

# CS3219 - AY2022/2023 Semester 1

G20

Hazel Tan  
Wang Xuanqi  
Goh Siau Chiak  
Marcus Tan

## 1. Introduction

### 1.1 Peerprep

Peerprep is a peer support system where users can experience collaborative whiteboard-style programming to practise for technical interviews. Besides coding on the same file at the same time, users can also chat with their partner in real time. Peerprep uses user authentication and keeps track of a user's previously attempted questions. Users can also request for a change of questions during a programming session. Questions are taken from LeetCode's collection of free questions and categorised into three difficulty levels (Easy, Medium, Hard). The sample size of our questions is kept small as we are in the minimal viable product stage.

### 1.2 Motivation

A technical coding interview is something that all Software Engineering students will eventually have to tackle. It has become an indispensable component of the application process of companies that are hiring Software Engineers. It is normal for aspiring Software Engineers to feel intimidated by these interviews, especially if they have never experienced one before. Some may find it stressful to be assessed live, while others may find it tough to articulate their thoughts clearly while coding at the same time.

To help these people, we want to create an interview preparation platform where students can find peers to practise whiteboard style interview questions together. We feel that the previously mentioned challenges can be overcome through rigorous practising, and we want to build a platform that can enable students to do this practice.

## 2. Functional Requirements

### 2.1 User Service

S/N	Functional Requirement	Priority
1.1	The system should allow users to create an account with username and password.	High
1.2	The system should ensure that every account created has a unique username.	High
1.3	The system should allow users to log into their accounts by entering their username and password.	High
1.4	The system should allow users to log out of their account.	High
1.5	The system should allow users to delete their account.	High
1.6	The system should allow users to change their password.	Medium

### 2.2 Matching Service

S/N	Functional Requirement	Priority
2.1	The system should allow users to select difficulty level of the questions they wish to attempt	High
2.2	The system should be able to match two waiting users with similar difficulty levels and put them into the same room	High
2.3	If there is a valid match, the system should match the users within 30s.	High
2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
2.5	The system should provide a means for the user to leave a room once matched.	Medium

## 2.3 Collaboration Service

S/N	Functional Requirement	Priority
3.1	The system should allow the user to type text into a text field.	High
3.2	The system should allow the user and the collaborator to work on the same piece of code together in near real time.	High
3.3	The system should allow the user to request for a change in question and facilitate said change if the collaborator agrees.	Medium
3.4	The system should allow the user to end the session once the user is ready to end the session.	Medium

## 2.4 Question Service

S/N	Functional Requirement	Priority
4.1	The system should allow users to retrieve a randomly selected question of a given difficulty.	High
4.2	The system should allow users to retrieve a question's title, description, difficulty and explanation given its unique identifier.	High
4.3	The system should allow users to retrieve all questions stored in the database.	Medium

## 2.5 Chat Service

S/N	Functional Requirement	Priority
5.1	The system should allow users to type messages into a chat box.	High
5.2	The system should allow users to send and receive messages from each other.	High
5.3	The system should deliver messages to each other in near real time.	High

## 2.6 History Service

S/N	Functional Requirement	Priority
6.1	The system should allow users to view which questions they have previously attempted.	High

## 3. Non-Functional Requirements

### 3.1 Performance Requirements

Key points:

- The application will not take more than 5 seconds to change from one screen to another.
- API calls from frontend to backend services should have a response time of at most 3 seconds during peak periods.

S/N	Non-functional Requirement	Priority
1.1	The application will not take more than 5 seconds to change from one screen to another.	Medium
1.2	API calls from frontend to backend services should have a response time of at most 3 seconds during peak periods.	Medium

To stress test our HTTP servers, we used *wrk*, a HTTP benchmarking tool. We tested a key endpoint from each of the services (Frontend, User, Question and History) using a sample of 50 users. The following results demonstrate that the maximum latency recorded is well under the limit of 3 seconds (< 3000ms).

Service	Endpoint	Average Latency	Maximum Latency
Frontend	/	9.57ms	41.40ms
User	/login	949.30ms	1560.0ms
Question	/question/difficulty	467.21ms	998.72ms
History	/history/{username}	377.28ms	672.57ms

Screenshots

Service	Results
Frontend	<pre>Running 5s test @ https://cs3219g20.as.r.appspot.com/ 50 threads and 50 connections Thread Stats   Avg      Stdev     Max    +/-  Stdev Latency       9.57ms   1.92ms   41.40ms  79.63% Req/Sec      104.09   13.58   140.00   70.92% 26043 requests in 5.03s, 25.88MB read Requests/sec: 5175.39 Transfer/sec:  5.14MB</pre>

User	<pre>Running 5s test @ https://user-service-dot-cs3219g20.as.r.appspot.com/login 50 threads and 50 connections Thread Stats   Avg      Stdev     Max    +/-  Stdev Latency    949.30ms  332.91ms  1.56s   75.55% Req/Sec     1.03      2.51    10.00   91.27% 229 requests in 5.05s, 166.16KB read Requests/sec: 45.35 Transfer/sec: 32.90KB</pre>
Question	<pre>Running 5s test @ https://question-service-dot-cs3219g20.as.r.appspot.com/question/difficulty/ 50 threads and 50 connections Thread Stats   Avg      Stdev     Max    +/-  Stdev Latency    467.21ms  237.55ms  998.72ms  62.72% Req/Sec     2.84      3.49    20.00   83.74% 467 requests in 5.05s, 240.34KB read Socket errors: connect 0, read 0, write 0, timeout 3 Requests/sec: 92.49 Transfer/sec: 47.60KB</pre>
History	<pre>Running 5s test @ https://history-service-dot-cs3219g20.as.r.appspot.com/history/sc 50 threads and 50 connections Thread Stats   Avg      Stdev     Max    +/-  Stdev Latency    377.28ms  145.14ms  672.57ms  70.83% Req/Sec     2.32      2.44    10.00   90.60% 266 requests in 5.05s, 164.18KB read Socket errors: connect 0, read 0, write 0, timeout 50 Requests/sec: 52.72 Transfer/sec: 32.54KB</pre>

To stress test our socket.io servers, we wrote our own benchmarking script. We tested the latency of connections to each of the services (Matching, Collaboration, Chat) using a sample of 50 users. The following results demonstrate that the maximum latency recorded is well under the limit of 3 seconds (< 3000ms).

Service	Average Latency	Maximum Latency
Matching	53.68ms	81ms
Collaboration	53.28ms	64ms
Chat	54.38ms	69ms

## Screenshots

Service	Results
Matching	<pre>average latency: 53.68ms maximum latency: 81ms</pre>
Collaboration	<pre>average latency: 53.28ms maximum latency: 64ms</pre>
Chat	<pre>average latency: 54.38ms maximum latency: 69ms</pre>

## 3.2 Usability Requirements

Key points:

- The application should be intuitive and easy to understand. Users should not spend too long trying to complete a task.
- Minimalistic, straightforward design

S/N	Non-functional Requirement	Priority
2.1	Intuitive and user friendly UI such that the whole mock interview process will be an easy flow for users.	High
2.2	Minimalistic, straightforward design where there are no unnecessary frontend components/animations/transitions distracting the user.	High
2.3	Users can easily access the application without the need to key in their login details every time they launch the website - cookie expires in 24 hours.	Medium
2.4	Error messages are easy to understand.	Low

We utilised the industry standard's System Usability Scale (SUS) to grade the usability of our product. We invited 10 participants to try Peerprep and provide their feedback by answering the following questions.

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

We used the calculator [here](#) to compute the scores based on our participants' responses.

- Minimum score: 62.5
- Maximum score: 90
- Average score: 80

### 3.3 Availability Requirements

Key points:

- The system should be up at least 99% of the time.

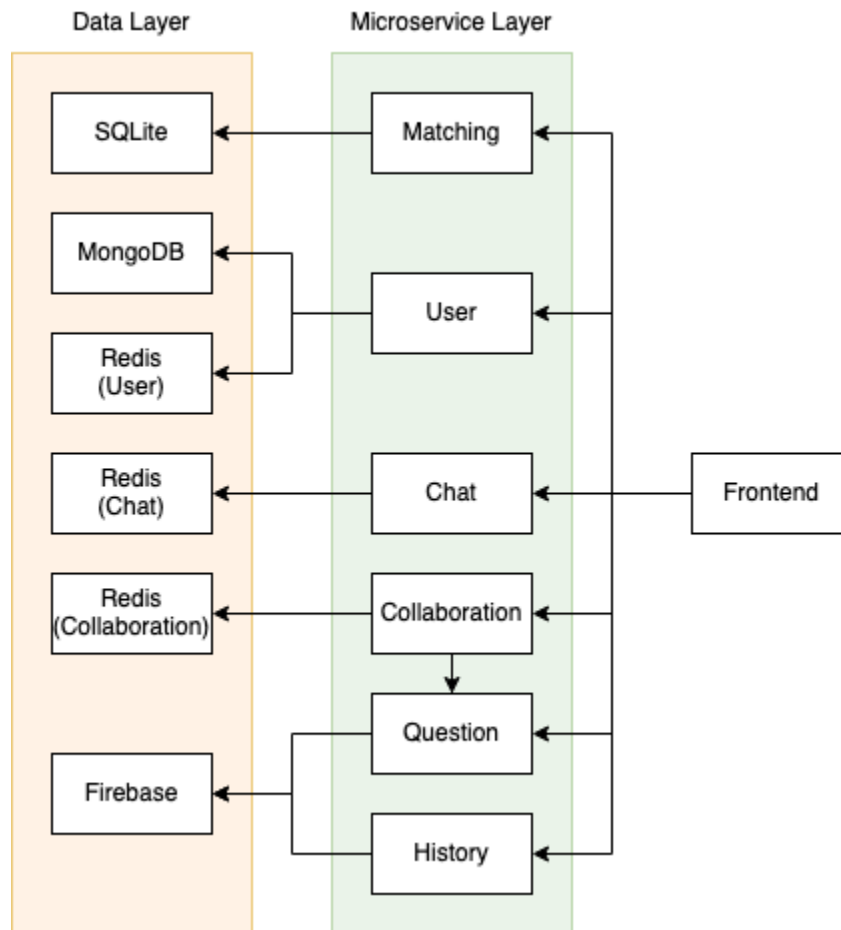
S/N	Non-functional Requirement	Priority
3.1	The application has at least 99% uptime.	High

According to Google Cloud Platform, Mongo Atlas, Firebase and Redis Cloud, there is high certainty of >99% uptime using their services with the corresponding software's configurations. This in turn ensures that our system, which is deployed to Google Cloud Platform and uses the respective cloud databases, is also available more than 99% of the time.



## 4. Architectural Design

### 4.1. Overview



Explanation:

1. A microservice architecture allows segregation of the functional requirements of the application into several domains.
  - a. Easier separation of work, can assign the responsibility of implementing each service to each team member
  - b. Independently deployed microservices makes the application more resilient. For example, even if the matching service is down, users can still login to their accounts provided that the user service is running.
2. A star topology with the frontend in the centre ensures loose coupling between services in the microservices layer.
  - a. Our backend microservices do not need to communicate with one another to support the functional requirements.
  - b. An exception is that the collaboration service calls the question service to get a random question identifier given a question difficulty. (See Section 4.2)

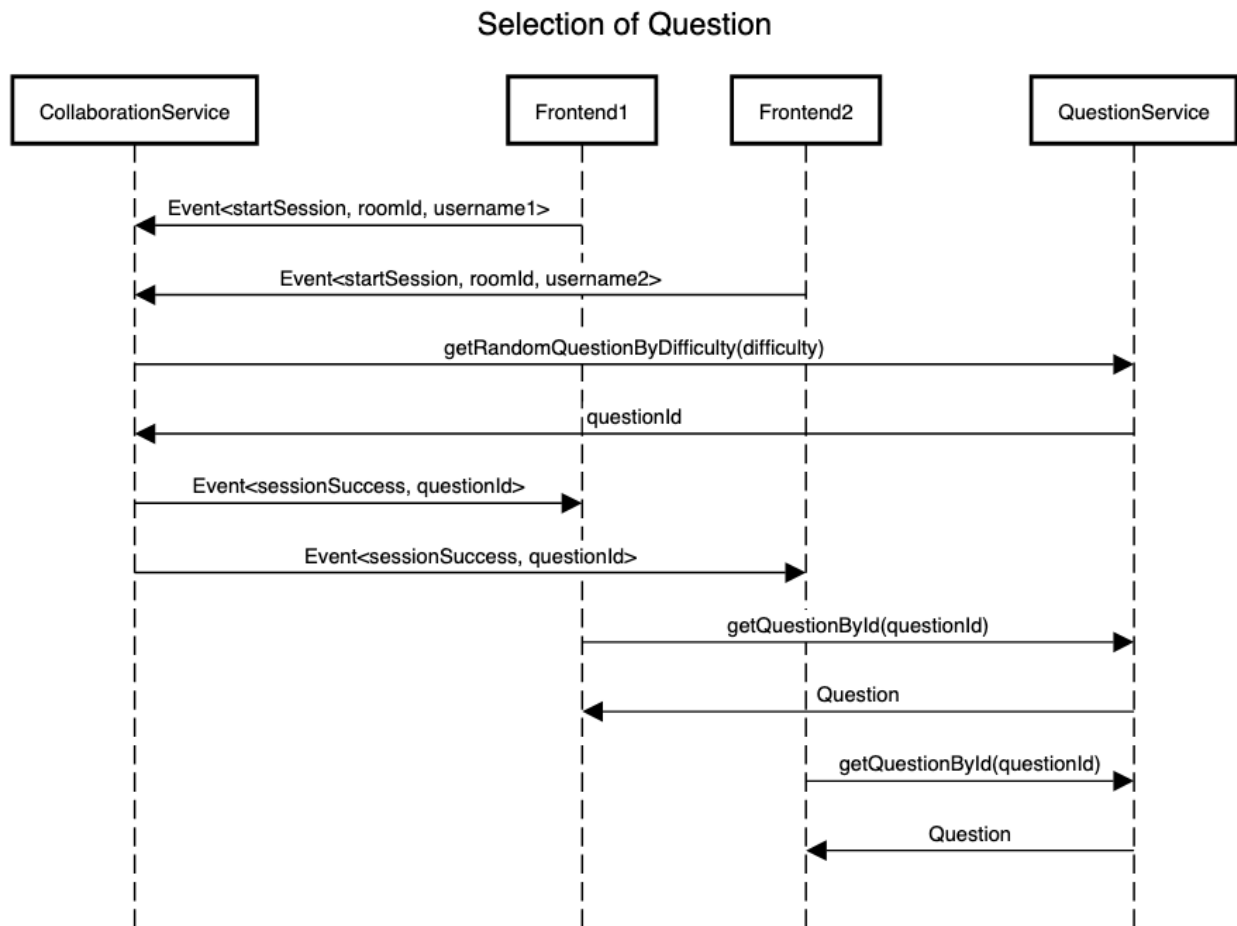
3. Each microservice uses its individual database to further decouple the microservices from one another, making the application more resilient.
  - a. Although the User, Collaboration and Chat services all use Redis, they each use their own server because they use different configurations that suit their needs.
  - b. An exception is the Firebase store that is shared between the Question and History services. (See Section 4.2)

