CS3219

AY 2023/2024 Semester 1

Group Project - Peerprep

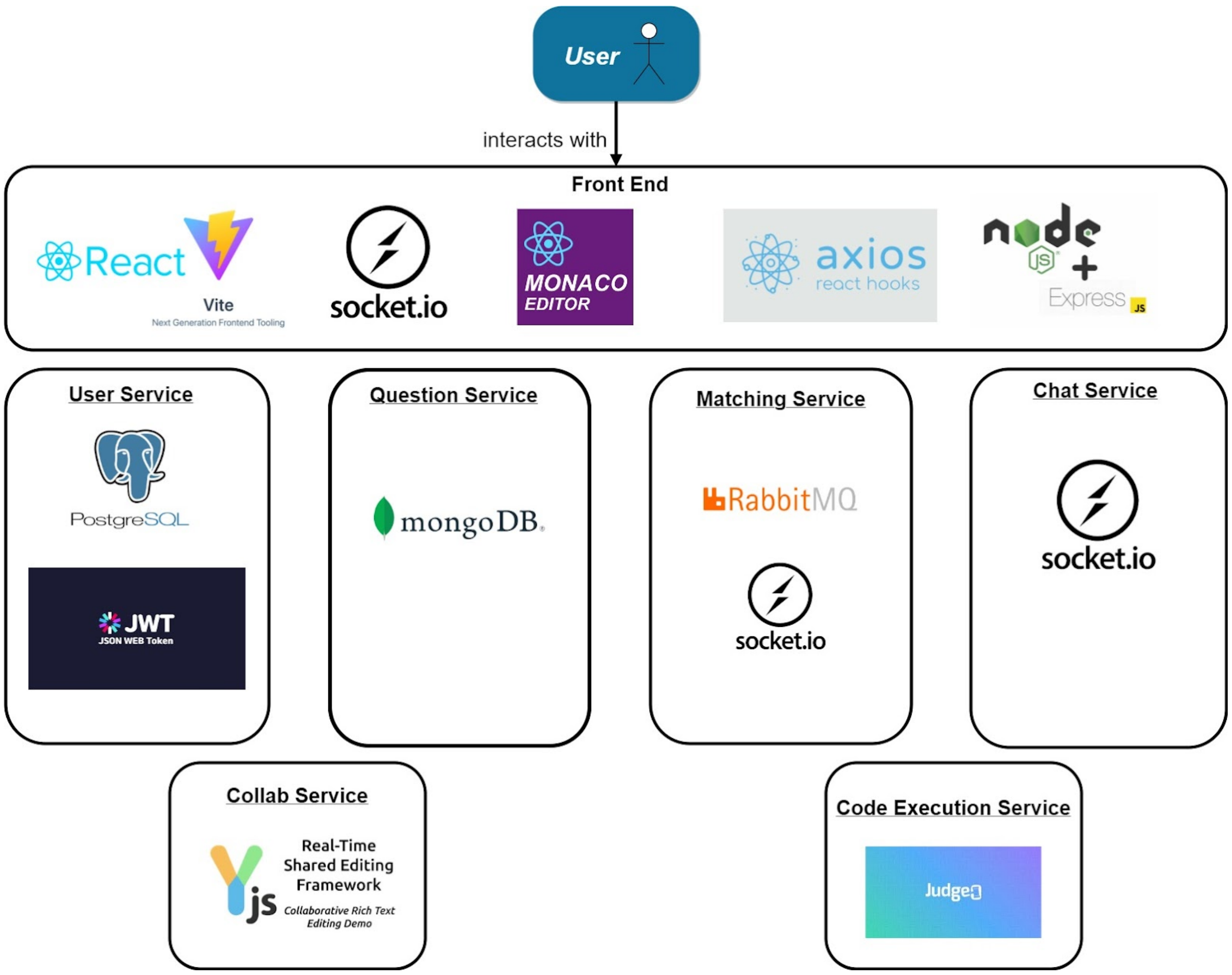
Group: 44

Full Name	Matriculation Number
Sheryl Kong	A0240686Y
Justin Cheng Linzhe	A0234455E
Lee Zi Yang	A0233436J
Khoo Jing Hong, Derrick	A0233877R
Nicholas Tan Jun Kai	A0234219J

Table of Contents

Design Documentation

Architecture and Tech Stack



As seen from the above diagram, our software architecture uses a microservice architecture, where we have 6 distinct microservices, the question service, matching service, user service collaboration service, chat service and code execution service, along with a front-end that pulls data from the relevant microservices.

Architectural Decision: Monolith vs Microservice

We decided that a microservice architecture suited our needs more than a Web MVC Layered Application (Also known as a Monolith). We wanted our system to be easily scalable and loosely coupled.

Scalability

A microservice architecture is easily scalable as each microservice contains its own independent mini-ecosystem of appropriate resources separate from other microservices. A Web MVC Layered application will not be as easily horizontally scalable as the entire monolith must be duplicated when a particular service inside the monolith is under heavy load which is more expensive and less responsive. On the other hand, for a microservice architecture, each microservice can be independently scaled according to demand with relatively low latency with orchestration services such as Kubernetes or other AWS/Google Cloud Services.

Separation of Concerns

We also chose a microservice architecture as it reduces the most harmful form of coupling, implementation coupling and enforces Separation of Concerns at an architectural level. Since each microservice handles different logic and is worked on by a different programmer, reduced implementation coupling allows for easier testing and debugging of the program as it can be broken down into small components and tested in isolation.

Backend Microservices

User Service

The user service's purpose is to enable users to sign in before using the application so that their progress can be saved and their identity can be established. New users would be able to register their credentials, as well as sign in to their existing accounts.

User backend

To manage our backend, our group has chosen to use PostgreSQL as our database. The SQL table can be hosted on the cloud but due to budget constraints, it is only run locally and managed by PGAdmin4. Our group has also chosen express.js as our framework for our REST API calls.

Components

index.js

This file is the entry of the user backend service. We can start the backend server, allowing it to listen to incoming HTTP requests by users.

server.js

In this file, we use the 'pg' library in a node.js environment to establish a connection with our local PostgreSQL database. This allows us to make SQL queries to our database easily.

validation.js

This file is responsible for the validation of new users who want to register their accounts. More specifically, the methods in this file can check if a certain email is already in the database or if a username has already been taken by someone else, by performing SQL queries to our database. We utilize these functions in various API endpoints.

users.js

We define the routes for the API endpoints here.

For routing purposes, the API endpoints for user backend starts with /api/users. Below is a summary of what each API endpoint does.

API Endpoints

User CRUD Operations:

HTTP request type	**Endpoint**	**Mechanism**
GET	/getUser/:token	Response: User data in the form of a JSON object 1. Retrieve the JWT token that is passed as a parameter named \`*token*\` into the route. 2. Validate the token's authenticity and extract its payload 3. Retrieve email information embedded in the token's payload 4. Perform a PostgreSQL query to fetch user data based on the email address extracted. 5. Returns the user data fetched from the database.
POST	/createUser	Response: Success message or error message in case of duplicate email or username 1. Extract necessary information, including username, email, and password from the request body. 2. Verify if the provided email address already exists in the database. 3. Verify if the username is already in use. 4. ERROR: API returns an appropriate error message with a 401 status code for duplicate error 5. SUCCESS: Generate a new unique identifier (newId) using uuidv4() and insert a new user entry into the database. Responds with a 201 status code.
PATCH	/updateUser/:token	Response: Success message or error message in case of duplicate username or token expiration 1. Retrieve the JWT token that is passed as a parameter named \`*token*\` into the route. 2. Validate the token's authenticity and extract its

		payload 3. Extracts the user's information to be updated based on the provided data in the request body. 4. Checks if the updated username already exists in the database. 5. Checks if a duplicate username is found in the database 6. ERROR: Returns an error message with a 401 status code for duplicate error and 403 status code if authentication token is expired 7. SUCCESS: Executes an update query in the Users table with the modified data. If successful, responds with a 201 status code and a message
DELETE	/deleteUser/:token	Response: Success message or error message in case of user not found or token expiration 1. Retrieve the JWT token that is passed as a parameter named `*token*` into the route. 2. Validate the token's authenticity and extract its payload 3. Decode token and obtain the user's email 4. Perform a SQL DELETE query on the Users table 5. ERROR: Responds with a 403 status code if token expired and 404 status code if user not found 6. SUCCESS: Responds with a 200 status code and a message indicating that the user has been successfully deleted

Other user operations:

POST	/loginUser	Response: User data in the form of a JSON object, with the user’s email, password, username, access token, role and completed questions 1. Retrieve the user’s email and password that is entered by them in the login page, and is subsequently passed into the route during the HTTP request 2. Check if the user’s email is existing in the database, if it is not in the database, send a response with status 401, informing the user that his email is not registered. 3. If the user entered an incorrect password, send a response with status 401, informing the user that his password is incorrect

		4. Perform an SQL query to retrieve the user’s information from the database, which includes the username, role and completed questions 5. Sign a JWT token using the user’s username and email 6. Send a response with status 200, with the appropriate user data returned in JSON
POST	/isUserOrAdmin	Response: Authorization message indicating user role or unauthorized status 1. Retrieve user role by executing query on the Users table to fetch the role 2. Authorized access: If the user is identified as an "admin," the API responds with a 200 status code. 3. Unauthorized access: If the user is not an "admin," the API responds with a 401 status code 4. This API is to ensure that only authenticated users (admin accounts) have the ability to perform CRUD operations on questions

Question-related operations:

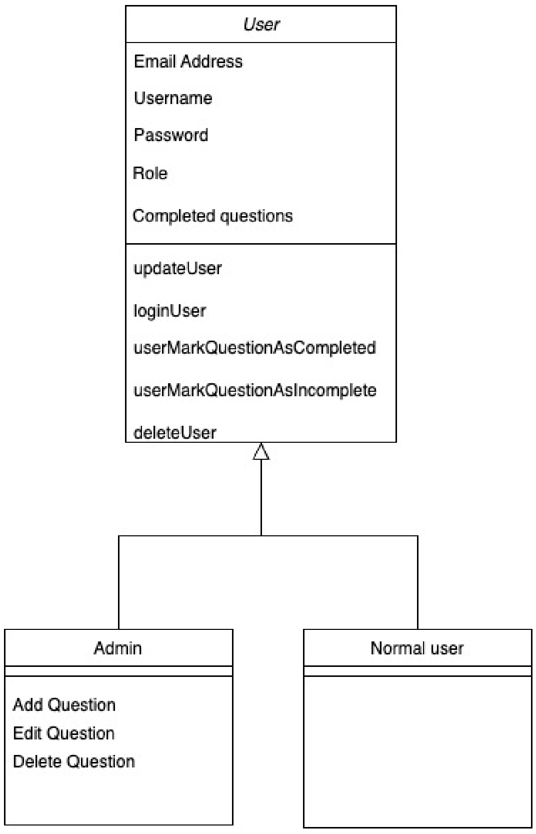
POST	/userMarkQuestionAsCompleted	Response: Success message or error message in the case where the question cannot be marked as completed. 1. Extract the user’s email and the question id from the request body. 2. Perform an SQL query to find the user’s entry and add the question id to the list of completed questions in the user’s entry. 3. ERROR: Responds with an error message and the appropriate status code. 4. SUCCESS: Responds with status code 201 with a message showing that the question has successfully been marked as completed.
POST	/userMarkQuestionAsIncomplete	Response: Success message or error message in the case where the question cannot be marked as incomplete. 1. Extract the user’s email and the question id from the request body. 2. Perform an SQL query to find the user’s entry and remove the question id to the list of completed questions in the user’s entry. 3. ERROR: Responds with an error message and

		the appropriate status code. 4. SUCCESS: Responds with status code 201 with a message showing that the question has successfully been marked as incomplete.
POST	/submitAttempt	Response: Success message or error message in the case where the attempt cannot be submitted. 1. Extract the necessary information, including the user's email, question id, date, code, and language from the request body. 2. Perform an SQL query to insert an attempt entry to the "attempts" table of the database with the extracted details. 3. ERROR: Responds with an error message and the appropriate status code. 4. SUCCESS: Responds with status code 201 with a message showing that the question has successfully been marked as incomplete.
GET	/getAttempts	Response: A list of attempts made by the user and an error message in the case where the attempts cannot be retrieved. 1. Extract the user's email from the request parameters. 2. Perform an SQL query to retrieve all entries from the attempts table with matching email addresses. 3. ERROR: Responds with an error message and the appropriate status code. 4. SUCCESS: Responds with status code 200 and returns the list of retrieved attempts.
GET	/getQuestionsAttemptedPerUser	Response: A list of objects containing information about each user and the number of unique questions that they have attempted. 1. Perform an SQL query to retrieve the number of unique questions each user has attempted. 2. ERROR: Responds with an error message and the appropriate status code. 3. SUCCESS: Responds with status code 200 and returns the list of objects with the retrieved information.
GET	/getAttemptsPerQuestion	Response: A list of objects containing information about each attempted question and the number of unique users which have made attempts. 1. Perform an SQL query to retrieve the number of unique users who have attempted each

		question. 2. ERROR: Responds with an error message and the appropriate status code. 3. SUCCESS: Responds with status code 200 and returns the list of objects with the retrieved information.
GET	/getLanguageUsage	Response: A list of objects containing information about the frequency of each language used across all attempts. 1. Perform an SQL query to retrieve the count of each language across all attempts. 4. ERROR: Responds with an error message and the appropriate status code. 5. SUCCESS: Responds with status code 200 and returns the list of objects with the retrieved information.

Database Model

Note: User Service uses PostgreSQL as the database.



Model: User		
Fields	Attributes	Description
id	• type: String • required: true • unique: true	UUID of the user that is auto-generated whenever a user account is created
email_address	• type: String • required: true • unique: true	Primary key, cannot be changed by user once account is created.
username	• type: String • required: true	Username of the account, set by the user.
password	• type: String • required: true	Password of the account set by the user.
role	• type: String • required: true	Represents access permissions of the user: “Admin”: User will be given the rights to create, update or delete the questions in the questions repo. “User”: User will be able to view questions but will not be able to create, update or delete questions in the repo.
completed_questions	• type: Array of strings • required: true	Keeps track of all questions completed by the user. Each string in the array corresponds to the _id of a question.

Services

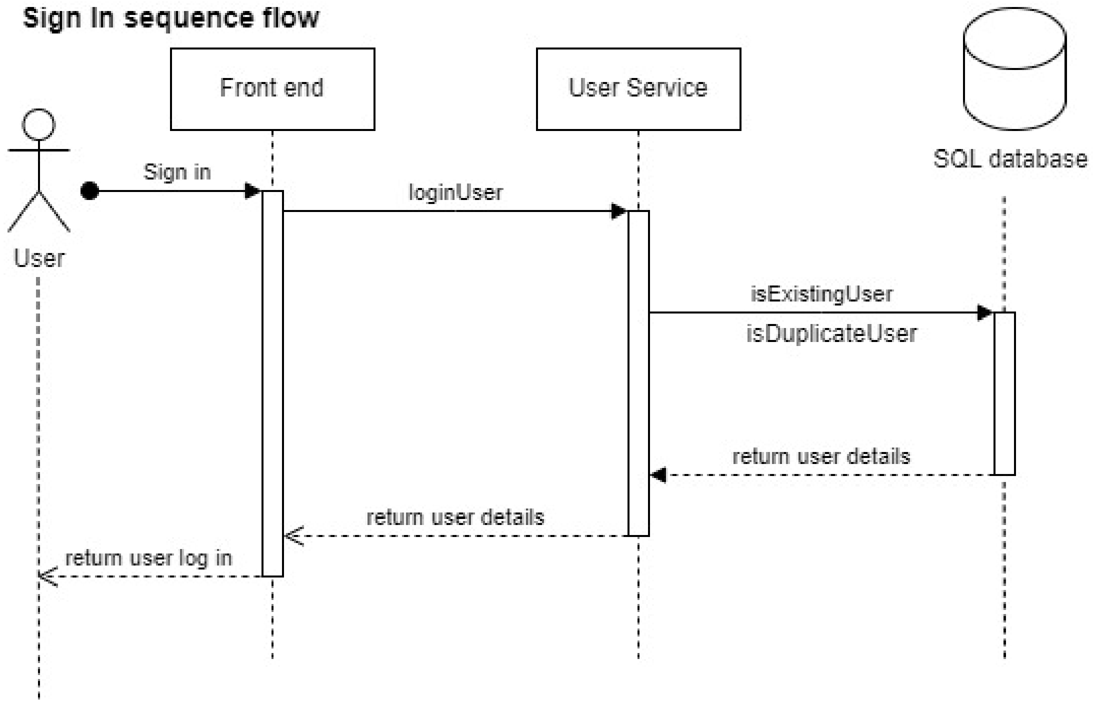
For the intermediary between our UI and backend, we have decided to implement ‘user_services.js. In this file, we use axios as the framework to hit the aforementioned API endpoints with HTTP requests.
When a relevant button is pushed during the login/register stage, we will trigger the functions in ‘user_services.js’, which will subsequently hit the endpoints in our user backend.

