Module Name: Software Engineering Principles and Patterns
Module Code: CS3219
Academic Year: 2022/2023 Semester 1

Group 35

Name	Matriculation Number
Ryan Cheung Jing Feng	**A0217587R**
Toh Bing Cheng	**A0214278A**
Lau Siaw Sam	**A0199107X**
Tay Jun Yang	**A0169792J**

Table of Contents:

Repository

Our repository is available on Github and can be accessed here.

Project Scope

Background

Increasingly, technical interviews are becoming the norm for companies looking to hire. Technical interviews are usually held online, and involve interviewers presenting interviewees with a few algorithmic questions. Apart from providing a solution, interviewees are expected to walk through their thought process and communicate with the interviewer. Technical interviews are also usually used in the final stages of the interview process, and hence is an important component which students may need to practice.

Project Scope

As such, we have developed PeerPrep in order to address these concerns. PeerPrep is a web based application that allows students to better prepare themselves for interviews. PeerPrep provides students with a platform to simulate technical interviews and practice algorithmic style questions at the same time. It utilizes a peer learning system in order to break away from the monotony of revising alone.

Tech Stack

For this project, our team has chosen the following tech stack to build PeerPrep.

Frontend	ReactJS
CSS	Material UI, TailwindCSS
Backend	Express, Node
Database	MongoDB, Redis
ORM	Mongoose
Deployment	Docker, Digital Ocean, GitHub Actions
Others	Socket.io

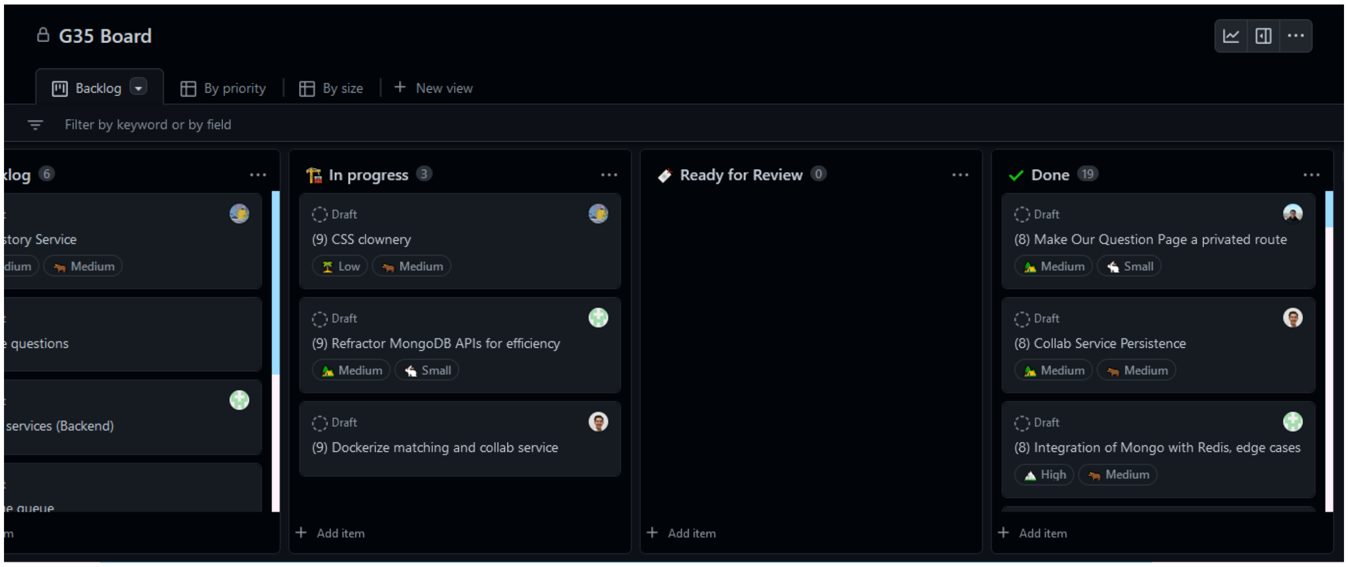
Library Dependencies

- axios
- express
- socket.io
- bcrypt
- mocha
- chai
- chai-http
- cors
- mongoose
- redis
- dontenv
- jsonwebtoken
- uuid
- js-cookie
- moment
- prismjs
- react
- react-router-dom
- react-simple-code-editor
- react-toastify
- tailwindcss
- mui
- emotion

Software Development Lifecycle

Our team follows the Agile model of the Software Development Lifecycle process with weekly sprints and tickets. We base our milestones similar to the timeline of the ones stated in the project description document.

To manage our agile workflow, we use several tools such as Github Boards to track the progress for our weekly sprints.


G35 Board

Backlog | By priority | By size | New view

Filter by keyword or by field

klog 6	In progress 3	Ready for Review 0	Done 19
story Service — dium, Medium	Draft (9) CSS clownery — Low, Medium		Draft (8) Make Our Question Page a privated route — Medium, Small
e questions	Draft (9) Refractor MongoDB APIs for efficiency — Medium, Small		Draft (8) Collab Service Persistence — Medium, Medium
services (Backend)	Draft (9) Dockerize matching and collab service		Draft (8) Integration of Mongo with Redis, edge cases — High, Medium
e queue			
m	Add item	Add item	Add item

A snapshot of our GitHub Board

As we used an iterative based approach in our development, we first started our development process by generating user stories in order to derive the functional and non-functional requirements.

User Stories

When designing PeerPrep, our team crafted several user stories by perceiving ourselves as clients. As PeerPrep is a tool made for students to collaborate and simulate technical interviews, we as students are part of the user base.

High Priority (Must haves)

1. As a new user, I want to create an account on PeerPrep.
2. As a user, I want to log into my account.
3. As a user, I want to delete my account.
4. As a user, I want to be offered a variety of questions ranging from difficulty and topic.
5. As a user, I want to be able to choose a difficulty level.
6. As a user, I want to be matched with others to attempt questions of similar difficulty together.
7. As a user, I want to be able to collaborate on shared code.
8. As a user, I want to be able to chat with my peer in real time.

Medium Priority (Nice to haves)

9. As a user, I want to change my password.
10. As a user, I want to rate peers who I have been matched with.
11. As a user, I want to see my rating to see what aspect of interviews I can improve on.

12. As a user, I want to avoid doing questions I’ve attempted before.
13. As a user, I want a clean and intuitive UI so that I know where to find different features.
14. As a user, I want to leave my search anytime.
15. As a user, I want to be able to reconnect to the room if I get disconnected.
16. As a user, I want to be able to see the chat history if I get disconnected.
17. As a user, I want to be able to see the code history if I get disconnected.

Low Priority (Unlikely to haves):

18. As a user, I want to view past questions I’ve attempted.
19. As a user, I want to be able to video call with my peer to simulate real life interviews.
20. As a user, I want to be able to have a few prompt questions to simulate interviews.
21. As a user, I want to be able to view recommended hints when I’m stuck.
22. As a user, I want to be able to view a model answer to compare to.
23. As a user, I want to be able to add friends to rematch with those I’ve collaborated with.
24. As a user, I want to be able to manage my friends easily.
25. As a user, I want to be able to invite my friend to my room.

Functional Requirements

From the user stories, we are able to segregate related stories into a contained unit and assign priorities to each unit. We then realized which microservices are needed and which user stories they address.

Feature	User Stories	Requirements	Priority
User Service	1, 2, 3, 9	The system should allow users to log in or create an account. Users should also be able to modify account details.	High
Matching Service	6, 14, 15	The system should allow users to choose the difficulty level, and allow them to find a match. The system should also handle exceptional events in the case of a disconnect or a reconnect.	High
Question Service	4, 5, 12	The system must be able to generate a question for the user based on their selected difficulty level. The system must also provide the user a selection of difficulties.	High
Collaboration Service	7, 8, 16, 17	The system should be able to provide a live platform for users to collaborate together.	Medium

		They should be able to have real time communication, and be able to have a real time code editor. The system should also handle exceptional events in the case of a disconnect or a reconnect.	
UI and Web App	13	The system should have a clean UI to ensure user experience is good. (Components are not cluttered)	Medium
Review Service	10, 11	The system should display user's reviews, as well as provide the option to leave a review for the partner after collaborating.	Medium
History Service	18	The system should keep track of previously attempted questions and have the ability to avoid showing repeated questions.	Low
Collaboration Service (Extension)	19, 20	The system should provide a live streaming platform for real time collaboration. The system should also provide several guiding questions typically asked in interviews.	Low
Question Service (Extension)	21, 22	The system should provide hints if users are stuck on a question. Additionally, the system can provide a model answer for users to compare against.	Low
User Service (Extension)	23, 24, 25	The system should allow users to add friends and maintain them in a friend list. Users should be able to start new rooms with their desired friends.	Low

Non Functional Requirements

We are also able to generate several non-functional requirements for our application based on the user stories.

Function	Requirements	Priority
Availability	The system should deal with faults and ensure uptime if possible.	High
Compatibility	The system should work with any modern browser of the latest version.	High

Persistency	The system should save after changes to minimize data loss.	High
Reliability	The system should not crash from undesirable inputs but instead reflect the relevant error message.	High
Testability	The system should recover from expected faults and not crash on bad inputs.	Medium
Security	The system should protect the user's details and encrypt sensitive information.	Medium

Notes on Diagrams

Sequence Diagrams

We follow the common convention that the return arrows of function calls and activation bars are optional in order to reduce clutter. However, in cases where the return values of the functions are meaningful, we include the return arrows with labels. Also, in cases where the lifetime of a function is ambiguous (for instance, with parallel or nested calls), we choose to draw activation bars to aid the reader.

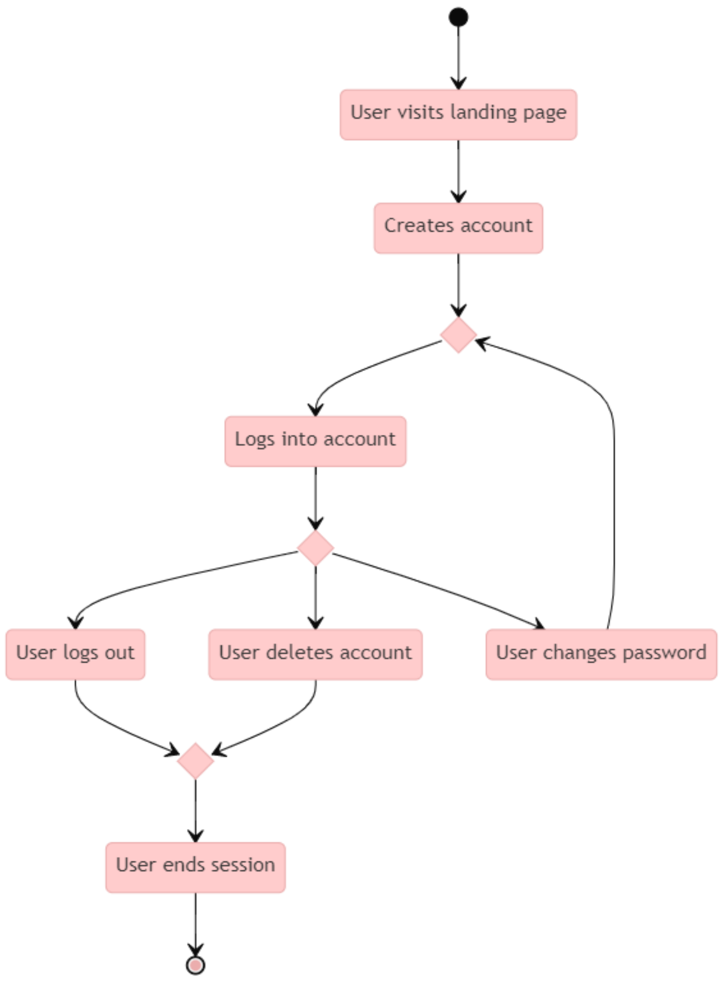
Class Diagrams

As our backend services consist of Express.js and Socket.io servers, which are not usually implemented with Javascript classes, our class diagrams are shown in terms of the components and files of a service instead. In the diagrams, each component is its own file, and the methods within each component are the methods within the file. A ‘public’ method is one that is exported and a ‘private’ method is one that is not.

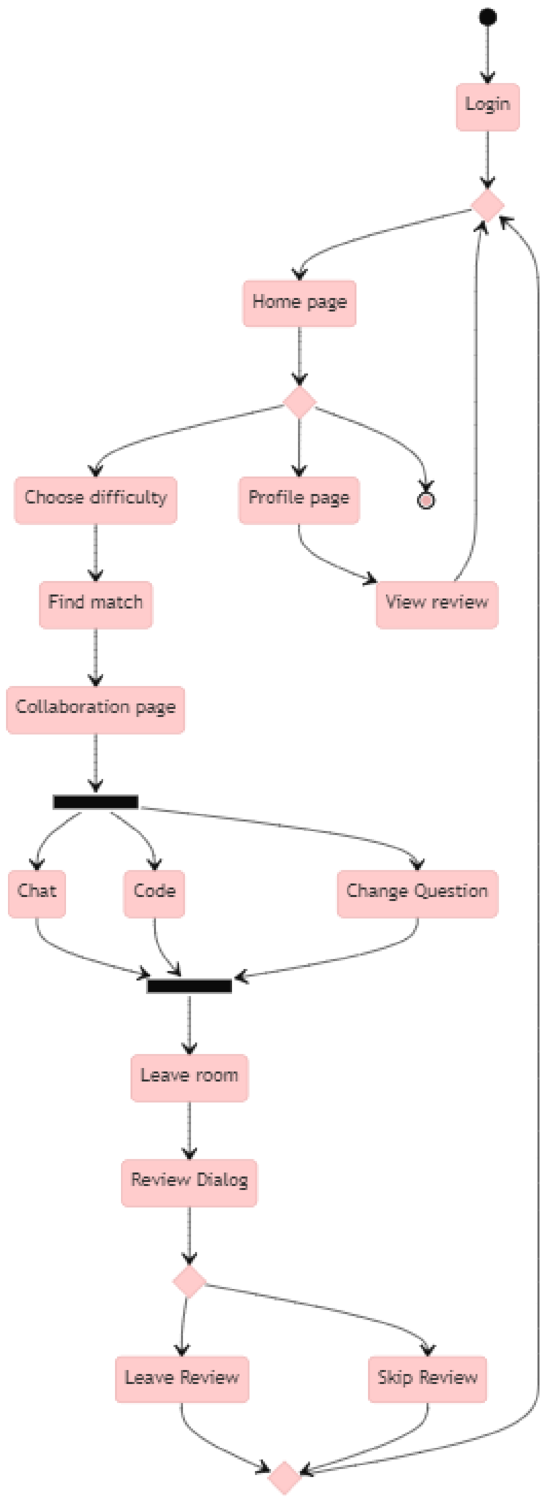
<u>Overall User Flow</u>

To better understand the workflow of our application, we have provided end to end activity diagrams to describe the actions that one may take while using our application.

