PEERPREP

CS3219 REPORT G47

Tan Jun Da, Yeo Wen Jue, Jovita Anderson, Tan Xuan Yun, Shanice Ng Wen Yi

Background

Peerprep is a team project created by a group of NUS CS students under the CS3219 team project. Peerprep allows peer matching and real-time collaboration in working on programming questions through code editors. Matched peers can communicate through chat to help each other work with the question(s).

Purpose

Peerprep aims to improve programming technical skills and teamwork by matching our users with peers online through our matching service to work on one or multiple problem questions as a team. Users in the same room can simulate an online assessment interview environment with their peers to gain strong confidence for their interviews. Peerprep's purpose is to create a community for users at any technical level to collaborate and work on technical questions like a duo in their comfort. With Peerprep, users will no longer need to face programming questions by themselves.

Individual Contribution

Name	Features Category	Contributions
Tan Jun Da	Assignment 3	Created additional roles for the users - User, SuperUser, Admin, SuperAdmin Created a Changing of User type page for Admins/SuperAdmins to change other user roles. Assign functionalities for each role so that there won't be any conflict of functionalities. Isolating and adding limits for different user roles functionality on the main page.
	M4	Created an exit for the current user in the room after his/her peer has exited the room. Debugging and fixing any issues with the service such as socket ID unable to allocate when the user first enters the home page.
	M5	Added buttons for backward navigation for the Profile page. Managed the layout of the UI for the room for the first few iterations.
	Non-tech	Managed report contents and layout. Wrote the main base of the report. (Purpose, background, NFR/FR, Microservices, Development process.) Wrote Collaboration Service.
Shanice Ng Wen Yi	Assignment 2a	Create and manage user profiles using Postgres - Registering user - Viewing Profiles - Updating profiles - Deregistering users from the platform Implemented a new command to automate the creation of new database and tables

		Design APIs for CRUD operations
	Assignment 4	Containerise question repository application Containerise user profile management application
	Assignment 5	Split features into microservices (into folders) - Created Dockerfile for each microservices Dockerizing all microservices - Created a docker-compose.yml file which consists of all the microservices and the necessary configuration as well as the environment files Bug fixes - Merging all mircoservices branches into dockerised branch and ensure all of the microservices are working in docker
	M6	Deploying the app on local staging environment (Docker-based)
	M5	Incorporated Bootstrap into the application to improve UI - Login, Create Account, VerifyOTP, ResendOTP, Profile, and Change User account pages
	Non-tech	Assignments 2 and 4 videos Design API, Ports and User-service diagrams for report Wrote User Service details for report
Jovita Anderson	Assignment 1	Created an initial web application to handle questions stored in local storage - Addition of new questions - Viewing individual questions - Deleting questions - Listing all questions in the required table format
	Assignment 2b	Updated questions are to be retrieved/stored from the cloud database (MongoDB atlas) - Designing APIs for the CRUD operations and maintaining question data
	M2	Updated data pushed into RabbitMQ queue with the necessary information for other services (Collaboration service)
	M3	Implemented question repository with CRUD operations through APIs, indexed by question creation time

	M4	Manage initial connection between matched pairs once users enter to room to allow communication Created the concurrent code editor between matched users
	M5	Incorporated Bootstrap into application to improve UI - Home, Profile History Table, CollaborationRoom, QuestionForm/List/Details/Queue, ConfirmModel, ErrorMessage Enhance the home page by incorporating a graphical representation, such as a line graph, pie chart, and calendar, to showcase users' question completion data.
	Non-tech	Assignment 1 video/README Wrote Question Service diagrams and details for report
Tan Xuan Yun	Assignment 3	Ensure seamless session management, implement mechanisms such with react context to allow users to stay on the same page even after a refresh Secure backend API routes by implementing authentication such as tokens to prevent unauthorized users
	M1	Include the user role in the JWT token payload to effectively capture and communicate the user's role throughout the authentication and authorization process.
	M3	Implement role-based authorization for question API routes, ensuring that only users with the designated roles have access to specific functionalities or endpoints.
	Non-tech	Assignment 3 video Assignment 3 README Contribute to report
Yeo Wen Jue	Assignment 3	Created the OTP token feature that authorizes account to access the PeerPrep application - Generation of OTP token upon account creation - Sending of OTP token via SMTP protocol - Ability to verify OTP tokens - Regeneration of OTP tokens for users that forgot

		their OTP token or failure to verify in set time
	Assignment 5	Implemented the backend portion of the matching service with the use of RabbitMQ’s CloudAMQP to create two queues, one (“user_queue”) for users to queue up, another (matched_pairs) to store matched users for collaboration service Added the timeout feature on the frontend for the various difficulty levels chose for queuing - Upon reaching the timeout time, dequeues users and removes them from the matching service
	M1	Implemented the frontend UI display of the history service as well as frontend UI display of the OTP token features
	M2	Implemented the matching service along with the timeout feature
	Non-tech	Wrote Matching Service and History Service contents.

Sub-Grp One Contribution

Name	Features	Contributions
Tan Jun Da	N1	Created a text-based chat communication through socket IO to allow users to communicate with their peers who belong in the same room.
	N4	Created an upvote system for each question to find the popularity of each user. Created a sorting feature based on the popularity in normal order, ascending order, or decreasing order. Created filtering of questions for selected tags for question types and difficulty of the questions. Created a dropdown box in the collaboration space to select questions on the fly.
Jovita Anderson	N1	Initialized socket IO communication between peers once they are matched from the queue Created “notification” in chat, to indicate when a user changes the selected question or changes to code editor language
	N5	Created a collaborative code editor with code formatting Created dropdown options for users to change their preferred coding language with syntax highlighting within the code editor for multiple languages such as Java, Python, C#, etc

Sub-Grp Two Contribution

Name	Features	Contributions
Shanice Ng Wen Yi	N11	Re-organise user-service to use routes and controllers to enhance modularity and maintainability Created an API gateway that redirects the microservices - \`npm run dev\` uses proxy to redirect - docker uses NGINX to redirect Enhanced the NGINX configuration by strategically setting headers to optimize the speed and efficiency of the API router. This refinement ensures a smoother and more responsive experience compared to conventional proxy redirection methods.
Tan Xuan Yun	N8	Create API test cases for User, Question, and History services using Mocha, Chai, and Chai-HTTP. Set up 'npm run test' and 'npm run test-ci' scripts to facilitate local and CI testing. Implement a GitHub Actions CI workflow for automated testing of the services, ensuring continuous integration and reliability.
Yeo Wen Jue	N2	Created a new feature, history, for users to keep track of their past collaboration attempts. During their collaboration attempt, **LIVE** edits are tracked and sent to the database for storage. Users can view their past attempts on their Profile page, where they can click into individual attempts to view their code.

Peerprep Functional Requirements

Note: Users specified in the table below are for all users (Not roles unless specified)

FR No.		Requirements	Sprint (Weeks)	Priority Level
FR1	1.1	User interface for the user to interact with	13	High
	1.2	User information storage for the user	1	High
	1.3	Users can log in with their user information	1	High
	1.4	Users can register a new account	1	High
	1.5	Users can update their username	1	High
	1.6	Users can update and change their password	1	High
FR2	2.1	Users can see the questions	1	High
	2.2	Question storage for the questions	1	High
	2.3	Ability to update the questions	1	High
	2.4	Ability to delete the questions	0.5	High
	2.5	Ability to add questions	0.5	High
	2.6	Ability to tag the questions by their difficulty (Easy, Medium, Hard)	1	High
	2.7	Ability to tag the questions by their categories	1	Medium
	2.8	Ability to tag the questions	1	High
FR3	3.1	Users are given a user role after the account creation	1	Medium
	3.2	Multiple roles for each user	1	High
	3.3	Superadmins can assign any roles to any accounts	1	High
	3.4	Admins can assign Superuser/User roles to any accounts	1	High
	3.5	Superadmin, Admin, and Superuser roles should	1	High

		be able to add/update/delete questions		
	3.6	The user role should not be able to add/update/delete questions	1	High
FR4	4.1	Users should stay logged in after reopening the browser	2	High
	4.2	Only authenticated users should be able to log in	1	High
FR5	5.1	OTP to be sent through email for authenticated users	2	Medium
	5.2	Users can verify their OTP	2	High
FR6	6.1	Users can match with another user	1	High
	6.2	Users should be able to leave the queue	1	Medium
	6.3	Users will be kicked out of the queue after 30 seconds	1	Medium
FR7	7	Users can have real-time collaboration on the code editor with another user	2	High
FR8	8.1	Users can view their History of questions they did in the collaboration space	2	Medium
	8.2	Users can view their code editor for the questions they did in the collaboration space	1	Medium
FR9	9	Users can change the programming language type they wish to use for the code editor	1	Medium
FR10	10	Users can chat with another user in the collaboration space	1	Medium
FR11	11.1	Users can upvote their favorite questions	1	Medium
	11.2	Users can sort the questions by their popularity	1	Medium
	11.3	Users can filter the questions by their tags	1	Medium
	11.4	Users can filter the questions by their difficulty	1	Medium

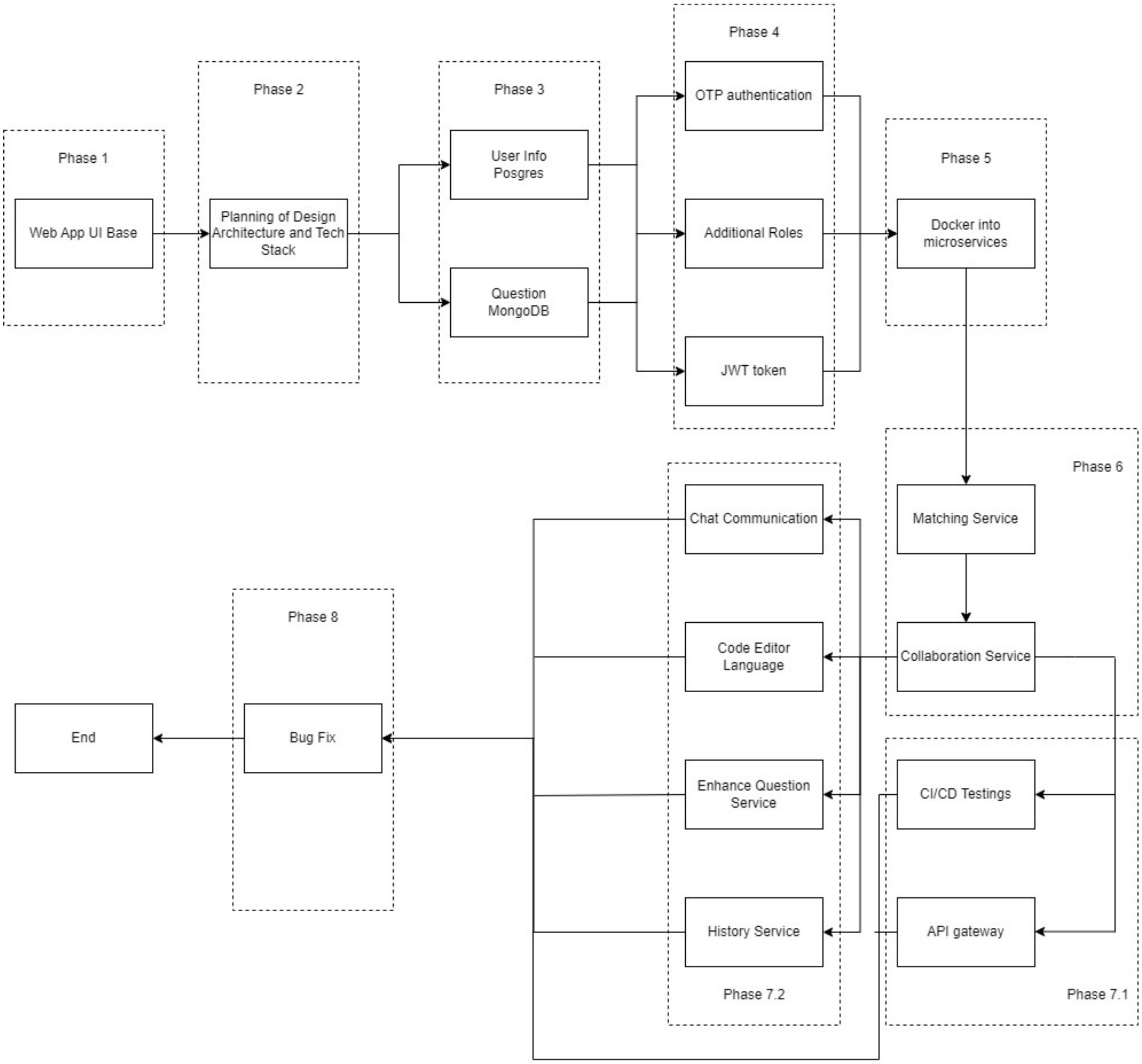
Peerprep Non-Functional Requirements

Note: Users specified in the table below are for all users (Not roles unless specified)

