to the collaboration page identified by the unique id of the created room in the url.
3. (Optional) A user clicks to join his previous room. With the unique ID from 1b, the user is redirected to the collaboration page specified by the room ID.

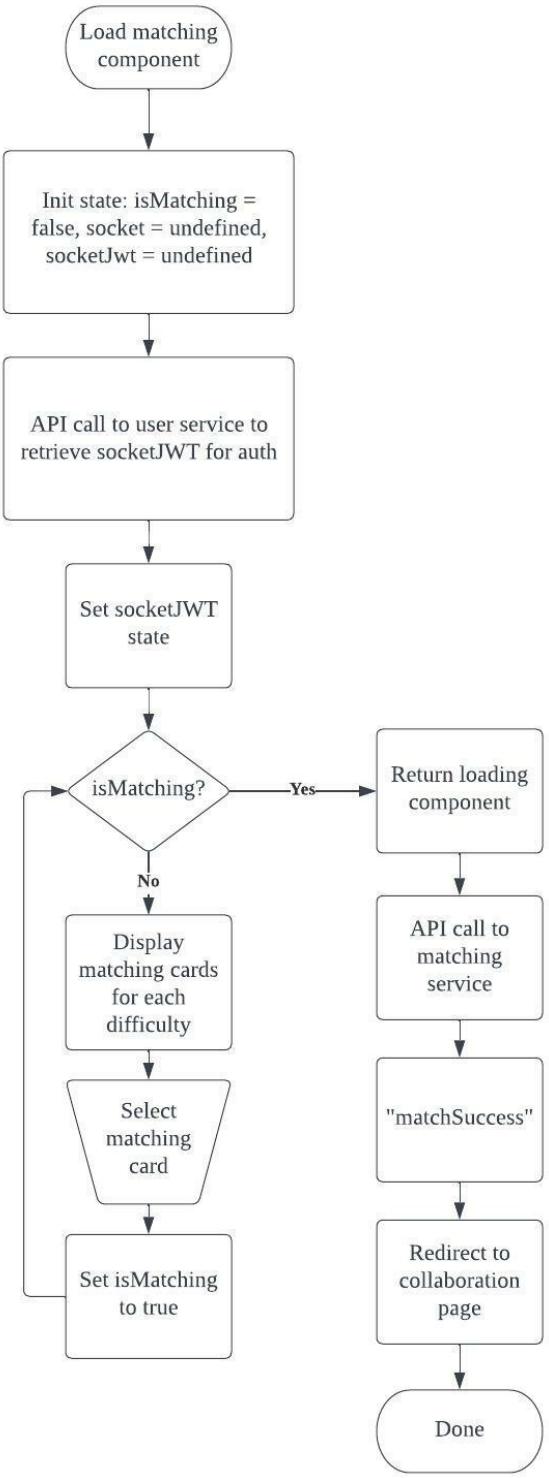


Figure 9: Activity diagram illustrating how the matching page works to match and redirect the user to a given room. On entering the page, a state is initialized with an uninitialized websocket and JWT. The JWT (for authenticating the websocket) is initialized by an API call to the user service. When the user looks for a match, a websocket connection to the Matching Service is opened, and the frontend waits for a "matchSuccess" event, upon which the user is redirected to the specified room.

5.3.3.The collaboration room page (/room?roomId={roomId})

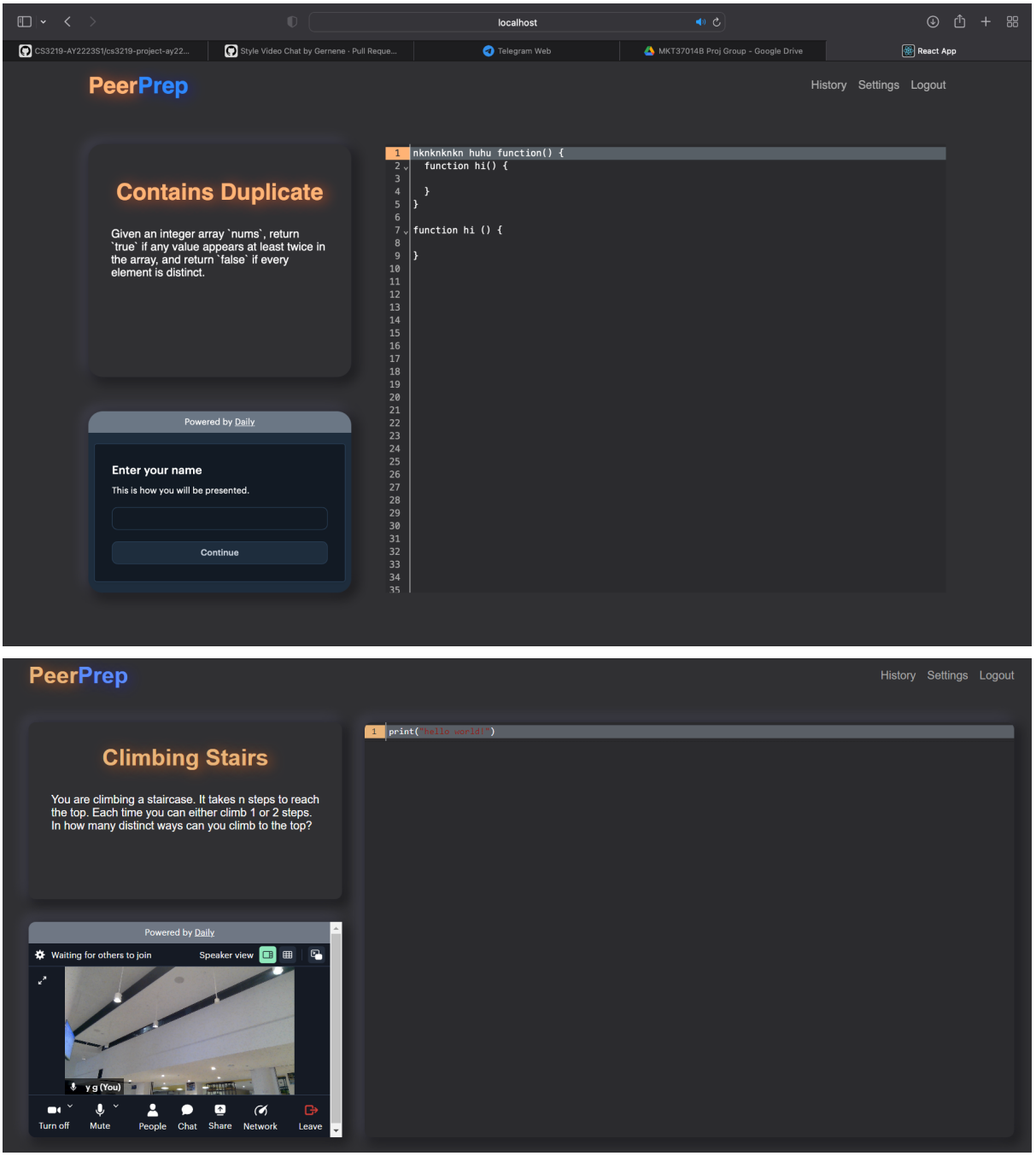


Figure 10: The collaboration room. Top left is the question card, bottom left is the video chat/text chat, and top right is the collaborative IDE. The top and bottom images are at different resolutions.

What it does: Allows two users to collaborate on an interview question using a collaborative editor and video chat

How it works:

1. The URL to the room contains the roomId as a query parameter.
    a. On initialization, the Collaboration Service is called to retrieve the room state (e.g. number of people in the room, code, etc.).

    b. An API call to the user service is also made to get another JWT for authenticating the websocket to be opened with the Collaboration Service.
2. To maintain persistence, a websocket connection is opened with the collaboration service.
3. Simultaneously, an API call to DailyVideo is made to open the connection for video and text chat with DailyVideo's WebRTC service.
4. On a click, highlight, or change to the text:
    a. The text from both users, cursor positions, highlights etc. are synced by WebRTC and then pushed back to both user's view DOMs.
    b. The update is tracked as a state. A debounced update (i.e. after 1s of inactivity) is pushed to the backend via the websocket connection for saving.

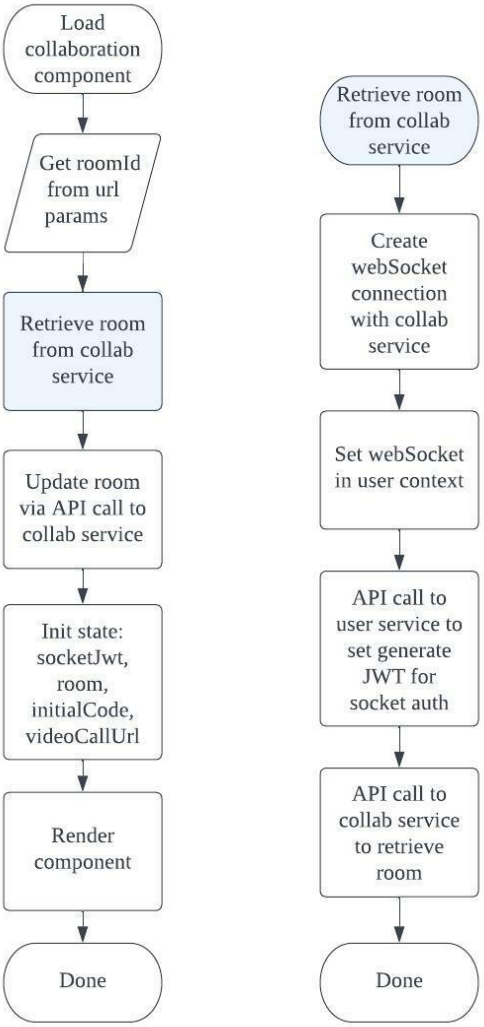


Figure 11: Activity diagram illustrating how the collaboration page is initialized and works to maintain a collaborative environment for the two users in the room.

5.3.4.The history page (/history)

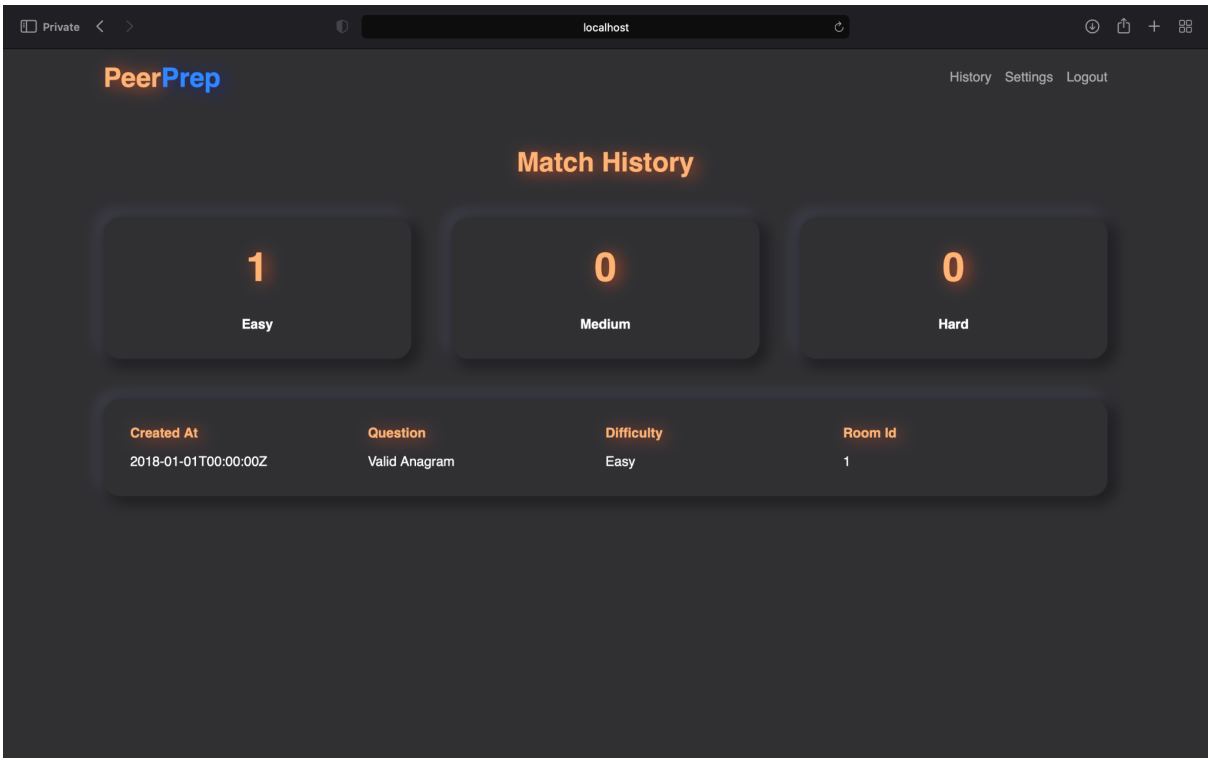


Figure 12: The history page.