Our group has also made use of another file ‘Utils.js’, which is responsible for checking the validity of a user’s JWT token. In this function, we will call the /getUser API endpoint to verify the user. This function is triggered every time we render a page in our application. Should a user have an expired JWT token, he/she will be logged out immediately.

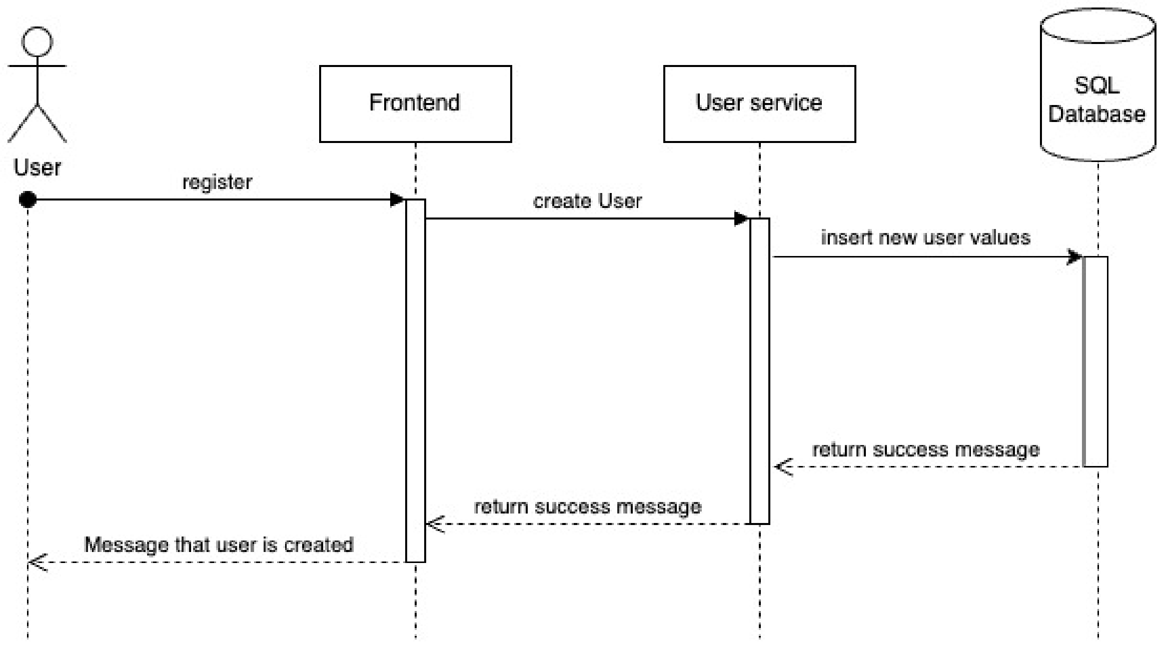
Internal Sequence Flow

The sequence flow diagrams for a user signing in or signing up is shown below:

Sign In sequence flow



Sign up sequence flow



Matching Service

The matching service's purpose is to enable users to select the difficulty of questions they want to attempt and get matched with other users who selected the same difficulty.

Components

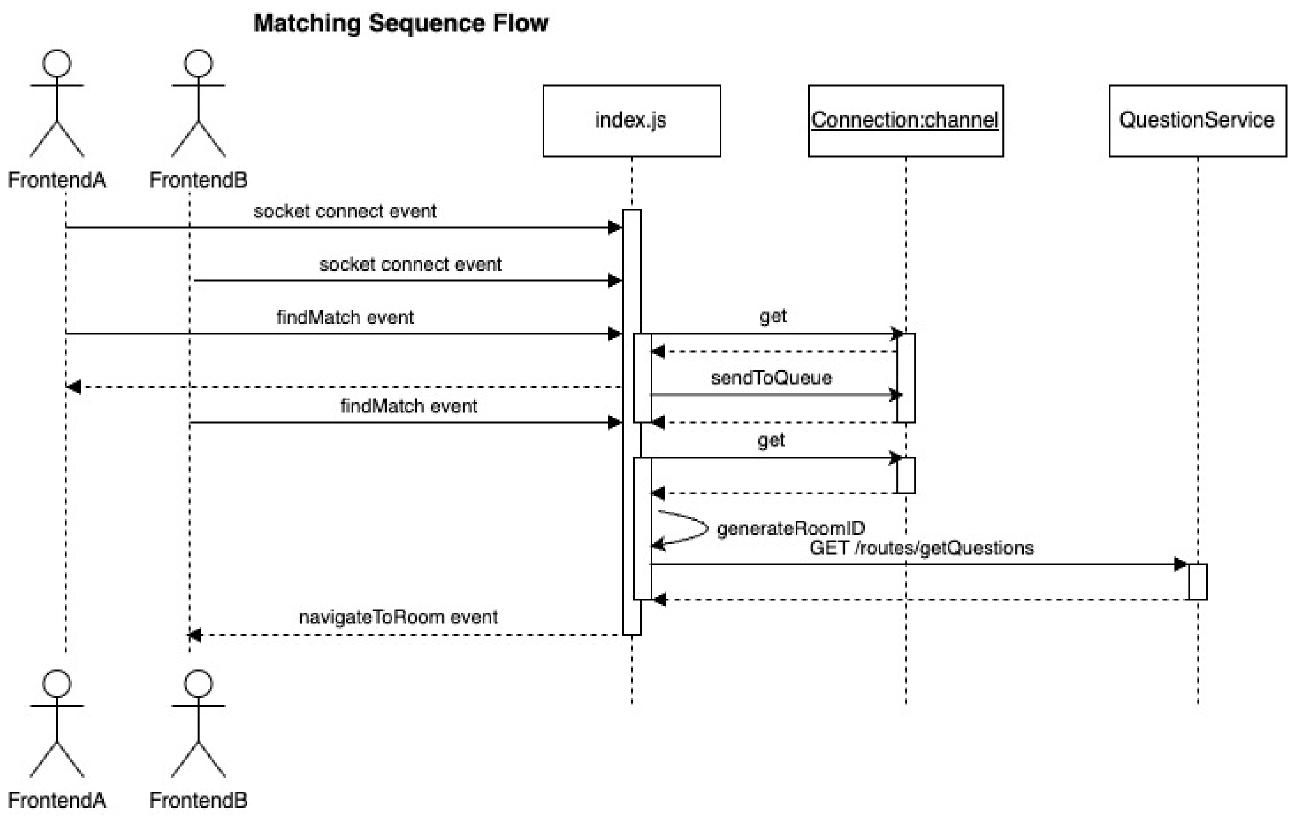
Index.js

The index.js module contains all logic for the use of WebSockets with the external library Socket.io in order to communicate in real time with the frontend, for e.g to get a match. The queueing client logic is handled in index.js, while RabbitMQ handles the main logic on the server side.

External Library: Socket.io and RabbitMQ
A Socket.io server socket is created in index.js to serve as the socket for the backend, and when clients request to match with another client, a client side Socket.io socket is created.
First, 3 RabbitMQ queues corresponding to the difficulties, 'Easy', 'Medium' and 'Hard' are created. Then, when the client requests to match with another client, the server would poll the respective queue and check if the queue is empty. If the queue is empty, the server would just send a message to the respective queue based on the difficulty that the client has selected. Otherwise, if the queue is not empty, the server will retrieve the message from the queue and the server socket will emit a message to both client sockets using their respective SocketID, which will trigger both users to be navigated to the same room.

Internal Sequence Flow

The average use case for the matching-service will be two frontend clients logging in and the two different users will each request a match to the same difficulty: Easy, Medium or Hard. How the matching-service handles this request is shown in the sequence diagram below.



Collaboration Service

The collaboration service employs *yjs* websocket to enable different clients to be able to persistently and synchronously code and communicate with each other. It serves one main function: Synchronisation of the code editors between the matched users.

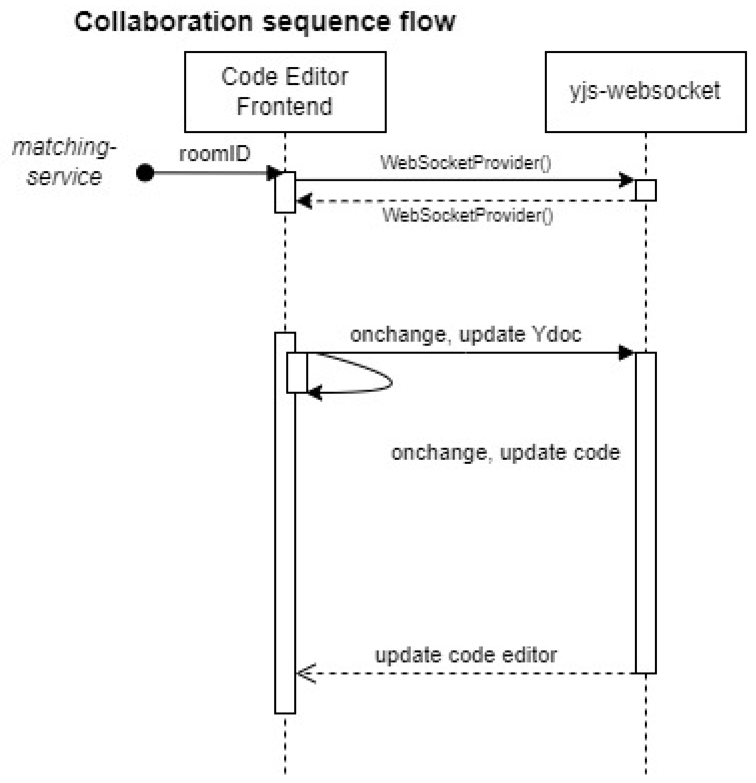
Components

There are no components in the yjs web-socket and it is a node-module.

<u>External Library: yjs websocket</u>
It is run by using npx y-websocket. This results in the opening of a local port. The matching-service will provide a uniquely generated room id for both parties, and the both will connect to a port with the special room id meaning that no one else can enter that room.

<u>External Library: Monaco Editor for React</u>
A library that is compatible with yjs websocket and also syntax highlighting for multiple languages.

Internal Sequence Flow



Question Service

The question service's purpose is to enable users to obtain questions from the question bank to be used to practice programming. It should also provide admins with the ability to add, update or delete questions.

It is a simple REST API connected to a database that enables basic question fetching operations.

Components

questionModel.js

questionModel.js contains the Schema created via Mongoose. It manages the data structure and logic of the question-service. The database model can be found below in a later section.

questionController.js

questionController contains the functions employed by the Question Service. It takes in the user's input and converts it to commands for the questionModel. It also manages sending responses to the user.

API Endpoints

The question service utilizes various API endpoints to perform CRUD operations. The API endpoints, as well as their details, are listed in the table below.

HTTP request type	**Endpoint**	**Mechanism**
GET	/getQuestions	Response: List of questions and their associated data in the form of a JSON object 1. Perform a MongoDB query operation to fetch all questions from the question database. 2. Returns the question data fetched from the database. 3. ERROR: Responds with a 500 status code and an error message. 4. SUCCESS: Responds with a 200 status code and a message.
POST	/addQuestion	Response: Success message or error message in case of duplicate question title. 1. Extract necessary information, including title, description, category and difficulty from the request body. 2. Verify that a question with an identical title (case-insensitive) does not already exist in the database. 3. ERROR: API returns an appropriate error message with a 500 status code. 4. SUCCESS: Generate a new unique _id insert a new question entry into the database. Responds with a 201 status code.
DELETE	/deleteQuestion	Response: Success message or error message in case of question not found. 1. Extract necessary information, including title, description, category and difficulty from the request body. 2. Create a temporary question object containing information of the question to be deleted. 3. Perform a MongoDB query to find and delete the question with attributes matching the temporary

		question object. 4. ERROR: API returns an appropriate error message with a 500 status code. 5. SUCCESS: Question gets deleted from the database. Responds with a 200 status code.
UPDATE	/updateQuestion	Response: Success message or error message should update fail. 1. Extract necessary information, including _id, title, description, category and difficulty from the request body. 6. Create a temporary question object containing information of the question to be deleted. 7. Perform a MongoDB query to find the question with a matching question _id. 8. Updates the question by replacing the rest of the fields with the fields in the temporary object. 9. ERROR: API returns an appropriate error message with a 500 status code. 10. SUCCESS: Question gets updated. Responds with a 200 status code.

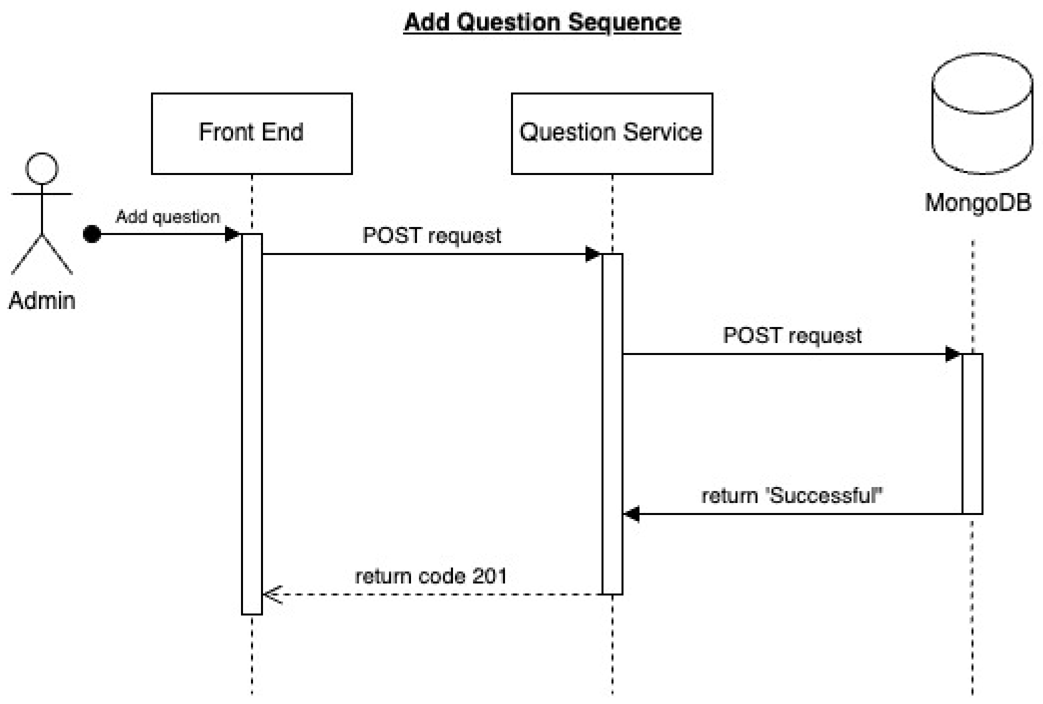
Database Models

Below is the database model used for the Question Service which as mentioned before.