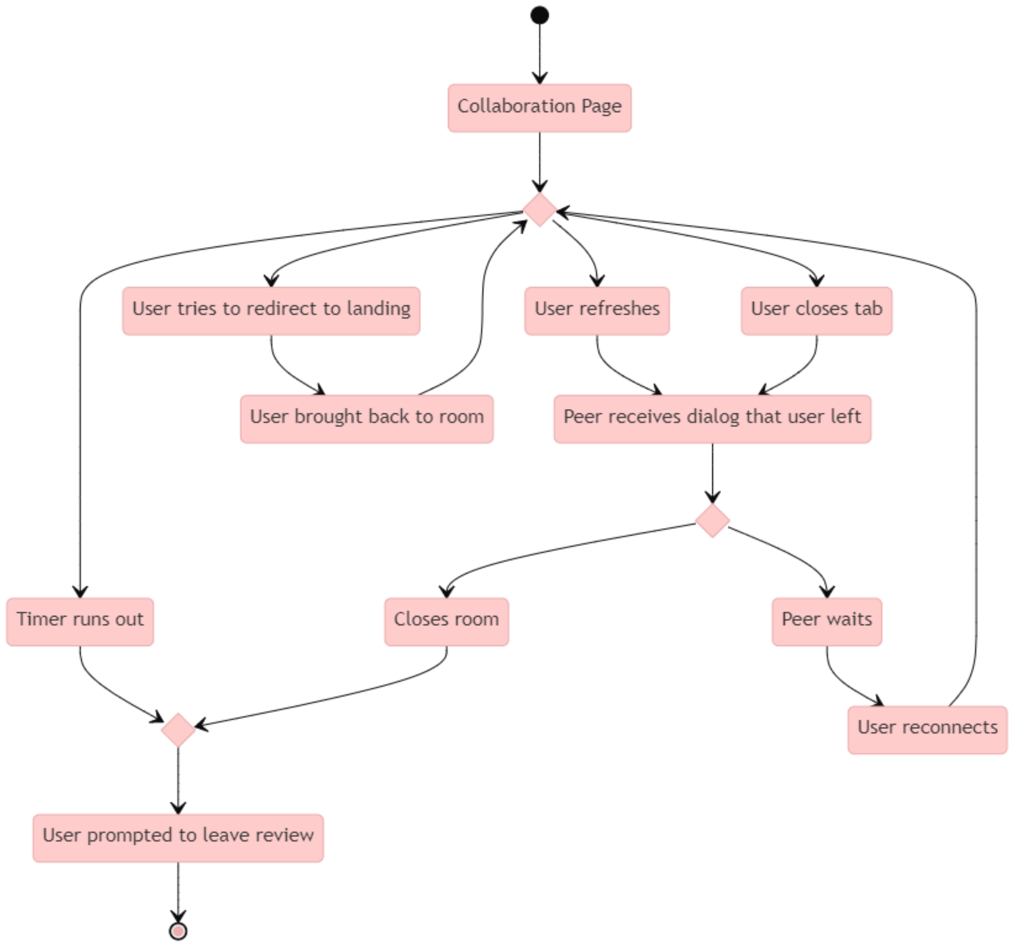
First-time User Flow



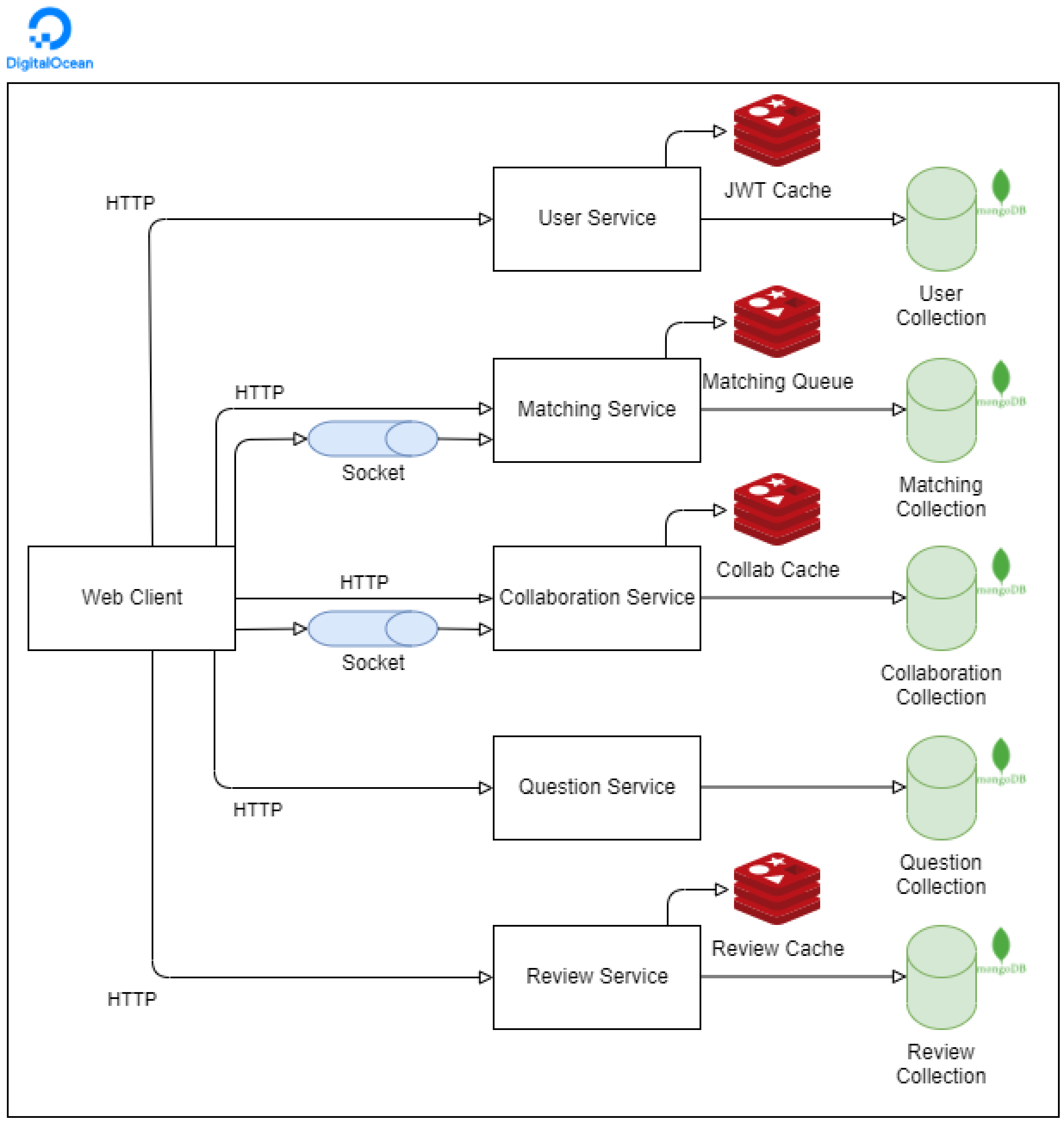
Normal User End-to-End Flow



User Flow on the Collaboration Page


Collaboration Page
User tries to redirect to landing
User refreshes
User closes tab
User brought back to room
Peer receives dialog that user left
Timer runs out
Closes room
Peer waits
User reconnects
User prompted to leave review

Architecture Diagram



PeerPrep's Architecture Diagram

Legend

- Square Boxes: Individual microservices
- Redis logos: Redis caches
- Green cylinder: MongoDB Database
- Blue cylinder: Socket

Shown above is the current architecture of PeerPrep at a high level. We are hosting our microservices on a DigitalOcean droplet. More details about our continuous deployment processes and HTTP communication can be found at the bottom of the document. Each backend service has its own collection. Several of our backend services use a redis cache, each for its own purpose.

Microservices

Frontend

Code Organization

As a React application, the frontend's codebase is centered around the concept of atomic and reusable components. As such, the codebase contains two types of React entities, pages and components. Pages correspond to a single page that is rendered at a given route by React Router. They are composed of multiple components and are responsible for composing them and laying them out on the page. On the other hand, components are smaller and reusable entities. In general, if something has to be rendered in more than one place, we extract it out to become a reusable component (e.g. a dialog component). On top of this hierarchy, all our React components interact with the backend APIs via service wrapper classes, which act as an abstraction barrier between React components and lower-level Axios HTTP calls.

Noteworthy Engineering Decisions

Below are some interesting choices the team made while developing the frontend, taking into account and weighing different considerations.

- When deciding how to store JWTs on clients, the team deliberated between two commonly-used methods: cookies and browser localStorage. In the end, the team decided to store JWTs using cookies as cookies are semantically meant for server-side use while localStorage is meant for client-side use. Additionally, the OWASP community recommends the use of cookies to store JWTs for security reasons. Lastly, cookies support expiry dates, which come in handy since JWTs expire.
- Use of React context for state management. Given that our application has multiple layers and nested components, we require a state management tool as passing state just between parents and children is insufficient. Our team deliberated between React context and Redux. Whilst our team acknowledges the benefits of Redux, we feel that given our requirements, it is unnecessary and adds too much overhead. We currently only require a UserContext, and as it does not have many fields at the moment, we thus opted for the simpler option of React context.
- We chose to use client-side rendering (CSR) instead of server-side rendering (SSR) solutions such as NextJS. CSR generally provides better performance and user interactivity as navigation doesn't require extra round-trips to the server. On the other

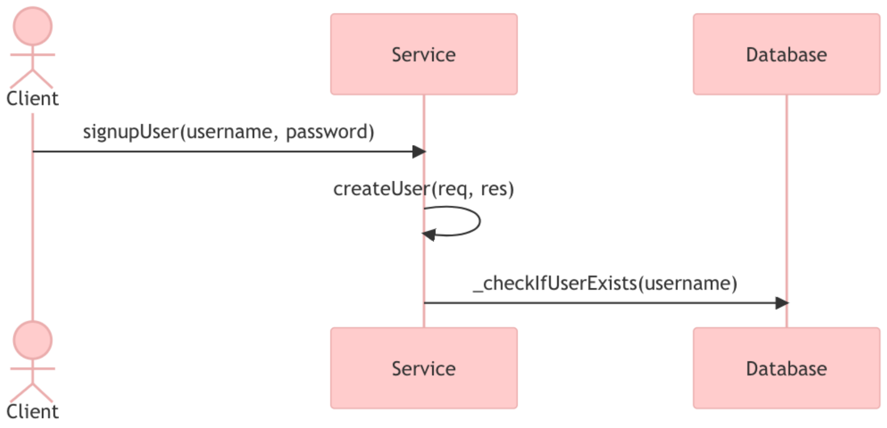
hand, SSR is generally used to improve the SEO of a web application as search engines cannot crawl and index CSR apps easily, since the site is rendered dynamically on the client side. For PeerPrep, however, there is no need for SEO-optimization since our pages do not store public indexable content. As such, we went with CSR.

User Service

The user service is responsible for all user authentication of the system. It supports the CRUD (Create, Retrieve, Update and Delete) operations of a user account. Each user account information is stored in the MongoDB user collection. Redis is used to ensure that a logged-in user possesses a valid JWT (JSON Web Token) in order to access the private routes of the system. The JWT expires after 15 minutes, after which any actions performed on private routes will redirect the user to the login page. When the user is placed in a collaboration room with another user, the JWT for both users is extended by 30 minutes. This is to ensure that both users possess a valid JWT during the collaboration and prevent any inconvenience which may arise from JWT expiration.

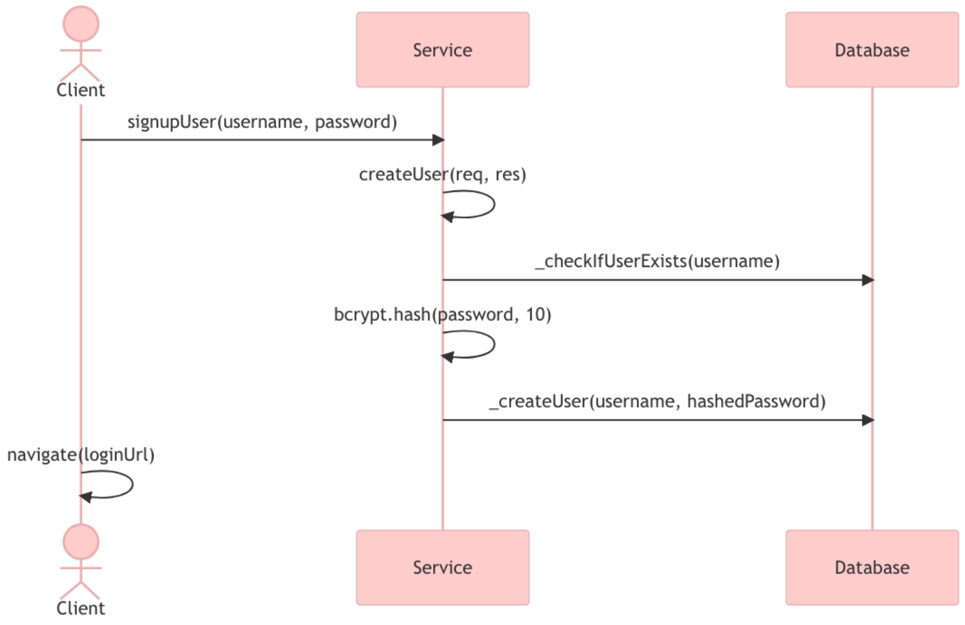
Signing Up

To access PeerPrep, a user will have to first create an account. The user inputs the username and password text fields before clicking the 'sign up' button. Basic input validation checks have been implemented on the frontend to ensure that both username and password text fields are not empty when the user clicks the 'sign up' button. After clicking on the 'sign up' button, the service initiates the createUser request which will then check whether the username exists in the database. If the username exists in the database, the service will then return a response to the client to notify the user.



Sequence diagram when username exists in the database

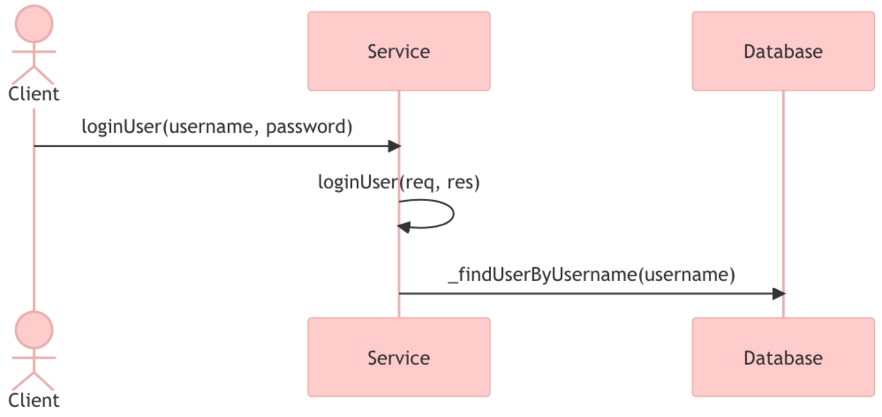
If the username does not exist in the database, the password is then salted and hashed with bcrypt library before the user (with the username and hashed password) is created and stored into the database. The service will then return a success response to the client to notify the user.