What it does: Displays a user's previous rooms and stats

How it works:

1. User enters page
2. Send API request to collaboration service to retrieve a user's previous rooms.
3. Count the number of questions of each difficulty level.
4. Display question count stats.
5. Display table of all fetched rooms.

5.3.5.The settings page (/settings)

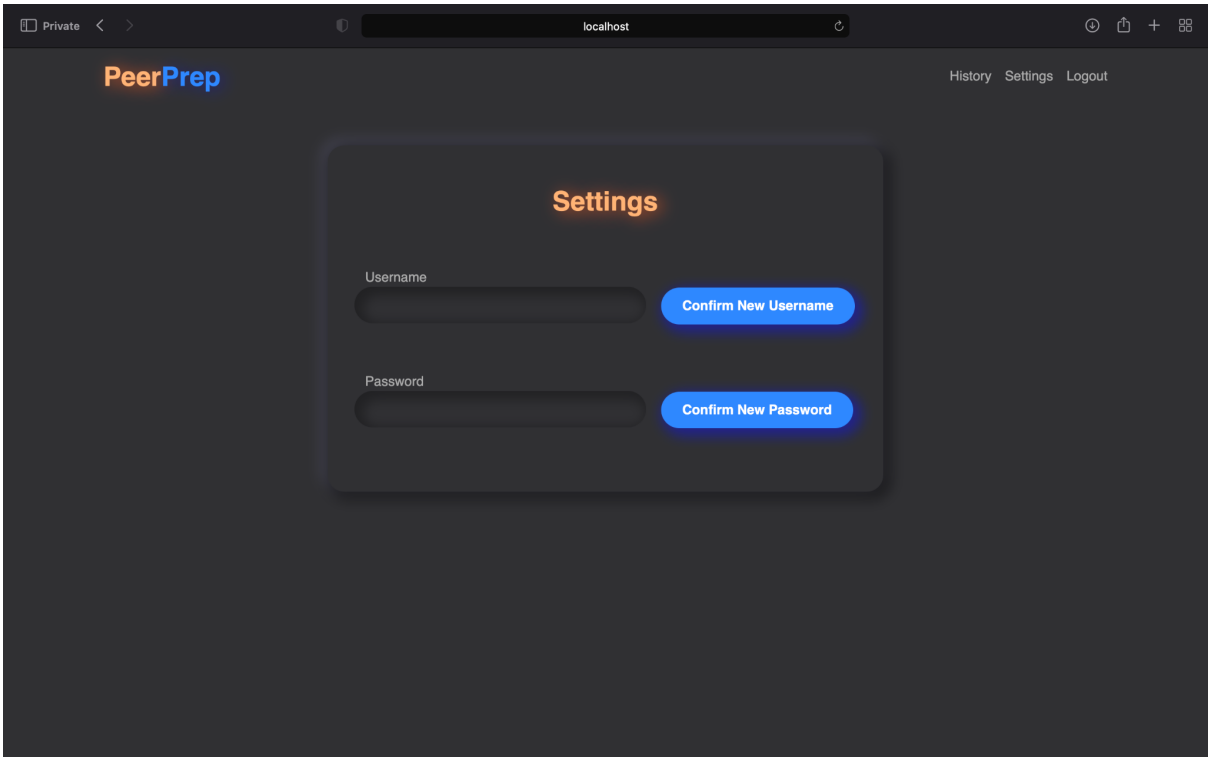


Figure 13: The settings page.

What it does: Allows user to change their username or password

How it works:

1. User enters page
2. User inputs new username or password
3. User submits form
4. Frontend performs basic validation of input
5. Send API POST request to user service to update user details

5.3.6.Development and deployment

Run docker from root directory, stop Front end container
Start up dev instance by npm i && npm start inside front end working directory
This lets you modify the front end and have it update in the normal create react app way

6.User service

6.1. Features, FRs and NFRs

FR	Description	Design Element
FR-ACC-1	Uses the received username and password to create a new account in mongoDB, returns an appropriate http code and payload depending on success.	Create user endpoint
FR-ACC-2	Throws an error if an invalid account is trying to be created	Backend validation in the handler methods
FR-ACC-3		
FR-ACC-4	Checks the received username and password against accounts in mongoDB, returns an appropriate http code and payload depending on success.	Login endpoint
FR-ACC-5	Checks the JWT token attached to the request for validity, returns an appropriate http code depending on success	Authentication endpoint
FR-ACC-6	Mutates the stored account credentials based on the provided information	Change username endpoint, change password endpoint
FR-ACC-7	Instructs the web browser to clear the login JWT	Logout endpoint

6.2. Architecture

Recall that the user service's main responsibilities are to handle everything user-related - that is, user CRUD operations and authentication (via JWT).

Because these operations rely on underlying database reads/writes, the user service is implemented with a layered architecture and the model view adapter patterns in mind.

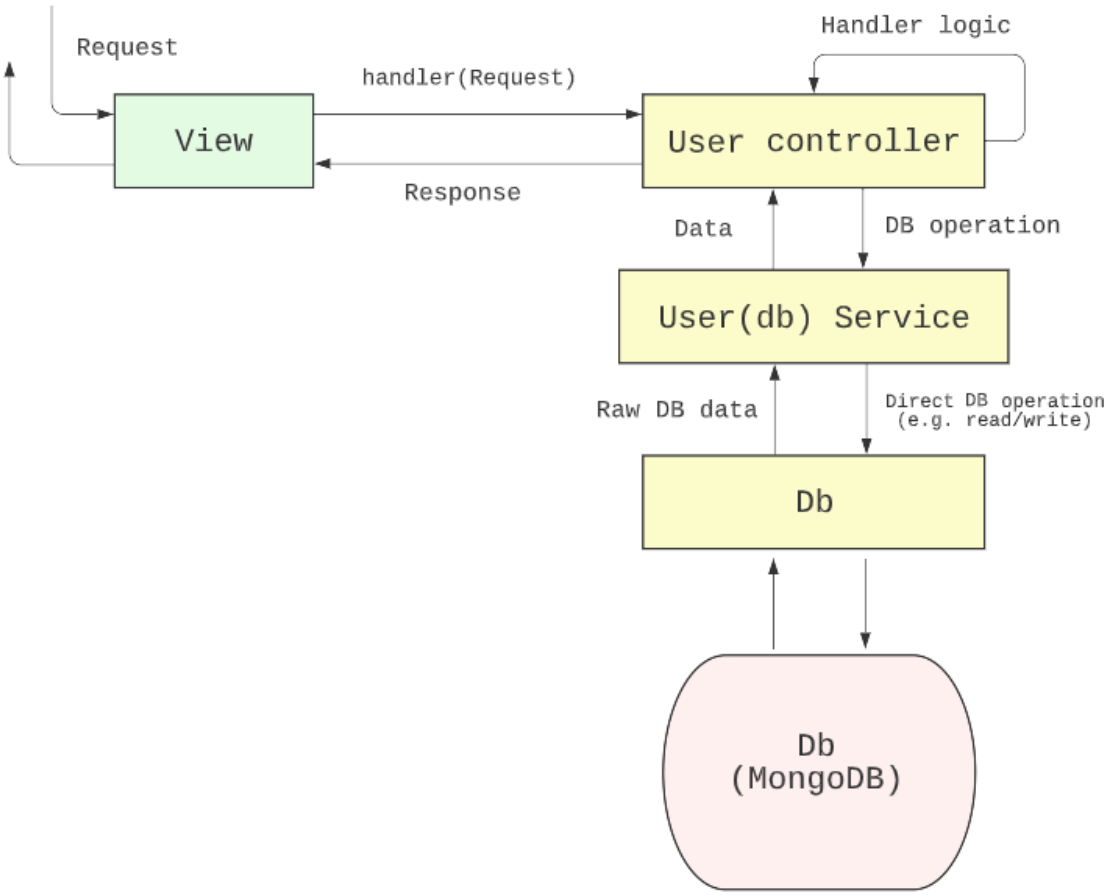


Figure 14: Architecture diagram for the user service. It is developed with the layered and MVA patterns in mind. The requests come in at the view layer, and are directed to the relevant user controller. As needed, the user controller calls services provided by the UserDbService, which interfaces with the Db (a database wrapper) component. These layers of abstraction are introduced to reduce coupling for better development in parallel.

Component	Responsibility
View	Exposes REST APIs to other microservices and frontend.
User controller	The controller, called by the view when a given request comes in. Serves as the middleman between the view (request layer) and underlying MongoDB database, preprocessing/post processing data as needed.
User service	Service class that handles low-level operations like directly interfacing with the DB. Is injected into the user controller for testability.
Model (MongoDB)	Wrapper class around the MongoDB database (via Mongoose library). Wrapper class allows us to separate the actual DB being used with the required operations.

6.3. API

Request example	Response example	Description
POST /signup		

{
"username": "3",
"password": "ok"
}
	200
	{
	"username": "3",
	"password": "$2b$10$zzlQzTUZKmvPuIejOTr0KesW8bSvMPDxY71R5jilE.OZue9IRhVnW",
	"id": "10300d21-5b6b-41c5-bf8f-ab899dabcafe"
	}
	
	409
	{
	"data": {
	"message": "User already exists."
	}
	}
		Handles a POST request from a request to sign up. Returns a 200 OK with user data upon success, or a response with a message field if an error occurs.
		
		Returns 409 if the desired username is in use by another account.

POST /login		

{
"username":"3",
"password":"ok"
}
	200
	{
	"username": "3",
	"password": "$2b$10$zzlQzTUZKmvPuIejOTr0KesW8bSvMPDxY71R5jilE.OZue9IRhVnW",
	"id": "10300d21-5b6b-41c5-bf8f-ab899dabcafe"
	}
	//JWT appended
	
	401
	{
	"data": {
		Logs in user using username and password
		Returns 200 with JWT if successful.
		
		Returns 401 with an error message if no matching username or password combination is found.

	"message": "Incorrect username/password!" } }	
PUT /changeUsername		

{
"username":"4",
"password":""
}
	200
	{
	"username": "3",
	"password": "$2b$10$zzlQzTUZKmvPuIejOTr0KesW8bSvMPDxY71R5jilE.OZue9IRhVnW",
	"id": "10300d21-5b6b-41c5-bf8f-ab899dabcafe"
	}
	
	409
	{
	"data": {
	"message": "User already exists"
	}
	}
	
	401
	{
	"data": {
	"message": "Invalid JWT"
	}
	}
		Changes the username used for identification and login for the account.
		
		Returns 409 if the desired username is in use by another account.

PUT /changePassword		

{
"username":"",
"password":"ok"
}
	200
	{
	"username": "4",
	"password": "$2b$10$zzlQzTUZKmvPuIejOTr0KesW8bSvMPDxY71R5jilE.OZue9IRhVnW",
		Changes the password used for login for the account.

	"id": "10300d21-5b6b-41c5-bf8f-ab899dabcafe"
	}
	
	401
	{
	"data": {
	"message": "Invalid JWT"
	}
	}	

POST /logout		

{}	200
	{
	"message": "Logged out."
	}
	//JWT set to “”
		Returns 200, and overwrites the JWT to an empty string to log the user out.