## 4.2. Architectural Decisions

### Decision: Collaboration service calls Question service

The sequence diagram below describes our solution.

- When the Collaboration service requests a random question, the Question service responds with just the questionId instead of the entire question object.
- The Collaboration service then communicates the questionId to the two Frontend clients who then individually request the same question using the given identifier.



Benefits	Costs
<ul style="list-style-type: none"><li>• Reduce as much coupling between the Collaboration and Question services as possible.</li></ul>	<ul style="list-style-type: none"><li>• More API calls to the Question service</li></ul>

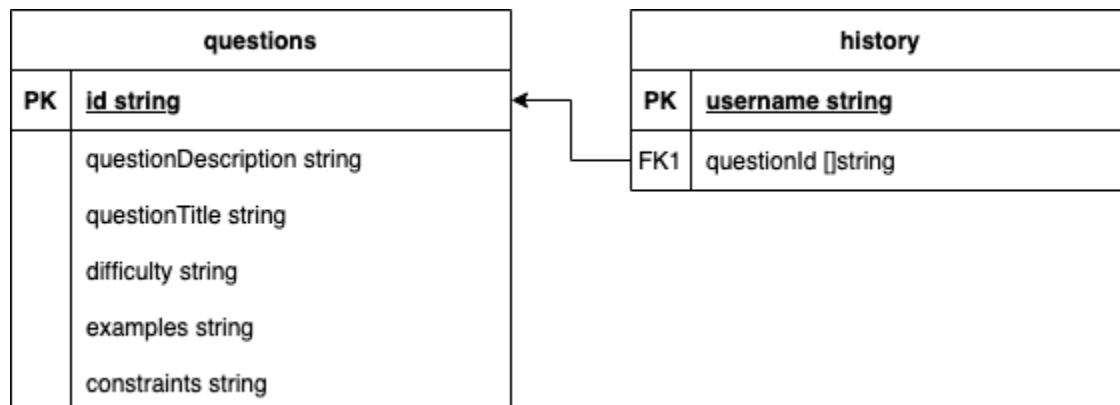
Alternatives:

- Collaboration service fetches and returns the entire question object
  - Fewer API calls to the backend
  - We believe the tradeoff is not worth the cost of increased coupling. The Collaboration service should not know what a question object looks like.
- Frontend fetches the random question from the Question service without involving the Collaboration service.
  - Results in two most likely different questions, one fetched from each user in the room.
  - According to our functional requirements, the same question should be relayed to both users in the same room, so this alternative does not work.

## Decision: Sharing a database between Question and History services

The entity relationship diagram below explains the structure of the Firebase cloud database that is shared between the Question and History services.

- A record in the history collection corresponds to the history of a user
- This history is represented as an array of question identifiers of the questions that they have previously attempted.



Benefits	Costs
<ul style="list-style-type: none"><li>• Fetching of a user's history will not involve the Question service at all</li><li>• Separating the Question and History services leaves room for History service to implement future features such as analytics.</li></ul>	<ul style="list-style-type: none"><li>• Increased coupling between Question and History services</li><li>• Potential duplicated code between Question and History services</li></ul>

Alternatives:

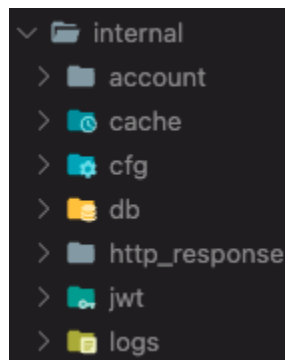
- History service uses its own database.
  - Decrease coupling between the Question and History services.
  - Either the History service or Frontend would have to call the Question service to fetch the actual question objects using the identifiers in the history collection.
- Combine Question and History services.
  - Solves most of the costs described above
  - Future extension of the service to support new features might result in it becoming bloated.

## 5. Design Patterns

### 5.1 MVC Pattern (Model View Controller)

The User Service utilizes the MVC pattern to separate:

- reading of and responding to HTTP requests (view)
- logic of user authentication (controller)
- and database schema and actions (model)



This is done by arranging code into packages in Golang, as shown in the folder structure above.

- `http_response` package abstracts writing a response to a HTTP request (view)
- `account` package implements the business logic for user authentication (controller)
- `db` package contains the schema of the account models that are stored in the database (model)

Redis was used as a cache to store invalid JWTs.

- Since JWT tokens are stored on the client side instead of on the server, the server is unable to invalidate the tokens.
- Our solution is to set an expiration duration of 24 hours on every JWT token and invalidate tokens by storing them in Redis with the same expiration duration when a user wants to log out.
- Subsequently, when a user tries to access an API endpoint, we do an additional check of whether their provided token is stored in Redis.
- This ensures that users who have logged out cannot use the same token to authenticate themselves.

## 5.2 MVA Pattern (Model View Adaptor)

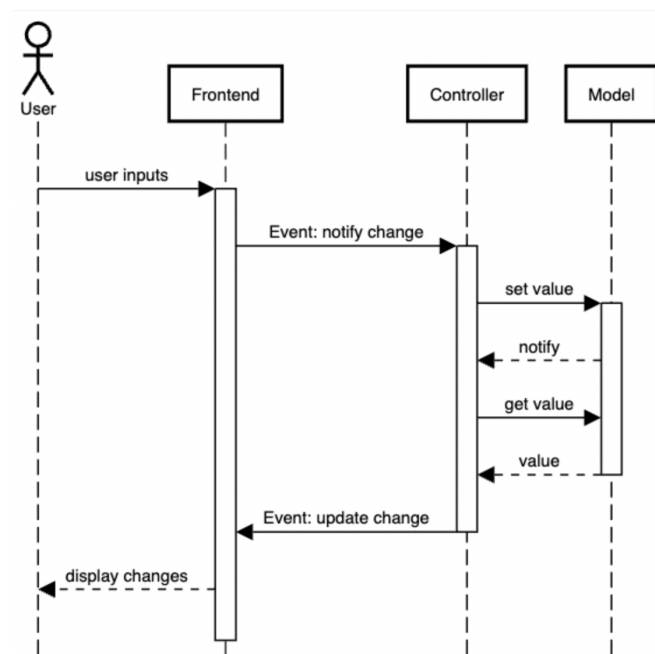
```

  collaboration-service
  collaboration-controller
  collaboration-model.js
  repository.js
  node_modules
  test
  repository.test.js
  .gitignore
  index.js
  package-lock.json
  package.json

```

The Matching, Collaboration and Chat services are designed with the MVA pattern, with the Model and Controller (Adaptor) being implemented at the backend and View being implemented at the frontend, as shown in the backend folder structure above.

Under this implementation, the Controller becomes a communication hub, accepting notifications from Model and user input events from the View, and that the Model and the View do not hold references to each other, and do not exchange data nor interact directly, as illustrated below:



Under this implementation, the View handles user input and illustrates data changes to the user. At the backend, the Controller handles incoming requests and business logic and delegates data-updating instructions to the Model, which in turn handles database schema and data

related logic. Once data is updated in the Model, the Model notifies the change to the Controller, who would then update the Frontend through an event.

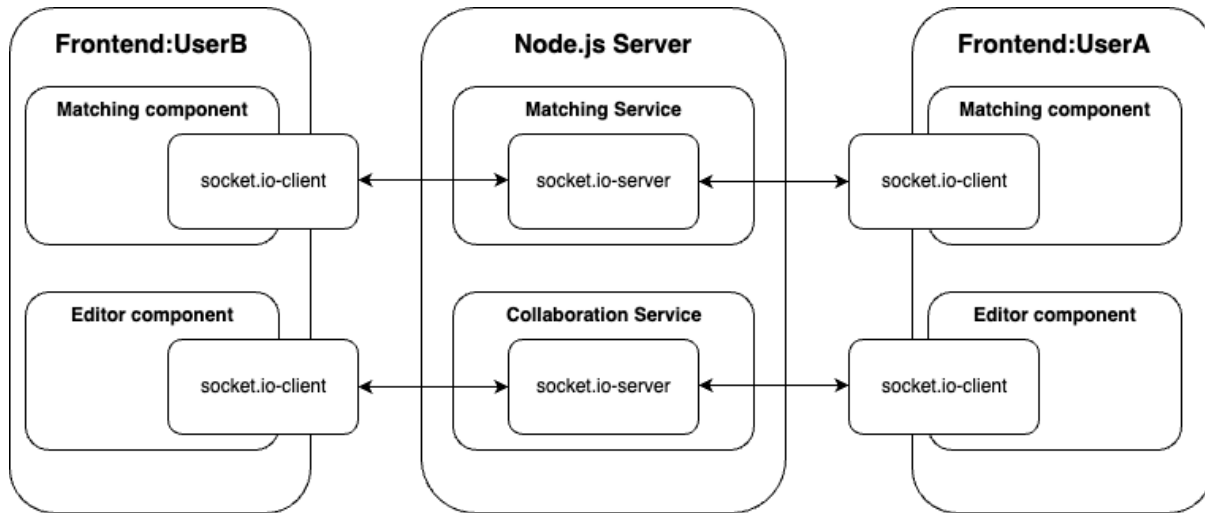
Benefits:

- Modularity ensures separation of concerns.
  - For example, the processing of user inputs, the maintenance of connection between different users, and the interactions with the database are all modularized into individual components.
- Facilitates extensibility
  - When a new form of user interaction is added, we only have to implement a new View-Controller pair, without needing to change the database logic in the Model.



## 5.3 Pub-Sub Pattern

We employed Socket-io to implement the real-time communication and event-driven approach for our functionality.



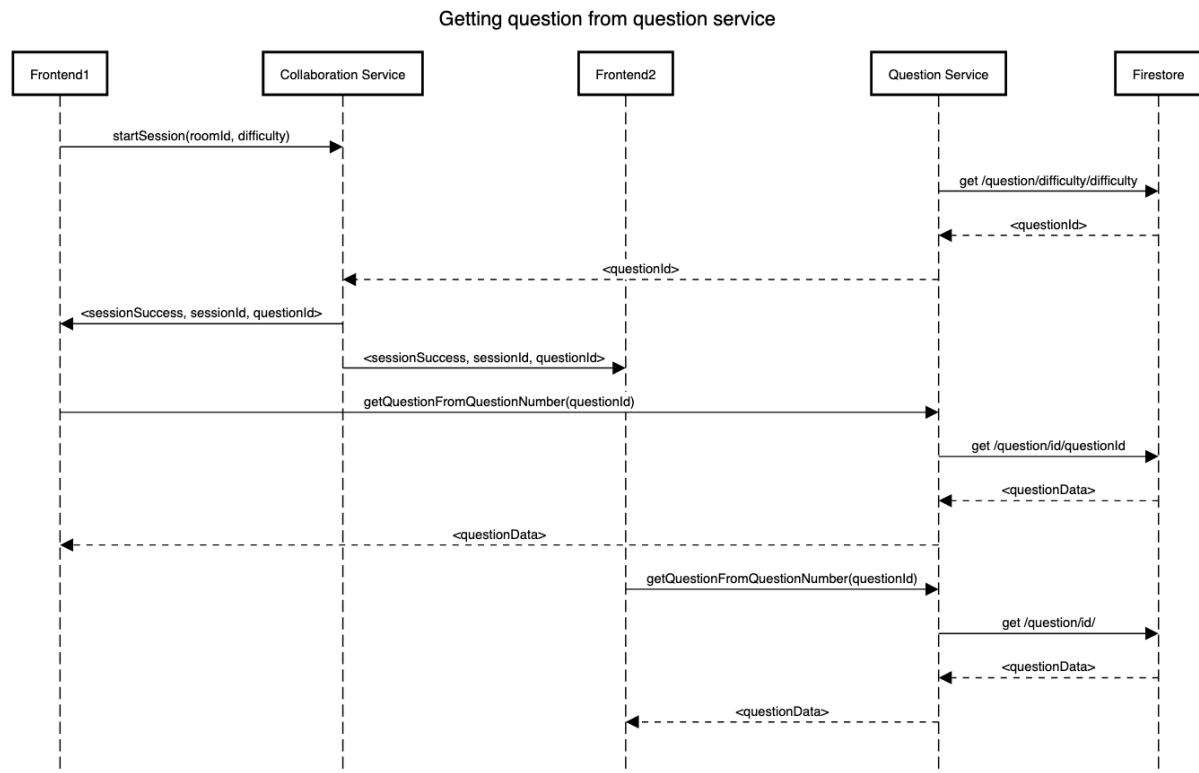
As illustrated above, on the server of each microservice, there is a Server socket handling and coordinating client connections. In the corresponding components in the frontend, there is Client socket subscribing to relevant Server socket to receive signals from the Server.

