CS3219 Project Report

Group: 48

Student	Student ID
Lai Chok Hoe	A0217507H
Tan Yu Tao	A0218279U
Cheng Jiyuqing	A0204848Y
Seah Zhi Xuan	A0227546Y

CONTENT PAGE

INTRODUCTION

Background

Increasingly, students face challenging technical interviews when applying for jobs which many have difficulty dealing with. Issues range from a lack of communication skills to articulate their thought process out loud to an outright inability to understand and solve the given problem.

Moreover, grinding practice questions can be tedious and monotonous. To solve these issues, we have created a coding interview preparation platform and peer matching system called **PeerPrep**, where students can match with peers to practice whiteboard-style interview questions together.

Purpose

We aim to create a web application that helps students better prepare themselves for technical interviews by using a peer learning system such as the one proposed above so students can learn from each other and break the monotony of revising alone.

Potential Use Case

A student who is keen to prepare for his technical interviews visits the site. He creates an account and then logs in. After logging in, the student selects the question difficulty level he wants to attempt today (easy, medium, or hard). The student then waits until he is matched with another online student who has selected the same difficulty level as him. If he is not successfully matched after 30 seconds, he can either choose to wait for another 30 seconds or change the difficulty he wants to attempt. If he is successfully matched, the student is provided with the question, a chat box and a collaborative text field in which he is expected to type his solution. This collaborative text field would be updated in near-real time, allowing both the student and his matched peer to collaborate on the provided question. After the students finish working on the question and are ready to end the session, they can individually click on the “Leave” button which brings

the student back to the question difficulty selection page. From this page, the student can log out or make another attempt.

Project Contribution

Student	Contributions
Lai Chok Hoe	1. Creating the difficulty selection page (MatchingPage) where students are presented with the choice of “easy”, “medium” and “hard” 2. Implementing a Pub-Sub mechanism using Socket.IO on the client/frontend side 3. Creating the room page (MatchingRoom) where students are presented with the question, a chatbox and a collaborative text field 4. Implemented the communication service responsible for functions of the chat box in the room page 5. Implemented the API request calls in from the matching service to the question service to get the question for the matched students 6. Created the Dockerfiles for each service and the Docker-compose.yml file for easy deployment of the app to the (local) staging environment using Docker.
Tan Yu Tao	1. Created backend API and routing for User service, Create, Update, Delete. 2. Bugfixed Update and Delete API, after regression due to implementation of password encryption. 3. Created backend API and routing for Question service, Create, Read, Update, Delete. API to get a random question given the difficulty parameter. 4. Created the frontend component frameworks for Question service. Allowing users to interact with the backend API.

	5. Implemented github actions workflow for continuous deployment 6. Deployed the various micro services to heroku and integrated them to the final production application.
Cheng Jiyuqing	1. Implemented API and created a new route for logging users in. 2. Implemented JWT and cookies for logins 3. Hashing and salting of password using Bcrypt package 4. Authorization by verifying JWT 5. Styled and improved frontend for login, sign up page. 6. Implemented additional frontend functions (change password, delete account) in a dynamic dropdown component with css-transition 7. Created and styled landing page 8. Improved and styled matching page and associated functions 9. Improved and styled matching room page, (frontend for communication service and collaboration service) 10. Improved and styled question page (question service frontend), changed multiple functions involving navigation to new page to in-page manipulation 11. Implemented forget password and send email to retrieve password function in the frontend (login page). But it is later taken out as there is not enough time to implement the backend functions.
Seah Zhi Xuan	1. Created Matching service backend using socket.io and sequelize to match two users with the same difficulty, sending a success event or a failure event if

	there is no match within 30s. 2. Created frontend editor component for collab service using codemirror and yjs that allows for code syntax highlighting, switching of programming language syntax highlighting, independent undo/redo for each user of the editor and showing presence and cursor location of other users. 3. Created Collab service backend server using yjs websockets that distributes the relevant information e.g. document updates and awareness information. 4. Assist with heroku deployment bug fixing for collab service 5. Assist with docker-compose regression bug fixing after deployment to heroku

Project FRs and NFRs

User Service

S/N	Functional Requirement	Priority
FR1.1	The system should allow users to create an account with username and password.	High
FR1.2	The system should ensure that every account created has a unique username.	High
FR1.3	The system should allow users to log into their accounts by entering their username and password.	High
FR1.4	The system should allow users to log out of their account.	High
FR1.5	The system should allow users to delete their account.	High
FR1.6	The system should allow users to change their password.	Medium
	Non-functional Requirement	
NFR1.1	Users' passwords should be hashed and salted before storing in the database. **Justification**: This is for security purposes. Only hashed and salted passwords should be stored in the database to protect user's credentials in case of data breach.	Medium

Matching Service

S/N	Functional Requirement	Priority
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
FR2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High
FR2.3	If there is a valid match, the system should match the users within 30s.	High
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
FR2.5	The system should provide a means for the user to leave a room once matched.	Medium
FR2.6	The system should provide the room id and question to matched users.	Medium
	Non-functional Requirement	
NFR2.1	The system should handle unexpected disconnection e.g. close tab, refresh page, logout. **Justification:** Enforces the robustness of the service such that even in unexpected disconnections, users will be removed from the matching queue such that other users do not get matched with a user that is not present.	High

Communication Service

S/N	Functional Requirement	Priority
FR3.1	The system should allow users to send messages to the other user he/she is matched with.	High
FR3.2	The system should allow users to receive messages from the other user he/she is matched with.	High
FR3.3	The system should only allow matched users to message each other for 1 to 1 communication.	High
FR3.4	The system should ensure that messages are sent to the correct rooms.	High
FR3.5	The system should guarantee that messages are always successfully sent to both users.	Medium
FR3.6	The system should ensure that the message history in either user's chat box are synced.	Medium
	Non-functional Requirement	
NFR3.1	The system should inform users when they are connected to the communication service. **Justification:** Improve the usability of the communication service. Provides a layer of verifiability to the users by providing information on the status of the communication service.	Low
NFR3.2	The system should inform users when the other user has left the room intentionally or unexpectedly.	Low

	Justification: Improve the usability of the communication service and ensure users are informed about the system's and the other user's status.	
NFR3.3	The system should mark messages with tags that uniquely identify them for easy reference. **Justification:** Improve the usability of the communication service by providing a point of reference towards specific messages during discussions.	Low

Question Service

S/N	Functional Requirement	Priority
FR4.1	The system should provide a means for the admin to add questions.	High
FR4.2	The system should allow storage and retrieval of questions based on difficulty.	High
FR4.3	The system should provide a means for the admin to update questions in the question bank.	Medium
FR 4.4	The system should provide a means for the admin to delete questions in the question bank.	High
FR 4.5	The system should randomize the question it retrieves.	Medium
	Non-functional Requirement	
NFR 4.1	Each difficulty level should have at least 5 questions to ensure randomness. **Justification:** Users will be less likely to attempt repeated questions. This ensures that users can have a more effective learning experience.	Low
NFR 4.2	The difficulty level of questions should be distinct (at least 3 difficulty levels). **Justification:** To ensure that users can select questions that are a match	Medium

	to their programming abilities, or to their desired difficulty	

Collaboration Service

S/N	Functional Requirement	Priority
FR5.1	The system should provide a collaborative editor that syncs between connected users.	High
FR5.2	The system should only allow collaboration between specified users.	High
FR5.3	The system should clean up after users disconnect.	High
FR5.4	The system should provide presence and cursor location to other users.	Medium
FR5.5	The system should provide independent undo and redo operations for each user.	Low
FR5.6	The system should provide code syntax highlighting for multiple programming languages.	Low
	Non-functional Requirement	
NFR5.1	Users should be able to see changes in the editor by the other user in real-time (within 1s). **Justification**: Specifies the performance of the system to ensure a smooth collaborative experience for users.	Medium

Fancy Frontend

S/N	Non-functional Requirement	Priority
NFR 5.1	The frontend should support all functions of the application	High
NFR 5.2	The system should be neat, intuitive for users and easy to navigate.	High
NFR 5.3	The system should display information dynamically.	Medium
NFR 5.4	The system should be aesthetically pleasing.	Medium

Other Non-Functional Requirements

S/N	Non-functional Requirement	Priority
NFR 6.1	The system should be usable and consistent on different browsers (Chrome, Safari, Edge and OperaGX) **Justification**: To ensure users who use different browsers can still collaborate together. Chrome, Safari and Edge and the most commonly used browsers.	High
NFR 6.2	The system should sign up, login, see matching timer within 2s of clicking respective buttons. **Justification**:	Medium

	This improves the performance of the application which provides a more fluent user experience.	

Developer documentation

For the duration of the project, an Agile approach was adopted. This allowed for greater flexibility and worked well with our microservice architecture.

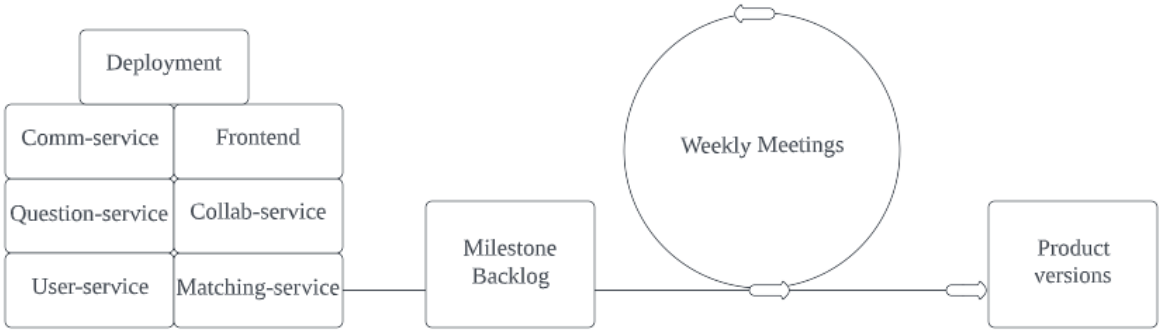


Fig. Weekly Scrum

For our project, we then adopted a modified version of Scrum. Instead of daily meetings, we held weekly meetings, to discuss project backlog and work allocation. This was decided on, due to our various school schedules and commitments, which meant that daily Scrum meetings were impossible. Weekly meetings however still kept the project iterative, providing us a soft deadline to complete our assigned project backlog.