GET / auth		

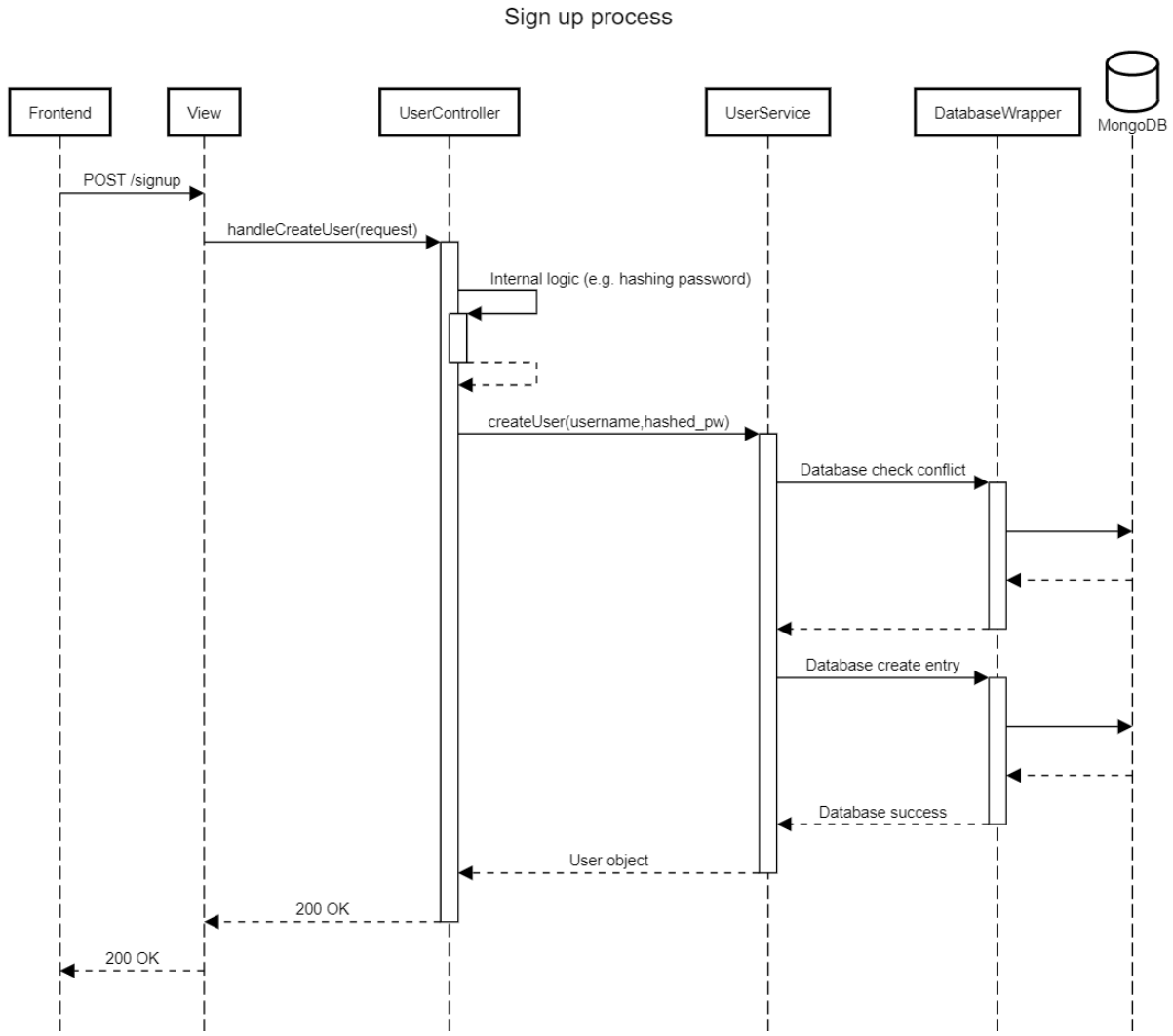
{}	200
	{
	"username": "3",
	"password": "$2b$10$zzlQzTUZKmvPuIejOTr0KesW8bSvMPDxY71R5jilE.OZue9IRhVnW",
	"id": "10300d21-5b6b-41c5-bf8f-ab899dabcafe",
	"iat": 1667181947,
	"iss": "user-service",
	"exp": 1667225147
	}
	
	401
	{
	"data": {
	"message": "Invalid JWT"
		Validates the attached JWT sent from client and returns 200 if it is valid.

	} }	

6.4. Examples (sequence diagrams)

Here, we document two representative examples using sequence diagrams to illustrate how components interact during operations.

Sign up and login



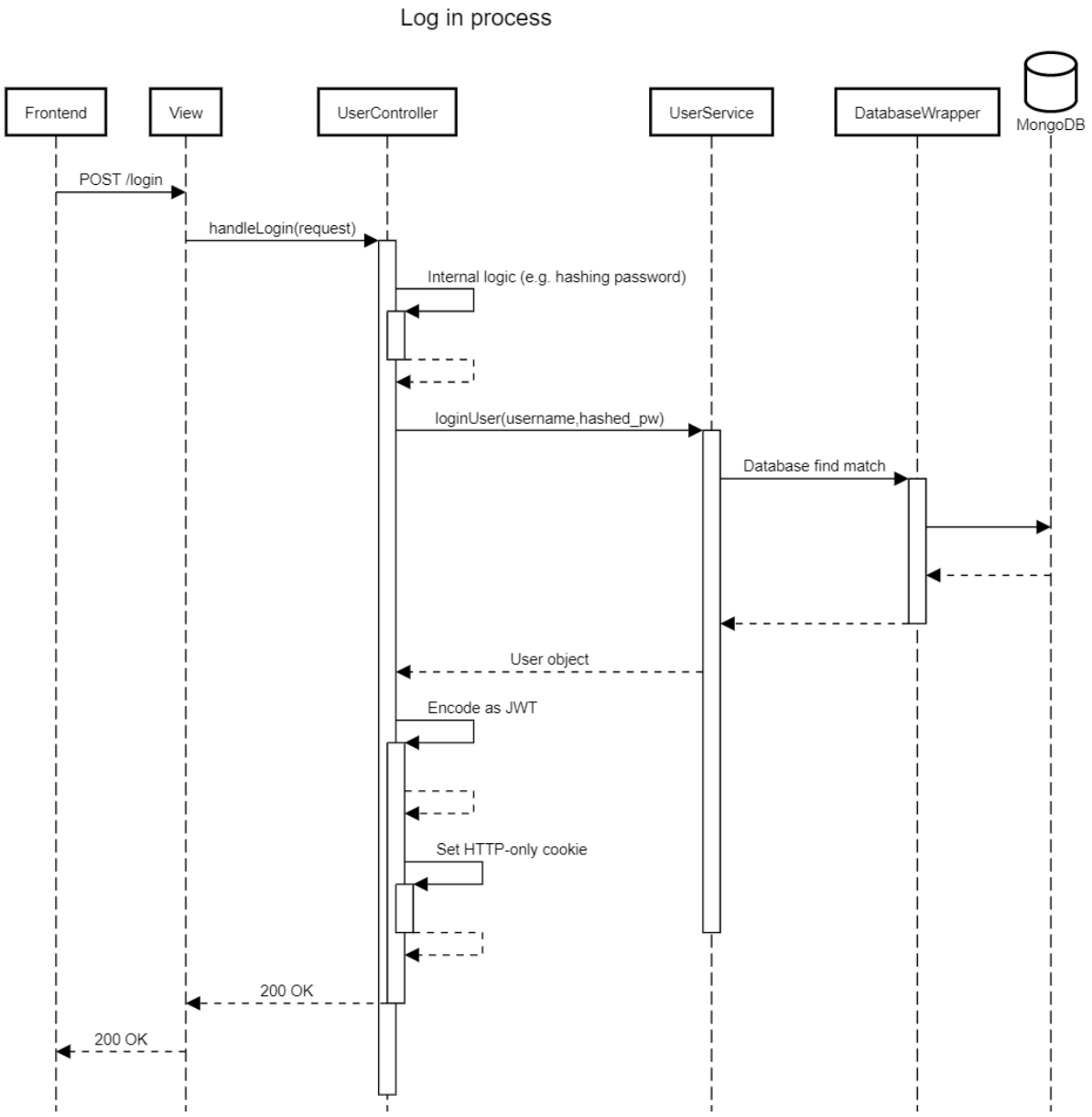
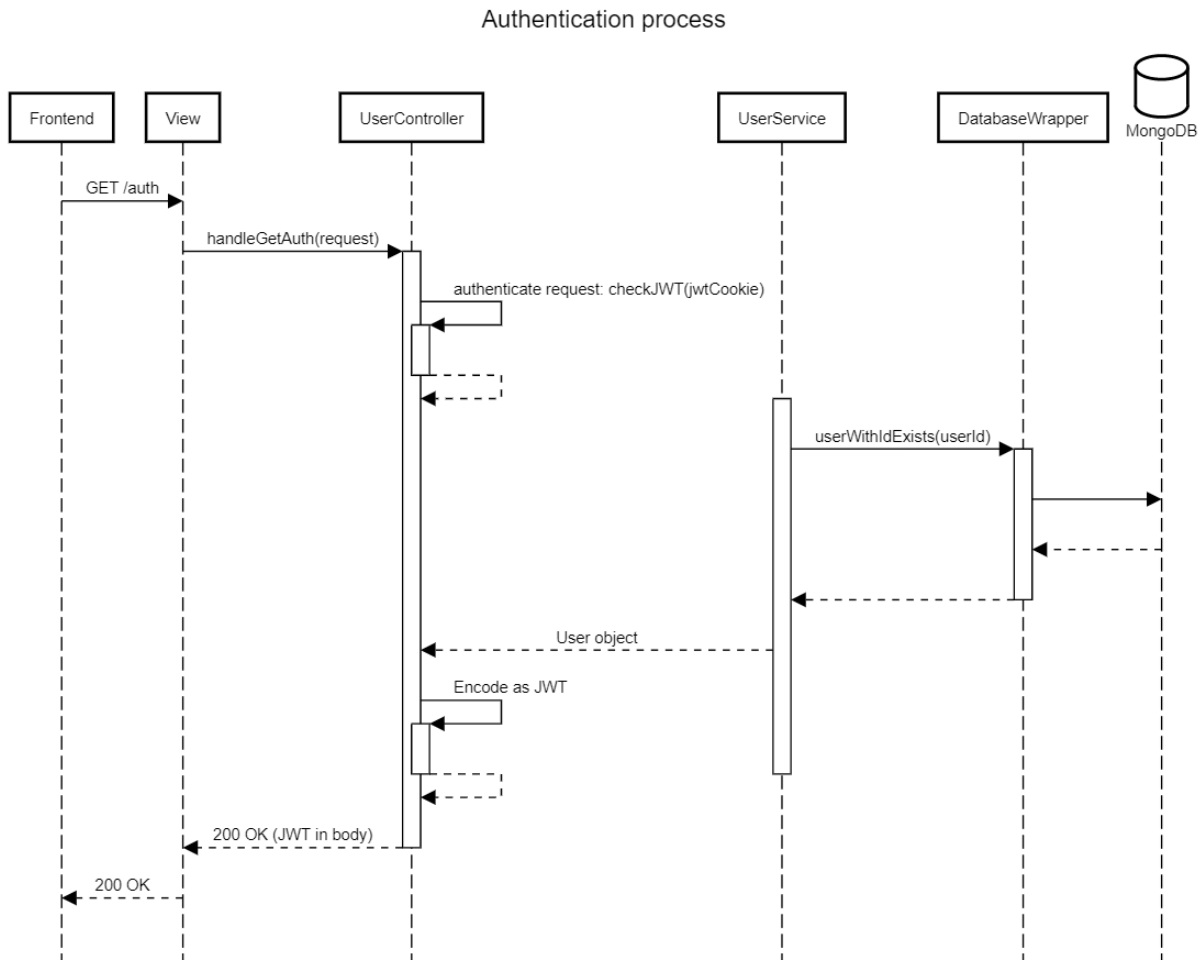


Figure 15: (Top) The signup process. (Bottom) The login process and how we set the JWT as a HTTP-only cookie for authentication.

1. Frontend makes a POST request to sign up at the /signup endpoint.
2. The request is forwarded to the user-controller handler, which hashes and salts the password.
3. The username and password are then passed to the user-service, which checks that the user does not exist, validates the credentials, writes to DB, then returns the user.
4. A 200 OK is returned to the frontend.
5. Frontend makes a POST request to login.
6. The request is forwarded to the user-controller handler, which hashes and salts the password, then passes the credentials to the service to check if it exists in the DB.
7. The service checks whether the user exists, and if it does, returns the user as a **`User`** data class.

