CS3219 Software Engineering Principles and Patterns

Group 39

Project Report

Source code: https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g39
Deployed Web Application: https://www.cs3219-peerprep-g39.com/

Project Team

Lim Boon Hai	A0217670E
Dillon Tan Kiat Wee	A0218033R
Kua Hong Peng James	A0218089W
Chuang Zhe Quan	A0218473B

1. Introduction

1.1 Background

In recent years, there has been an increase in the number of students who face challenging technical interviews when applying for software engineering jobs. Such challenges 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.

Even though the availability of technical assessment platforms such as Leetcode and HackerRank can help students practice their foundations, these platforms mostly cater for individuals to practise their algorithmic skills alone, which are effective for interviews that involve take home assignments.However, more companies as incorporating other forms of technical assessments such as live coding and pair programming. Such technical assessments would require students to articulate their thoughts to others and come up with a solution as a team. Currently, there are no technical assessment platforms that provide such functionality. Moreover, grinding practice questions can be tedious and monotonous.

1.2 Purpose

With our product, PeerPrep, we hope to reduce the challenges faced by providing students with a web application to practice and solve technical interview questions together with another student. PeerPrep incorporates a peer learning and collaboration system which allows students to work together and learn from each other when solving technical interview questions. This not only helps to break the monotony of revising alone, it also helps to improve the communication skills of the student. Students are also able to learn alternative solutions from their peers by solving the same problems.

1.3 Project Scope

PeerPrep is a web application targeted at penultimate and final year students from the School of Computing in NUS to help them better prepare themselves for their technical interviews. We have chosen penultimate and final year students because these students are most likely to be applying for internships compared to freshman or sophomore students. Hence PeerPrep would be the most beneficial to them.

2. Contributions

2.1 Technical Contributions

Name	Technical Contributions
Lim Boon Hai	Matching-Service, Frontend Design and Implementation
Dillon Tan Kiat Wee	Testing of microservices, Continuous Integration
Kua Hong Peng James	Code Editor, Code Sync, Deployment
Chuang Zhe Quan	User Service, Question Service, History Service

2.2 Non-Technical Contributions

Name	Non-Technical Contributions
Lim Boon Hai	[4.2] Performance Requirements [4.4] Usability Requirements [6] Tech Stack [8.2] Matching Service [9] Frontend Deployment
Dillon Tan Kiat Wee	[1] Introduction [3] Functional Requirements [4] Non Functional Requirements [5.2] Continuous Integration using GitHub Actions [6] Tech Stack
Kua Hong Peng James	[5] Software Development Process [5.1] Continuous Integration/Continuous Deployment Pipeline [5.3] Continuous Deployment [6] Tech Stack [7.1] High Level Architecture [8.3] Code Sync
Chuang Zhe Quan	[7.1] High level Architecture [7.2] Design Patterns [7.3] Microservice Architectural Decisions [8.1] User Service

3. Functional Requirements

The following specifies the functional requirements (FR) of our application. All planned FRs have been implemented and deployed to production. In the following section, we would detail out the individual requirements which were planned for each aspect of the application.

Each functional requirement is also prioritised based on the priority legend below.

Priority Legend	
High	A critical requirement without which the product is not acceptable to the users.
Medium	A necessary but deferrable requirement that makes the product less usable but still functional.
Low	A nice to have feature if there are resources but the application functions well without it.

3.1 User Service

The User Service is required to allow users to sign up for an account and access the PeerPrep application.

S/N	Functional Requirement	Priority	Implementation
FR1.1	The system should allow users to create an account with username and password.	High	Sign up button
FR1.2	The system should ensure that every account created has a unique username.	High	Username is checked against database for duplicates during sign up
FR1.3	The system should allow users to log into their accounts by entering their username and password.	High	Login button
FR1.4	The system should allow users to log out of their account.	High	Logout button
FR1.5	The system should allow admins to delete user accounts.	High	deleteUser function
FR1.6	The system should allow users to change their password.	Medium	Update password button

3.2 Matching Service

The main use of PeerPrep is to be able to match up individuals. To be able to have such a functionality, a Matching Service is essential.

S/N	Functional Requirement	Priority	Implementation
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High	Difficulty button in homepage
FR2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High	createMatch function
FR2.3	If there is a valid match, the system should match the users within 30 seconds.	High	createMatch function
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High	createMatch function
FR2.5	The system should provide a means for the user to leave the queue before a match is found.	Medium	Cancel button

3.3 History Service

The History Service is created to allow the users to have access to their past attempted problems and monitor their progress.

S/N	Functional Requirement	Priority	Implementation
FR3.1	The system should be able to retrieve the questions that the user has attempted before	High	Attempt history dashboard
FR3.2	The system should be able to sort the questions that the user has attempted before by difficulty	High	Attempt history dashboard

3.4 Question Service

The Question Service is required to allow admins to add coding questions to the question bank. This service is also used to generate questions for the Room Service.

