CS3219 Final Report Group 1

PeerPrep

Team Members	Student No.	Email
Chen Hui	A0187372W	e0322956@u.nus.edu
Evelyne Juliet	A0188715R	e0324299@u.nus.edu
Kelvin Harris	A0187349M	e0322933@u.nus.edu
Philip Alexander Boediman	A0192235H	philip.ab@u.nus.edu

Github Repo:
https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g1

Deployed app:
https://cs3219-peerprep-330604.as.r.appspot.com/

Table of Contents

1 Background and purpose **2**
- 1.1 Background 2
- 1.2 Purpose 2

2 Contributions **2**

3 Functional Requirements **7**

4 Non-functional Requirements **8**

5 Design Architecture **9**

6 Developer Documentation **10**
- 6.1 Security 10
- 6.2 Authentication 11
  - 6.2.1 Access and refresh tokens for logins 11
  - 6.2.2 Redis for refresh token caching 11
  - 6.2.3 Interceptors that retrieves and puts access and refresh token in the browser 12
- 6.3 Deployment 13
- 6.4 Socket.IO 14
  - 6.4.1 Matching 14
  - 6.4.2 Chat Messenger 15
  - 6.4.3 End Session 16
- 6.5 Collaborative Editor 17
- 6.6 Database 17
- 6.7 Development Process 18
- 6.8 Design Patterns 18
  - 6.8.1 MVC 18
  - 6.8.2 Pub-Sub 20
- 6.9 Design Decisions 20
  - 6.9.1 Monolithic vs Microservices 20
  - 6.9.2 Socket.IO 20
  - 6.9.3 Yjs and CodeMirror 21
  - 6.9.4 MongoDB 21
- 6.10 Application Programming Interface (APIs) 22

7 Technology Stack **25**

8 Improvements and Enhancements **25**

9 Reflections and Learning Points **26**

10 Screenshots **26**

1 Background and purpose

1.1 Background

Traditionally, students practise technical interviews online individually. Some of these examples can be kaggle, leetcode, etc. However, there doesn't exist an option where two similarly skilled students could practise together on the same question while exchanging thoughts back and forth, simulating the process of a technical interview.

Such a method is useful because there is an abundance of students who would want to practise for technical interviews, but lack the interviewers to sit through a mock interview with them. As such, Peerprep is a webapp that helps students practise technical interviews when applying for jobs.
Common 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.

1.2 Purpose

To tackle such a problem, we would then be required to allow two students of similar skill levels to match with each other, and collaboratively edit on the same document while sharing their thoughts to come up with a solution to a technical question.

Peerprep allows users to register an account and choose a difficulty to practise their technical questions on (easy, medium hard), and be matched with someone else of a similar skill level and collaborate to solve the question together.

2 Contributions

We brainstormed and studied the implementations beforehand. The table below shows the date at which each deliverable was accomplished.

Date (To be completed by 2359)	Deliverable	Assignee	Remarks
Week 7 (27 Sept to 3 Oct)			
28/9	Filter out functional requirements and non-functional requirements	Everyone	
3/10	Tech stack and architecture diagram	Everyone	

Week 8 (4 to 10 Oct)			
10/10	Research deployment on GCP	Phillip	Include CD
10/10	Research redis, authentication	Chen Hui	Hashed password to store on DB
10/10	Initial frontend design with user flow	Evelyne + Kelvin	Mock up, paper drawings
Week 9 (11 to 17 Oct)			
14/10	Skeleton code	Phillip	
17/10	Set up Mongo DB Atlas linked with node.js	Chen Hui	
15/10	Design user flow in detail with (in text)	Evelyne	High priority
17/10	Prototype for Frontend (Based on user flow, can use figma.com)	Kelvin + Evelyne	High priority Two person https://www.figma.com/file/SeJqQhmxkaWC7uXMEtRY4M/CS3219-Wireframe?node-id=0%3A1
Week 10 (18 to 24 Oct)			
20/10	Standardise the database model & Set up relevant model on Node/ Express	Philip	(Test it works using postman)
22/10	Implement API for questions and users.	Philip	
24/10	Authorisation/authentication Task C (Use jwt, node) (Backend)	Chen Hui	Can improve using Redis for caching later on. To integrate with FE

22/10	Dashboard, Statistics Frontend	Evelyne + Kelvin	(Need prototype) + registration
22/10	Matching System Frontend	Evelyne + Kelvin	(Need prototype)
22/10	Collaborative editor Frontend	Evelyne + Kelvin	(Need prototype)
22/10	Frontend for registration + login + signout (Buttons and all)	Evelyne + Kelvin	
24/10	Link registration to backend and storing of tokens in browser	Chen Hui	
24/10	Use redis to cache refresh tokens instead of using mongodb	Chen Hui	
24/10	Matching functionality (Redis or socket.io)	Kelvin	
Week 11 (25 to 31 Oct)			
28/10	Integrate Redux to store user state	Kelvin	(important)
28/10	Update header based on user state	Kelvin	
27/10	CD on GitHub (Backend)	Chen Hui	Include deployment (GitHub actions) + GCP
28/10	Pop ups for logins, register,	Philip	register: notify on using username that is taken login: notify on invalid username & password
28/10	Dashboard pull from backend and make a chart	Evelyne	
28/10	Combine Server and Api folder	Kelvin	
Week 12 (1 to 7 Nov)			

	Work on Final Report	Everyone	Include diagrams for component interactions
5 Nov	Fix CD for backend on GCP and main branch	Chen Hui	
5 Nov	Change backend path in frontend to deployed endpoint	Philip	
5 Nov	Research on conflict merge for collaborative editor	Everyone	
7 Nov	Implement and integrate conflict merge for collaborative editor using CodeMirror and Yjs	Philip + Evelyne	
7 Nov	Add endpoint questions that user has not done based on difficulty to do matching	Philip	
7 Nov	Make sure logout works across frontend storage and backend	Chen Hui	
7 Nov	When no matching user is found after 30 seconds of queuing, the user will be returned to the question difficulty selection page.	Kelvin	
7 Nov	Create a Chat Messenger (backend + frontend)	Kelvin	
7 Nov	Add End Session functionality	Kelvin	
7 Nov	Add notification when all questions are done by user for the chosen difficulty level	Evelyne	
7 Nov	Fix: Only match users with overlap in questions not done	Evelyne	
8 Nov	CD for frontend	Philip	https://javascript.plainenglish.io/quickly-deploy-your-react-app-on-go

			ogles-app-engine-6bb97480cc9c
Week 13 (8 to 10 Nov)			
9/11	Code deliverable	Everyone	Soft deadline
10/11	Final report	Everyone	Hard deadline
10/11	Code deliverable + Final report touch ups	Everyone	Hard deadline

3 Functional Requirements

Below includes the functional requirements which the team manages to deliver.

S/N	Requirement	Priority
User Authentication		
F1.1	The application should allow users to register for an account.	High
F1.2	The application should allow users to sign into their account.	High
F1.3	The application should allow users to logout of their account.	High
Dashboard and Statistics		
F2.1	The application should allow users to view the number of questions solved.	Medium
F2.2	The application should allow users to choose a difficulty level before starting the matching process.	High
F2.3	When no matching user is found after 30 seconds of queuing, the user will be returned to the question difficulty selection page.	High
F2.4	When a matching user is found, both users should be redirected to the question page.	High
Collaborative Session		
F3.1	The question page should display a question based on the difficulty level selected by both users.	High
F3.2	The question page should display a text field in which both users can input their thoughts and collaborate in real-time.	High
F3.3	The question page should display a messenger UI in which users can communicate with messages.	Medium
F3.4	The application should allow any of the users to end the session by clicking the “Finish” button. When any of the users end the session, both users should return to the	High

	question difficulty selection page.	
F3.5	Both users should not be matched with questions which have been encountered by any of the users.	Medium

4 Non-functional Requirements

Below are the non functional requirements which the team manages to deliver.

