CS3219 PeerPrep Documentation

Group 53

Ng Seng Leng A0217455A

Neha George A0264992M

Isaac Tan A0217444H

Introduction

Background

Coding interview questions have become the standard for companies to gauge the standard of potential employees and interns. As such, more than ever there is a need for platforms to offer coding practice. Platforms like LeetCode may offer relevant questions, but our application distinguishes itself with these features:

1. Collaborative coding space
2. Chat to practice communicating ideas clearly

Many new users of platforms like LeetCode are intimidated by the questions and may not know where to begin. If they get stuck, they may feel alone and discouraged. They also are not able to communicate their thought process and where they are confused to anyone.
Our collaborative approach to coding is more akin to real job contexts, as no employee is in a vacuum. It encourages good communication, which is a very important trait of a skilled engineer. It also brings an aspect of “fun” to the interview preparation process, which can be an important motivating factor.

Application Start-Up

1. Open 5 separate terminal windows. For steps 2-6, conduct each in a new terminal window.
2. **`cd matching-service && npm install && npm start`**
3. **`cd question-service && npm install && npm start`**
4. a) **`cd user-service &&`**
   b) Make a copy of **`.env.sample`** file and rename it to **`.env`**.
   c) Start up local mongoDB service in the background with **`brew services start mongodb-community@6.0`** on a Mac or whatever method is needed for non-mac OSes to get a mongoDB instance running
   d) Uncomment and change the **`DB_LOCAL_URI`** in the **`.env`** file to **`mongodb://localhost:27017/${any_db_name_you_want}.`**
   e) Change **`SECRET_KEY`** to any string.
   f) **`yarn install && yarn start`**
5. **`cd chat-service && yarn install && yarn start`**
6. **`cd frontend && yarn install && yarn start`**
7. Navigate to localhost:3000 on 2 different browsers to simulate 2 different sessions.

Functional and Non-Functional Requirements

User service

Functional Requirements	**Priority**
Users can create a new account with a username and password	High
Users can sign into an existing account with a username and password	High
Users can log out of the platform	High
Users can delete their account	High
Each user should have a unique username	High
Users can change their password	Medium
Non-Functional Requirements	
Passwords should be salted and hashed (Security)	High
Accounts are secured with JWT authentication (Security)	High
Sessions are persistent for users (Usability)	Medium

Matching service

Functional Requirements	**Priority**
Users should be allowed to select the difficulty of questions they want to attempt	High
Users should be able to match with other users with similar difficulty levels and put them in the same room	High
Users should be matched within 30 seconds if there is a valid match	High
Users should be informed if no match is available if no match can be found within 30 seconds	High
The system should provide the user with the means to leave a room once matched	High
Non-Functional Requirements	

Question Service

Functional Requirements	**Priority**
Questions need to be indexed in terms of difficulty (Low, Medium, High)	High
Questions need to be indexed in terms of Topic	High
Ability to store Images and diagrams for questions	High
Non-Functional Requirements	
Question Service should still operate even if Leetcode goes down (Availability)	High

Chat Service

Functional Requirements	**Priority**
User must be able to see the other user's messages in real time	High
User needs to be able to see who sent which message	High
Non-Functional Requirements	
User needs to alerted when match leaves the room (Usability)	High

Collaboration Service

Functional Requirements	**Priority**
User must be able to see the other user's edits in real time	High
User must be able to see the other user's presence in the same room	High
Non-Functional Requirements	
Other user's inputs should have a delay of no more than 5 seconds (Performance)	Medium

Technologies and Architecture

Tech Stack

Frontend	Material UI, React
User Service	MongoDB, NestJS
Matching Service	Sequelize, Socket.io
Question Service	Sequelize, Express
Collaboration Service	WebRTC with CRDT data storage
Chat Service	Socket.io

Architectural Decisions

Microservices vs Monolithic Architecture

While both are feasible for building this application, we found the microservice architecture more effective for this project.

Benefit	**Elaboration**
Ease for testing	Our team decided to work on different services concurrently. A microservice approach allows us to work on different services and merge them effectively during sprint meetings. Testing can also be done easily, as each component and service can be tested before merging.
Scalability for developing new features	A microservice architecture's modularity means that additional features like the chat and coding features can be added easily in the future, without significant changes to the current codebase

Marco Architectural Model

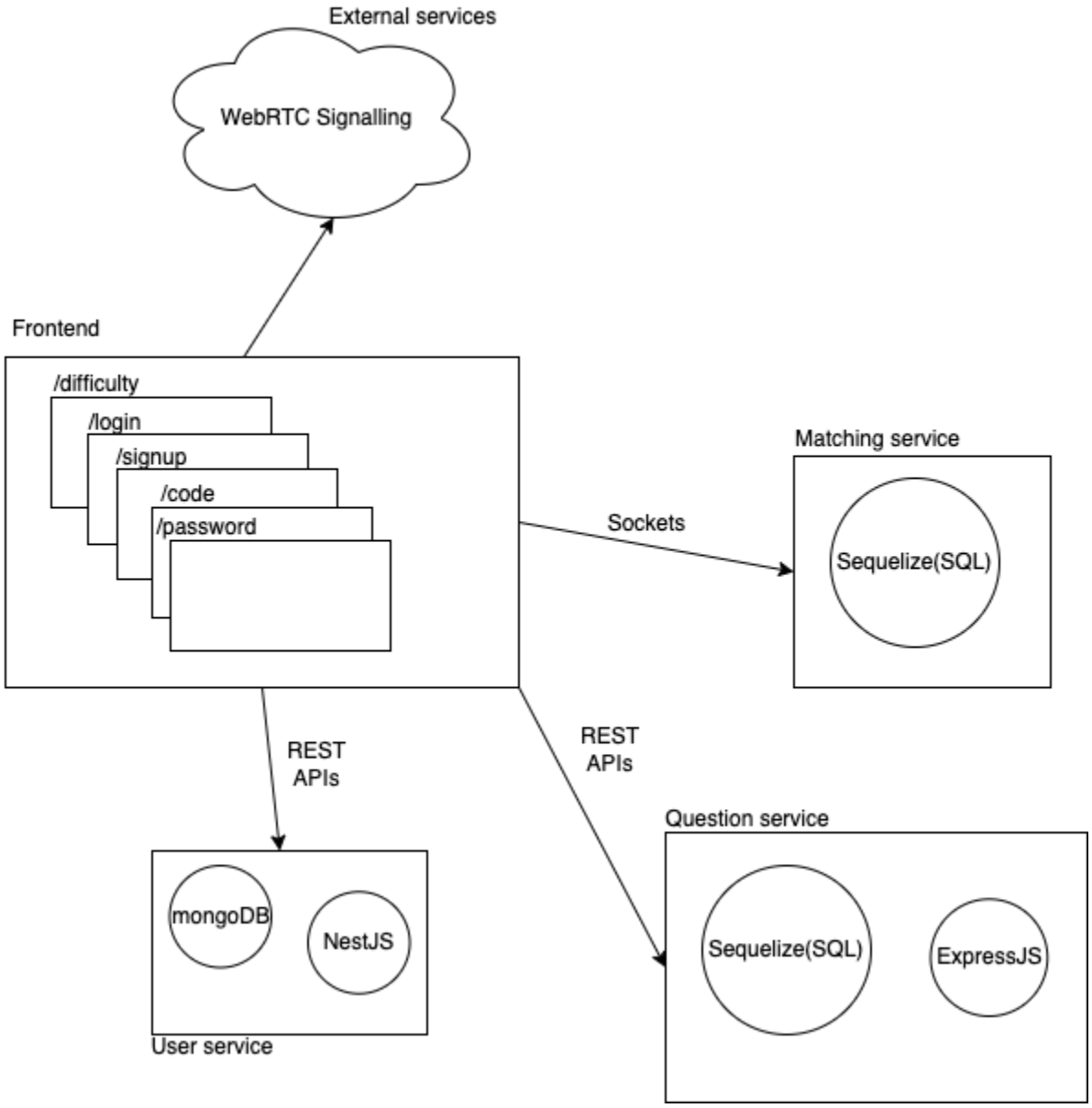


Figure 1: Overview of chosen architecture

As our services are not deployed, most of the backend services exist by itself on localhost, hence the frontend has to call on the services individually instead of taking advantage of an API gateway to route the frontend requests. The elements in figure 1 are self-explanatory, where each of the shapes represents the lower-level services that each backend service employs and forms up their internal architecture, while the frontend comprises pages a user can access.