Project Timeline

	Zhi Xuan	Chok Hoe	Yu Tao	Yuqing
Week 3 - 4 22 Aug ~ 2 Sep	Requirements churning Role allocation Propose timeline			
Week 5 - 6 5 Sep ~ 16 Sep	Set up backend for Matching service	Set up frontend for Matching Service	Set up backend for User service API for Creating user	Implement Frontend page for login Hashing and salting of password
Week 7 26 Sep ~ 1 Oct	Add logic for match queueing	Bug fixes for Matching Service	Complete API Update Delete User	Add JWT and Cookies
Week 8 3 Oct ~ 7 Oct	Implement Sequelize ORM and match model Bug fixing	Set up frontend for the Matching Room	Set up backend for question serviceAPI for creating get, delete, update Question	Authorisation Frontend mock up with Figma Add landing page
Week 9 10 Oct ~ 14 Oct	Set up frontend for Collab service	Set up backend for communication service	Update User service Update and Delete after regression due to Bcrypt	Style sign up, login page

<table>
<tr><td></td><td></td><td></td><td>implementation
Bug fix for Question service getting random question</td><td></td></tr>
<tr><td>Week 10
17 Oct ~ 21 Oct</td><td>Set up frontend for Collab service</td><td>Complete hookup from Matching Room to Communicatio n and Question Service</td><td>Complete frontend component for Question service</td><td>Style matching page, implement dynamic dropdown</td></tr>
<tr><td rowspan="2">Week 11
24 Oct ~ 28 Oct</td><td>Change Collab service connection provider</td><td rowspan="2">Set up deployment to local staging environment</td><td rowspan="2">Deployment of each microservice/ troubleshooting deployment</td><td>Style matching room</td></tr>
<tr><td rowspan="2">Set up Collab service backend</td><td rowspan="2">Implement delete account, change password in matching page</td></tr>
<tr><td rowspan="2">Week 12
31 Oct ~ 4 Nov</td><td rowspan="2">Assist with integration and testing of local deployment and deployed</td><td rowspan="2">Final deployment for Collab service Integration testing of</td><td rowspan="2">Style question page</td></tr>
<tr><td>Assist with heroku deployment</td></tr>
</table>

	and docker	application	deployed application	
Week 13 7 Nov ~ 9 Nov	Bug fixing Report writing			

Role allocation

Matching Service	Chok Hoe (Frontend) Zhi Xuan (Backend)
User Service	Yuqing (Frontend) Yu Tao (Backend)
Question Service	Yuqing (Frontend) Yu Tao (Backend)
Communication Service	Chok Hoe (Frontend & Backend)
Collaboration Service	Zhi Xuan (Frontend & Backend)
UI Design	Yuqing
Deployment	Yu Tao & Chok Hoe

Foreseen Challenges

Time management

Considering each of our varying school schedules, as well as the unknown of working with unfamiliar technology; Time taken to complete each of our allocated components will vary greatly, with delays in completing assigned components to be expected. As such, enhancements planned, might end up being not completed.

Learning and adapting to new techstack

Building on the time management aspect, having to learn new tech stacks unaided will surely add to time spent. As well, learning how to implement functionalities and fix bugs will add to the amount of time taken to complete components. But once familiar with the technology, progress in implementation of similar services or fixing of bugs will take significantly less time.

Challenges to deploy web application

Since this is our first time deploying an application to the web, compounded by the fact that we have decided on a microservice architecture. How the deployment is carried out, such as the order in which services are deployed and how the various services are integrated will be a challenge that will take a considerable amount of time to complete.

Adopting a microservice design

When starting the project, we knew we wanted to make use of microservices in our project. However, for all of us, this was the first time tackling the challenge of implementing microservices. While it did make the delegation of workload easier, there was definitely a greater sense of responsibility held by each member for each microservice that they have worked on.

Architectural Design

Overall Architecture

When selecting between micro service or monolithic architecture, the table below was drafted to inform our decision.

Standpoint	Micro service	Monolith
Pros	• Reduced coupling between components • Breaks project down to manageable sections • Easier for future extension and scaling • Easier to delegate responsibilities • Workflow is not dependent on others progress • Helps with the ease of testing independent functionalities	• Most familiar, architecture • Easy to install dependencies and deploy • Clearer picture of the overall application
Cons	• Unfamiliar with project structure • Deployment requires deployment of individual components • More attention needed to categorize functions into unique microservices	• Components are highly coupled • Scaling for future enhancements is difficult • Merging in changes between teammates can be tedious • Testing of the individual

		functionalities become more complex

Factoring in the given time-frame of our project, the distinct labels already provided for the functionalities of the application, and the various commitments that each member had outside of the project, we adopted the **microservices architecture**. This would allow us to complete various components in parallel and test their functionalities independently. The microservice architecture also helped to simplify task allocation and allowed us to tackle our project backlog in a more dynamic manner. Members who had completed their task could quickly refer to and work on tasks detailed in our project backlog. A microservices architecture also improved the scalability of our final application, allowing for the easy planning and implementation of future features. The diagram below details the general architectural design, along with an architecture diagram.

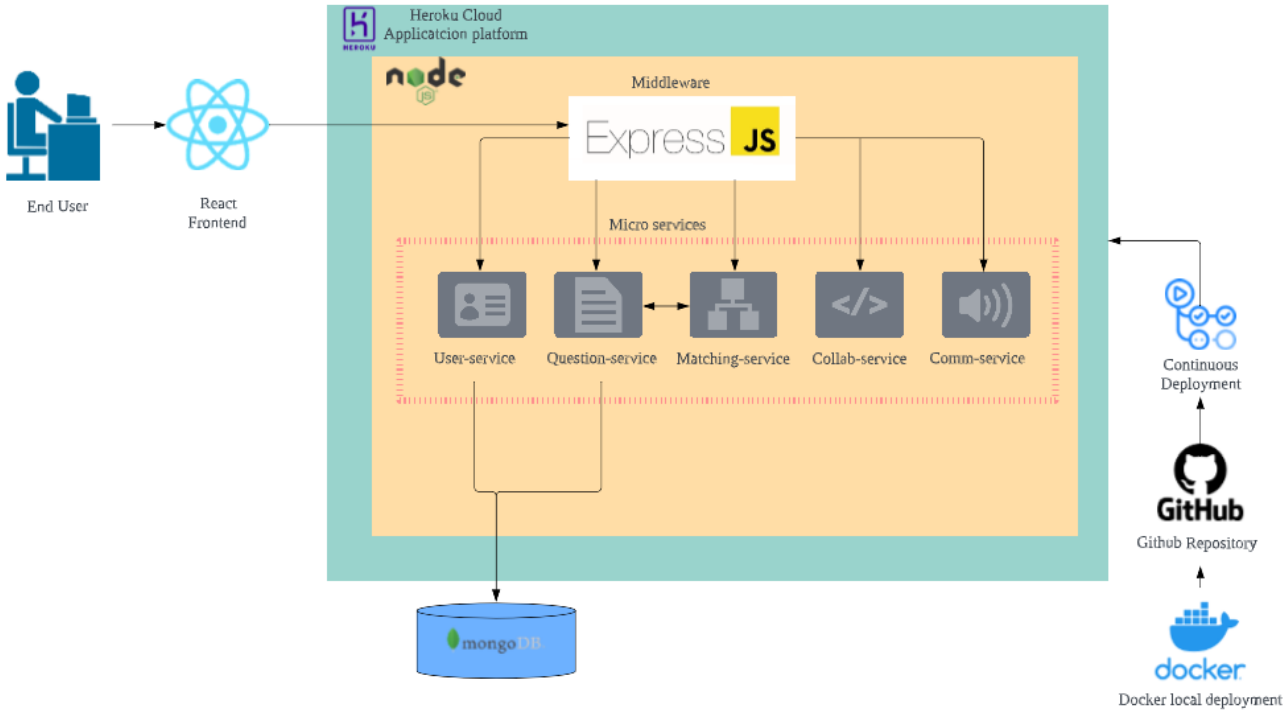


Fig. Architectural diagram

Services

Based on our identified Functional Requirements and Non-Functional Requirements, we implemented the following services, each handling different responsibilities.

- Frontend - main component that users interact with, allows users access to the various APIs
- User service - contains the APIs to allow user account creation, update, delete and login. User account is used to login into the web application.
- Question service - contains the APIs to get, create, update and delete questions. Questions will be displayed in the frontend after users have matched into a matching room
- Collab service - contains the server that connects Users and allows them to collaborate in the same space. When one user types code, it will appear on the other user's frontend.
- Matching service - has the responsibility of matching two users who have selected the same difficulty. Without this service, users will not be able to collaborate with each other.
- Communication service - acts as a broker to allow users to send messages between users. This provides the basic means for communication when using our application without a reliance on third party solutions

Design and Messaging Patterns

Model-View-Controller (MVC) Pattern

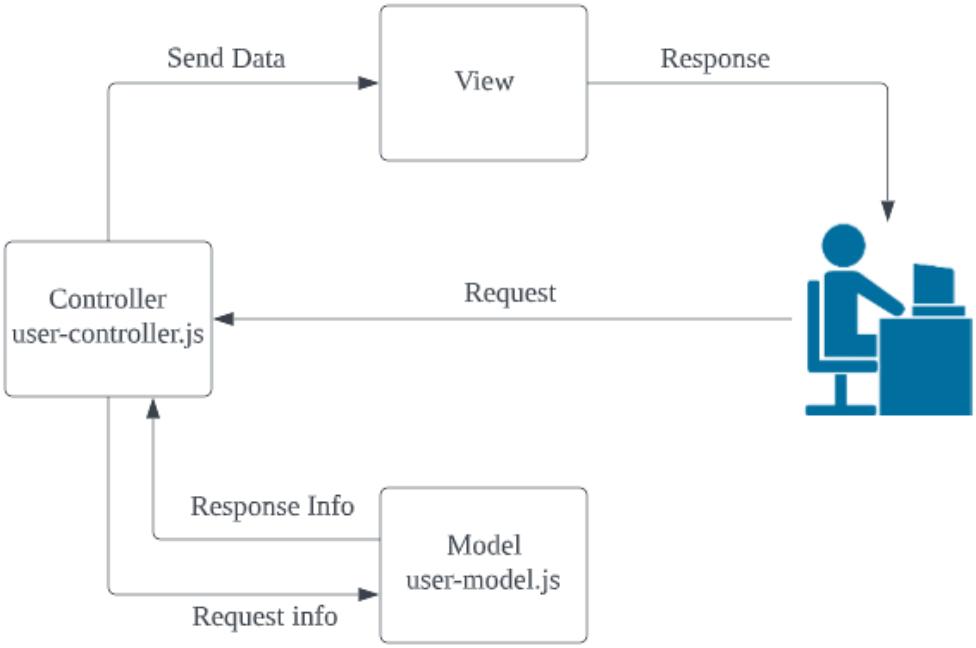


Fig. MVC pattern used in user service

The MVC pattern is utilized in our user and question service model. This pattern provides us with separation of concern between the frontend, backend and database.

- Users input and output is handled by the view and controller component. Both of which are linked with the frontend.
- Data related concerns are handled solely by the Model component which only interacts through the controller component.
- This allows easier management of data between the backend and frontend.

Pub-Sub pattern

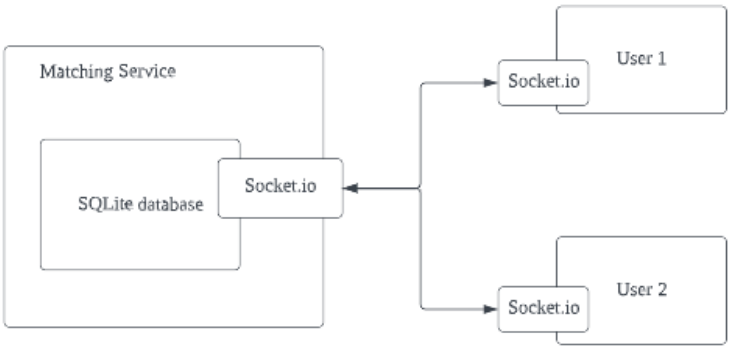


Fig. Pub-Sub messaging pattern in matching service

In the matching service, the Pub-Sub messaging pattern is used to facilitate asynchronous message passing with multiple receivers. It is used to notify relevant users when a match has been successfully made.