S/N	Requirement	Priority
Security		
NF1.1	Users must be logged in to access the user's dashboard and queue for matching.	High
NF1.2	Access token must have a lifecycle (short-lived) and should not be usable for an indefinite amount of time	High
NF1.3	Once a user logs out, all tokens must be removed from any storage	High
Reliability		
NF2.1	Collaborative editing on the text field should be consistent and correct, and should not result in version conflict.	High
NF2.2	The dashboard should display the user's data correctly.	Medium
Performance		
NF3.1	Collaborative editing in the text field of the question page should be responsive.	High
NF3.2	The average time it takes to match two users should be less than 10 seconds.	Low
Capacity and Scalability		
NF4.1	The system is able to store up to 10000 questions.	Medium

Our NFR quality attributes are prioritised in the following order from highest to lowest priority:

1. Security
2. Reliability
3. Performance
4. Capacity and Scalability

We prioritise security first because we believe that it is the basis for a production level application. This means only authenticated users are able to access the permitted resources. Reliability comes next as the main features of our application revolves around the collaborative session. Hence, we need to make sure that the collaborative editor and chat messenger work correctly. Finally, it makes sense to enhance the performance and scalability of our application when security and reliability are guaranteed.

5 Design Architecture

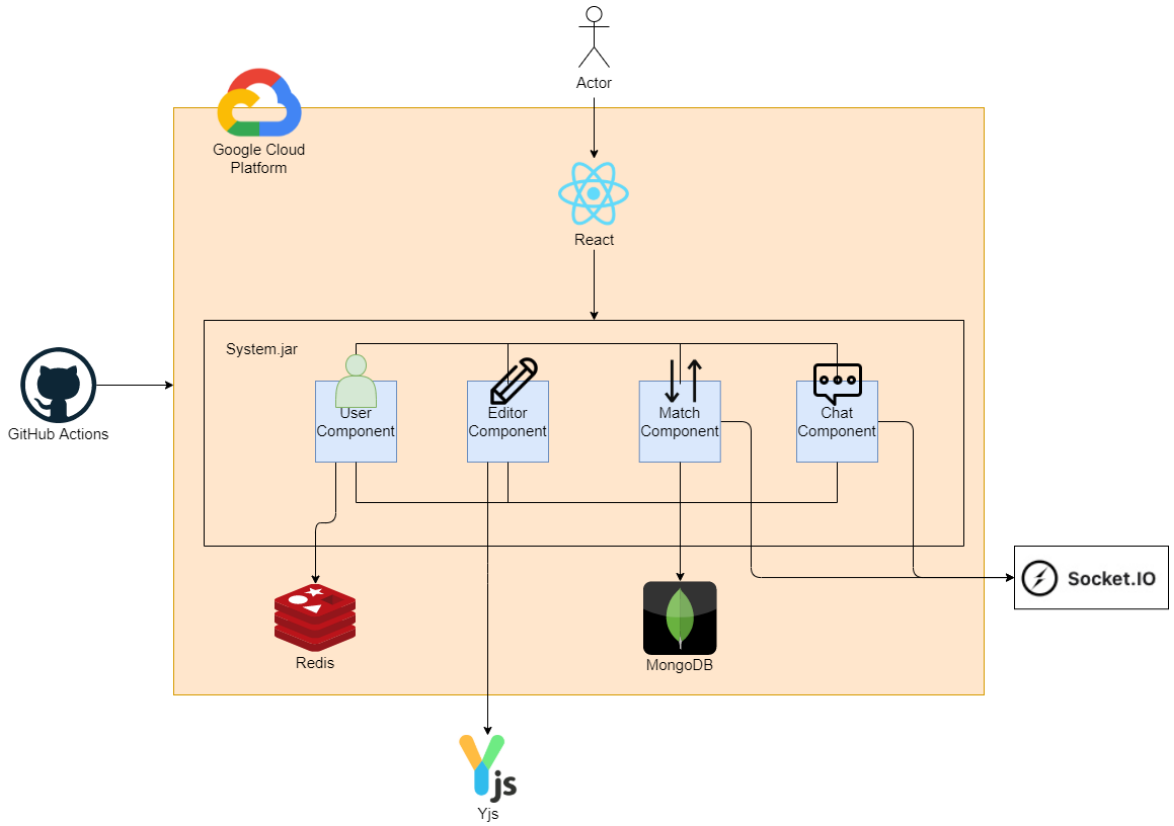

Figure 5.1. Architecture Diagram

Our software application uses a monolithic architecture with two major services: the Backend and Frontend service.

6 Developer Documentation

6.1 Security

The application is made secure by adding the following features:

- JWT (jsonwebtoken)
- bcryptjs
- Middleware for authentication and authorisation check

```
//middleware function to check if the incoming request in authenticated:
exports.checkAuthenticated = (req, res, next) => {
    // get the token stored in the custom header called 'x-auth-token'
    // console.log("Check auth req: ", req.get("x-auth-token"))
    const token = req.get("x-auth-token");
    //send error message if no token is found:
    if (!token) {
        return res.status(401).json({ error: "Access denied, token missing!" });
    } else {
        try {
            //if the incoming request has a valid token, we extract the payload from the token and attach it to the request object.
            const payload = jwt.verify(token, privateKey);
            req.user = payload.user;
            console.log("user: ", payload.user)
            next();
        } catch (error) {
            // token can be expired or invalid. Send appropriate errors in each case:
            if (error.name === "TokenExpiredError") {
                return res.status(401).json({ error: "Session timed out, please login again" });
            } else if (error.name === "JsonWebTokenError") {
                return res.status(401).json({ error: "Invalid token, please login again!" });
            } else {
                //catch other unprecedented errors
                console.error(error);
                return res.status(400).json({ error });
            }
        }
    }
};
```

Figure 6.1.1. Authentication middleware implementation

```
// Users
router.put("/updateUser", Middleware.checkAuthenticated, UserController.addAnsweredQuestion);
router.get("/user", Middleware.checkAuthenticated, UserController.getUserData);
router.get("/user/:difficulty", Middleware.checkAuthenticated, UserController.getQuestionsNotDoneBy);
```

Figure 6.1.2. Authentication middleware usage

We use JWT to generate and verify both access-token and refresh-token. These tokens will be used for authentication and authorization purposes. We use *bcryptjs*, a Node library to hash a user's password when storing them in the DB. *bcryptjs* has a random salt feature to better secure the storage of the password in the DB (i.e prevent rainbow attack). The middleware that we have makes use of both of these 2 for the authentication and authorization of a particular user. The access-token that is generated has a life cycle of 30 minutes, thus a stolen access-token cannot be used for an indefinite amount of time.

6.2 Authentication

6.2.1 Access and refresh tokens for logins

We did our own server side authentication with hashed passwords, instead of using existing authentication libraries such as gmail or microsoft logins.

When a user signs up or logs in, he is given an access token and a refresh token. The access token is constantly being used to authenticate the user's request to the server side, instead of doing a client side authentication.