8. The user-controller encodes the `User` object as a JWT and adds it as a HTTP-only cookie, before returning a 200 OK.

Authentication by other microservice for websocket



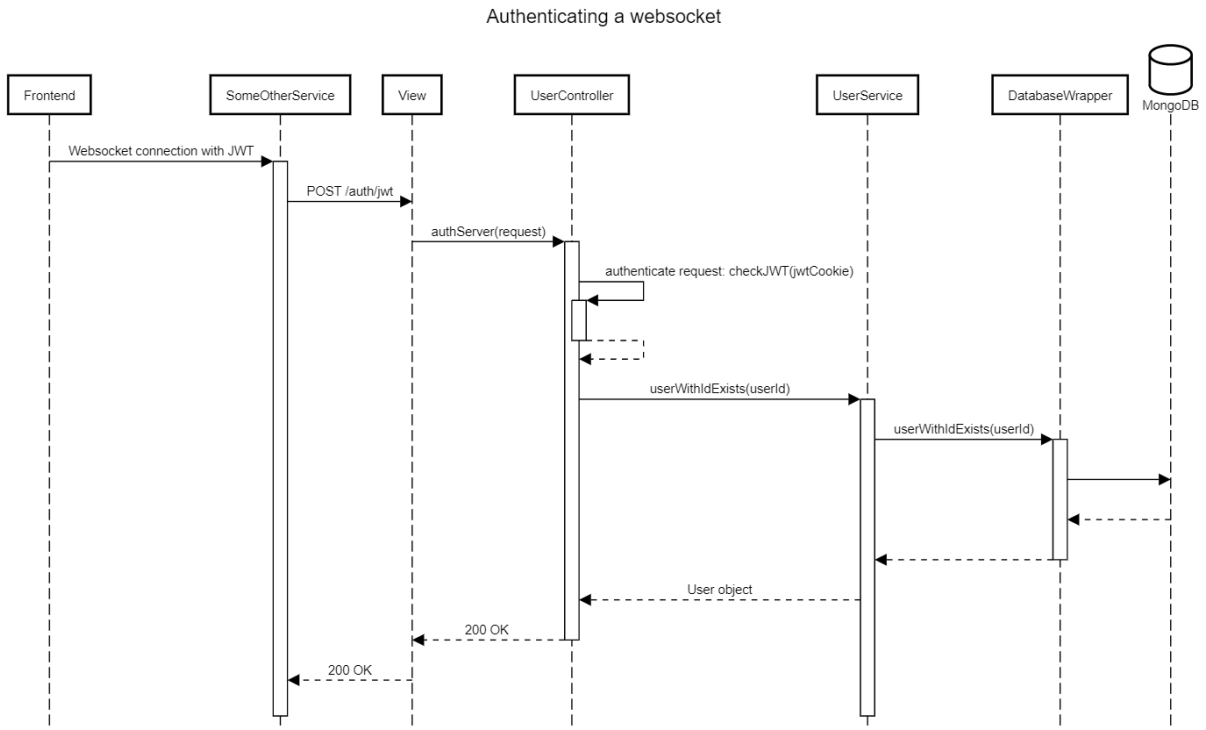


Figure 16: (Top) How the frontend gets a new JWT for a new websocket connection. (Bottom) How the frontend uses the JWT to authenticate a websocket connection with another service - the other service comes back to the user service to check that the JWT is valid.

6.5. Development and deployment

Tech stack
Express.js, Typescript, supertest/jest, MongoDB (via mongoose)

Running the service, adding tests

The service can be run locally directly through npm, where it will require a local mongodb instance to connect to as defined by the .env file. Alternatively it can be launched through docker with the other services.

See the README.md for specific guide.

7.Question service

7.1. Features, FRs and NFRs

FR	Description	Design Element
FR-MAT-3	randomly selects a question of the given difficulty for pair to work on	POST /difficulty endpoint
FR-GEN-3	retrieves a specific question based on the question ID stored in history	GET /qid endpoint

7.2. Architecture

The question service is implemented as a simple wrapper around the database with basic CRUD operations, much like the user service. For that reason, it's organized in a very similar way.

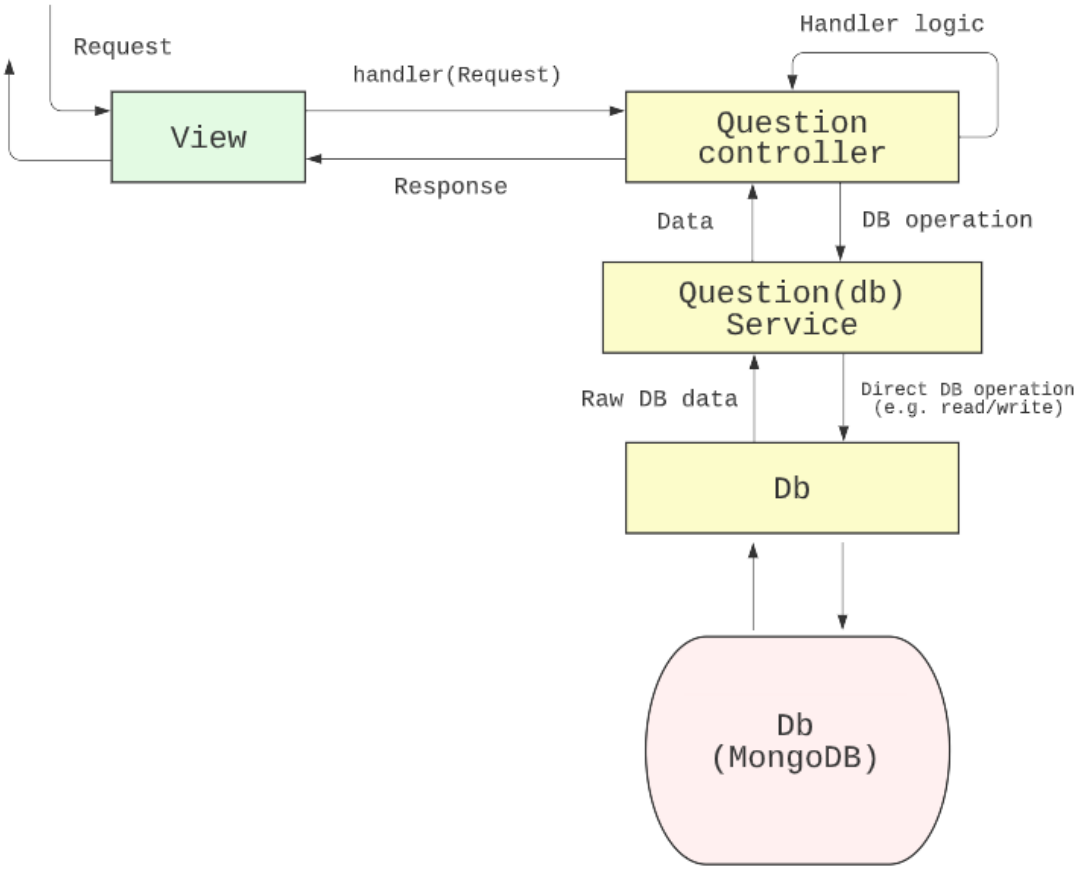


Figure 17: Architecture diagram for the user service. It is developed with the layered and MVA patterns in mind. The requests come in at the view layer, and are directed to the relevant handlers. As needed, the controller calls services provided by the QuestionService, which interfaces with the Db (a database wrapper) component. These layers of abstraction are introduced to reduce coupling for better development in parallel.

7.3. API

Request example	Response example	Description
POST /difficulty		

<pre>{
  difficulty: 0
}</pre>
	<pre>{
	    "qid": "1",
	    "title": "Contains Duplicate",
	    "description": "Given an integer array `nums`, return `true` if any value appears at least twice in the array, and return `false` if every element is distinct.",
	    "difficulty": 0,
	    "topic": 0
	}</pre>
		Handles a POST request from a request to retrieve a random question by question difficulty. Returns a 200 OK with a question data upon success, or a response with a message field if an error occurs.

GET /qid		

<pre>/qid?qid=2
 	
{}</pre>
	<pre>{
	    "qid": "2",
	    "title": "Valid Anagram",
	    "description": "Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`, and `false` otherwise.\nAn Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.",</pre>
		Returns 200 OK with the question data with the specified Question ID (qid), or a response with a message field if an error occurs.

	`"difficulty": 0,`
	`"topic": 0`
	`}`	

7.4. How it works

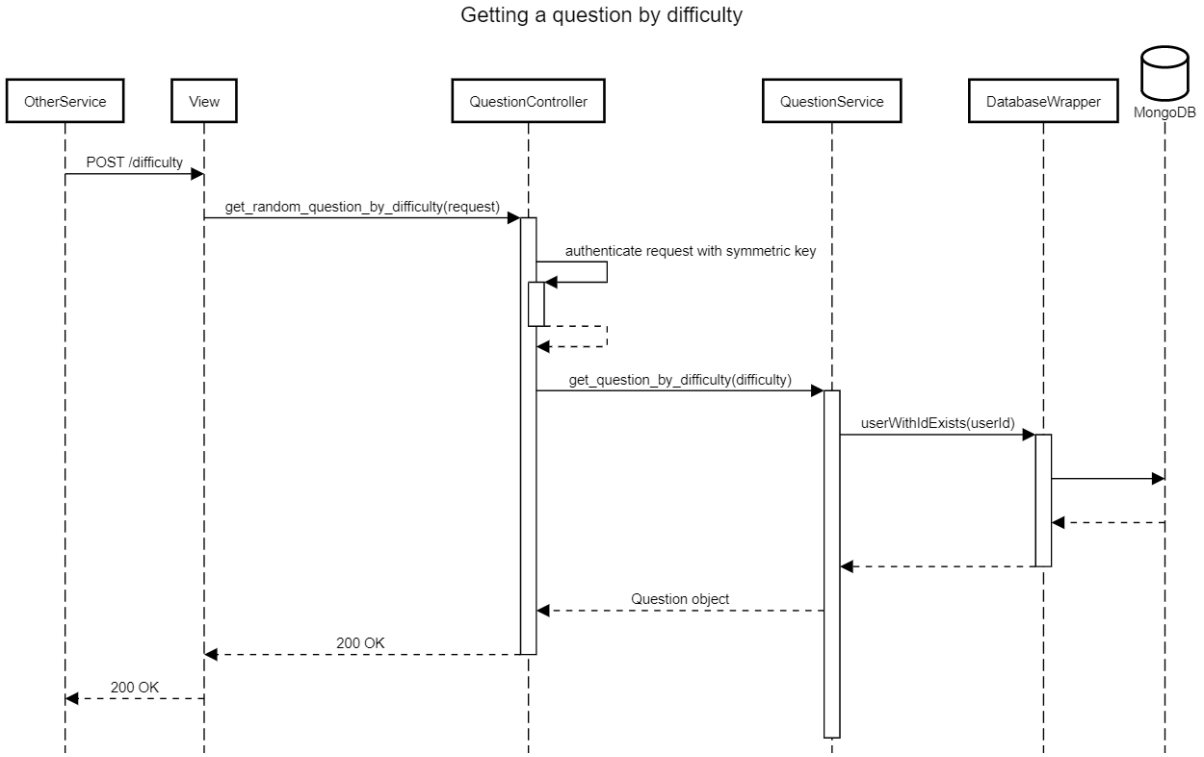


Figure 18: How the question service gets a random question.

7.5. Development and deployment

Tech stack
Express.js, Typescript, supertest/jest, MongoDB (via mongoose)

Development and deployment
See the README.md.

8.Matching service

8.1. Features, FRs and NFRs

FR	Description	Design Element
FR-MAT-1	Two logged in users who select the same difficulty should be paired and put in the same room	Pairs 2 pending matches and creates room via call to collaboration service
FR-MAT-2	The user matching service must allow users to cancel their search for matches when they are not yet allocated a match by the system.	Queue service handles/discards messages from disconnected sockets.
FR-MAT-3	Users should be able to select at least three difficulties - easy, medium, hard for the criteria in matching to another user.	Queue service provides separate queue for each difficulty level and sorts incoming messages accordingly

8.2. Architecture

8.2.1. High level organization

The matching service is designed with the broker and producer-consumer patterns in mind.

The matching service's key responsibility is to enqueue users from the frontend, match them and then create a room for them and letting the frontend web server know which room has been created. With this workflow, there are three key domains - socket management, queuing and room management.

With these in mind, we can organize the Matching Service into a SocketService, QueueService and RoomService.

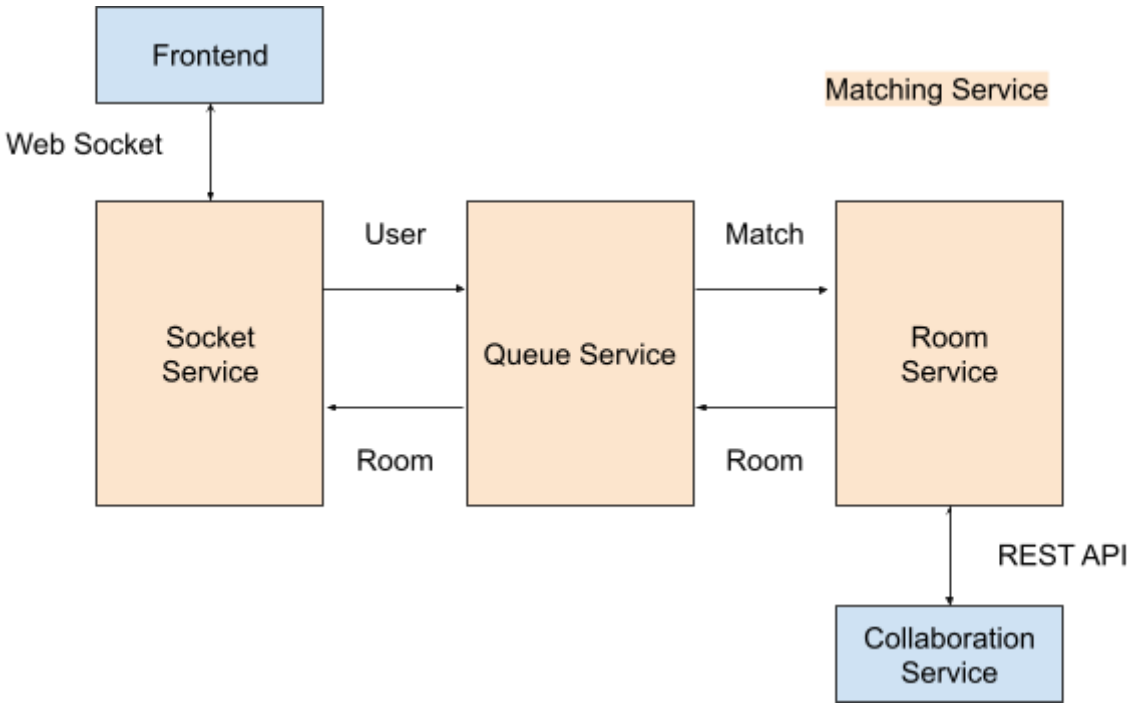


Figure 19: A simple architecture diagram of how the components within the Matching Service are organized by Single Responsibility.