NFR No.	Requirements	Sprint (Weeks)	Priority Level
NFR1	Services should be separated into independent microservices	2	High
NFR2	UI should work on different sizes of tab	2	Low
NFR3	Peerprep should work on any browser	1	Medium
NFR4	The questions database should be able to hold all the questions	1	High
NFR5	The user information database should be able to hold all the users	1	High
NFR6	Nginx controller to control the rerouting of API gateway	1	Medium
NFR7	CI/CD for Peerprep for automated testings	3	Medium

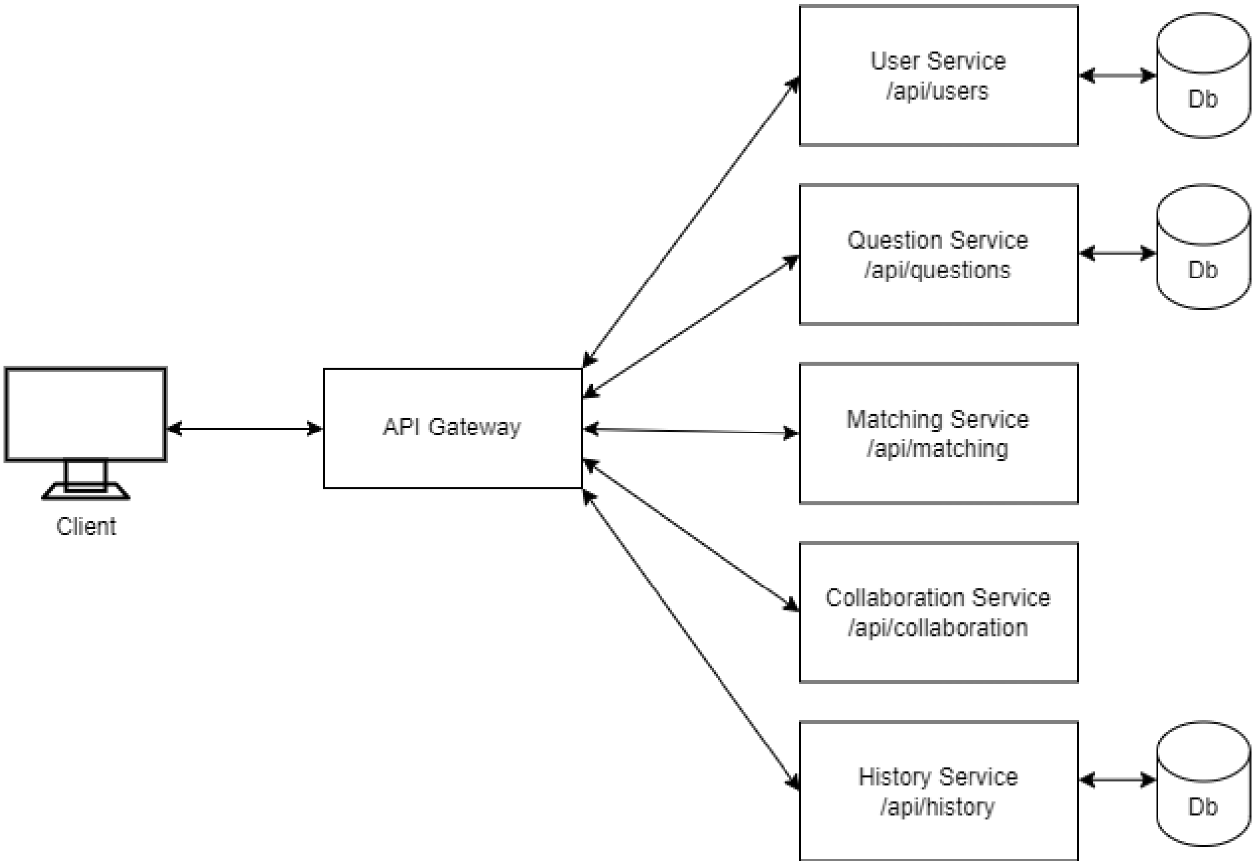
Development process

During this development, our group used the waterfall process development cycle because we were familiar with the domain and solutions and knew the requirements for the project. Since it is a greenfield project, we had to create it from scratch. We started with design implementation and started the project off with the base User Interface of the web application. We slowly add new features till the end when we bug fixed and deploy it. All the phases are done sequentially but the subtasks are done in parallel.



Microservices Architecture

Peerprep is split into 5 microservices. Splitting it into 5 microservices helps to isolate each service for security and independence from any failure. Each of these services' purpose and functionality will be discussed later in the report.



Reasons for using microservice architecture instead of monolith:

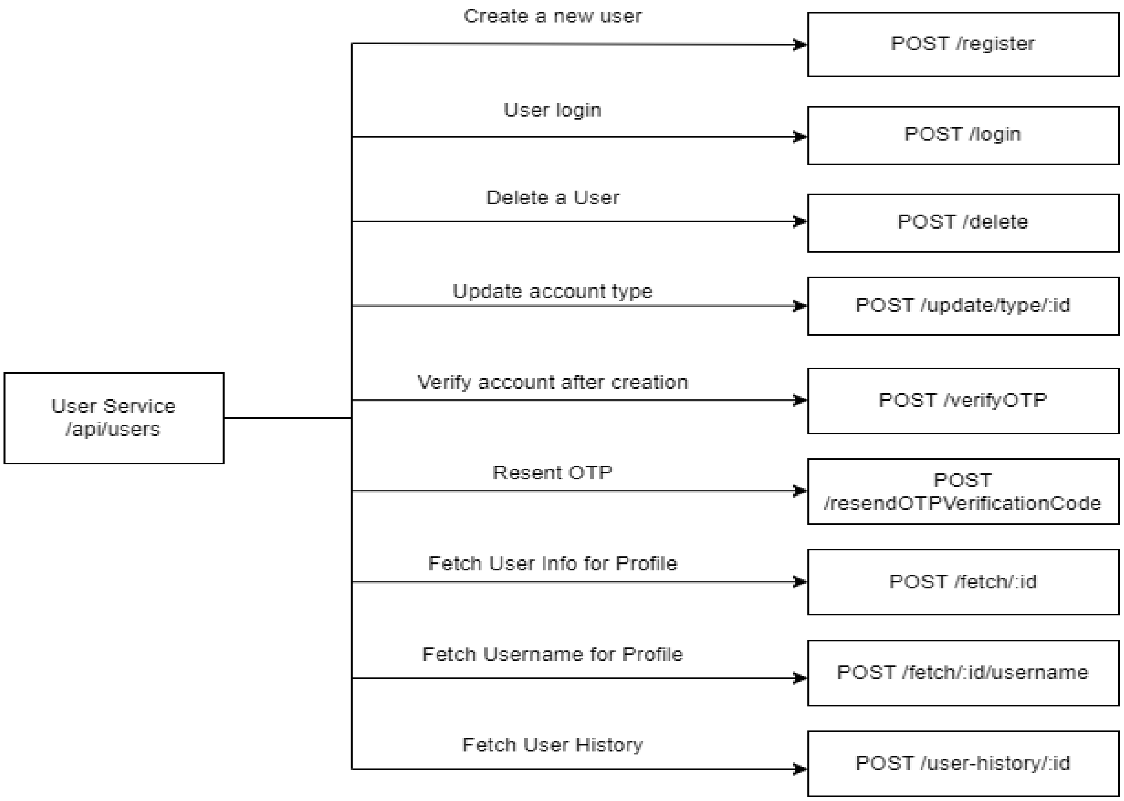
- Decentralized Development: Different teams can independently work on distinct microservices, fostering parallel development and specialization.
- Coupling Reduction: Microservices architecture minimizes interdependencies between modules, promoting loose coupling and isolating the impact of changes within specific services.
- Enhanced Testing and Development: The isolated nature of microservices facilitates easier testing, development, and maintenance of individual components, contributing to overall system agility and robustness.

User Service

User Service is responsible for handling the user data naming login credentials, authentication status, user ID, username, and user roles by allowing CRUD operations. These data are important to deal with unauthorized usage or operations on the web application and for the frontend view. The User Service adds a layer of security by automatically hashing users' passwords upon signing up for an account. This robust measure ensures the protection of sensitive user data, enhancing the overall security posture of our application.

Furthermore, the User Service leverages a REST API, thereby enhancing scalability and facilitating seamless integration with other components of the application.

Finally, the User Service API routes are protected with role-based authorization using JWT tokens. This ensures that only authenticated users who verified their email possess the privilege to access and utilize these routes.



The User Service is designed to seamlessly interact with the front-end application, through a set of 9 distinct endpoints, each meticulously configured to correspond with specific HTTP methods and routes to accommodate a wide range of user-related operations.

- **POST /register**: Creates a new user account saved in postgresql
- **POST /login**: Verifies the email and password keyed in by the user to login
- **POST /delete**: Deregister the user from the platform, deleting in postgresql
- **POST /update/type/:id**: Update account type according to the role provided
- **POST /verifyOTP**: Checks if entered otp matches the otp send after creating an account
- **POST /resendOTPVerificationCode**: User can generate new OTP code
- **POST /fetch/:id**: Fetch users' accounts information for display in Profile page
- **POST /fetch/:id/username**: Fetch users' username to display in Profile page
- **POST /user-history/:id**: Fetch users' previous attempts to display in Profile page

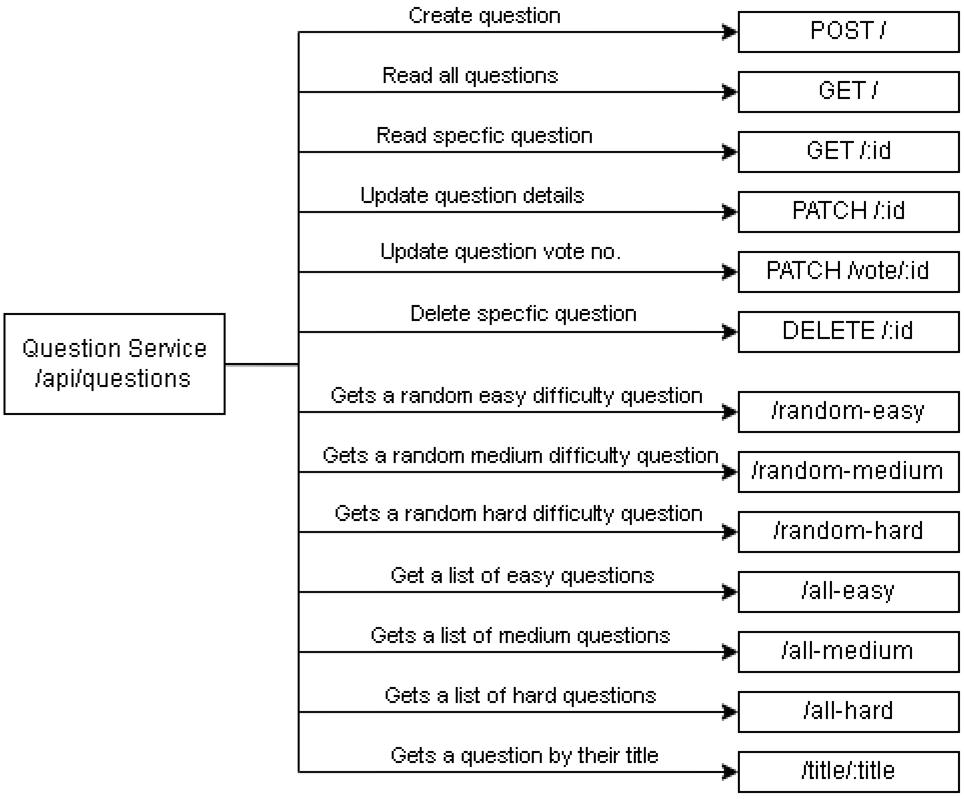
Reasons why we use **POST** for user-service:

1. Security Concerns: When performing operations that involve user accounts, such as registration, login, and updates, using POST helps in securing sensitive information. POST requests allow us to send data in the request body, which can be encrypted, providing an extra layer of security.
2. Payload Flexibility: POST requests allow us to send a specific payload in the request body. This is useful for operations where a significant amount of data is needed, such as during registration or when updating user account types.

Question Service

Question Service is responsible for storing the questions and allowing CRUD operations on question data using REST APIs. Question Service has a port that links it to the frontend application to allow the showcase of questionnaires and perform any Creation/Deletion/Updating of questions.

By employing REST APIs, the Question Service effectively decouples the client and server aspects of the application. This separation of concerns ensures scalability, as it allows the client and server to evolve independently.