Development process

User Stories

User story ID	user story
US-1	as a user, I want to create a new account with a unique username and a password of my choosing
US-2	as a user, I want to sign in to my platform with my username and password
US-3	as a user, I want to log out of the platform so others cannot use my account on the same device
US-4	as a user, I want to delete my account when I am done with it
US-5	as a user, I want to change my password
MS-1	as a user, I want to select difficulty of questions i want to do with a peer
MS-2	as a user, I want to match with another user that selected the same difficulty as me and enter a room to code in
MS-3	as a user, I want to be matched with another user within 30 seconds or have a failure message shown to me
MS-4	As a user, I want to be able to leave the room after I matched with another user
CS-1	as a user, I want to be able to chat to the other user in realtime
CS-2	as a user, I want to see who typed what messages in the chatbox
CS-3	as a user, I want to be able to know if my match left the collaboration room
CoS-1	as a user, I want to be able to see the other user's code edits in realtime
CoS-2	as a user, I want my inputs to not have a delay of more than 15 seconds when sending and receiving code edits
QS-1	as a user, I want to see a question related to level of difficulty selected by me and my match

key: US - user service, MS - matching service, CS - chat service, CoS - collaboration service, QS - question service

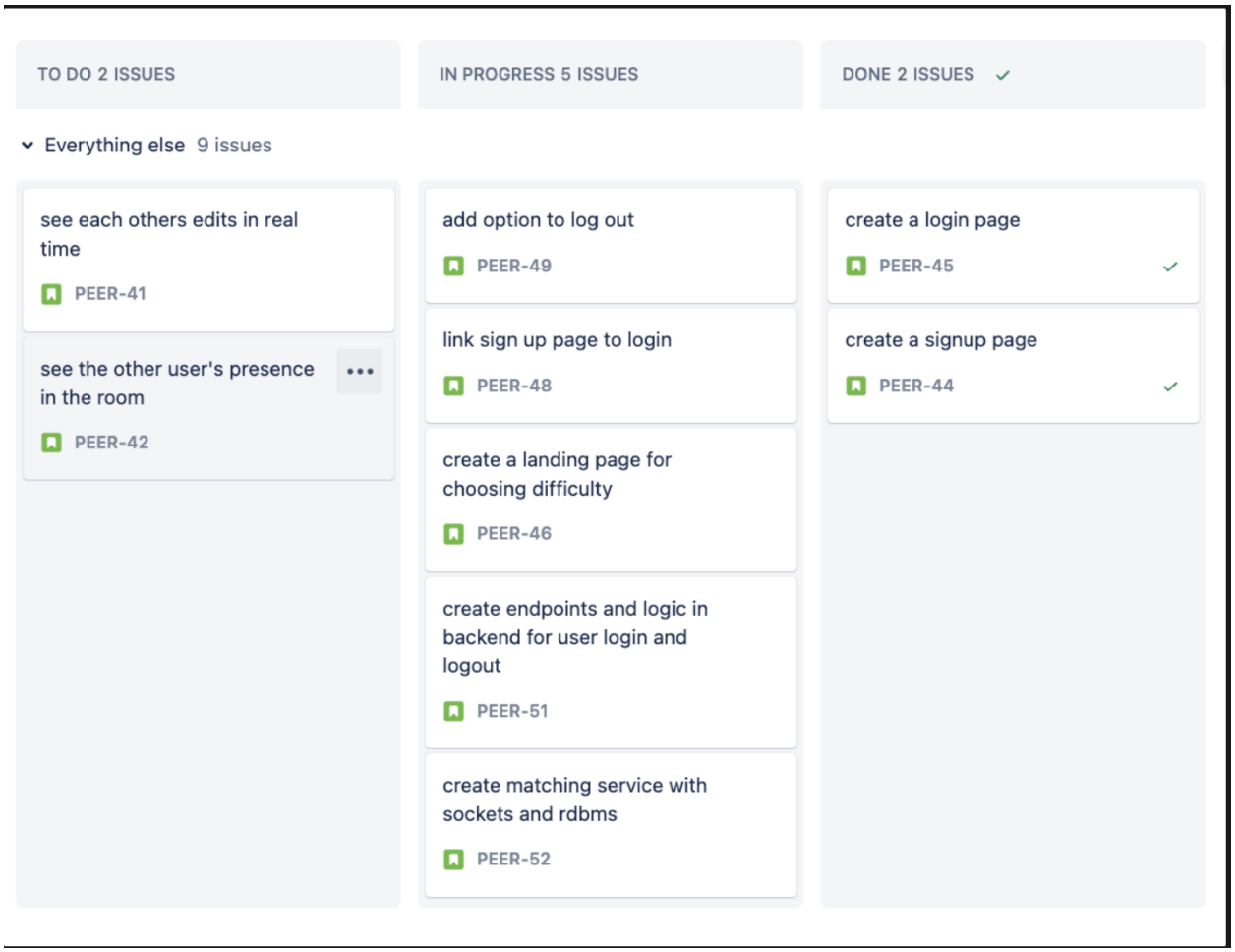
Weekly Sprints

Sprint	Attempted User stories	Finished user stories	Tasks assigned	Tasks completed	Comments
0 (week 3)			Preplanning phase (service implementation, architecture, deploying local instead of on cloud services)		
1 (week 4)	US-1, US-2, MS-1, MS-2		- begin work on a signup page **[US-1]** - begin work on a login page **[US-2]** - create a temporary landing page, to be difficulty page **[MS-1]** - begin work on user service (login/signup) **[US-1, US-2]** - begin work on matching service **[MS-2]**		
2 (week 5)	US-1, US-2, MS-1, MS-2		- finish frontend interface for login and signup page **[US-1, US-2]** - implementing routes for login and signup in backend **[US-1, US-2]** - start implementing frontend interface for difficulty page **[MS-1]** - implementing DB logic for storing match requests and deleting on fail via socket communication **[MS-2]**	- begin work on a signup page **[US-1]** - begin work on a login page **[US-2]** - create a temporary landing page, to be difficulty page **[MS-1]** - begin work on user service (login/signup) **[US-1, US-2]** - begin work on matching service **[MS-2]**	
3 (week 6)	US-1, US-2, MS-1, MS-2, MS-3		- integrate user service with login and signup page **[US-1, US-2]** - integrate difficulty page with matching service **[MS-1, MS-2, MS-3]** **Add-ons from initial task mid-week:** - modify method of matching to: frontend has 30sec timer, after timer emit fail to backend, backend stops matching and deletes entry from DB. **[MS-2, MS-3]**	- finish frontend interface for login and signup page **[US-1, US-2]** - implementing routes for login and signup in backend **[US-1, US-2]** - start implementing frontend interface for difficulty page **[MS-1]** - implementing DB logic for storing match requests and deleting on fail via socket communication **[MS-2]**	- discovered issues with frontend timer and backend timer mismatch, propose method of frontend leading the matching process
4 (week 7)	MS-4, CS-1, CoS-1, QS-1	US-1, US-2, MS-1, MS-2, MS-3	- implement logic to signal room leaving to other user **[MS-4]** - do research and experiment on best ways to implement a realtime collaborative code editor **[CoS-1]** - do research and experiment on chatting with another user **[CS-1]** - begin researching on how to obtain a question bank that can be stored in a database **[QS-1]**	- integrate user service with login and signup page **[US-1, US-2]** - integrate difficulty page with matching service **[MS-1, MS-2, MS-3]** - modify method of matching to: frontend has 30sec timer, after timer emit fail to backend, backend stops matching and deletes entry from DB. **[MS-2, MS-3]**	
5 (week 8)	CS-1, CS-2, CoS-1, QS-1	**MS**, US-1, US-2	- found API that provides all Leetcode questions, creating python script to parse and store json file **[QS-1]** - made a working chat function on frontend using sockets, but messages are broadcast to all users, need to find way to put users in rooms **[CS-1, CS-2]** **Carry over:** - do research and experiment on best ways to implement a realtime collaborative code editor **[CoS-1]**	- implement logic to signal room leaving to other user **[MS-4]** - begin researching on how to obtain a question bank that can be stored in a database **[QS-1]** - do research and experiment on chatting with another user **[CS-1]**	- debating on using webRTC or sockets to implement the collaborative code editor as webRTC requires setting up of signalling servers, STUN servers and TURN servers; looking into how to to set those up as well