Client
Service
Database
signupUser(username, password)
createUser(req, res)
_checkIfUserExists(username)
bcrypt.hash(password, 10)
_createUser(username, hashedPassword)
navigate(loginUrl)
Client
Service
Database

Sequence diagram when account is created successfully

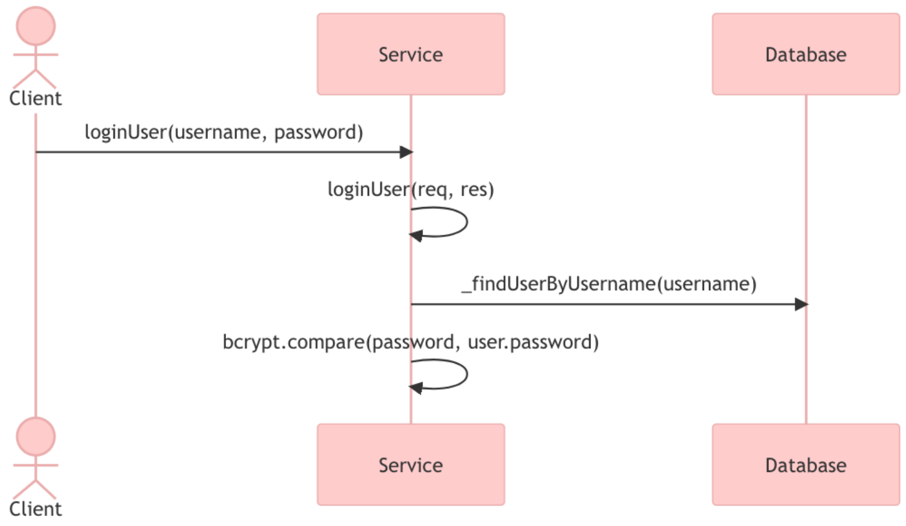
Logging In

To access PeerPrep, a user will have to log in to their account. The user inputs the username and password text fields before clicking the ‘log in’ button. Basic input validation checks have been implemented on the frontend to ensure that both username and password text fields are not empty when the user clicks the ‘log in’ button. After clicking on the ‘log in’ button, the service initiates the loginUser request which will then find the user by the username. If the username does not exist in the database, the service will then return a ‘wrong username and/or password’ response to the client to notify the user.



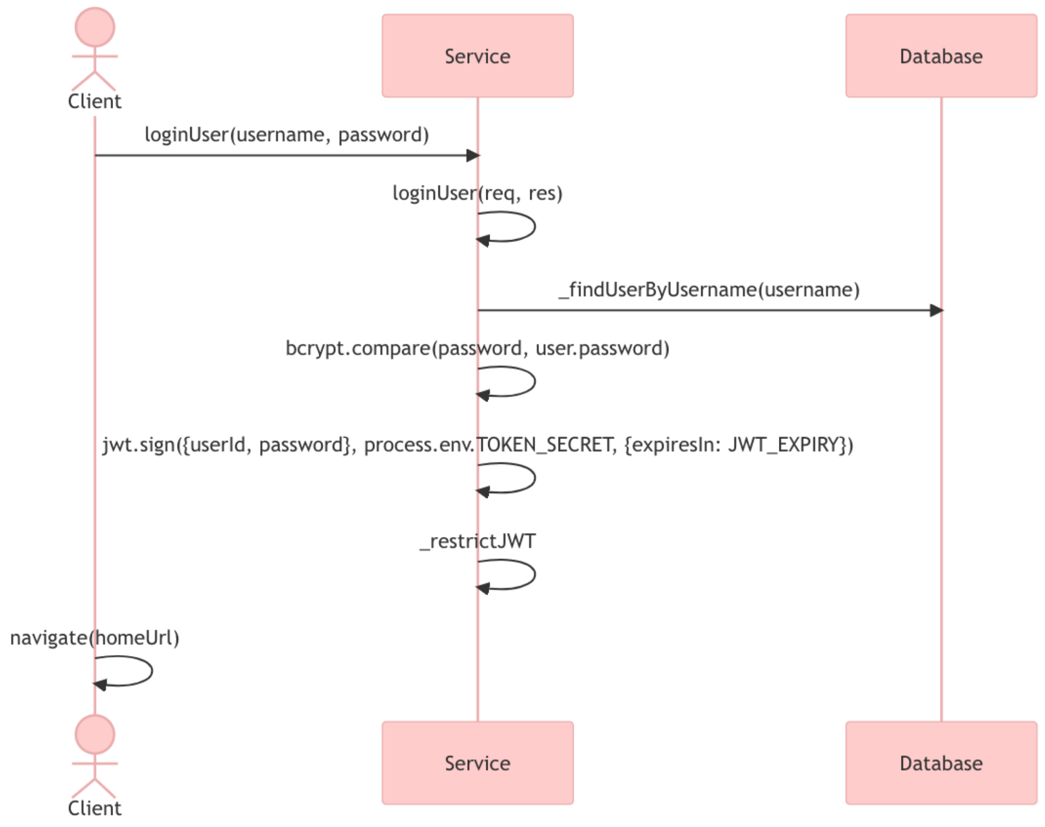
Sequence diagram when username is wrong

If the username exists in the database, the service will then compare and check whether the input password and the user's password are the same. If the passwords are different, the service will then return a 'wrong username and/or password' response to the client to notify the user.


Client
Service
Database
loginUser(username, password)
loginUser(req, res)
_findUserByUsername(username)
bcrypt.compare(password, user.password)
Client
Service
Database

Sequence diagram when username is correct and password is wrong

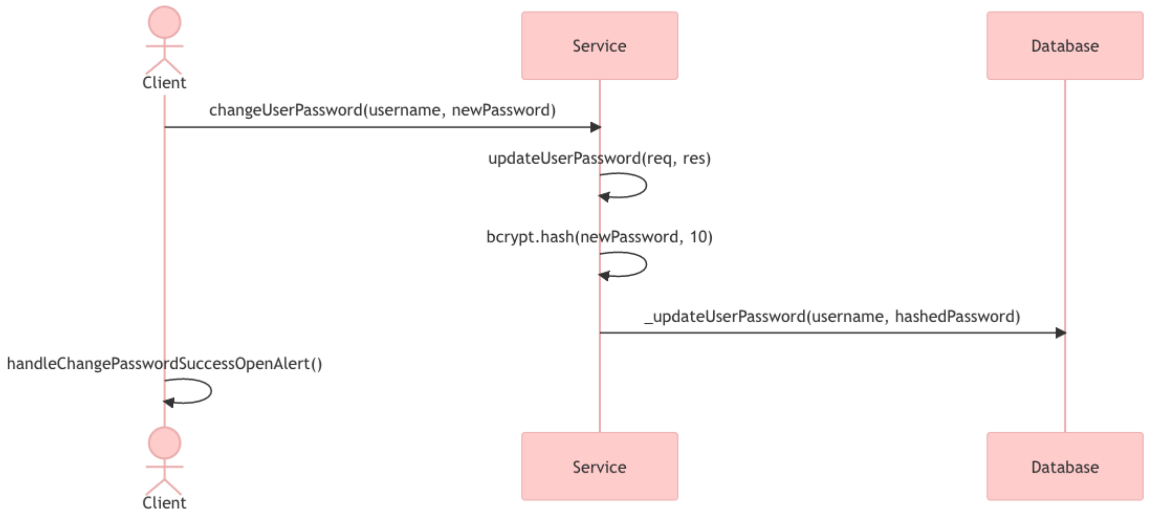
If the passwords are the same, the service will then sign a JWT with the user's credentials, a secret key and an expiration time of 15 minutes. The service will then restrict the JWT, which ensures that the user only has one valid JWT at any given time (which is the given token). This prevents the user from using the app on multiple devices at the same time. The service will then return the success response to the client to notify the user.


Client
Service
Database
loginUser(username, password)
loginUser(req, res)
_findUserByUsername(username)
bcrypt.compare(password, user.password)
jwt.sign({userId, password}, process.env.TOKEN_SECRET, {expiresIn: JWT_EXPIRY})
_restrictJWT
navigate(homeUrl)
Client
Service
Database

Sequence diagram when user is logged in successfully

Changing Password

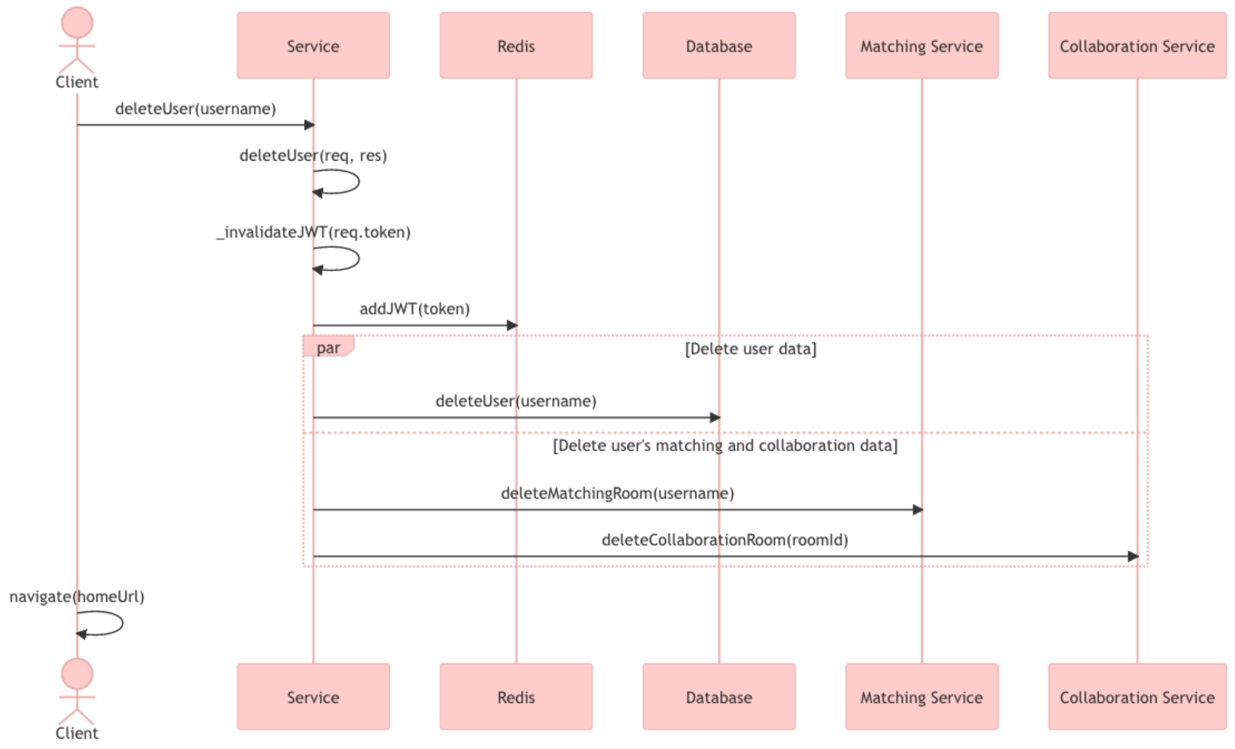
A user is able to change their password in the Profile page. Basic input validation checks have been implemented on the frontend to ensure that the new password text field is not empty when the user clicks the ‘submit’ button. After clicking on the ‘submit’ button, the service initiates the updateUserPassword request. The new password is then hashed with bcrypt library before updating the user’s password in the database. The service will then return a success response to the client to notify the user. A snackbar alert is then displayed to inform the user that the password has been changed successfully.


Client
Service
Database
changeUserPassword(username, newPassword)
updateUserPassword(req, res)
bcrypt.hash(newPassword, 10)
_updateUserPassword(username, hashedPassword)
handleChangePasswordSuccessOpenAlert()
Client
Service
Database

Sequence diagram when a user changes password

Deleting Account

A user can delete their account from the profile page. When the user service receives the request, it first invalidates the user’s JWT that is attached to the request’s header. Then, it perform 2 things in parallel: it deletes the user’s data from the user service’s database collection, and it also makes inter-microservice HTTP calls to the matching and collaboration services to delete any data associated with the user.



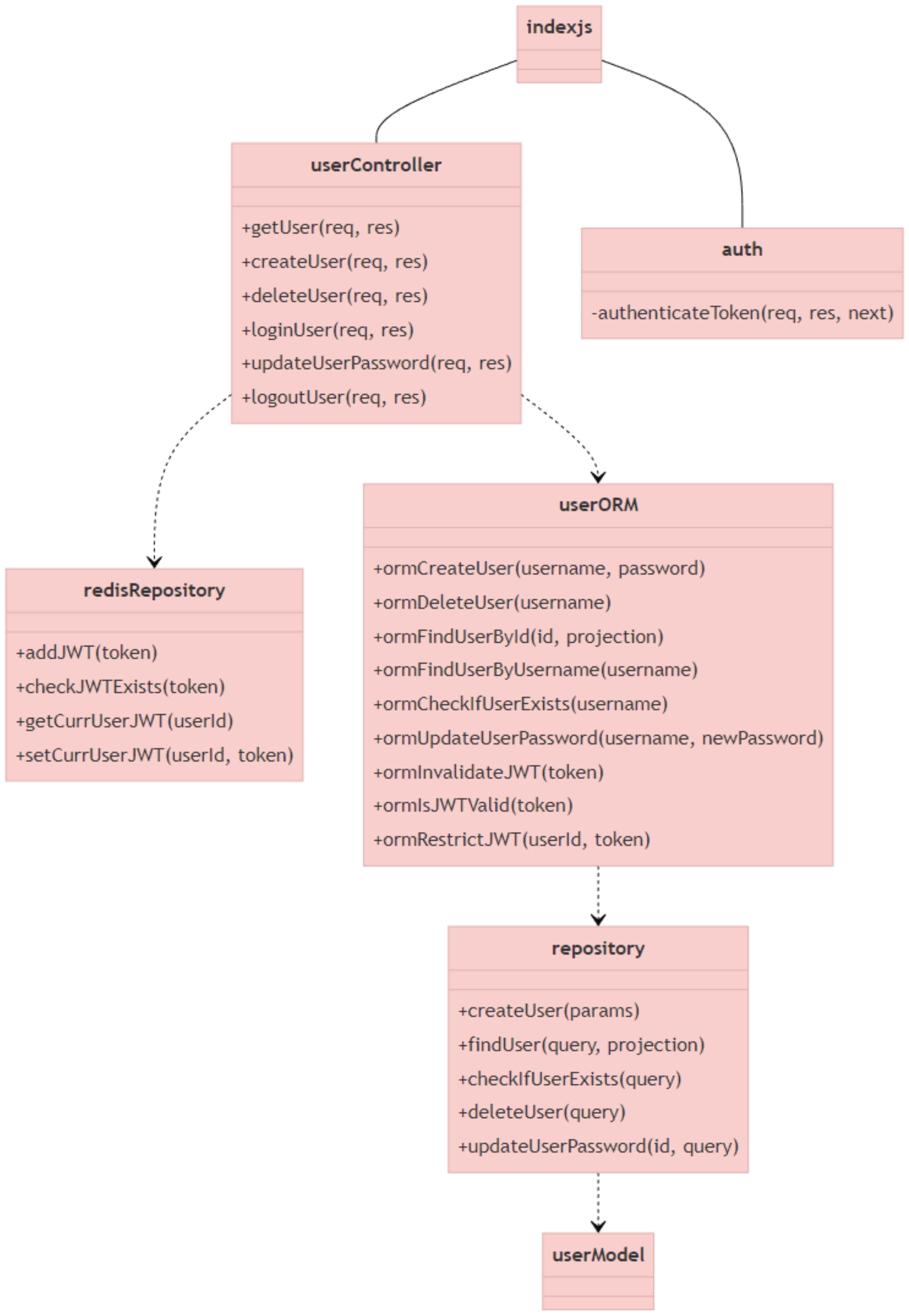
Sequence Diagram when a user account is deleted