Each endpoint is tailored to interact with the front-end application, enabling users to perform necessary actions such as viewing, creating, modifying, or deleting questions depending on their account role (authentication). For the Question Service there are a total of 13 endpoints, each mapped to a specific HTTP method and route that collectively cater to a variety of operations on questions:

- **POST /** : Allows for the creation of new questions
- **GET /** : Provides a list of all questions, addressing the 'Read' operation for multiple entities.
- **GET /:id** : Retrieves a single question by its unique identifier, enabling the 'Read' operation for an individual question.
- **PATCH /:id** : Updates the details such as the title, complexity, category and description of an existing question
- **PATCH /vote/:id** : Specifically designed to update the vote count of a question
- **DELETE /:id** : Deletes a specific question

Additionally, the service provides 7 more endpoints to fetch questions based on their difficulty levels and titles which is used in the collaboration room to retrieve questions based on their selected difficulty.

- **/random-easy**, **/random-medium**, **/random-hard** : Retrieves a random question classified under easy/medium/hard difficulty.
- **/all-easy, /all-medium, and /all-hard** : Retrieves a collection of questions filtered by their difficulty levels, allowing users to access a broader range of questions within a specific complexity category.
- **/title/:title** : Retrieves a question by matching titles

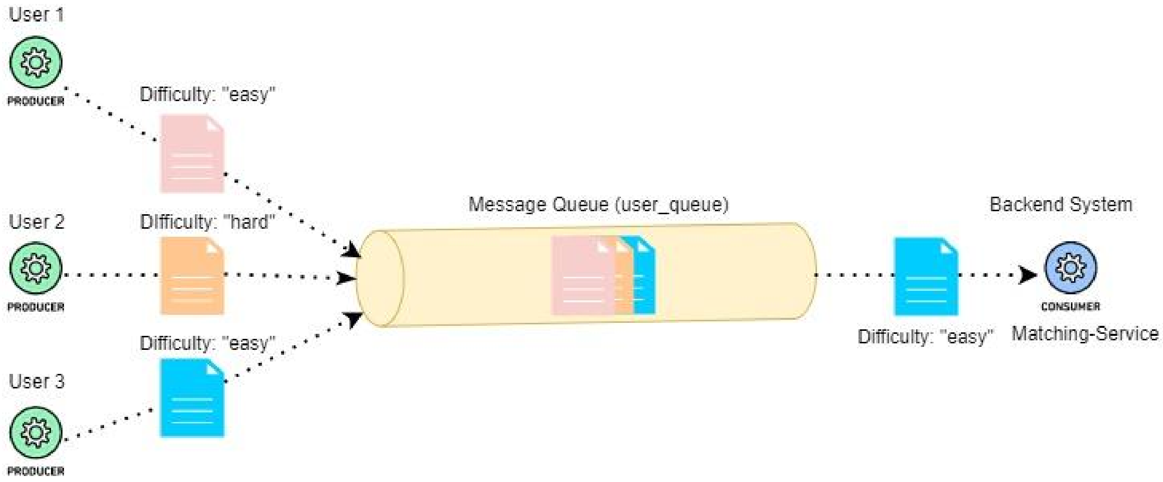
Certain endpoints relating to the updating and deleting of questions are only accessible to users with the role of superuser, admin and superadmin.

Reason for using RESTful API instead of monolith:

- (Separation of concerns) Client and server are separated therefore systems are less tightly coupled
- (Scalability, Reliability) Server is bound by the number of concurrent requests and not the number of clients interacting
- (Generality to the component interface) Uniform way of interacting with a given server

Matching Service

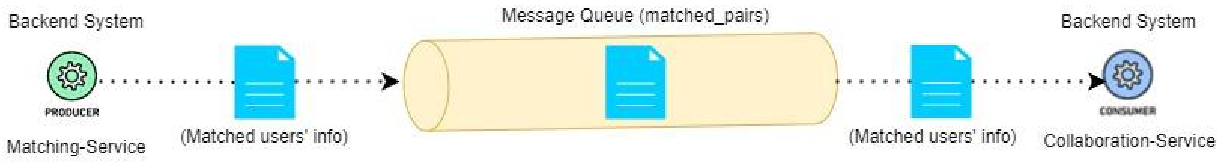
Matching Service is responsible for the matching of the users based on the difficulty level of questions ('Easy', 'Medium', 'Hard') they have chosen. Each difficulty level comes with its own timeout time ('30s', '45s', '60s'), which upon hitting, will remove the users from the queue and prompt them to queue again. The timeout time allocated for the various difficulty levels is designed to improve users' experience by reducing the need for them to constantly queue when a match cannot be found within the designated time. Hence, the time allocation for queuing players is set to increase the amount of time to wait before queuing a user as the level of difficulty they have chosen increases as it would be generally more difficult to find a player who chose a "hard" question as compared to an "easy" question.



During the queuing process, users act as **producers** and send their information to the **message queue** ("user_queue") on CloudAMQP, supported by RabbitMQ, one of the numerous popular message queuing systems. The backend system acts as the **sole consumer** that constantly retrieves information from this message queue and performs either one of the following actions:

1. Add the user into its queue dictionary if there aren't any existing users with the same difficulty level in its existing queue dictionary
2. If there is an existing user with the same difficulty level as the current user, remove the user in the queue dictionary pair both users up to act as the **producer,** and send the matched pair's information to **another message queue** ("matched_pairs")

on CloudAMQP. This information will then be **consumed** by the collaboration service.



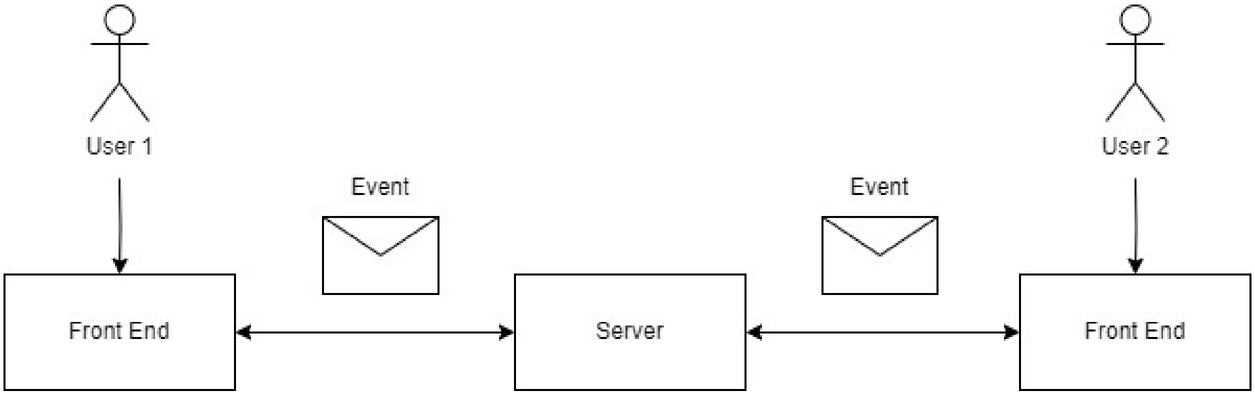
Through the information above, it is shown that the matching service makes use of the following message patterns to facilitate the queuing system: **single receiver** and **one-way Point-to-Point** channel. In particular, the **single receiver** is used in the earlier portion of the matching service where multiple users could queue up at the same time and send their information to the message queue and this would be consumed only by the backend system. On the other hand, the **one-way Point-to-Point** channel is adopted in the latter part when users are matched and there is a need to send it to a message queue so that the collaboration service could use this information at a later time. The use of these asynchronous message passing supports asynchronous persistent communication, acting as an intermediate storage for messages when receivers are inactive/unavailable to process these messages. This guarantees all messages sent by the senders (producers) will eventually be in the message queue where the recipient consumes.

Reason for Event-Driven communication instead of Request-Response communication:

- Employing an asynchronous pattern in the system's design enhances efficiency by allowing tasks to execute independently.
- The integration of real-time data processing elevates the system's responsiveness, ensuring that users can be matched promptly.
- Enforces loose coupling between components enhances system flexibility and maintainability

Collaboration Service

Collaboration Service is responsible for the collaboration space after matching. This service is dependent on the matching service to match the users first before it can connect the users through a socket room. This room will hold two matched users together and the two users can work on their collaborative space. Collaboration service also supports other features such as chatting and changing languages.

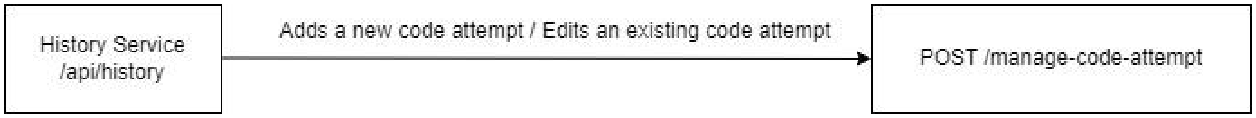


Collaboration Service is separated from the other services to reduce any coupling. Features requiring any collaboration will be handled by this service on the backend. This collaboration uses an Event-Based architecture where the actions on the Front End are highly dependable on the actions received. This service allows the server to keep track of multiple rooms at a time with two users in each room using the socket room ID.

History Service

History Service is responsible for recording and saving all code attempts in the collaboration space and displaying them on users' profiles to allow them to review their past attempts. Our team decided to separate the History Service from the User Service as it has database records that are different from the ones used for the User Service. By doing so, we are adopting the **Separation of Concern** (SOC) design principle. This reduces coupling within the user service and runs both User Service / History Service individually, allowing both services to continue running even when one is down. This also helps to reduce the traffic flow into the User Service, thereby improving User Service.

Like other services, the History Service also employs **REST** APIs to effectively decouple the client and server aspects of the application. This provides scalability and usability benefits to our application to allow it to support a wider range of consumers.


History Service
/api/history
Adds a new code attempt / Edits an existing code attempt
POST /manage-code-attempt

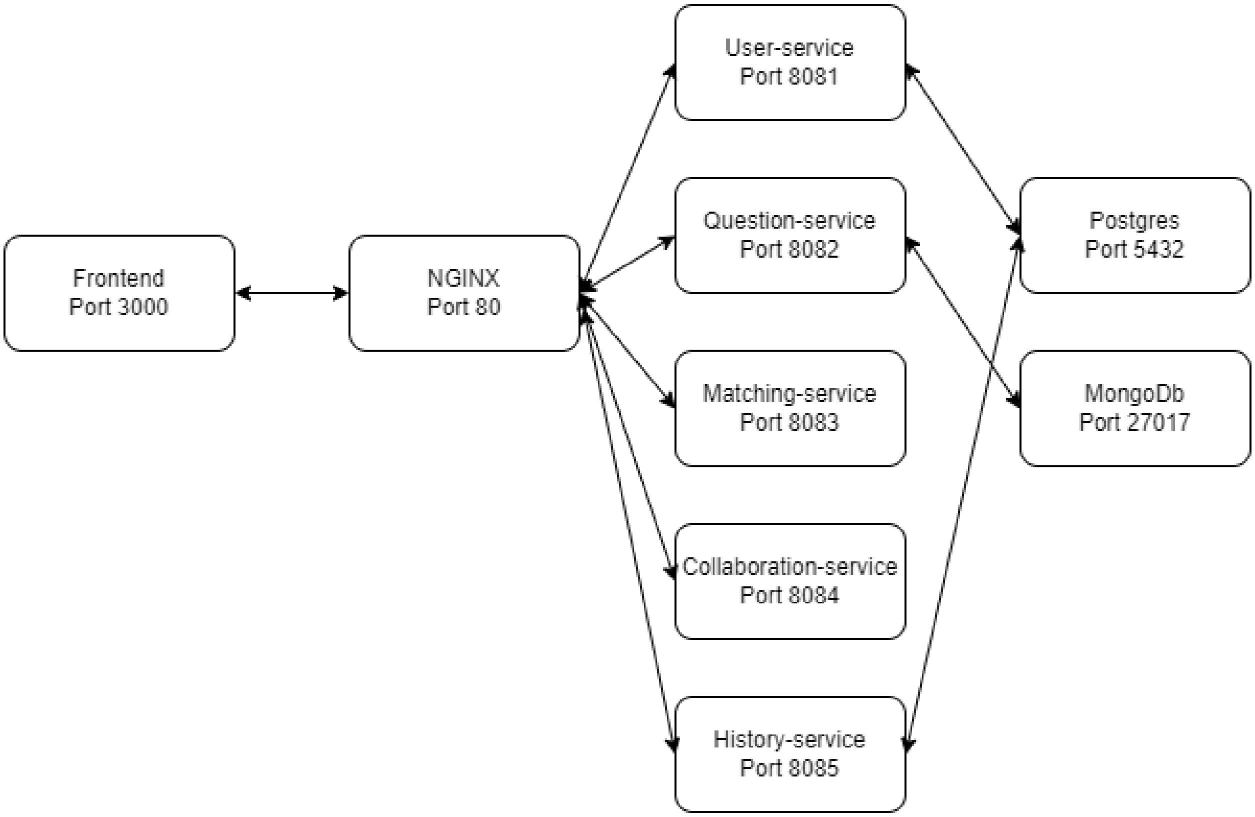
However, unlike other services, the History Service has only one endpoint at this moment as its only use is to keep track of all collaboration attempts. In the future, this is scalable, and more endpoints could be added, when more features are added to the application, such as coding competitions, to keep track of the various records. Below describes the mapping of the endpoint to the HTTP method and route:

- **POST /manage-code-attempt** : Enables the addition of a new code attempt / updating an existing collaboration attempt within the PostgreSQL database

Development Considerations

Technology use	Consideration
Frontend: ● Languages: HTML, CSS, Javascript ● Web framework: Bootstrap, ReactJs Frontend back: Node.js	● Ensure cross-browser compatibility and responsiveness with Bootstrap. ● Leverage the component-based architecture of ReactJs for efficient UI development. ● Node.js for server-side rendering and handling backend interactions.
Question Service: ● Database: MongoDB (Nosql)	● Nosql is suitable for flexible schema design, ideal for handling various question structures.
User Service: ● Database: Postgres (SQL) ● OTP storage: MongoDB(Nosql)	● SQL databases like Postgres provide transactional integrity, suitable for user-related data.
History Service: ● Database: Postgres (SQL)	● SQL databases are suitable for storing historical data with well-defined relationships
Collaboration Service: ● Connection: Socket.io	● Socket.io enables real-time bidirectional communication, suitable for users to collaborate on coding.
DevOps (CI/CD): Github actions	● GitHub Actions integrated directly into the GitHub repository and automation of build, test, and deployment processes
Containerization software: Docker	● Docker widely preferred as container software
Project Management Tool: Github	● Using Github for project management offer a centralized platform that enhances collaboration, version control, and issue tracking

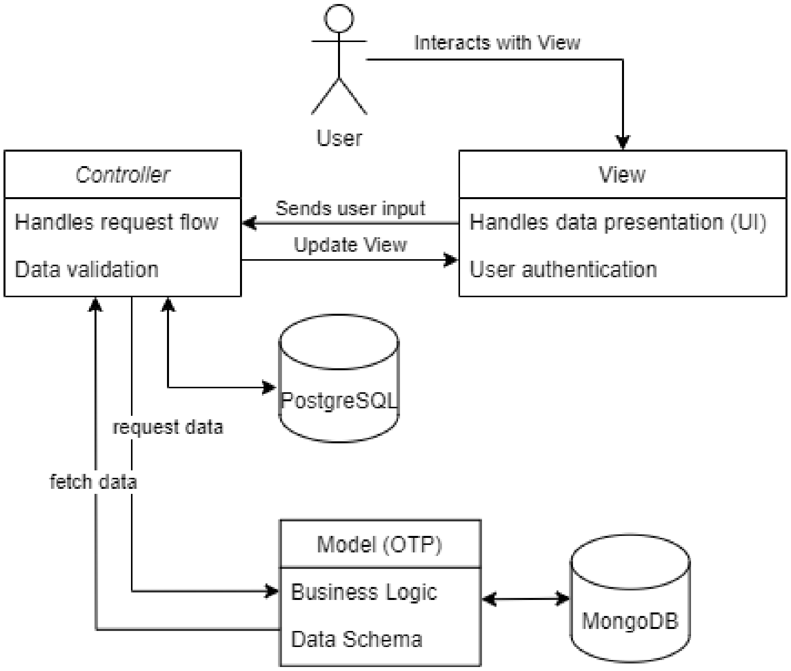
Microservices Ports:



In our microservices architecture, each microservice operates on its dedicated port, ensuring encapsulation and modularity. These microservices, serving distinct functions, are orchestrated by an NGINX server acting as a reverse proxy. The NGINX server is responsible for listening to incoming requests and efficiently routing them to the appropriate microservices based on their designated ports and endpoints, effectively serving as a gateway to our system. Furthermore, for the client-side interaction, our frontend application, deployed within a Docker container, initiates requests to the NGINX server container by specifying the endpoint of the intended microservice. This design facilitates scalability, fault tolerance, and ease of management, enabling seamless communication between the front end and the diverse microservices within our containerized environment.

User Service

The User Service architecture follows the Model-View-Controller (MVC) pattern, which divides the application into three interconnected components. This approach fosters a structured code organization, making application development and maintenance more manageable.



Model: Defines the format of the data housed in the database and encompasses the Mongoose model along with the connection to MongoDB, outlining the structure of both data storage and business logic.

View: Comprises React UI components responsible for overseeing the presentation of question details on the user interface, managing user inputs, and displaying data in a user-friendly format. This component engages with the user, collects input, and subsequently forwards it to the Controller for further processing.

Controller: Encompassing functions that manage incoming requests to perform operations such as creating, reading, updating, or deleting user accounts. Subsequently, it communicates the relevant responses to the View, serving as the intermediary that translates user actions (such as clicking a button to register) into the Postgres database.

Furthermore, it also communicates with the Model to do the relevant updates to MongoDB for any OTP-related actions.

Structure of Accounts table in Postgres database:

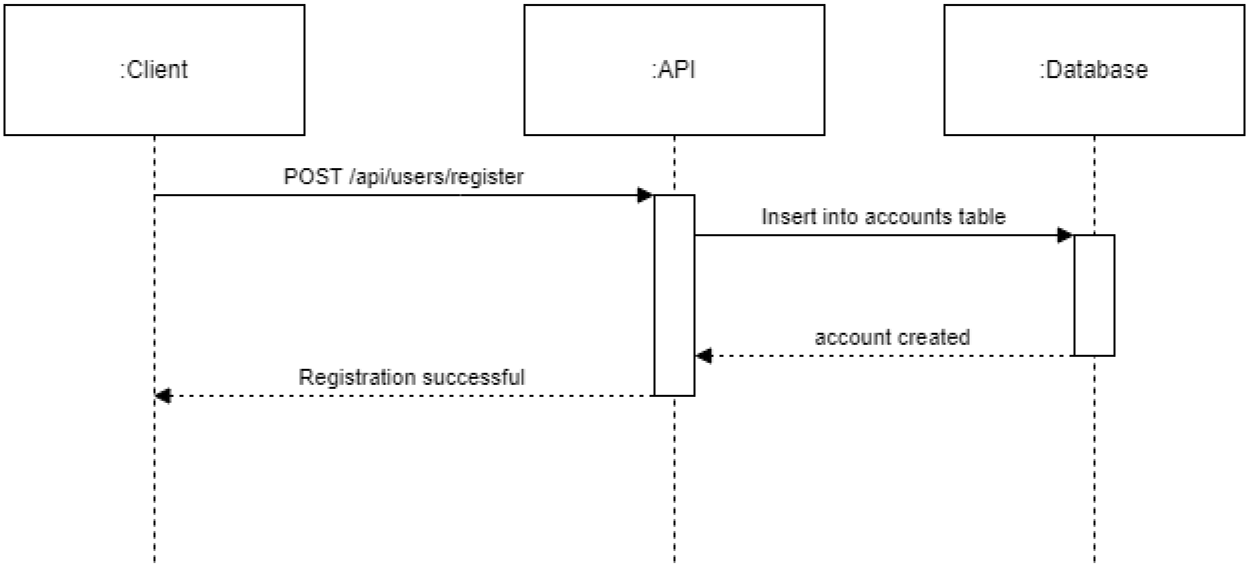
Accounts	
integer	user_id
char varying (255)	username
char varying (255)	email
char varying (255)	password
char varying (255)	account_type
boolean	authentication_stats

Structure of OTP table in MongoDB database:

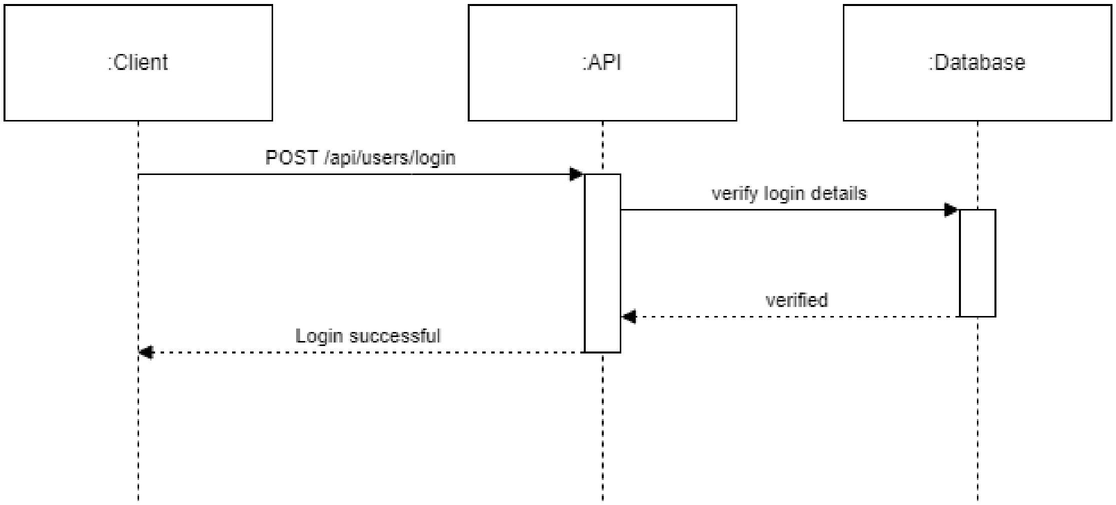
otp	
String	userId
String	email
String	otp
Date	createdAt
Date	expiresAt

Sequence Diagrams for CRUD operations for user account:

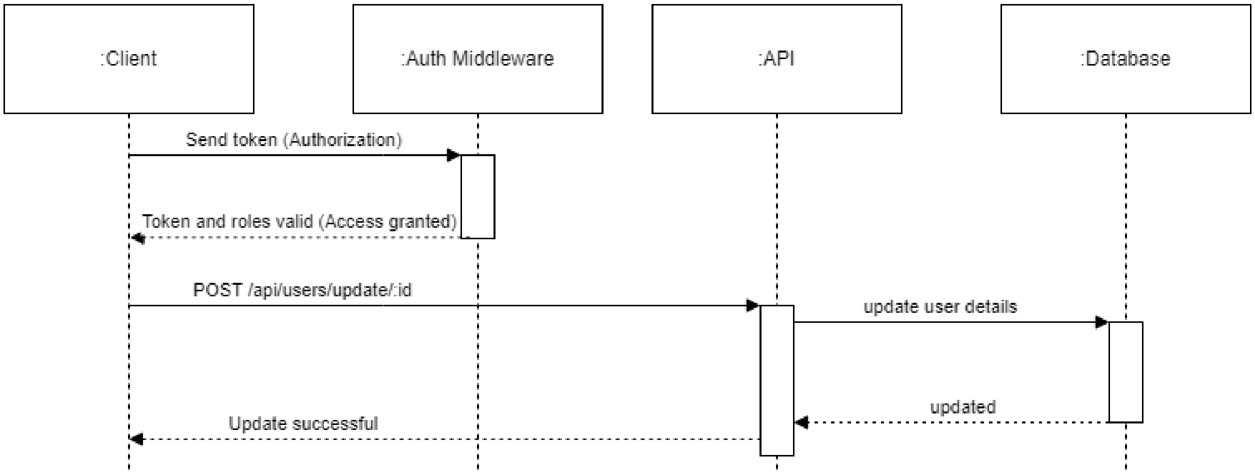
Registering for a new user:



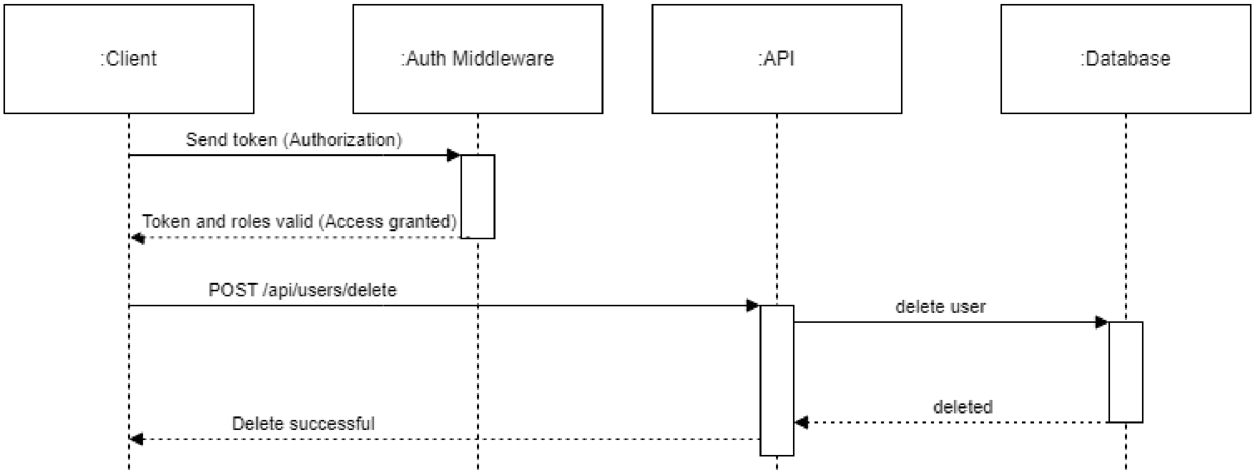
User login:



Update user details:

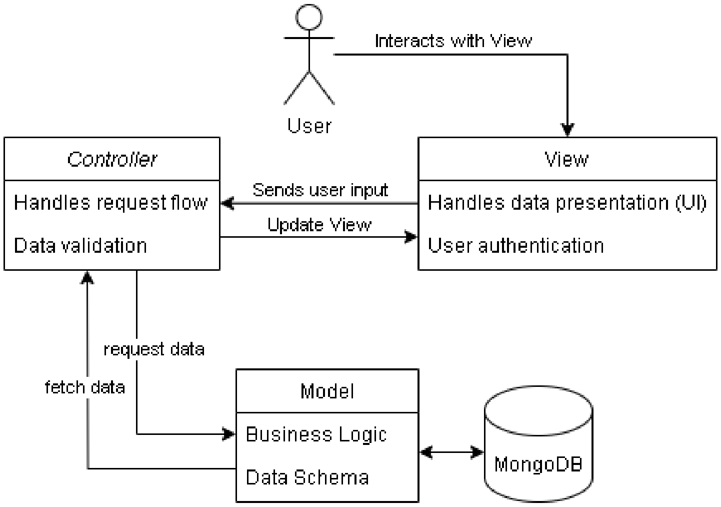

:Client
:Auth Middleware
:API
:Database
Send token (Authorization)
Token and roles valid (Access granted)
POST /api/users/update/:id
update user details
updated
Update successful

Delete user account:



Question Service

The Question Service architecture adheres to the **Model-View-Controller (MVC)** pattern. It separates the application into three interconnected components, thereby promoting organized code structure and enabling more manageable application development and maintenance.

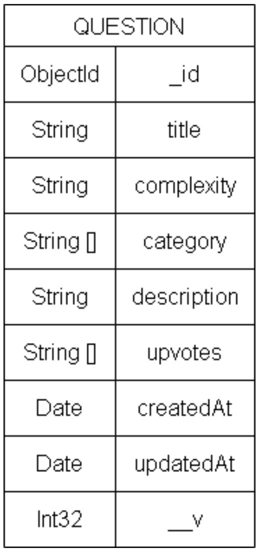


Model: Specifies the structure of the data stored in the database and business logic consisting of the Mongoose model and connection to MongoDB.

View: Consists of React UI components that manage the rendering of the question details on the UI, handling user inputs, and presenting the data in a user-friendly manner. It interacts with the user, gathering input that is then sent to the Controller to be processed.

Controller: It contains functions that handle the incoming requests, interact with the Model to create, read, update, or delete data, and then send back the appropriate responses to the View. The intermediary translates the user's actions (e.g., clicking a button to fetch a question) into operations on the Model.

Structure of the data stored in the MongoDB database

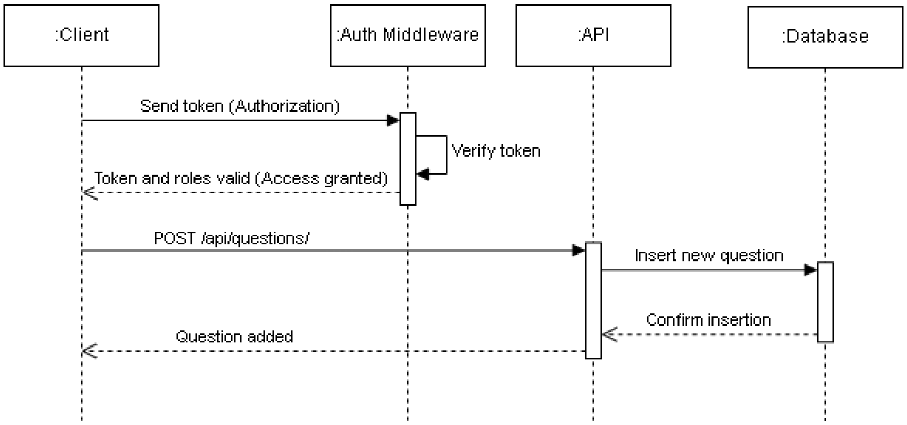


QUESTION	
ObjectId	_id
String	title
String	complexity
String []	category
String	description
String []	upvotes
Date	createdAt
Date	updatedAt
Int32	__v

Attributes like the title, complexity, category, description, and upvotes are directly added or manipulated by the users. Conversely, attributes such as _id, createdAt, updatedAt, and __v are automatically generated by MongoDB. These fields are not manually handled by us but are intrinsic to the MongoDB system, serving as unique identifiers and timestamps that track the creation and modification of each question record.

Sequence Diagram of Question CRUD operations

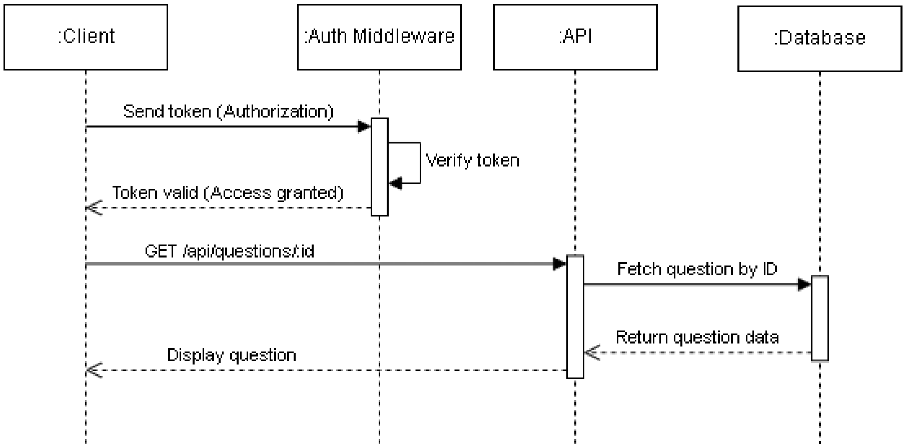
Create question



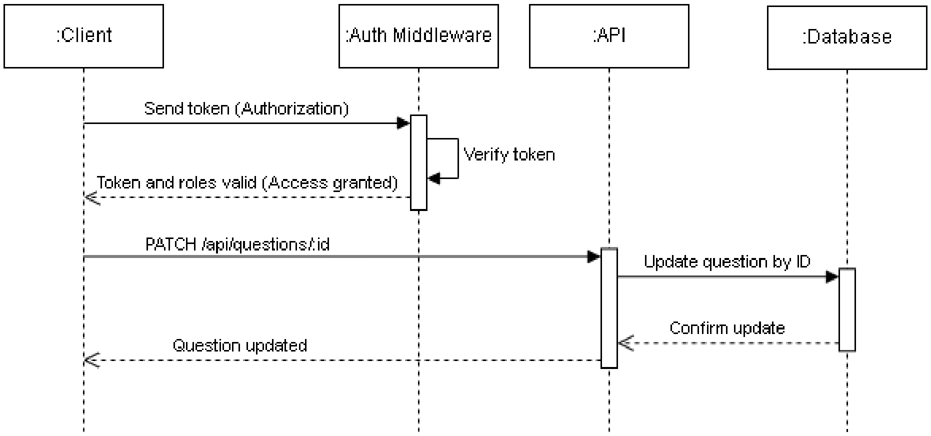
Starting with the client sending an authorization token to Auth Middleware, which validates the token and grants access. Upon successful authentication, the client's request to read all questions is processed by the API (Controller), which interacts with the Database (Model) to fetch the data. Once the data is retrieved, it is sent back through the API to the client (View), where it is ultimately displayed in the user interface.

The operation of the Read/Update and Delete question is similar to the Create question.