An access token is short lived, therefore there is also a refresh token given to the user. This refresh token helps to refresh the access token, when the user's access token is expired. Our refresh tokens are stored in redis to last up to a day.

6.2.2 Redis for refresh token caching

Since there is a need to start a server for redis, we deployed a redis instance on GCP.

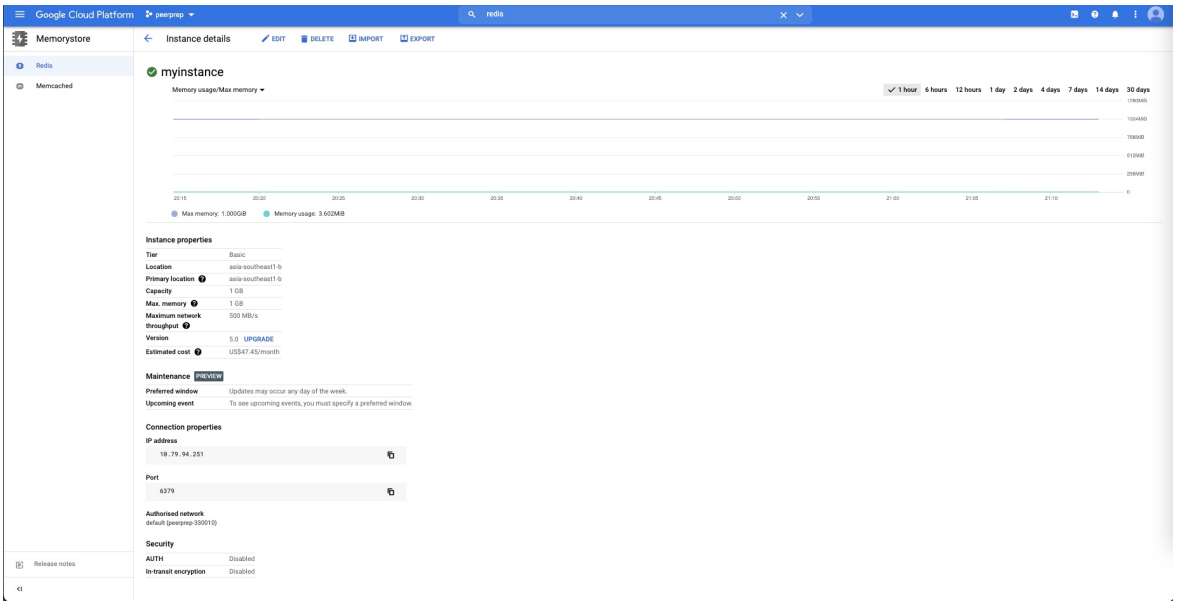

Figure 6.2.2.1. redis deployment

Our configuration for connecting to the redis instance is as shown below.

```yaml
runtime: nodejs14

# Update with Redis instance details
env_variables:
  REDISHOST: '10.79.94.251'
  REDISPORT: '6379'

# Update with Serverless VPC Access connector details
vpc_access_connector:
  name: 'projects/peerprep-330010/locations/asia-southeast1/connectors/redisconnector'
```

Figure 6.2.2.2. redis configuration

The following screenshot shows an example of a successful refresh token using Postman to our deployed endpoint.

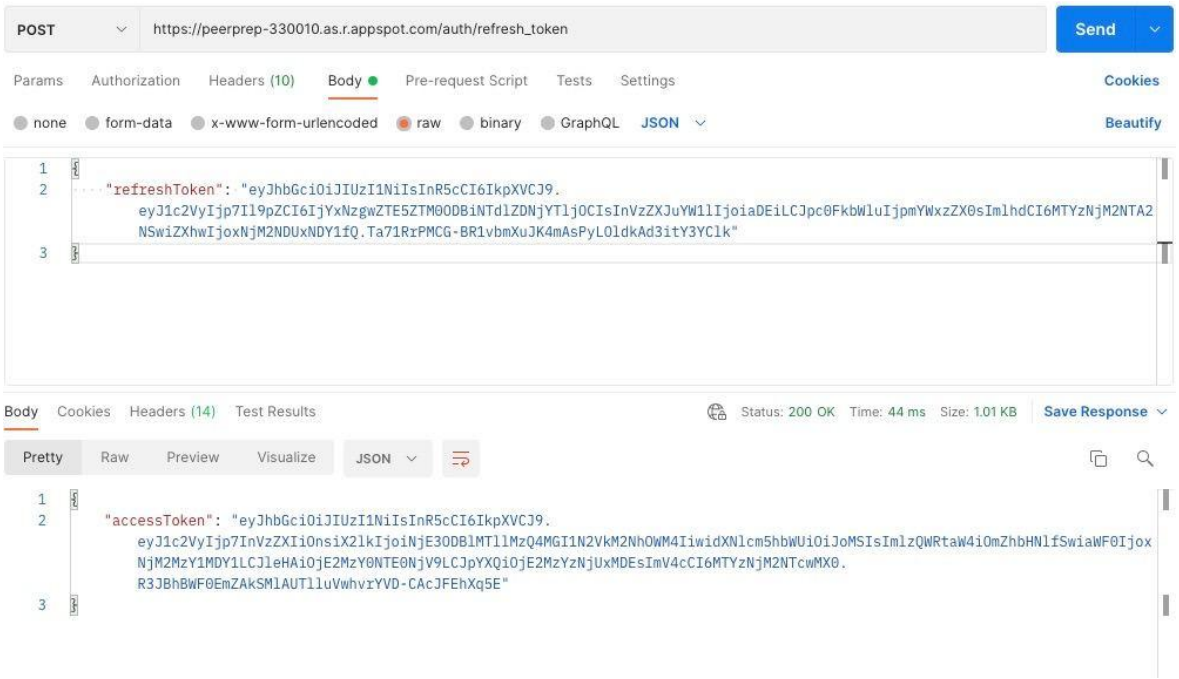

Figure 6.2.2.3. Refresh token with Postman

6.2.3 Interceptors that retrieves and puts access and refresh token in the browser

We use axios services which allows the use of interceptors in our request and responses, which would help to store and retrieve the value of access and refresh tokens during our request and responses, especially for requests related to authentication (i.e login, signup, renew access token).

It also helps with setting the expiry time of the user's access token which is stored in the local storage. Thus, everytime the client wants to make a request using the access token, the client can check against the expiry time if the access token has expired.

By doing so, it conveniently handles all the retrieval and storage of the tokens when the client is making any kinds of requests related to authentications.

```
import axios from "axios";

const dbURL = "http://localhost:8080/"

export const axiosService = axios.create({
    baseURL: dbURL,
});

axiosService.interceptors.request.use((config) => {
    config.data['accessToken'] = localStorage.getItem('accessToken')
    config.data['refreshToken'] = localStorage.getItem('refreshToken')
    console.log("manipulated config: ", config)
    return config;
}, (error) => {
     return Promise.reject(error.message);
});

axiosService.interceptors.response.use((response) => {
    console.log("Perform the response mani here: ", response)
    localStorage.setItem('accessToken', response.data["accessToken"]);
    if (response.data["refreshToken"] != undefined) {
        localStorage.setItem('refreshToken', response.data["refreshToken"]);
    }
    localStorage.setItem('expireTime', Date.now() + 10*60*1000);
    return response;
}, (error) => {
    return Promise.reject(error.message);
});
```

Figure 6.2.3.1. axios service

6.3 Deployment

We Github Actions for our continuous deployment providers and we deployed our app's frontend and backend to google app engine on GCP. Note that our redis is also hosted on GCP.