- Here users begin matching by sending a ‘match’ request to the matching service.
- The match is then queued in the SQLite database for 30s.
- Once the Matching service detects that there is a match between users, a successful match event is published to the relevant users. Otherwise, after 30 seconds, a failed match event is published to the user that failed to match.
- Matched users are then redirected to the collaboration space by the frontend.

In the communication service, the Pub-Sub messaging pattern is used to facilitate a asynchronous message passing between a pair of users in the same room. By using the ‘room’ channels, we were able to isolate the communication between users to protect the privacy of their discussions.

- Users in a ‘room’ first provide an input message before sending a ‘sendMessage’ request to the communication service.
- In response, the communication service passes the message back to the ‘room’ by sending a ‘receiveMessage’ request to all users in the room.

- Upon receiving the request, all users in the room will make an update to their chat box with the message, the name of the user who sent it and a message identification number.

Data Transfer Object (DTO) Pattern

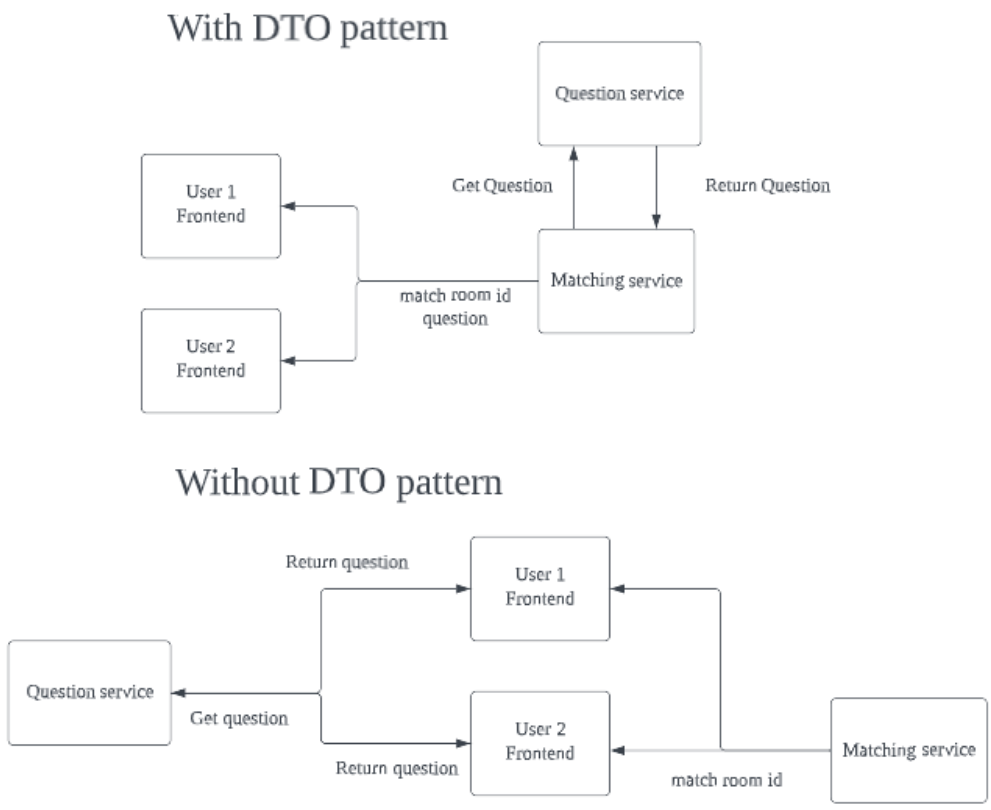


Fig. With and Without DTO pattern between Matching, Questions and Frontend service

The Data Transfer Object pattern is also implemented in our Matching service. DTO is used here to reduce the number of requests to the question service database.

- The Matching service publishes a match success event to relevant users, when a user is successfully matched with another user.
- The matched Users are assigned the same match room Id which will be used to ensure the users are in the same room (or subscribed to the same topic in pub-sub terminology) for communication service and collaboration service.

- To reduce the number of API calls to the question service and to ensure that both users get the same question from question service, we batch up the question with the match room id in matching service as a JSON object to send to both users.

Database per service pattern

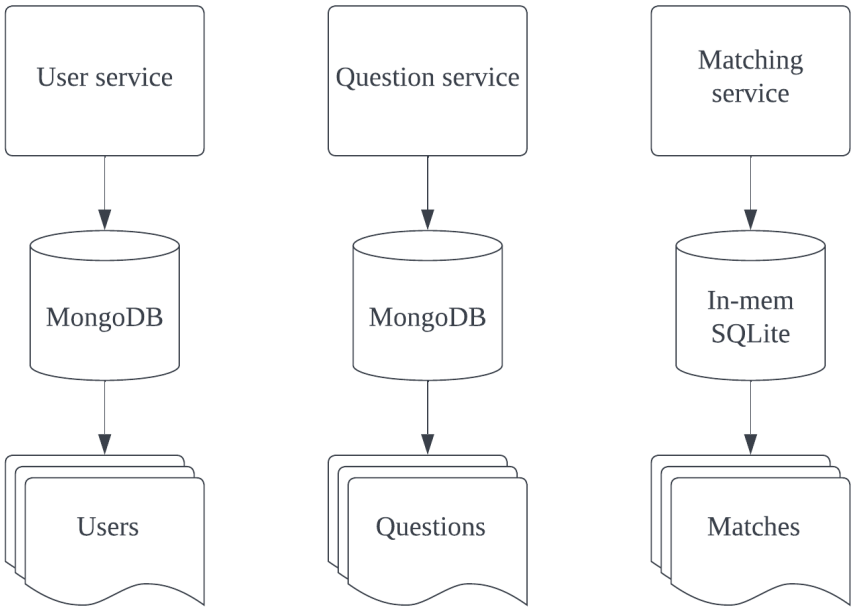


Fig. Database per service pattern in microservices architecture

As part of the microservices architecture, each service has its own database schema. This provides loose coupling between service databases and allows each service to have a database that is best suited to its needs. For example, question service stores its questions as documents in MongoDB, while matching service stores matches in memory SQLite as it only stores a few matches for faster read/write operations (detailed explanation in the later sections).

This also makes the maintenance of data used by each service much easier and can also improve the reliability of the database. In the event that any service should go

down and a rollback is needed to prevent corruption of data, there would be no impact on the data storage and creation done by the unaffected services.

Tech Stack

Component	Technology	Explanation
Frontend	React JS	React is the most popular and commonly used JavaScript library for frontend building. Hence we want to learn and use React JS for our application.
	Figma	Figma is used to design user interfaces. It allows us to preview each page, as well as some CSS configuration.
Backend	Express.js/Node.js	Node.js is a widely used runtime environment for the implementation of application backends. Express.js is a commonly used package that is part of Node.js specifically used for the development of web-application backends.
Database	MongoDB	MongoDB Is a no-SQL database. Allows for greater freedom in data stored in the database. Documents do not have to follow the rigid fields defined in a SQL database.

Database	SQLite	SQLite is implemented as an in-memory instance for this application. Matching service will only have a few matches in the database at any one point in time thus an in-memory instance is sufficient space-wise and faster for read/write operations.
Deployment	Heroku Free Tier	Heroku is a simple web application hosting site, which supports various forms of continuous deployment. It is fairly easy to use and deploy a web application to.
	Docker	Docker simplifies the process for installing and running the application for the first time in a new system. Using Docker, a new developer would just need to run the appropriate docker command to install the dependencies and run the application.
Pub-Sub Messaging	Socket.io	Socket.io provides event based communication between client and server which makes it ideal for handling asynchronous match requests and sending match success/failure events.

CI/CD	GitHub Actions	GitHub Actions is the most user-friendly and easy to use CI/CD platform that is already compatible with our version control tool git.
Orchestration Service	Docker-Compose	Docker-Compose is a simple orchestration tool that makes a good alternative to implementing an orchestration service. A new developer need only run a single docker-compose command, which then makes use of the DockerFile in each service to spin up the entire application.
Project Management Tools	GitHub Issues	Allows for tracking of any issues and creation of related pull requests. This allows for greater transparency on what project backlogs still require attention.
CRDT for collaborative editing	Yjs	Yjs provides a Conflict-free replicated data type (CRDT) for collaborative editing where changes are automatically propagated to other peers and are merged without conflicts. This allows us to avoid creating a collaborative editing algorithm from scratch.

Component Design

User Service

<table>
<tr><td colspan="2">Name: User Service</td><td>Description:
User services provides an API for finding, creating, updating and deleting users.</td></tr>
<tr><td colspan="3">Capabilities:
General user management</td></tr>
<tr><td colspan="2">Service API</td><td>Collaborators</td></tr>
<tr><td>Commands
<code>createUser()</code>
<code>logUserIn()</code>
<code>patchUser()</code>
<code>deleteUser()</code></td><td>Queries
<code>getUser()</code></td><td>Database</td></tr>
</table>

The user service handles the users sensitive information, such as their password and email. Hence the inclusion of an encryption service is paramount for user data security. Bcrypt and JSON web token were our chosen middleware to encrypt, authenticate and parse the token for users to login to their account. The user service has three basic API for the user to interact with, Create, Update and Delete. Along with these API, there are other APIs to login users which are used by the frontend login page to authenticate users. The database in use for this service is mongoDB

- Create - the Create API takes in the users username, email and password. The password is encrypted using Bcrypt before it is stored in the database along with the username and email.
- Update - the Update API, allows users to update their password. It accepts the users, username and password and newpassword. It uses the user's username and password to verify their account, before updating the old password with the new one. New password is encrypted and updated accordingly.

- Delete - the delete API, takes in the users username and password for verification. It then deletes the user's account from the database.
- Login - the Login API, takes in the users username and password. It validates their account, before returning a JSON web token to be used by the frontend for authentication purposes.

Question Service

<table>
<tr><td colspan="2">Name: Question Service</td><td>Description:
Question service provides API for finding, creating, updating and deleting questions.</td></tr>
<tr><td colspan="3">Capabilities:
Management of questions in database</td></tr>
<tr><td colspan="2">Service API</td><td>Collaborators</td></tr>
<tr><td>Commands
<code>createQuestion()</code>
<code>deleteQuestion()</code>
<code>updateQuestion()</code></td><td>Queries
<code>getAllQuestion()</code>
<code>getQuestion()</code>
<code>viewQuestion()</code></td><td>Database</td></tr>
</table>

The question service is in charge of the storage of questions, it also handles randomly producing questions for the matching service, which is then served to the matched users. The Question service has 4 basic API in total, Create, Get, Update and Delete. The database in use for this service is mongoDB

- Create - The Create API allows the creation of questions to be stored in the database. The question model has only 2 fields, difficulty and the question body.
- Get - The Get API, is actually divided into 3 different get API, “Get all”, “Get by ID” and a “Get random”. “Get all”, simply gets all the questions in the database and is used in the frontend to display all questions for management. “Get by ID”, gets a specific question and is used for modification of the question. “Get random”, gets a random question from the database, based on the difficulty provided. This is used by the matching service, to serve a random question to the matched user.

- Update - Update allows for the updating of a question in the database. The difficulty and question body can both be updated as needed.
- Delete - The delete API, deletes the selected question from the database.