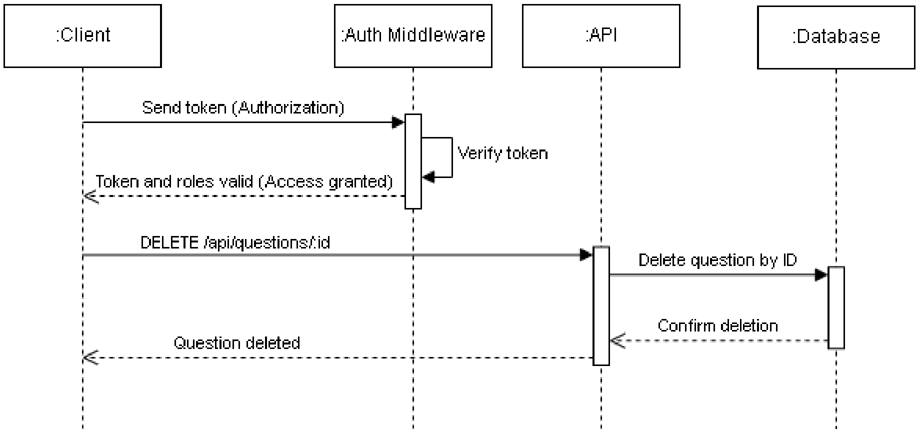
Read specfic question



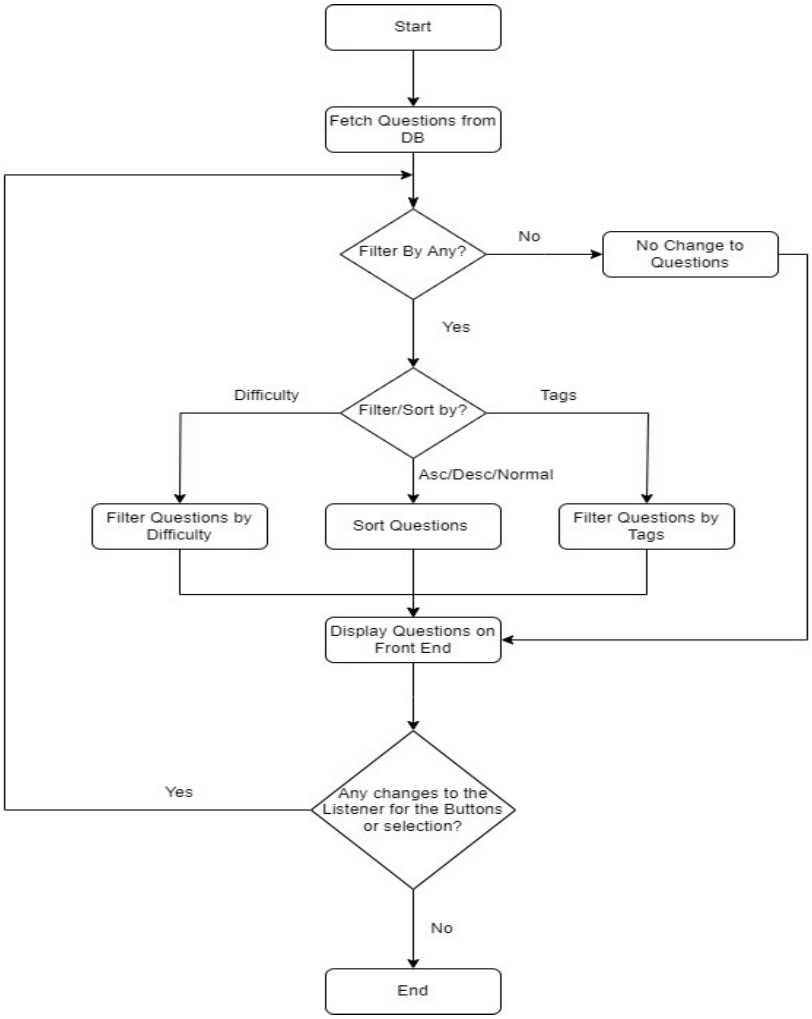
Update question



Delete a question



Enhanced Question Service

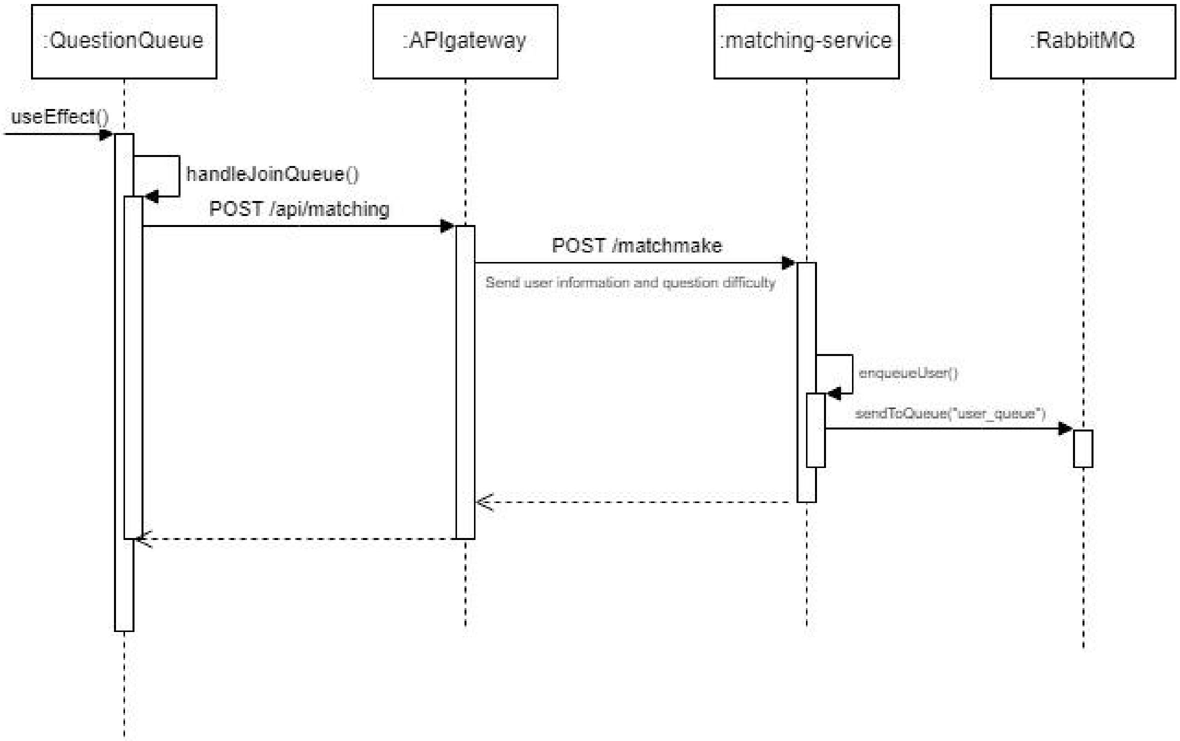


The questions are tagged according to their specific roles but they are not useful if we cannot filter them. Hence, this was one of our design considerations to have a better question service for the users. In the later iterations, an upvote system is created to allow users to determine the popularity of the questions added. This requires a new table for the database as the previous ones were used for the assignments and will break any functionalities. Creation of new routes in the middleware and updating the model is required for the frontend view to go through the controller and achieve any updates required with the model before it gets reflected in the database. The enhanced question service also makes use of the Model-View-Controller design pattern so that questions give a clean structure that provides modularity as all the components have a Single Responsibility and helps to separate any concerns among the components. It allows developers to independently work on the components without affecting other components in the question service. As the original question service has the MVC design pattern, you can see the scalability in the MVC for the enhanced question service. The enhanced question service also allows the filtering of questions by tags, difficulties, and popularity. Users in the collaboration room are allowed to pick questions “on the fly” based on their selected difficulty level. The MVC provides a good base to scale certain parts of the project well. Changes can be made easily without affecting the original question service.

Matching Service

The matching service architecture follows the message patterns, **Single Receiver,** and **One-way Point-to-Point** channel as mentioned earlier. The following section shows Sequence Diagrams of the matching service at work when users queue up for matches.

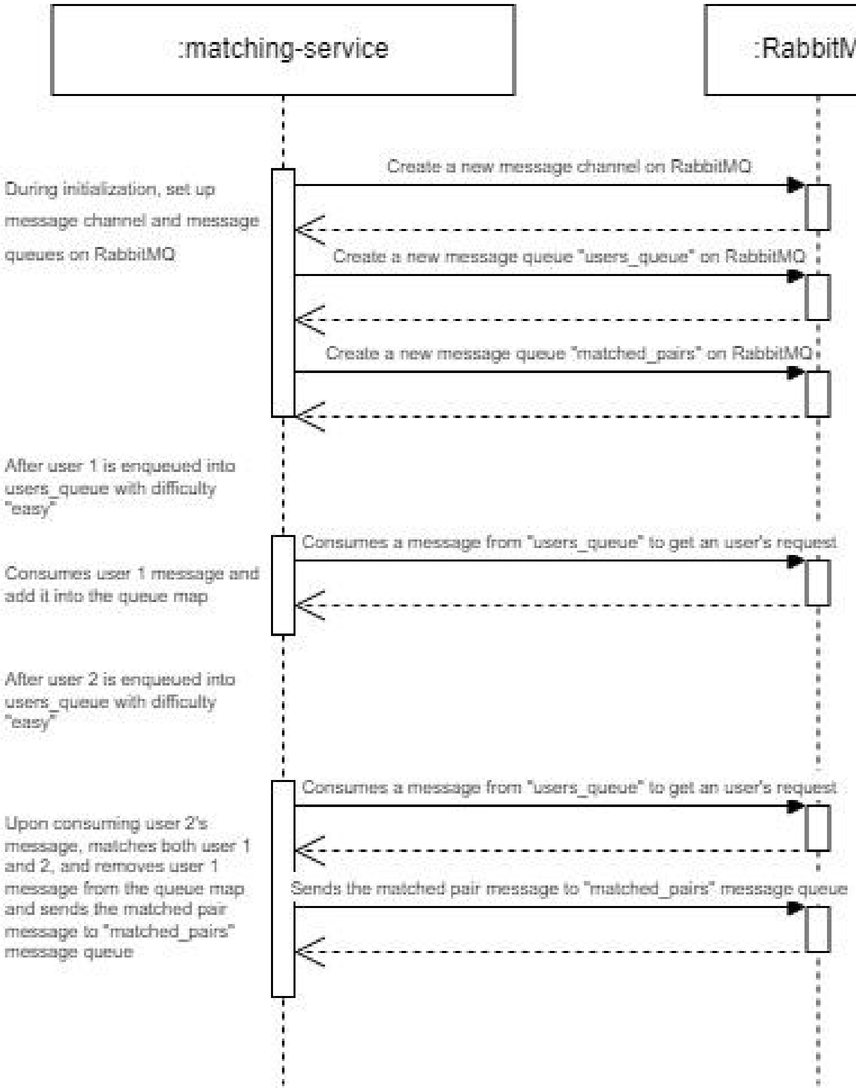
Sequence Diagram of user queuing for a match:



Upon receiving a signal from the frontend (QuestionQueue) of the user's request to queue up, it sends a POST request containing the user's information, mainly username, email address, question difficulty level, and socketId, to the backend going through the API gateway to the matching service. The matching service then acts as a producer, connects to RabbitMQ's message channel, and sends the user's information to the message queue ("user_queue").

*Do note that the socketId's value is generated through collaboration service via socket-io during the initialization of the matchmaking service and has been excluded from this sequence diagram.

Sequence Diagram of Matching Service retrieving information from RabbitMQ and match users:

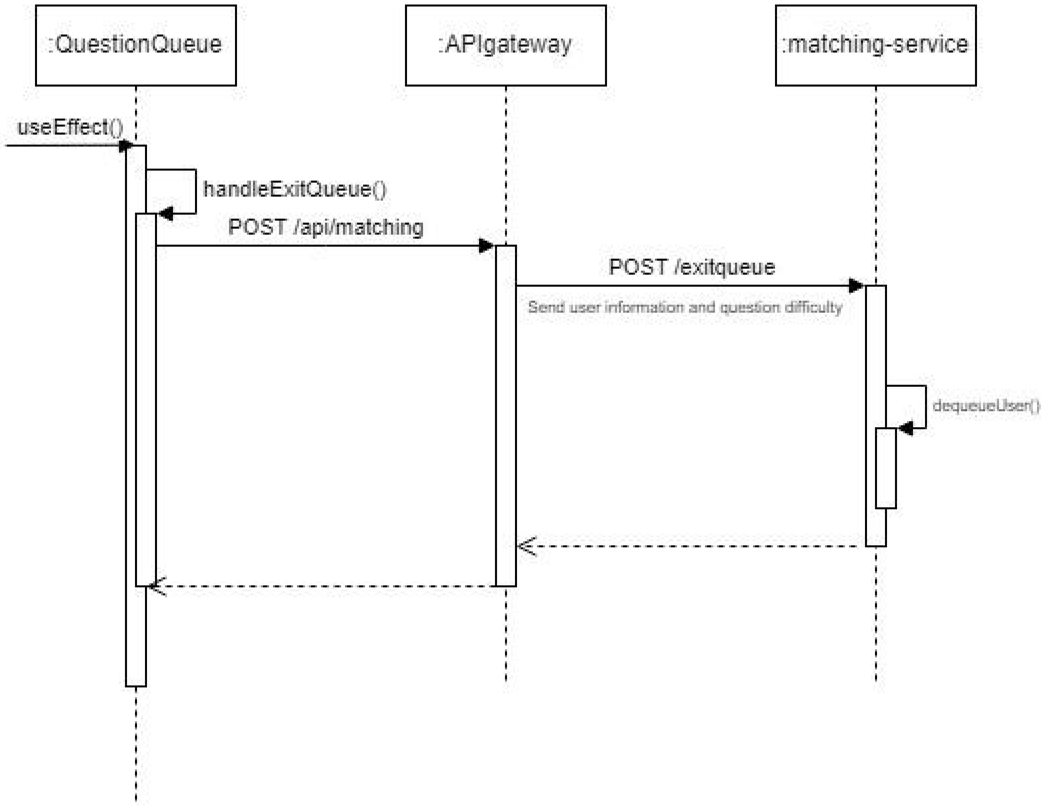

:matching-service
:RabbitMQ
During initialization, set up message channel and message queues on RabbitMQ
Create a new message channel on RabbitMQ
Create a new message queue "users_queue" on RabbitMQ
Create a new message queue "matched_pairs" on RabbitMQ
After user 1 is enqueued into users_queue with difficulty "easy"
Consumes a message from "users_queue" to get an user's request
Consumes user 1 message and add it into the queue map
After user 2 is enqueued into users_queue with difficulty "easy"
Consumes a message from "users_queue" to get an user's request
Upon consuming user 2's message, matches both user 1 and 2, and removes user 1 message from the queue map and sends the matched pair message to "matched_pairs" message queue
Sends the matched pair message to "matched_pairs" message queue

During the initialization phase of the matching service, it will first connect to RabbitMQ and set up a new message channel with two message queues, “users_queue” for users to send their information over during queuing and “matched_pairs” for keeping track of matched users and for collaboration service.

Afterward, when users start queuing up for matches, the matching service will consume from the message queue “users_queue” to retrieve information of users queuing up and add into its queue map that stores respectively key: question difficulty level and value:

users’ particulars (username and email). This helps to match users of the same difficulty level and send both users' information as a single message to the message queue “matched_pairs” for use by the collaboration service.