S/N	Functional Requirement	Priority	Implementation
FR4.1	The system should have at least 10 practice questions	High	Questions stored

	for users to practise with.		in MongoDB
FR4.2	The system should be able to retrieve questions based on difficulty.	High	getQuestionsBy Difficulty API call
FR4.3	The system should allow admin to be able to add new questions to the question bank.	High	createQuestion API call

3.5 Frontend

The Frontend is required to allow users to be able to interact with the application on the browser. This service is also used to implement certain Usability NFRs.

S/N	Functional Requirement	Priority	Implementation
FR5.1	The system should display the questions that the user has attempted before.	High	Homepage
FR5.2	Changes to the code must be reflected on both users' screens in real time.	High	Code sync using Twilio Sync
FR5.3	The system should close the room and return the users back to the home screen.	High	End session button
FR5.4	When one user leaves the room, the room should close and the other user should be informed of the closure.	High	Notification shown when another user leaves the room
FR5.5	The system should record the question that the room has attempted.	Medium	Match object has a tag question
FR5.6	The system should not log the user out when the user refreshes the page.	Medium	User credentials are stored in localStorage

4. Non Functional Requirements

The section following specifies the non-functional requirements (NFRs) of our application that are grouped into system's quality attributes. As part of our initial planning, we utilised a Quality Attributes Prioritization Matrix to rank the Non-Functional Aspects required for our application, as shown below.

Attributes	Score	Availability	Integrity	Performance	Reliability	Robustness	Security	Usability	Verifiability
Availability	4		←	↑	←	←	↑	↑	←
Integrity	1			↑	↑	←	↑	↑	↑
Performance	6				←	←	↑	←	←
Reliability	3					←	↑	↑	←
Robustness	1						↑	↑	←
Security	6							↑	←
Usability	6								←
Verifiability	1								

Quality Attributes Prioritization Matrix

Based on our requirement prioritisation above, we decided to focus on NFRs that fulfil our performance, usability and security requirements, as they have the highest scores. Each individual Non-Functional aspect is detailed below.

4.1 System Requirements

Different users may use different browsers or different systems to access the application. To make it easy to access the application, we must ensure the following system requirements.

S/N	Non Functional Requirement	Priority
NFR1.1	The application should be able to run in most web browsers without any major compatibility issues.	High

4.2 Performance Requirements

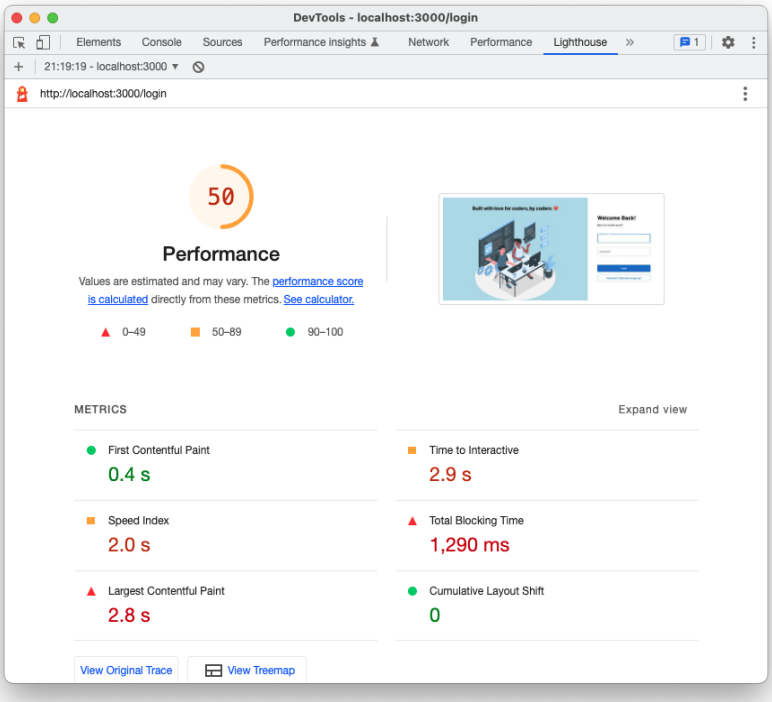
A long waiting time can cause the user to lose attention or change their intention to use the application, worsening user experience. To ensure that users are able to, and will use our application, we must ensure the following performance requirements.

S/N	Non Functional Requirement	Priority
NFR2.1	The application should render in less than **5** seconds.	High
NFR2.2	The system should support **200** simultaneous users accessing the system.	High

NFR2.3	The code sync between users should occur within **1** seconds after one user finishes typing.	High

NFR2.1 Render Performance

To determine the render performance of our application, we utilised Google's Lighthouse to obtain the typical metrics which are used to determine the performance of a web application. Attached below is a sample screenshot of Google's Lighthouse Summary Report, for `/login`.



Summary Report By Google Lighthouse on `/login`

Below are the summary results of the relevant metrics which we have obtained for our application, where the meaning of each metric is indicated below:

- **Speed Index**, which measures how quickly contents of the page is populated
- **First Contentful Paint,** which measures the time at which the first text/content is painted on the page.
- **Time To Interactive,** which measures the time taken for the page to be fully interactive.