### Event Driven Communication

- The Server and Clients communicate via Events, which contains an event-header whose name is known to both Server and Clients, and a request body with an arbitrary amount of data decided by the sender of the Event.
- Both Server and Clients agree to a single definition for each event, but have their own interpretations on how to handle these events.
- Real-time collaboration between different users in a coding room is achieved. The server subscribes multiple clients to a single channel, such that messages could be distributed exclusively within the channel.

## 5.4 Database Per Service

We implemented the Database-per-service pattern, where each service has its own data store (as seen in section 4.1). Each service is unable to access data stores that they do not own. This allows microservices to remain loosely coupled, enabling independent development, deployment and scaling of each microservice. Each microservice can use different types of database to best suit their requirements. Changes to individual databases also do not impact other services, preventing a single point of failure and increasing the resilience of the application.



For example, the Collaboration service and Frontend call Question service APIs when access to the question database is required, instead of directly querying the question database.

There is an exception to the database per service pattern in the Question and History services which both use Firestore, a NoSQL cloud database to store question content information and user question history. Questions are stored in a “questions” collection, and user history is stored in a “history” collection. To reduce calls to the Question service, the History service also accesses the “question” collection to retrieve question titles for the corresponding question identifiers recorded in user history.

## 6. DevOps







### 6.1 Sprint Process

At the start of each Milestone, we assigned each member as the owner of a particular microservice/component, who are fully responsible for the planning, developing and testing of their respective scopes. For example, in Milestone2, we have owners of Question Service, Collaboration Service, Frontend and DevOps.

During each Milestone, we work with weekly sprints. On the Wednesday of each week, we conduct meetings to:

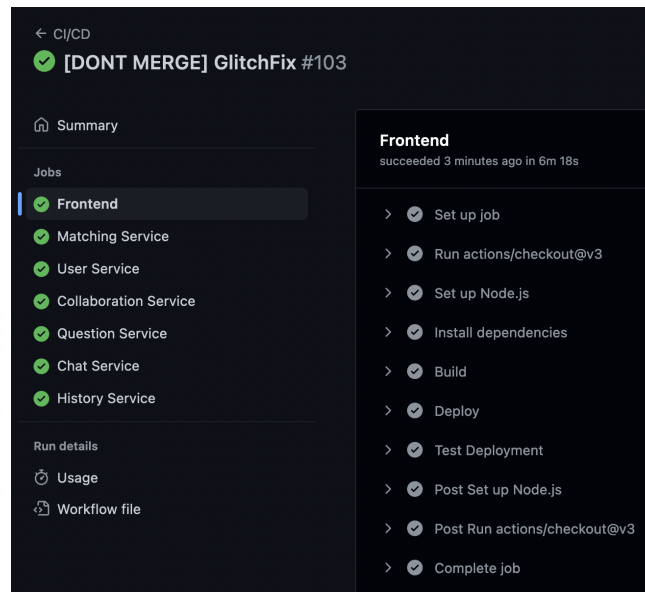
1. Update the team with each member's progress
2. Discussing the overall progress of the team
3. Discuss and decide what works are to be done for the upcoming sprint
4. Discuss relevant concerns and raise need for help

We keep track of decisions of each meeting through a shared Google Doc, and features to be done through GitHub Issues, for example:

 [Matching Service] FR2.5 - The system should provide a means for the user to leave a room once matched.	 1
#16 by xuanqi966 was closed on 19 Sep	
 [Matching Service] FR2.4 - The system should inform the users that no match is available if a match cannot be found within 30 seconds.	 1
#15 by xuanqi966 was closed on 19 Sep	
 [Matching Service] FR2.3 - If there is a valid match, the system should match the users within 30s.	 1
#14 by xuanqi966 was closed on 19 Sep	

## 6.2 CI/CD

We use Github Actions to trigger our Continuous Integration and Continuous Deployment workflows whenever a pull request is created/updated and whenever the main branch is updated. Each microservice has its own CI/CD workflow because different services have different dependencies.



Summary of steps found in all workflows:

- Set up runtime (Go or NodeJS)
- Install dependencies
- Build (Frontend and User services)
- Run tests
- Deploy to Google Cloud Platform
- Test deployment

<div>Actions</div> <div>New workflow</div> <div>All workflows</div> <div>CI/CD</div> <div>Management</div> <div>Caches</div>	<div> <div>✗ Merge pull request #36 from CS3219-AY2223S1/frontend/glitch...</div> <div>CI/CD #101: Commit 9a922f1 pushed by xuanqi966</div> <div>main</div> <div>1 hour ago</div> <div>11m 19s</div> <div>...</div> </div>
	<div> <div>✓ add logging statements</div> <div>CI/CD #100: Pull request #36 synchronize by Hazel1603</div> <div>frontend/glitchFix</div> <div>2 hours ago</div> <div>5m 25s</div> <div>...</div> </div>
	<div> <div>✗ add logging statements</div> <div>CI/CD #99: Pull request #36 opened by Hazel1603</div> <div>frontend/glitchFix</div> <div>2 hours ago</div> <div>11m 11s</div> <div>...</div> </div>
	<div> <div>✓ Merge pull request #35 from CS3219-AY2223S1/frontend/glitch...</div> <div>CI/CD #98: Commit cbe2f90 pushed by xuanqi966</div> <div>main</div> <div>2 hours ago</div> <div>5m 17s</div> <div>...</div> </div>
	<div> <div>✓ Frontend/glitchFix Attempt 2</div> <div>CI/CD #97: Pull request #35 synchronize by Hazel1603</div> <div>frontend/glitchFix</div> <div>2 hours ago</div> <div>4m 15s</div> <div>...</div> </div>
	<div> <div>✗ Frontend/glitchFix Attempt 2</div> <div>CI/CD #96: Pull request #35 synchronize by Hazel1603</div> <div>frontend/glitchFix</div> <div>3 hours ago</div> <div>4m 16s</div> <div>...</div> </div>
	<div> <div>✓ Frontend/glitchFix Attempt 2</div> <div>CI/CD #95: Pull request #35 opened by Hazel1603</div> <div>frontend/glitchFix</div> <div>3 hours ago</div> <div>5m 40s</div> <div>...</div> </div>
	<div> <div>✓ Benchmarking</div> <div>CI/CD #94: Pull request #34 synchronize by sc-arecrow</div> <div>benchmark</div> <div>12 hours ago</div> <div>5m 21s</div> <div>...</div> </div>
	<div> <div>✓ Benchmarking</div> <div>CI/CD #93: Pull request #34 synchronize by sc-arecrow</div> <div>benchmark</div> <div>13 hours ago</div> <div>5m 53s</div> <div>...</div> </div>

If any of the workflows fail, the pull request is not allowed to be merged into main. This makes our main branch (and hence our deployed application) more resistant to bugs. If all of the workflows pass, the changes are automatically deployed to Google Cloud Platform. This ensures that there is no delay in releasing new features to our users.

## 6.3 Deployment

### Local Development

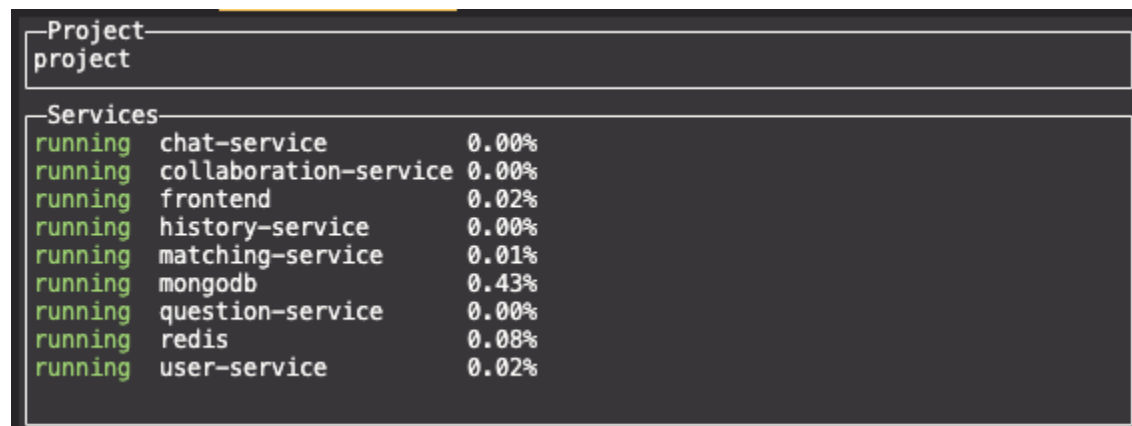
Port mapping on localhost:

Service	Port	Service	Port
Frontend	3000		
User	8000	Question	8003
Matching	8001	Chat	8004
Collaboration	8002	History	8005

We also use a local MongoDB server and Redis server, and a cloud database Firebase for the question service.

### Staging

Each microservice is dockerized following the instructions in their respective Dockerfiles. Docker compose is used to orchestrate the deployment of the containers.



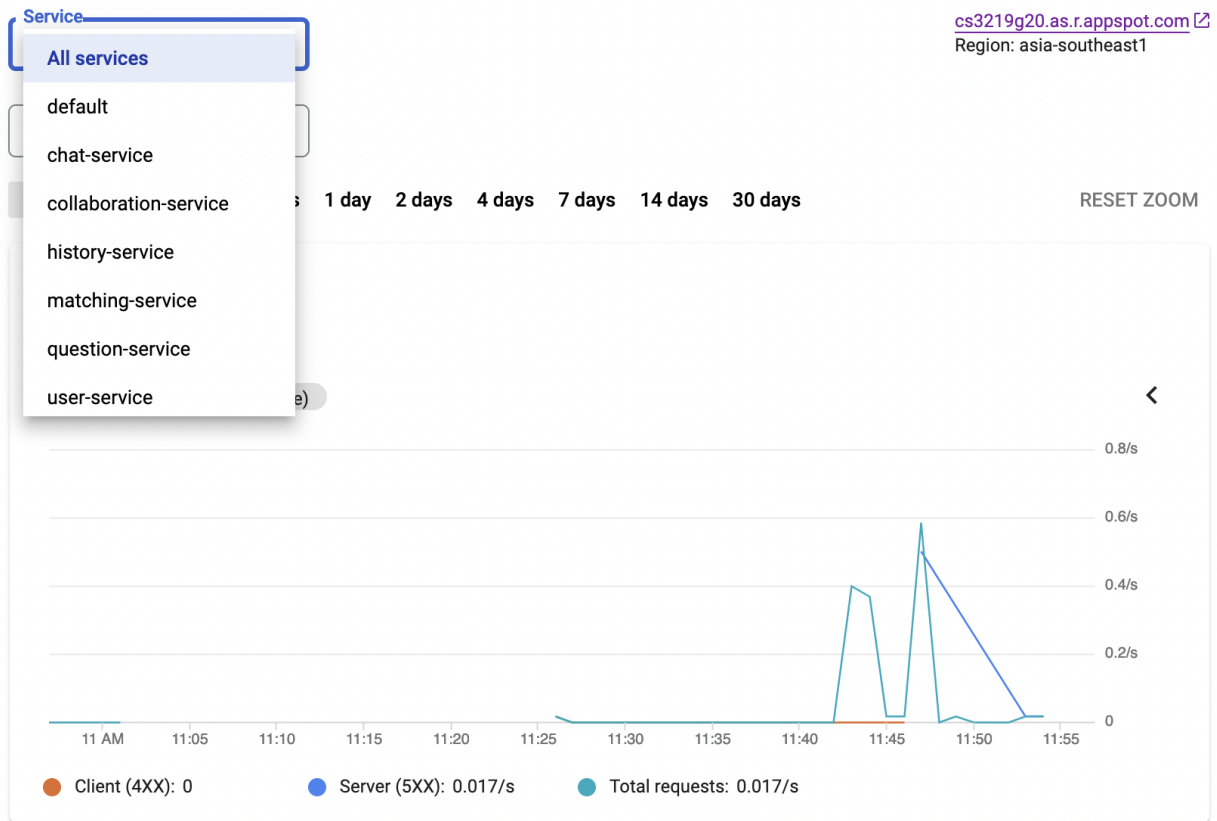
```
Project
project

Services
running chat-service      0.00%
running collaboration-service 0.00%
running frontend          0.02%
running history-service   0.00%
running matching-service  0.01%
running mongodb           0.43%
running question-service  0.00%
running redis             0.08%
running user-service      0.02%
```

We decided to use docker compose over Kubernetes because it is easier to use and provides simpler management of each container.