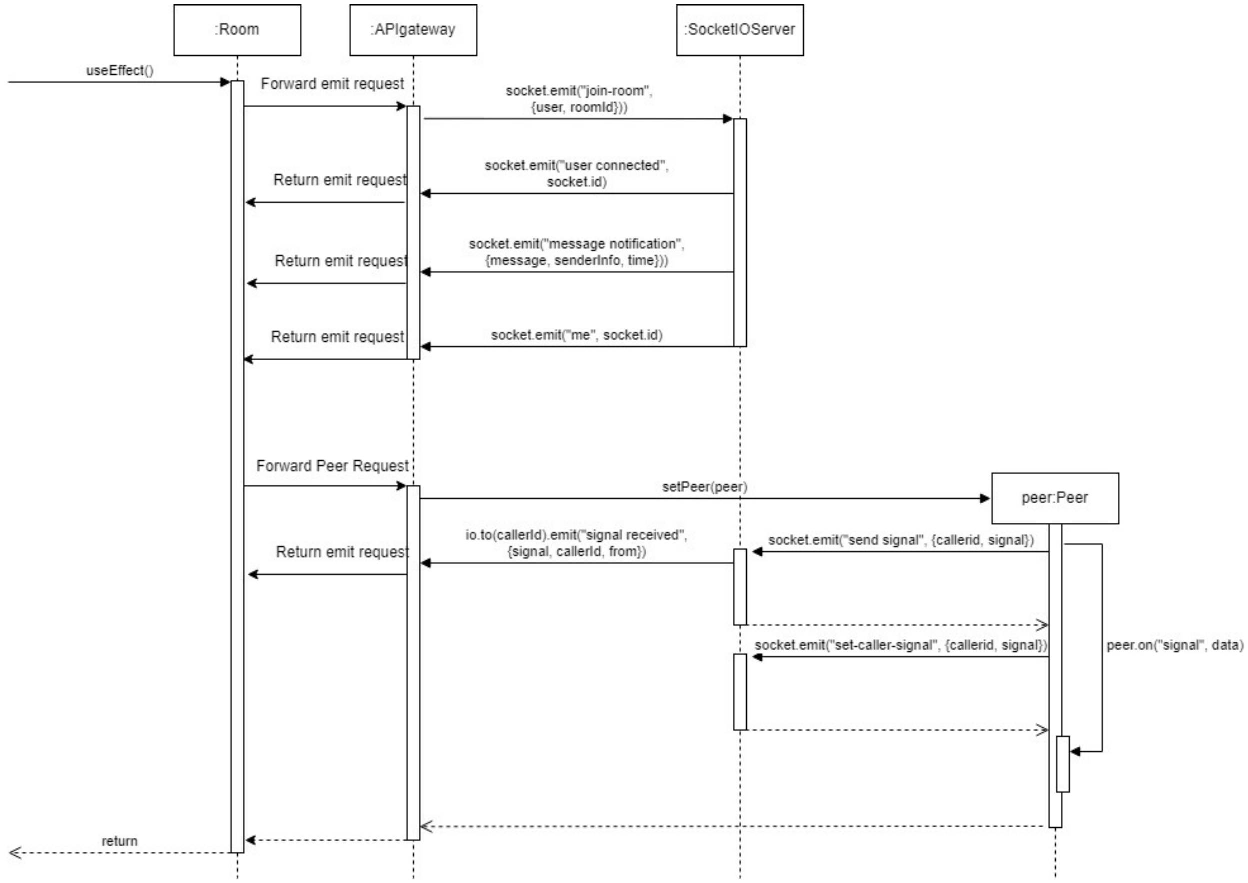
Sequence Diagram of timeout situation:



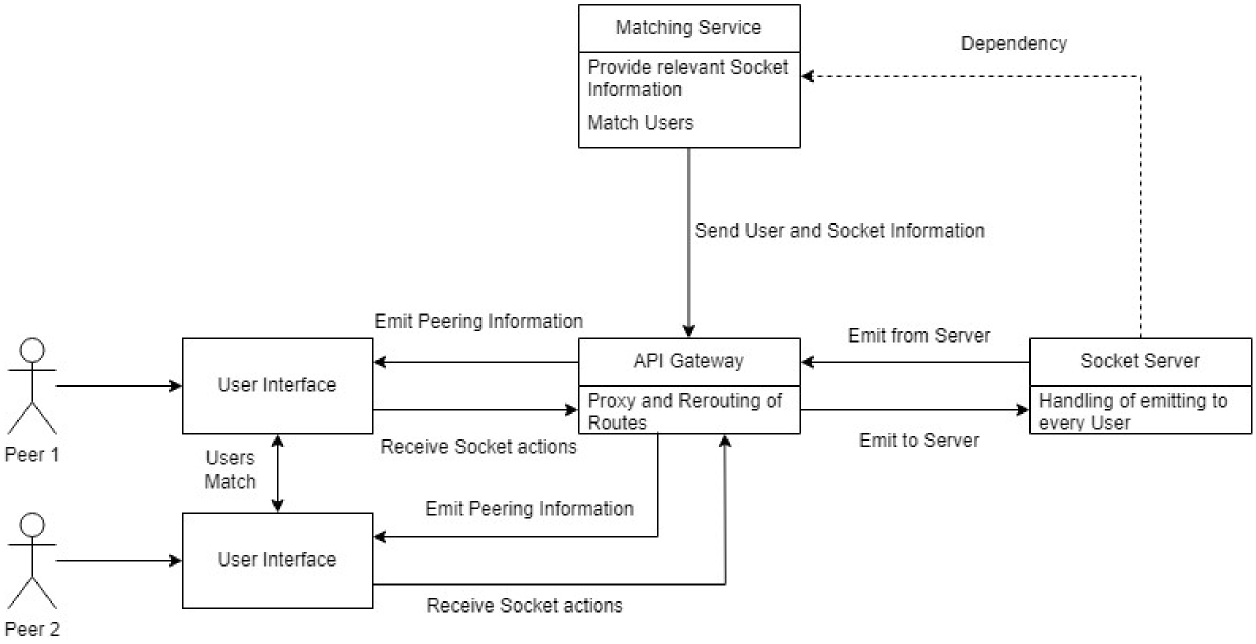
The above sequence diagram depicts the case of timeout situations, where users do not get matchmake with another user within the given time. In these situations, The frontend will trigger the timeout event that sends a POST request with the user’s information, username, email address, question difficulty level, and socketId, through the API gateway to the backend. This reaches the matching service and removes the user from the queue map, used to help match users in the matchmaking process.

This sequence diagram also depicts the exact scenario when users decide to manually cancel matchmaking while waiting for a match.

Collaboration Service

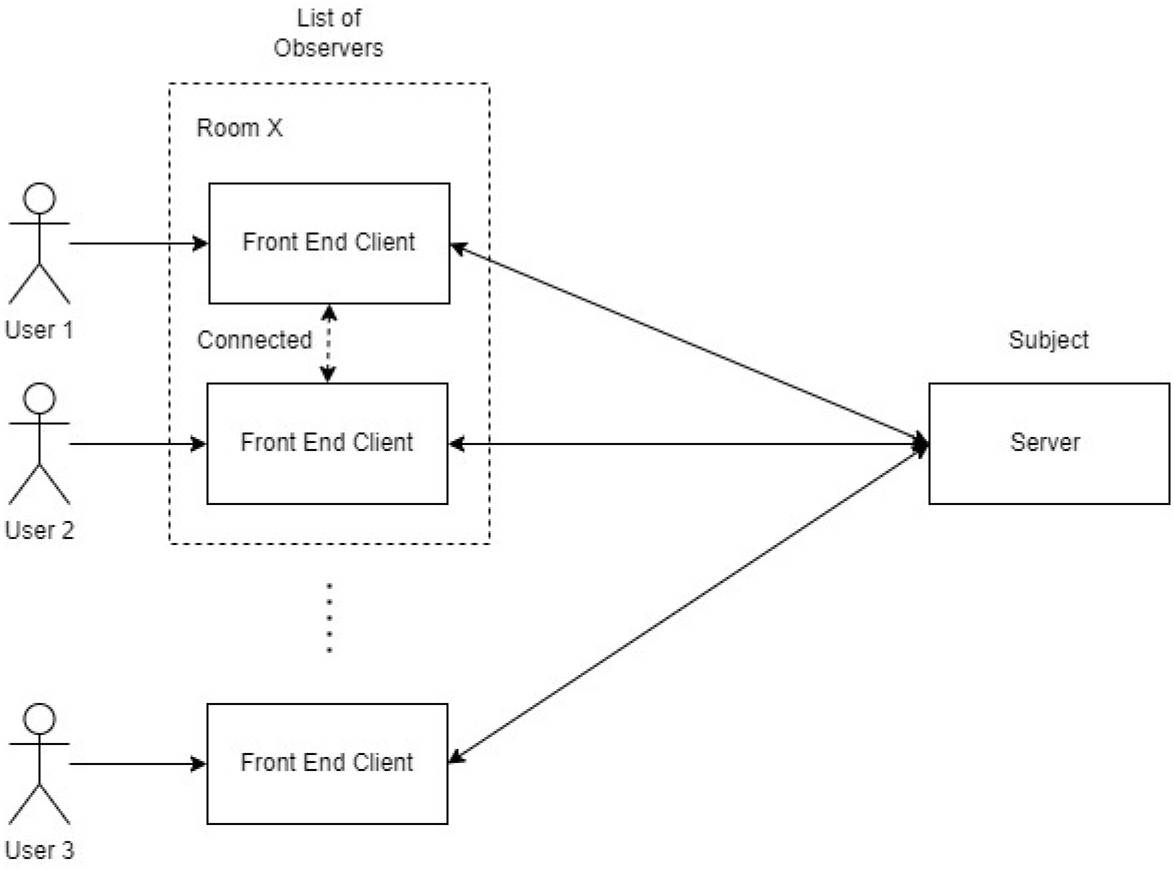

:Room
:APIgateway
:SocketIOServer
useEffect()
Forward emit request
socket.emit("join-room", {user, roomId}))
socket.emit("user connected", socket.id)
Return emit request
socket.emit("message notification", {message, senderInfo, time}))
Return emit request
socket.emit("me", socket.id)
Return emit request
Forward Peer Request
setPeer(peer)
peer:Peer
io.to(callerId).emit("signal received", {signal, callerId, from})
socket.emit("send signal", {callerid, signal})
Return emit request
socket.emit("set-caller-signal", {callerid, signal})
peer.on("signal", data)
return

The collaboration service uses a socket IO server to form connections between the matched users. Matched users will first emit through the socket using the "join room" keyword through the server with their user details and room ID allocated. The server side will then emit all the information that the user requires so the peer can receive any information for the socket connection. The user's front end will set a peer which will listen for any signal and set the peer accordingly. The diagram above only shows the flow of making the connection by sending the signal. All the emits will go through the API gateway where the API gateway will help to forward the requests back and forth from the front end to the backend.



Users are matched through the matching service and pass the user's information such as **username**, **email**, **difficulty level**, and **socketId**. The collaboration service is dependent on the matching service to provide this information for both frontends to relay information to one another and establish a connection using the socket ID. The users will be on two separate web page applications and communicate with each other by socket events through the Collaboration Service. The nginx controller will help to control the proxy and routing of the API requests through the API gateway making it more secure.

Design Patterns of Collaboration Service

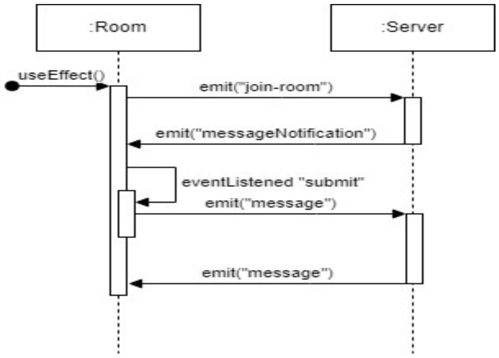


The Collaboration Service consists of different design patterns.

The first pattern is the **observer pattern**. The server is a subject that maintains a list of connected clients and helps to broadcast to clients in the same room or to every client based on the "message". The observers are the clients that listen for the client-specified events. The clients use a listener ("socket.on()") function to listen for any client-specified events. The events will have an event name such as "message-notification" with any data attached that is emitted from the subject which is the server. The server listens for any events from the client (users) and emits them to the respective room or everyone depending on the event specifications.

The second pattern is the **event-driven** paradigm. Every action is driven by an event that is emitted from client to server and server to client. The server emits any events that are required for both clients and listens for any events from each client or client connection. The client listens for any event response from the server and performs custom actions accordingly with the data received. Most actions of the client are mostly based on the event emitted from another client through the server. It acts like a relay system.