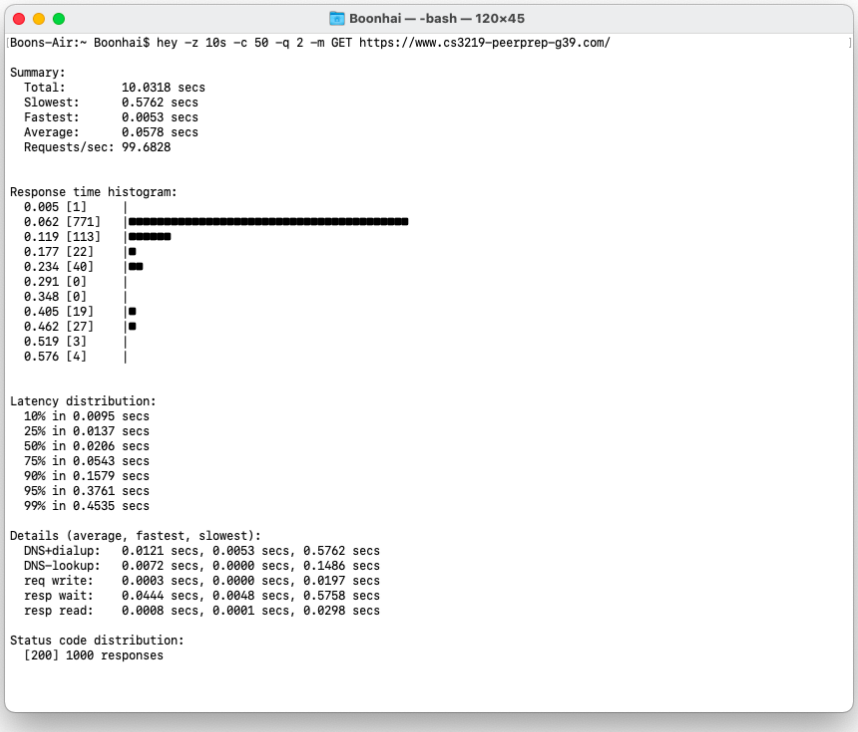
Routes	Relevant Lighthouse Scores (in seconds)
/login	Speed Index: 2.0 First Contentful Paint: 0.4 Time To Interactive: 2.9

/signup	Speed Index: 3.0 First Contentful Paint: 0.4 Time To Interactive: 3.5
/home	Speed Index: 1.7 First Contentful Paint: 0.4 Time To Interactive: 3.2
/passwordchange	Speed Index: 2.6 First Contentful Paint: 0.4 Time To Interactive: 2.6

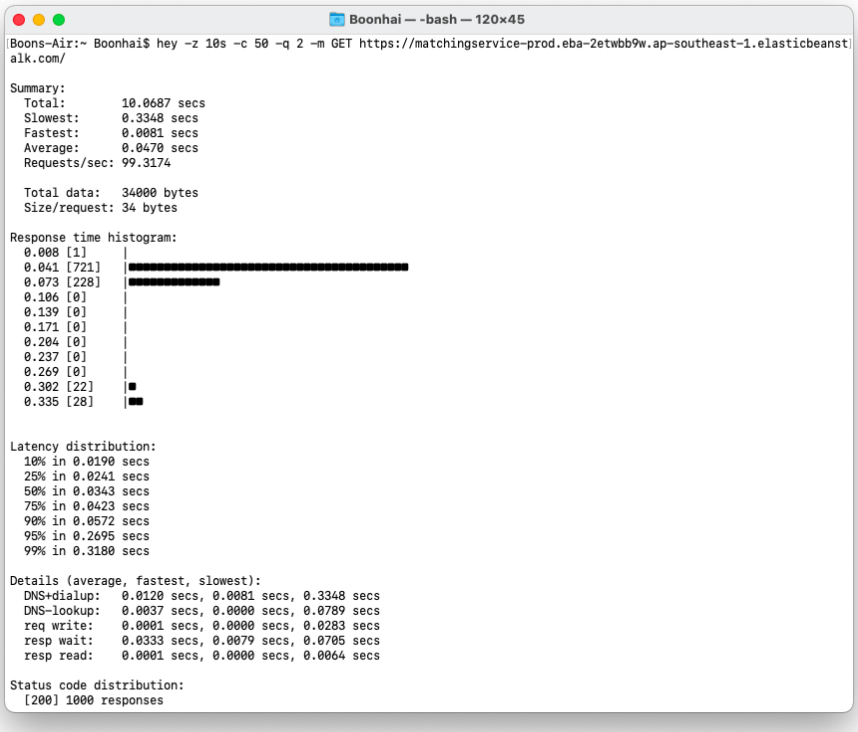
As seen from above, all the pages are able to completely render and be interactive in less than 5 seconds. This suggests that **NFR2.1** is met.