6 (week 9)	CoS-1, CoS-2, CS-3, QS-1	**MS**, US-1, US-2, CS-1, CS-2	- use vjs library and webRTC to implement code editor with minimal setup on frontend **[CoS-1, CoS-2]** - implement socket to listen for room leave event **[CS-3]** - implement question service endpoints to fetch data from leetcode question database (local) **[QS-1]**	- do research and experiment on best ways to implement a realtime collaborative code editor **[CoS-1]** - made a working chat function on frontend using sockets, but messages are broadcast to all users, need to find way to put users in rooms **[CS-1, CS-2]** - found API that provides all Leetcode questions, creating python script to parse and store json file **[QS-1]**	- a unique roomId is generated for the collaboration and chat by combining two socketIds
7 (week 10)	US-3, US-4, US-5	**MS, CS, CoS, QS**	- implement log out route **[US-3]** - implement delete account route **[US-4]** - implement change password route **[US-5]**	- use vjs library and webRTC to implement code editor with minimal setup on frontend **[CoS-1, CoS-2]** - implement socket to listen for room leave event **[CS-3]** - implement question service endpoints to fetch data from leetcode question database (local) **[QS-1]**	
8 (week 11)	US-Misc	**MS, CS, CoS, QS**	- implement user persistence (when page is refreshed, user remains logged in but is redirected to homepage instead of log in page) **[US-Misc]**	- implement log out route **[US-3]** - implement delete account route **[US-4]** - implement change password route **[US-5]**	
9 (week 12)		Completed	- start cleaning up documentation - start presentation slides	- implement user persistence (when page is refreshed, user remains logged in but is redirected to homepage instead of log in page) **[US-Misc]**	

We adopted the **agile development process** using the **Scrum methodology** of doing weekly sprints in order to progress on our project development. Each sprint duration is 1 week long and our initial sprint is spent on deciding what frameworks, tech stacks, features and user stories we are going to work on and develop, and also to plan for the first sprint. Every subsequent sprint starts on the monday of the week and ends on the monday of the following week, where we

then hold a meeting to evaluate **what has been done**, what **needs more work** on and **what to do for the upcoming sprints.** Throughout the week, if anyone has any issues, instead of holding a daily scrum meeting we send messages in our telegram chat to see if anyone has any comments or helpful inputs, or if we need to propose a new change of direction. This fits our workflow better as we personally find daily scrum meetings a little bit time consuming and this ad-hoc updating allows for more flexibility while still being able to keep others updated about our individual progress.

Jira



We use Jira as our task tracker for the week to visualize our tasks. The Jira board is updated with the week's tasks as discussed during the sprint meeting and over the week the tasks are progressed along from TODO to PROGRESS to DONE and as individual members we check the board daily to see how our other members are progressing and if tasks have been stuck at TODO for a while we give them a nudge or find out if something is hindering their progress.

Services

User Service

This user service represents the Model and View component of the Model View Controller (MVC) Pattern. The logic that handles requests from the frontend concerning user authentication and authorisation and logic that handles updating of the NoSQL database for user information can be found here.

NoSQL vs SQL

NoSQL databases are preferred over SQL databases since they are more horizontally scalable and in our use case since we are not able to easily increase the resource consumption on a single server due to free tier limits, we can instead depend on starting more servers should the traffic for this particular service increase, so NoSQL is used here over SQL.

Error Handling

Errors are handled on the user service by returning the user a pop up error box with the error message inside. Some common errors that are handled by the application are

- Empty fields
- User does not exist (log in)
- User already exists (sign up) (refer to Fig. 2)

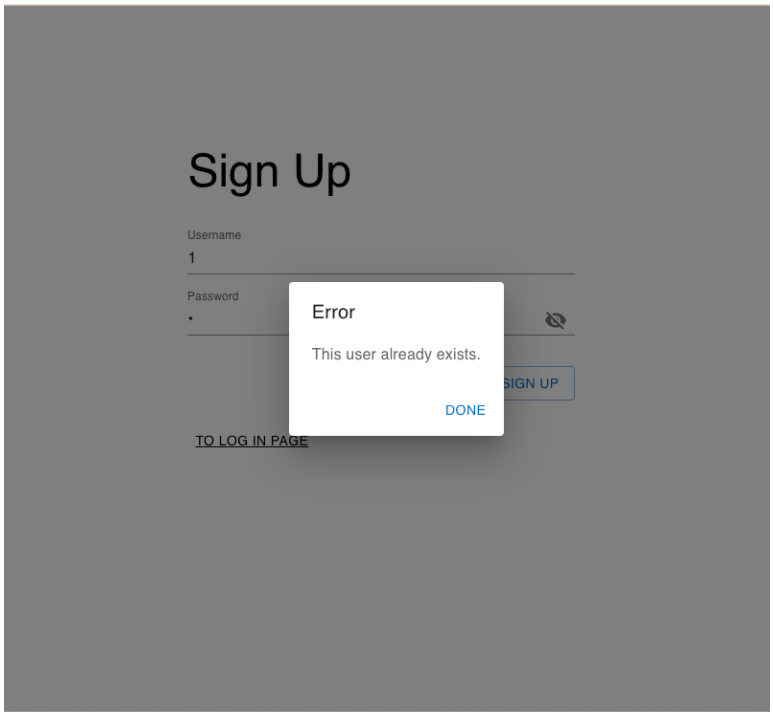


Figure 2: The pop up referenced above is shown in this image

Model View Controller Pattern (MVC)

Reasons for choosing to employ this pattern:

Benefit	Elaboration
Reduce Coupling and improve testing	MVC helps to reduce coupling, which improves **testability** of the different components
Separation of concerns	Decoupling the data layer from the logic layer allows us to easily modify and replace layers if need be without having to modify a lot of code, saving on development time

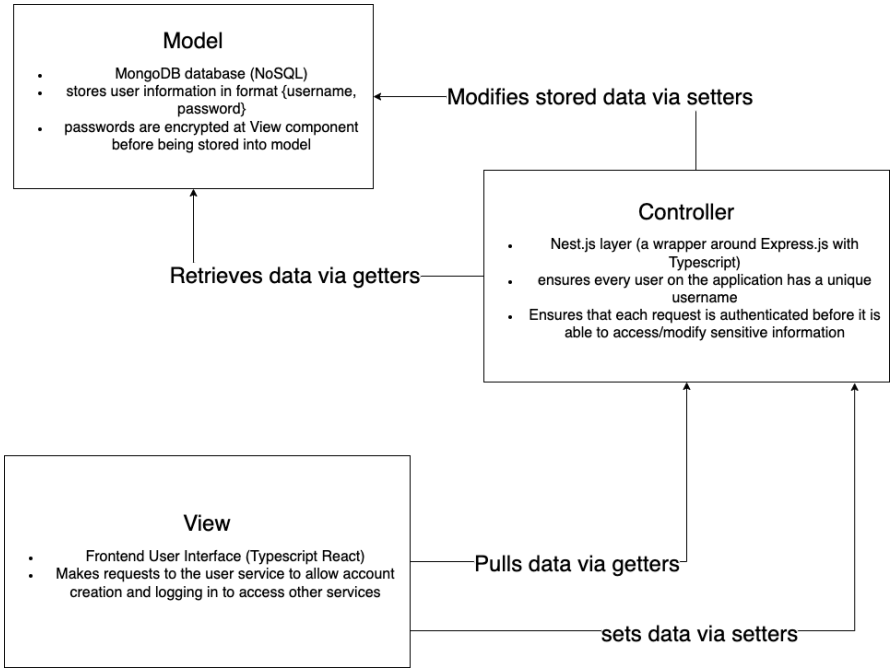


Figure 3: MVC diagram

Session persistence

A small issue with the original implementation was that there was no way of making sure the user was logged in after he refreshed the page or if he accidentally closed the tab, making it very annoying during the development process and potentially annoying for future users as well. So, in order to ensure user persistence, a request is sent to the backend everytime the page is first loaded in to check if the user is currently in a session. A NoSQL database is used to track sessions and an entry is created when the user logs in and is deleted when the user explicitly logs out. The user activity can be seen in Figure 4.