Database Schema

Each user account information is stored in the MongoDB user collection. A user has the following schema:

```
username: {
    type: String,
    required: true,
    unique: true,
},
password: {
    type: String,
    required: true,
},
```

Service Components


indexjs
userController
+getUser(req, res)
+createUser(req, res)
+deleteUser(req, res)
+loginUser(req, res)
+updateUserPassword(req, res)
+logoutUser(req, res)
auth
-authenticateToken(req, res, next)
redisRepository
+addJWT(token)
+checkJWTExists(token)
+getCurrUserJWT(userId)
+setCurrUserJWT(userId, token)
userORM
+ormCreateUser(username, password)
+ormDeleteUser(username)
+ormFindUserById(id, projection)
+ormFindUserByUsername(username)
+ormCheckIfUserExists(username)
+ormUpdateUserPassword(username, newPassword)
+ormInvalidateJWT(token)
+ormIsJWTValid(token)
+ormRestrictJWT(userId, token)
repository
+createUser(params)
+findUser(query, projection)
+checkIfUserExists(query)
+deleteUser(query)
+updateUserPassword(id, query)
userModel

Class diagram for the user service

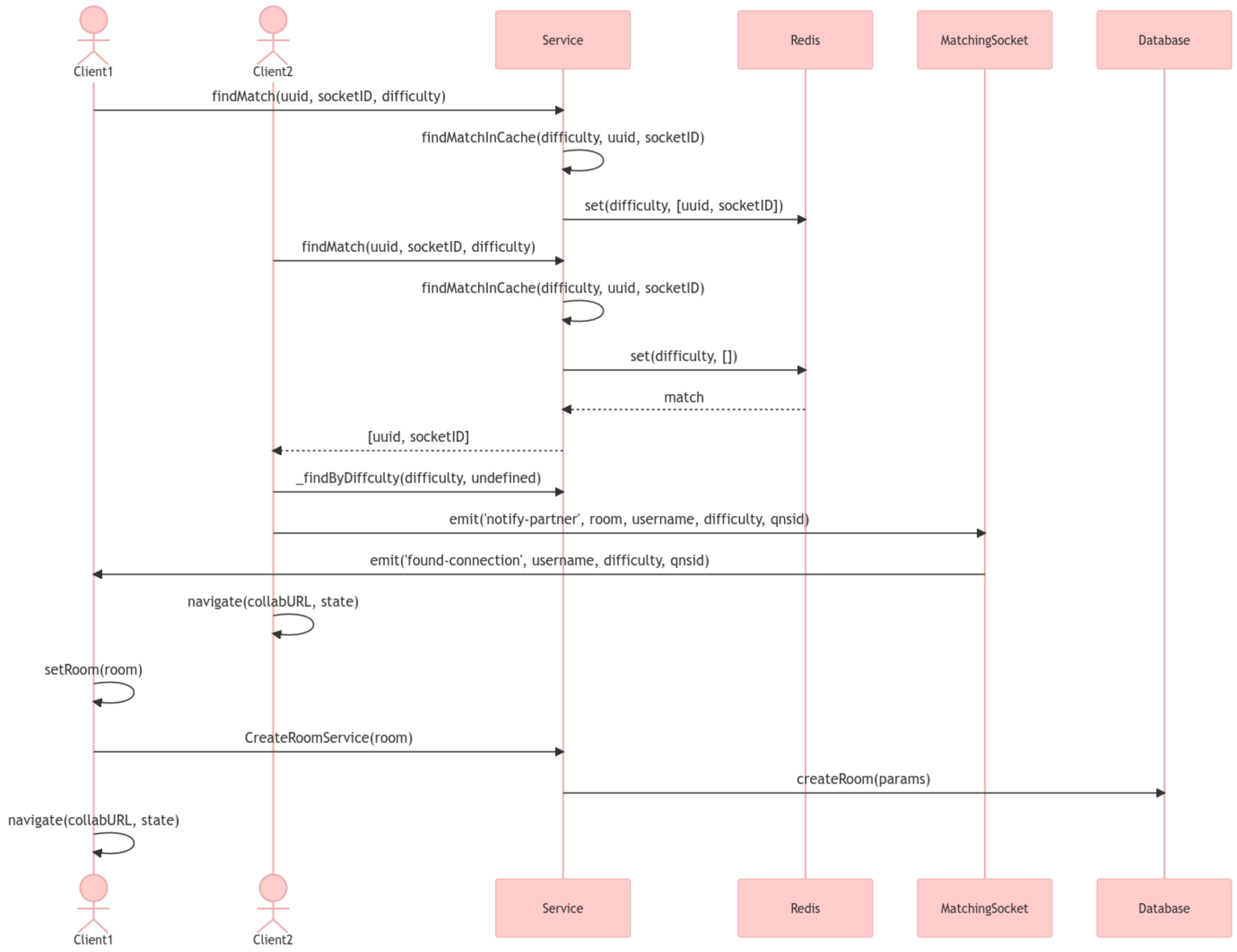
From a high-level perspective, the user service can be split into three overall layers, the routing layer, the controller layer, and the model layer. In our application, the index.js component is responsible for receiving and routing requests to controllers. It hands off control to the controllers, which are selectively called depending on the request received. Some controllers depend on and interact with the redisRepository component, which is responsible for Redis operations. Others depend on the ORM component, which encapsulates business logic related to database operations. The actual database operations (Mongoose calls) are then contained in the repository component, which naturally depends on a Mongoose schema.

Matching Service

The matching service is responsible for setting up a connection between two users to be matched based on the difficulty the user chooses in real-time. This service is responsible for the creation and storage of the critical information regarding the room of the two matched users. The service mainly uses Redis for the storage of the user who initiated the finding of a match, and socket.io to inform the user that a successful match has been found and established. Matching Service also provides HTTP REST endpoints which allows CRUD operations to be performed on the rooms stored in MongoDB. MongoDB is utilized to persist the data of the rooms, which enables users to rejoin a room after disconnecting.

Finding a Match

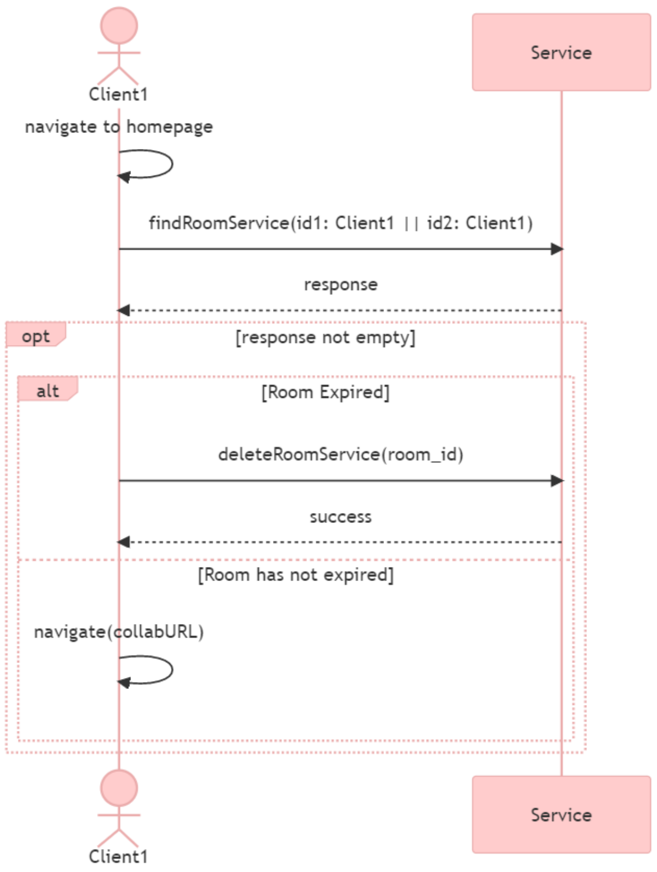
The matching service revolves around matching and establishing a connection between the two matched users. Both users would be redirected to the same room where they have the same room information and question.


Client1
Client2
Service
Redis
MatchingSocket
Database
findMatch(uuid, socketID, difficulty)
findMatchInCache(difficulty, uuid, socketID)
set(difficulty, [uuid, socketID])
findMatch(uuid, socketID, difficulty)
findMatchInCache(difficulty, uuid, socketID)
set(difficulty, [])
match
[uuid, socketID]
_findByDiffculty(difficulty, undefined)
emit('notify-partner', room, username, difficulty, qnsid)
emit('found-connection', username, difficulty, qnsid)
navigate(collabURL, state)
setRoom(room)
CreateRoomService(room)
createRoom(params)
navigate(collabURL, state)
Client1
Client2
Service
Redis
MatchingSocket
Database

Sequence diagram when a user tries to find a match

Rejoining an Existing Matched Room

When a user disconnects, such as when closing the browser tab or navigating to another page (e.g. the profile page), the user can rejoin a room. They can only do so if the room has not expired yet and the other user is still waiting in the room.



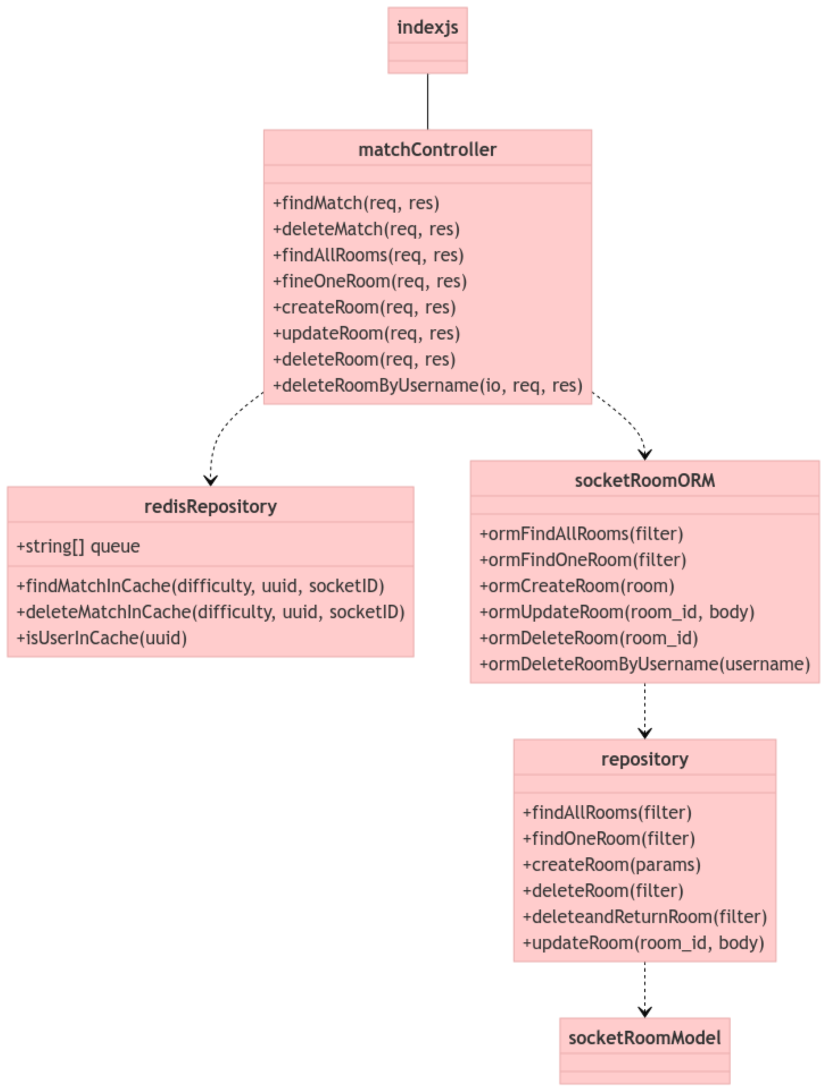
Sequence diagram when a user rejoins a matched room

Database Schema

The matching room is persisted in MongoDB so that the data can be retrieved when a client reconnects. A matching room has the following schema:

```
room_id: {
    type: String,
    required: true,
    unique: true,
},
qnsid: {
    type: Number,
    required: true,
},
difficulty: {
    type: String,
    required: true,
},
id1: {
    type: String,
    required: true,
    unique: true,
},
id2: {
    type: String,
    required: true,
    unique: true,
},
datetime: {
    type: Date,
    required: true,
    default: Date.now,
},
```

Service Components



Class diagram for the matching service

The matching service follows the same 3-layer structure as all our services. In particular, the redisRepository component here stores a hashmap containing three queues of different difficulties.

Alternative Design Decisions

At the initial stage of development, our team followed the rubrics FR 2.2 & FR 2.3 where we created a model for pending matches. The idea was to match users of the same difficulty using MongoDB. This was later replaced by Redis, which is an in-memory database. It performs much faster as compared to MongoDB. Persistence is not a concern while queuing users as if a redis node fails, the user would only need to re-queue. However, persistence of the room data is important, hence we chose to use MongoDB for said purpose instead.