NFR2.2 Scaling Performance

To determine whether our application was able to support a large number of users, a load test was conducted on our deployed application. We sent a total of 100 requests over a period of 10 seconds. Below are the screenshots of the tests run on the frontend client as well as Matching Service.



Report of Load Test to Frontend Client



Report of Load Test to Matching-Service

From the load test results, we note that all requests made have a return response of `200 OK`. This suggests that our system is able to accommodate 1000 concurrent users accessing our system simultaneously. Hence, it is capable of supporting 200 users and even more on our website, and **NFR2.2** is met.

4.3 Security Requirements

It is important that only registered and authorised users can have access to the application. On our side as developers, it is important that we protect the sensitive information of the users, such as their passwords.

S/N	Non Functional Requirement	Priority	Implementation
NFR3.1	All API endpoints except `/login` and `/signup` should be authenticated with the JWT token.	High	Route authentication on frontend using JWT
NFR3.2	Users must be registered by the system before being allowed to log into the application.	High	Login system

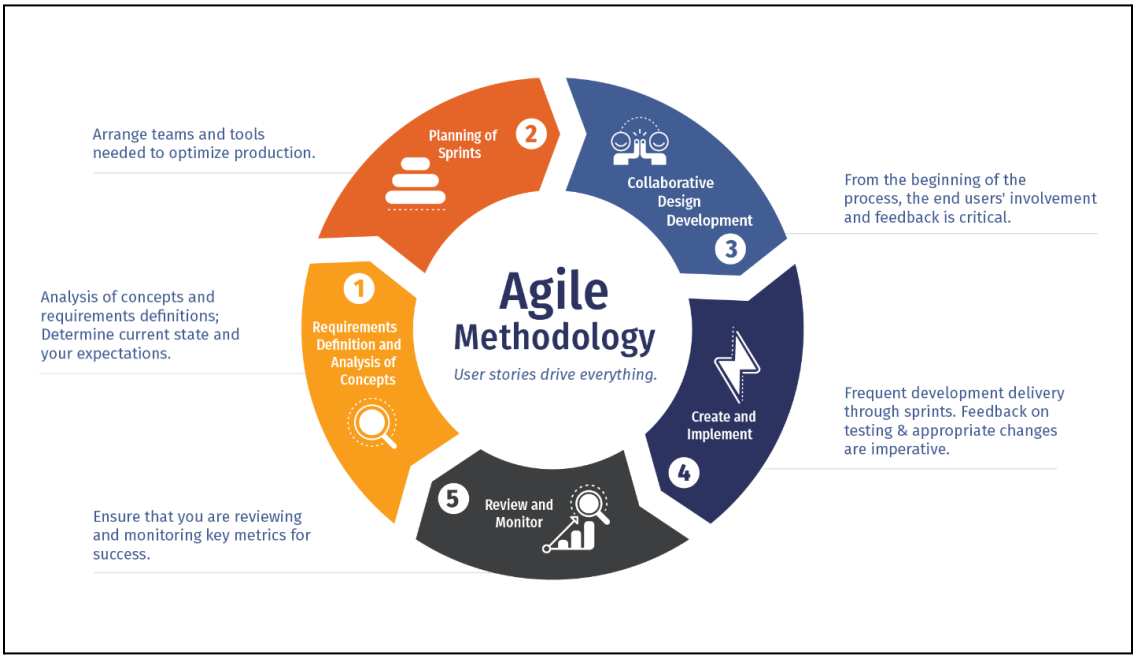
NFR3.3	Users' passwords should be encrypted and not stored as plain text.	High	Bcrypt encryption

4.4 Usability Requirements

For users to want to use the application, it is important for the application to be intuitive and easy to use. In addition, it is important to make sure the user's experience in using the application is as smooth as possible. The system should not leave the user wondering why certain actions are unsuccessful, and any potential actions done on the system by the user should be predictable. To do so, we should fulfil the following usability requirements.

S/N	Non Functional Requirement	Priority	Implementation
NFR4.1	The application's User Interface should ensure that each action that a user can engage in on the page is clearly indicated and easily understandable.	High	Frontend styling, text in elements
NFR4.2	The application should provide clear feedback on any action undertaken by the User to allow the user to know if an action is successful or unsuccessful.	Medium	Alert banners and dialogs
NFR4.3	The application should follow a colour and font scheme which allows for the content to be easily legible.	Low	Frontend styling
NFR4.4	The application should ensure confirmation from the User before any irreversible actions are made on the system.	Low	Confirmation dialog

5. Software Development Process



Agile Methodology Summary Infographic

Our team adopted an agile methodology process for this project, similar to the one shown in the above figure. We maintain requirements in the form of functional requirements in a product backlog, which are ranked based on their priority. Below is a snippet of the Functional Requirement allocation to each of the group members in the early stages of development.