Matching Service

Name: Matching Service	**Description**: The Matching Service is responsible for matching users based on the difficulty level of questions selected.
Capabilities: General user management	
Service API	**Collaborators**

Event Published
<u>Client Side</u>
`'match'`
<u>Server Side</u>
`'matchSuccess'`
`'matchFail'`
	Question Service
	`getQuestion()`
	
	Database

The Matching service is responsible for matching users and providing them with the same room id, as well as questions from question service.

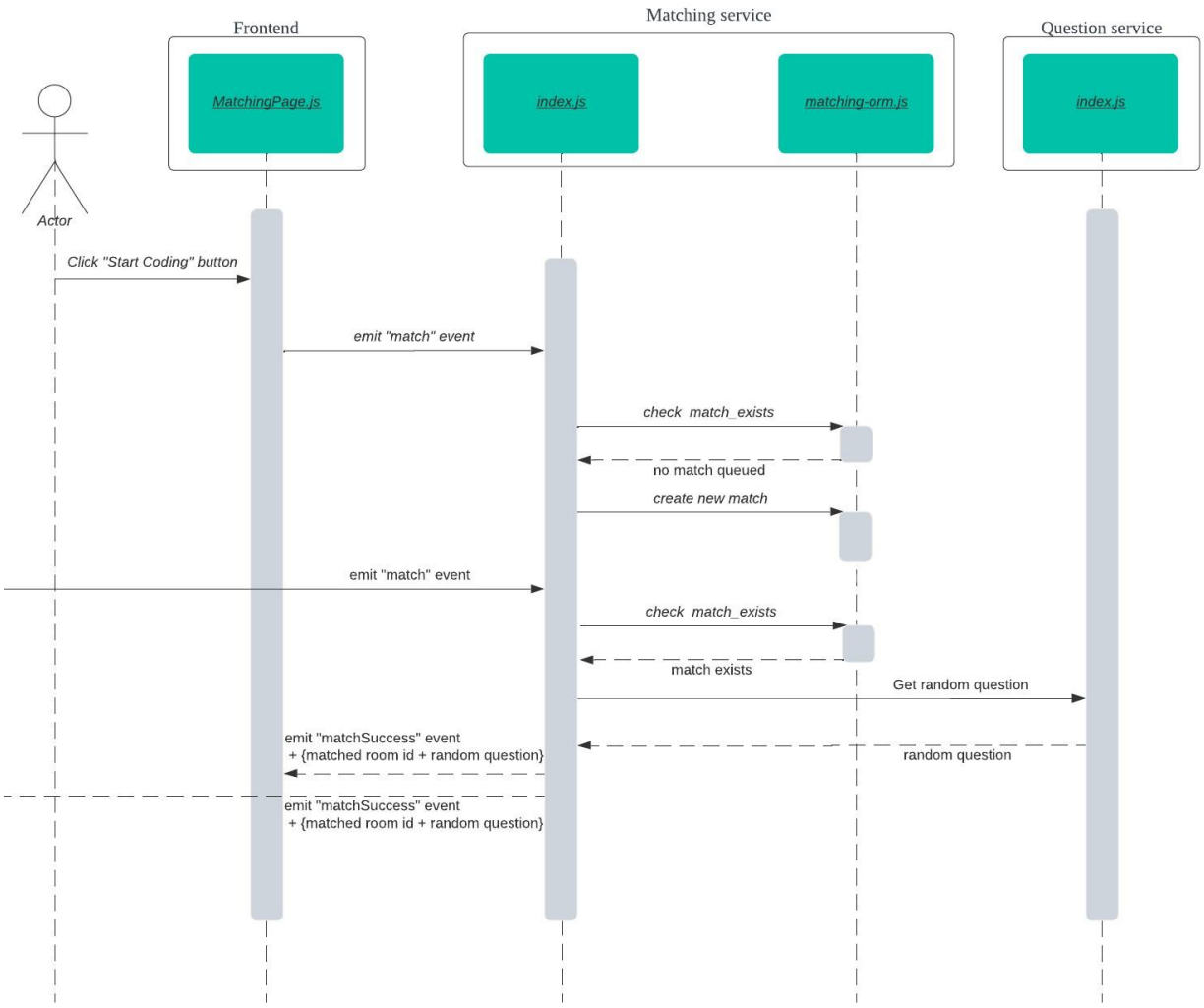


Fig. Sequence diagram for matching

When a user clicks on “Start Coding”, a match event is emitted to matching service. Matching service will check if there is currently a match with the same difficulty in its database.

If there is no match currently in the database, it will create a new match, store it in the database and create a 30s timer. Once the 30s timer is up, the Matching service will either remove the match from the database or do nothing if the match has already been removed.

If there is a match in the database, Matching service will take out the match along with its attributes such as socket id to emit the matched room id to both users via socket.io. It will also query question service for a random question to bundle with the match room id and publish it to both users.

By implementing Matching service this way, there will only be at most 1 match stored per difficulty, thus not needing a large database.

Communication Service

Name: Communication Service	**Description**: The Communication Service is responsible for simple text based communication between matched users.
Capabilities: Text based communication, Chat history	
Service API	**Collaborators**

Event Published
<u>Client Side</u>
`'joinRoom'`
`'sendMessage'`
`'disconnecting'`
<u>Server Side</u>
`'receiveMessage'`
`'userDisconnect'`
`'setUsername'`	

The communication service handles the basic text-based communication between a pair of users in the same room. The complete functionalities of the service include: sending and receiving of text, asynchronous message handling, pseudo-synchronous chat history, room-based communication and status updates. We make use of a variety of methods to achieve this which will be detailed below.

Pub-Sub

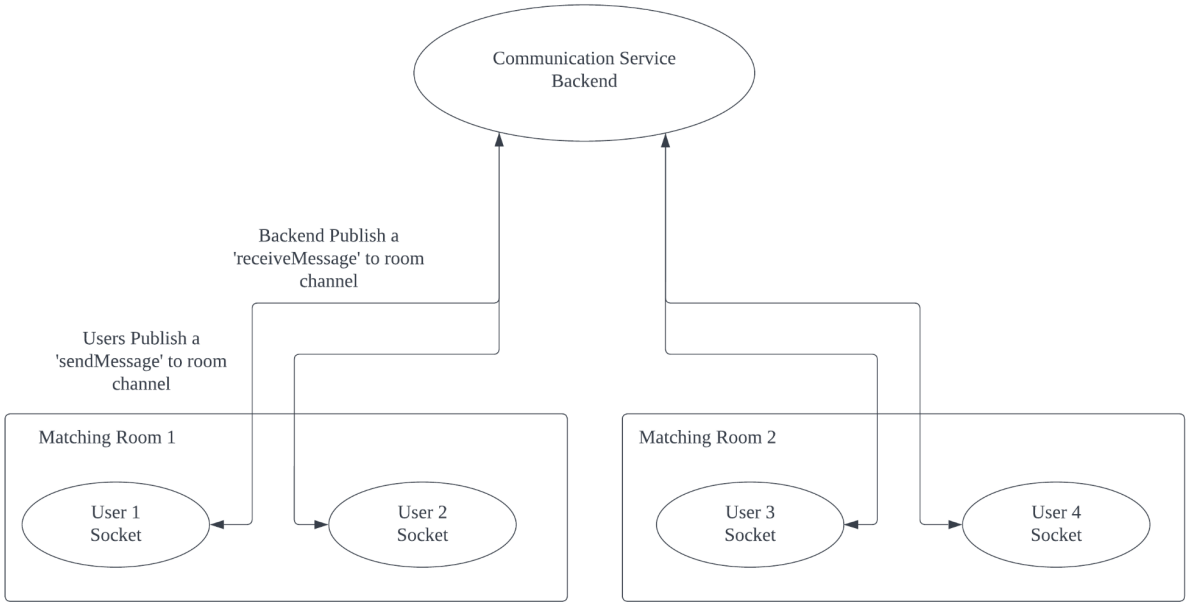


Fig. Pub-Sub Diagram

The communication service implements a publish-subscribe messaging design, where users and the backend serves as both a publisher and subscriber in their specific rooms. All of this is achieved using the socket.io library. To send a message, the user emits a ‘sendMessage’ event which the backend is listening for. In response to this event, the backend emits a ‘receiveMessage’ event back to the users in the room. The users will then update the state of their chat history with the message and relevant information. A breakdown of design considerations are as follows:

1) Handling the transfer of message

   In our initial design of the communication service, we planned for the backend to broadcast the ‘receiveMessage’ event to all users currently subscribed to its channel. In response, the users would check the sender and room number of the message to determine if they are the recipient.

   However, this proved to be highly inefficient and added unnecessary load to the backend. Since all but 1 user would be the recipient of the broadcasted message, it did not make sense to use broadcasting despite the simplicity of

implementation.

Instead we made use of ‘rooms’ in socket.io to segregate users into rooms with the user they were matched with. Not only did this reduce the volume of traffic per message sent, we also guaranteed the privacy of communication between matched users since messages are directed to specific rooms.

2) Handling the state of chat history

When initially planning out how the implementation of the communication service would be done, we had to decide how the messages from users would be passed. Our initial idea was for the communication service to have a database storing the chat history of all rooms it is subscribed to in a given session. After much deliberation, we determined that the need for a dedicated database would be unnecessary. In addition, the amount of data transferred between the backend and rooms would increase exponentially as the number of rooms increased.

The alternative we settled on was to have the state of chat history be stored on the frontend as a simple variable. This proved to be a sufficient compromise since each room only handled communications between 2 users. More importantly, it significantly reduced the load on the backend, now only needing to retransmit a single message at a time.

3) Ensuring pseudo-synchronous chat history

In the initial development of the communication service, it was designed such that when a user sends a message, it immediately updates the state of chat history in their instance of the frontend. This posed two problems.

Firstly, due to uncontrollable reasons like latency, poor internet connection etc. we realized that the chat history of matched users in the same room can have an obvious delay. Naturally, the user that sends the message would see a visual update of the chat history immediately whereas the receiving user does not.

Under normal load, this would be a negligible delay but under heavy load, the backend might take a longer time to emit the message to the receiving user.

Secondly, in the event that the communication service backend goes down, the user sending the message still sees the message on screen which is undesirable.

Our solution was to hold off on updating the state of chat history until a user receives a message. This also helped to loosely ensure that the chat history of both users remains in sync so long as the communication service backend is running.

Status updates

Building on our use of socket.io we also took into consideration one-time events that occur during a session such as “When a user first enters the matching room”, “When the user successfully connects to the Communication Service”, “When the user leave the ‘room’ gracefully” and “When the user is unexpectedly disconnected”. Making use of custom events, we were able to customize the expected behavior of the communication service in each event.

1) When a user first enters the matching room
   A ‘joinRoom’ event is published by the user. In response, the backend would connect the channel between the user and itself to a specified ‘room’.

2) When the user successfully connects to the Communication Service
   Upon successful connection using Socket.io, the frontend will update the state of the chat history with a message that informs the user of a successful connection. This information was useful for users to verify that the chat box is working.