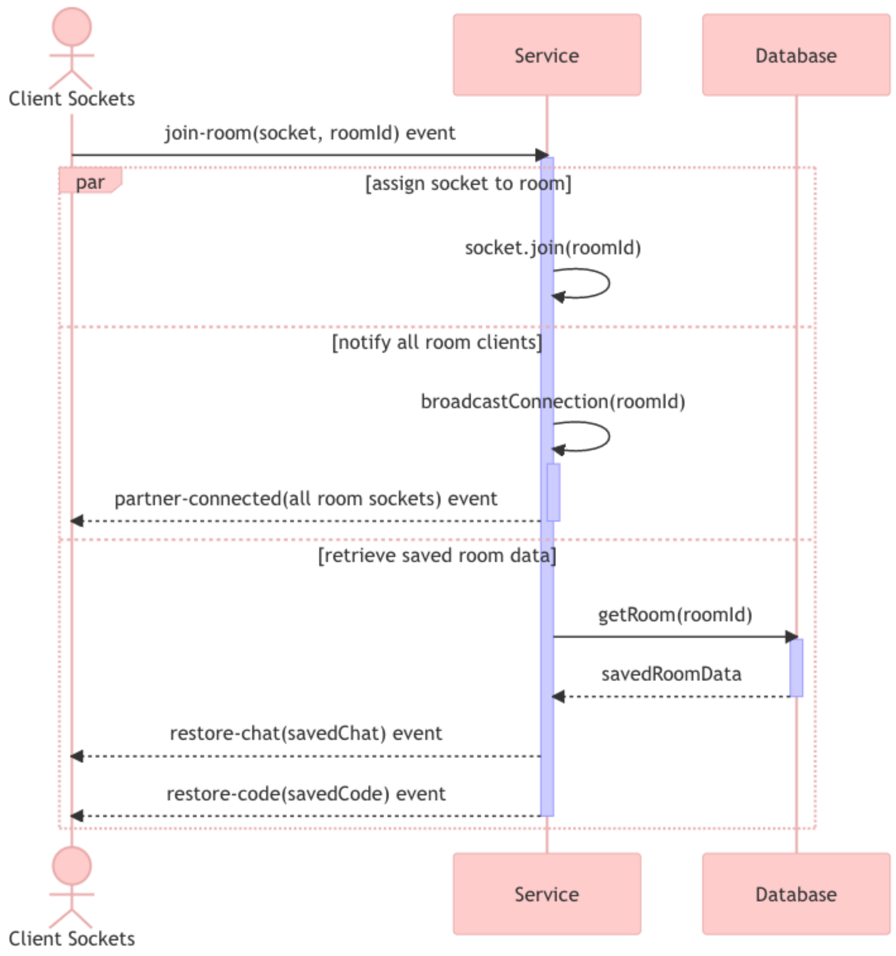
```
{
  "room_id": "sam",
  "difficulty": "hard",
  "id1": "sam",
  "id2": "", // empty fields would mean waiting for a match
  "id1_present": true,
  "id2_present": false,
   // if id2 is not empty and present is false, means user disconnect/left
  "datetime": "2022-10-02T02:02:42.625Z"
}
```

Collaboration Service

The collaboration service is responsible for letting two matched users collaborate in real-time. This includes supporting a live code editor and chat box, and also allows either user to change the question seen by both partners. The service is mainly a socket.io service, although it also supports a single HTTP REST endpoint which will be discussed towards the end of this section.

Joining a Collaboration Room

The collaboration service revolves around the main concept of 2 users being placed in a single room, which allows them to communicate with each other via WebSockets.


Client Sockets
Service
Database
join-room(socket, roomId) event
par
[assign socket to room]
socket.join(roomId)
[notify all room clients]
broadcastConnection(roomId)
partner-connected(all room sockets) event
[retrieve saved room data]
getRoom(roomId)
savedRoomData
restore-chat(savedChat) event
restore-code(savedCode) event
Service
Database
Client Sockets

Sequence diagram when a client joins a room

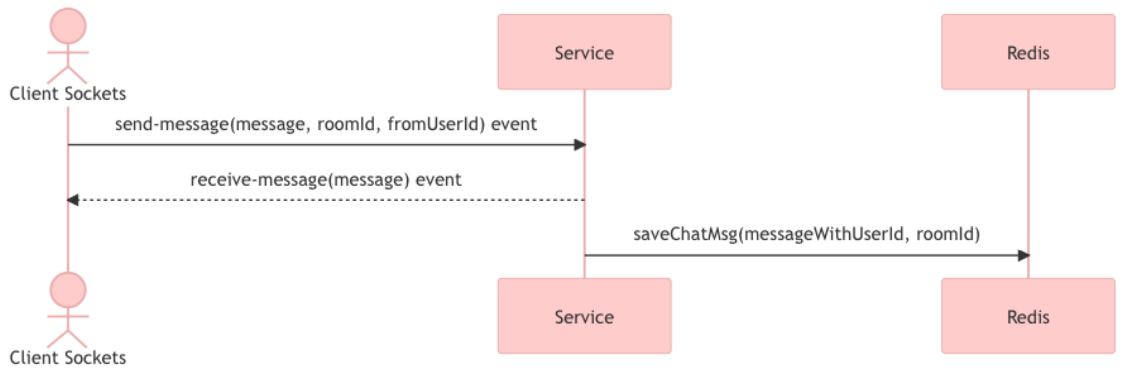
As the service is mainly a socket.io server, clients communicate with it via socket.io events. Three things happen in parallel when a client wants to join a collaboration room.

Firstly, the service assigns the requesting socket to a socket.io room with the roomId provided by the client.

Secondly, the service emits an event to all sockets in that same room, telling them that a new socket has joined. It also sends a list of the sockets in the event payload. This allows a client to check if it's the only one in the room or not.

Lastly, the service checks the database to see if any saved data for this room exists. This includes saved chat history/messages and saved code (from the shared code editor). It then emits 2 events back to the client, one for each type of data. This allows the client to restore the last-known state of the collaboration session, which is helpful if a user leaves the collaboration page or refreshes it accidentally.

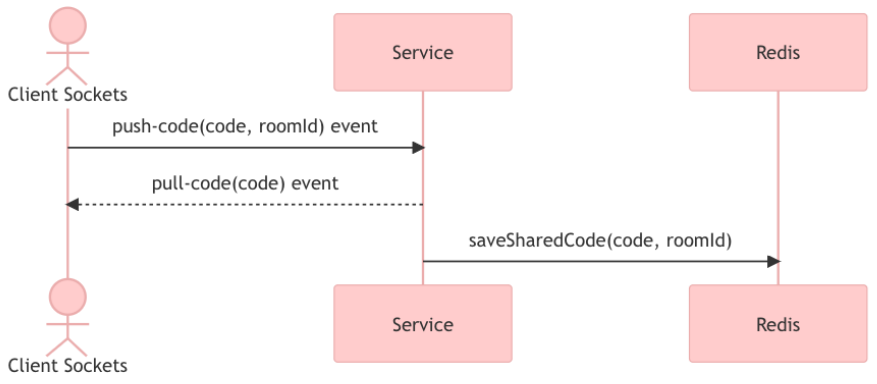
Sending a Chat Message


Client Sockets
Service
Redis
send-message(message, roomId, fromUserId) event
receive-message(message) event
saveChatMsg(messageWithUserId, roomId)
Client Sockets
Service
Redis

Sequence diagram when a chat message is sent by a user

To send a message, a client socket emits a ‘send-message’ event, which contains the message text, a room id, and the sender’s user id. The collaboration service then emits a ‘receive-message’ event containing the same message text, which allows the other partner in the room to receive and render the chat message. Then, the service serializes the message and the sender’s user id as a JSON object and saves it in Redis. This allows us to persist the chat history when needed later on. We chose to use Redis as a cache to prevent writing too frequently to the database.

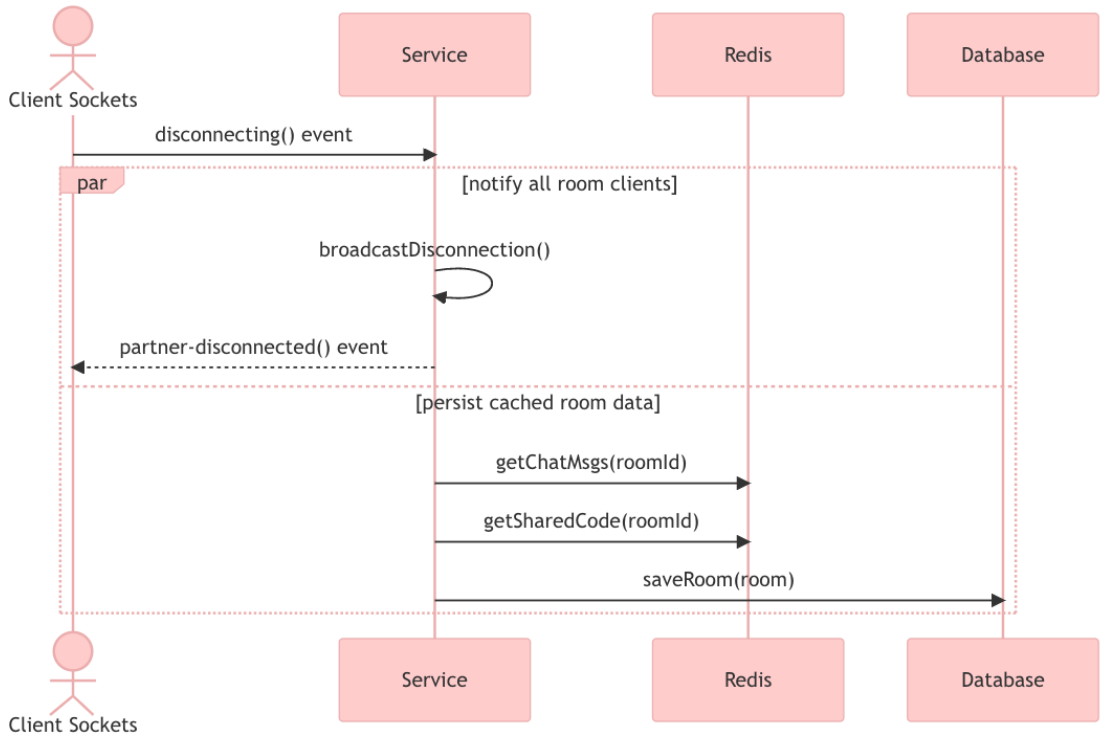
Typing Into the Shared Code Editor



Sequence diagram when a user types into the code editor

When either user types in the shared code editor, the client socket sends the code in a ‘push-code’ event to the collaboration service. The collaboration service then emits a ‘pull-code’ event containing this code, allowing the other partner to receive the new state and render accordingly. Just like what was done with chat messages, the service then saves the code in Redis so that it can be persisted later on. Here, the importance of using a cache is even more stark as ‘push-code’ events are emitted on every keystroke and thus in very high frequencies. It would be unwise to write to the database on every ‘push-code’ event received by the server.

Disconnecting

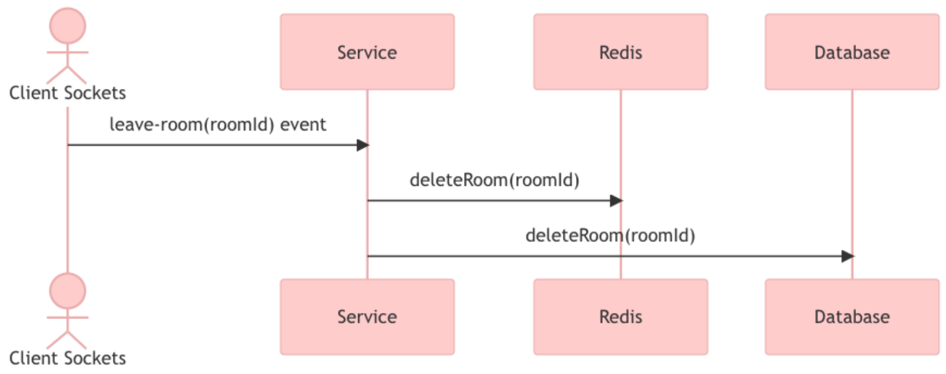


Sequence diagram when a user disconnects

A user’s socket may disconnect for a number of reasons, including (but not restricted to) closing the tab or refreshing the page accidentally. When this occurs, the socket emits a ‘disconnecting’ event to the collaboration service. The service first notifies the partner’s socket via a ‘partner-disconnected’ event.

This allows clients to decide what happens when a user is alone in a room (we choose to show a dialog and prevent the user from changing the state of the room, which might lead to messy data consistency issues). At the same time, the service flushes the room’s data cached in Redis to the database for persistence. When the user reconnects later, the service can then restore the state of the room for the client, as seen in the section ‘Joining a collaboration room’.

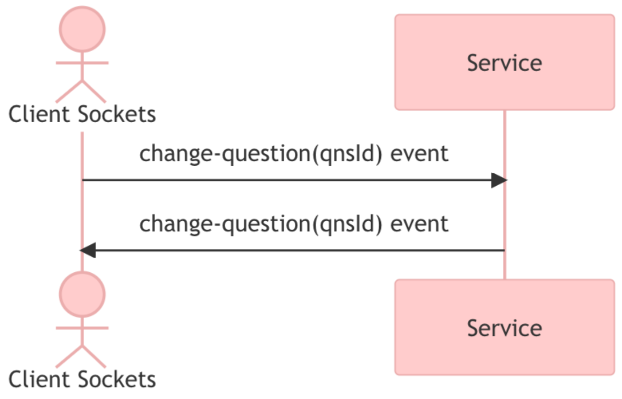
Leaving a Room



Sequence diagram when a user leaves the room

When a user clicks on the ‘leave room’ button on the front end, a ‘leave-room’ event is emitted from the user’s socket to the collaboration service. The service then proceeds to delete the room’s data from both the Redis cache and the database.

Changing Questions



Sequence diagram when a user changes the room’s question

When a user changes the question in a room by getting a new question from the question service, the user’s socket emits a ‘change-question’ event with the new question’s ID. The collaboration service then broadcasts this same event to the user’s partner in order to provide the partner with the new question ID.

HTTP Endpoint to Delete Rooms

While the collaboration service is mostly a socket.io server, it is also an Express.js HTTP server with a single REST endpoint. This endpoint allows a client to make an HTTP DELETE request to delete a room. This behavior is the same as when a user leaves a room, so no sequence will be provided here. This endpoint allows PeerPrep's user service to delete a user's collaboration data when a user account is deleted via an inter-microservice HTTP request-reply call.