TASK TITLE	FUNCTIONAL REQUIREMENT	TASK OWNER	WEEK	PRIORITY
Set up Github classroom and repository		Dillon	4	High
Set up user service and authentication	1.1 - 1.6	Zhe Quan	4	High
Set up MongoDB Atlas, match model	3.5	Dillon	4	High
Draft login page and matching page	1.3	James	4	High
Draft difficulty selection page	2.1	Boon Hai	4	High
Create question model	4.2, 4.3	Zhe Quan	5	High
Create question bank	4.1	Zhe Quan	5	High
Draft session page	5.2	James	5	High
Update FRs/NFRs in documentation		Dillon	5	High
Create matching service logic using SocketIO	2.2	Boon Hai	5	High
Integrate frontend with backend logic	2.3, 2.4, 5,4, 5.5	Boon Hai	6	High
Implement route authentication		James	6	High
Synchronise code between users	5.2	James	7	High
Add dashboard on home page	5.1	Boon Hai	7	High
Revamp frontend pages	3.4	Boon Hai	7	High
Create learning pathway/history service	3.1, 3.2	Zhe Quan	7	High
Add test cases		Dillon	7	High
Set up Github actions and CI workflow		Dillon	7	High

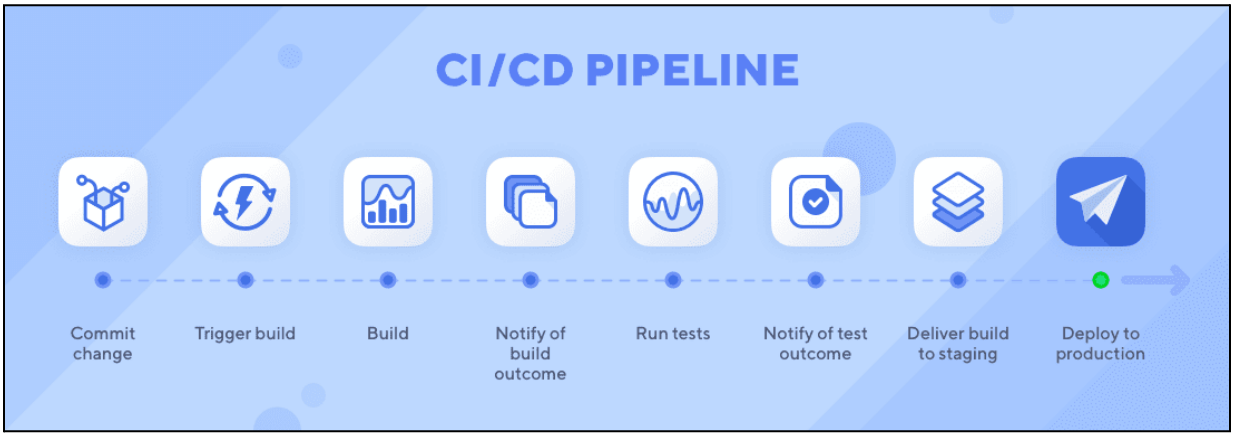
Functional Requirement Allocation Schedule

We adopted iterative and incremental development, breaking down the development into short iterations that last for one week. During each iteration, we add a small set of functionality into our application based on the priorities established within our team. Subsequent iterations are meant to enrich the initial features and fix any bugs with the already implemented features that may have been identified.

For the first half of the project up to recess week, we first prioritised features that needed to be developed in order to build our minimum viable product. We would meet once every week to update the work that we have done in the past week, any issues faced that needs to be resolved, and make decisions on key matters as a group. In the process, we would also share the concerns that we have in terms of development. We would then discuss and assign the work that needs to be completed within the next week. At the same time, we would review and test each other's pull requests before merging it to our master branch.

For the second half of the project after recess week, we were focused on polishing the look of the application, adding additional features that we deem as being useful to improve the functionality of our application, as well as implementing the process of continuous integration and deployment, elaborated more in the section below.

5.1 Continuous Integration/Continuous Deployment Pipeline



Outline for CI/CD Pipeline

For our project, we decided to implement the CI/CD pipeline as illustrated in the figure above. This will help establish a continuous and automated cycle in which deliveries are completed faster with more value. An effective CI/CD process allows us to maintain a quick and responsive feedback loop whereby we can resolve bugs quicker and more efficiently. We will have more confidence in integrating our code in smaller chunks daily.

5.2 Continuous Integration using GitHub Actions

By integrating test automation into the pipeline, we can enable faster builds and deployment by continuously generating feedback based on test results. Currently, integrated testing is done for all 4 of our microservices: User Service, Question Service, Matching Service and History Service. We created a workflow file for each of the microservices, which causes GitHub Actions to trigger an automated test whenever changes are pushed to GitHub.

Each of the environment variables used in the microservice testing are stored as encrypted secrets using the GitHub Actions Secrets API. Examples stored as secrets include our MongoDB URI and our JWT secret key used in validating a user's JWT token.

The workflow file `ci.yml` consists of 4 jobs, one for each of our microservice. Github Actions will run all the jobs whenever we push a commit on GitHub .