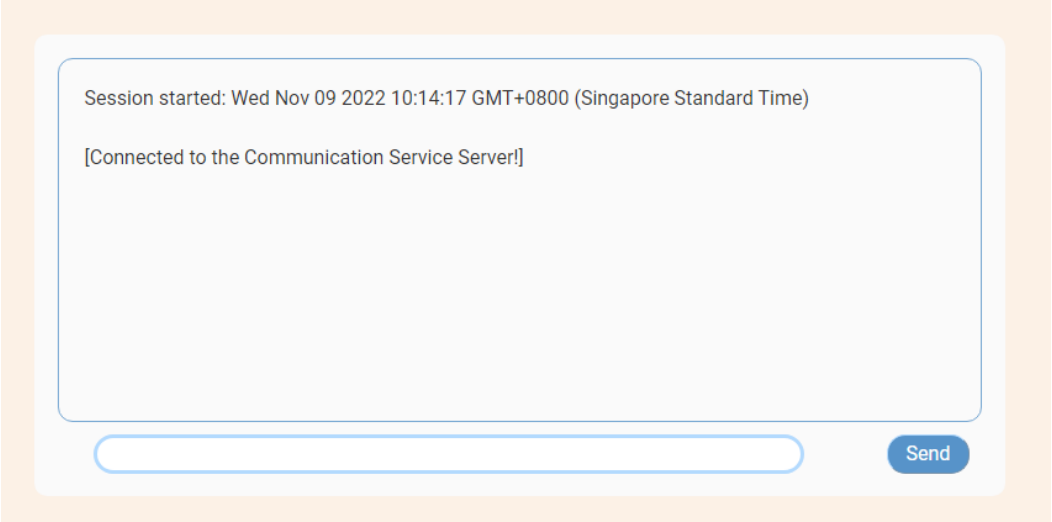


Fig. Successful Connection Message

3) <u>When users leave the ‘room’ gracefully</u>

As part of the functionality of the matching room frontend, users are able to leave the room using the ‘Home’ or ‘leave’ button. In either case, we consider this a graceful exit and handle it by sending a simple message to the room that a user has left.

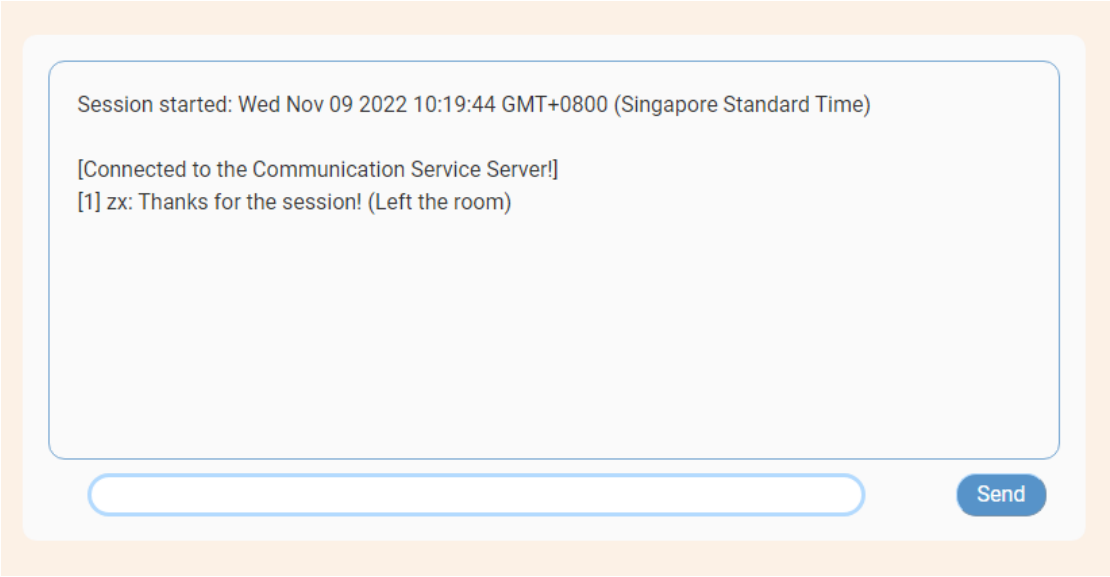


Fig. Message for Graceful Exit

4) When the user is unexpectedly disconnected

In the event that a user unexpectedly disconnects (closing the tab, browser crashes, internet connection is down etc.) we want to inform the other matched user in the room of this. The backend will listen on ‘disconnect’ events and in response send a message to the room informing the remaining user of the disconnection.

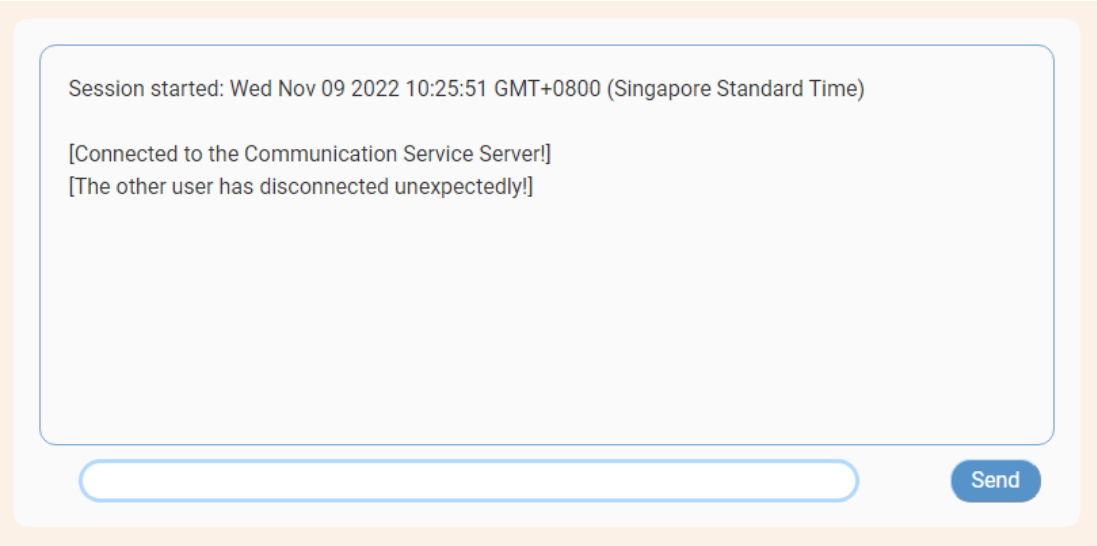


Fig. Message for unexpected disconnection

Collab Service

The Collab service allows for real-time collaboration between participants in the room via a concurrent code editor. The frontend is implemented as a react component with the Yjs library, Codemirror editor, and connects to the backend node js server using a websocket provider for Yjs called y-websocket.

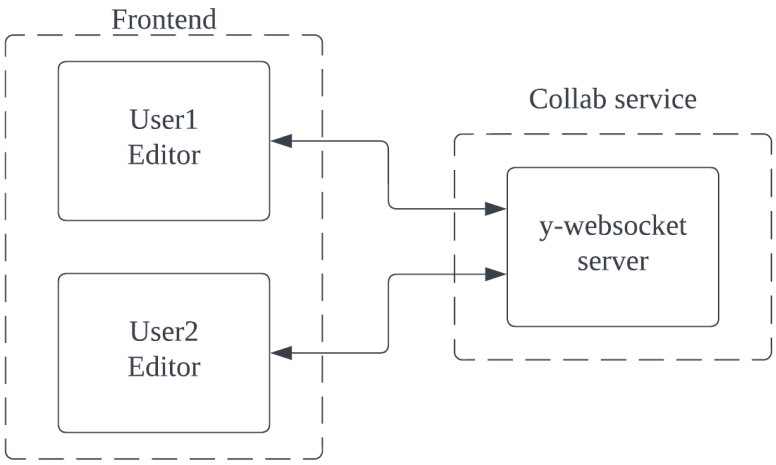


Fig. Collab service architecture

The Yjs library is a Conflict-free replicated data type (CRDT) implementation that allows document changes to be automatically distributed to other peers and merged without merge conflicts. It comes with many useful services and features such as supporting many rich-text editors, network providers, offline support, and more. In our use case, we use Codemirror as our editor and websockets for Yjs to communicate. Yjs also supports independent undo/redo operations and allows us to easily share cursor locations and highlighted text to clients with their respective username. This would not have been easy to implement if we were to create our own mechanism for collaborative editing with a pub sub mechanism like socket io. Furthermore, with the integration with the Codemirror editor, we are able to make use of Codemirrors features like making use of their editor themes, add gutter line numbers and allow us to add and even dynamically change syntax highlighting for different programming languages.

Frontend

UI Prototype

Before implementing the frontend components, we first designed the different pages using Figma to experiment and visualize our design.

Here is the link to our Figma file:
https://www.figma.com/file/qJgvK8ANWNyLZYhdcA9Rm5/UI-Design?node-id=0%3A1