Database Schemas

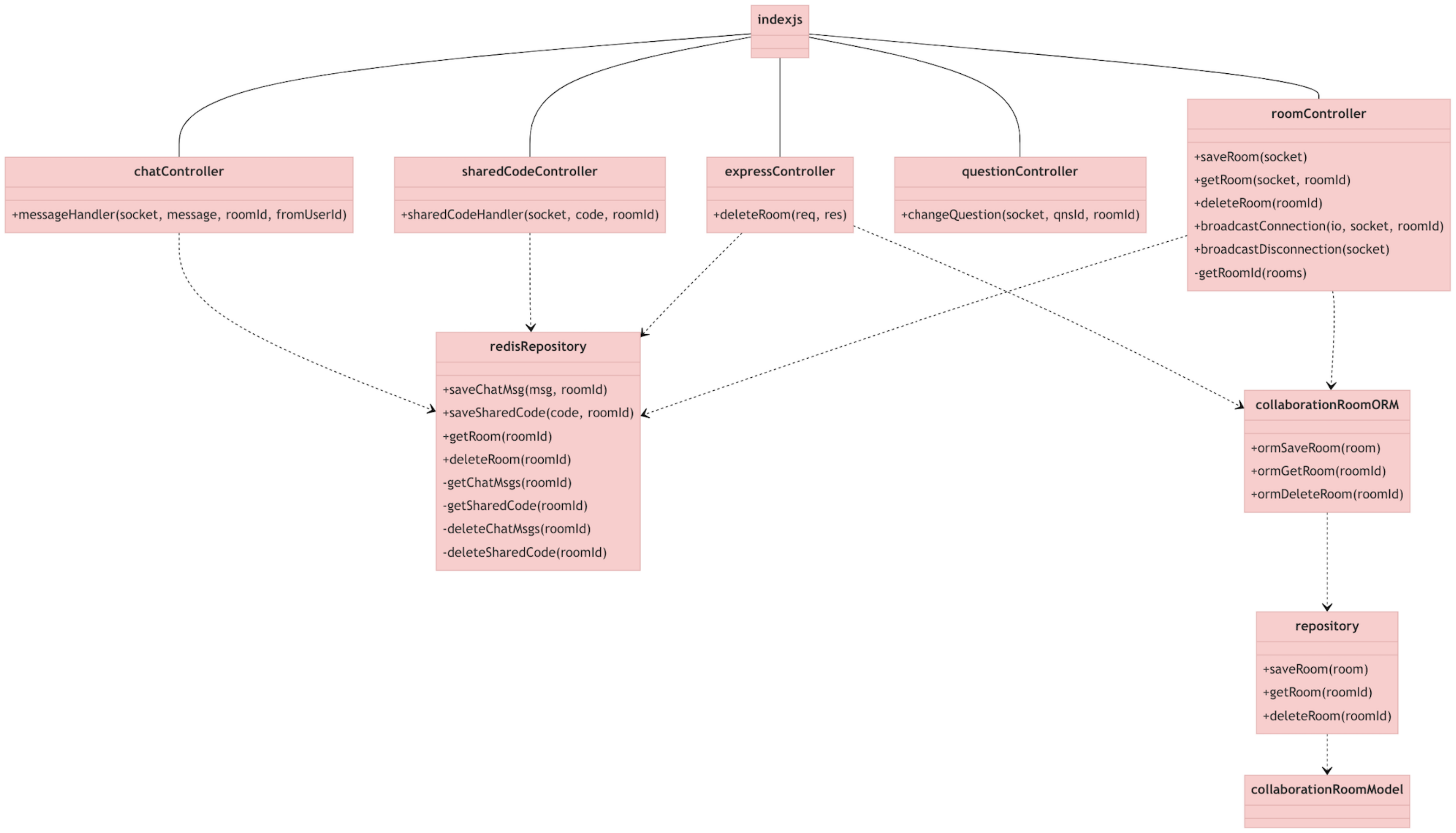
Each collaboration room is persisted in MongoDB so that its data can be retrieved when a client requires it. A collaboration room has the following MongoDB/Mongoose schema:

```
// ChatMsgModelSchema
{
  content: {
    type: String,
    required: true,
  },
  from: {
    type: mongoose.Schema.ObjectId,
    required: true,
  }
}

// CollaborationRoomModelSchema
{
  roomId: {
    type: String,
    required: true,
    unique: true,
  },
  code: {
    type: String,
  },
  chat: [ChatMsgModelSchema]
}
```

The main data saved in the database are the chat messages and shared code. Shared code is simply saved as a plain text string, and it should be noted that this string is never evaluated or executed without sanitation for security reasons. Chat messages are stored as an array. Each chat message contains the text content of the message and also the MongoDB user ID of the sender. The same data format is used for our Redis cache.

Service Components



Class diagram for the collaboration service

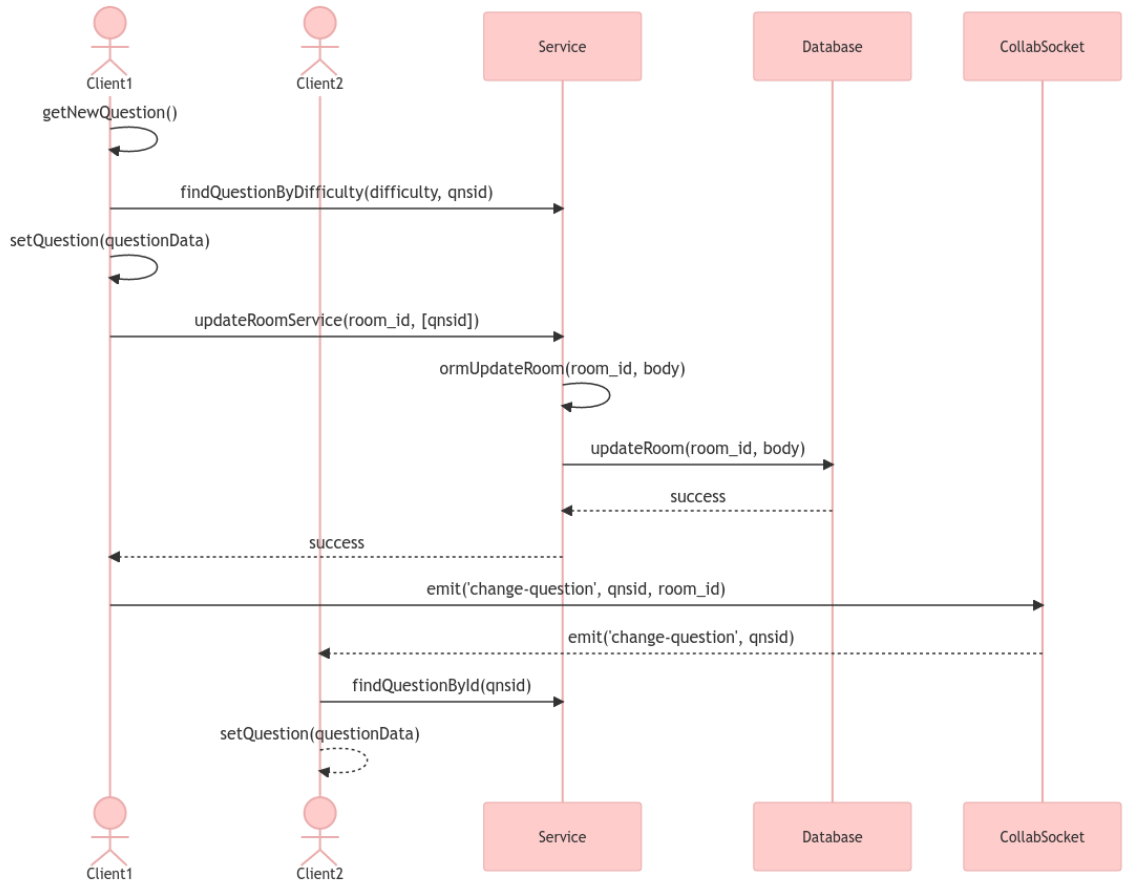
The collboration service follows the same 3-layer structure as all our services. In particular, the redisRepository component stores cached room data temporarily before it is persisted to the database.

Question Service

The question service provides users with a randomized question of the selected difficulty they have chosen via an HTTP call.

Changing Questions

When a user requests to change questions, the user will receive a new randomized question while extending their session time.


Client1
Client2
Service
Database
CollabSocket
getNewQuestion()
findQuestionByDifficulty(difficulty, qnsid)
setQuestion(questionData)
updateRoomService(room_id, [qnsid])
ormUpdateRoom(room_id, body)
updateRoom(room_id, body)
success
success
emit('change-question', qnsid, room_id)
emit('change-question', qnsid)
findQuestionById(qnsid)
setQuestion(questionData)

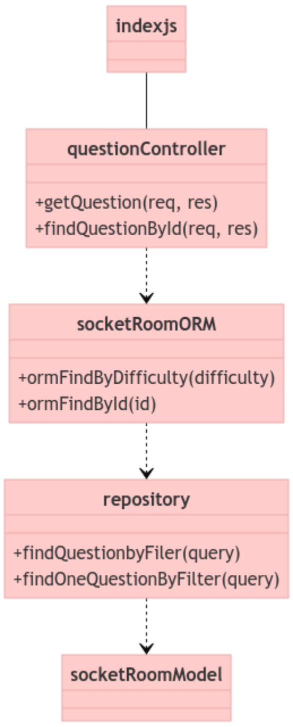
Sequence diagram when a user changes questions

Database Schemas

```
title: {
    type: String,
    required: true,
    unique: true,
},
qnsid: {
    type: Number,
    required: true,
    unique: true,
},
difficulty: {
    type: String,
    required: true,
},
description: {
    type: String,
    required: true,
},
ex_1_input: {
    type: String,
},
ex_1_output: {
    type: String,
},
ex_1_explanation: {
    type: String,
},
ex_2_input: {
    type: String,
},
ex_2_output: {
    type: String,
},
ex_2_explanation: {
    type: String,
},
ex_3_input: {
    type: String,
},
ex_3_output: {
    type: String,
},
ex_3_explanation: {
    type: String,
},
```

Our schema follows the format of a Leetcode question, where we each question can have at most 3 examples.

Service Components



indexjs
questionController
+getQuestion(req, res)
+findQuestionById(req, res)
socketRoomORM
+ormFindByDifficulty(difficulty)
+ormFindById(id)
repository
+findQuestionbyFiler(query)
+findOneQuestionByFilter(query)
socketRoomModel

Class diagram for the question service

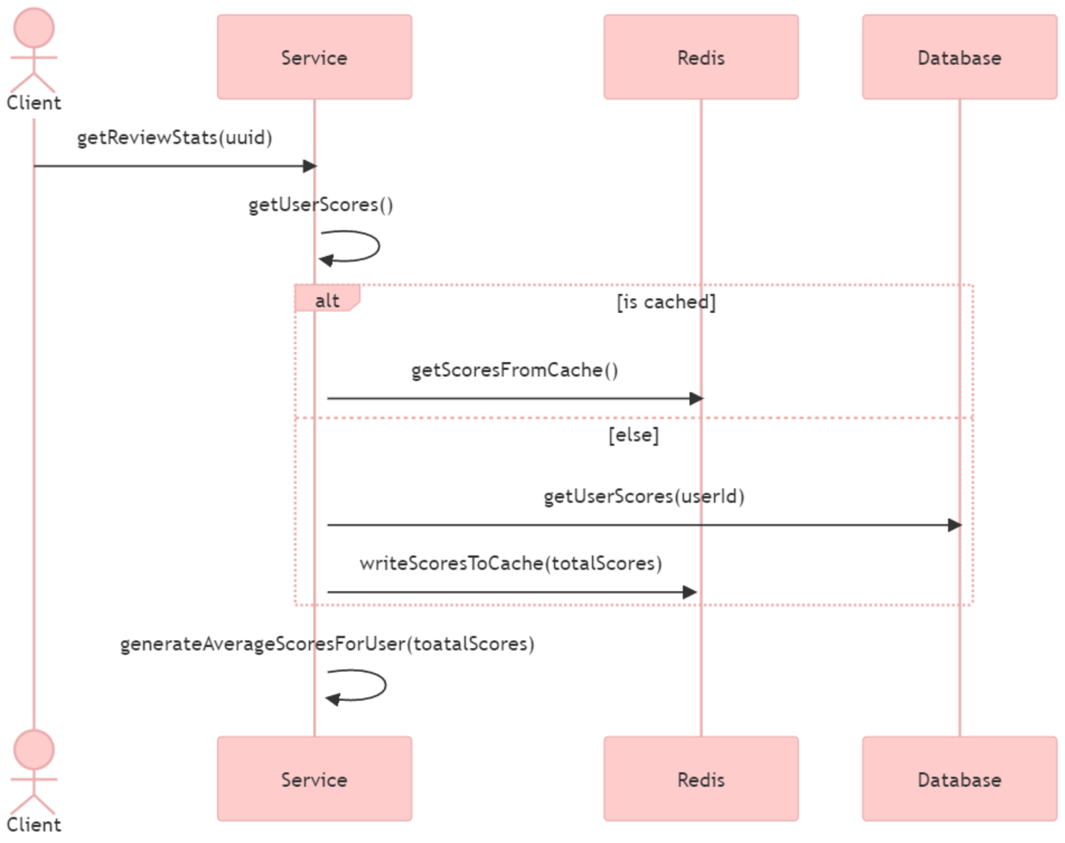
The question service follows the same 3-layer structure as all our services.

Review Service

The review service provides users with reviews that they have received from their peers. Currently, the architecture of the review service utilizes its own collection in MongoDB and uses Redis for caching scores to reduce the total number of HTTP calls.

Viewing Ratings

Each time the user logs into their account and views their profile, we query the entire collection and filter by the user's uuid in order to aggregate all the reviews from said user. We then cache this result into Redis, and subsequent calls to get review data will be read from the cache. This is done in order to save the amount of database reads that we perform in total, as they are relatively expensive as compared to reading from the cache. Any new reviews that the user receives while being logged in will update both the cache and database. As such, we only query the database at most once per user login.



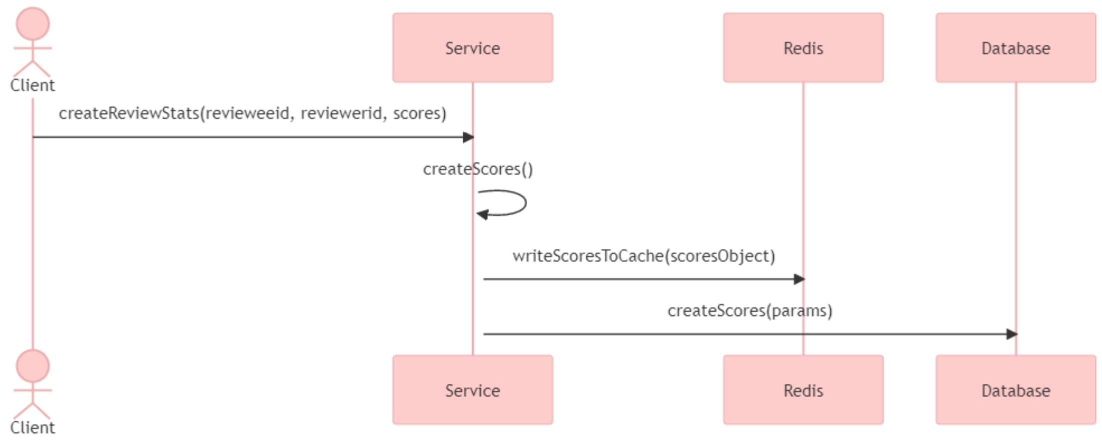
Sequence diagram when a user fetches his/her own review ratings

Furthermore, our team acknowledges that having to filter through the entire collection for records regarding a user may be a costly operation. Thus, in order to improve performance, we have decided to index our field revieweeid in order to speed up queries.

Leaving a Review

As mentioned above, when a new review is given to a user, we add a new entry to the database as well as update the redis cache. Our cache stores the total scores that they have received in each field as well as the total number of reviews they have. However, our frontend displays the user's average score. Thus, to get the average score, we have to perform averaging on the backend and return the relevant scores.

We have decided to store the total scores and not the average scores in the cache as it simplifies the process when a new review is given. With our current implementation, we can simply increment the new scores to the current total and recalculate the average. If we stored average scores, we would have to deal with decimals. This is undesirable due to rounding issues when recalculating the average.



Sequence diagram when a user leaves a review for his/her partner

In order to support extensibility for new criteria, we collate the fields for reviews under an array and pass this array along to functions for processing. Hence, removing or adding a criteria is simple. To do so, one would just have to edit the array to add or remove fields. This design thus supports the Open Closed Principle.