Each job follows this format:

1. Assign environment variables based on the GitHub Actions secrets API
2. Change directory into the according microservice
3. Checkout repo and setup NodeJS
4. Download dependencies
5. Run test cases

Below is an example of the structure of the `test-user-service` job:

```yaml
name: ci
on: [push]

jobs:
  test-user-service:

    runs-on: ubuntu-latest
    env:
      TOKEN_KEY: ${{ secrets.TOKEN_KEY }}
      DB_LOCAL_URI: ${{ secrets.DB_LOCAL_URI }}
      ENV: ${{ secrets.ENV }}

    defaults:
      run:
        working-directory: user-service

    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 16.x
      - name: Download dependencies
        run: |
          npm ci
          npm update
      - name: Run user service test cases
        run: npm test
```

GitHub Actions Workflow for User Service

Each microservice contains a test folder which contains the test cases. The test cases are written in JavaScript using the Mocha testing framework, as well as the Chai assertion library. The reason why we chose to use Mocha and Chai was because it was relatively easy to set up and it is flexible enough to meet our needs for our test cases.

The test cases were written with unit testing in mind. We wanted to make sure that each function of our microservices performs as expected. By performing unit testing, we are able to single out functions that do not work as intended, avoiding the need for us to identify the source of error, such that we can focus on fixing the error. As such, most of the test cases involve the testing of edge cases and verifying the HTTP status code returned by each function of the microservice.

Here is an example of the test case used in the testing of User Service:

```
describe("Test createUser function /api/user/signup", function () {
  it("should return error 400 for invalid username", (done) => {
    chai
      .request(index)
      .post("/api/user/signup")
      .send({
        username: "",
        password: "admin0123",
      })
      .end((err, res) => {
        expect(res).to.have.status(400);
        expect(res.body).to.be.a("object");
        expect(res.body.message).to.equal(
          "Username and/or Password are missing!"
        );
        done();
      });
  }),
    it("should return error 400 for invalid password", (done) => {
      chai
        .request(index)
        .post("/api/user/signup")
        .send({
          username: "testUser2500",
          password: "",
        })
        .end((err, res) => {
          expect(res).to.have.status(400);
          expect(res.body).to.be.a("object");
          expect(res.body.message).to.equal(
            "Username and/or Password are missing!"
          );
          done();
        });
    }),
```

Sample of test cases for User Service

5.3 Continuous Deployment

Building on the Continuous Integration from the previous section, we also incorporated Continuous Deployment, which automates the release of our application to deployment. On the condition that all the unit tests have passed, GitHub Actions will trigger the deployment workflow that deploys the backend services to AWS Elastic Beanstalk. Similarly, AWS Amplify automatically builds and deploys our frontend on detection of code changes pushed to the repository as we have linked our main branch to it.

```
1  name: Deploy to EB
2
3  on:
4    workflow_run:
5      workflows: [ci]
6      branches: [ main ]
7      types:
8        - completed
9
10 jobs:
11   deploy-history-service:
12     if: ${{ github.event.workflow_run.conclusion == 'success' }}
13     runs-on: ubuntu-latest
14     steps:
15     - name: Checkout source code
16       uses: actions/checkout@v3
```

Snippet from deployment YAML file

In line 12 of the deployment workflow file, it is specified that `deploy-history-service` job only runs if the workflow run of `ci` is successful. This means that changes to History Service are only deployed to production on the condition that all the unit tests are passing. The same logic applies to all the other backend services.

6. Tech Stack

Component	Application/Framework
Frontend	ReactJS, MaterialUI, React-JSS
Backend	NodeJS, Express, Socket.io
Database	MongoDB
CI/CD Tools	GitHub Actions, Mocha, Chai
Deployment	AWS Elastic Beanstalk, AWS Amplify

The following are the considerations that we have taken into account while deciding on the specific technological stacks to use.

ReactJS (For Frontend)
ReactJS is an open source frontend JavaScript library that is developed by Facebook. It allows developers to build complex User Interfaces through the composition of small and isolated pieces of code known in React as “components”.

ReactJS was chosen by our team due to the following:

- Most of our team had prior experience with ReactJS, so we could reduce the learning curve and better focus on learning other tools which would be required in building the other parts of the application.
- ReactJS is currently maintained and has many other users. Thus, it would be much easier to find solutions to any of the issues we may potentially face.
- ReactJS has many tooling and supporting libraries which could be used to help build the frontend of our application.
- ReactJS also provides modularization of frontend via components, which allows better readability and maintainability of frontend code.

MongoDB using Mongo Atlas (For Database)
MongoDB is a non-relational document database that allows developers to store unstructured data, and supports full indexing support and replication. In addition, MongoDB is run on a cloud service called Mongo Atlas, which handles the deployment and management of the database.

MongoDB was chosen by our team due to the following:

- We chose MongoDB as its flexible schema design allows non-defined attributes to be modified on the fly, which makes it highly scalable.
- MongoDB also offers querying and indexing speeds that are way higher than the average relational database.