The logs for our continuous deployments can be viewed in the GitHub Actions tab of our repository. [link]

Frontend: https://cs3219-peerprep-330604.as.r.appspot.com/
Backend: https://peerprep-330010.as.r.appspot.com/

6.4 Socket.IO

6.4.1 Matching

We used Socket.IO for maintaining the collaborative sessions for Peerprep. Socket.IO is a JavaScript library for real time web applications which supports bidirectional communication using primarily WebSockets. We also utilised Socket.IO “rooms”, where sockets can join and leave.

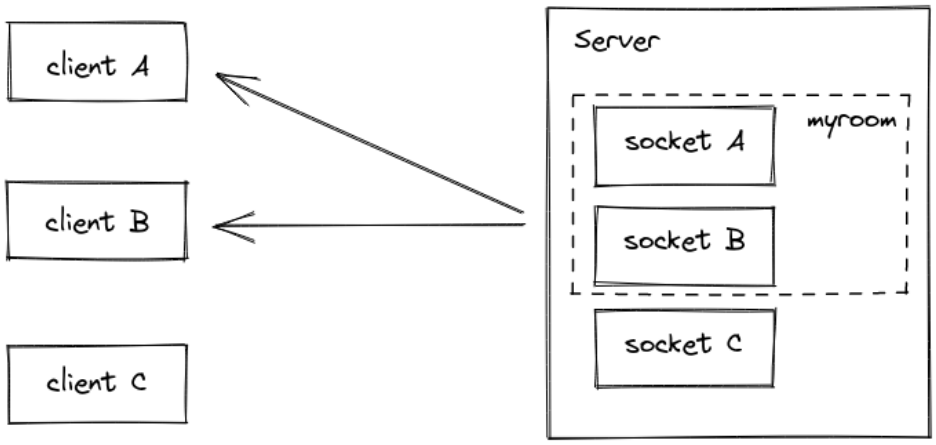

Figure 6.4.1.1. Socket.IO rooms[1]

Socket.IO rooms are maintained in the same backend server as the application. This is because both the Express and Socket.IO connection use the same port.

```
const app = require('express')();
const server = require('http').createServer(app);
const io = require('socket.io')(server);
io.on('connection', () => { /* … */ });
server.listen(3000);
```

Figure 6.4.1.2. Using Socket.IO with Express[2]

The number of connections allowed for each room is limited to 2 as per the requirement of Peerprep.

The communication channels in Socket.IO use the pub-sub messaging design pattern. In our scenario, client and server act as both publisher and subscriber.

[1] https://socket.io/docs/v3/rooms/
[2] https://github.com/socketio/socket.io#in-conjunction-with-express

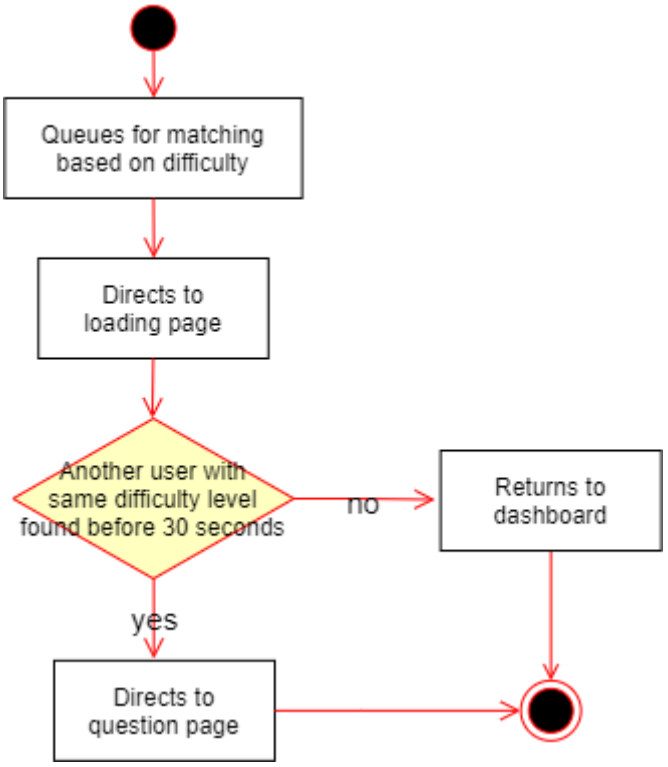

Figure 6.4.1.3. Activity diagram for Matching functionality

When the first user joins a session, a room will be created with a unique ID and a difficulty tag chosen. Each room ID is generated using the `uuidv4` module, which generates a unique identifier, concatenated with the difficulty tag. When another user joins, the system would do the following:

1. Check if there is a room with matching difficulty level
2. Check if there is any question not done by both users

If there is a match, both users would be subscribed to the same room and be redirected to the collaborative question page. The system picks a random question that has not been done by both parties of the difficulty level specified.

Hence, the client side has two listeners when joining a session; **`matched`** for when another matching user is found and goes to the collaborative page, and **`connected`** for when the client is joining a new room and goes to the loading page.

6.4.2 Chat Messenger

We also used the same Socket.IO rooms for the real time chat messaging between two parties. New messages are broadcasted over to the room using roomId and subscribers receive the message at the other end.

Client

We also found out that the socket needs to be “turned off” when updating React state. This prevents sockets from emitting multiple events which causes the application to crash with memory overflow.

```
useEffect(() => {
  const addMessage = ({ username, message }) => {
    setMessages([...messages, { username, message }]);
    const panel = document.getElementById("panel-body");
    panel.scrollTop = panel.scrollHeight;
  }
  socket.on('receiveMessage', addMessage);
  return () => {
    socket.off("receiveMessage", addMessage);
  };
});

const sendMessage = () => {
  const message = document.getElementById('message-input').value;
  document.getElementById('message-input').value = '';
  socket.emit('sendMessage', { roomId: props.roomId, message:
    message, username: username });
}
```

Server

```
client.on('sendMessage', ({ roomId, message, username }) => {
  rooms[roomId]['messages'].push({ username, message });
  io.to(roomId).emit('receiveMessage', { username, message });
});
```

6.4.3 End Session

Any party can end the session by simply clicking the ‘End Session’ button on the collaborative page. When this is done, the backend does a cleanup. `roomId` corresponding to the ended room will be removed from the list and both parties will leave/unsubscribe the room.

6.5 Collaborative Editor

For real time collaborative editing between two parties on Peerprep:

- CRDT implementation: Yjs
- Text Editor & Syntax highlighting: CodeMirror
- Connection: WebRTC

Features include:

- Sync CodeMirror editor
- Shared Cursors
- Shared Undo / Redo (each client has its own undo-/redo-history)
- Successfully manage concurrents edits

6.6 Database

We used mongodb atlas to host our database. A snapshot of our data can be described as such:

Below is an example of a question stored in the questions collection. It contains information about a particular question.