Model: Question		
Fields	**Attributes**	**Description**
_id	• type: String • required: true • unique: true	uuid of the question (this field is auto-generated whenever a new question is created)
title	• type: String • required: true • unique: true	Title of the question
description	• type: String • required: true	Details of the questions including examples and constraints (if any)
category	• type: String • required: true	Used to group/tag questions.
difficulty	• type: String • required: true	Difficulty that of the question: EASY, MEDIUM or HARD

Internal Sequence Flow

This sequence diagram depicts how a use case of the Question Service would look like:



Chatbox Service (N1)

The chatbox service's purpose is to enable users who are matched and assigned to the same session room to communicate via text messages.

Overall Microservice Architecture:

The microservice consists of both backend and frontend components. The backend establishes socket connections, handles message transmission, manages user interactions, and maintains the chat state. The frontend provides a user interface for sending and displaying messages, interacting with the chatbox, and communicating with the backend via socket connections.

This microservice architecture facilitates real-time communication between users through a chat box interface, utilizing socket connections to exchange messages and manage user interactions in a seamless and responsive manner.

Components

Backend components

chat.js

Description: Handles socket connections, message handling, and user interactions.

1. Manages socket connections, message handling, room joining/leaving, and user disconnection.
2. Establishes socket connections and manages events like connection, getMessages, message, disconnect, connect_error, leaveRoom, and joinRoom.

index.js

Description: Entry file for the backend. Imports the chat module and sets up the server.

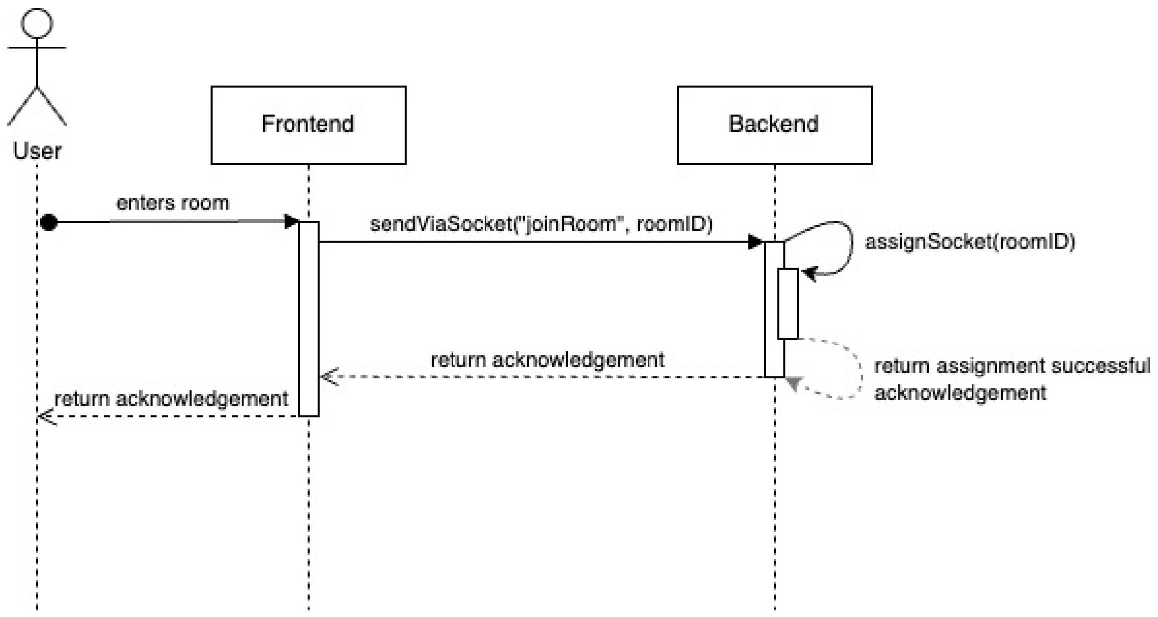
1. Imports and initializes the chat module.
2. Set up the server to listen on port 3000.

Internal Sequence Flow

<u>Join Room Request</u>

The first use case for the chatbox-service will be two frontend clients joining the same chatroom (upon being matched by our matching-service). The respective client sockets will be assigned to the same room based on the ID of the matched room they are in.

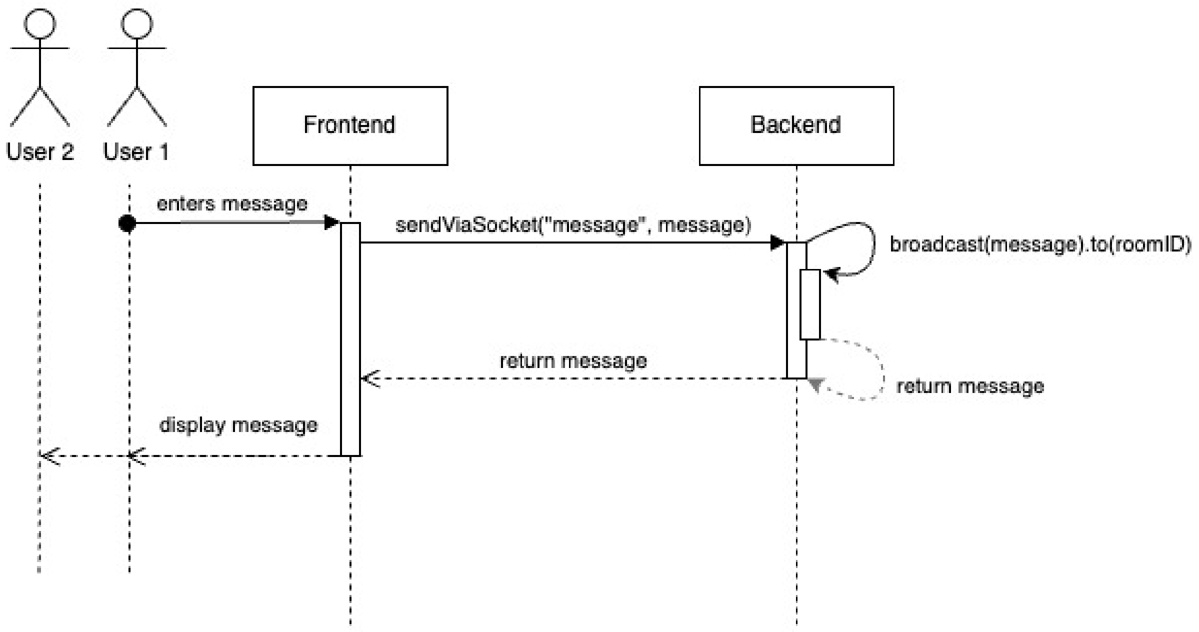
How the chatbox-service handles this 'joinRoom' request is shown in the sequence diagram below.



Send Message Request

The second use case for the chatbox-service will be when one of the frontend clients sends a message. The client's socket will send a request to the server, who will then handle the request by broadcasting the message to the other client socket in the same room.

How the chat box service handles this 'message' request is shown in the sequence diagram below.



Code Execution Service (N3)

The code execution service is to enable users to compile and run their code, which will print out any outputs, be it a stdout output or a stderr error. Similar to our user service, we use REST APIs to send HTTP requests to an external server for code compilation.

Overall Microservice Architecture:

The microservice consists of both backend and frontend components. The backend establishes HTTP connections with the external server for compilation, and retrieves the responses from the external server. The frontend provides a user interface for compiling code and displaying the output.

Components

index.js

This is the entry file for the backend, which sets up the server which listens for incoming HTTP requests.

The starting point of the routes is /routes/codeExecution.

codeExecutorController.js

This file defines the logic behind each endpoint. This code uses the axios library to send HTTP requests to the external code-executing Judge API server.

Below are the descriptions of the different asynchronous functions used:

getLanguages:

- Retrieves a list of supported programming languages.
- Utilizes Axios to make a GET request to the specified API endpoint.
- Responds with retrieved language information or a server error message.

getLanguage:

- Retrieves information about a specific programming language based on the provided language ID in the request params.
- Utilizes Axios to make a GET request to the API endpoint with the specified language ID.
- Responds with the retrieved language data or a server error message.

createSubmission:

- Creates a code submission for execution.
- Expects language ID, source code, and optional standard input (stdin) from the request body.
- Utilizes Axios to make a POST request to the API endpoint with the provided submission details.
- Responds with a token of the current submission or a server error message.

getSubmission:

- Retrieves the output of a code submission based on the provided submission token in the request params.
- Utilizes Axios to make a GET request to the API endpoint with the specified submission token.
- Responds with the retrieved code output or a server error message.

In order to show the result of a compiled code to a user, we would have to call ‘createSubmission’, followed by ‘getSubmission’ of the token of the submission that we want to retrieve. Subsequently, we can obtain the exact output through some manipulation of the JSON response.

codeExecution.js

This code defines the routes of the endpoint so that we are able to send HTTP request messages to our backend server.

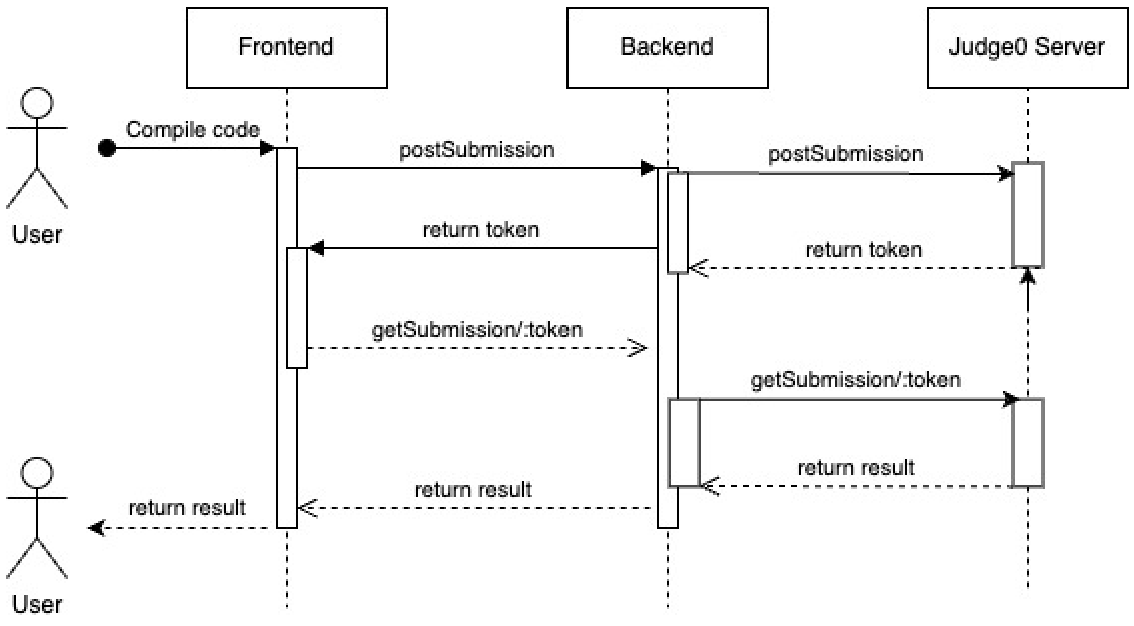
code_execution_service.js

As a bridge between our frontend and backend, we decide to use axios calls to hit our API endpoints in this file.

External Library: Judge API

Judge API is an external library that the application utilizes for code execution. Essentially, it serves as a tool to compile and run code snippets submitted by users, allowing the application to assess and display the output within the Code Execution Modal. We would have to subscribe and use our own respective API keys derived from https://rapidapi.com/judge0-official/api/judge0-extra-ce/ for code compilation.

Internal Sequence Flow



This diagram shows the flow of logic when a user attempts to compile his code. After compiling code, a function will be called to post the submission into the Judge0 server via our backend. A token will be returned to us, and the frontend will attempt to retrieve the result via said token until a result is returned from the Judge0 server. We will then display this result on our frontend.

History Service (N2)

The history service allows users to monitor the questions they've tried, along with details about each attempt, including the attempt date and time, as well as the submitted code.

Overall Microservice Architecture:

The microservice comprises frontend components and leverages backend components from other microservices. On the backend, it utilizes components from the Question Service Backend to retrieve question information and the User Service Backend to execute CRUD operations on user attempts.

Components Leveraged

The table demonstrates how the microservice employs backend components and API endpoints from other services. Each API endpoint, request type, and its purpose are outlined in the table below. Further information about each component can be explored in the relevant sections of their corresponding microservices.

Microservice	HTTP Request Type	API Endpoint	Purpose
User Service	POST	/submitAttempt	Submits an attempt entry to the user database.
	GET	/getAttempts	Retrieves a specific user's attempts from the user database.
	GET	/getQuestionsAttemptedPerUser	Retrieves statistics on the number of questions attempted by each user from the user database.
	GET	/getAttemptsPerQuestion	Retrieves statistics on the number of attempts by a unique user per question.
	GET	/getLanguageUsage	Retrieves statistics on the frequency of the usage of each language across all attempts.

Question Service	GET	/getQuestions	Retrieve questions and their details from the question database.

Database Model

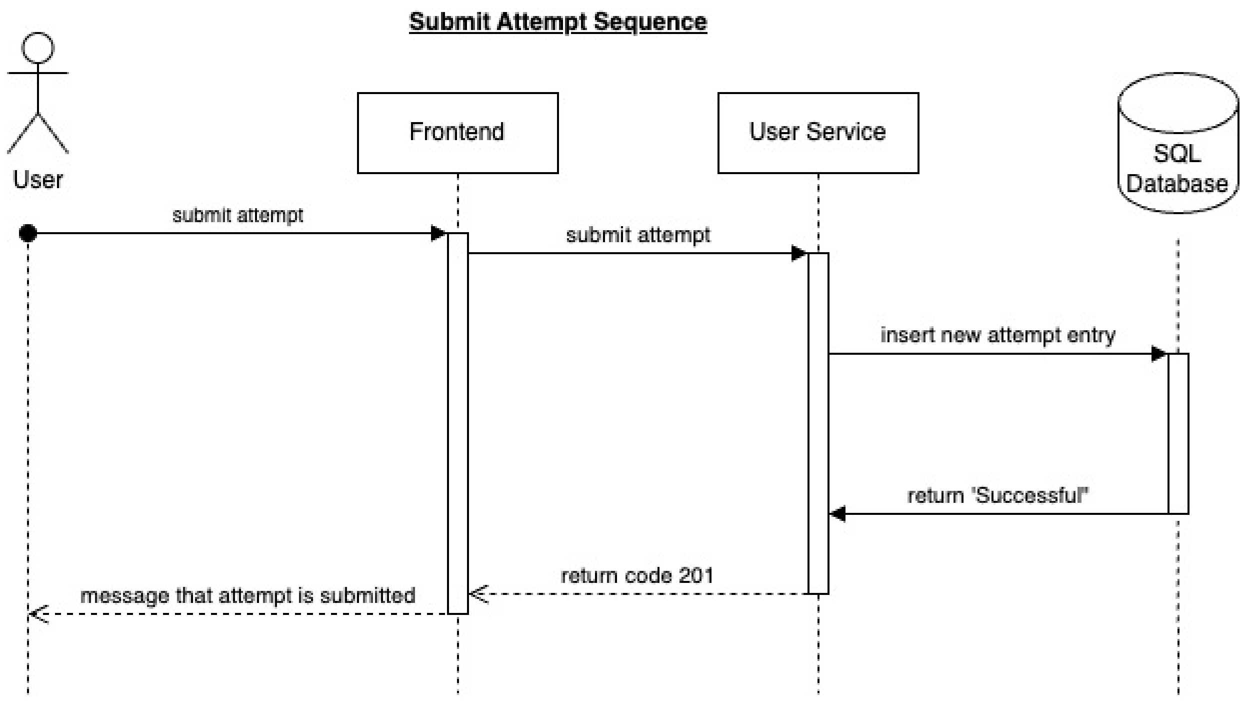
Recognizing the user-specific nature of attempts, we chose to integrate the attempts database with the user database. Hence, we store all attempts in the user database by creating a new table `attempts`. Details of the attempts table is shown in the table below.