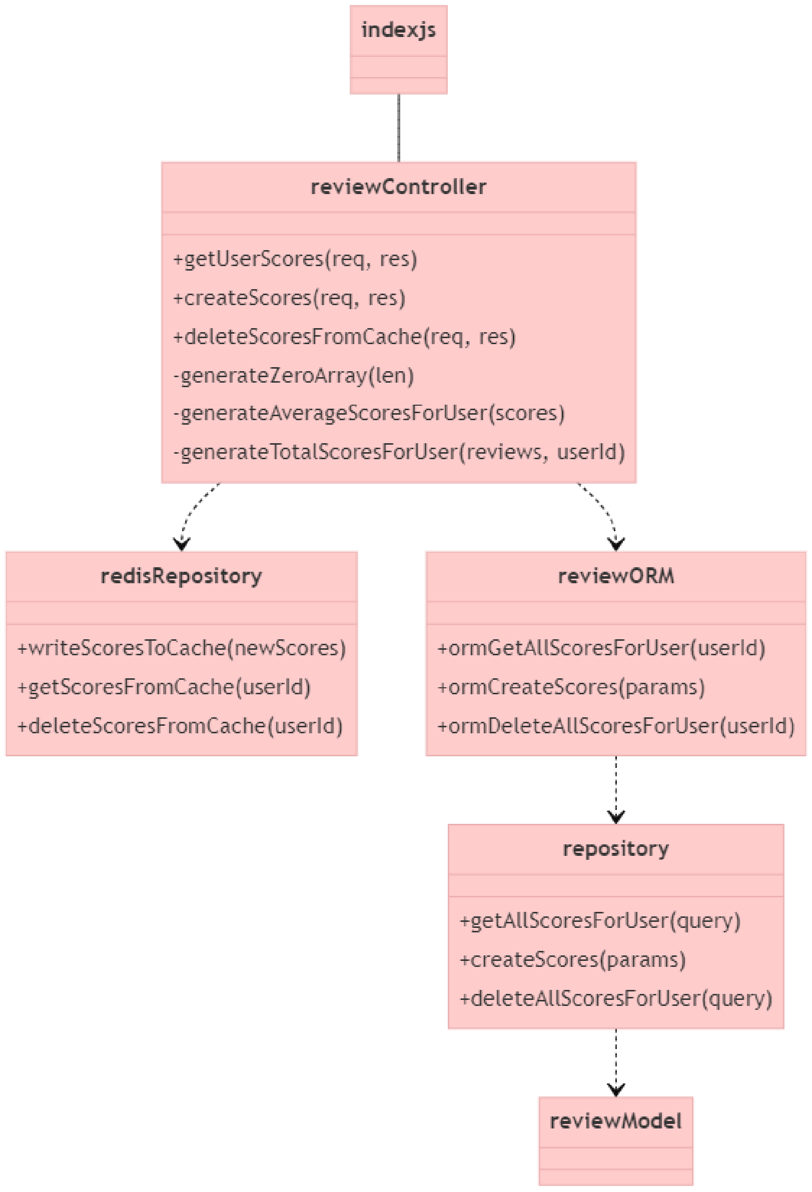
```
export const FIELDS = [
 'codeCorrectness',
 'codeDesign',
 'codeStyle',
 'communicationStyle',
 'timeManagement',
]
```

Database Schema

```
revieweeid: {
    type: ObjectId,
    required: true,
    index: true,
},
reviewerid: {
    type: ObjectId,
    required: true,
},
scores: {
    type: [Number],
    required: true,
}
```

As mentioned above, reviweeid is indexed as we will often need to lookup based on this attribute.

Service Components


indexjs
reviewController
+getUserScores(req, res)
+createScores(req, res)
+deleteScoresFromCache(req, res)
-generateZeroArray(len)
-generateAverageScoresForUser(scores)
-generateTotalScoresForUser(reviews, userId)
redisRepository
+writeScoresToCache(newScores)
+getScoresFromCache(userId)
+deleteScoresFromCache(userId)
reviewORM
+ormGetAllScoresForUser(userId)
+ormCreateScores(params)
+ormDeleteAllScoresForUser(userId)
repository
+getAllScoresForUser(query)
+createScores(params)
+deleteAllScoresForUser(query)
reviewModel

Class diagram for the review service

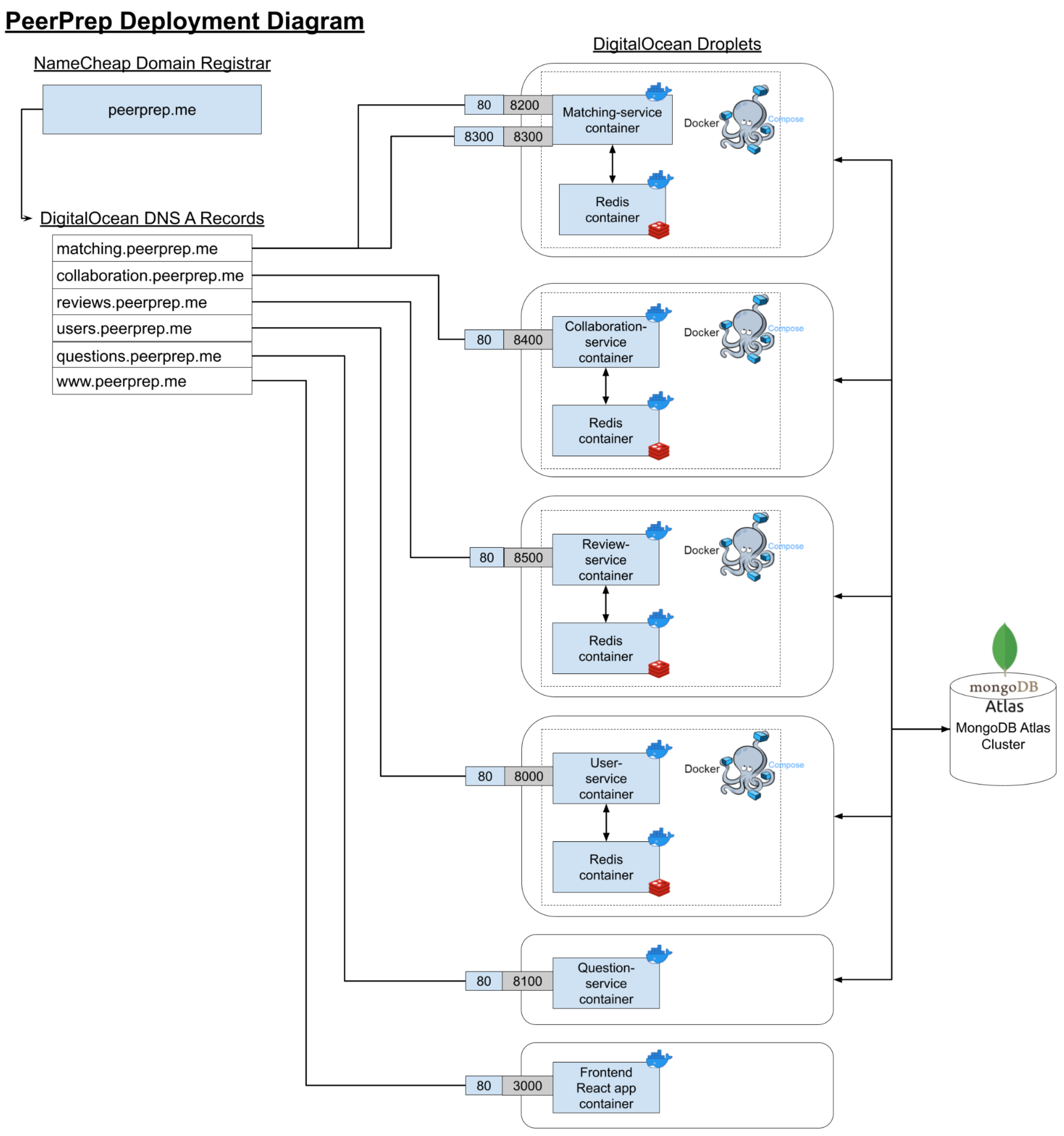
The review service follows the same 3-layer structure as all our services.

Alternative Design Decisions

Making reviews an attribute of a user in the User collection. We could potentially store the total scores or average scores of each user as an attribute within each User. This way, we wouldn't need to support an additional microservice, and can easily query for the scores by just reading off the attribute.

However, this method has some drawbacks. In the case where we want to allow users to see the past reviews they have given, we would have to store all the reviews in a collection. If we store these reviews in the User collection, it violates the single responsibility principle. Our user microservice would be coupled to reviews directly and contain additional state that it should not keep track of.

Deployment Details



PeerPrep's deployment diagram

Peerprep's microservices (and the front-end) are deployed on separate DigitalOcean Droplets. This makes it easy to scale each service as required and also prevents catastrophic failure when a single Droplet running all the services goes down.

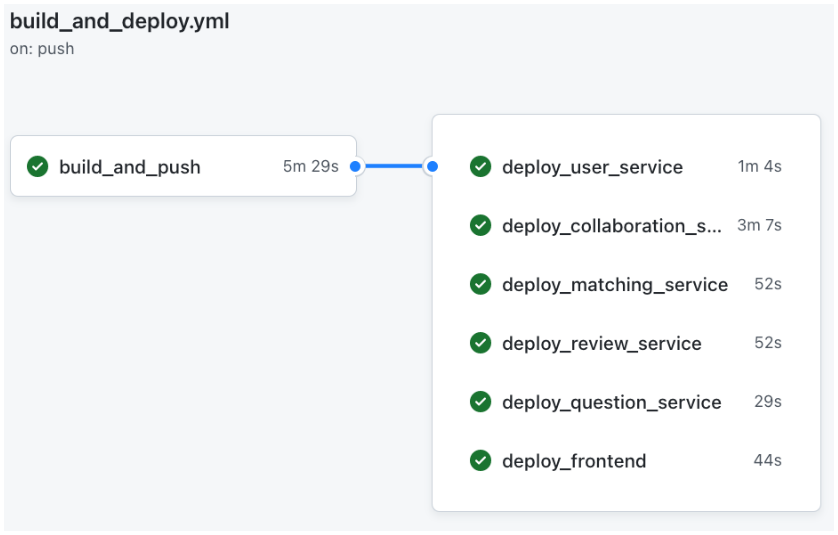
We have purchased the peerprep.me domain from NameCheap, which is a registrar. We then use DigitalOcean to manage our DNS records by pointing NameCheap to DigitalOcean's nameservers. By creating an A record for each microservice, each microservice resides in its own subdomain, which can be seen in the diagram above. This makes it semantically easy to access the microservices.

All our services are dockerized and each Droplet comes with Docker installed. Certain services (e.g. user service) utilize a Redis instance. For these services, the service and its own Redis server are deployed on a single Droplet in a Docker Compose stack. They then communicate within the Docker network set up by the stack. For other services that do not require Redis, they are simply run as a Docker container on the host Droplet. You can also see the host port to container port mappings in the deployment diagram above.

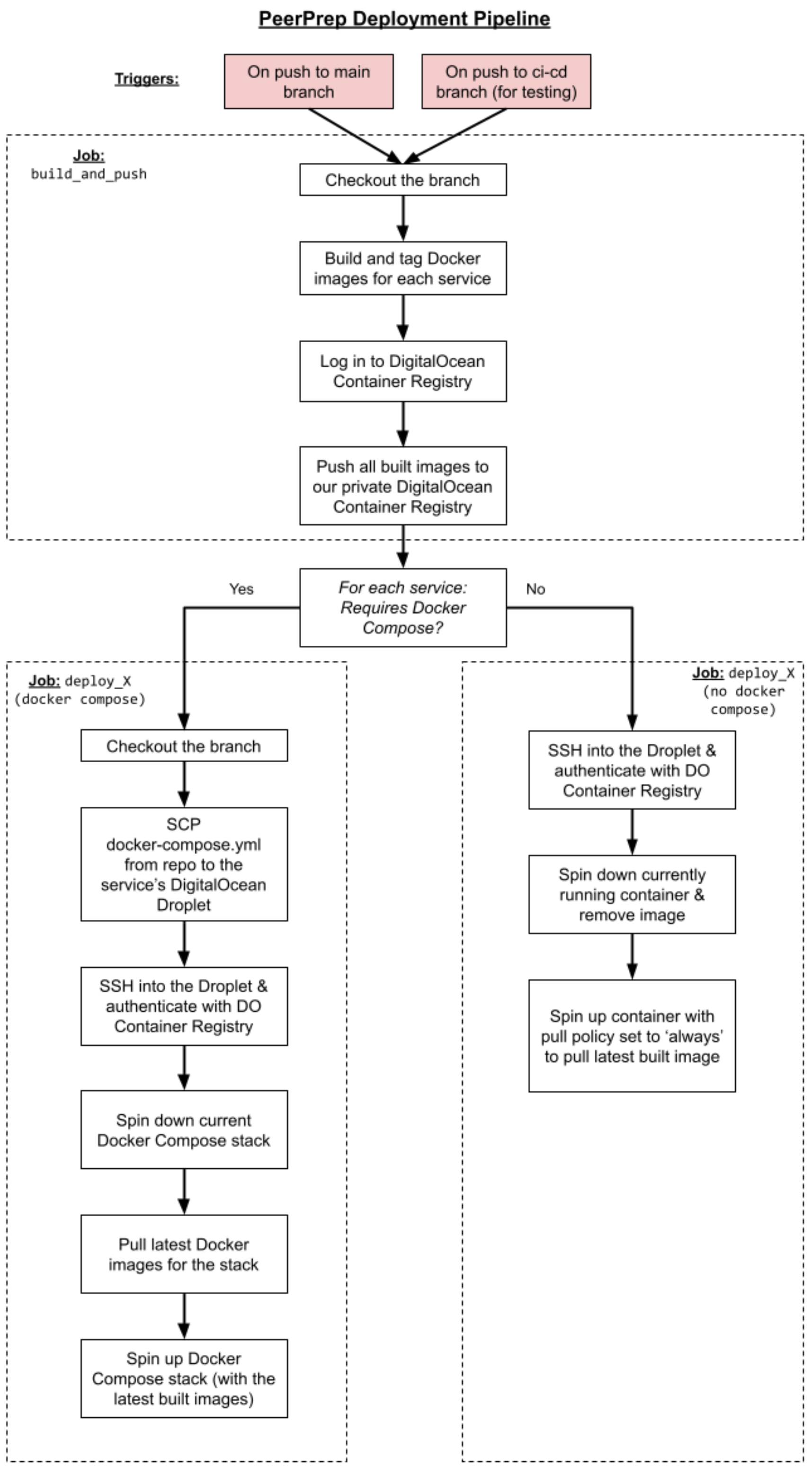
DigitalOcean was chosen as our cloud provider due to its ease of use. Providers such as AWS generally have a steeper learning curve, require more configuration (IAM, VPC, etc.), and are more complex to maintain. We chose to use Droplets to host our services instead of DigitalOcean's App Platform due to the flexibility provided. A Droplet is simply a VM and can do anything an ubuntu machine can. For instance, since we wanted to deploy Docker Compose stacks, we had to use Droplets. Beyond DigitalOcean, we chose NameCheap as our domain registrar as it provided a free domain under the GitHub Student Pack.