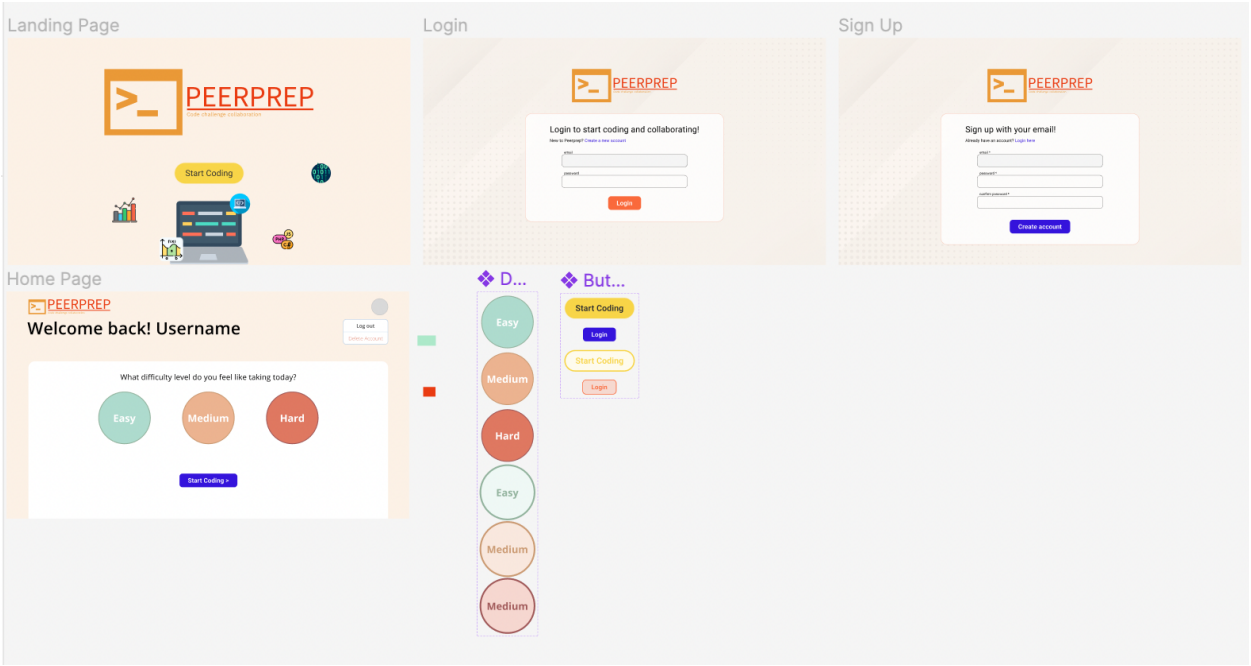


Fig. Screenshot of our Figma file

Frontend Architecture

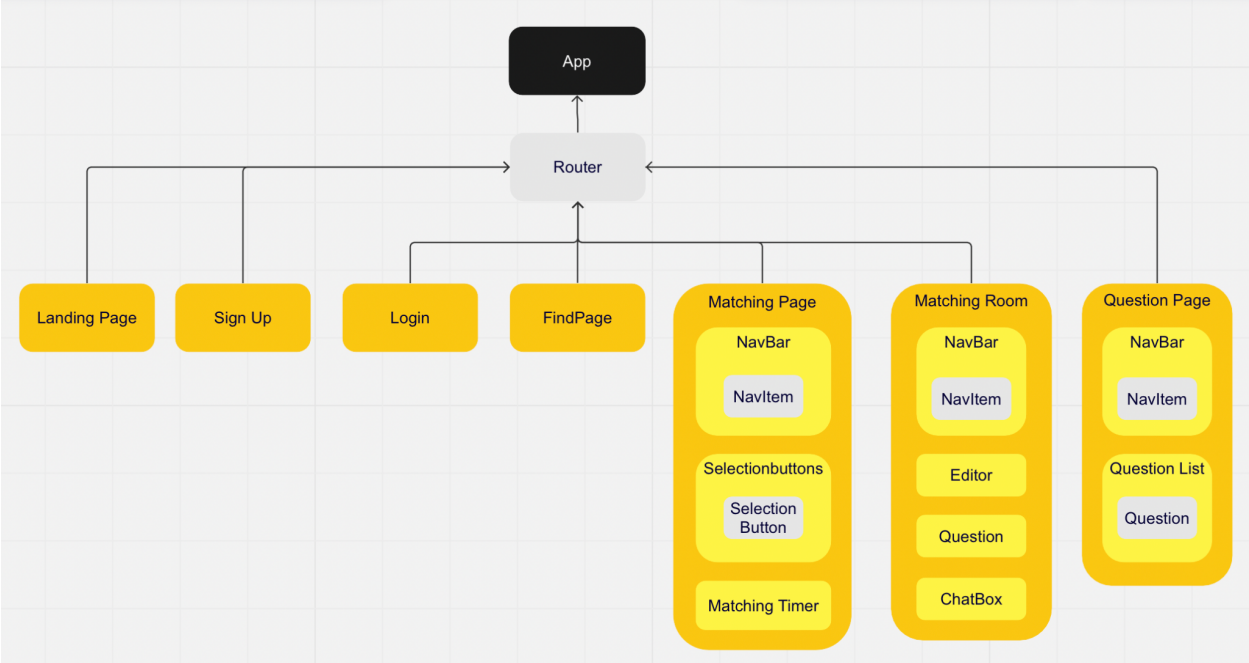
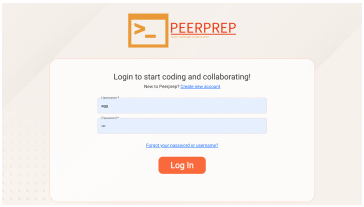
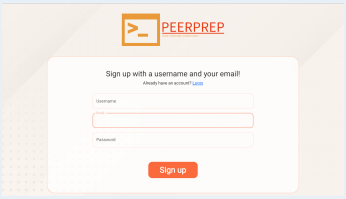
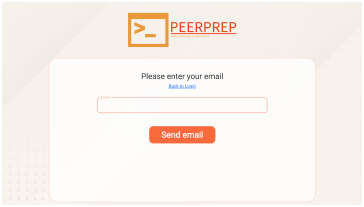
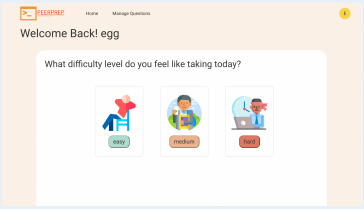
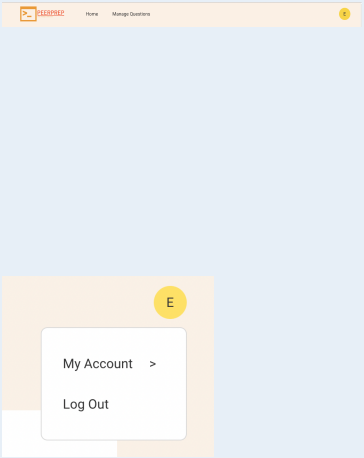
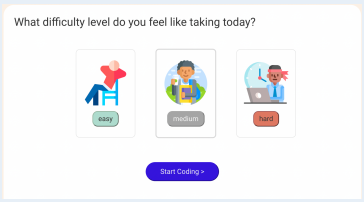
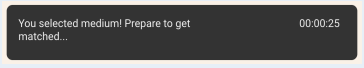


Fig. Frontend architecture diagram

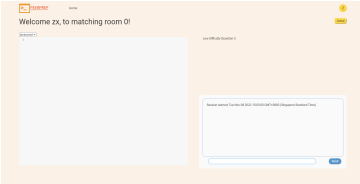
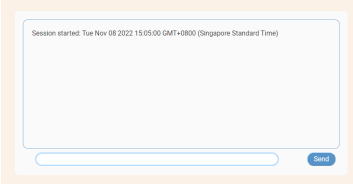
React Components

Page	Components	Purpose	Screenshot
-	App	Manages routing, root component of the application	-
Landing page	LandingPage	A landing page users see when they first load the application. Direct user to **login page**.	

Login page	LoginPage	Allows users to login using their username and password. Handles authentication, saves JWT in cookies. Offers a route to **sign-up page** for new users. Offers a route to **FindPage** if users forget their password.	
Sign up page	SignupPage	Allows users register a new account with username, email and password. Handles logic to validate username (must be unique) and email (must be in email format) Calls backend to create a new user.	

Find password page	FindPage	Allows users to find their password through email[1] Provide route to go back to login after password recovery email is sent	
Matching page	MatchingPage	Home page of the application, where users choose a difficulty to match with other users	
	NavBar	Navigation bar that contains links to navigate to **manage question page** Contains an profile icon which upon clicking, opens a **dropdown** for additional functions: logout, delete account, change password	
	Selectionbuttons	Contains buttons that represent different difficulties.Users can choose a difficulty and start matching	
	Matching Timer	Start timing 30 seconds	

[1] This function is not implemented in the backend due to time constraints. However, given more time we would definitely implement this function properly. Also see “Functionality Enhancement and Improvement”

		after user clicks ‘start coding’ If successfully matched, the user will be directed to **matching room**	
Matching room	MatchingRoom	The main page where users collaborate with one another	
	Editor	Code editor that synchronizes across the two users. Users can select a language and the code syntax highlighting will be formatted according to the language	
	Question	Displays a random question from the selected difficulty level	
	ChatBox	Users can type and send messages. System status and the status of the other user will be displayed here as	

		messages as well.	

Deployment

Deployment of the project was handled in two stages; the creation of docker files to aid in local deployment; the deployment of the application to hosting sites through the use of github actions to Heroku

Docker

To aid in local deployment as well as testing, the inclusion of docker files was a must.

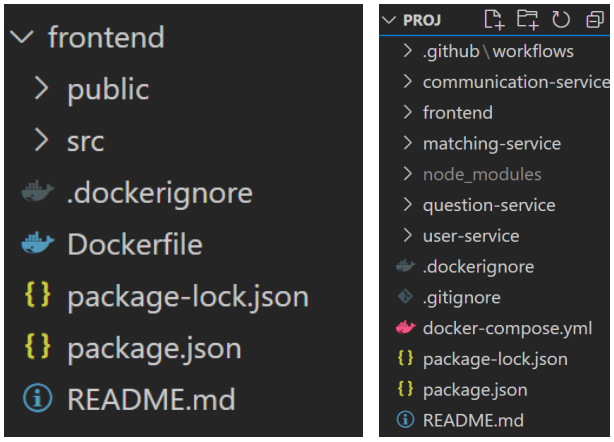


Fig. DockerFIle and docker-compose example

Each service would have its own dockerfile,which details what commands to install dependencies and run the services. A docker-compose.yml file is then also included in the source folder, which when run, runs every service's dockerfiles, simplifying local deployment.

Docker was also the chosen method, for it would also simplify production deployment.

Heroku

The hosting site chosen was Heroku. Heroku was chosen over other deployment services such as AWS, Google or Azure, for its simplicity and speed of deployment. Heroku was also the most familiar deployment service among the group, which meant any errors that occurred during deployment could be handled quickly.

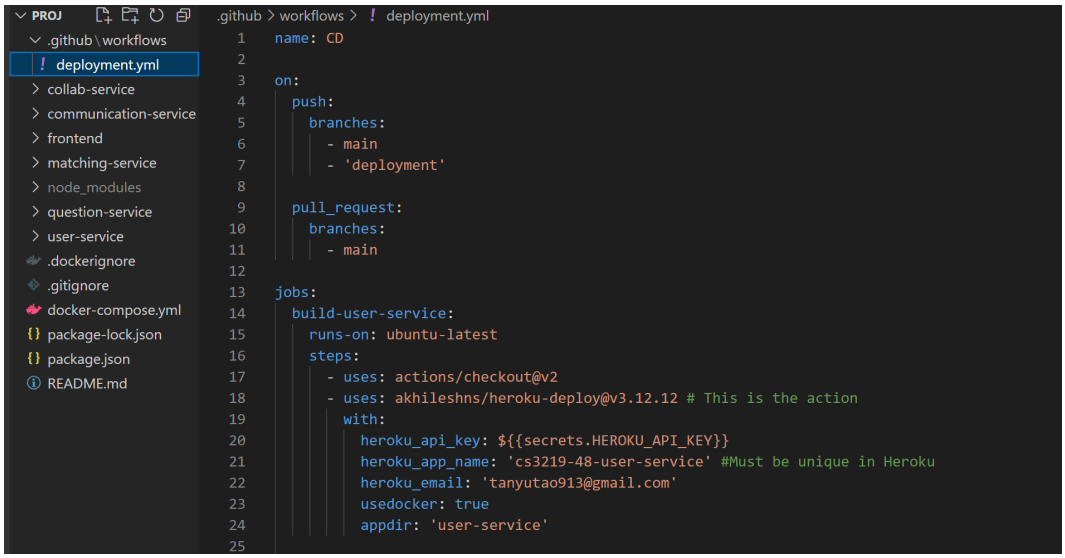


Fig. github actions heroku deployment workflow

To aid in the process of deployment, continuous deployment was also decided upon. This then finalized our decision for the process of deployment, which would be carried out through github actions, to then deploy our services to Heroku. With the docker files already present, deployment was then just the simple act of directing the deployment action to the correct working directory for each service and using it for deployment. The use of Github actions would also allow us to implement continuous integration in the future should we want to implement it.