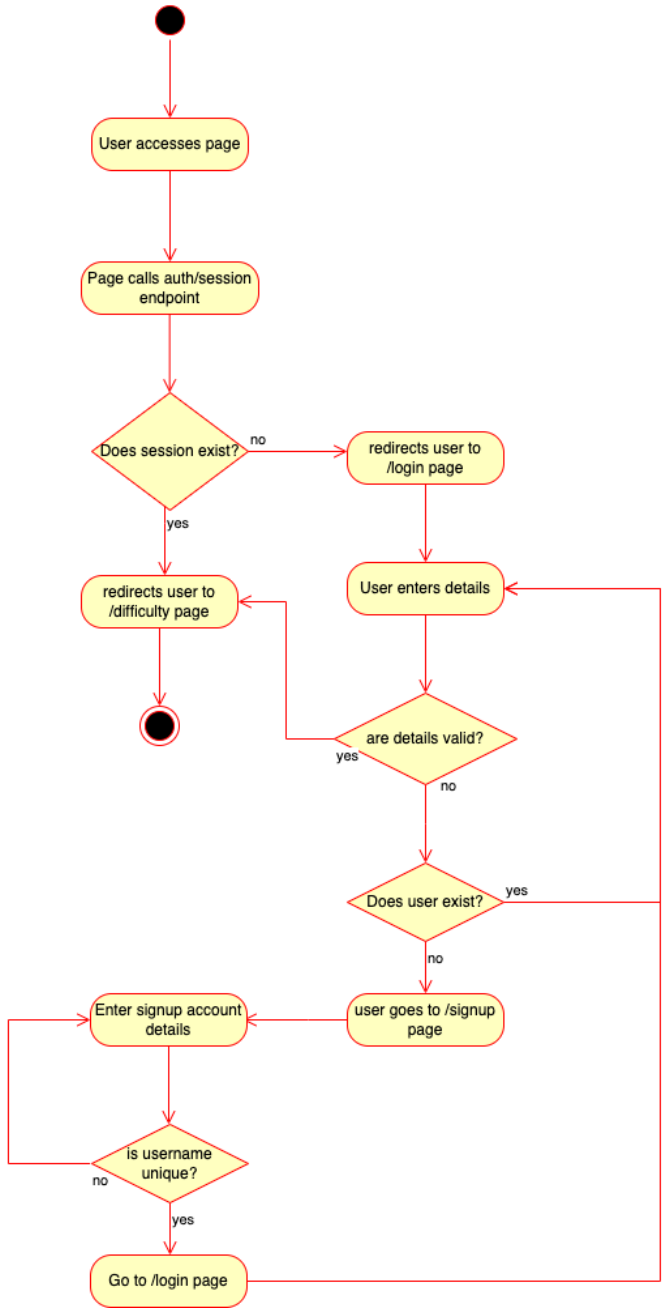


Figure 4: Activity diagram for logging in to application

Matching Service

The matching service uses Socket.IO's sockets to create matches between two users that would like to practice questions of the same difficulty. It deals with the following events: **`match`**, **`matchFail`**, **`matchSuccess`**, and **`matching`**.

Here is a brief description of how each event is used:

Event	Description
match	This is emitted from the frontend to the service when a user requests a match.
matchFail	This is emitted from the frontend to the service when the frontend's 30 second timer has terminated without finding a match. This informs the backend to delete the temporary database entry created for the user's match request.
matchSuccess	This is emitted from the backend to the frontend when there is a successful match.
matching	This is emitted from the backend to the frontend when a match is being searched for.

An important design decision for the matching service was having the frontend deal with the timer as opposed to the backend. This was to ensure a seamless user experience. The frontend will timeout 30 seconds after the user's click, as desired, and we can also easily display the number of seconds remaining to the user without communicating with the backend.

One design alternative that was considered was having the backend also keep a timer and emit the matchFail event, but we found that often a frontend-backend communication delay meant that the user experience was adversely affected because the timers were misaligned. Another alternative would be having only the backend keep a timer, but an evident issue with this is that an event would need to be emitted every second so that the frontend would be able to update the "seconds remaining." Given communication delay issues, this did not seem like a clever alternative — thus it was discarded.

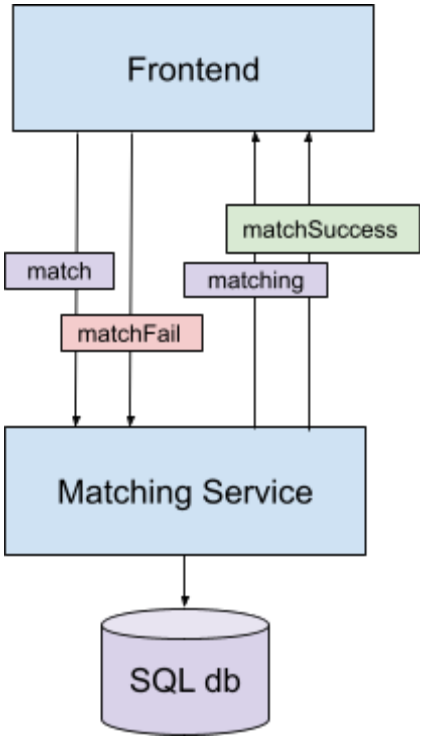


Figure 5: Diagram of the matching service's architecture

Error Handling

If a match was unsuccessful, the user would be redirected to the difficulty page at the end of the 30s, giving them a chance to try again with the same difficulty or try to find a match with a different difficulty

Chat Service

The chat service allows connected or matched users to send messages in real time to each other. This facilitates the discussion between the 2 users as they code out a solution to the coding problem together.

This was implemented through the use of sockets. Upon being matched by the matching service, the matching service returns a unique roomID to the 2 matched users.

This roomID is then used to create a unique socket room, where subscribers to this roomID receive messages from the user who is currently sending messages. When a new message is sent to the socket in a **“newmsg”** event, the same message content is then broadcast in a **“msg2”** event to the other participant in the socket room.

This is the publisher subscriber model, where each user takes turns to both be the publisher as well as the subscriber.

Unexpected Events and Error Handling

The coding match could leave unexpectedly halfway during the coding exercise. This is handled by sockets as well. When it is detected that a socket connection has been disconnected, the socket emits a **“friendleft”** event which is then handled by the frontend.