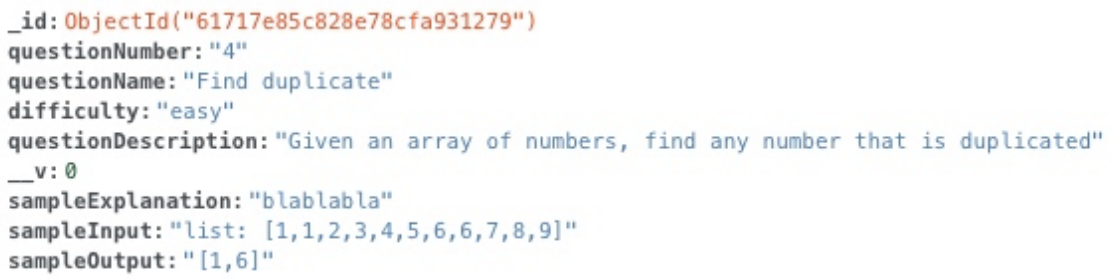

Figure 6.6.1. Question object

Below is an example of a user stored in the users collection. Notice that the password is hashed, and contains the questions done by the user.

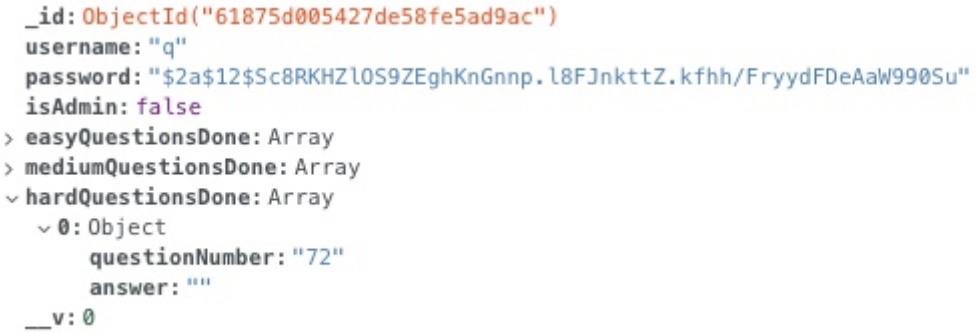

Figure 6.6.2. User object

6.7 Development Process

We have decided to follow an agile development methodology with weekly meetings to discuss each other's progress as well as to check in on one another if anyone is stuck at any part of their tasks. We have decided to follow an iterative breadth-first development process because of the reason that this allows us to develop the different components of the software in parallel. Doing so, we will also be able to integrate the different parts of the application early in our development cycle.

6.8 Design Patterns

6.8.1 MVC

We have decided to use MVC for one of our backend implementations by using mongoose package from NodeJS as part of our integration process with the MongoDB database that we use in our application. Mongoose helps us model the data by creating a predefined structure that represents each in the DB. Here is an example of our question model.

```
// Setup schema
var questionSchema = new mongoose.Schema({
    questionNumber: {
        type: String,
        required: true,
        unique: true,
        dropDups: true
    },
    questionName: {
        type: String,
        required: true,
    },
    difficulty: {
        type: String,
        required: true
    },
    questionDescription: {
        type: String,
        required: true
    },
    sampleInput: {
        type: String
    },
    sampleOutput: {
        type: String
    },
    sampleExplanation: {
        type: String
    }
});
```

Figure 6.8.1.1. Question model

The mongoose also allows us to view (fetch) our datas from the MongoDB database, and returns us the data base on the model that we have defined like one of the above (Figure. question model). The mongoose package also helps us interact with the MongoDB database to enable us to control (insert/update) the database with the controllers we defined. The controllers' methods can be called using the endpoint defined in routes.

```
// Questions
router.get("/getQuestions", QuestionController.getAllQuestion);
router.post("/addQuestion", QuestionController.addQuestion);
router.delete("/deleteQuestion", QuestionController.deleteQuestion);
router.put("/updateQuestion", QuestionController.updateQuestion);

// Users
router.put("/updateUser", Middleware.checkAuthenticated, UserController.addAnsweredQuestion);
router.get("/user", Middleware.checkAuthenticated, UserController.getUserData);
router.get("/user/:difficulty", Middleware.checkAuthenticated, UserController.getQuestionsNotDoneBy);
```

Figure 6.8.1.2. MongoDB routes

The interaction between User and the MVC model can be seen in the diagram below.

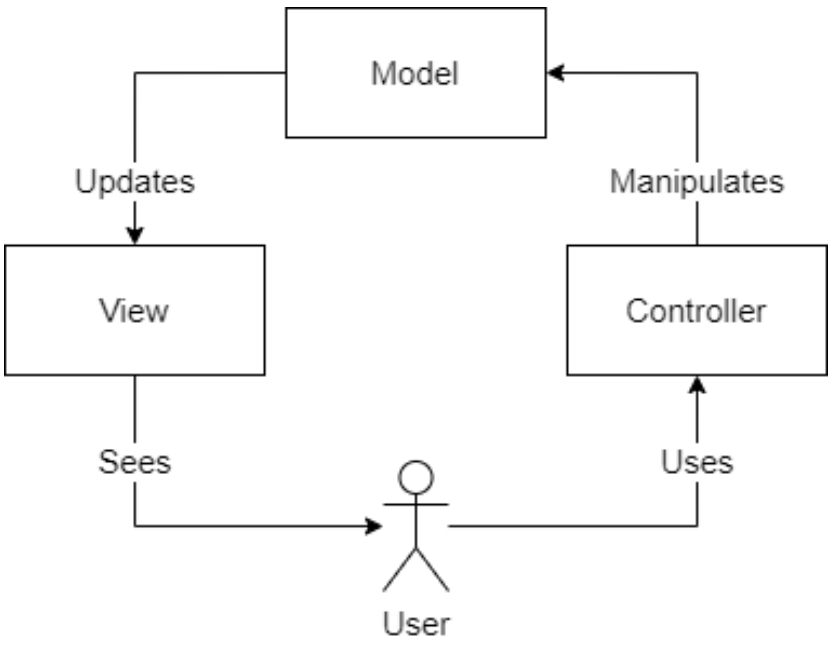

Figure 6.8.1.3. MVC Model

6.8.2 Pub-Sub

We have also decided to use a pub-sub messaging pattern by utilising Socket.io to implement our matching (for allocating the room between each user) and our chat functionality. The publishers that are involved in this case are the users that are in the client service who are looking for a match, the backend service that notifies the user that the user has been matched and also the backend service that helps to emit a message to end the session for a particular room.

While the subscribers that are involved in this case are the users on the client side who are in the waiting room waiting for a match as well as the client service that receives a message to end the session.

Please refer to section 6.4 for more details on the pub-sub implementations.

6.9 Design Decisions

6.9.1 Monolithic vs Microservices

Given limited time and resources, we decided to go with the monolithic architecture as it is simpler to manage, develop and deploy. Monolithic applications also allow ease of end-to-end testing as all the components that are in the backend and frontend lie in a single application. Additionally, since we don't have any centralized logging system, loggings in monolith architecture through console logs would be much easier to understand and it gives us a better overview of the program flow. Moreover, as the complexity of the project and the requirements of the application are unlikely to increase in the future, the problem of scalability is not pertinent in this case. Thus, with a fixed set of Functional and Non-Functional requirements, we believe that the monolithic architecture is the best approach that we should adopt.

6.9.2 Socket.IO

For both matching and chat functionality, WebSocket protocol is used for providing full-duplex communication. It allows for higher efficiency as compared to HTTP REST which has the request/response overhead for each message sent.