Component	Responsibility
Socket service	Manages socket connection and send/receives socket events from the frontend.
Queue service	Initializes queues and handles queues for each difficulty level. Produce and consume pending matches (messages).
Room service	Handles interfacing with the Collaboration Service.

8.2.2. Designing the queue service

Now, let's consider the matching task conceptually in terms of actors and their producer consumer relationships.

- There are different types of users.
- *Between* components, the SocketService pushes users into the QueueService, and the QueueService, on successful match, pushes matches to the SocketService.
- *Within* the QueueService, on push of a new user, the QueueService must somehow organize the user by type, match them (if possible), and then push a match out.

With this mental model in mind, we know that we need a queue for each user type. We also need a broker to direct the users to the correct queue, and a consumer to match the users in these queues.

The QueueService handles both message brokering and consuming. On push from the SocketService, it directs a user to the correct queue. At the same time, if conditions are appropriate (there is a waiting user for a match), the consumer creates a match and makes a call to RoomService to create a room. QueueService then pushes the room to SocketService.

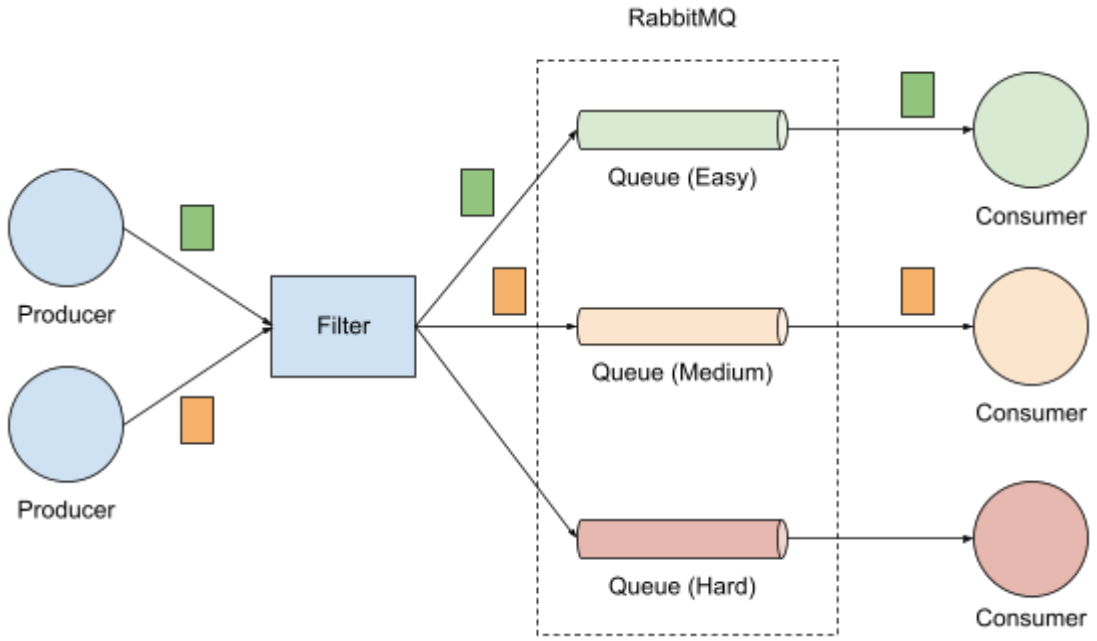
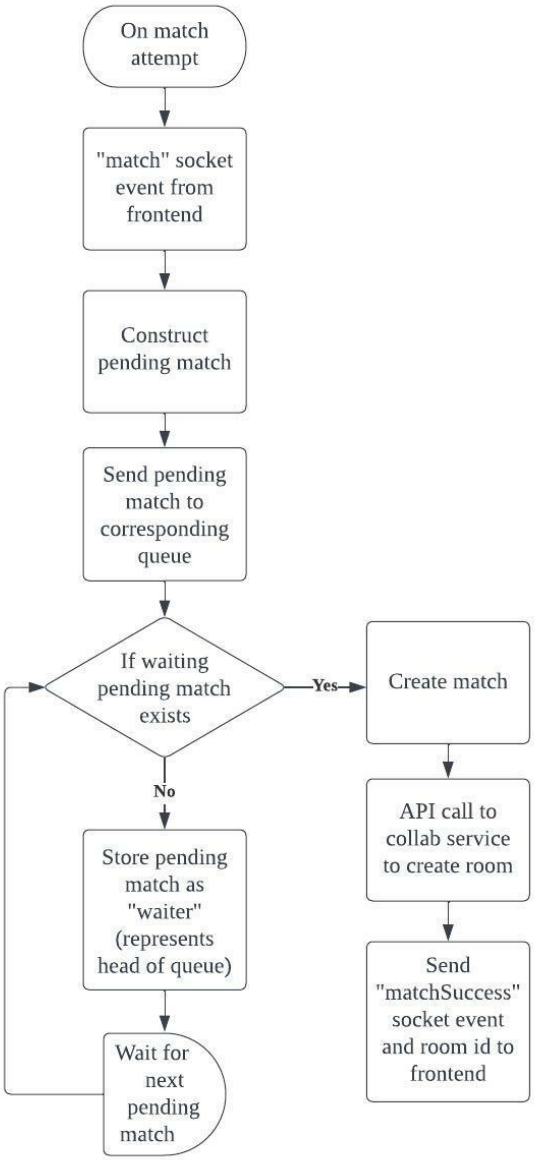
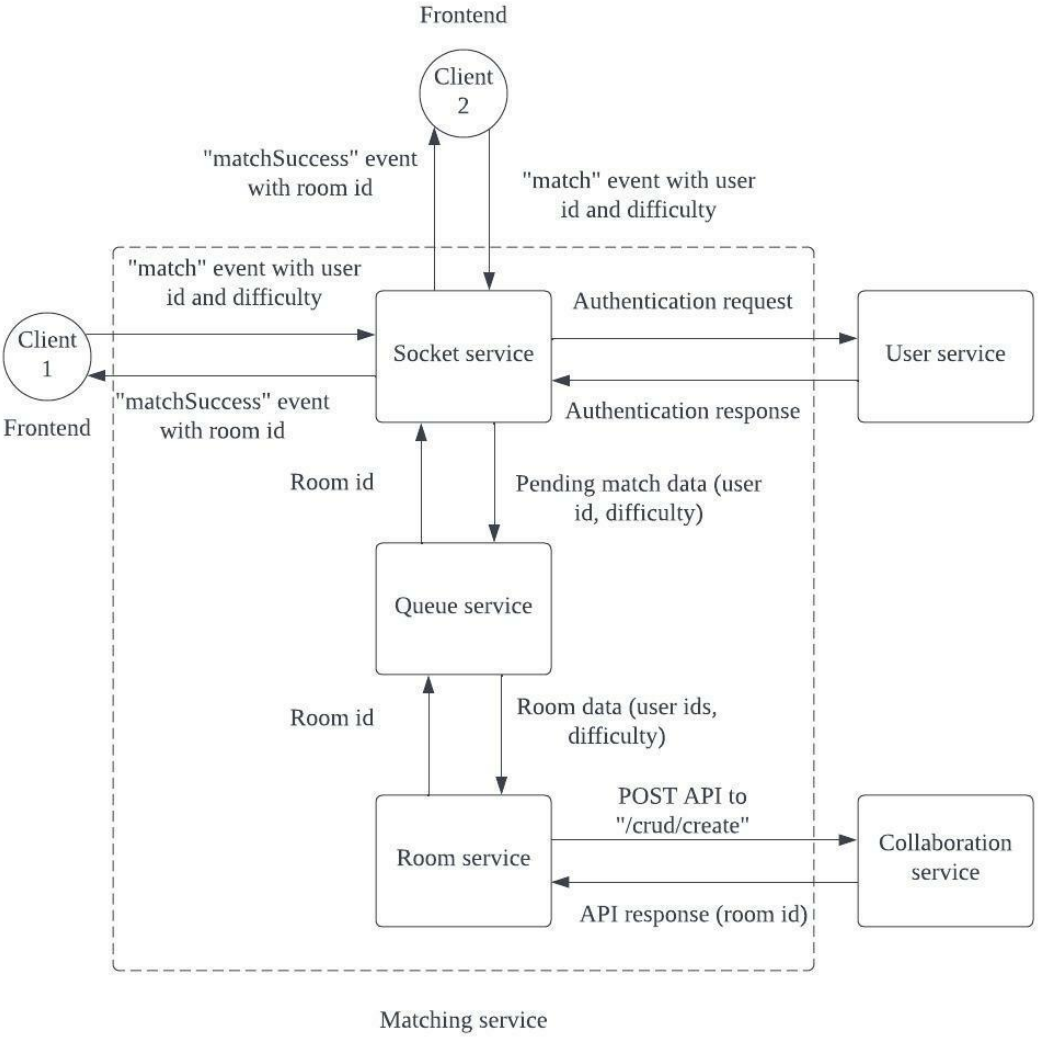


Figure 20: An illustration of how the QueueService wraps around RabbitMQ queues and serves as a broker-producer and consumer of these queues.

8.3. Putting them together

Together, therefore, we can illustrate how the Matching Service works with an activity and sequence diagram.


On match attempt
"match" socket event from frontend
Construct pending match
Send pending match to corresponding queue
If waiting pending match exists
Yes
Create match
No
Store pending match as "waiter" (represents head of queue)
Wait for next pending match
API call to collab service to create room
Send "matchSuccess" socket event and room id to frontend


Frontend
Client 2
"matchSuccess" event with room id
"match" event with user id and difficulty
"match" event with user id and difficulty
Client 1
Frontend
Socket service
Authentication request
User service
"matchSuccess" event with room id
Authentication response
Room id
Pending match data (user id, difficulty)
Queue service
Room id
Room data (user ids, difficulty)
POST API to "/crud/create"
Room service
Collaboration service
API response (room id)
Matching service

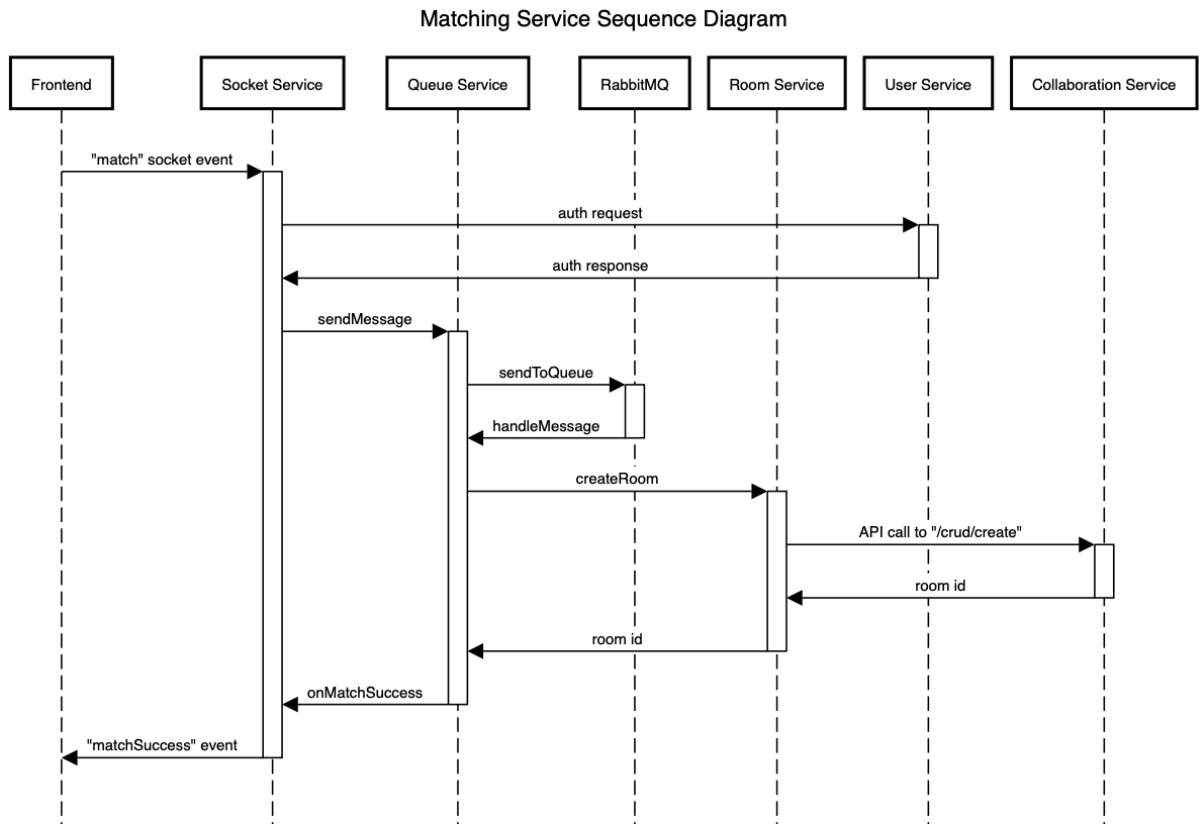


Figure 21: How the Matching Service is organized to match users. (Top) Activity diagram describing the logic. (MIddle) Activity diagram imposed onto the architectural components. (Bottom) Sequence diagram describing the same events.