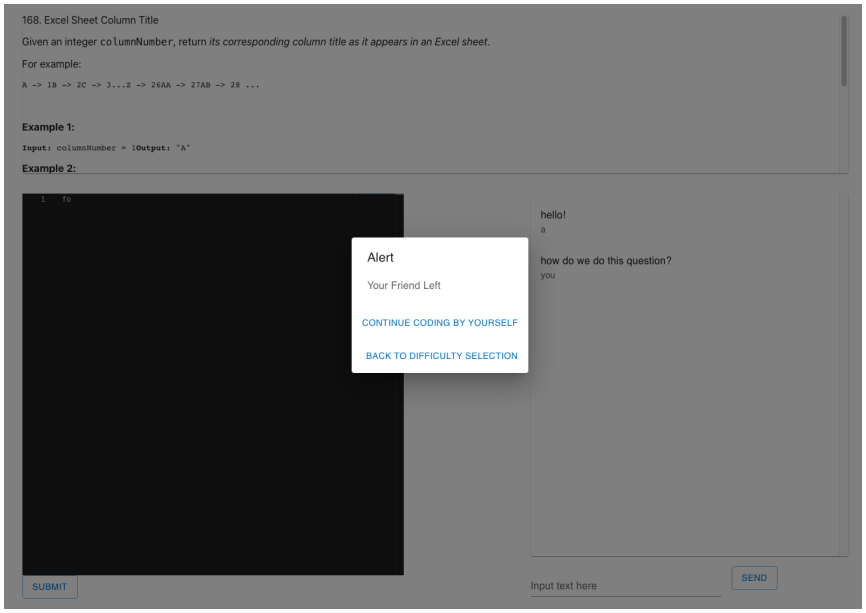


Figure 6: Error handling for match leaving the codePage

As seen from the screenshot, an option is given to the remaining participant if they would like to stay and continue tackling the question individually or go back to the difficulty screen to find a

different coding buddy.

Design Choices

Publisher Subscriber Pattern
Since we only require a single topic (roomID), this design pattern can be implemented sufficiently using socket rooms.

Benefit	Elaboration
Improve Scalability for the future	Pub-Sub allows us to increase group sizes in the future, larger groups can subscribe to the same roomID, which allows more than 2 individuals to collaborate on the same coding problem
Decoupling of Publishers and Subscribers	Producers and consumers of a message do not need to know who else is in the room. They only need to know the roomID to send and receive messages from rather than the existence of every entity in the room.

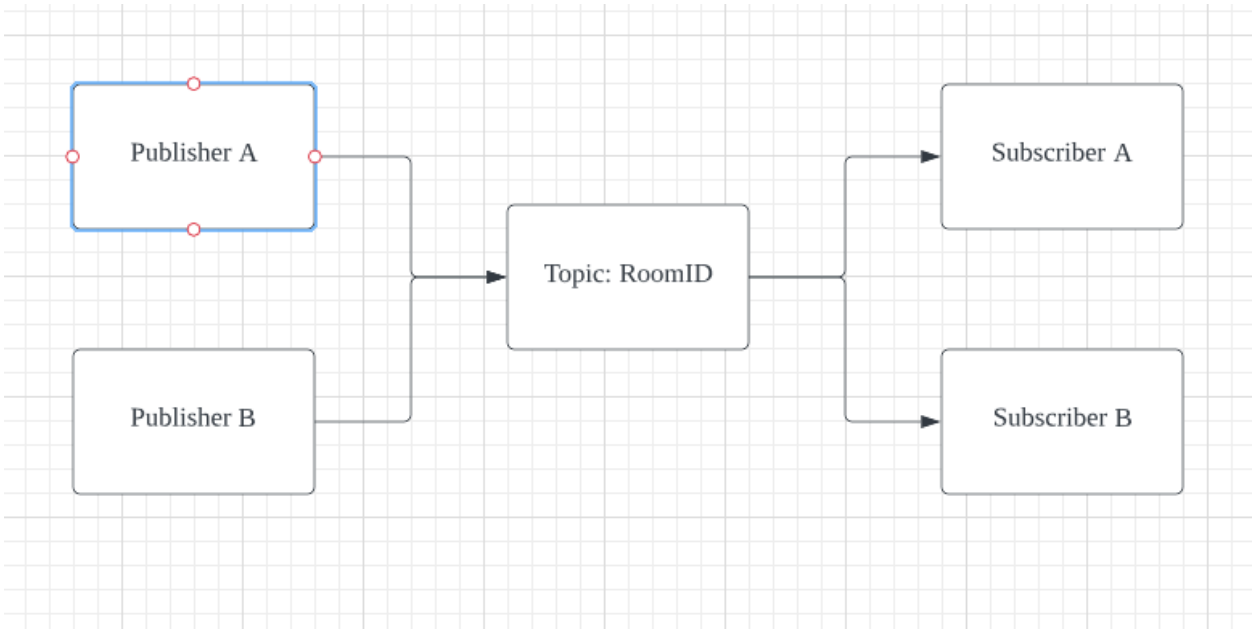


Figure 7: Rough overview of how Pub Sub Design pattern operates

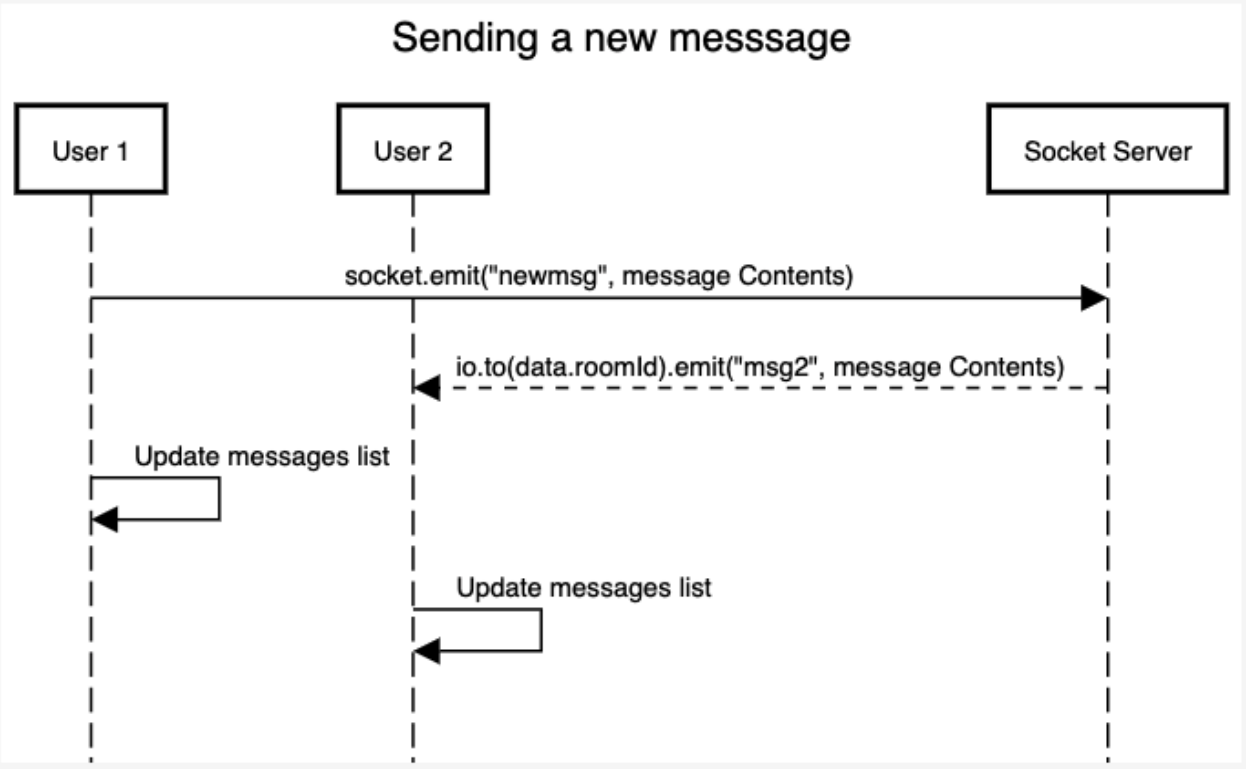


Figure 8: Sequence diagram for sending out a new message

Alternative implementations

Request & Response Model

Rather than having users “subscribe” or join a specific socket room, the client could request information from the server at short time intervals. The server would store the messages between the 2 participants.

However, it does not make sense for clients to keep making requests to the server to remain updated, even when there are no new messages being sent. This results in a lot of wasted requests being made and is not as suitable for a chat service.

Question Service

The question service is used to retrieve randomized coding questions for the users to solve. These questions have a difficulty attribute which is used to determine which question is shown to the user.