Model: Attempt		
Fields	**Attributes**	**Description**
id	• type: int • required: true • unique: true	Unique ID used as the primary key for each attempt. Autogenerated by the database.
email_address	• type: String • required: true	Email address of the user who made the attempt. Foreign key used to identify the user.
question_id	• type: String • required: true	The unique ID of a question generated by the MongoDB question database. Used as a foreign key to identify the question attempted.
date_attempted	• type: timestamp • required: true	Stores the date and time of the attempt.
code	• type: String • required: true	Stores the code submitted during the attempt.
language_label	• type: String • required: true	The language of the code used during the attempt.
language_id	• type: String • required: true	Used to identify the language of the code used during the attempt.

Internal Sequence Flow

This sequence diagram depicts how a use case of the History Service would look like:

Submitting an attempt:



Frontend

The front-end used is React with Vite. It was chosen because Vite.js has a significantly faster development server, better hot-reloading experience, and built-in support for modern JavaScript features like ES modules and TypeScript, resulting in a more efficient and enjoyable React development process. Our group chose Mantine UI as our main UI provider because of its versatility and ease of implementation.

Home Page

The home page consists of the following components.

Mini Question Display

To display the question bank in rows and summarized form taken from the question database.

Completion Status

A UI indicator on how many questions the current logged in user has completed.

Questions Attempted

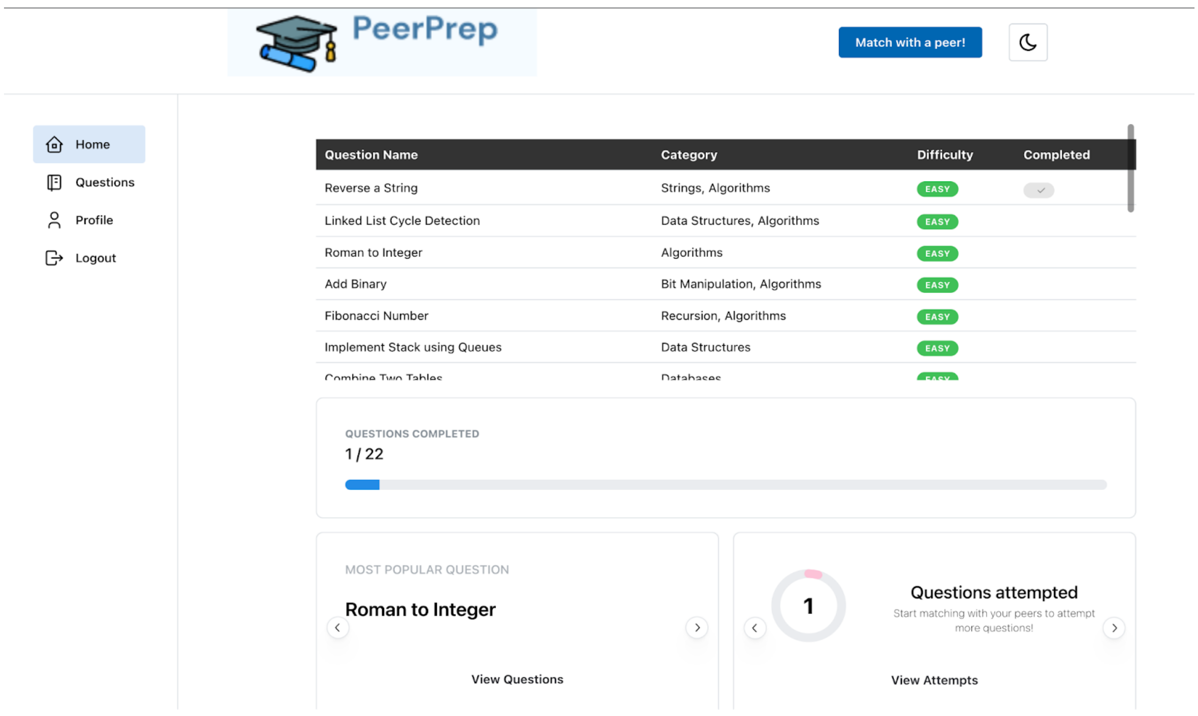
A UI indicator on how many questions the user has attempted, and the user can click on it to track his or her attempts.

Vertical Navigation Bar

A navigation bar that brings users to pages such as home page, questions and settings page.

Match With Peer Button

A button that brings users to the matching modal page, which we will cover later.



Question Name	Category	Difficulty	Completed
Reverse a String	Strings, Algorithms	EASY	✓
Linked List Cycle Detection	Data Structures, Algorithms	EASY	
Roman to Integer	Algorithms	EASY	
Add Binary	Bit Manipulation, Algorithms	EASY	
Fibonacci Number	Recursion, Algorithms	EASY	
Implement Stack using Queues	Data Structures	EASY	
Combine Two Tables	Databases	EASY	

QUESTIONS COMPLETED
1 / 22

MOST POPULAR QUESTION
Roman to Integer
View Questions

1
Questions attempted
Start matching with your peers to attempt more questions!
View Attempts

Questions Page

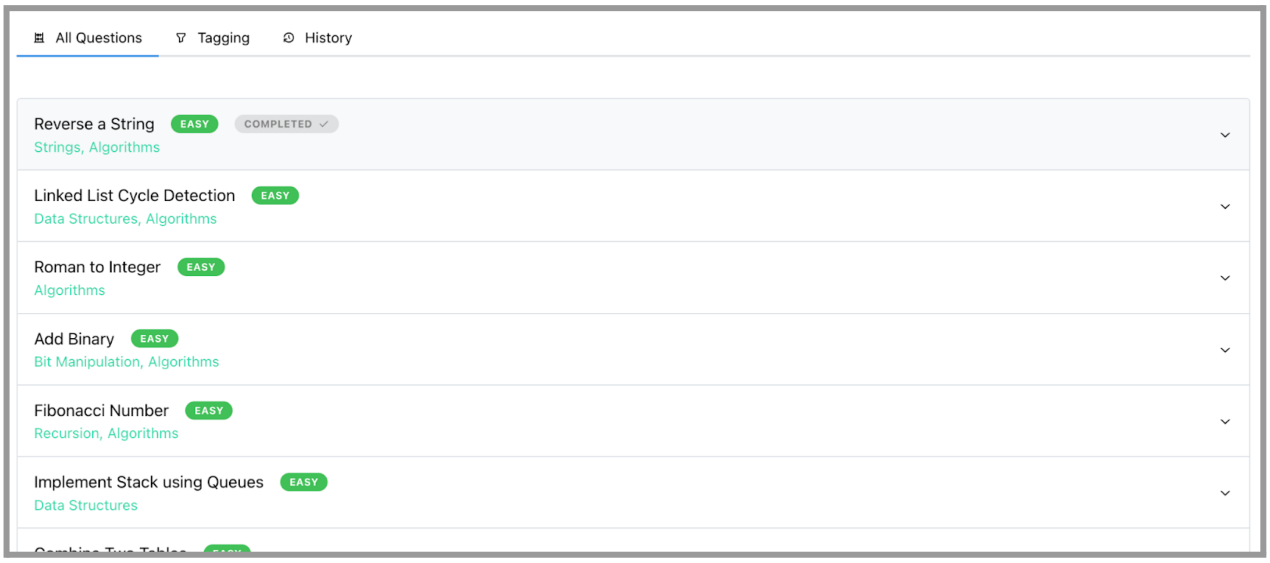
Role-Based Access Control Summary

The table presented below provides an overview of the features accessible to various roles. ‘Y’ indicates that the feature is accessible.

Feature	**User**	**Admin**
View question	Y	Y
Update question		Y
Delete question		Y
Create question		Y
Toggle question completion	Y	Y

All Questions

The first section in the Questions page allows the users to view the list of available questions in a glance. Questions are tagged with its difficulty level and status of completion for easy reference.



Reverse a String EASY COMPLETED ✓
Strings, Algorithms

Linked List Cycle Detection EASY
Data Structures, Algorithms

Roman to Integer EASY
Algorithms

Add Binary EASY
Bit Manipulation, Algorithms

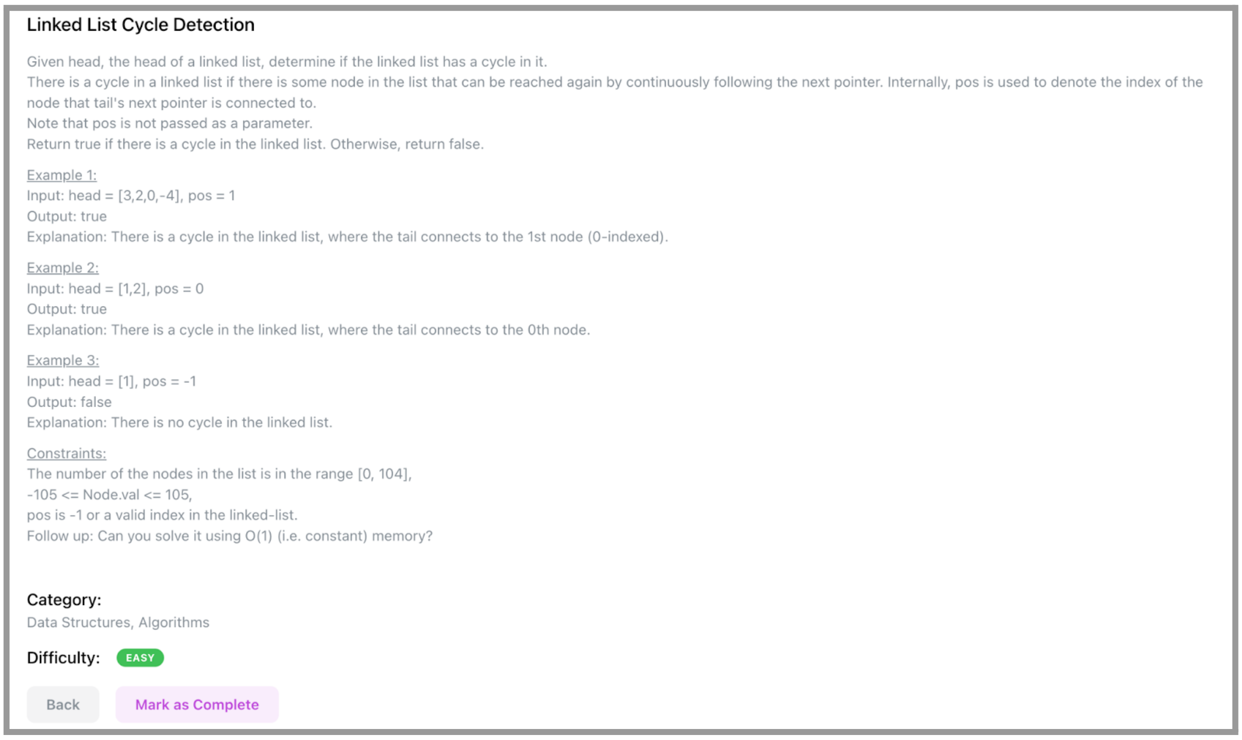
Fibonacci Number EASY
Recursion, Algorithms

Implement Stack using Queues EASY
Data Structures

Question Management

View Questions

Extended question panel that will be shown when the user chooses to view a question in detail. The full question is shown with its description, examples and category. “Mark as complete” button can be used to mark a question as complete.


Linked List Cycle Detection

Given head, the head of a linked list, determine if the linked list has a cycle in it.
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to.
Note that pos is not passed as a parameter.
Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:
Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.

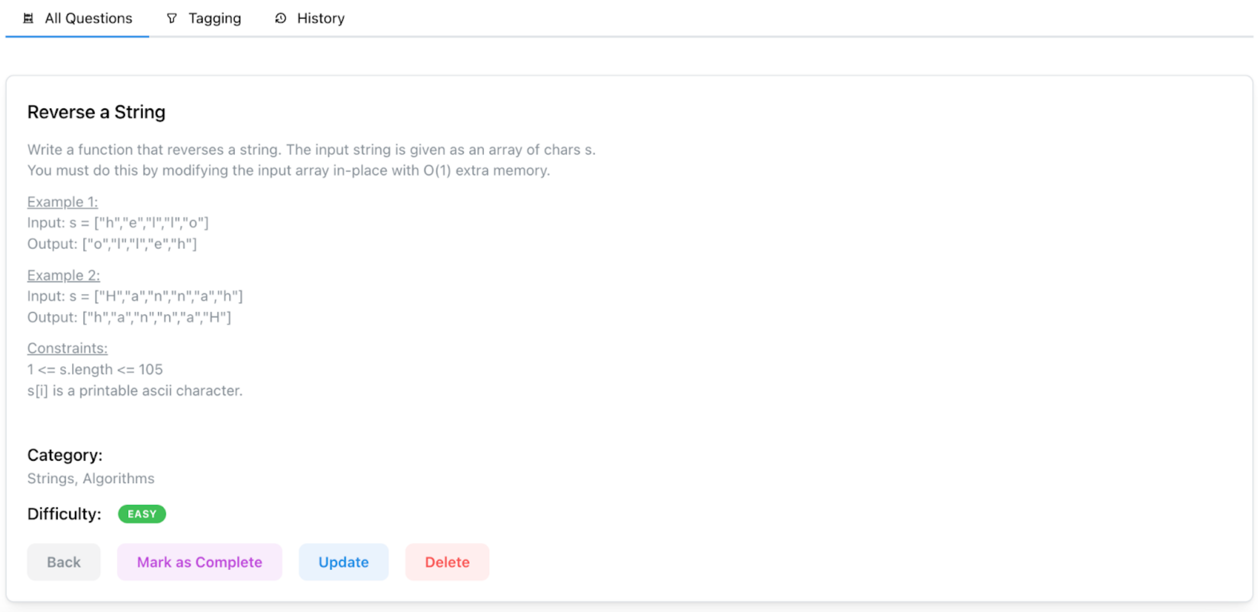
Constraints:
The number of the nodes in the list is in the range [0, 104].
-105 <= Node.val <= 105,
pos is -1 or a valid index in the linked-list.
Follow up: Can you solve it using O(1) (i.e. constant) memory?

Category:
Data Structures, Algorithms

Difficulty: EASY

Delete and Update buttons

Only users with admin rights will have access to the update and delete buttons.



Reverse a String

Write a function that reverses a string. The input string is given as an array of chars s.
You must do this by modifying the input array in-place with O(1) extra memory.

Example 1:
Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]

Example 2:
Input: s = ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]

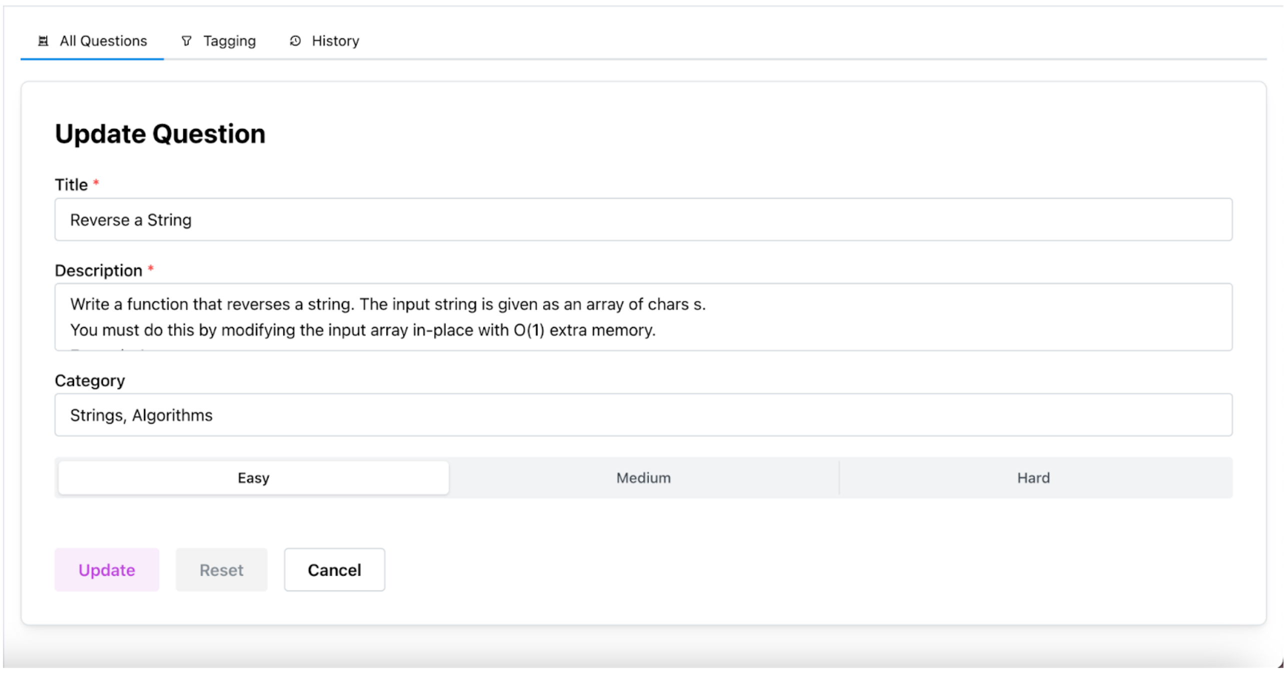
Constraints:
1 <= s.length <= 105
s[i] is a printable ascii character.