1. *Clients connect via web sockets managed by the SocketService.*
2. *The SocketService pushes the data over to the QueueService*
3. *Queue Service brokers the messages and directs them to the correct queues (hosted by RabbitMQ) by difficulty level. These allow for independent queuing.*
4. *A consumer (in the QueueService) consumes the queued user, and when it forms a pair, pushes the pair to the RoomService.*
5. *The RoomService handles interfacing with the CollaborationService to create a room, before returning it to the SocketService, which notifies the frontend of the success.*

8.4. Handling disconnections

There are two main ways to handle disconnections.

Method	**Description**
Let dead users stay in queue	The idea is to let disconnected users stay in the queue. As the consumer consumes, we check whether they are alive to determine if they can be used for a match.
Remove dead users from the queue	We maintain the integrity of the queue by always removing a user from the queue upon a “disconnect” event.

Resolution
We let dead users stay in the queue due to the better time complexity in the average case, as well as ease of implementation.

8.5. API

Request example	**Description**
socket.on(“match”)	

`args: {`
`  difficulty: number,`
`  userId: string,`
`}`
	Creates pending match and adds pending match to queue for corresponding difficulty.

8.6. Development and deployment

Tech stack
NodeJS, RabbitMQ, SocketIO

Development, deployment
See the README.md.

9.Collaboration service

9.1. Features, FRs and NFRs

FR	Description	Design Element
FR-PRO-4	Users can re-enter a room, so that they can reattempt the question.	Users can access history of rooms or last room
FR-PRO-5	Code from previous attempts should persist when a user refreshes, disconnects or re-enters a room, so that he can continue where he left off.	Real-time connection for updating via websocket connection
FR-PRO-6	Paired users in the same room should be able to see where the other user is editing so that they know what is going on.	Y-WebRTC integration handles real-time syncing
FR-PRO-7	Paired users in the same room should be able to see where the other user is highlighting so that they can communicate about the code.	
NFR-SER-4	Communication channels between components should be authenticated	JWT authentication for REST calls, JWT authentication for websocket connections
FR-COM-1	Paired Users can communicate through text chat	Integration with DailyVideo
FR-COM-2	Paired Users can communicate through video and voice	
FR-GEN-3	Users should be able to look at the questions they have attempted before	Endpoint to retrieve history
FR-GEN-4	Users should only be authorized to view their own history	JWT authentication for REST calls

9.2. Architecture

Recall that the collaboration service's main responsibilities are to:

- Handle CRUD operations on rooms
- Update room states in real-time for persistence
- Sync changes between users in the same room

- Provide a means of real-time video or text chat

With these responsibilities, we implement a manager class for each of these - CRUDManager, RoomConnectionManager, Y-WebRTC, DailyVideo - respectively mapping to each of the roles above.

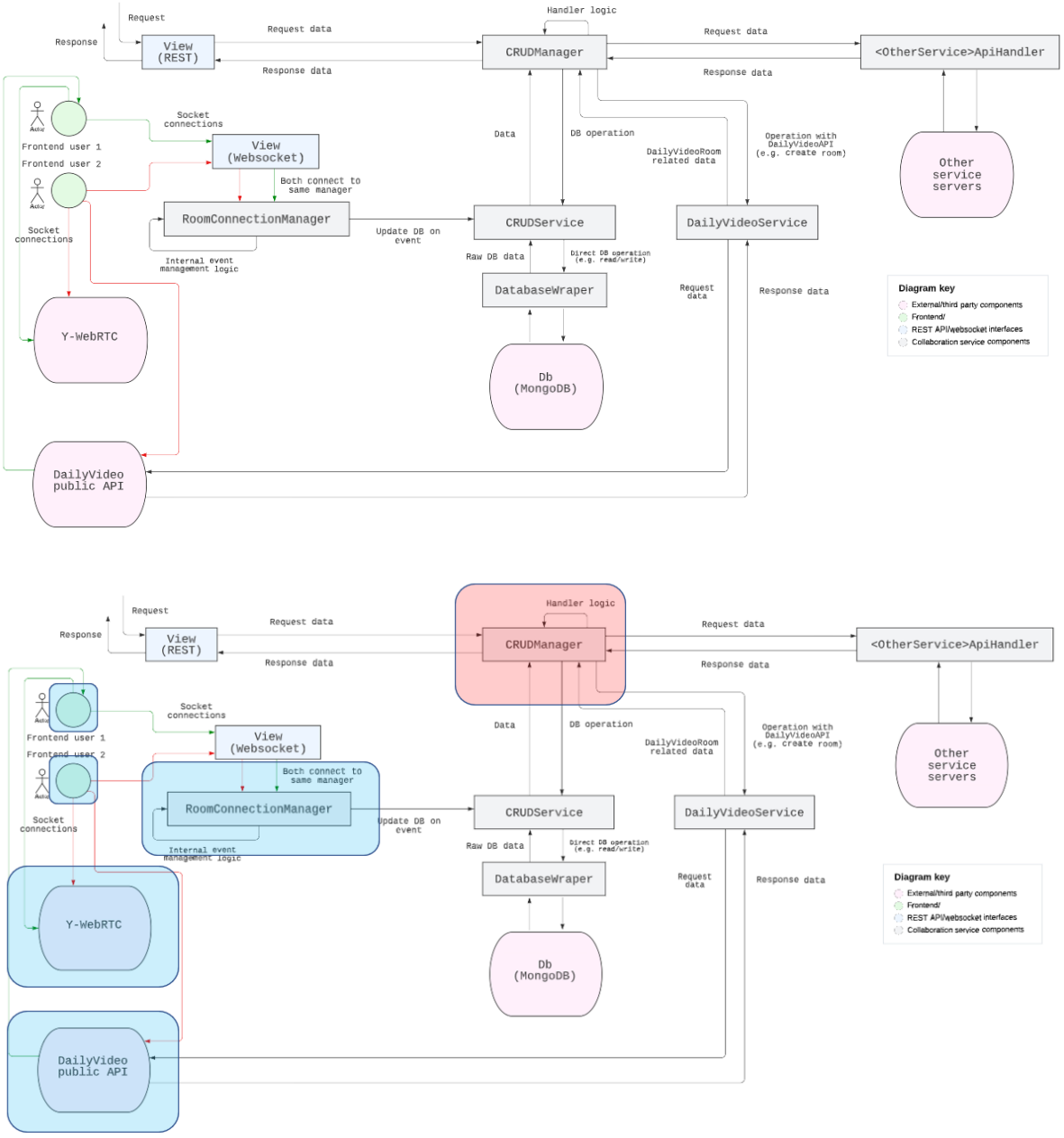


Figure 22: (Top) Architecture diagram of collaboration service. REST calls for CRUD operations are directed by the View (REST) to the CRUDManager. Websocket connections for CRUD are directed by the View (Websocket) to the RoomConnectionManager. Room document syncing is handled by Y-WebRTC. Video/text chat is handled by DailyVideo. (Bottom) Annotated diagram, where blue indicates usage in real-time room participation via websockets, and red indicates usage via REST calls.

Component	Responsibility

View	Exposes REST APIs/websockets to other microservices/the frontend.
CRUDManger	The controller, called by the view when a given request comes in. Serves as the middleman between the view (request layer) and underlying MongoDB database, preprocessing/post processing data as needed.
CRUDService	Service class that handles low-level operations like directly interfacing with the DB. Is injected into the CRUDManager and RoomConnectionManager classes for testability.
DailyVideoService	Service class that handles direct interfacing with the DailyVideo API, to do things like create a room.
Y-WebRTC	An externally implemented real-time-connection service that handles the real-time document syncing.
DailyVideo	The actual web server hosting the Daily Video video/text chat service.
RoomConnectionManager	Similar to the CRUDManager, but essentially a websocket version dependent on a publisher-listener pattern rather than REST API calls. On an edit event published to the websocket connection, the RoomConnectionManager uses its CRUDService to push the changes to DB.
<Some service>ApiHandler	Wrapper class wrapping around the API calls to other services. These include QuestionServiceApiHandler and UserServiceApiHandler.
DatabaseWrapper	Wrapper class around the MongoDB database (via Mongoose library). Wrapper class allows us to separate the actual DB being used with the required operations.

We provide more details here with the class diagrams. The key points:

1. *Room* and *Question* are the main dataclasses, and *Room*s contain a *RoomState*
2. *RoomConnectionManager* and *CrudManager* are at the highest levels of abstraction
3. *RoomConnectionManager* has a *CrudService*
4. *CrudManager* has a *CrudService*, *DailyVideoService*, *UserServiceApiHandler*, *QuestionServiceApiHandler*

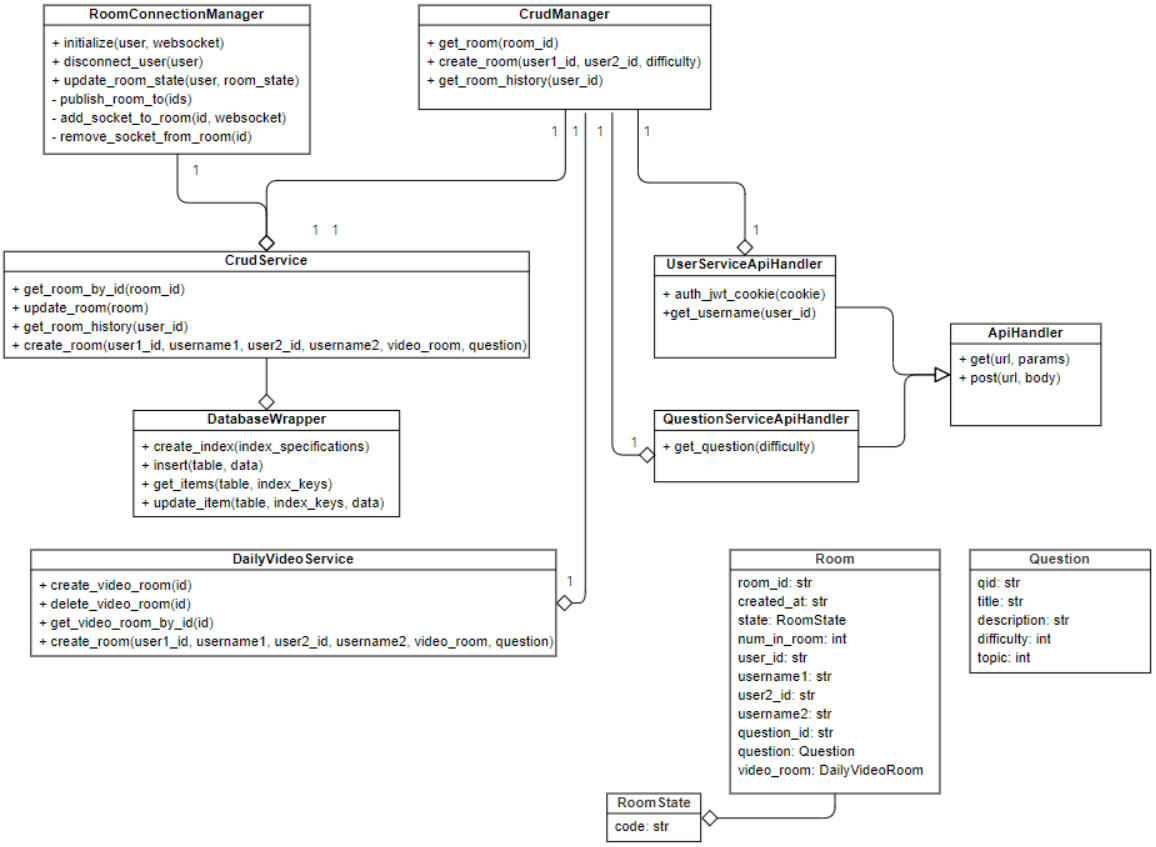


Figure 23: Class diagram for the Collaboration service.