## Production

Each microservice is hosted as a service on Google App Engine, under the same project. Services will be able to communicate with one another via individually assigned URLs.



The application can be accessed at:  
<https://cs3219g20.as.r.appspot.com>

## 7. Application Design

Microservice	Stack	Explanation
User Service	Golang MongoDB Redis Cache	<ul style="list-style-type: none"><li>• Golang has great out-of-the-box solutions in their <code>net/http</code> library for writing web servers</li><li>• We chose MongoDB over SQL databases because in the future, user accounts may contain varying fields depending on their permissions and roles. A NoSQL database provides more freedom in terms of model schema.</li><li>• See previous section for Redis explanation</li></ul>
Matching Service Collaboration Service Chat Service	Javascript ExpressJS NodeJS Socket-io Sequelize-SQLite Redis Cache	<ul style="list-style-type: none"><li>• NodeJS provides powerful solution in building real-time web server with efficient package manage of npm</li><li>• Socket-io is suitable for the Pub-Sub Pattern and Event-driven implementation intended by our project</li><li>• Sequelize-SQLite provides object representation of data, facilitating efficient database operations through its abstractions</li></ul>
Question Service History Service	Javascript ExpressJS NodeJS Firestore NoSQL database	<ul style="list-style-type: none"><li>• NoSQL database allows for more flexibility in question content data fields</li><li>• Firestore has native SDKs that reduces development complexity</li></ul>
Frontend	React Material UI ESLint	<ul style="list-style-type: none"><li>• Virtual DOM feature that allows for quick rendering of UI</li><li>• Popular framework that has a lot of community packages, support and help</li><li>• Developer has prior experience in React</li></ul>



## 8. Backend

### 8.1 User Microservice

Deployment URL: <https://user-service-dot-cs3219g20.as.r.appspot.com>

The User service is responsible for handling user accounts and authentication.

POST	/accounts	Register a new user with the given credentials
Request body: { "username": string, "password": string }		Success status code: 201. Response body: { "jwt": string }
Other possible status codes: <ul style="list-style-type: none"><li>• 400: if request body is malformed</li><li>• 409: if requested username already exists</li><li>• 500: other server errors</li></ul>		

GET	/accounts/{username}	Get account details
Request body: { "jwt": string }		Success status code: 200. Response body: { "username": string }
Other possible status codes: <ul style="list-style-type: none"><li>• 400: if request body is malformed</li><li>• 401: if given JWT is invalid</li><li>• 404: if requested username (encoded in the JWT) does not exist</li><li>• 500: other server errors</li></ul>		

PUT	/accounts/{username}	Update password
Request body: { "jwt": string, "old_password": string, "new_password": string }		Success status code: 200. No response body.
Other possible status codes: <ul style="list-style-type: none"><li>• 400: if request body is malformed or if provided old password is incorrect</li><li>• 401: if given JWT is invalid</li><li>• 404: if requested username (encoded in the JWT) does not exist</li><li>• 500: other server errors</li></ul>		

DELETE	/accounts/{username}	Delete account
Request body: <pre>{   "jwt": string }</pre>		Success status code: 200. No response body.
Other possible status codes: <ul style="list-style-type: none"> <li>• 400: if request body is malformed</li> <li>• 401: if given JWT is invalid</li> <li>• 404: if requested username (encoded in the JWT) does not exist</li> <li>• 500: other server errors</li> </ul>		

POST	/login	Login with the given credentials
Request body: <pre>{   "username": string,   "password": string }</pre>		Success status code: 200. Response body: <pre>{   "jwt": string }</pre>
Other possible status codes: <ul style="list-style-type: none"> <li>• 400: if request body is malformed</li> <li>• 401: if given JWT is invalid</li> <li>• 404: if requested username (encoded in the JWT) does not exist</li> <li>• 500: other server errors</li> </ul>		

POST	/logout	Logout
Request body: <pre>{   "jwt": string }</pre>		Success status code: 200. No response body.
Other possible status codes: <ul style="list-style-type: none"> <li>• 400: if request body is malformed</li> <li>• 401: if given JWT is invalid</li> <li>• 500: other server errors</li> </ul>		

## 8.2 Question Microservice

Deployment URL: <https://question-service-dot-cs3219g20.as.r.appspot.com>

The question service is responsible for storing and retrieving questions.

GET	/questions	Get all questions
-----	------------	-------------------

No request body	Success status code: 200. Response body: [ questionId string ]
Other possible status codes: <ul style="list-style-type: none"> <li>400: if no questions were retrieved</li> </ul>	

GET	/questions/difficulty/{difficulty}	Get all questions of a given difficulty
No request body		Success status code: 200. Response body: [ questionId string ]
Other possible status codes: <ul style="list-style-type: none"><li>• 400: if no questions were retrieved</li></ul>		

GET	/question/difficulty/{difficulty}	Get a random question of a given difficulty
No request body		Success status code: 200. Response body: questionId string
Other possible status codes: <ul style="list-style-type: none"><li>400: if no questions were retrieved</li></ul>		

GET	/question/id/{id}	Get question content of a particular question ID
No request body		Success status code: 200. Response body: { "constraints": string, "difficulty": string, "questionDescription": string, "questionTitle": string, "examples": string }
Other possible status codes: <ul style="list-style-type: none"><li>404: if no questions were found with the given ID</li></ul>		

## 8.3 History Microservice

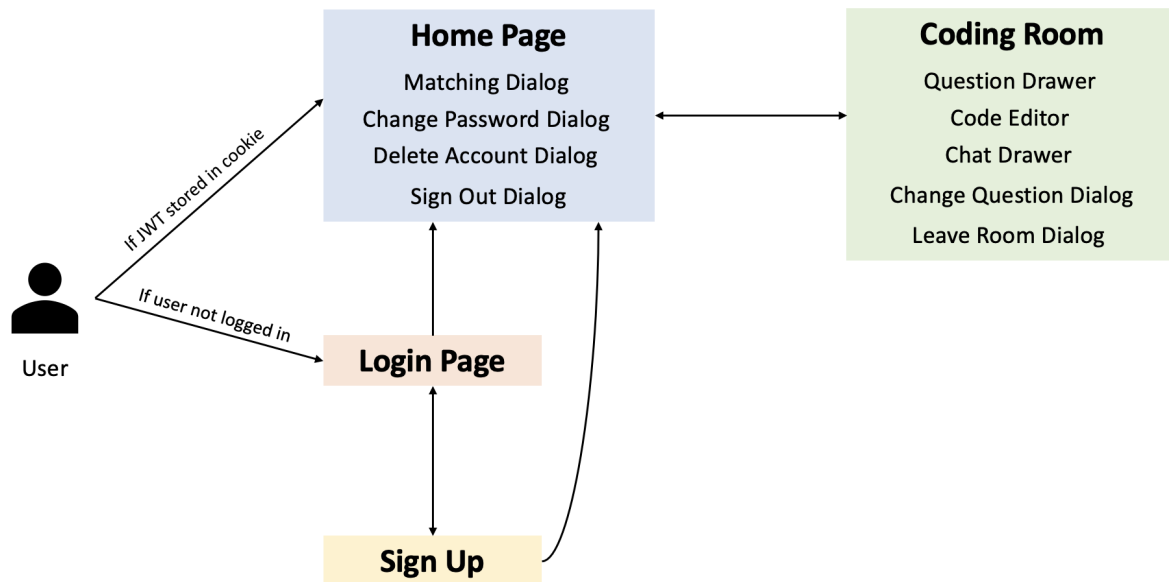
Deployment URL: <https://history-service-dot-cs3219g20.as.r.appspot.com>

The history service is responsible for recording and retrieving past question attempts of users.

POST	/history	Updates user question attempt history
Request body: <pre>{   "username": string,   "questionId": string }</pre>		Success status code: 200. No response body.

GET	/history/{username}	Retrieves a given user's question attempt history
No request body		Success status code: 200. Response body: <pre>[   questionId string ]</pre>
Other possible status codes: <ul style="list-style-type: none"> <li>• 400: if given username is invalid/does not have an existing question history</li> </ul>		

## 9. Frontend



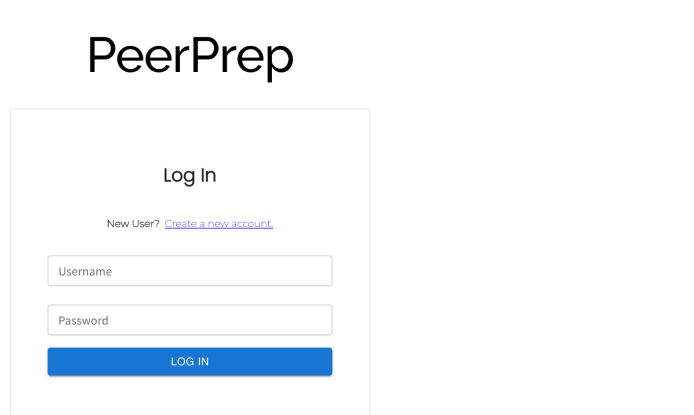
Our team uses **React** framework to build our application. React is an open-source JavaScript framework that is used for building fast, scalable user interfaces simply for both single-page applications & progressive web apps.

The fundamental frontend library we used was **Material UI**. Material UI provides a wide diversity of components, each being highly customisable without compromising on the functionality. This allows our application to be well-styled and responsive.

The **App** component serves as a **controller** class, which routes to 4 pages - Login, Sign Up, Home Page and Coding Room. To allow easy login, we also created our own Route component, which checks if a user is logged in. If they are logged in, they will be automatically redirected to the Homepage without having to login again.

# 10. Application Screenshots

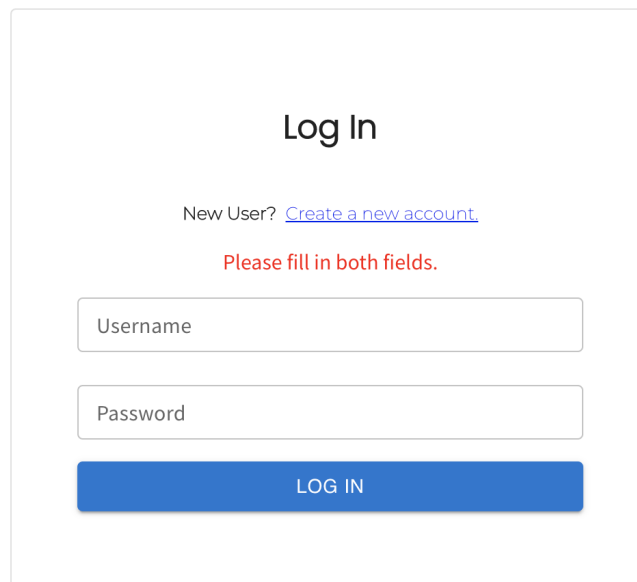
## 10.1 User Authentication



The screenshot shows the PeerPrep login interface. At the top, the text "PeerPrep" is displayed in a large, bold, black font. Below this, the heading "Log In" is centered. Underneath the heading, there is a link that says "New User? [Create a new account.](#)". Below the link are two input fields: "Username" and "Password". At the bottom of the form is a blue button with the text "LOG IN" in white capital letters.

Figure: Login Page

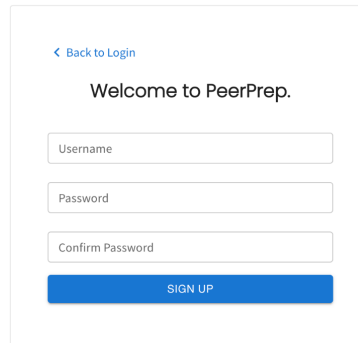
From the Login Page, users can click on the 'Create a new account' hyperlink to navigate to the Sign Up page. If the user tries to login using the wrong password or a non-existent username, the frontend will display the error message to the user.



This screenshot shows the same PeerPrep login interface as the previous one, but with an error message. The heading "Log In" and the link "New User? [Create a new account.](#)" are still present. Below the link, the error message "Please fill in both fields." is displayed in red text. Below the error message are the "Username" and "Password" input fields. At the bottom is the blue "LOG IN" button.

Figure: Login Page with Error Message

# PeerPrep

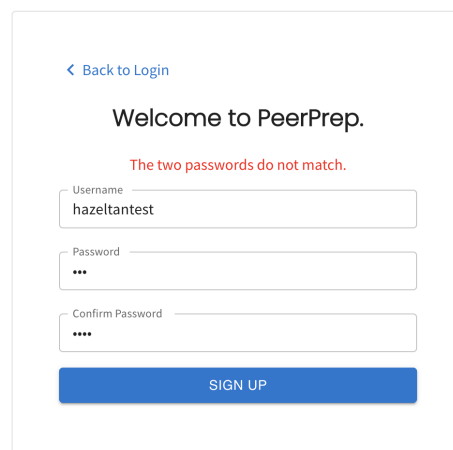


The image shows a web form for signing up on PeerPrep. At the top left is a blue link '< Back to Login'. Below it is the text 'Welcome to PeerPrep.' followed by three input fields: 'Username', 'Password', and 'Confirm Password'. At the bottom is a blue button labeled 'SIGN UP'.