Application Screenshots

Fig. Landing Page

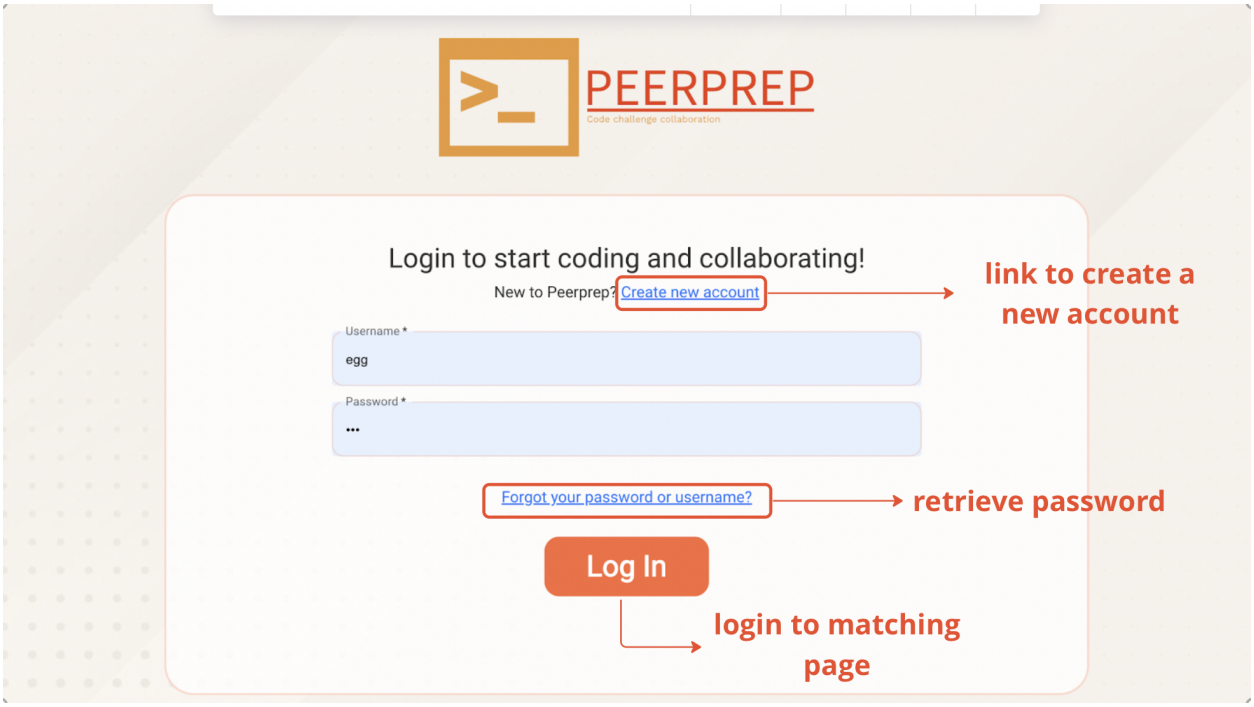


Fig. Login Page

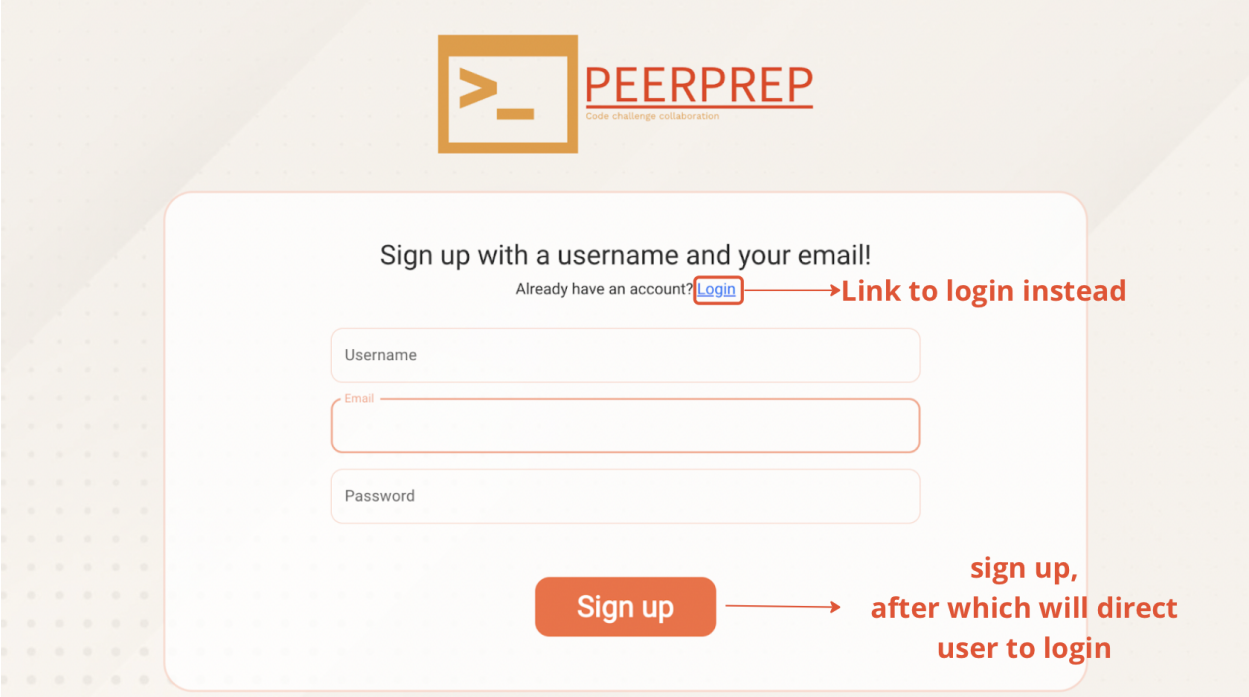


Fig. Signup page

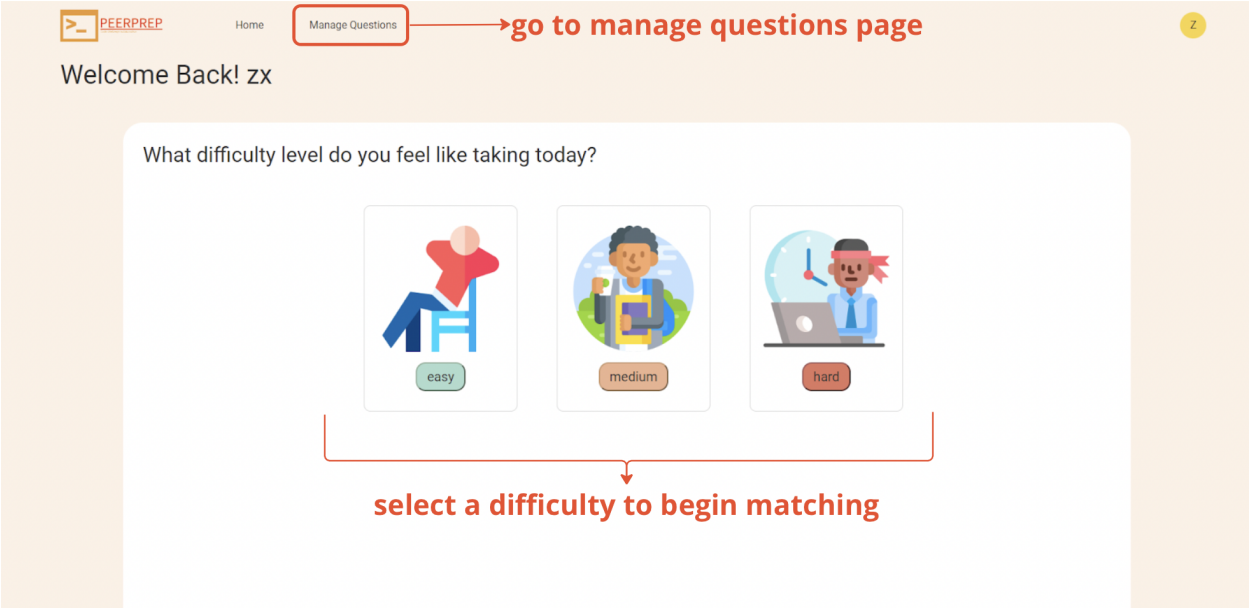


Fig. Matching Page

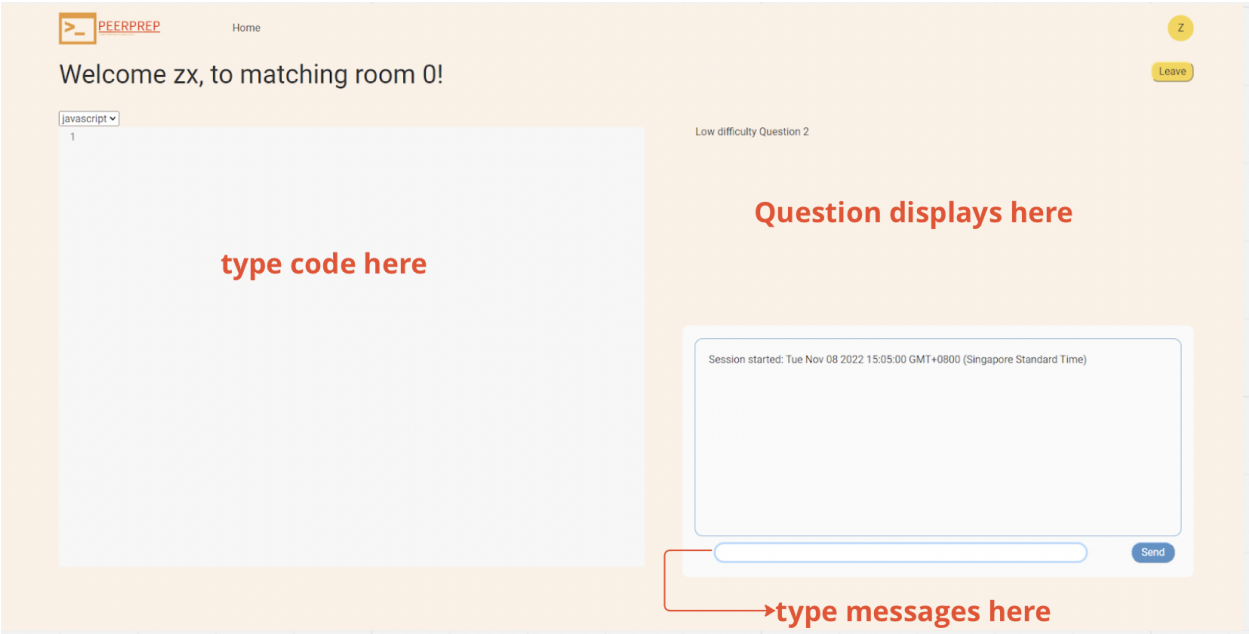


Fig. Matching Room

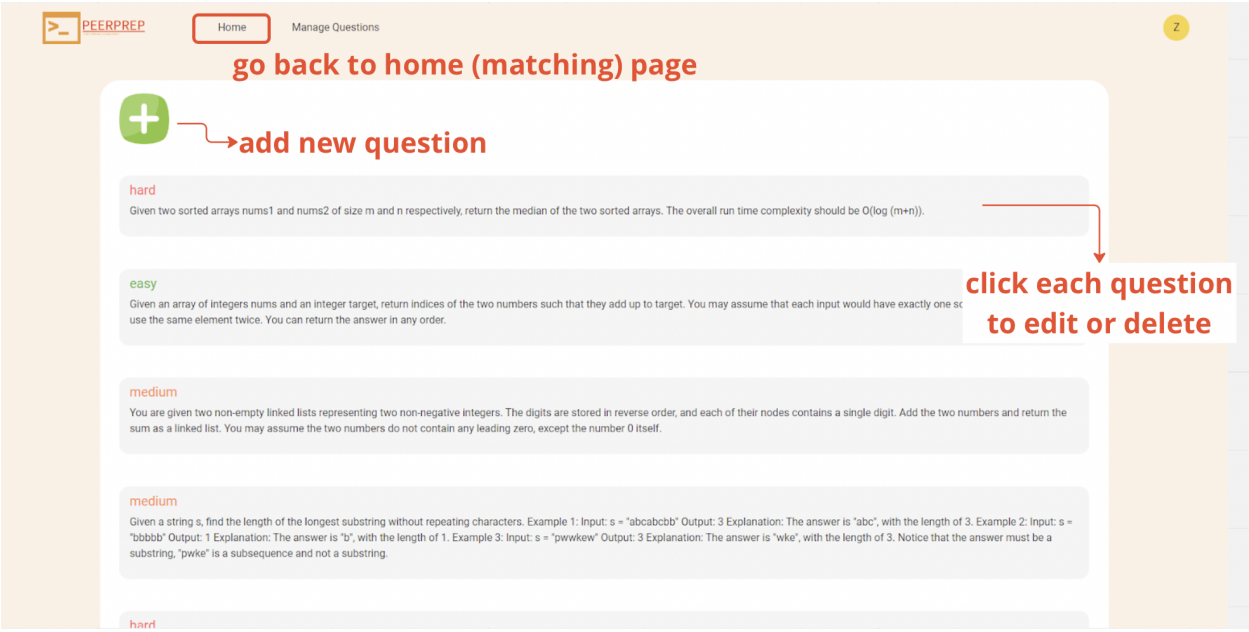


Fig. Question Page

Functionality enhancements and improvements

Various possible areas for improvement have also been considered, throughout the development process. Below are various improvements sorted by their various relevant components.