We have decided to adopt Socket.IO library as the main tool for maintaining the communication channels. It is built on top of Web While there are other libraries available such, Socket.IO offers a robust set of functionalities that complements our requirements, including:

1. Fallback to technologies other than WebSockets when the client does not support it such as HTTP Long Polling.[3]

[3] https://socket.io/docs/v4/how-it-works/#transports

2. Feature to broadcast messages to connected clients.
3. Server-side feature to support arbitrary channels called ‘rooms’.
4. APIs that are easy to work with.

While there seems to be overhead in terms of performance as compared to Vanilla WebSocket, this concern may not be noticeable or important for our application.

6.9.3 Yjs and CodeMirror

We have decided to use Yjs for our conflict resolution solution in our collaborative editing platform because it is an open-source software that is widely popular among JavaScript developers and it uses a robust CRDT algorithm to manage conflict resolution. It is also being well-maintained by the group of developers on GitHub and has strong support from the online community on various platforms. This ensures the reliability of the application such that the editor is consistent and correct.

We have also decided to choose CodeMirror as the text editor in our collaborative editing react application as it is one of the few text editors that supports seamless integrations with the Yjs package via “y-codemirror” package. We have also opted to use WebRTC as it is the underlying connection for the package that we use (y-webrtc) that allows for easy integration with Yjs.

This implementation ensures the performance of the editor by improving the editor’s latency between the two users to be real-time.

6.9.4 MongoDB

We have decided to use MongoDB as our database engine because of our familiarity in integrating with MongoDB via NodeJS. MongoDB is also one of the most popular database engines out there as it is: 1. Free to use 2. It has excellent performance with simple queries.

Additionally, MongoDB uses NoSQL databases and we find that NoSQL databases are a good fit for our software application as the datas that we are dealing with is not very structured data. For example, Users data in our Database contains an array of numbers for the questions that they have done and this can be easily implemented using NoSQL databases whereas in SQL, it doesn’t have any support for such data types and thus we would have to stringify the array for storage and parse the string to an array when retrieving the data which makes implementation less convenient (in general, we need to use object relational mapping for SQL). In MongoDB, such data types can be mapped directly to the underlying data structure in Javascript, and the object as a whole can be retrieved and represented directly as a JSON object in javascript.

Moreover, MongoDB works well with a large amount of data and convenient scaling as it allows us to scale the database horizontally, as compared to other SQL

database engines that only allows for vertical scaling. Thus we can fulfill the scalability aspect for our data storage.

6.10 Application Programming Interface (APIs)

Questions

GET	/db-data/question/get
Description	Get all questions available in the database
POST	/db-data/question/add
Description	Add question to database

body	<pre>{
	  "number": {question number},
	  "name": "{question title}",
	  "difficulty": "{question difficulty}",
	  "description": "{question description}",
	  "sampleInput": "{sample question input}",
	  "sampleOutput": "{sample question output}",
	  "sampleExplanation": "{sample explanation}"
	}</pre>

PUT	/db-data/question/update
Description	Modify a specified question in the database

body	<pre>{
	  "number": {question number},
	  "name": "{question title}",
	  "difficulty": "{question difficulty}",
	  "description": "{question description}",
	  "sampleInput": "{sample question input}",
	  "sampleOutput": "{sample question output}",
	  "sampleExplanation": "{sample explanation}"
	}</pre>

DELETE	/db-data/question/delete
Description	Delete question in the database

body	{
	"number": {question number}
	}

Users

GET	/db-data/user
Description	Get all of the user data

header	{
	"x-auth-token": "{user's access token}"
	}

GET	/db-data/user/{:difficulty}
Description	Get all questions that are not done by the user

header	{
	"x-auth-token": "{user's access token}"
	}

PUT	/db-data/user/update
Description	Add a question to the list of questions that have been done by the user

header	{
	"x-auth-token": "{user's access token}"
	}

body	{
	"userName": "{userName}",
	"difficulty": "{question difficult}",
	"number": "{question number}",
	"answer": "{the answer that the user had written in the editor}"
	}

Authentication

POST	/auth/login
Description	Authenticate the user

body	<pre>{
	  "username": "{username}",
	  "password": "{password}"
	}</pre>

POST	/auth/signup
Description	Register the user with a given role (admin or not admin)

body	<pre>{
	  "username": "{username}",
	  "password": "{password}"
	  "isAdmin": **{true/false}**
	}</pre>

POST	/auth/refresh_token
Description	Gets new access token for the user

body	<pre>{
	  "refreshToken": "{refresh-token}"
	}</pre>

DELETE	/auth/logout
Description	Logs a user out of the session by removing the user's access-token and refresh-token

body	<pre>{
	  "refreshToken": "{refresh-token}"
	}</pre>

7 Technology Stack

The following table summarises the technology stack employed for the development of Peerprep.

	Technology
Frontend	React
Backend	Express.js/Node.js
Database	MongoDB, Redis cache
Deployment	GCP and GCP redis
Pub/Sub Messaging	Socket.io
CI/CD	GitHub Actions
Project Management Tools	Google Meets, Zoom, Google Docs

8 Improvements and Enhancements

The dashboard, although working as intended, can be more descriptive. Having referenced from leetcode, leetcode allows more descriptive and multi layers of filtering to showcase the questions that you have done, have not yet, attempted, etc.

Such a dashboard is definitely more informative and can motivate students to want to do more questions. As such, an improvement would definitely be to make the dashboard show more details of the user's progress. For instance, time spent on the app, time spent doing questions, how many friends have he met, etc.

Additionally, a great improvement would be to allow users to be able to choose a language for the various syntax and static type (for statically typed language) checkings. On top of these, users should be able to run the codes so that they can actually test their code out instead of copying and pasting their codes somewhere else to test out their implementations.

Lastly, it will make the overall experience better for users to be able to see suggested solutions to each of the problems at the end of the discussion between the pair of peers working on the problem.

9 Reflections and Learning Points

Our team went through a lot of trouble trying to set up the CD in the later stage. This was due to unforeseen circumstances, such that not realising the need to have a separate instance for redis. (As compared to starting redis-server locally). There was also a lot of time spent trying to figure it out without knowing how to debug on Github Actions and GCP console.

Having completed the project, we felt that we should prioritise the CD part earlier, as it also helps with verifying what we have done can be reproduced in production, so in the later stages we do not have to re-verify.

Currently both access tokens and refresh tokens are stored in the local storage of the browser. Although usable, it is not very secure. For instance, the security of access tokens are of the same level as refresh tokens because both of them are found in the local storage. In practise, refresh tokens should be way more secure than access tokens, and should not be easily accessible by other attackers.