Figure: Sign Up Page

Users can return back to the Login page by clicking on the 'Back to Login' hyperlink. Before sending the request to the User microservice, the frontend validates that all fields are filled and the passwords are the same. The User service validates that the username is new. The frontend shows different error messages based on the errors.

Errors	Error Message displayed
Fields not filled (Frontend validation)	Please fill in all fields.
New passwords not the same (Frontend validation)	The two passwords do not match.
Status code = 409 (from User Service)	This username already exists.



The image shows the same sign-up form as before, but with an error message. Above the 'Password' field, the text 'The two passwords do not match.' is displayed in red. The 'Username' field contains the text 'hazeltantest'. The 'Password' field has three dots, and the 'Confirm Password' field has four dots. The 'SIGN UP' button is still at the bottom.

Figure: Example of Sign Up page with error message

Once the user logs in, their jwtToken will be saved as a cookie. To increase usability, we wrote a custom route on React so that users will be automatically redirected from /login or /signup to /landing if they are already logged in. Conversely, if users are not logged in and they try to enter the Home Page or Coding Room, they will also be redirected to the login page.

```
export const PrivateRoute = () => {  
  const auth = isAuthenticated()  
  
  if (!auth) {  
    return <Navigate to="/login" />  
  } else {  
    return <Outlet />  
  }  
}
```

Figure: PrivateRoute for automatic redirection

```
<Router>  
  <Routes>  
    <Route exact path="/" element={<Navigate replace to="/login" />}</Route>  
    <Route path="/signup" element={<SignupPage/>}/>  
    <Route path="/login" element={<LoginPage/>}/>  
  
    <Route exact path="/landing" element={<PrivateRoute/>}>  
      <Route exact path="/landing" element={<LandingPage/>}/>  
    </Route>  
    <Route exact path="/room" element={<PrivateRoute/>}>  
      <Route exact path="/room" element={<CodingRoom/>}/>  
    </Route>  
  </Routes>  
</Router>
```

Figure: Router Component that uses PrivateRoute



## 10.2 User Home Page

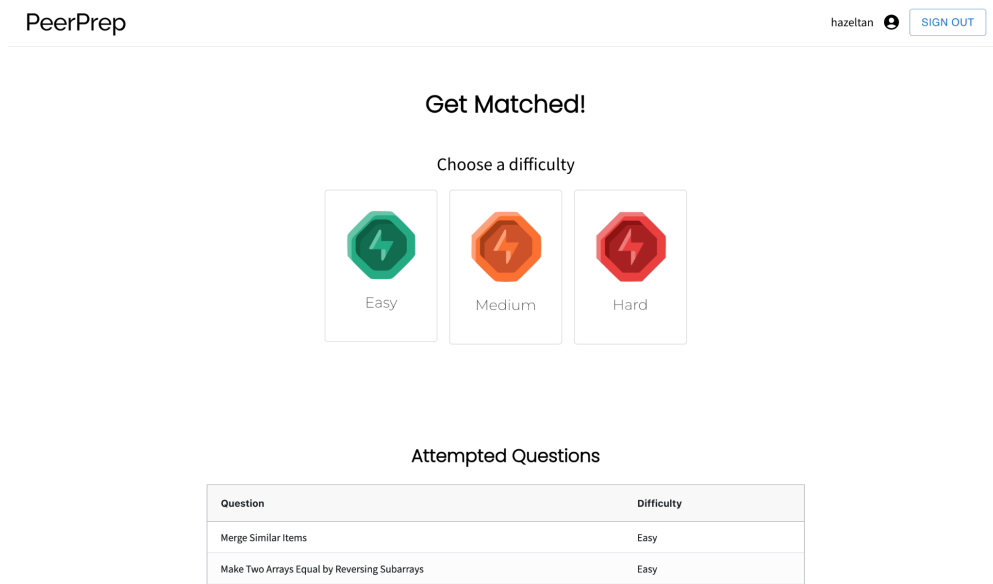


Figure: User Home Page

The user home page has 2 parts - the Matching section and the Attempted Questions sections. We want to make matching easy for users, so the buttons are placed right in the middle. The Attempted Questions section allows them to see the questions that they have attempted before. It can be sorted according to difficulty.

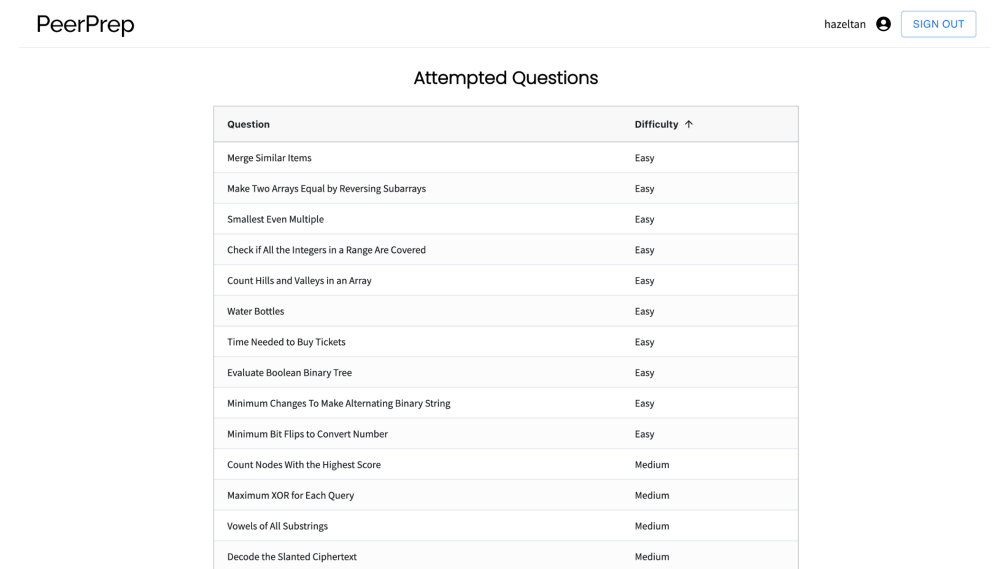


Figure: Attempted Questions, sorted by Difficulty

## 10.3 User Actions

On the Home Page, users can delete their account, change their password and sign out.

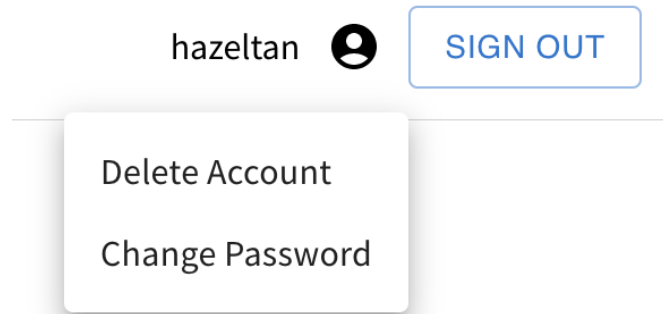


Figure: User actions available on top right of Home Page

### 10.3.1 Delete Account

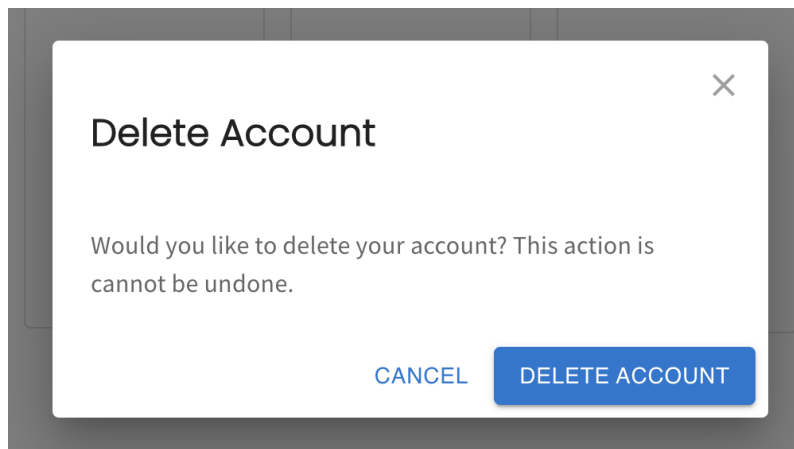
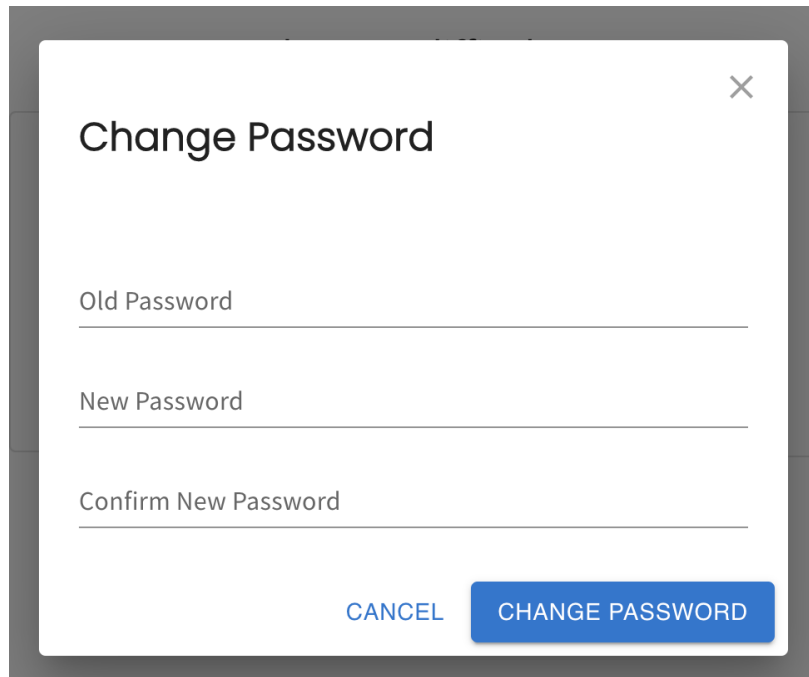


Figure: Delete Account Dialog

Once users delete their account, they will be redirected back to the Login Page, where they can navigate to the Sign Up page to create a new account.

### 10.3.2 Change Password

A modal dialog box titled "Change Password" with a close button (X) in the top right corner. It contains three text input fields labeled "Old Password", "New Password", and "Confirm New Password". At the bottom, there are two buttons: "CANCEL" and "CHANGE PASSWORD".

Change Password

Old Password

New Password

Confirm New Password

CANCEL CHANGE PASSWORD

Figure: Change Password Dialog

The dialogs validates that:

- All fields are filled and
- The new passwords are the same

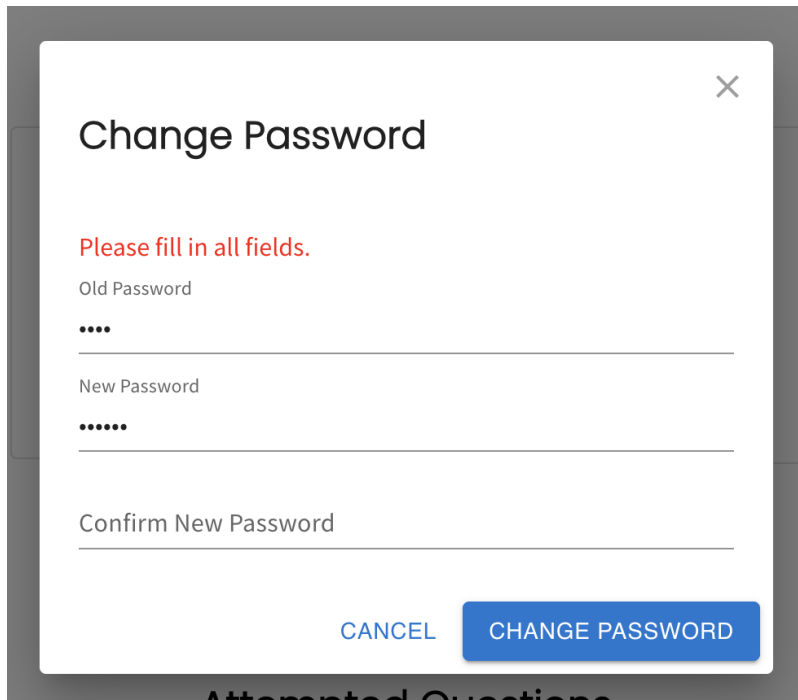
before sending the request to the User Microservice.

The User Microservice checks that:

- The username exists
- The old password is correct

If the username is incorrect or the old password is wrong, the frontend will display the corresponding error message.