User Service

The user-model can see various improvements and functionality enhancements. Our current User's already have an email field.

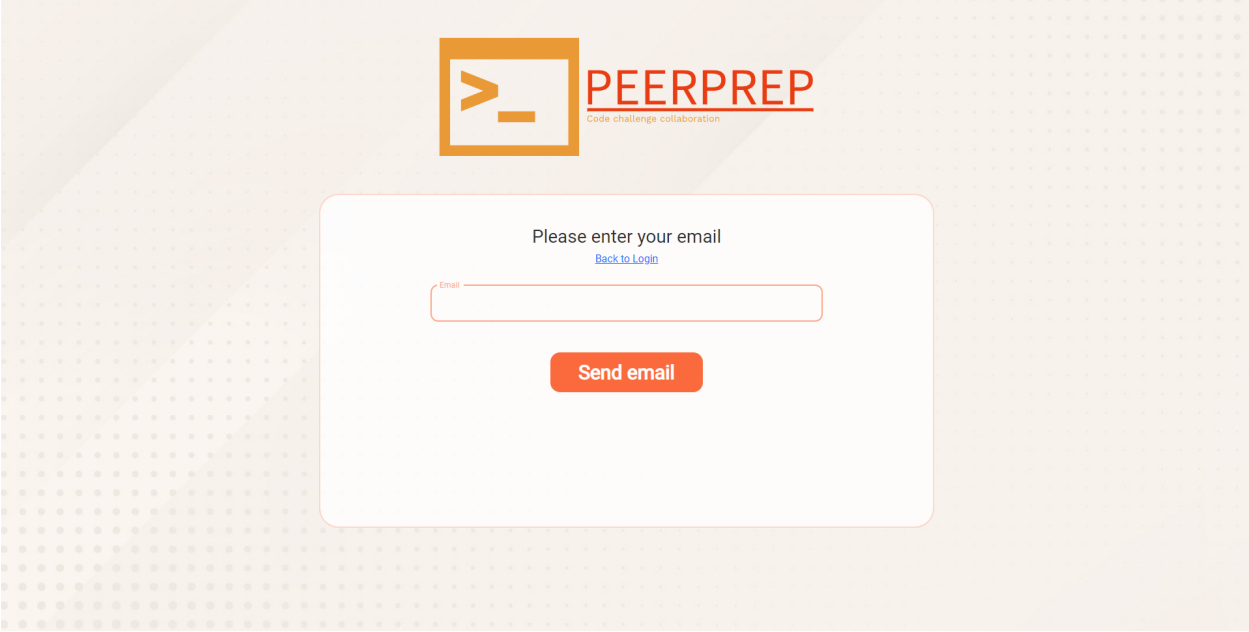


Fig. password retrieval page

The functionality to then allow users to retrieve their passwords using their provided email would then be one possible enhancement that would be good to have.

Another improvement that was considered is to have various roles which can be assigned to users from which various functionalities could be sectioned off based on authorization level.

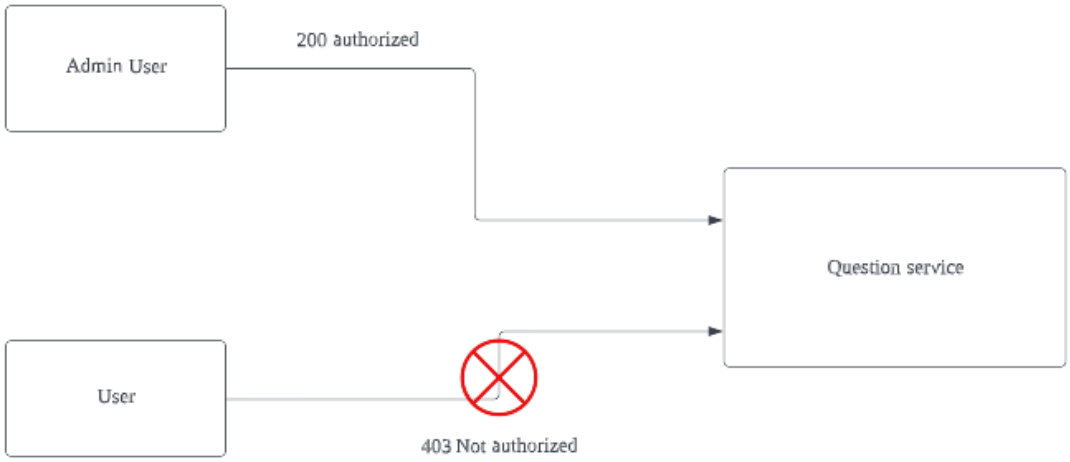


Fig. question service authorization diagram

The question service is one such functionality that we considered restricting based on User authorization level. These improvements can be easily implemented thanks to our choice of using a no-sql database. Where databases do not have to strictly have identical documents unlike those in a standard sql database. The microservice architecture also allows us to implement changes without it resulting in regression in other services.

Question service

The question service currently only allows the storage of questions as string, hence one improvement that could be considered is to allow the storage of questions with images. This would then constitute additional changes to the frontend component as well to properly display questions that have images.

Redis could also be implemented in the question service to help reduce loading and access to the database when getting all questions.

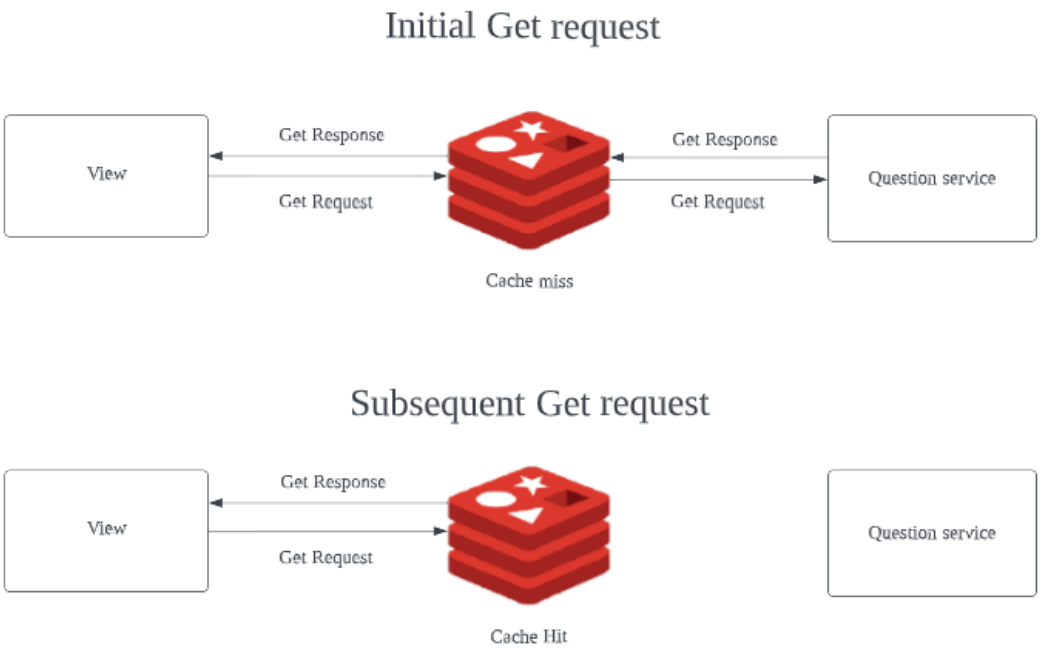


Fig. redis get diagram

This could greatly benefit user experience if a search functionality is included into the question service, by reducing subsequent search times after the initial request to the database

Reducing the coupling between Matching and Question Service

Currently, the retrieval of questions from the question service for a specified difficulty results in a randomly selected question which is intended. To retrieve these questions, the matching room frontend would make a GET request to the Question Service. However, after repeated testing, we concluded that this feature made it impossible for 2 matched users making independent GET requests to receive the same question consistently.

As a stopgap measure, we updated the implementation of the matching service such that it now makes the GET request before emitting the ‘matchSuccess’ event. While this did achieve what we want, it also led to coupling between both services.

One possible solution is to instead have the matching service generate a random question number corresponding to a question id stored in the question service database. This random number would be generated and passed on to the matching room frontend and used as an argument to retrieve a specific question. A POST request would be made instead and to handle the case where the random number generated does not correspond to any id in the database, we could use modulo arithmetic to reduce the range of the randomly generated number to the total number of existing questions in the database.

Additional Services

History service could be implemented, to track the various questions attempted by the user. This enhancement could also be implemented through the user service, with an additional array field. Either of which can be easily implemented thanks to the microservice architecture.

A compiler could also be implemented. This could be carried out through the use of external packages to help with the compilation. This service could be easily extended onto the frontend portion of the application.

Reflections/learning points from the project process

Through this project, we have acquired the ability to design and develop web applications from start to finish. We have learnt how to apply the knowledge from lectures (such as development cycles, software requirement specification, different software architectures, design and messaging patterns) to real application development.

Relating those concepts back to CS2103T, we can now better appreciate how microservices architecture differs from the monolith structure and its advantages. The concepts and the keywords we learnt also greatly enhanced and sped up the communication and decision making process during our weekly meetings.

We were exposed to many trending technologies that are currently in use by major technological companies. Skills in React, NodeJS, Docker or Heroku will definitely prove to be invaluable in our future software engineering journey, be it for internships or personal projects. Having to learn all these technologies on our own in such a short time frame was also a real test of our ability to adapt, which is an important quality for software engineers. Through more than 10 weeks' continuous teamwork, we have further honed our communication and problem solving skills and learnt how to deal with fast-paced projects and tackle the challenges as they come.

This was also an eye-opening experience to full stack development and gave us an idea of what devops could be like in our future projects. More importantly, different members of the team were equipped with the knowledge to deal with different problems which greatly reduced the burden of executing full stack development and deployment. This also made the learning process smoother as we had an expert in each field to refer to.

Lastly, looking into the future, we hope to bring the skills and knowledge gained from this project not just to our future workplace, but in any future software engineering projects we might work on.