9.3. External integrations

9.3.1. Y-WebRTC

Y-WebRTC is a free and open-source project providing web browsers and mobile applications with real-time communication - and one of its features is document syncing.

Syncing text in real time requires operational transforms and would be extremely tedious and challenging to implement. Instead, therefore, we utilize the open-source Y-WebRTC service which implements a local server and a frontend API to sync text documents.

Find the docs here.

9.3.2. Daily WebRTC Video

For similar reasons, real-time text chat and video require management of real-time connections and persistence - and again outsourcing to an external API makes development smoother. We choose the Daily API for its minimal set up plug-and-play nature.

See the docs here.

9.4. API

Request example	Response example	Description
POST /crud/create		

<pre>{
"user1_id": "3",
"user2_id": "2",
"difficulty": 1
}</pre>
	<pre>{
	"room_id": "1"
	}</pre>
		Creates a room, given the participating user IDs and a question difficulty. Returns a room ID.
		
		If the call fails authentication, returns a 401 unauthorized.

GET /get_room_history		

	<pre>{"data": [{
	"room_id": "3",
	"created_at": "YYYY-MM-DDTHH:mm:ss.sssZ",
	"state": {
	"code": "print(“hello world”)",
	},
	"num_in_room": 1,
	"user1_id": "2",
	"username1": "admin",
	"user2_id": "2",
	"username2": "another",
	"question": {
	"qid": "1",
	"title": "String manipulation",
	"Description": "Join a string.",
	"difficulty": 1,
	"topic": 1,
	},
	"video_room_url": "http//:dailyvideoapi.com"
	}]}</pre>
		Gets all rooms that the specified user has participated in.
		
		Precondition: request should have a JWT cookie encoding a user.
		
		If the call fails authentication, returns a 401 unauthorized.

GET /get_room/{room_id}		

	{
	"room_id": "3",
	"created_at": "YYYY-MM-DDTHH:mm:ss.sssZ",
	"state": {
	"code": "print(“hello world”)",
	},
	"num_in_room": 1,
	"user1_id": "2",
	"username1": "admin",
	"user2_id": "2",
	"username2": "another",
	"question": {
	"qid": "1",
	"title": "String manipulation",
	"Description": "Join a string.",
	"difficulty": 1,
	"topic": 1,
	},
	"video_room_url": "http//:dailyvideoapi.com"
	}
		Gets a room specified by the room_id.
		
		Precondition: request should have a JWT cookie encoding a user.
		
		If the call fails authentication, returns a 401 unauthorized.

9.5. Sequence Diagrams

Here, we demonstrate two representative examples.

Creating a room

1. Service (e.g. matching service) makes a POST request to /crud/create
2. View validates the request, creating a CRUDManager and injecting a CRUDService, QuestionServiceApiHandler and UserServiceApiHandler into it.
3. The CRUDManager queries the question service for a question of the specified difficulty, the user service for the user’s usernames, and then passes these to the CRUDService to create a room.
4. The CRUDService validates the inputs, writes to DB, and returns the created object as a data class.
5. The CRUDManager forwards the data class to the view layer, then eventually returns it to the requester.

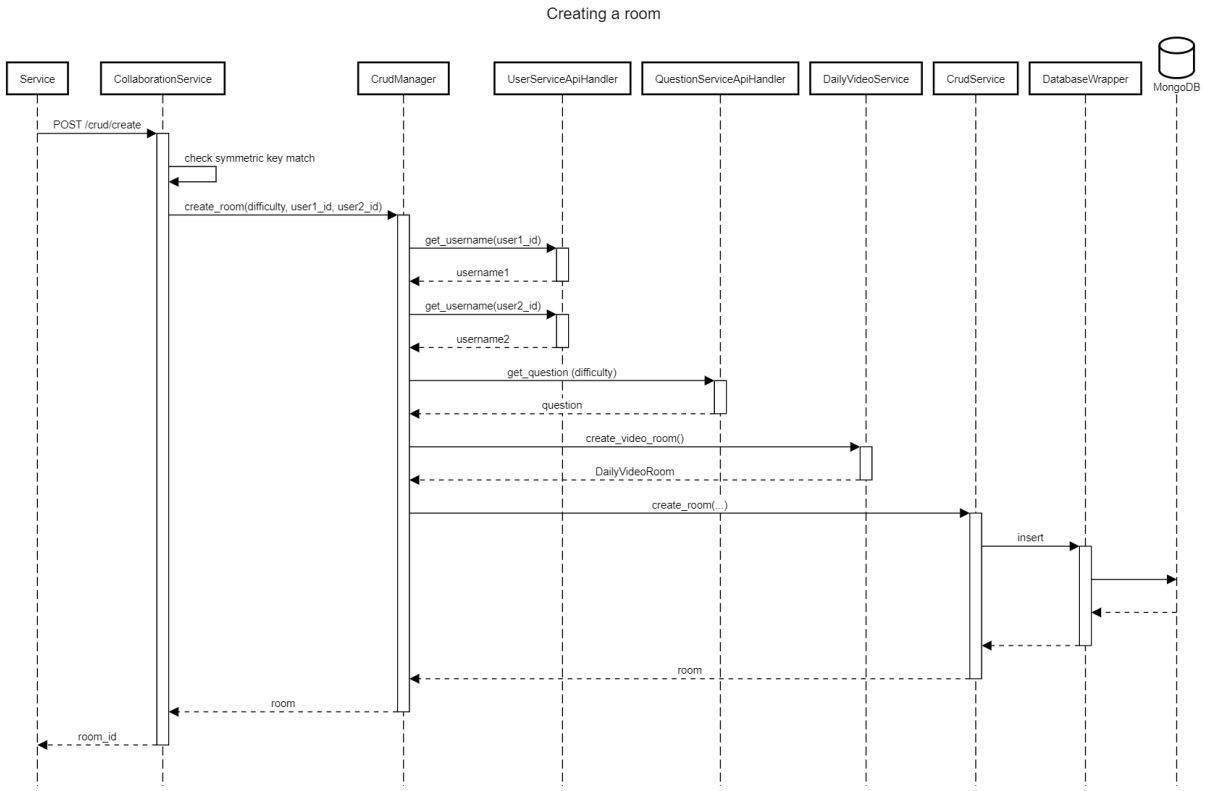


Figure 24: Sequence diagram illustrating how the Matching Service (or any other service) can create a room. The collaboration service checks that a symmetric key matches, directs the request to the CRUDManager, which gets the details (username, question, etc.) and hands it to the CRUDService to actually create the room and save to DB.

Websocket connection from the frontend in a room with 2 users

1. An existing room with User1 and User2 exists. Without loss of generality, we consider User1 and User2 as any two users.
2. User1 makes a websocket connection to /room with a JWT token.
3. Using FastAPI dependency injection, we expect the request to have a JWT cookie, and we authenticate it with the user service via an asynchronous call if the cookie exists. If this step fails, the connection is closed.
4. A RoomConnectionManager is created *if it does not exist for this room*, else the existing one is *retrieved from memory* (a hashtable keyed on room ID). The websocket is passed to the room connection manager for initialization, and upon success publishes the room to the frontend and now listens for changes.
5. User1 connects to the YWebRTC server to sync its frontend document state.
6. User2 connects, and does the same things from 2-5
7. User1 now makes some change in the frontend document..
8. YWebRTC handles the frontend synchronization, then publishes the document to both User1 and User2.
9. The debounced state from User1 is passed through the websocket as a JSON file.
10. The new state is saved to DB by the RoomConnectionManager.
11. User2 decides to disconnect.
12. For persistence, User2's room state (which should be the same as User1) is saved to DB.

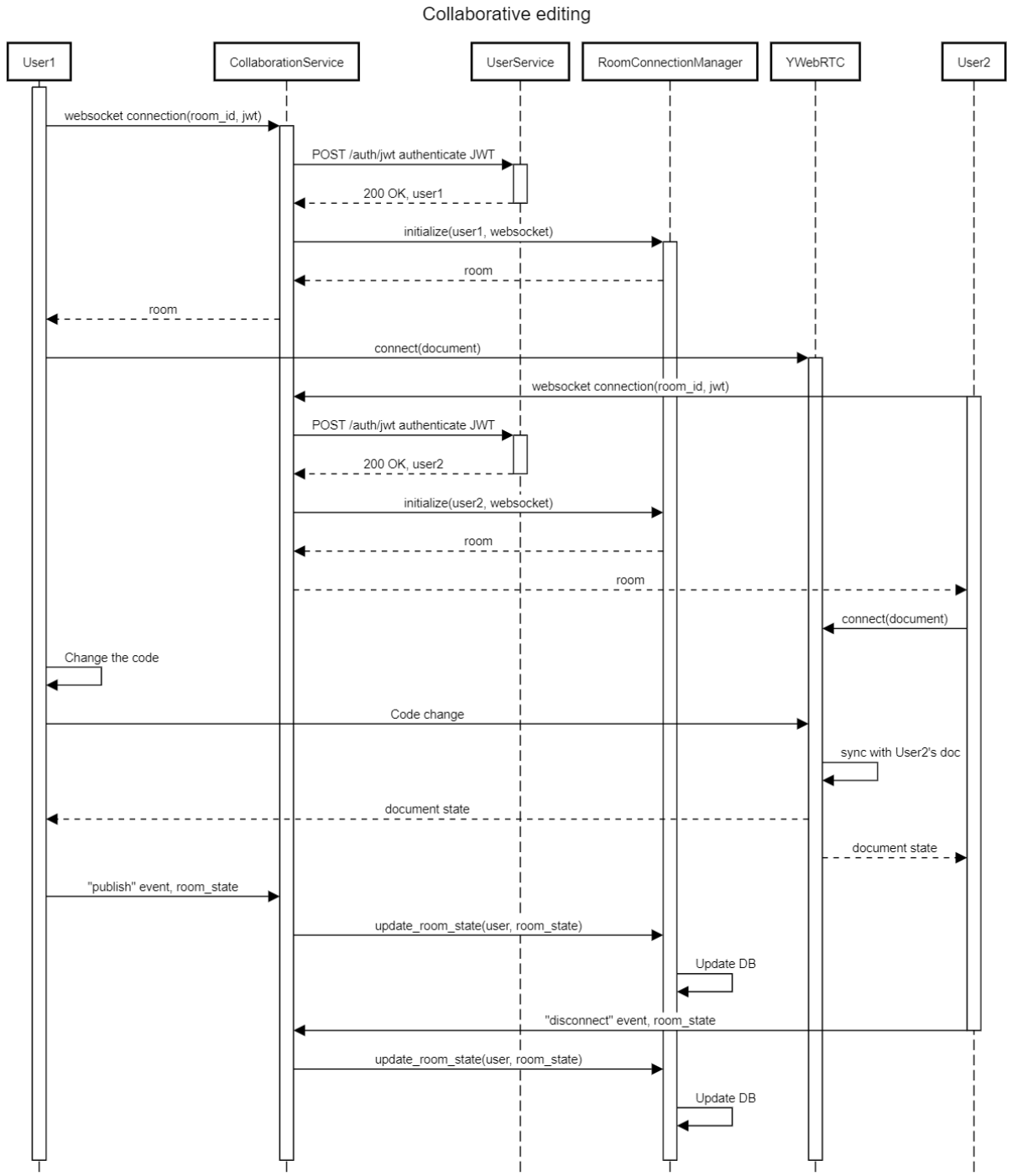


Figure 25: Sequence diagram illustrating how collaborative editing is managed by the Collaboration Service. The key steps - user makes a websocket connection with a JWT, collaboration service authenticates it by checking with the user service. User makes a change, Y-WebRTC handles syncing, and then the debounced change is pushed and saved to DB.

9.6. Testing

We use pytest as the main test framework due to its simple integration with FastAPI and ability to mock objects and services easily.