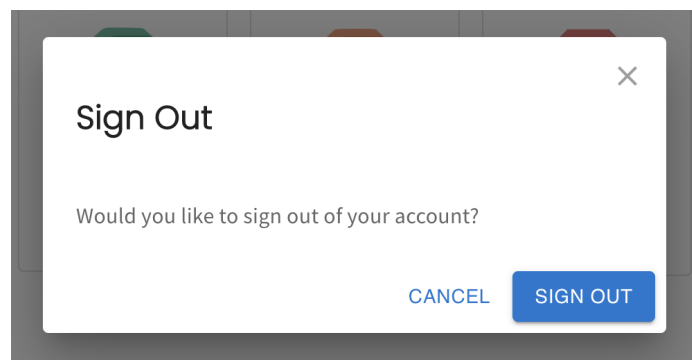
Errors	Error Message displayed
Fields not filled (Frontend validation)	Please fill in all fields.
New passwords not the same (Frontend validation)	The two passwords do not match.
unknown username (from User Service)	Username is incorrect
incorrect password (from User Service)	The old password is incorrect.



A screenshot of a 'Change Password' dialog box. The dialog has a title bar with a close button (X) in the top right corner. The main content area contains the title 'Change Password' followed by a red error message: 'Please fill in all fields.' Below this, there are three input fields: 'Old Password' (with four dots), 'New Password' (with six dots), and 'Confirm New Password' (with six dots). At the bottom right, there are two buttons: a blue 'CANCEL' button and a blue 'CHANGE PASSWORD' button.

Figure: Example of Dialog with error displayed

### 10.3.3 Sign Out



A screenshot of a 'Sign Out' dialog box. The dialog has a title bar with a close button (X) in the top right corner. The main content area contains the title 'Sign Out' followed by the question: 'Would you like to sign out of your account?'. At the bottom right, there are two buttons: a blue 'CANCEL' button and a blue 'SIGN OUT' button.

Figure: Sign Out Dialog

After users confirm, they will be redirected to the Login Page. Their jwtToken will also be removed.

## 10.4 Matching System

Upon selecting a difficulty between **Easy**, **Medium** and **Hard**, a modal will appear to inform users that they are currently being matched. The loader's Progress circle decreased according to the time left, giving users a visual indication of the progress. Should users want to leave the matching, they can click anywhere outside the dialog.

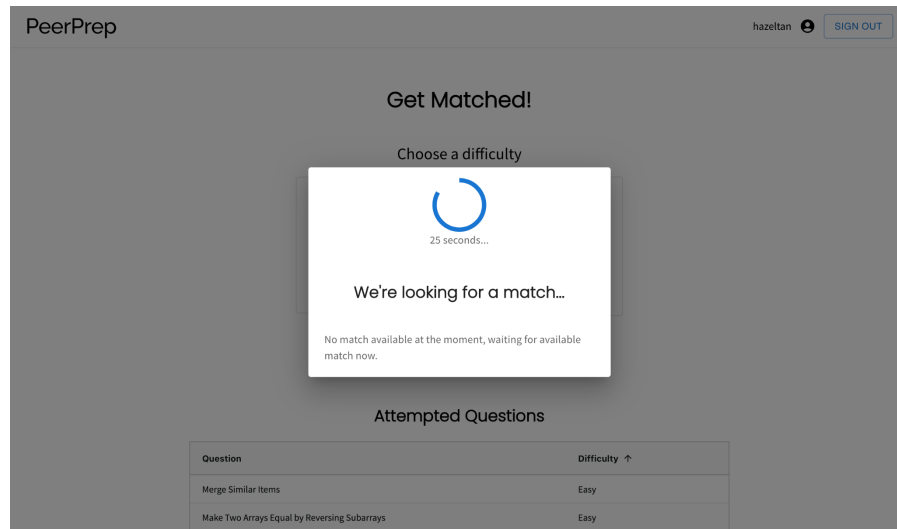


Figure: Matching Dialog

In the case where a match cannot be found within 30s, the user will be notified and be prompted to try again later.

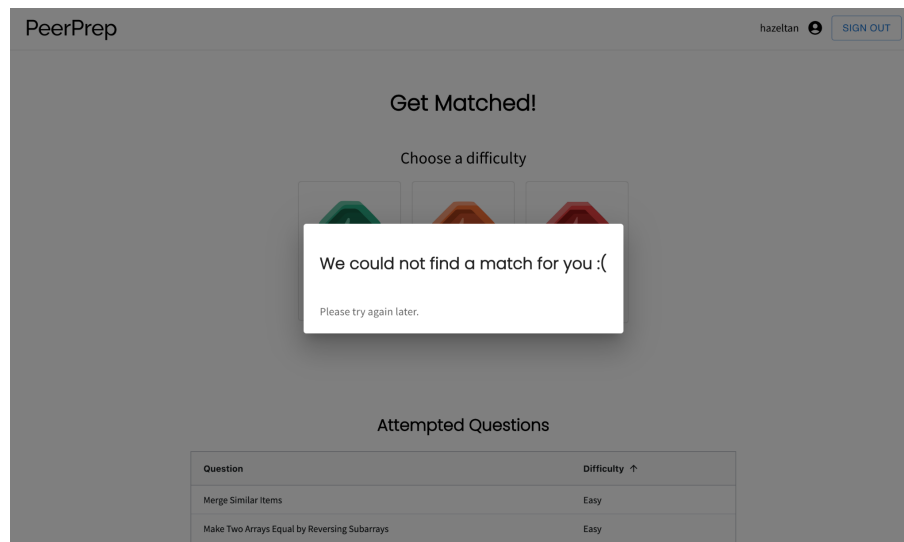


Figure: Match Failed

If the user was successfully matched, they will receive a confirmation dialog and be redirected to the coding room after 2 seconds.

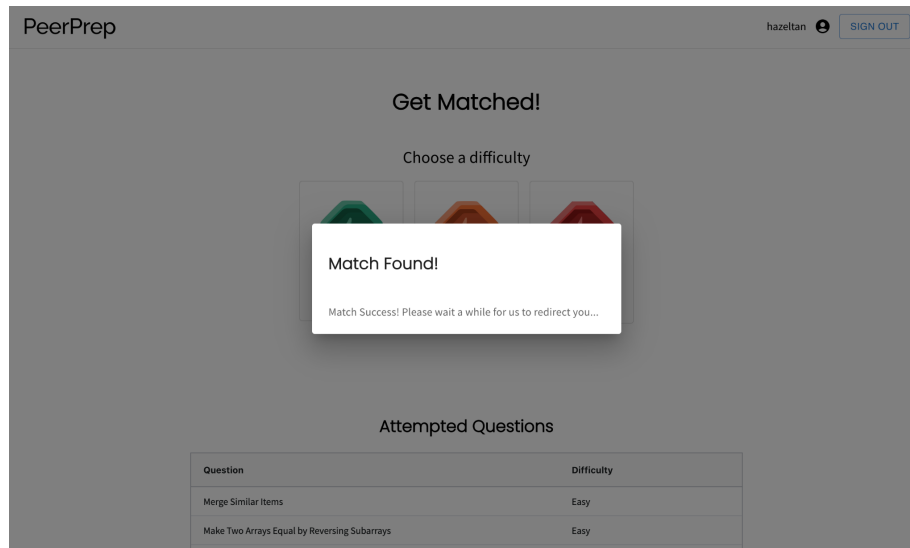


Figure: Match Success

Here is a high-level Activity Diagram for Matching:

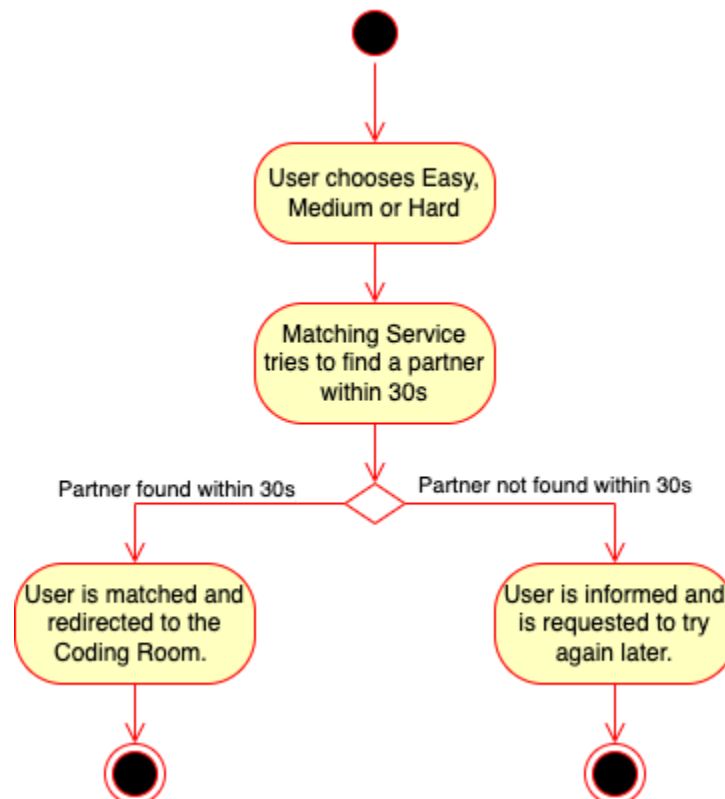


Figure: Matching Activity Diagram

## 10.5 Coding Room

Once the match is successful, users will be routed to the coding room, where they will be assigned a random question of the selected difficulty. There are 3 sections to the coding room: the Question Section, the real-time Code Editor and the real-time Chat.

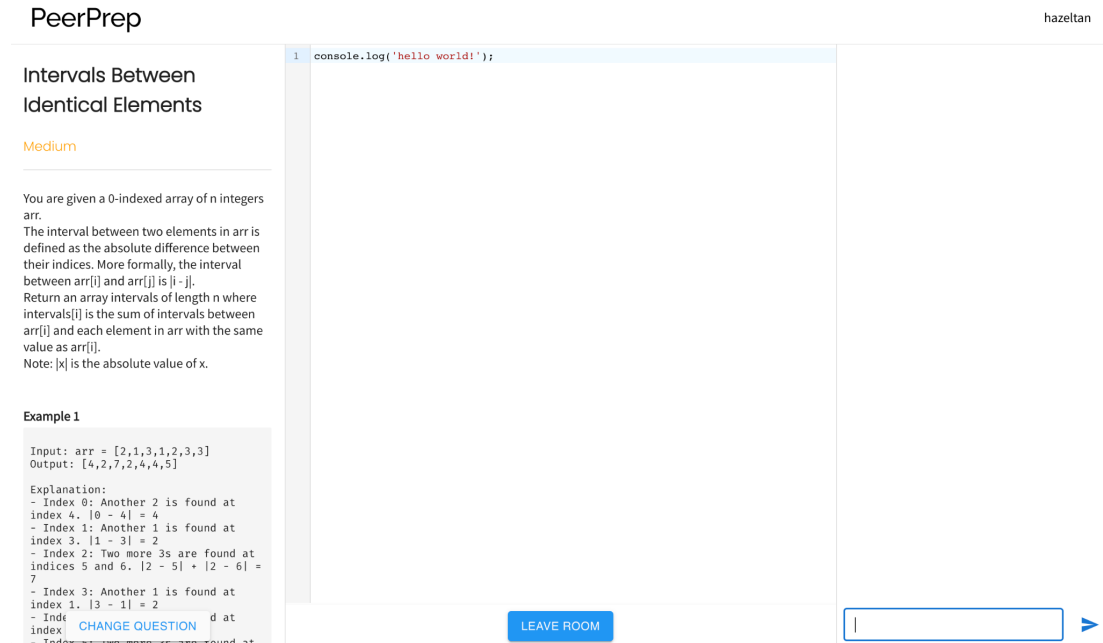


Figure: Coding Room upon initialisation

Starting a session requires the frontend to communicate with Collaboration Service to retrieve the questions and support the real-time code editor.

### Start Session (Collaboration Service)

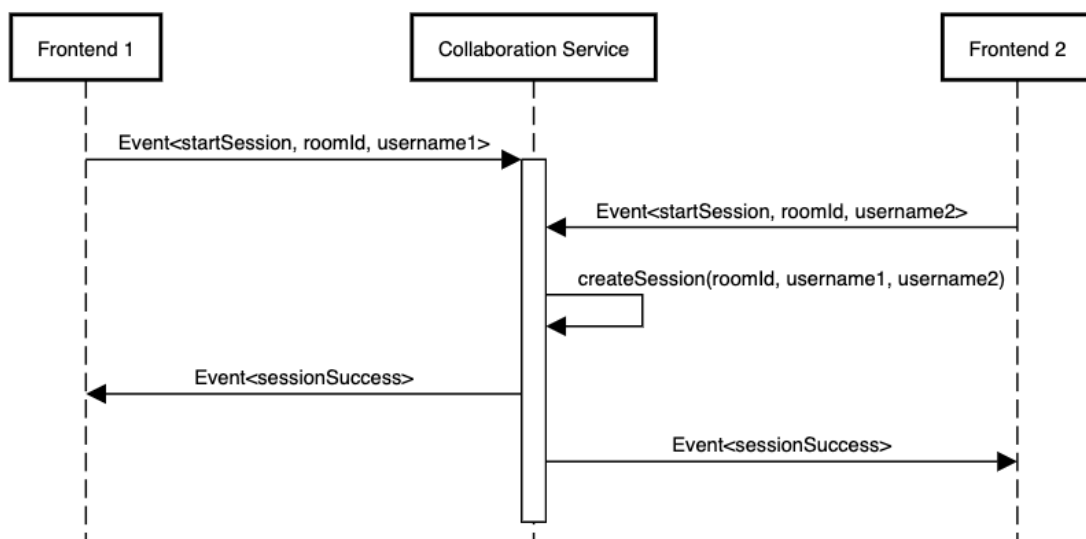


Figure: Start Session Sequence Diagram (Collaboration Service)

It also needs to start a session with the Chat Service for the real-time messaging functionality.