Category:
Strings, Algorithms

Difficulty: EASY

Update Question

For users with admin rights to update questions.



Update Question

Title *

Reverse a String

Description *

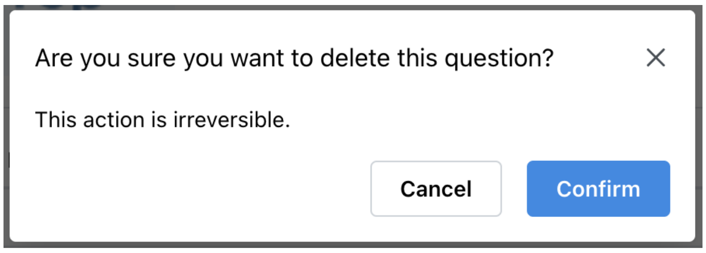
Write a function that reverses a string. The input string is given as an array of chars s.
You must do this by modifying the input array in-place with O(1) extra memory.

Category

Strings, Algorithms

Delete Question

Confirmation modal for users with admin rights to delete a question.

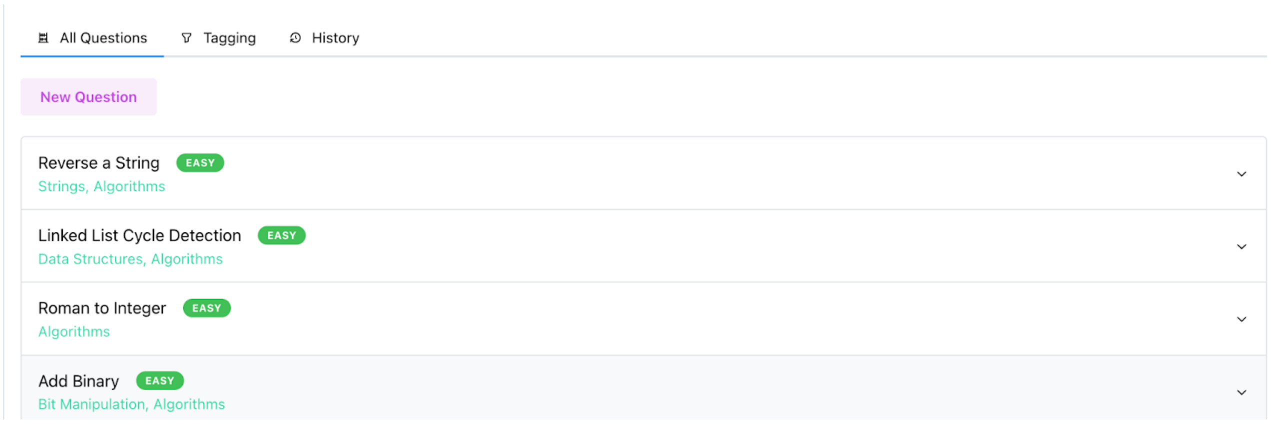


Are you sure you want to delete this question?

This action is irreversible.

Create Question Button

Only users with admin rights will have access to this button which will allow them to create new questions.



Reverse a String EASY
Strings, Algorithms

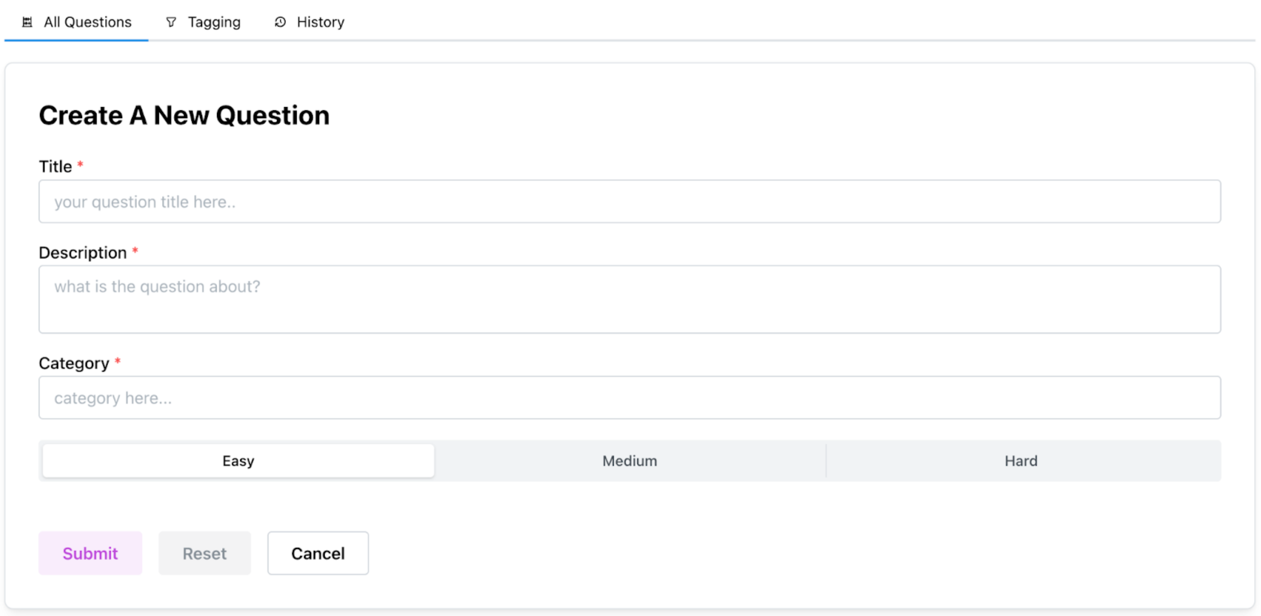
Linked List Cycle Detection EASY
Data Structures, Algorithms

Roman to Integer EASY
Algorithms

Add Binary EASY
Bit Manipulation, Algorithms

Create Question

For users with admin rights to create a new question.



Create A New Question

Title *

your question title here..

Description *

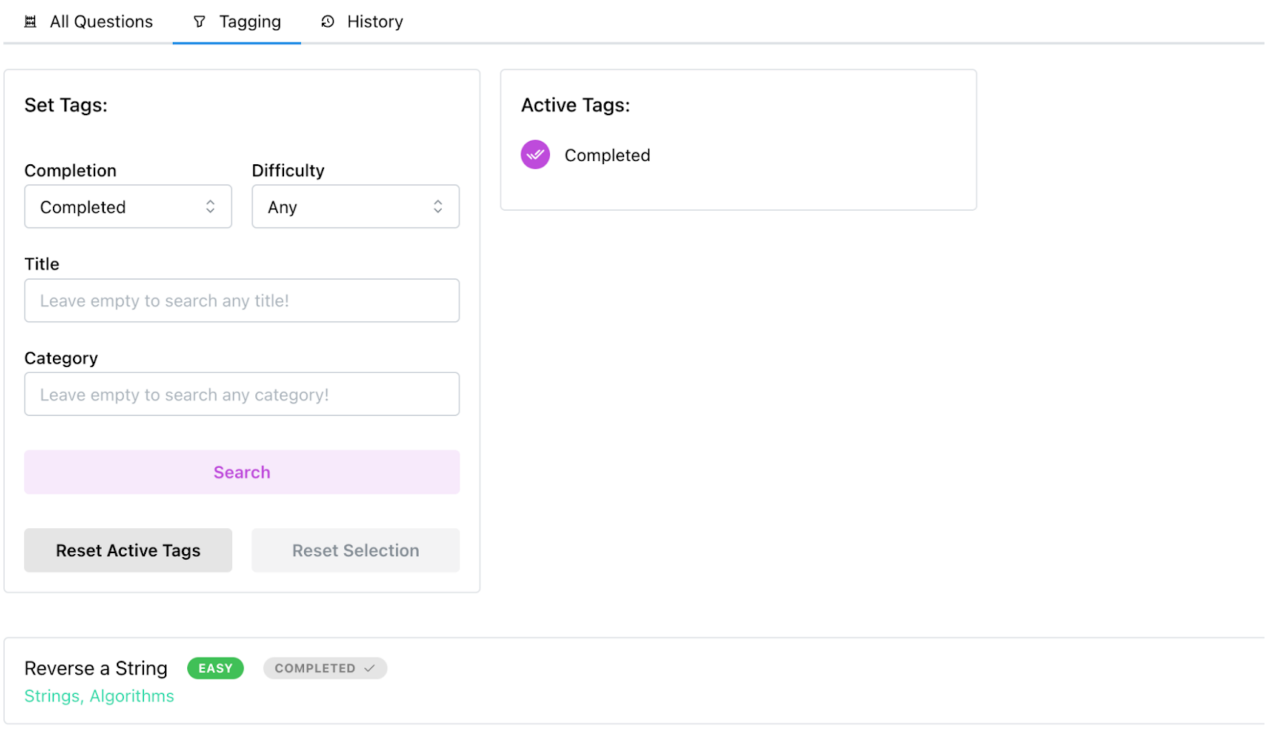
what is the question about?

Category *

category here...

Tagging (N4)

A page for users to filter out the questions based on relevant tags.



Set Tags:

Completion: Completed

Difficulty: Any

Title

Leave empty to search any title!

Category

Leave empty to search any category!

Active Tags:

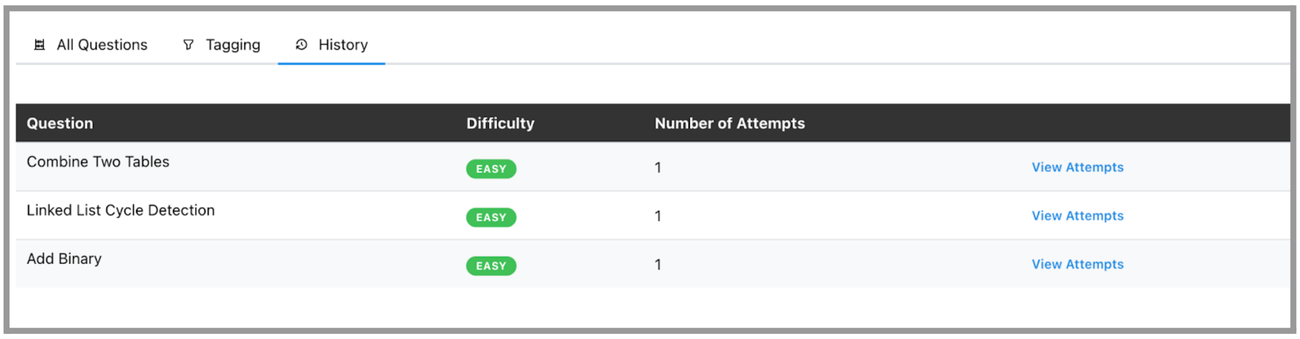
Completed

Reverse a String EASY COMPLETED ✓

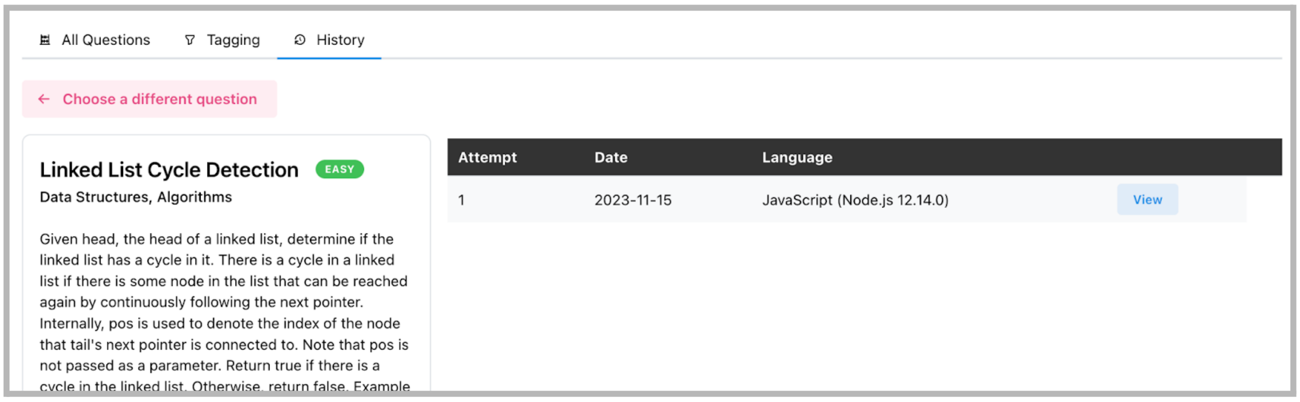
Strings, Algorithms

History (N2)

A page for users to view their past attempts on questions. They can view the question details, date attempted, and the code they submitted.



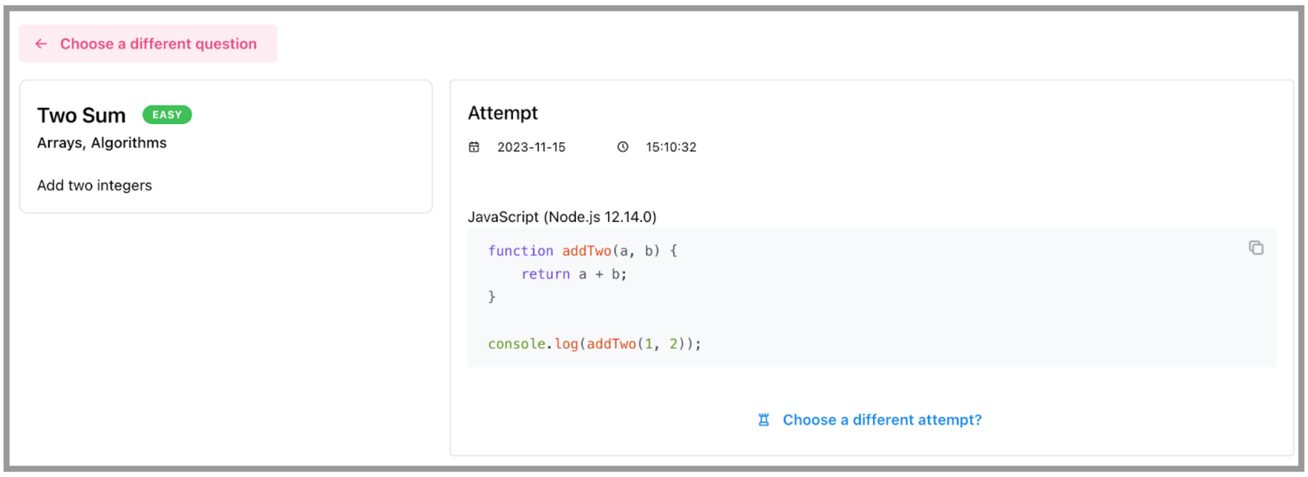
Question	Difficulty	Number of Attempts	
Combine Two Tables	EASY	1	View Attempts
Linked List Cycle Detection	EASY	1	View Attempts
Add Binary	EASY	1	View Attempts



Linked List Cycle Detection EASY
Data Structures, Algorithms

Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter. Return true if there is a cycle in the linked list. Otherwise, return false. Example

Attempt	Date	Language	
1	2023-11-15	JavaScript (Node.js 12.14.0)	View



Two Sum EASY
Arrays, Algorithms

Add two integers

Attempt
2023-11-15 15:10:32

JavaScript (Node.js 12.14.0)

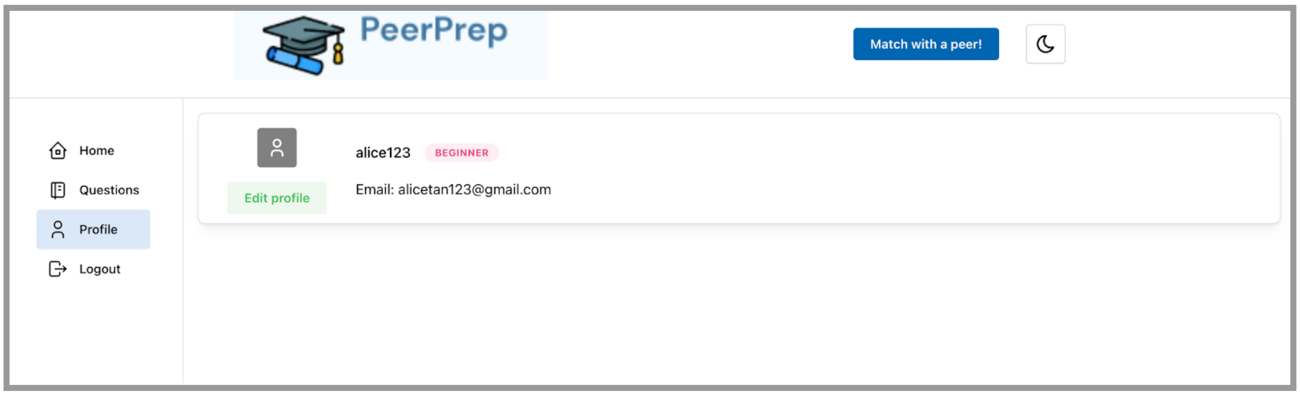
```
function addTwo(a, b) {
    return a + b;
}

console.log(addTwo(1, 2));
```

Settings Page

Profile Management

Buttons to edit profile information of the user or even delete the profile, and indicates if the current user is an admin or not. It also displays the current level of the user.