A leetcode scraper found online was used to seed the database with questions, found here. This script dumped the questions into a single html file, which was then parsed by question into the different fields in order to appropriately seed the database.

The database stores the **`questionTitle`**, **`questionBody`**, and **`difficulty`**. It made the most sense to use a SQL database here, as the schema flexibility that NoSQL offers was not necessary. Also, relational databases are more robust and consistent, which is important when designing high-quality applications.

There is a singular GET endpoint that takes in two query parameters, a seed (integer) and the difficulty (string). The seed is generated from the room id and used on the backend to determine which question is given to the user. We generate the seed by summing the ASCII values of all the characters in the room id, which ensures randomness of the question since the room id is unique. This also ensures that both users in the room get the same question, since the backend guarantees that for a given seed and difficulty, the same question will be returned.

We chose this method to ensure randomness and that both users in a room get the same question because it had low performance overhead. Alternatively, we could have stored a question id in our match table, or even created a new table mapping room id to question id. However, this would involve more database reads and writes, which introduces more performance overhead.

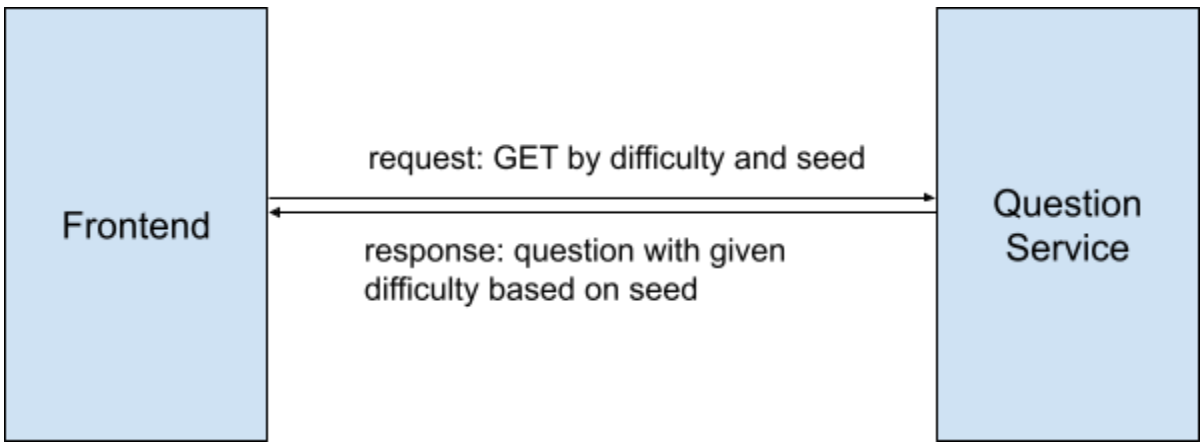


Figure 9: API calls between frontend and question service

Collaboration Service

The collaboration service is implemented using WebRTC, which enables real time communication between web browsers without the implementation of a server to transmit data. To ensure that there are no race conditions (conflicts in text edits), we use an implementation of CRDT (Conflict-free Replicated Data Types) with a Yjs wrapper library in order to transmit text data that is able to resolve itself through internal implementations in the library. Yjs also works with WebRTC technology out of the box so creating a real time collaborative text-editor was a matter of putting the two together.

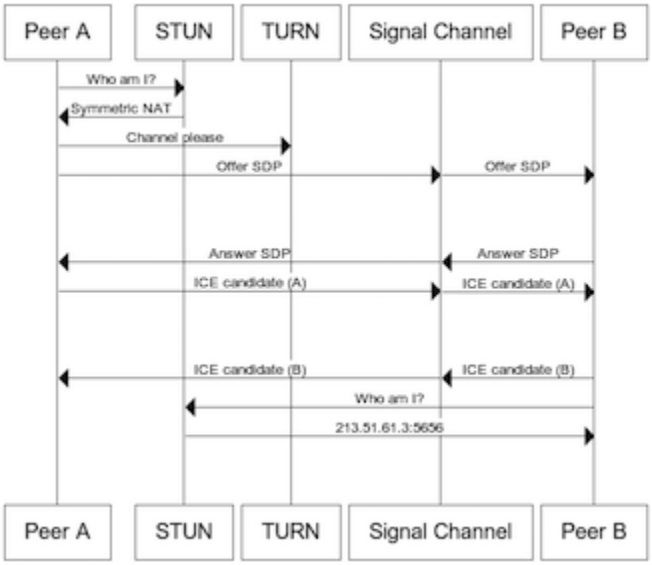


Figure 10: A traditional WebRTC setup

Traditionally, WebRTC requires signaling servers that match a waiting connection to another waiting connection, but this also requires setting up of STUN or TURN servers, which are used to establish peer connections between two wanting peers which, in our case, are two PeerPrep users. However, an easier method involves all peers connecting to all other peers. Then, to identify which code from the code editor they are editing, the data is stored within a dictionary with the keys being their unique roomIds and the content being the text in the code editor. This ensures that only the peers belonging to the same room are able to edit their code. The chance of key collision is very small, since our roomIds are 40 characters long as they are built from appending two socketIds together (randomly generated 20 character strings that are guaranteed to be unique) the process can be found below from matching to establishing a connection for the editor:

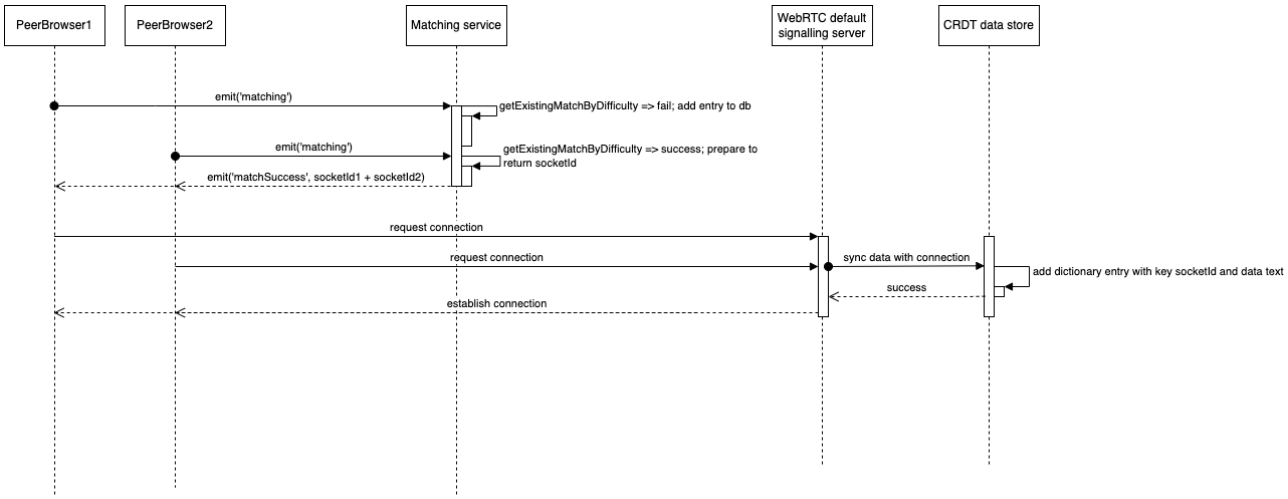


Figure 10: Activity diagram for matching and collaborative editing

Future improvements

- Adding roles to users such as a superuser or a subscription user so that they can match with other subscription users and access questions that are more premium. This can be done with authorization implementation in the user service by attaching roles to user in the mongoDB schema
- Add the ability to collaborate with multiple users (“mob programming” style instead of “pair programming”)
- Add video or voice call functionalities to the application. Both can be built upon WebRTC technologies as well and should not be hard to extend upon the existing code
- Allow users to run and compile their code in the code editor. This is nice to have as some coding interviews do not allow the code to be run during the coding interviews and forcing the students to talk out their code execution is actually better.
- Adding a record of question history to the app so users won’t end up redoing the same questions over and over again.
- Add the ability for User to specify a username to match with so that they are able to code with their friends.
- Recommended solutions can also be provided for the user in the future