All CRUD operations have unit tests written for them with a mock database for testing implemented via the open-source MongoMock library.

9.7. Development and deployment

Tech stack
FastAPI, pytest, MongoMock/PyMongo

Running the service, running/adding tests
See the README.md.

10. Database: MongoDB

MongoDB is an easy-to-use, easy-to-scale NoSQL database. Given the needs of *PeerPrep* as a prototype handling generally unstructured data objects (e.g. a user, a room, etc.) and no real need for strictly structured data - MongoDB arises as an easy and natural choice.

For ease of local development and testing, we populate the local MongoDB instance with data from a JSON file, allowing us to run end-to-end tests and work with dummy data as needed.

See the docs for .js and .py ORM libraries:

- Mongoose v6.7.0: API docs (mongoosejs.com)
- PyMongo 4.3.2 Documentation — PyMongo 4.3.2 documentation
- MongoDB Documentation

11. How the services work together (under the hood)

Create account, login, find match, enter room, collaborate

Sequence

1. User creates an account. Frontend makes a POST request to User Service to create the account.
2. User logs in with credentials. Frontend makes a POST request to User Service to authenticate. Successful authentication returns a 200 OK and sets a HTTP-only JWT cookie.
3. User tries to find a match, selecting a difficulty. Frontend makes a POST request to User Service to get a JWT for authenticating the websocket connection. User Service returns it.
4. Frontend then opens a websocket connection with Matching Service and the JWT as a query parameter.
5. Matching Service makes a POST request to User Service to check the validity of the JWT, returning a 200 OK on success.
6. When a match is found, Matching Service makes a POST request to Collaboration Service to create a room with the specified difficulty and matched users.
7. Collaboration Service makes a POST request to Question Service to get a random question of that difficulty, and to User Service for the user's usernames.
8. Collaboration Service makes a POST request to DailyVideo API to create a video chat room.
9. Collaboration Service then writes the created room to MongoDB, and returns the room ID to Matching Service, which then returns that to the frontend before closing the websocket connection.
10. User is redirected to the collaboration page.
11. The frontend repeats step 4 to get a new JWT for authentication with Collaboration Service. A websocket connection is opened with the Collaboration Service.

12. Collaboration Service makes a POST request to User Service to authenticate the JWT, and upon success, publishes the room to the frontend.
13. Frontend opens a websocket connection to Y-WebRTC server to sync the document state, and another websocket connection to the DailyVideo server for video chat.
14. Upon a change in the code, Y-WebRTC syncs it, and then a debounced state is pushed to Collaboration Service, which saves it to MongoDB.

12.DevOps

12.1.Sprint planning

We followed an adjusted AGILE methodology. We had weekly stand ups and sprint meetings where we discuss what we did, blockers faced, and tasks to add to our backlog/add to the coming sprint.

To track sprint tasks, we used GitHub Projects (and its various views, like a Kanban board) and its integration with GitHub issues.

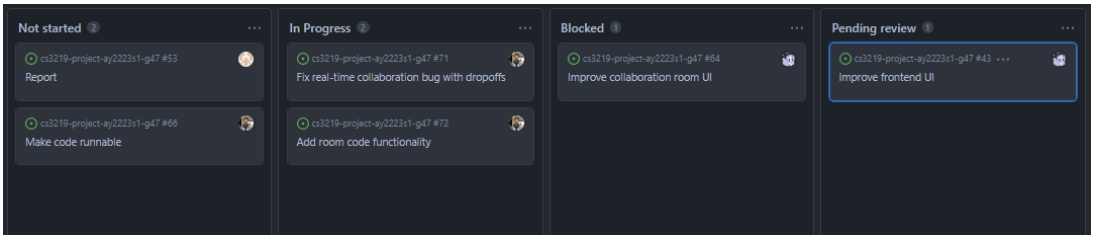


Figure 26: A screenshot of our Kanban board with some issues pending.

12.2.Continuous Integration

To avoid regressions, Github Actions was used to run automated tests on every pull request to merge into the main branch. These tests were service specific, with one for each microservice.

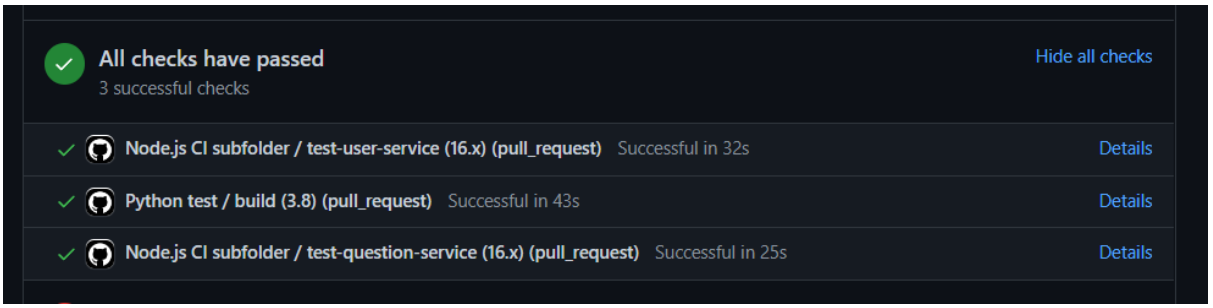


Figure 27: A screenshot of our CI runs on a pull request.

13.Reflections

This project has allowed us to go through the entire software building process from the ground up. We were able to learn and implement different tools and tech stack alongside new concepts learnt in the lectures.

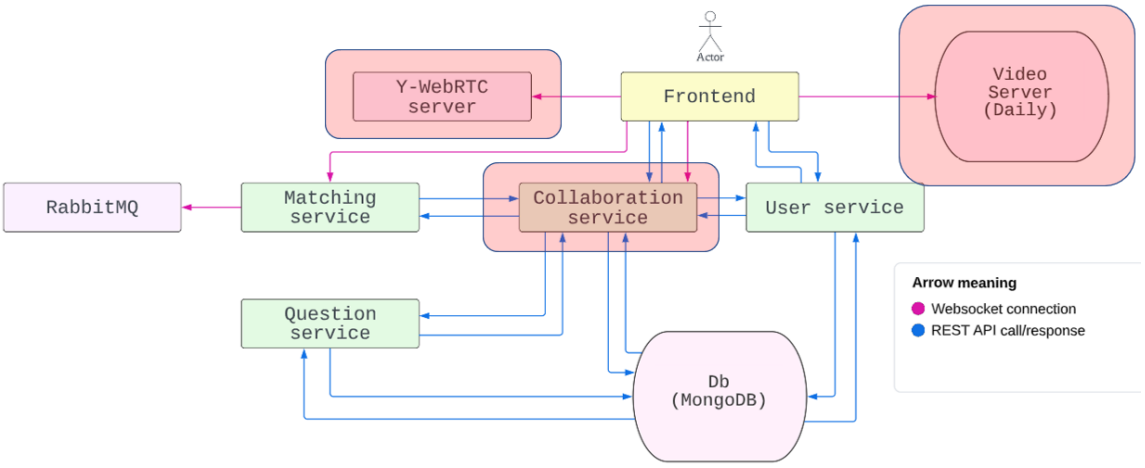
13.1.Areas of improvement/features to implement

13.1.1.Ingress controllers

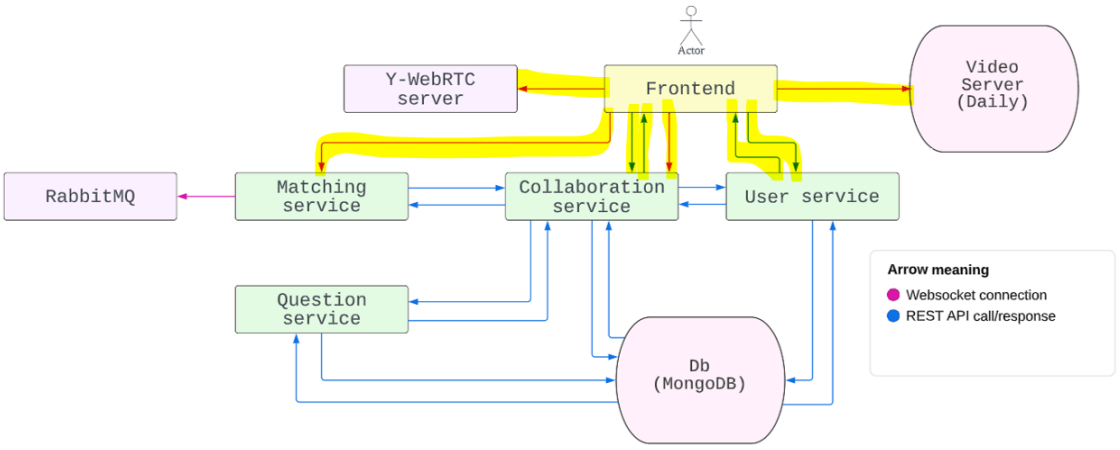
At present, the microservices are organized in a “choreographed” way, making coordination and tracing through the code challenging, and could be a problem as more features are

added. As an example, the collaboration service consists of three components, but is currently accessed through 3 separate interfaces.

Three components all *technically* belong to collaboration service



Frontend communicates individually with services



We can solve these issues with ingress controllers

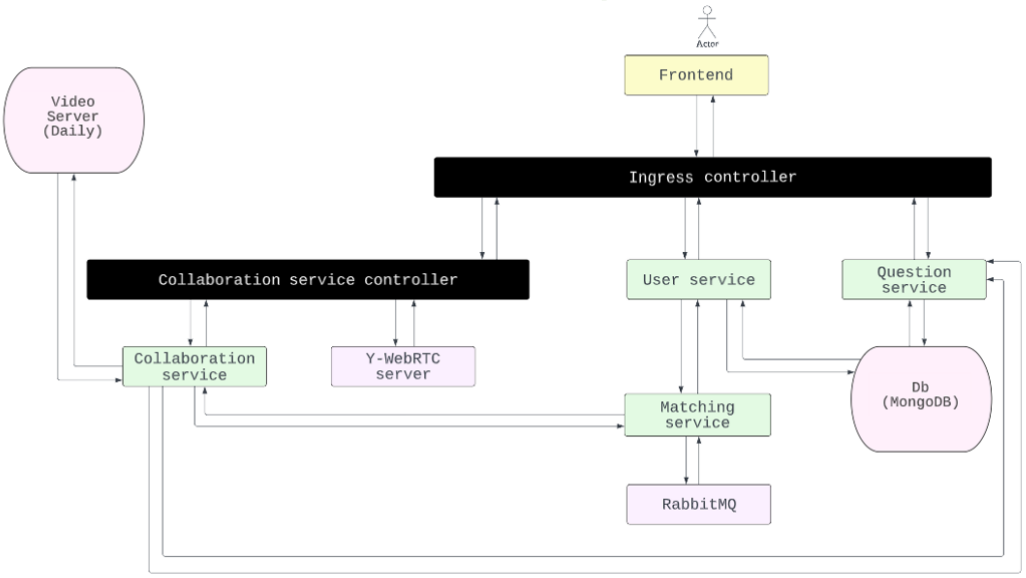


Figure 28: A graphical illustration of why ingress controllers would be useful. (Top) All three components technically make up the collaboration service, but are accessed through independent interfaces. (Middle) The frontend web server currently directly interfaces with each of the microservices. (Bottom) The introduction of two ingress controllers would have with traceability, as the frontend would see a "single backend", while the individual components of the collaboration service would not be exposed.

13.1.2.Code execution

The live collaborative editor is currently just a text editor that has syntax highlighting. It would be beneficial for *PeerPrep* to allow for the code to be compiled/executed so that the users can test their programs and verify that it works and debug the code together, which would be beneficial. This could also be paired with provided test cases to run the program with.

13.1.3. Provide user feedback

Features like a session timer, percentile, code execution speed and statistics (e.g. percentile) would help users to gain timely feedback on their performance.

Additionally, the choice of question topics could be useful here too.

13.1.4.More collaborative features

Another simple feature that would be useful would be to have a room code, allowing friends and peers to coordinate their sessions.

13.1.5.Telemetry

Exception logging, user tracking and analysis, etc. would be nice-to-haves for development.

13.2.Contributions

Name	Contributions
TAN YONG-JIA, NAAMAN	● Collaboration service ○ Live code editor ○ Video chat ● User Service ● Question Service ● DevOps (Dockerize, CI, Local deployment) ● Frontend ● Project Report, Slides
GERNENE TAN	● Matching service ● Frontend ● DevOps (Dockerize) ● Project Report

DANIEL TAN REN JIE	● User service ○ Authentication ● DevOps (Dockerize, CI) ● Project Report
LEE HERN PING	● Question service