PeerPrep

Match with a peer!

Home

Questions

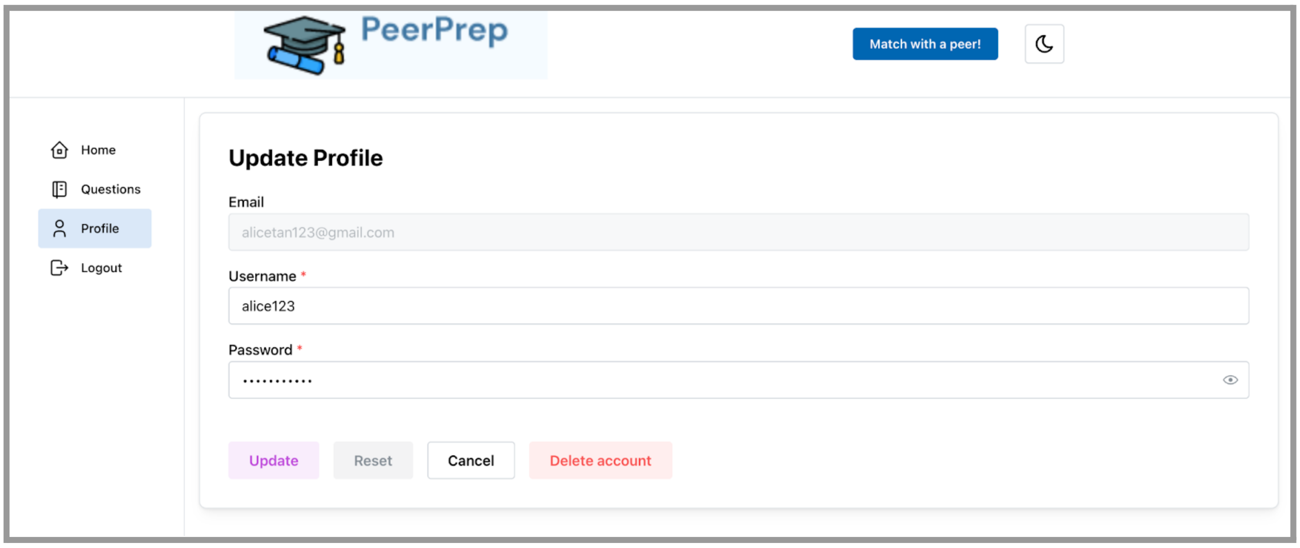
Profile

Logout

alice123 BEGINNER

Edit profile

Email: alicetan123@gmail.com


PeerPrep

Match with a peer!

Home

Questions

Profile

Logout

Update Profile

Email: alicetan123@gmail.com

Username *: alice123

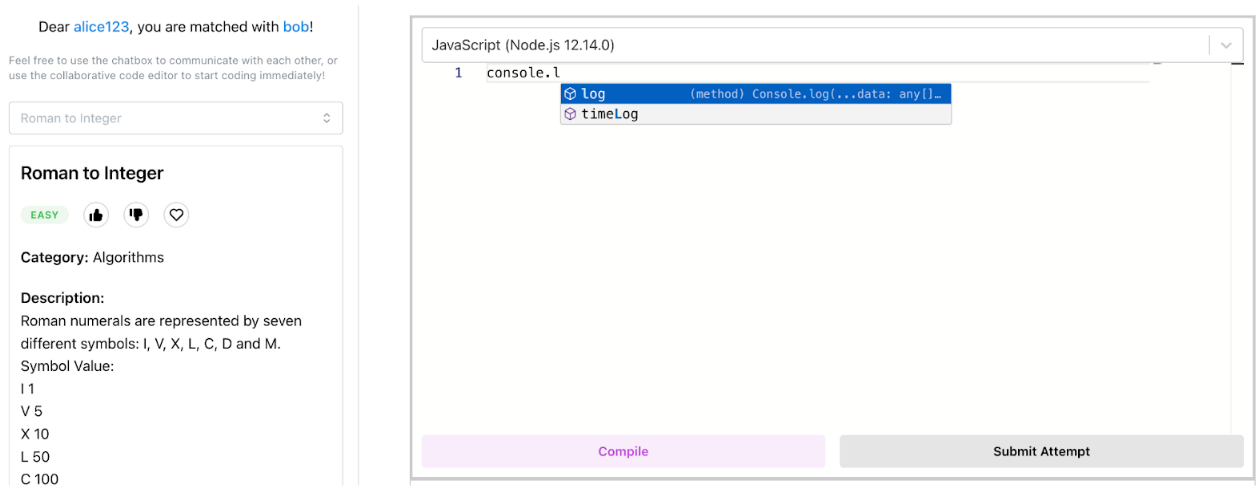
Password *: ••••••••••••

Update | Reset | Cancel | Delete account

Room

Code Editor with Syntax Highlighting (N5)

Monaco code editor that supports yjs-websocket and syntax highlighting for multiple languages. It also has auto-complete for some languages.


Dear alice123, you are matched with bob!

Feel free to use the chatbox to communicate with each other, or use the collaborative code editor to start coding immediately!

Roman to Integer

Roman to Integer

EASY

Category: Algorithms

Description:
Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.
Symbol Value:
I 1
V 5
X 10
L 50
C 100

JavaScript (Node.js 12.14.0)

```
console.l
  log        (method) Console.log(...data: any[]…
  timeLog
```

Compile | Submit Attempt

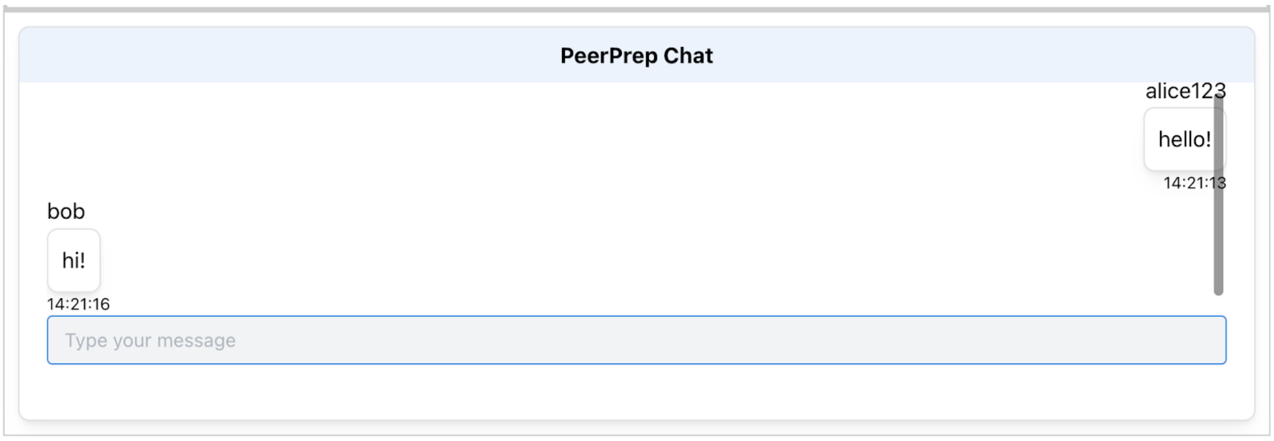
External Library: Monaco Editor for React
Monaco Editor is a code editor that powers VS Code. This library handles the setup process of a monaco editor in the front-end client and provides a clean interface for interacting.

External Library: Socket.io Client
To enable a persistent connection between the clients for synchronization, we are using Socket.io Client to interface with the Collaboration Service API, which is also powered by Socket.io. Upon rendering of the Editor, the front-end client initiates a connection with the Collaboration Service API and is henceforth listening for changes in the Common Editor by their partner.

Chatbox (N1)

Users can communicate with each other through a chat box placed at the bottom of the room. Users can see the message that was sent, along with the sender's name, content, and timestamp.

External Library: Socket.io Client
We employed the socket.io client library to instantiate client-side sockets, enabling communication between users within the chat system. Interaction with socket.io occurs during the initial creation of a designated room, where client sockets are allocated to the same room. Subsequently, when exchanging messages between users, these messages are transmitted using the socket.io framework.

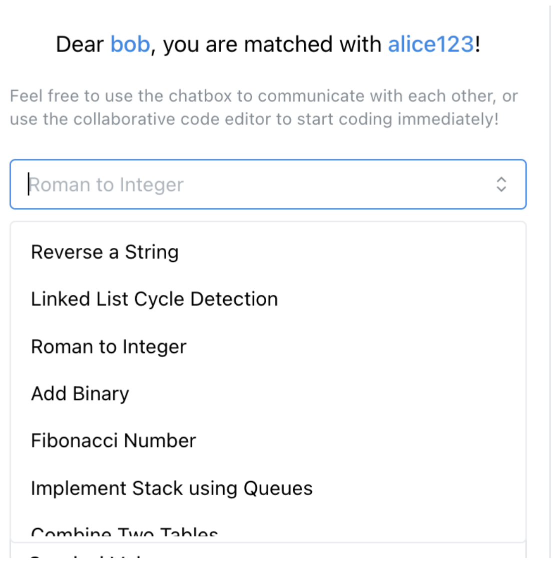

PeerPrep Chat

alice123
hello!
14:21:13

bob
hi!
14:21:16

Change Question Modal (N4)

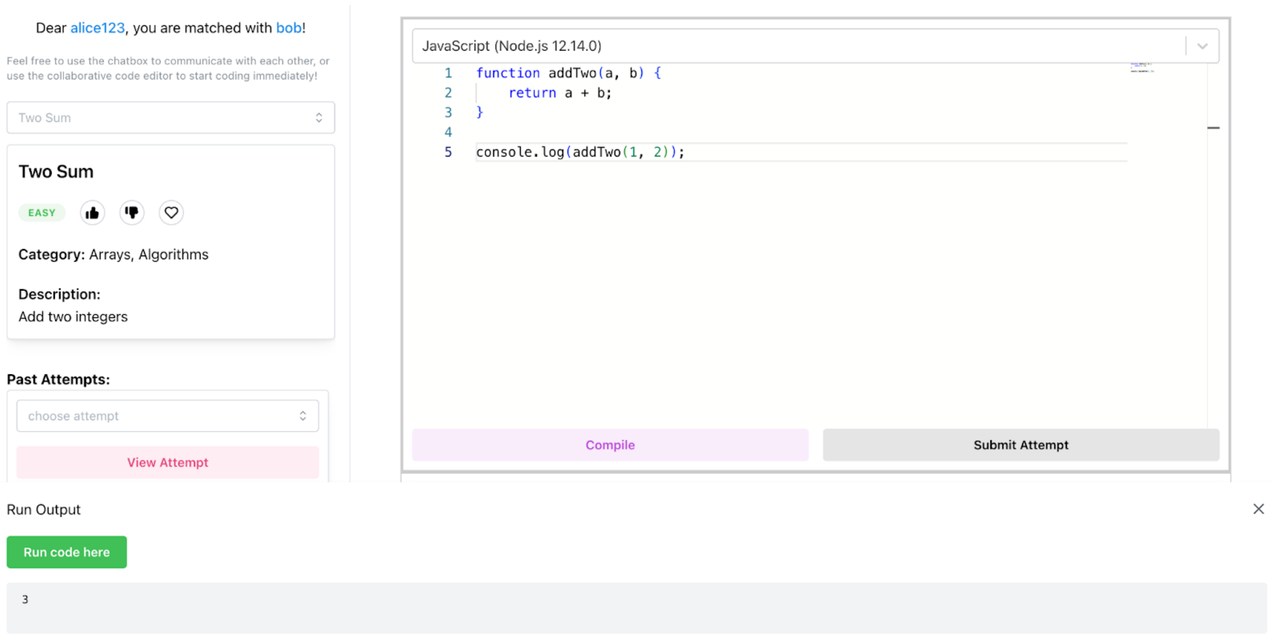
This is a Modal that appears when users click on a button to change the question being collaborated on. In this modal, users can specify the difficulty of the question they would like to attempt, and subsequently choose a specific question to attempt. The selected question is then shown in Markdown inside the modal. Clicking on the "Confirm Question Change" button will result in a change in question for both users in the session.


Dear bob, you are matched with alice123!

Feel free to use the chatbox to communicate with each other, or use the collaborative code editor to start coding immediately!

Code Execution Modal (N3)

This modal appears when the user clicks “Compile”. The output of the code will be shown in a code box for the user to see.


Dear alice123, you are matched with bob!

Feel free to use the chatbox to communicate with each other, or use the collaborative code editor to start coding immediately!