- MongoDB is also built on a horizontal-scale-out architecture that distributes the server load across several servers through a process called auto-sharding, allowing us to store extensive amounts of data without having to worry about load distribution.

Twilio Sync (For Code Collaboration Service)
Twilio Sync is a state synchronisation service that offers two-way real time communication between users. It adopts a Pub-Sub mechanism and in this model, the authoritative state of the application, which in this case, is the collaborative code that both users are working on, is updated to the cloud and is actively replicated to all users.

Twilio Sync was chosen due to the following advantages it provides:
- Easy to set up and lightweight
- Supports built-in authentication processes
- Only milliseconds of latency, ensuring responsiveness for the end user

GitHub Actions (For Continuous Integration and Continuous Delivery)
GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows users to automate their build, test and deployment pipeline. Users can create workflows that build and test every pull request to the repository, as well as deploying merged pull requests to production.

GitHub Actions was chosen by our team due to the following:
- GitHub Actions is much easier to set up and configure compared to other tools. As a cloud service, GitHub Actions can be implemented without any prior installation.
- GitHub Actions allows for asynchronous CI/CD which allows us to save time when we deploy our product.

AWS Elastic Beanstalk & AWS Amplify (For Deployment)
The scalable, reliable, cost-effective yet high-performance factors of AWS was the driving motivation behind the adoption of AWS as the choice of our deployment platform.

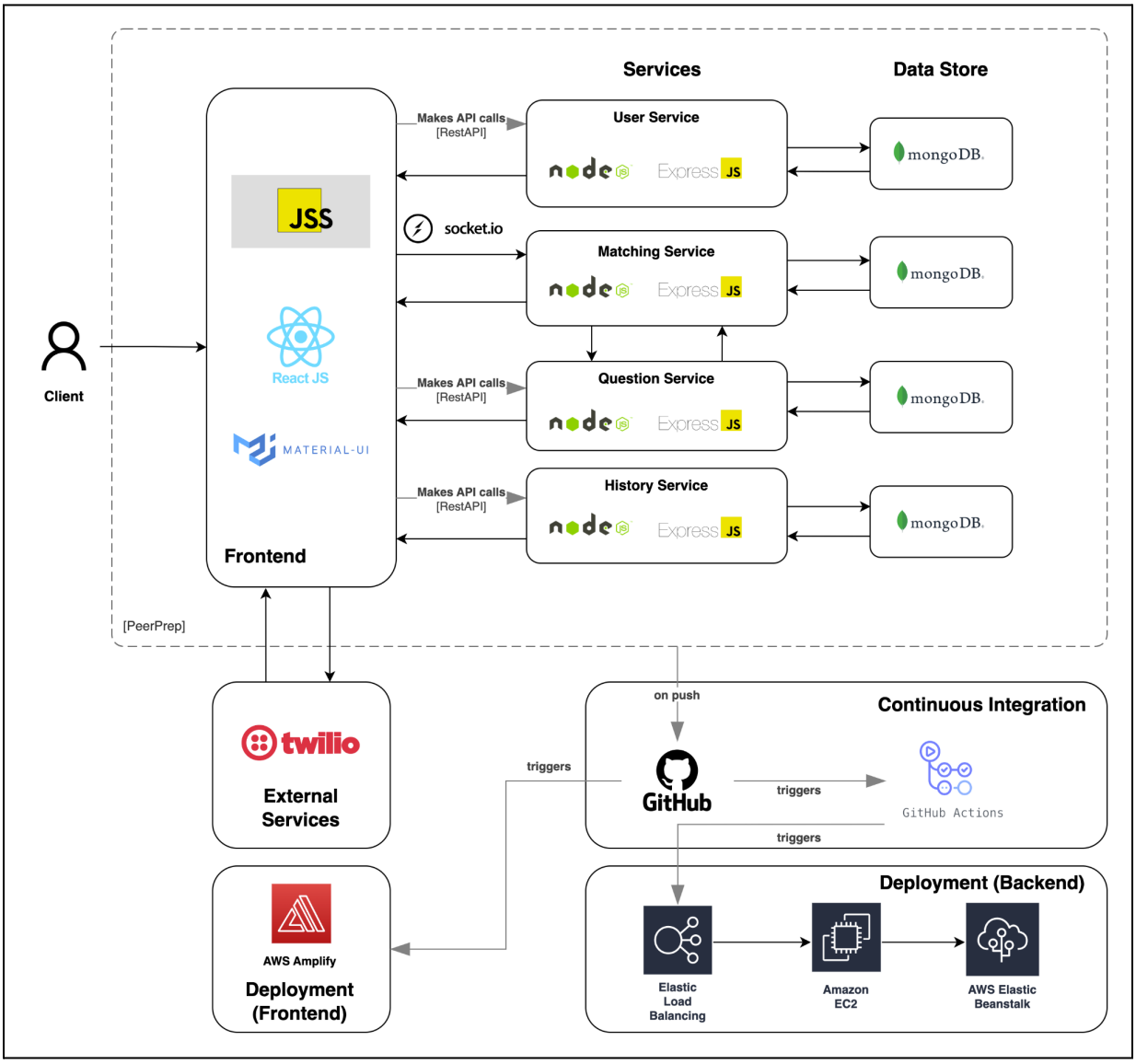
AWS Elastic Beanstalk automates the configuration and provisioning of other AWS services such as EC2, Elastic Load Balancing to create a web service. It also has built-in auto-scaling functionality to support load distribution.