### Start Session (Chat Service)

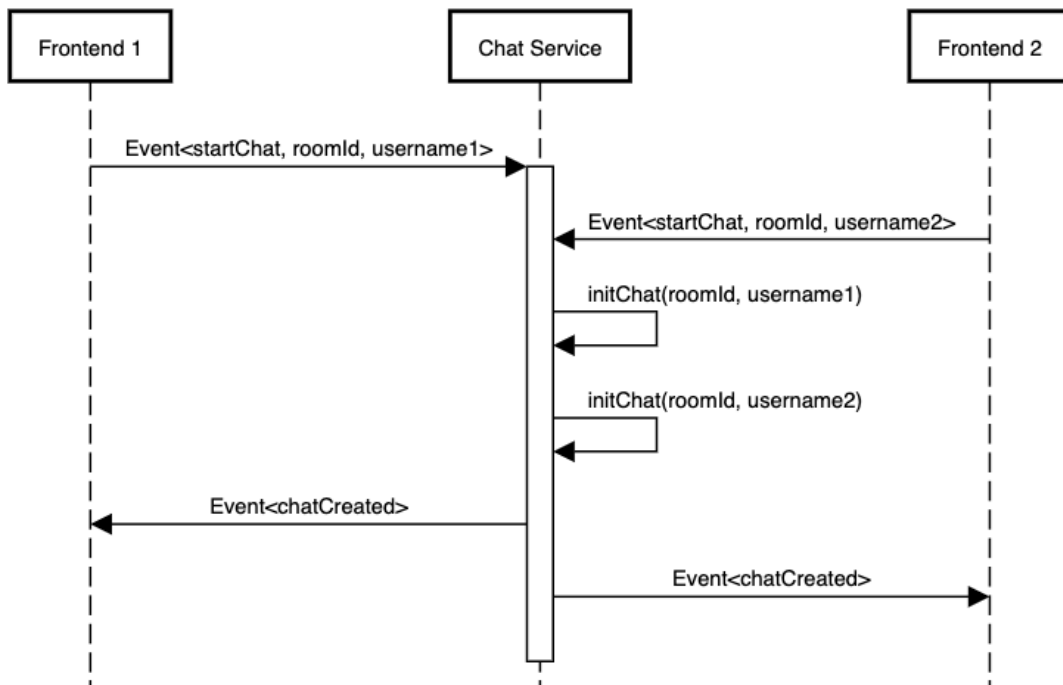


Figure: Start Session Sequence Diagram (Chat Service)



## 10.6 Change Question

To give users flexibility, we allow them to change questions during the session.

This functionality can be triggered by clicking on the 'Change Question' button in the Question section.

PeerPrep

Intervals Between Identical Elements

Medium

You are given a 0-indexed array of  $n$  integers `arr`.

The interval between two elements in `arr` is defined as the absolute difference between their indices. More formally, the interval between `arr[i]` and `arr[j]` is  $|i - j|$ .

Return an array `intervals` of length  $n$  where `intervals[i]` is the sum of intervals between `arr[i]` and each element in `arr` with the same value as `arr[i]`.

Note:  $|x|$  is the absolute value of  $x$ .

Example 1

```
Input: arr = [2,1,3,1,2,3,3]
Output: [4,2,7,2,4,4,5]
```

Explanation:

- Index 0: Another 2 is found at index 4.  $|0 - 4| = 4$
- Index 1: Another 1 is found at index 3.  $|1 - 3| = 2$
- Index 2: Two more 3s are found at indices 5 and 6.  $|2 - 5| + |2 - 6| = 7$
- Index 3: Another 1 is found at index 1.  $|3 - 1| = 2$
- Index 4: Another 2 is found at index 0.  $|4 - 0| = 4$
- Index 5: Another 3 is found at index 2.  $|5 - 2| = 3$
- Index 6: Another 3 is found at index 2.  $|6 - 2| = 4$

1

```
console.log('hello world!');
```

CHANGE QUESTION

Figure: Change Question Button

The Change Question functionality has a total of 3 steps. We will reference the user who initiated the change to be **Frontend 1** and his partner to be **Frontend 2**.

Step 1: **Frontend 1** confirms that he wants to change the question.

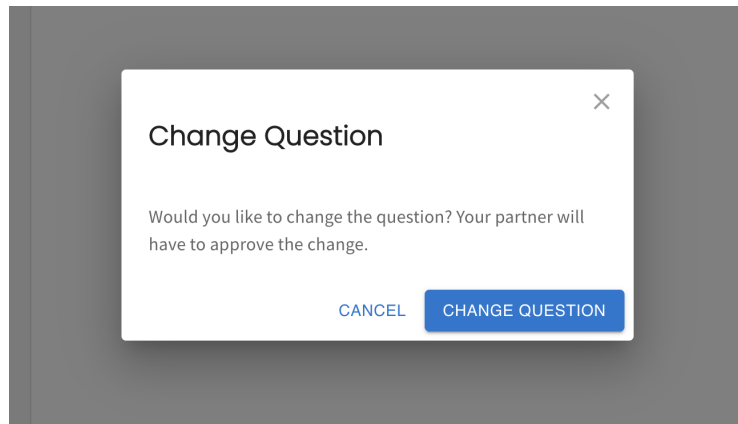


Figure: Frontend 1 confirmation dialog

Step 2: After Frontend 1 confirms, **Frontend 2** will be informed and they can decide if they want to approve the change in question.

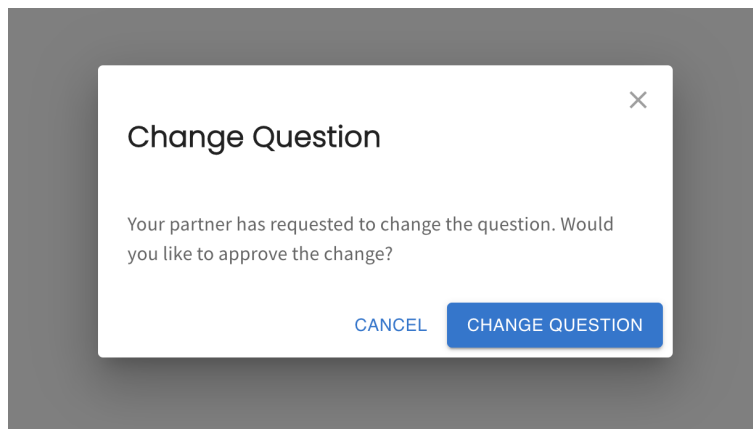


Figure: Frontend 2 approve dialog

While **Frontend 1** is waiting for approval, a dialog will be displayed.

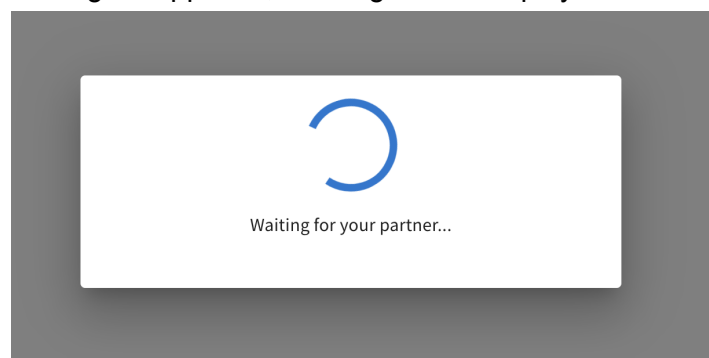


Figure: Frontend 1 waiting dialog

Step 3a: Once **Frontend 2** has approved the change, both users will be shown a confirmation dialog that a new question is being retrieved. The dialog will close once the new question has been loaded.

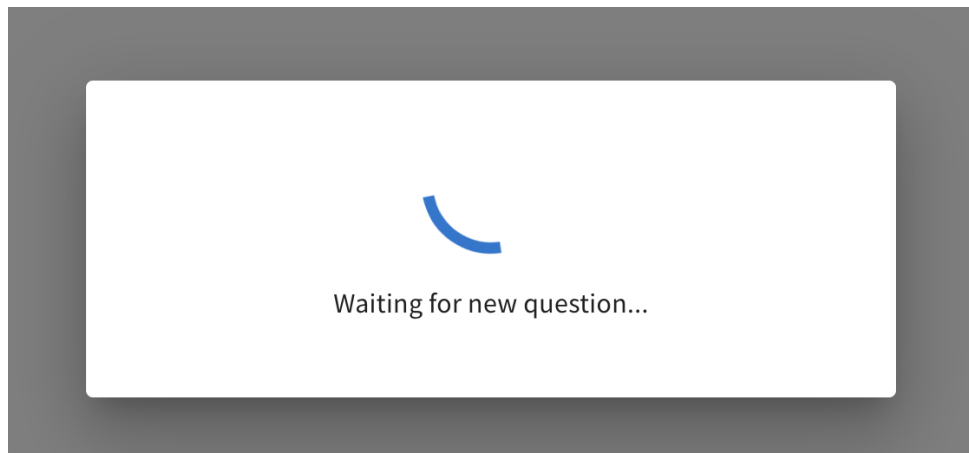


Figure: New Question Dialog

Step 3b: If **Frontend 2** decides that they do not want to change the question, **Frontend 1** will be informed that their partner rejected the change in question.

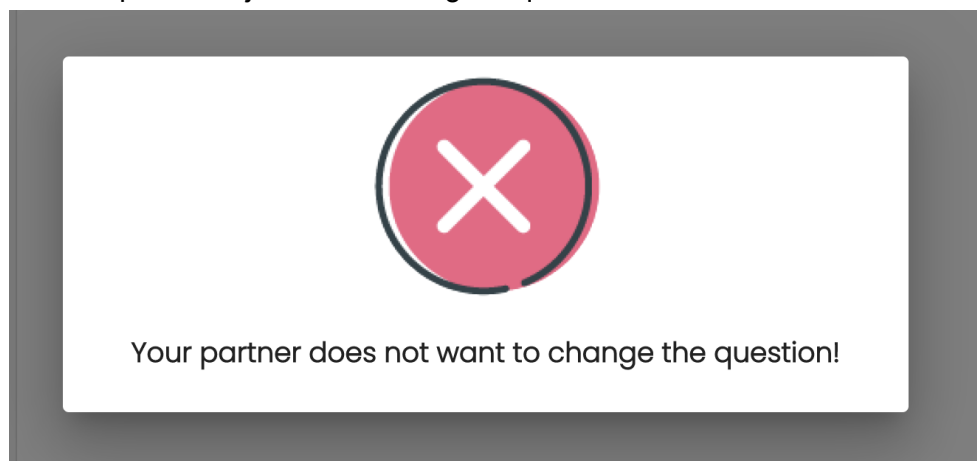


Figure: Partner rejects change in question

Here is a high-level activity diagram for the Change Question functionality.

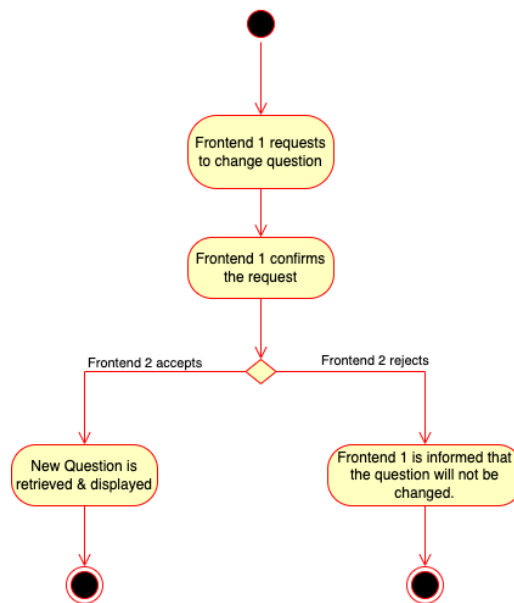


Figure: Change Question Activity Diagram

Here is a sequence diagram between Frontend 1, Frontend 2, Collaboration Service and Question Service.