Two Sum

Category: Arrays, Algorithms

Description:
Add two integers

Past Attempts:

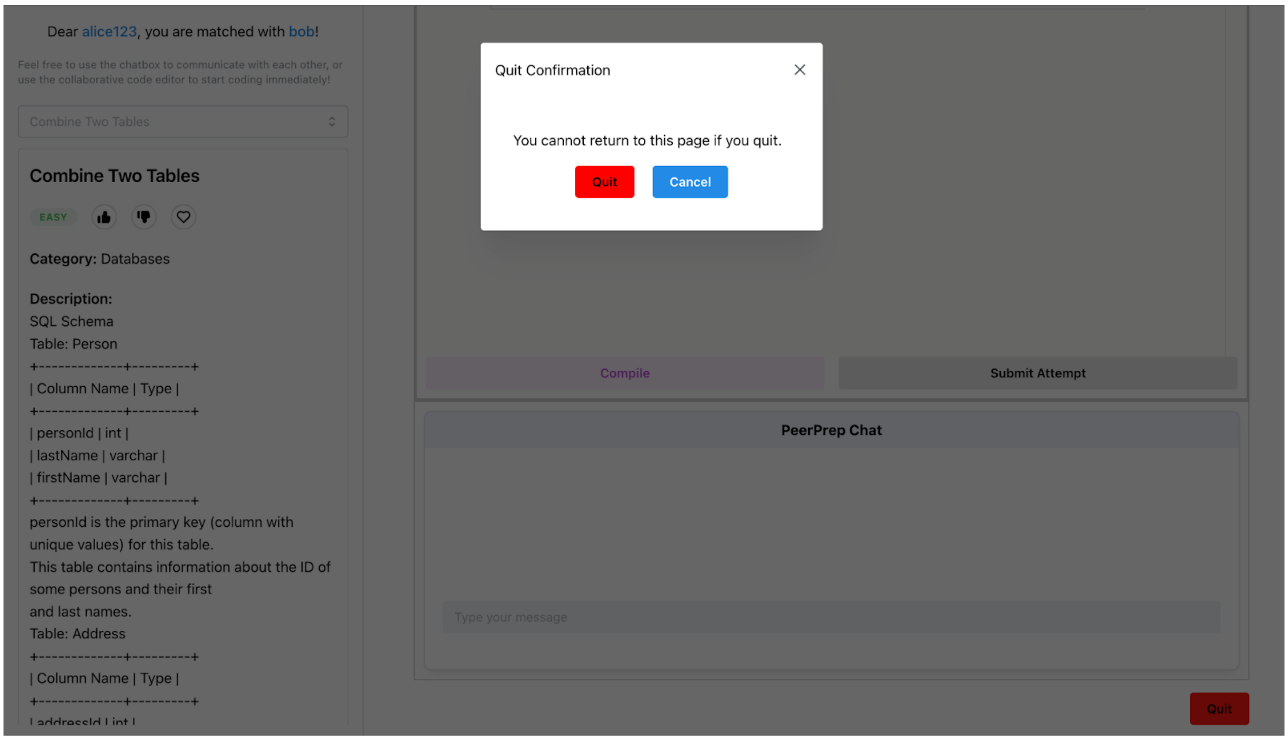
```
function addTwo(a, b) {
    return a + b;
}

console.log(addTwo(1, 2));
```

```
3
```

Quit Modal

The modal appears when the user selects the quit button. Quitting will return the user to the main page.



Dear alice123, you are matched with bob!

Feel free to use the chatbox to communicate with each other, or use the collaborative code editor to start coding immediately!

Combine Two Tables

Combine Two Tables

EASY

Category: Databases

Description:
SQL Schema
Table: Person

```
+-------------+---------+
| Column Name | Type |
+-------------+---------+
| personId | int |
| lastName | varchar |
| firstName | varchar |
+-------------+---------+
```

personId is the primary key (column with unique values) for this table.
This table contains information about the ID of some persons and their first and last names.
Table: Address

```
+-------------+---------+
| Column Name | Type |
+-------------+---------+
| addressId | int |
```

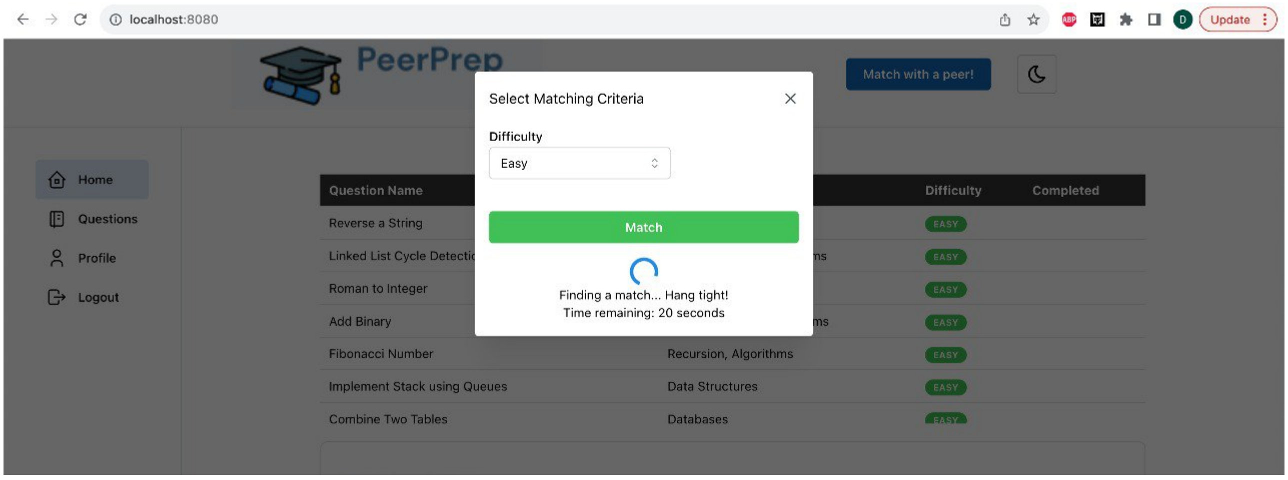
Quit Confirmation

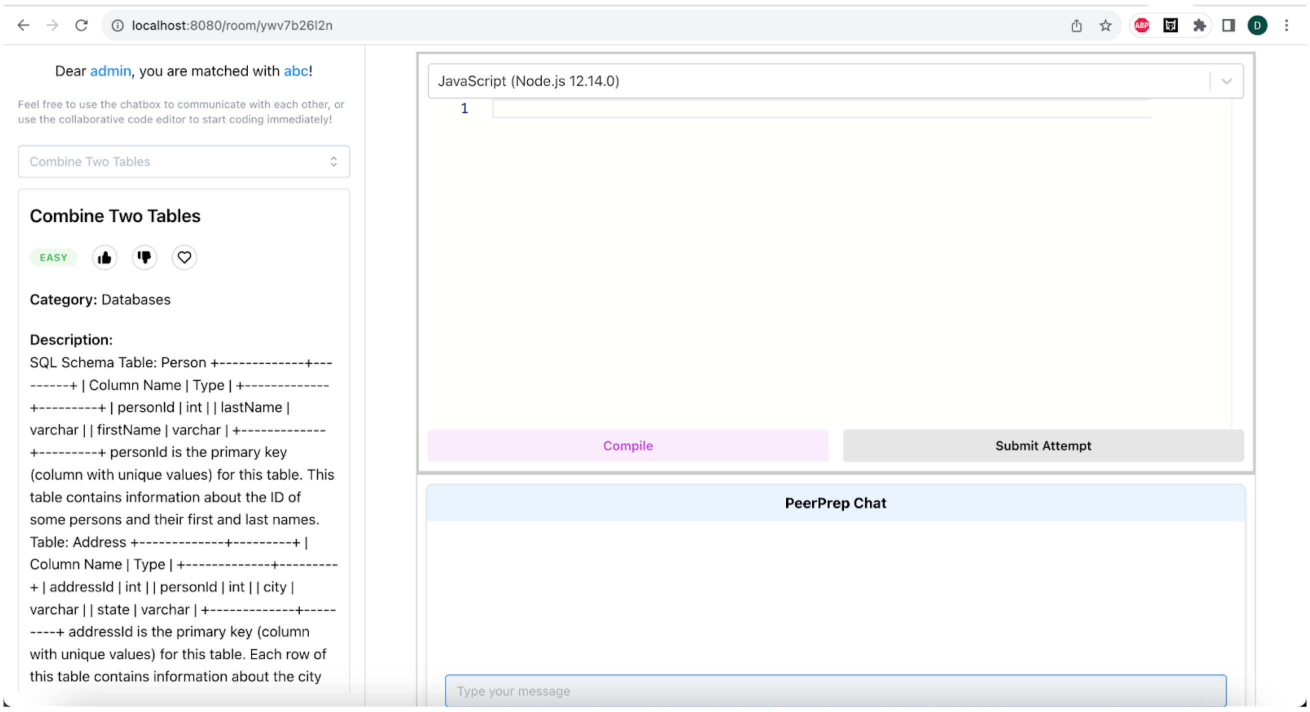
You cannot return to this page if you quit.

Matching Modal

The matching modal is where users will select their difficulty and wait for a match. Users will be connected to the matching-service via sockets which will match users together.

<u>External Library: Socket.io Client</u>
To enable a persistent connection between the clients and the backend server, we are using Socket.io Client to interface with the Matching Service API, which is also powered by Socket.io. Upon rendering of the 'Select Matching Criteria' and the button 'Match' being clicked, the front-end client initiates a connection with the Matching Service API. The front-end client then updates the Matching Service API via socket events, for e.g find-match. The Matching Service API also updates relevant front-end clients when users are matched.





Dear admin, you are matched with abc!

Feel free to use the chatbox to communicate with each other, or use the collaborative code editor to start coding immediately!

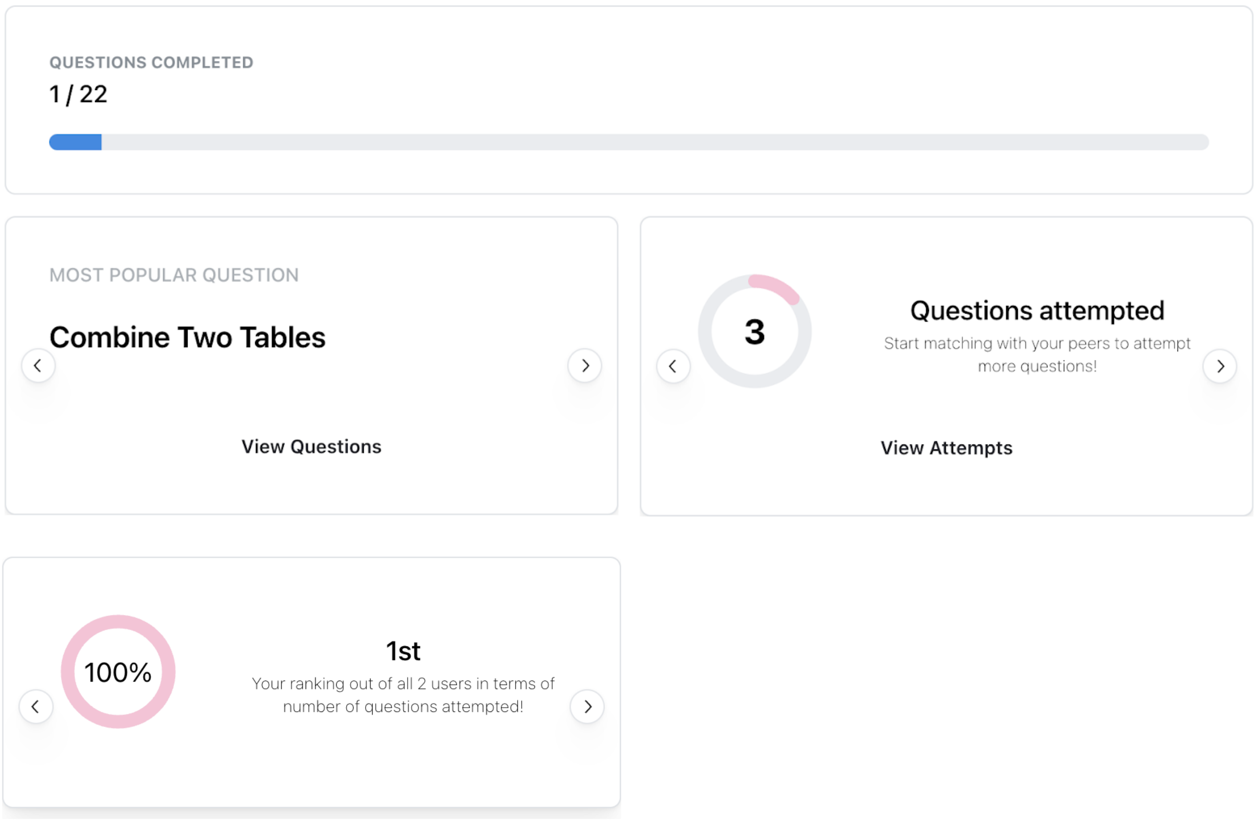
Combine Two Tables

Category: Databases

Description:
SQL Schema Table: Person +-------------+---------+ | Column Name | Type | +-------------+---------+ | personId | int | | lastName | varchar | | firstName | varchar | +-------------+---------+ personId is the primary key (column with unique values) for this table. This table contains information about the ID of some persons and their first and last names. Table: Address +-------------+---------+ | Column Name | Type | +-------------+---------+ | addressId | int | | personId | int | | city | varchar | | state | varchar | +-------------+---------+ addressId is the primary key (column with unique values) for this table. Each row of this table contains information about the city

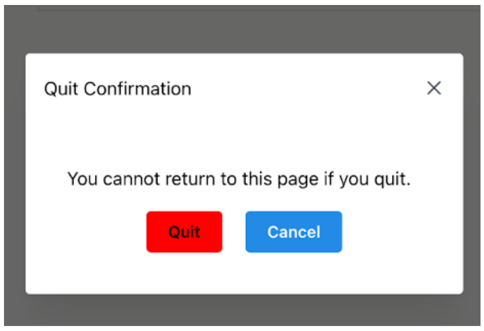
High Usability with Good UI/UX design

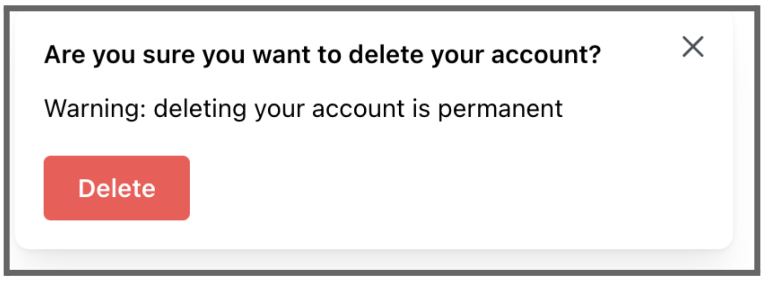
In this project we made high usability and good UI/UX design a priority through an NFR (see more below in Requirements).


QUESTIONS COMPLETED
1 / 22
MOST POPULAR QUESTION
Combine Two Tables
View Questions
3
Questions attempted
Start matching with your peers to attempt more questions!
View Attempts
100%
1st
Your ranking out of all 2 users in terms of number of questions attempted!

As seen from the above screen shot of the dashboard on our home page, we attempt to effectively enhance the user experience, through presenting statistics that they might be interested in.

For instance, the user is able to see a progress bar indicating the number of questions they have completed, as well as the number of questions they have attempted. They are also able to see their ranking compared to their peers, motivating them to attempt more questions.




Are you sure you want to delete your account?
Warning: deleting your account is permanent
Delete

Another instance of effective UI/UX design is shown in Quit Modal, where the user is prompted for confirmation upon quitting the room. This confirmation prevents the user from accidentally exiting the session room and losing their progress.

Throughout the app, we consistently implemented confirmation notifications upon critical user actions (e.g. when a user updates/deletes their account).

Containerization Details

Matching, Question, Collaboration Service, Code Execution, Chatbox, User, Frontend

The matching, question, collaboration, code execution, chatbox and user services are Node.js backend services, which can all be containerized into Docker containers. Similarly, for the frontend, it can also be containerized into a Docker container.

Containerization with Docker

For ease of deployment in the future, the Matching Service, Question Service, Collaboration Service, Code Execution Service, Chatbox Service, User Service and Frontend service have been containerized, following the motto “write once, run anywhere”. With these containers, we can run the services on a variety of architectures without having to rewrite code to fit a particular architecture. Containerization also allows the services to be scaled easily with container orchestration technologies in the future.

Local Deployment: Docker Compose

For ease of local testing, we have also included a local deployment setup that involves docker compose.

Ensure the docker daemon is running locally on your computer. Then, we can simply run the command `docker-compose -f src/docker-compose.yml build` to build the images locally and then run `docker-compose -f src/docker-compose.yml up -d` in the root folder to set up all six services running in containers locally. To stop and remove the containers, we can run the command `docker-compose -f src/docker-compose.yml down` and this will ensure that all containers running locally will be stopped and removed.

More information on how to run a complete local deployment of our system can be found in our README on GitHub.