Work Distribution

Name	Contributions
Ng Seng Leng	● developed the logic for logging in and signing out a user ● developed the logic for deleting a user and changing the password of a user ● implemented the hashing of passwords before storing entry into mongoDB ● implemented jwt tokens for authenticating requests to user service and for persisting user sessions ● implemented simple user session persistence ● implemented a collaborative code editor with WebRTC and CDRT on the frontend to ensure no race conditions when editing simultaneously ● implemented the ReactContext data store on frontend for data manipulation and storage ● added dropdown menu for option to logout, change password and delete account on frontend ● added documentation for user service and collaboration service ● contributed to weekly sprint meetings
Neha George	● Contributed to weekly sprint meetings ● Created the question service that allows the frontend to randomly get questions by difficulty for a given room id ● Wrote a python script parses questions from leetcode in order to populate the question service's database

	● Created the matching service that uses sockets to connect two users that select the same difficulty, and times out otherwise after 30 seconds ● Added sqlite database connection for use in matching service and question service
Isaac Tan	● Created the difficulty/main home screen ● Implemented frontend sockets for matching service ● Implemented the countdown timer on the frontend ● Implemented Scrollable Chat box with sockets for codepage ● Implemented the backend chat service through a unique roomID. ● added documentation for functional requirements and non functional requirements and chat service ● contributed to weekly sprint meetings

Reflections

Neha: As an exchange student, I really appreciated the opportunity to work with local Singaporean students. It made me appreciate that software development and processes are universal in our ever so globalized world. I also enjoyed the ability to make our own design decisions and conduct sprint meetings where we had leadership roles. In my internship experience, I am not always experienced enough to have this much agency.

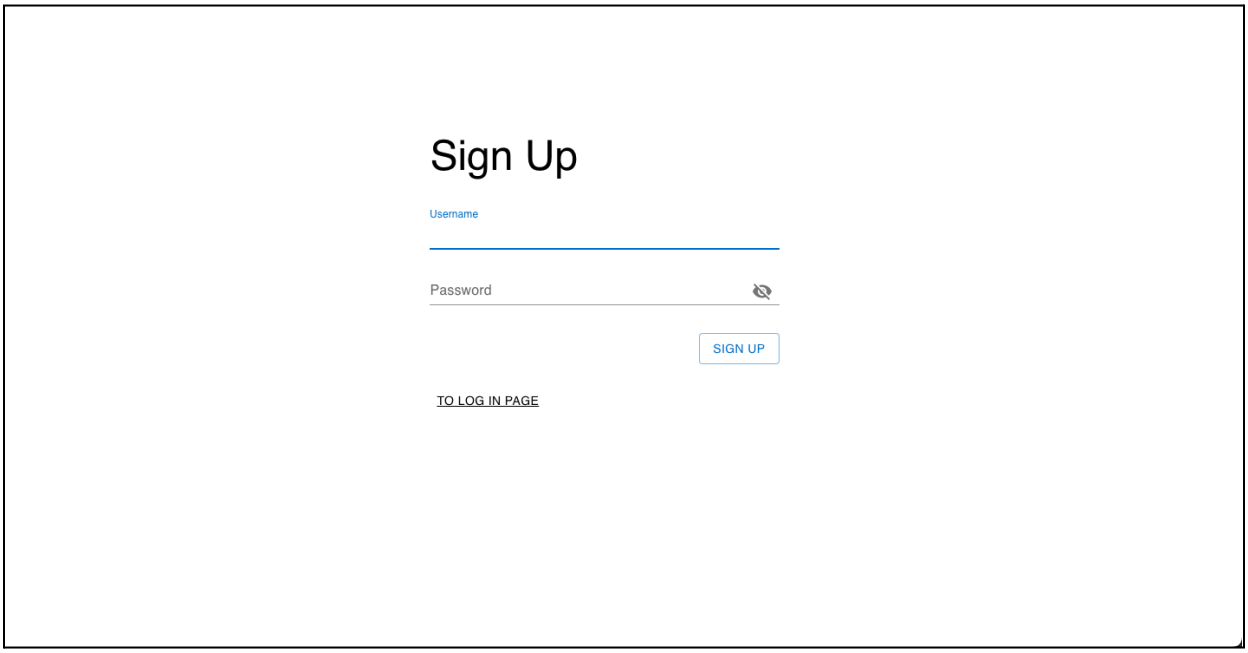
Isaac: It is the first time that I am working on the frontend for a project. I think that my knowledge of software engineering was quite limited before this module, so it was lucky that my teammates were willing to help guide me throughout the project. I think that it was an interesting experience that helped me to apply some of the concepts taught in the lectures like design patterns in a more practical context.

Seng Leng: As someone who has no prior experience with backend development, this project was an informative one and allowed me to pick up skills relevant to crafting APIs, as well as

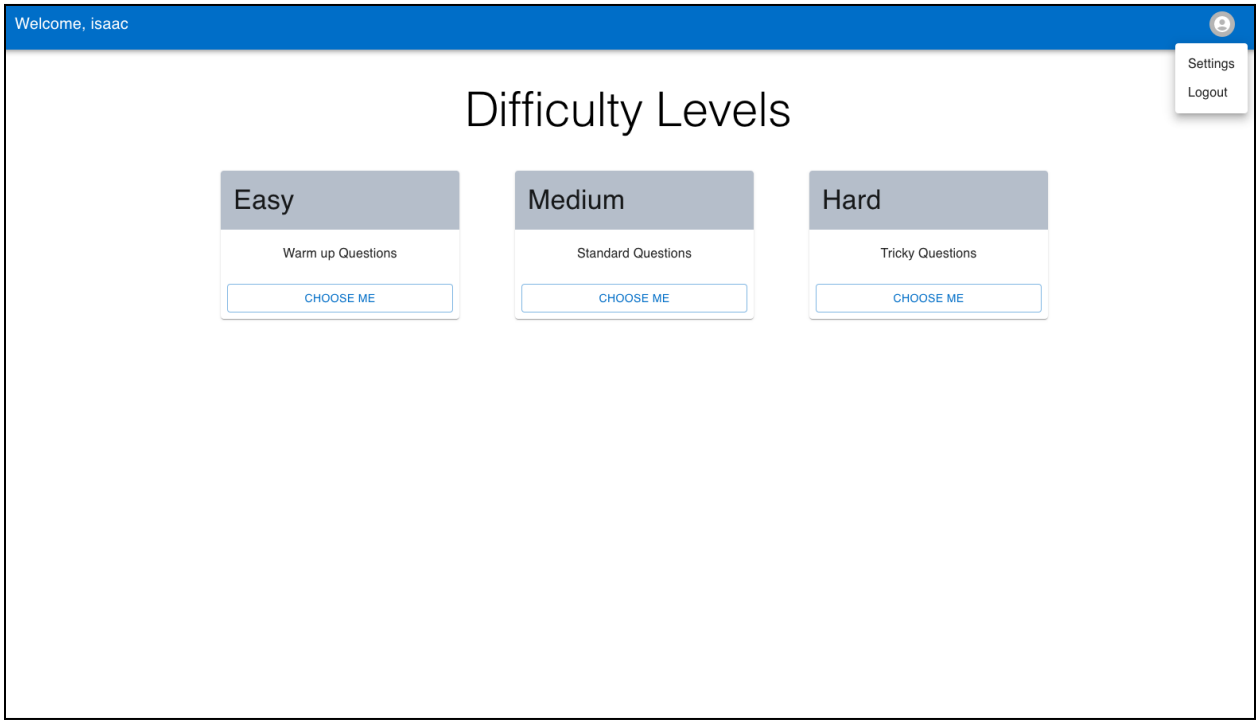
working with databases as well as having a high level understanding of basic backend deployments, even though we did not manage to deploy in the end. I think the lack of direction given by the teaching team gave us more freedom in developing our application, but I would like to have seen more enforcing of content being applied on our application.