Lastly, all our services share a single MongoDB database cluster hosted on MongoDB Atlas. Each service generally works with its own collection/set of collections within the database.

Deployment Pipeline


build_and_deploy.yml
on: push
build_and_push 5m 29s
deploy_user_service 1m 4s
deploy_collaboration_s... 3m 7s
deploy_matching_service 52s
deploy_review_service 52s
deploy_question_service 29s
deploy_frontend 44s

A single run of PeerPrep's continuous delivery & deployment pipeline


PeerPrep Deployment Pipeline
Triggers:
On push to main branch
On push to ci-cd branch (for testing)
Job: build_and_push
Checkout the branch
Build and tag Docker images for each service
Log in to DigitalOcean Container Registry
Push all built images to our private DigitalOcean Container Registry
For each service: Requires Docker Compose?
Yes
No
Job: deploy_X (docker compose)
Checkout the branch
SCP docker-compose.yml from repo to the service's DigitalOcean Droplet
SSH into the Droplet & authenticate with DO Container Registry
Spin down current Docker Compose stack
Pull latest Docker images for the stack
Spin up Docker Compose stack (with the latest built images)
Job: deploy_X (no docker compose)
SSH into the Droplet & authenticate with DO Container Registry
Spin down currently running container & remove image
Spin up container with pull policy set to 'always' to pull latest built image

PeerPrep's continuous deployment pipeline

Our deployment pipeline builds and deploys all our services when a commit is pushed to the main branch (and the ci-cd branch for pipeline testing purposes). Ideally, microservices should reside in their own repositories so that they can be built and deployed independently. However, due to the module's requirements, all our services reside in a monorepo. Since each push to the main branch may contain changes to any number of files and services, it is non-trivial to detect changes to individual services by analyzing Git commits and selectively deploying them. As such, we build and deploy all our services regardless of the changes made.

Our team uses Github Actions as our CI/CD runner. This provides a number of benefits, including low/no cost (free 2000 minutes/month) and ease of management - it is convenient to view job runs and manage secrets all without leaving the GitHub repository page.

There are two main stages to our pipeline. Firstly, a GitHub Actions job is responsible for building the services. It starts off by checking out the current branch of the job, accessing the Dockerfiles in each of the service directories, and building them. It then authenticates the job runner with DigitalOcean using our GitHub repository secrets and pushes all the built images to our private repository on the DigitalOcean Container Registry.

Next, every service gets deployed in its own job and these jobs are run simultaneously. As seen from the deployment diagram in the previous subsection, some services require Redis. For these services, the GitHub Actions job first checks out the current branch and uses scp (secure copy protocol) to transfer the service's docker-compose.yml file to its Droplet. This allows us to update the Docker Compose configuration at any time and makes sure that the Droplet uses the latest configuration. The job then uses SSH to access the Droplet and executes commands: it first spins down the Docker Compose stack that's currently running on the Droplet, then pulls the latest images from the DigitalOcean Container Registry (which were built and pushed in the previous job), and finally spins up the stack again.

As for services that do not require Redis (or any other image besides the service itself), the job simply uses SSH to access the Droplet, stops the Docker container currently running, and starts it again with a "docker run" command that also pulls the latest image.

Other Significant Design Considerations

Using HTTP Request-Reply Pattern between Microservices

As of the current state of PeerPrep, communication between microservices is kept to a minimum. There are only a select number of scenarios that require them:

1. When a user deletes his/her account, the user service orchestrates cleanup operations required in other services. Specifically, the user service communicates with the matching service to delete all associated "match rooms", and the collaboration service to delete all associated "collaboration rooms".

2. The review service caches aggregate statistics of a user in Redis so that expensive aggregation database queries are kept to a minimum. However, we've decided that a user's cached data should be invalidated every time he/she logs out. As such, when the user service logs a user out, it also communicates with the review service to invalidate the latter's cache for that user.

The team took several points into consideration before deciding to use a synchronous HTTP request-reply pattern for the above 2 scenarios:

- The operations are not time/performance-sensitive. When an account is deleted, there is simply no rush to clean things up. And when a user logs out, there is also no critical deadline to meet when clearing the cached statistics. As such, the team felt that a synchronous request-reply protocol would be sufficient. The team also ruled out gRPC as its performance gains are superfluous; using gRPC would introduce unwarranted complexity in all the services involved, which can lead to greater developer overhead to learn, set up, and maintain the system.
- The team does not foresee these operations occurring in large volumes. It is realistic to expect that account deletions do not happen much and that a user logs out only once per session. As such, while the team initially considered a message-passing solution, we felt that such a solution would be over the top and underutilized, not to mention the overhead of adding complexity and cost to manage a message queue such as AWS SQS.
- All in all, we believe strongly in effective iteration and to avoid the mistake of over-optimizing too early in a project. As the project evolves and the requirements for inter-service communication change, the team can always revisit this design choice again.

Using Redis as a Cache

PeerPrep utilizes Redis as a cache in a number of services, as discussed above. In general, the team uses Redis as a cache over simple in-application heap storage due to the following reasons:

- Redis can be more performant. Garbage-collected languages (including Javascript) do not deal well with large heaps. Application performance can be adversely impacted if the garbage collector has to traverse a large heap during its "stop the world" operations.
- Redis provides useful turnkey features such as TTL and persistence strategies. By storing data with in-application heap memory, such features would be non-trivial to implement and use.
- If required in the future, it is easy to use a managed, hosted, and replicated Redis server with high resiliency that multiple services can access. Switching from the local Redis server on a service's Droplet to a remote one is simply a matter of changing a URI. On the other hand, this would not be possible using in-application memory.

Software Patterns

Chain of Responsibility

Express.js readily lends itself to the Chain of Responsibility design pattern. The framework supports router-level middleware, which allows us to define multiple handlers sequentially for a given route. All these handler functions have the same signature and can be seen as implementations of a common handler interface. When a request comes in and is matched to a route, it gets passed from one handler to the next. Each handler can read and write from the request and response objects, choose when to hand control to the next handler, or end the request-response flow.

In the user service, we make use of this pattern to implement our authentication checks in a clean and reusable manner. We have an authentication middleware which is responsible for extracting the authorization header from a request and verifying the JWT. If the JWT is invalid or has been blacklisted, the middleware immediately terminates the handler chain by returning a 401 or 403 response. Otherwise, it parses the payload of the JWT, attaches it the request object, and hands control over to the next handler. If we need to change our authentication logic in the future, modifications would be isolated to a single handler.

We use this middleware in all routes that require authentication checks:

```
router.get('/', authenticateToken, getUser)
router.post('/', createUser)
router.put('/', authenticateToken, updateUserPassword)
router.delete('/', authenticateToken, deleteUser)
router.post('/login', loginUser)
router.post('/logout', authenticateToken, logoutUser)
```

Facade

In general, we try to maintain facades between higher-level components and lower-level components/libraries. This prevents higher-level components from being too entangled with specificities, such as the exact function calls and configurations required to perform a task. For instance, on the frontend, API calls to the user service are made through a single file that acts as a facade. This facade contains methods that make the actual HTTP calls using Axios, and also handles specifics such as cookies, headers, authentication checks and redirects. Without this facade, all React components that need to query the user service would be littered with these low-level details. If we were to change a single detail, for instance, a cookie implementation, we would have to modify all these files.

The code snippet below is an example of a low-level detail in the facade that is hidden from React components. The facade contains a custom Axios instance that intercepts outgoing requests, reads cookies, and adds an authorization header. This prevents React components from being bogged down with the specifics needed to make an authenticated and authorized HTTP request.

```
axiosWithAuth.interceptors.request.use(
 (config) => {
   return {
     ...config,
     ...getAuthHeader(),
   }
 },
 (error) => Promise.reject(error)
)
```

Unit Testing

Our team has employed several testing libraries to perform unit testing on a few of our API endpoints. In this context, a unit is a single HTTP endpoint. This ensures that each individual endpoint works correctly as expected and allows us to catch bugs early on, leading to reliability in our application. It also makes refactoring easy and thus improves maintainability since developers can refactor single endpoints with confidence that any breaking changes can be immediately caught. On top of this, we ensure that there is a mix of positive and negative test cases to make our unit tests more robust and comprehensive.

We use the following libraries to craft and run our unit tests: mocha, chai and chai-http.

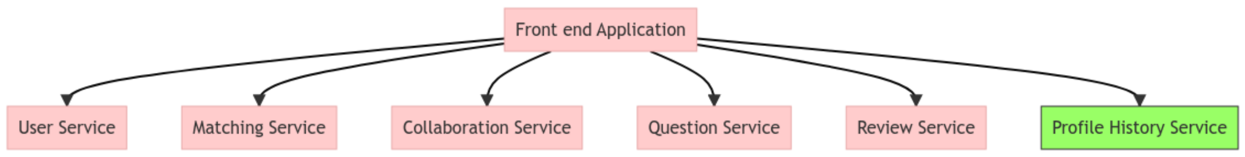
Future Plans for Improvement

Detailed Information on Interview History

Currently, our application does not store user's past interviews sessions or code.

To provide more benefits for our registered users, we can enable a history feature to allow them to better review their past performance and have better tools to track their own progress as well. This can be done by having a component that will expand into the history page. This page would show the submitted code and details such as the time taken to complete and timestamp with the code snapshots to show how the development of the code is evolving from time to time. Furthermore, tracking past questions can be extended to allow the user to be able to opt out from previously attempted questions. This will ensure users only view new questions.

To achieve this, we can add on a microservice called the Profile History Service to handle the user interview history functionalities. This microservice should also be scalable with increased usage, so we hope to support this feature with the ability to sort and filter on the Profile History Service API endpoint.

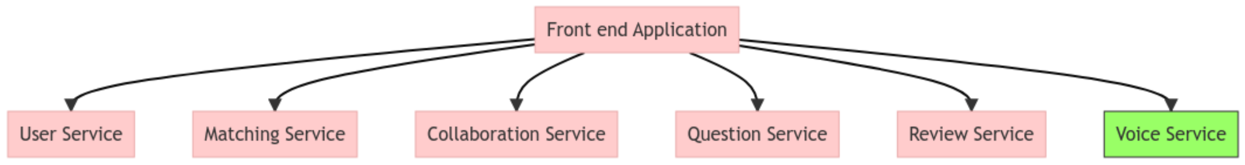


Possible architecture to support the interview histories

Voice Chat Functionality

While the chat functionality does the job of allowing the users to communicate, we believe that adding in a Voice Chat functionality would improve interaction and spark more fruitful and insightful discussion between the users as they are able to communicate more spontaneously and they do not have to keep switching between the code and the chat components which would improve the overall user experience as well.

The Voice Chat Service would be an additional microservice that helps to establish the connection between the two users and ensure that both parties have a smooth audio quality.



Possible architecture to support voice chats

Review Comments

Currently, users are allowed to review each other by giving a 1-5 star rating for each of our pre-chosen attributes (e.g. code quality). In the future, we believe that allowing users to leave text comments in a review will help them learn better. A user can then view a list of his/her reviews, which consists of rating scores and comments, on the profile page. This would involve extending the current review microservice to handle textual comments.

Full-fledged IDE & Interpreter

In the future, we hope to improve the code editing and interview experience. Right now, our code editor is simply a text box with syntax highlighting support. However, we hope to improve this experience by using an IDE instead, which would provide useful features such as linting and autocompletion. Additionally, it would be great to include an interpreter that can execute the written code and print out the output.

API Documentation

Given more time, our team would hope to implement tools such as Swagger to allow our API to follow a specification. Furthermore, such tools neatly organize API calls into sections and provide documentation on expected inputs and outputs, which will greatly benefit future developers.

Message Queue

Another feature that our team would hope to implement in the future would be message queues. As mentioned above, we hope to implement more microservices into our architecture. As we do so, there is an increased chance that the microservices will need to communicate with each other. To reduce coupling, we hope to implement message queues such as RabbitMQ or Moleculer to allow microservices to communicate directly.

End-to-End Testing

A more robust testing framework such as Jest, WebDriver and Playwright would also be desirable. It provides us automated testing for frontend components as well and solves some of the issues with unit testing several services due to tokens or permissions.

Scalability

As more users use PeerPrep, there may be a need to scale it with Kubernetes or even provide a load balancer. At our current stage, we believe there isn’t a need to do so due to the low traffic of our application. However, as our application scales, having such technology would greatly benefit the performance of the application.

Individual Contributions

Name	Technical Contributions	Non-technical Contributions
Ryan	- Implement user service backend with Sam - Implement queuing logic and socket setup for matching service - Implement question service - Implement review service backend - Refactor frontend components - UI and CSS styling - Add sample unit tests	- Final report - Final presentation - Product design - Project management - README - Documenting Requirements
Bing Cheng	- Implement Matching Service via MongoDB - Implement countdown timer - Implement change question feature - Fixing bugs	- Final report - Final presentation - Product design - Project management
Sam	- Implement the user service backend with Ryan - Implement user account features & authentication checks on frontend - Set up continuous delivery & deployment - Implement the collaboration service & frontend components for collaboration - Implement reviews on profile page - Implement inter-microservice calls	- Final report - Final presentation - Product design - Project management
Jun Yang	- Implement Home page - Implement Profile page with Change Password and Delete Account frontend dialog components - Implement AlertDialog, ConfirmationDialog and SnackbarAlert components for general use - Implement Matching Service frontend - Refractor and implement frontend components for Collaboration Service - Implement Review Service frontend - Fix frontend bugs - UI and CSS styling	- Final report - Final presentation - Product design - Project management

Reflections

The team thoroughly enjoyed the entire development process of this project. On a whole, we learned a great deal in terms of project management, product design, full-stack development, and also deployment. In particular, here are some of our most significant learning points:

- Crafting and prioritizing product requirements. We had to learn how to think like our end-users in order to come up with sensible user requirements. However, when we had too many ideas on the table, we also had to learn how to prioritize our requirements, estimate the effort and time needed to implement them, and finally decide on which requirements to develop.
- Throughout the span of this project, we had to be flexible and pick up many new technologies as the need arose. The team had to pick up React, Express, MongoDB with Mongoose, learn about JWTs, cookies, and many other aspects of full-stack web development. As this was some of our members' first time delving into full-stack development, it was definitely a challenging task to keep pace and apply what we have learnt.
- Choosing, designing, and implementing a deployment pipeline was difficult as none of us had experience in doing so from scratch. We had to learn how to weigh various factors when choosing a cloud provider, including cost, documentation quality, community support, and developer experience. After that, we had to learn how to use Github Actions and DigitalOcean to deploy all our services.

All in all, the team faced many challenges, both technical and non-technical, and managed to overcome them all together. We communicated regularly and effectively over a group chat, made use of clear tasks and issues on GitHub, and did not hesitate to help one another when needed. By the end of the project, everyone had valuable takeaways from one another.