User Service

Local Deployment: PGadmin4 (PostgreSQL)

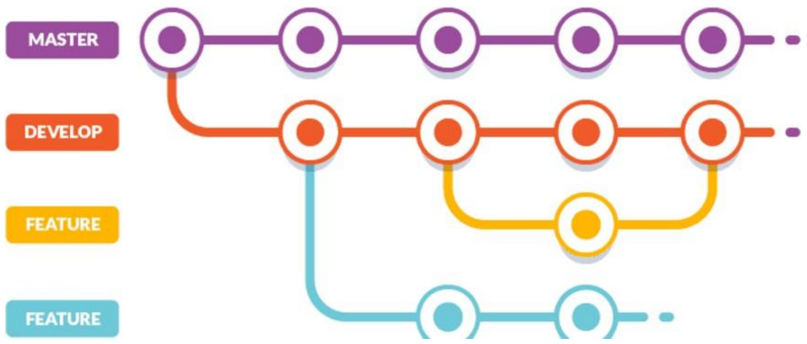
In our local deployment, we leverage PostgreSQL as the database engine. We decided to use PGAdmin4 as the graphical administration tool since it provides an intuitive interface for us. PGAdmin4 enhances our database management with features like visual query building, performance monitoring, and role management, ensuring efficient local development and administration. This pairing offers a dependable, scalable, and open-source solution for managing our user database, allowing seamless adjustments and expansions to the schema as required.

Use of PostgreSQL: Horizontal Scalability

PostgreSQL supports horizontal scaling, allowing distribution of the database workload across multiple nodes. This facilitates handling increased data and user loads.

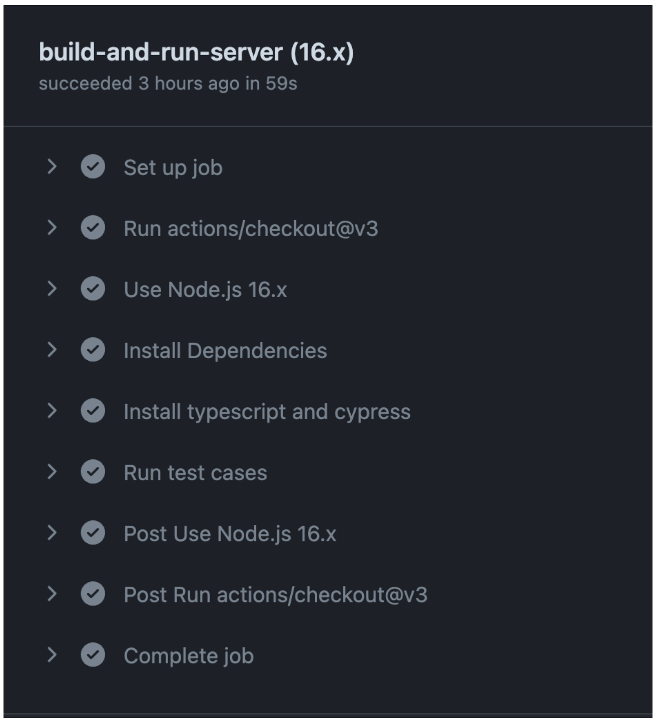
Development Process

Feature Branch Workflow



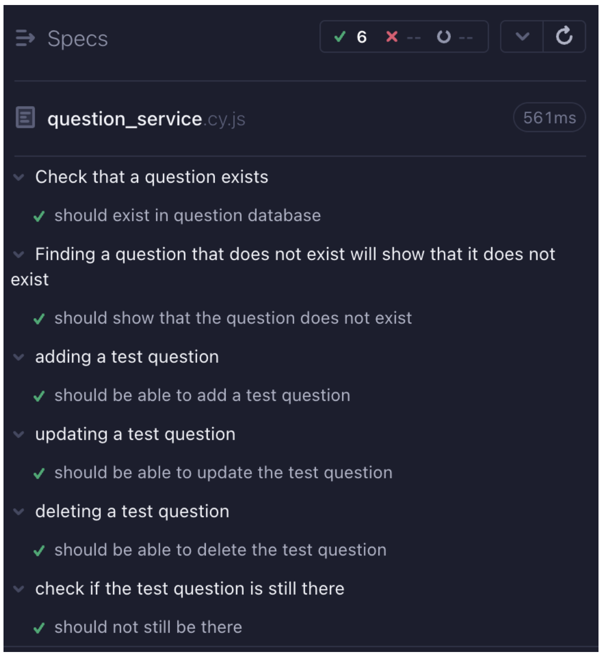
In this module, the team opted for a branching workflow. Here, each team member developing a feature creates a new branch from the main branch. After completing the feature, they submit a pull request to merge it back into the main branch.

CI with automation testing (N8)



build-and-run-server (16.x)
succeeded 3 hours ago in 59s

- ✓ Set up job
- ✓ Run actions/checkout@v3
- ✓ Use Node.js 16.x
- ✓ Install Dependencies
- ✓ Install typescript and cypress
- ✓ Run test cases
- ✓ Post Use Node.js 16.x
- ✓ Post Run actions/checkout@v3
- ✓ Complete job



Specs ✓ 6 ✕ -- --

question_service.cy.js 561ms

- Check that a question exists
  - ✓ should exist in question database
- Finding a question that does not exist will show that it does not exist
  - ✓ should show that the question does not exist
- adding a test question
  - ✓ should be able to add a test question
- updating a test question
  - ✓ should be able to update the test question
- deleting a test question
  - ✓ should be able to delete the test question
- check if the test question is still there
  - ✓ should not still be there

As a nice-to-have feature, we have incorporated Continuous Integration (CI) with automation unit testing.

Automation testing

Our group has decided to use Cypress as our choice of automation testing framework. Cypress is able to provide fast and reliable end-to-end testing for our web application. It is also a user-friendly framework with a syntax that is easy to learn and understand, prioritizing on readability and continuity.

In our case, we decided against UI testing because our user service backend database is PostgreSQL, which is hosted locally and not deployed on cloud. As such, we are not able to integrate our user service backend into github actions, which means that we cannot sign in to our application without a database of user credentials. It is also this reason that unfortunately deters us from testing the API endpoints of our user service backend, matching service backend and collab service backend.

Nevertheless, we could still use Cypress to test our question service backend because the database is hosted on MongoDB, which is on the cloud. Our group tried to test question service as extensively as possible by making sure that every endpoint works.

Continuous Integration

Whenever a push or a pull request to the main branch is made, the CI workflow is run, automatically setting up the local Node.js testing environment and installing dependencies for Cypress testing. The CI workflow then runs the unit test suites for the question service. This way, whenever a change is made to a feature or a new feature is written, we can ensure that there are no regressions to existing features, as PRs cannot be merged without first passing CI.

Project Management

Software Development Lifecycle

Our team opted for an Agile Software Development Life Cycle (SDLC) to manage our project. We engaged in iterative sprints, typically lasting around two weeks, to enhance our project. A weekly standup meeting was held, during which team members shared updates on their progress since the previous week. We used Github Issues to assign new features and big fixes to our group members.

At the conclusion of each sprint, we hold a sprint meeting to review successes and areas for improvement. Additionally, we update our personal Project Board which functions as our backlog. A Google Doc version of the Sprint Planning we've undertaken is presented below.

Since the completion of Milestone 1 we had a mostly complete product that served the basic functionality that we wished to provide. Every week since, we iterated on the product and gradually added nice-to-have features to it, adhering to the iterative nature of Agile development.

Project Sub-Groups

Sub group 1 (Justin, Sheryl)	Sub group 2 (Nic, Derrick, Zi Yang)
(N1) Communication: Chat function	**(N2) History:** List of questions attempted along with the date-time of attempt
(N3) Code execution: execute code in a sandbox environment and present results	**(N4) Enhance question service** to enable managing questions, on the fly during a session initiation.
(N8) CI pipeline with unit testing: execute unit testing using Github Actions every time a pull request or push is made	**(N5) Code editor** with syntax highlighting for multiple languages

Requirements

Functional Requirements

FR ID	Component	Feature Description	Priority
FR 1.1	User Service	The system should allow that new users can register, given that their email account is not in the database	Must-have
FR 1.2	User Service	The system should allow existing users to login, given they key in their email account and password correctly	Must-have
FR 1.3	User Service	The system should allow users to log out of their account	Must-have
FR 1.4	User Service	The system should maintain a record of questions attempted by a user, as well as the date and time of attempt, and the current code at that state.	Nice-to-Have
FR 1.5	User Service	The system should allow users to saved finished question attempts	Should-have
FR 1.6	User Service	The system should identify if a user is a normal user or an admin, and give question CRUD privileges to an admin, while preventing normal users from modifying questions	Must-have
FR 1.7	User Service	The system should allow users to edit their profile information or even delete their profile, giving a warning to them beforehand.	Must-have
FR 2.1	Matching Service	The system should allow users to select the difficulty of the questions they wish to attempt.	Must-have
FR 2.2	Matching Service	The system should be able to match two waiting users with similar difficulty levels and put them in the same room	Must-have
FR 2.3	Matching Service	If there is a valid match, the system should match the users within 30s	Must-have

FR 2.4	Matching Service	The system should inform the users that no match is available if a match cannot be found within 30 seconds	Must-have
FR 2.5	Matching Service	The system should notify users of their waiting status while waiting for a match.	Must-have
FR 2.6	Collaboration Service	The system should provide a means for the user to leave a room once matched	Must-have
FR 2.7	Collaboration Service	The system should allow users to edit their code at the same time and have both editors be in sync	Must-have
FR 2.8	Collaboration Service	The system should allow users in the same room to chat with one another via real time chat messaging	Nice-to-Have
FR 2.9	Collaboration Service	The system should close the room once both users leave the room, and they cannot enter the same room again.	Must-have
FR 3.1	Dashboard	The system should allow users to see which questions they have attempted before	Nice-to-Have
FR 3.2	Dashboard	The system should allow users to see which questions they have saved before	Nice-to-Have
FR 3.3	Dashboard	The system should allow users to view a breakdown of the questions attempted and saved based on difficulty levels	Nice-to-Have
FR 4.1	User Service	The system should allow users to continue their attempt from previously saved code	Should-have
FR 4.2	Question Service	The system should allow users to obtain specific questions from the question bank on the fly.	Nice-to-Have
FR 4.3	Question Service	The system should select a random question from the question bank once 2 users match	Must-have
FR 4.4	Question Service	The system should be able to hold a bank of questions.	Must-have
FR 4.5	Code Execution Service	The user should be able to execute his/her code in the code and see the output of the code, even if there was an error.	Nice-to-Have
FR 4.6	Code Syntax	The user should be able to view his/her code easily with the help of indents, autocomplete and syntax highlighting for multiple languages.	Nice-to-Have
FR 4.7	Question Service	The user should be able to manage questions by tagging them according to category, difficulty and completion status.	Nice-to-Have

FR 4.8	Question Service	The user can obtain questions on the fly during a session.	Nice-to-Have

Non-functional Requirements

Below are the non-functional requirements that we strived to achieve in the project.

NFR 1.1	The system should be built using a microservice architecture.
NFR 1.2	The response time of the system should be fast. (All user actions should be responded to under 3 seconds)
NFR 1.3	The system should feature high usability in terms of user experience.
NFR 1.4	The system should be dockerized to facilitate future deployment.
NFR 1.5	The system's bugs should be clamped out as much as possible with testing.

Sprint Planning

Milestone 1

Sprint 1 (Week 5 and Week 6)

Persons in charge	Status	FRs / NFRs
Justin, Sheryl	Completed	FR 1.1
Justin, Sheryl	Completed	FR 1.2
Justin, Sheryl	Completed	FR 1.3
Justin, Sheryl	Shift to wk 7	FR 1.6
Justin, Sheryl	Completed	FR 1.7
Derrick, Nicholas, Zi Yang	Completed	FR 4.4

Milestone 2

Sprint 2 (Recess Week and Week 7)

Person in charge	Status	FRs / NFRs

Justin, Sheryl	Completed	FR 1.6
Zi Yang	Completed	FR 2.1
Everyone	Completed	NFR 1.1
Derrick	Shift to wk 8	FR 2.2

Sprint 3 (Week 8 and Week 9)

Person in charge	Status	FRs / NFRs
Derrick	Completed	FR 2.2
Derrick	Completed	FR 2.3
Derrick	Completed	FR 2.4
Derrick	Completed	FR 4.3
Zi Yang	Completed	FR 2.6
Zi Yang	Completed	FR 2.5
Zi Yang	Completed	FR 2.7
Zi Yang	Completed	FR 4.6

Milestone 3

Sprint 4 (Week 10 and Week 11)

Person in charge	Status	FRs / NFRs
Sheryl	Completed	FR 2.8
Zi Yang	Completed	FR 2.9
Nicholas	Completed	FR 3.1
Nicholas	Completed	FR 3.2
Nicholas	Completed	FR 3.3
Nicholas	Completed	FR 4.1
Sheryl / Nicholas	Completed	FR 4.2
Justin	Completed	FR 4.5

Nicholas	Completed	FR 4.7
Justin / Zi Yang	Completed	NFR 1.5
Everyone	Completed	NFR 1.3

Final Sprint (Week 12 and 13)

Person(s) in charge	Status	FRs / NFRs
Derrick, Justin	Completed	NFR 1.4

Improvements and Enhancements

User Service

User Registration and Authentication

- Implement a robust email verification system to enhance the security of new user registrations.
- Enhance the user profile recording system by including additional metrics such as success rate and time spent on questions.
- Explore other common forms of authentication such as OAuth for enhanced security.

Communication Service

Voice call

- Further improve our communication service by having a voice call feature in future iterations of PeerPrep.

Collaboration Service

Question display

- Further improve the UI for question display in the room when users are matched, perhaps by using markdown. This way, we could include images as well to further enhance the readability of the questions.

Overall User Experience

Gamification

- We could gamify our application such as having levels, and possibly experience points after completing questions and a set time limit for instance. Reaching a certain level would unlock new questions, or bonuses.

Deployment for production

Since our app is only able to support local deployment for now, one suggestion for improvement would be for us to deploy the backend services to the cloud. One such example could be to migrate our PostgreSQL database to a managed cloud service (like AWS RDS, Google Cloud SQL, or Azure Database for PostgreSQL) for better scalability, automated backups, and security features.

Learning points

1. **Communication**

   At the start of the project, we did not communicate effectively regarding the part of the code that we are working on, causing there to be frequent merge conflicts that have to be resolved. This caused us to waste precious time and effort.

   Learning point: After the first few weeks, we made sure to keep each other in the loop and work on separate parts of the code.

2. **Integrated testing**

   Initially, we overlooked the importance of running tests with each code push. During a critical phase, one of us pushed a seemingly minor code change without proper testing, assuming it wouldn't affect the system. Unfortunately, the change caused unforeseen issues that went unnoticed until a later time, and caused us to spend a significant amount of time debugging it.

   Learning point: It highlighted the necessity of implementing automated testing scripts that execute seamlessly upon code submission to GitHub. Conducting these tests ensures that even minor changes are thoroughly checked against the existing codebase, helping to identify and rectify issues early in the development cycle.

Appendices

Appendix A: Grading nice-to-haves

Nice-to-have outlined in Section 3.4 of PeerPrep Project Description	**Satisfied by which project feature outlined in this document**	**FRs / NFRs**
(Nice to have) Communication service – provides features such as chat, voice, and/or video calling among the participants in the room, once the users have been matched	Chat features in the collaboration-service	FR 2.8
(Nice to have) Obtaining another question on the fly during a session if the current question is too difficult.	Enhance question service to enable managing questions, retrieving questions on the fly during a session initiation.	FR 4.2
(Nice to have) The user should be able to manage questions by tagging them according to category, difficulty and completion status.	Enhance question service to enable managing questions, retrieving questions on the fly during a session initiation.	FR 4.7
(Nice to have) Learning pathway/history service – maintains a record of the past attempts e.g., difficulty levels or questions attempted	Question service and user service providing users with a way to track their question history. In this case, we have many nice-to-haves.	FR 3.1 FR 3.2 FR 3.3

(Nice to have) Syntax highlighting for multiple languages for increased usability; easier to code and debug their code.	Enhance collaboration service by providing an improved code editor. With, syntax highlighting for multiple languages.	FR 4.6
(Nice to have) Code execution for users; can see their output and debug accordingly for better user experience since they can see the output of their code.	Code execution: Implement a mechanism to execute code in a sandboxed environment, and retrieve+present the results in the collaborative workspace.	FR 4.5
(Nice to have) Demonstrate effective usage of CI/CD in the project.	Usage of Github Actions to test the services.	NFR 1.5