Appendix

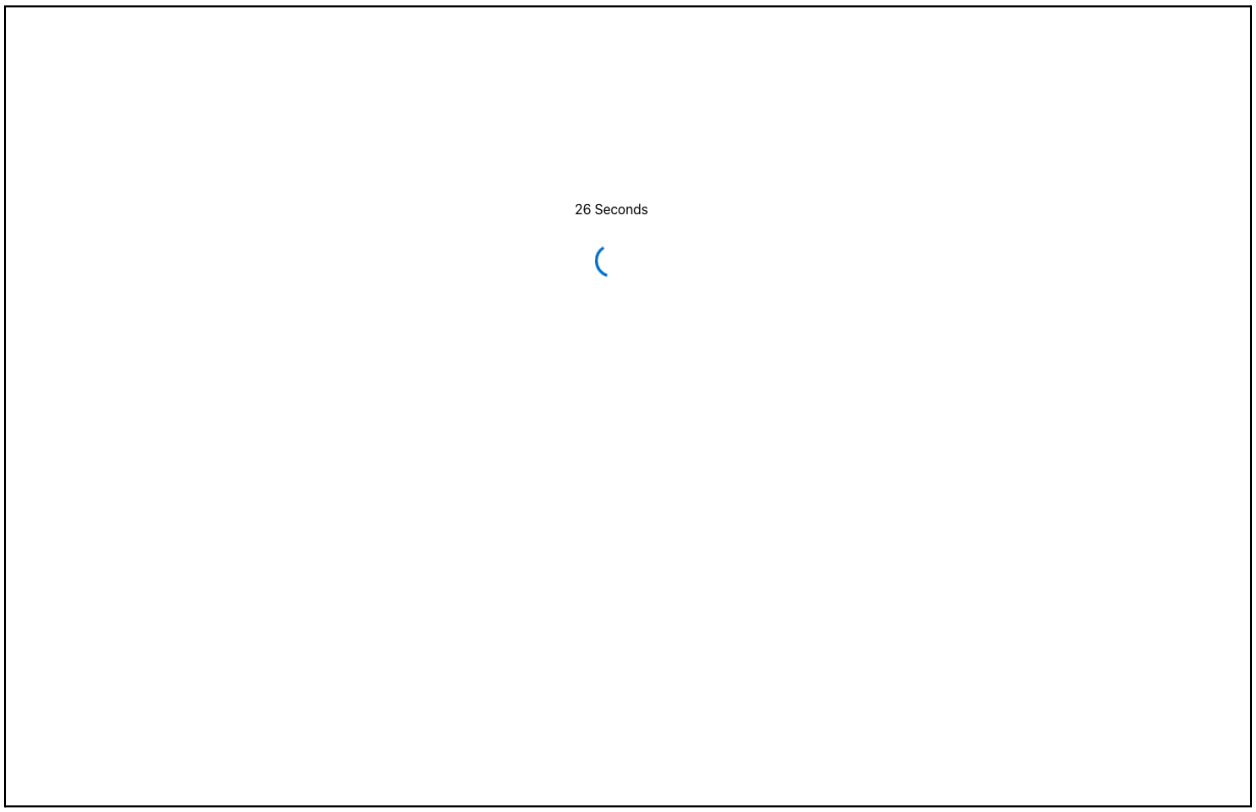
Signing up and logging in



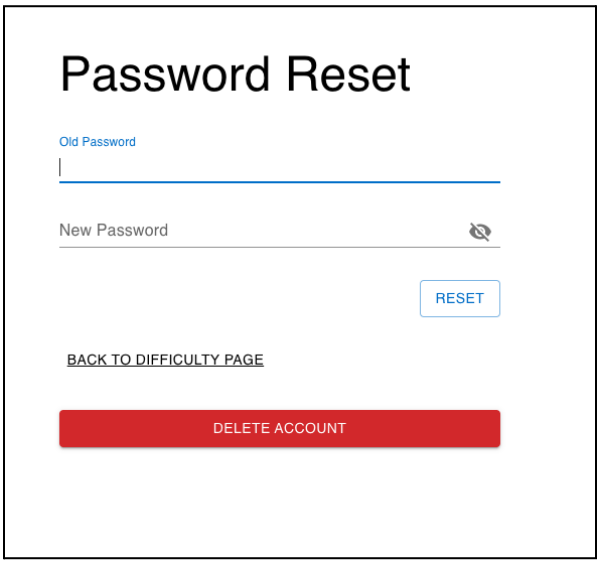
Main Page



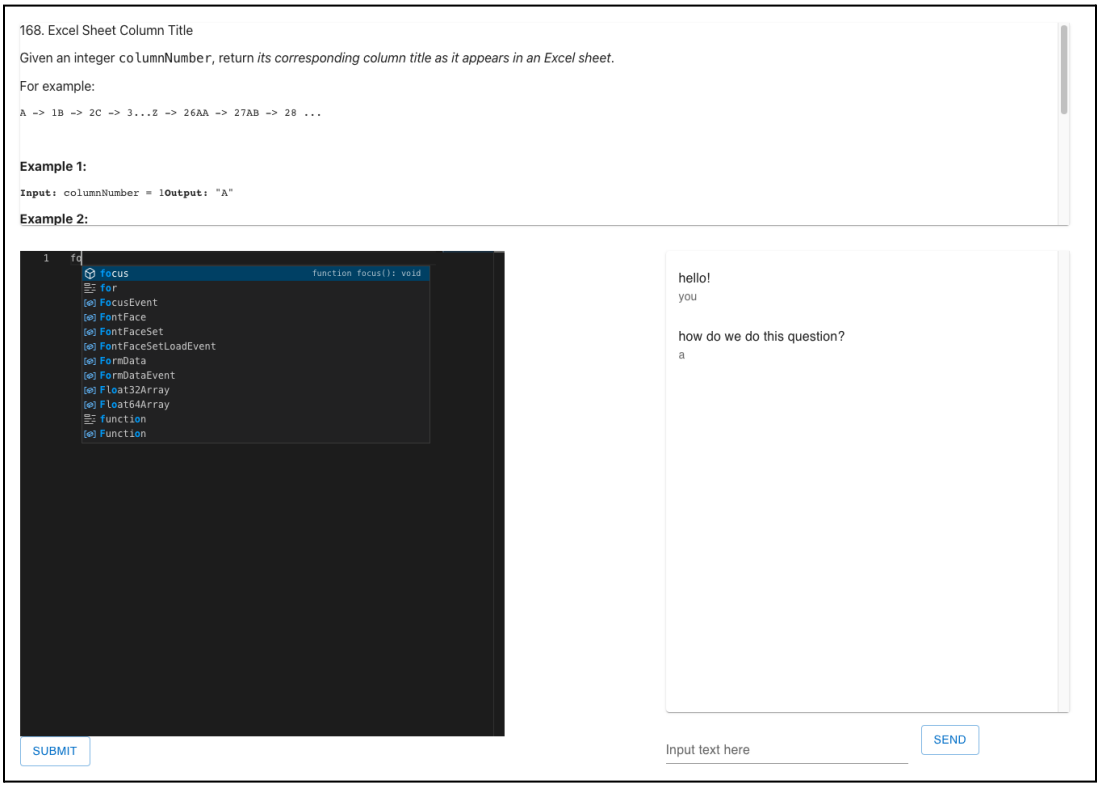
Loading Screen



Forget Password Screen



Collab Page


168. Excel Sheet Column Title
Given an integer columnNumber, return its corresponding column title as it appears in an Excel sheet.
For example:
A -> 1B -> 2C -> 3...Z -> 26AA -> 27AB -> 28 ...
Example 1:
Input: columnNumber = 1Output: "A"
Example 2:
1 fo
focus function focus(): void
for
FocusEvent
FontFace
FontFaceSet
FontFaceSetLoadEvent
FormData
FormDataEvent
Float32Array
Float64Array
function
Function
SUBMIT
hello!
you
how do we do this question?
a
Input text here
SEND