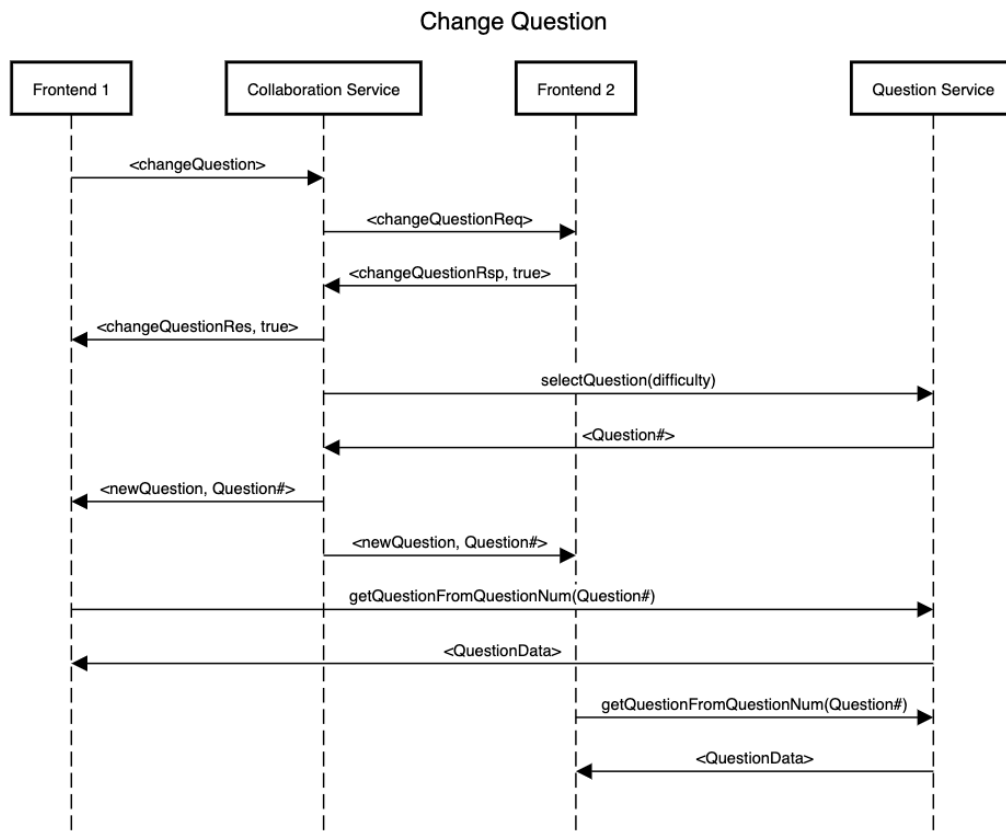


Figure: Sequence Diagram of Changing Question (Frontend 1 initialises the change)

We had the option for Collaboration Microservice to retrieve the question data from the question service to be returned to frontends. However, even though this reduces the number of calls frontend makes, we felt that this increased the coupling between Collaboration Microservice and Question Microservice.

## 10.7 Code Editor and Chat

Both the code editor and chat are similar as they involve real-time communication between the two users.

When a user enters text into the code editor, the socket will emit an event to the microservice. The microservice will then broadcast the event.

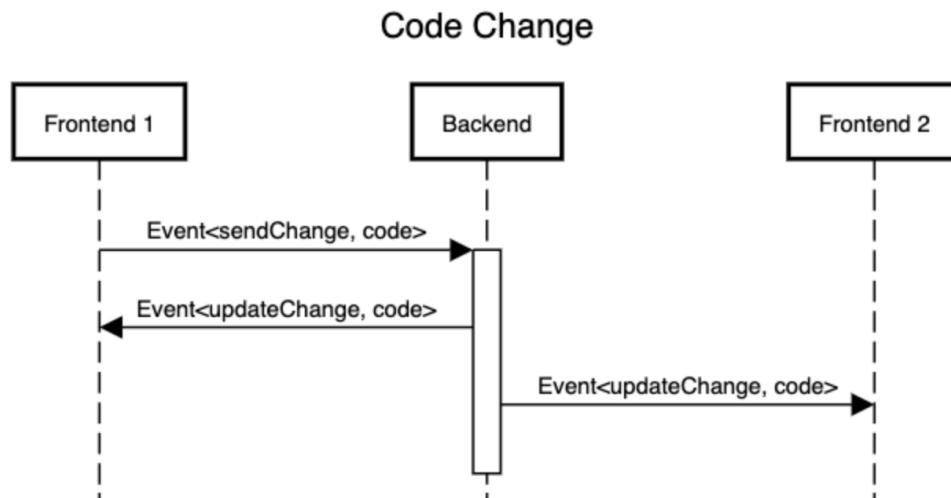


Figure: Sequence Diagram for code change

The Chat Microservice follows a similar flow. Here are some screenshots of the chat:



Figure: Frontend 1 Chat



Figure: Frontend 2 Chat

## 10.8 Leave Room

When users are done with their session, they can leave the room by clicking on the Leave Room button at the bottom of the Code Editor.

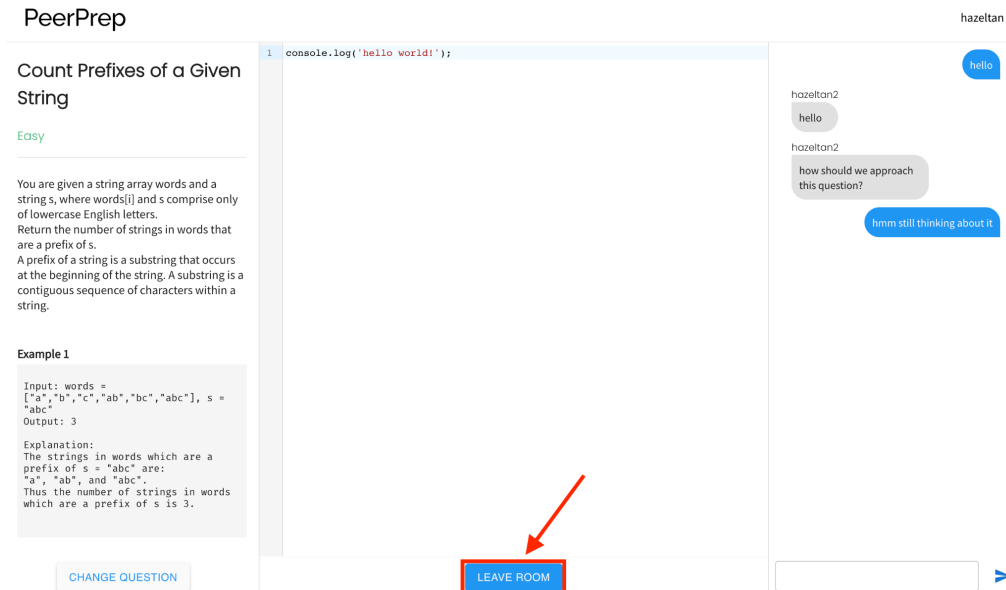


Figure: Leave Room Button

The user, Frontend 1, will then be prompted to confirm that they want to leave the room. Frontend 1 will be redirected back to the Home Page once they have left the room.

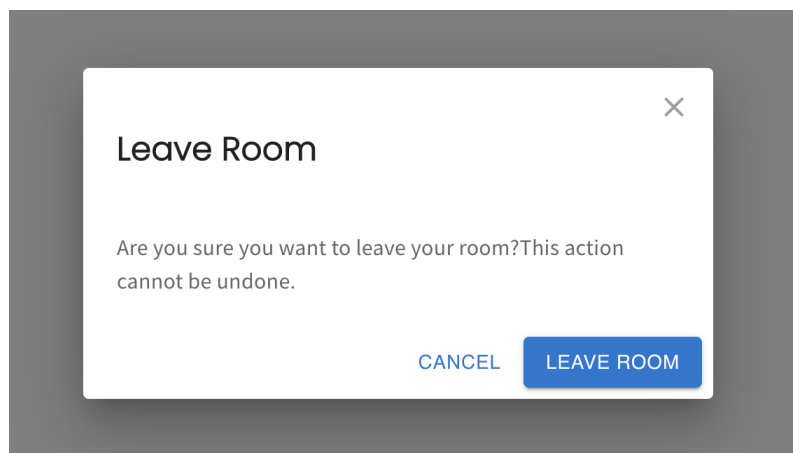


Figure: Leave Room Dialog

After Frontend 1 leaves, Frontend 2 will be informed and given a choice to stay or leave the room.

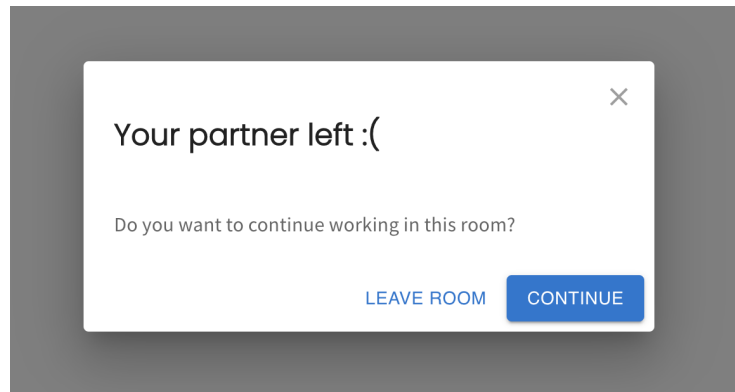


Figure: Frontend 2 notification

If they leave the room, they will be redirected back to the Home Page as well. If they continue, they can continue working on the code in the room.

Leaving the session involves communicating with Matching Service, Collaboration Service and Chat Service to close their respective sessions.

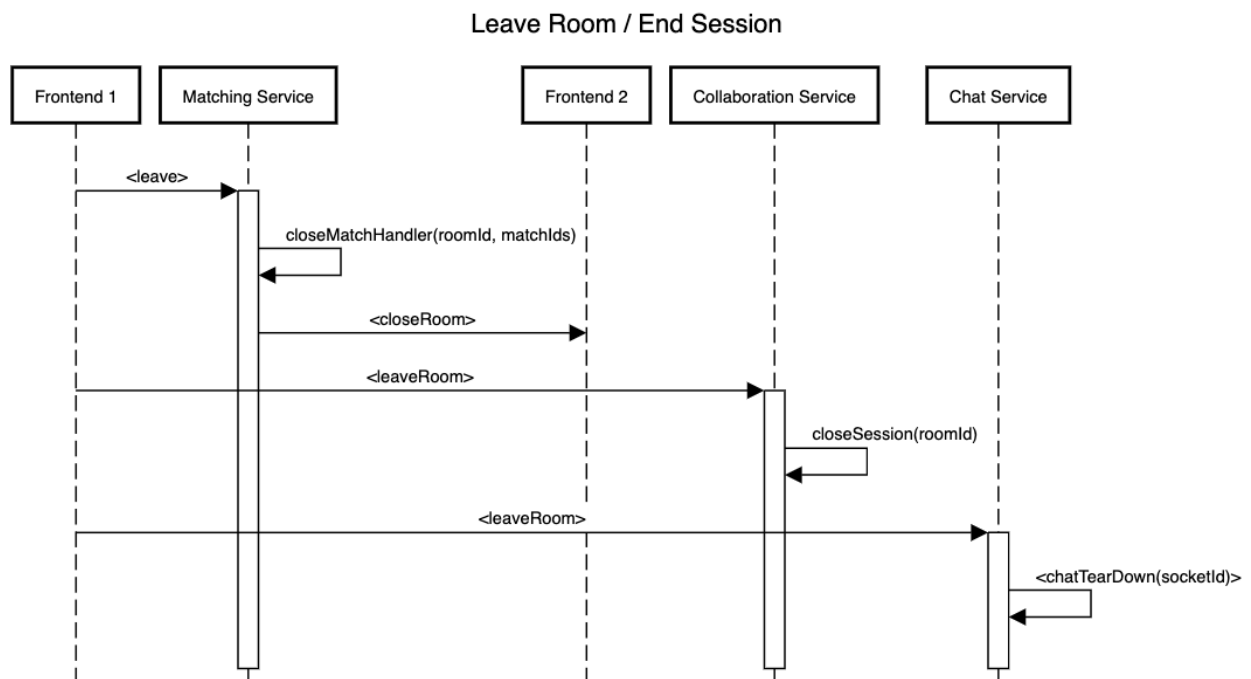


Figure: Leave Room Sequence Diagram (perspective of Frontend 1)



# 11. Remarks

## 11.1 Challenges Faced

One of the main challenges we faced was solving bugs that only occur in production. We had to learn how to read console outputs using Google Cloud console. Furthermore, every time we wanted to tweak some code, we would have to wait for a few minutes to deploy the changes to Google Cloud. This made debugging tedious and inefficient.

We also accidentally hit the quota for our Firebase cloud database one day before the deadline. This was a result of running too many stress tests on our Question and History services. Given the limited time frame, we decided to temporarily solve the problem by upgrading our Firebase cloud database to a paid version for a few days.

## 11.2 Potential Extension Features

### **History Dashboard Augmentation**

One possible extension of the history service is to display simple user statistics in the frontend. For example, we could show a pie chart to illustrate the breakdown of previously attempted questions by difficulty or topic. Users can benefit from this data analysis to figure out which areas they need to have more practice on.

This would involve some processing of data on the history service and some frontend code to draw the charts. This was originally planned to be part of Milestone 3, but we had to postpone it to focus on debugging issues with the collaboration service on the production environment.

### **History Memory Augmentation**

Another possible extension of the history service is to store a user's attempt for a question and subsequently allow the user to view and edit their attempt. Users can learn from looking at what they wrote previously and even improve their own solution in their own time.

This would involve augmenting the database of the history service to store a user's attempt in addition to just the questions that they have attempted. It would also make sense to create a new endpoint for the frontend client to specifically fetch, for one question, its identifier together with the user's attempt for that question.

## 11.3 Reflections

Our group underestimated the amount of time and effort that we had to put into this project. This was mainly due to the fact that we had to learn new technologies such as implementing the Pub-Sub pattern with socket.io, implementing CI/CD with Github Actions and deploying services to Google Cloud Platform.

Nevertheless, we found this to be a fulfilling experience and we think we would have been able to implement more features if we had more time. This was especially pertinent in Milestone 3 where we spent most of our time debugging problems in production.

The four of us have all taken something away from working on Peerprep. This knowledge and experience will serve us well in our future endeavours.