10 Screenshots

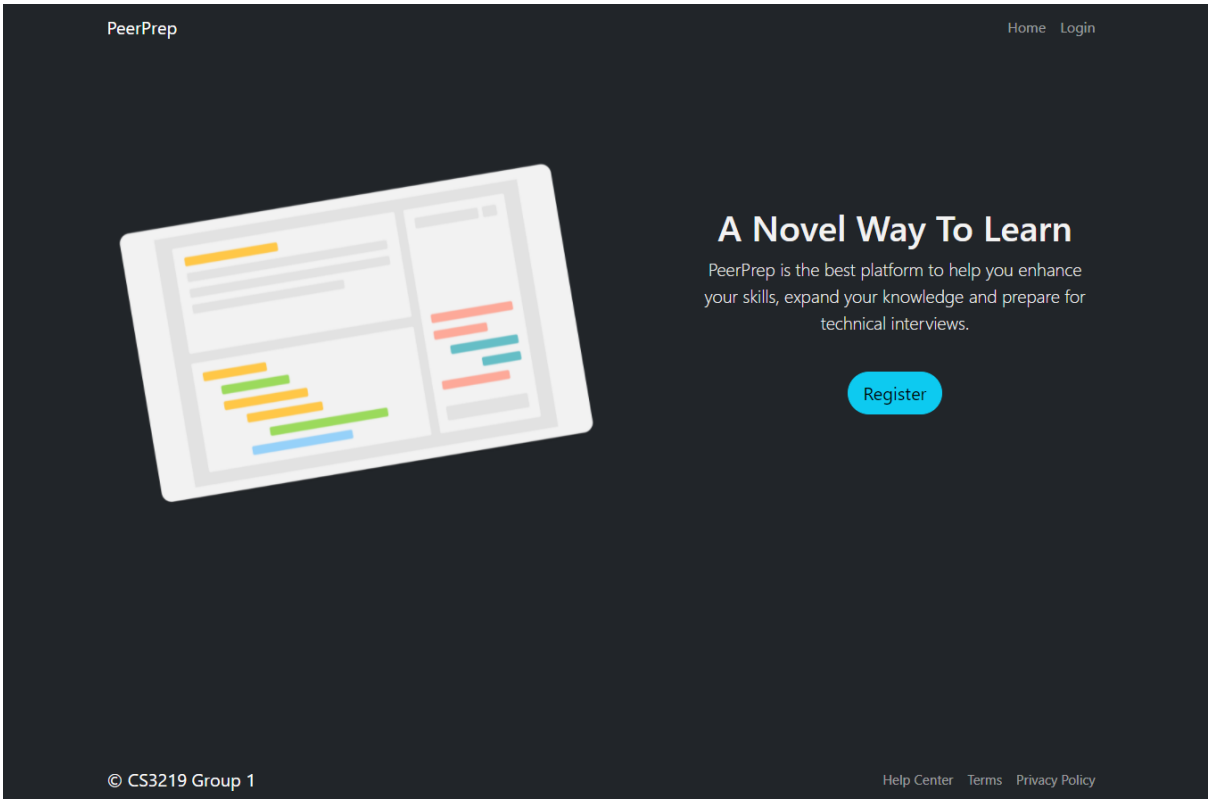

Figure 10.1. Homepage

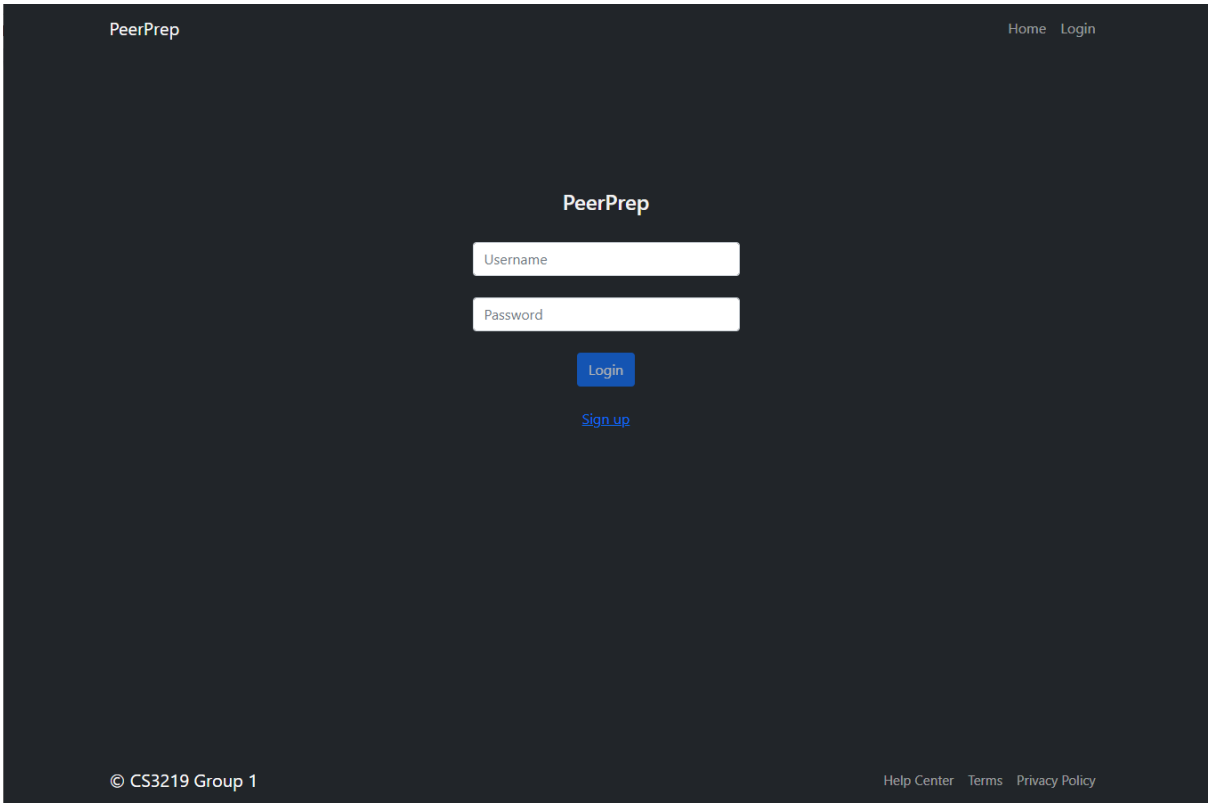

Figure 10.2. Login page

Figure 10.3. Dashboard

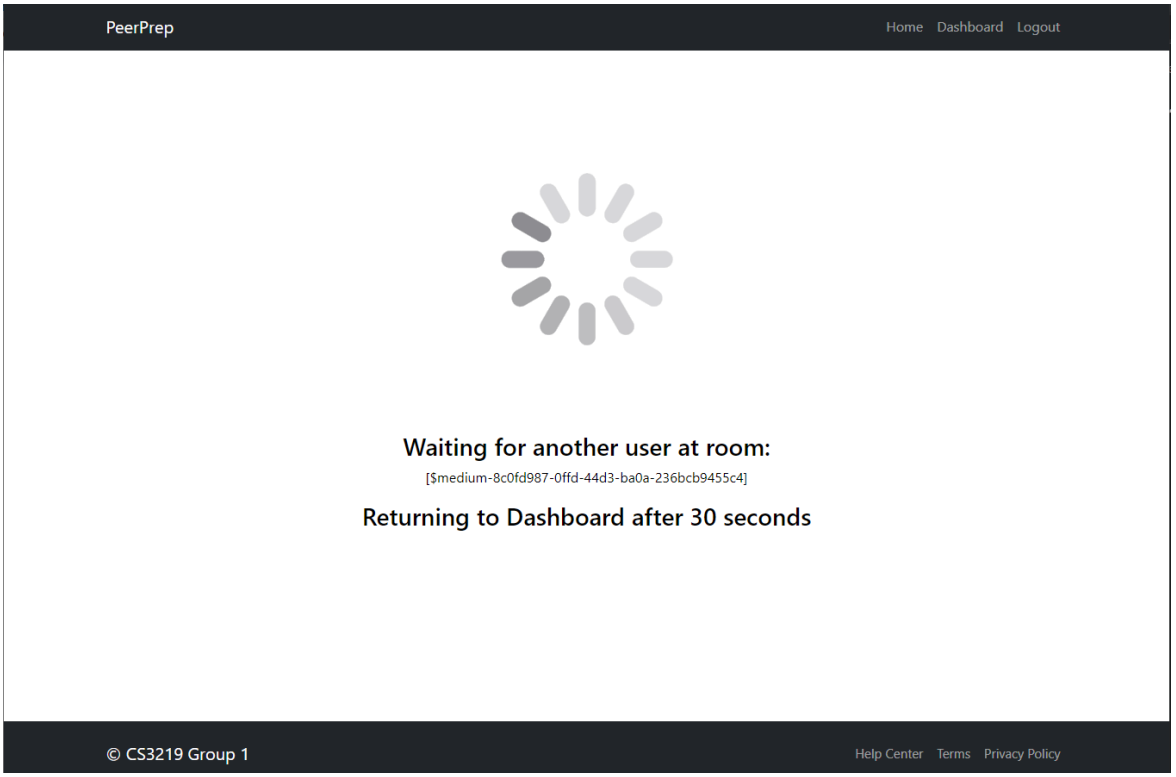