Chat Feature with Collaboration Service

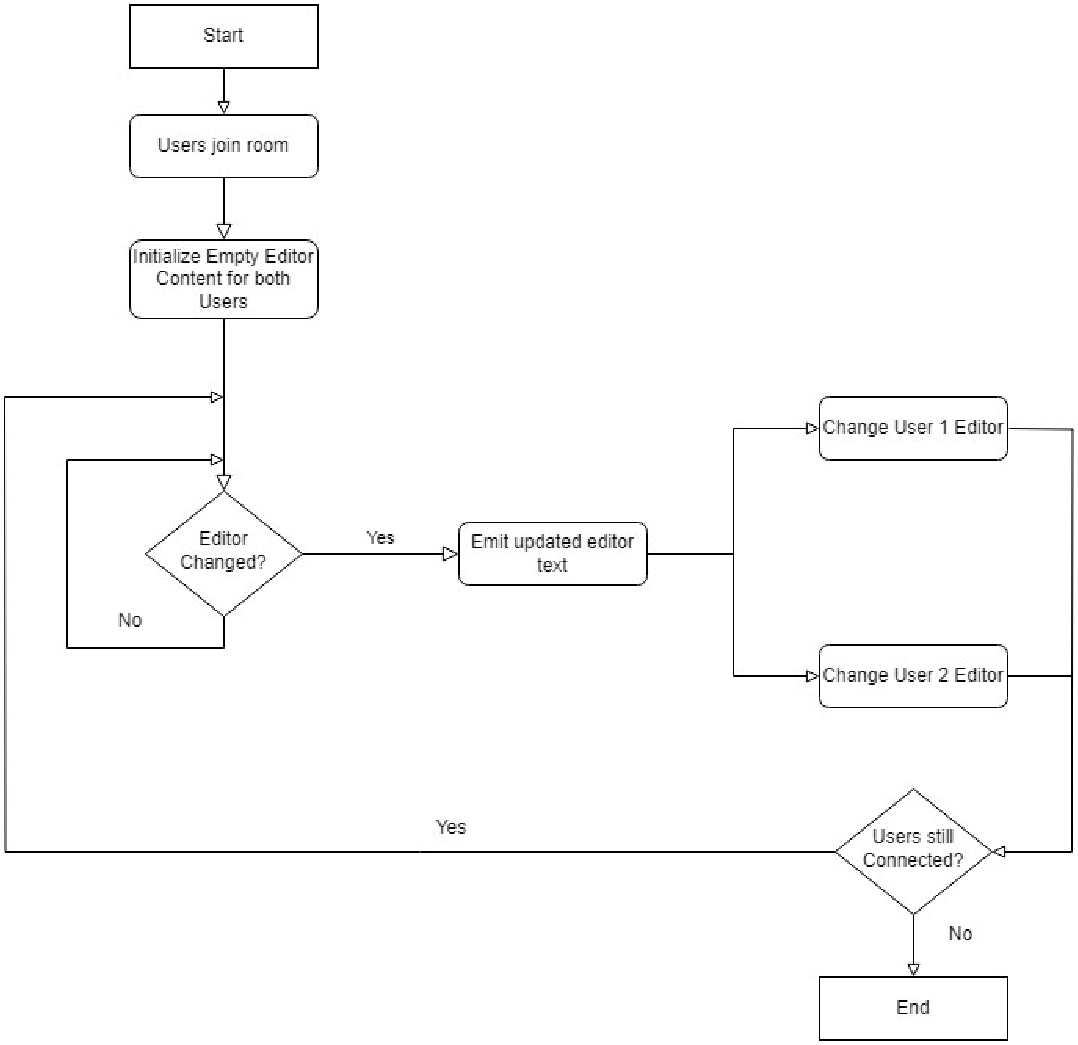


Note: This sequence diagram contains function labels with “code” word parameters only for easier reading.

When both users enter the room, the server will emit a **“messageNotification”** event name to show that both users are connected to the room. The ***chatForm*** will listen to the ***submit*** button clicked by the user through the chat. If the submit button is clicked, the message and the event name **“message”** will be emitted to the server. The server will emit the message to every user in the room and both users will be able to see the messages.

This design pattern is a form of a **pub sub** system where users (subscribers) can subscribe to certain channels and different features to emit any information for the chat room to change “state” for everyone in the same socket room. The user that sends the message is the publisher will publish any change of “state” to both users through the server which is the message broker and both users will receive it. They will each individually decide how to use it on their end. This helps to remove any coupling between any channels or features. For example, user A emits that he will like to send a message to user B and the server will publish to both user A and B that user A has sent a message and it will be displayed accordingly for both users.

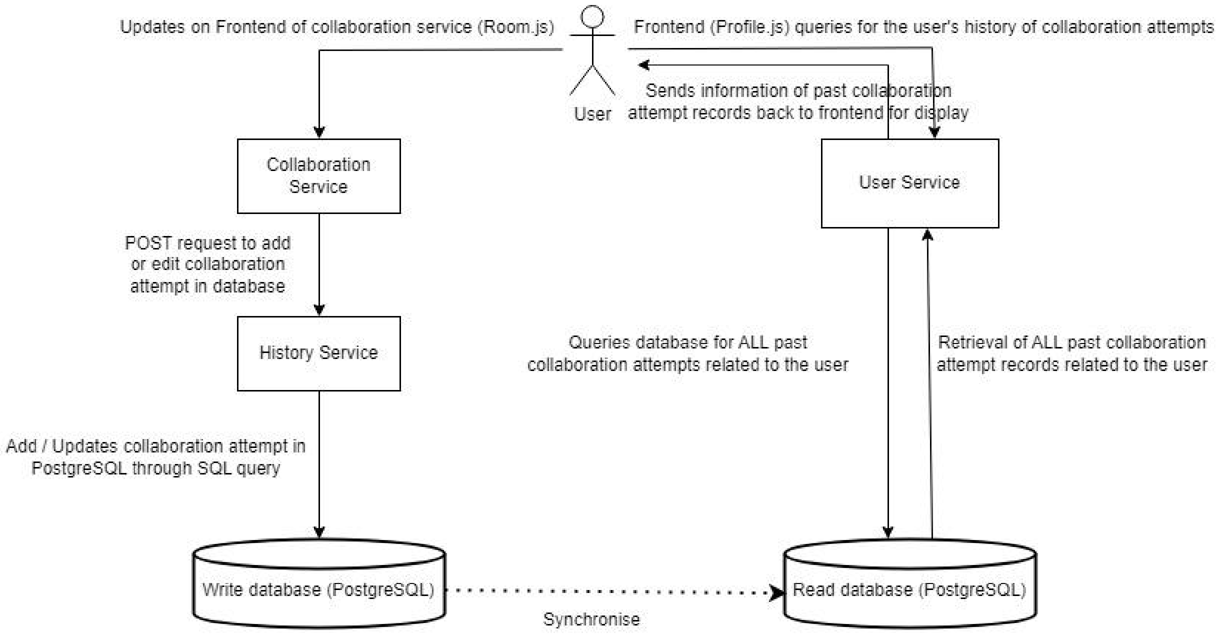
Code Editor Feature with Collaboration Service



Both users will be initialized with an empty editor after joining the socket room. There is a listener for any editor changes and will activate the handleEditorChange function. This function will emit the updated editor text and event name where the front end Room page of both Users will receive the event and change the editor to the updated version. If the users are still connected, they will be continuously listening for any code change from their editor. Otherwise, the editor will stop listening for any text change.

History Service

The History Service architecture follows the **Command-Query-Responsibility Segregation (CQRS)** pattern. The following diagram depicts how the CQRS pattern is used for the History Service.


Updates on Frontend of collaboration service (Room.js)
User
Frontend (Profile.js) queries for the user's history of collaboration attempts
Sends information of past collaboration attempt records back to frontend for display
Collaboration Service
User Service
POST request to add or edit collaboration attempt in database
History Service
Queries database for ALL past collaboration attempts related to the user
Retrieval of ALL past collaboration attempt records related to the user
Add / Updates collaboration attempt in PostgreSQL through SQL query
Write database (PostgreSQL)
Read database (PostgreSQL)
Synchronise

Upon any of the following updates on the frontend of the collaboration service, Room.js, whether there is an edit of code, change of programming language, or a refresh of question, it will trigger a **POST** request to the History Service's backend, that includes information of both users' email addresses, question information, roomId, code, programming language and current date and time.

On the backend, the History Service will then send an SQL query with this information to check if there's an existing entry in the PostgreSQL database with matching user emails and matching roomId. If there's an existing entry, the History Service will send an **UPDATE SQL** query to update the information in the existing entry. If no entry is found, it will send an **INSERT SQL** query to insert this new information as a new entry to the PostgreSQL database. PostgreSQL will then synchronize internally and ensure when new SQL queries come in for reading purposes, it will be detected.

As the user clicks on his/her Profile to perform any updates / view past collaboration attempts, the frontend will automatically trigger a **POST** request query to the backend of User Service to retrieve these history records by providing the user’s authorization token and user’s email as details. At the User Service, it then sends a SQL query to obtain all collaboration attempt records in the PostgreSQL database that has the user’s email recorded down as these are the attempts the current user has been in. These records will then be sent back to User Service as a single reply and be forwarded to the frontend for processing and display to the user. Users can then click on individual collaboration attempts that are displayed in a table. This will then link them to a collaboration room that displays the code they have written previously, along with information on the question they have previously attempted.

Other development artifacts

API test cases have been created for the User, Questions, and History services, allowing developers to thoroughly assess their respective APIs. Leveraging the Mocha testing framework, along with Chai and Chai-Http, each service is accompanied by its dedicated test file. These tests exclusively focus on the functionality of the service's API, utilizing stubs instead of actual databases to ensure controlled and predictable testing environments. Developers can seamlessly run these tests locally with a single command (`npm run test`).

Furthermore, for continuous integration, a GitHub Actions workflow has been created to automatically execute all test cases whenever a developer pushes changes to the master branch. This integration not only speeds up the testing process but also upholds code quality standards by ensuring that all API tests are rigorously validated before code changes are merged into the main codebase.

Suggested improvements and enhancements to the delivered application

We wished to develop the web application further but due to a shortage of project time, we cannot enhance some of the features further. The following are the enhancements that we hoped to improve on:

- Deploying the frontend to a cloud service.
- Implementation of more features such as:
  - Code execution
  - Displaying suggested answers to users
  - Showing users the time and space complexity of their code
- Improving the security aspect of our application
- Improving handling of any abrupt refresh or exit of one peer
- Providing a Contact Us platform / webpage for users to contact administrators for help and troubleshooting

Reflections and Learning Points

“From this project, many design patterns can be used for many scenarios. I have learned that not all design patterns fit certain situations perfectly. However, design patterns used correctly allow scalability and create independent components that are effective and useful. For example, the question service MVC architecture allows the new enhancement without affecting the original functionality. Working with the full software development life cycle was tough as we were managing to assign tasks and trying to choose the best database management and which software components to use for our design. Everything felt uneasy at first but over time through the course, we gained more insights and experience with the various design patterns and architectures to execute our project. The toughest issue with the project is using and setting up the docker for running the program so that the collaboration service and matching service can still be independent of each other but communicate through the API. However, the docker, allows us to separate our services by microservices which allows them to work independently and more securely. Through weekly scrums and sprints with the team, I am glad we have put together a great project and a lifelong experience taking on this project.” **- Jun Da**

“Throughout this project, I've gained a deep understanding of various design patterns and had the opportunity to experiment with several of them in our development process. This hands-on experience has broadened my perspective on architecting robust and maintainable software. Additionally, delving into the intricacies of Docker has been instrumental in streamlining our deployment process. Implementing Docker has not only facilitated local deployment, ensuring a consistent application environment for every team member, but it has also eradicated potential compatibility issues. Engaging in weekly scrums has fostered effective communication and collaboration within our team. Working collectively to build a project that mirrors frameworks used in the professional realm has been a valuable experience. This endeavor not only provides a glimpse into the real-world practices of software engineering but also equips me with skills and insights crucial for my future as a software engineer.” - **Shanice**

“Through this project, I’ve gained a deepened understanding of the various Software Engineering concepts and design patterns that were taught during lectures and tutorial sessions by applying them to our project. These include REST Architectural Style, MVC model and the various message patterns. The one that stood out most to me would be the message patterns where I applied both the Point-to-Point and single receiver message patterns for the matching service. This provided me the opportunity to see the usefulness and pick up some of the commonly used SWE concepts in the industry. Taking on a full stack green field project proved to be a challenging task even though we were in a team of 5, as we had limited experiences in the implementation of certain aspects of the project and we had to juggle between the project and demands from our other school modules. Nonetheless, it was still rather an interesting and useful project, essential to take up as a Software Engineering student.” **- Wen Jue**

“From this project, I have gained valuable knowledge about authentication and authorization role management using JWT tokens. The experience of creating comprehensive test cases within a framework and implementing a CI workflow through GitHub Actions has not only enhanced my technical proficiency but also instilled a sense of efficiency in the development process. I have also learned to use microservices architecture and leveraging Docker for design considerations has broadened my understanding of scalable and modular solutions. By collaborating within a team, I have learned to communicate my challenges to both mentors and teammates. Despite being relatively new to the technologies employed, I embraced the learning curve, attempted to implement features, and sought guidance when needed.” **-Xuan Yun**

“Throughout this project, I had the opportunity to delve into the various design patterns and concepts we studied in lectures and apply them to our development process. Along the way, I came to appreciate the microservices architectural style for its loose coupling and independently deployable capabilities, allowing us to work on different parts simultaneously without affecting one another. This also made it easier to implement Docker, as each service could build its own image and run independently. From this project, I also gained hands-on experience in implementing RESTful APIs and the MVC model for the question service, while also exploring new frameworks like socket-io, which enables bidirectional and event-based communication for the collaboration service. Despite the occasional challenges we faced along the way, my teammates and I supported each other and shared valuable insights for improvement. The weekly scrums held fostered effective teamwork, and I'm proud of the project we've collectively created. This experience has equipped me with valuable skills and insights that will undoubtedly benefit my future as a software engineer.” **-Jovita Anderson**