Figure 10.4. Loading page

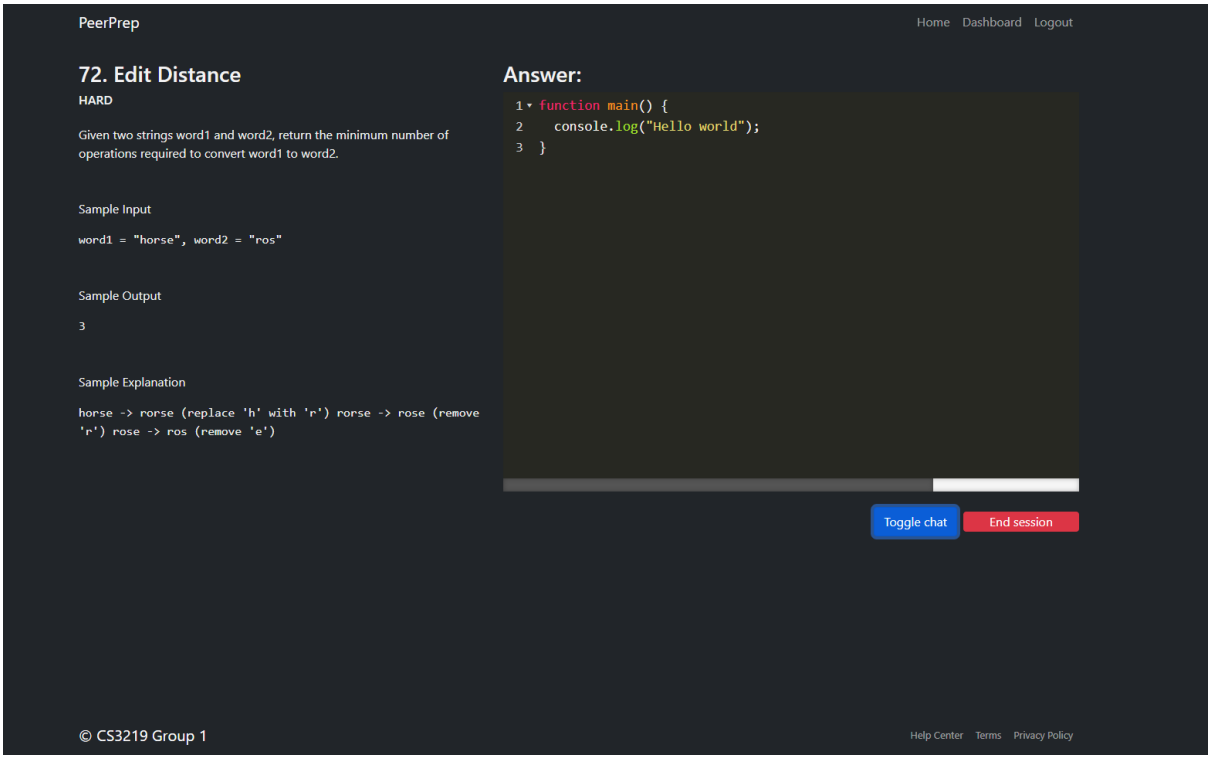

Figure 10.5. Collaborative page (Editor)

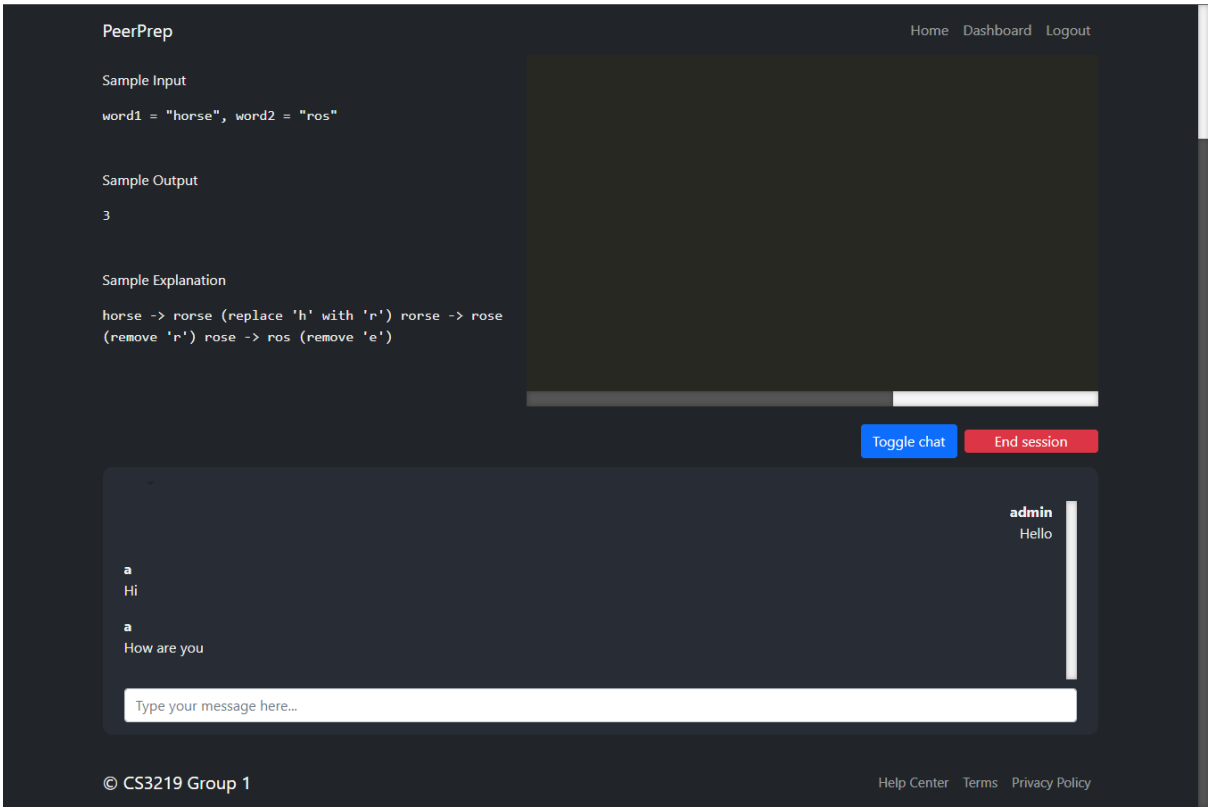

Figure 10.6. Collaborative page (Chat)