AWS Amplify allows us to deploy our frontend very quickly. Once we have specified the build configuration file, there is an auto-build function that allows us to link our GitHub repository and any changes pushed to the repository will trigger a redeployment of the frontend automatically.

7. Application Design

7.1 High Level Architecture

Microservices vs. Monolithic Architecture



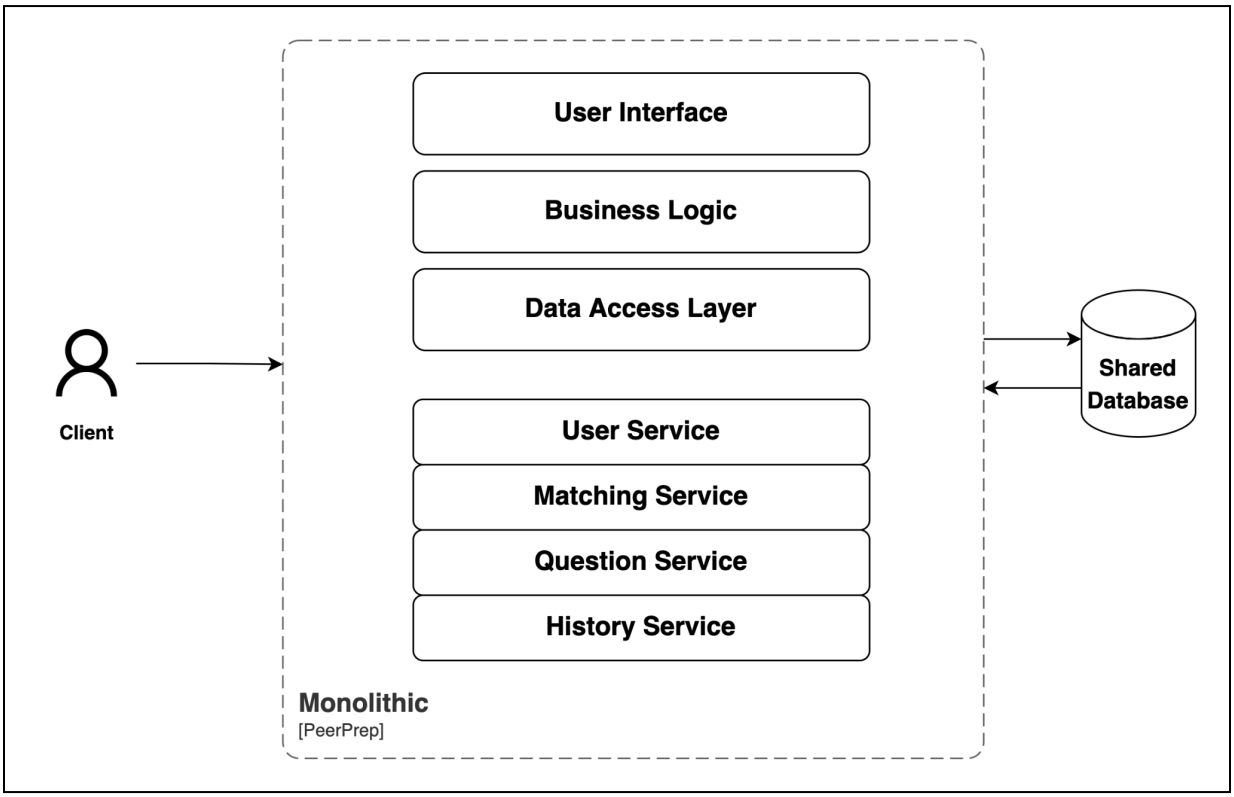
High Level Architecture Diagram of PeerPrep using Microservice Architecture

Our team decided to shape our application using the microservices architecture. This is due to the following reasons:

1. Microservices architecture comprises of components that are loosely coupled that makes the application flexible to changes. This makes it easier for debugging and performing maintenance.

2. Microservices also offer better fault isolation, when one service is down, the other services are not affected.
3. The addition of new services can be integrated seamlessly into the application without having to rewrite the entire application.
4. Microservices allow for independent deployability, new services can be deployed without having to deploy other services. Updating, testing, deployment and scaling can occur independently within each service.
5. We can choose to use different technologies inside each service, allowing us to pick the right tool for each job rather than having to select a more standardised, one-size-fits-all approach that often ends up being the lowest common denominator.

Before settling for the microservices architecture, we have also considered the use of an alternative architecture style, the monolithic approach, as illustrated below.



High Level Architecture Diagram of PeerPrep using Monolithic Architecture

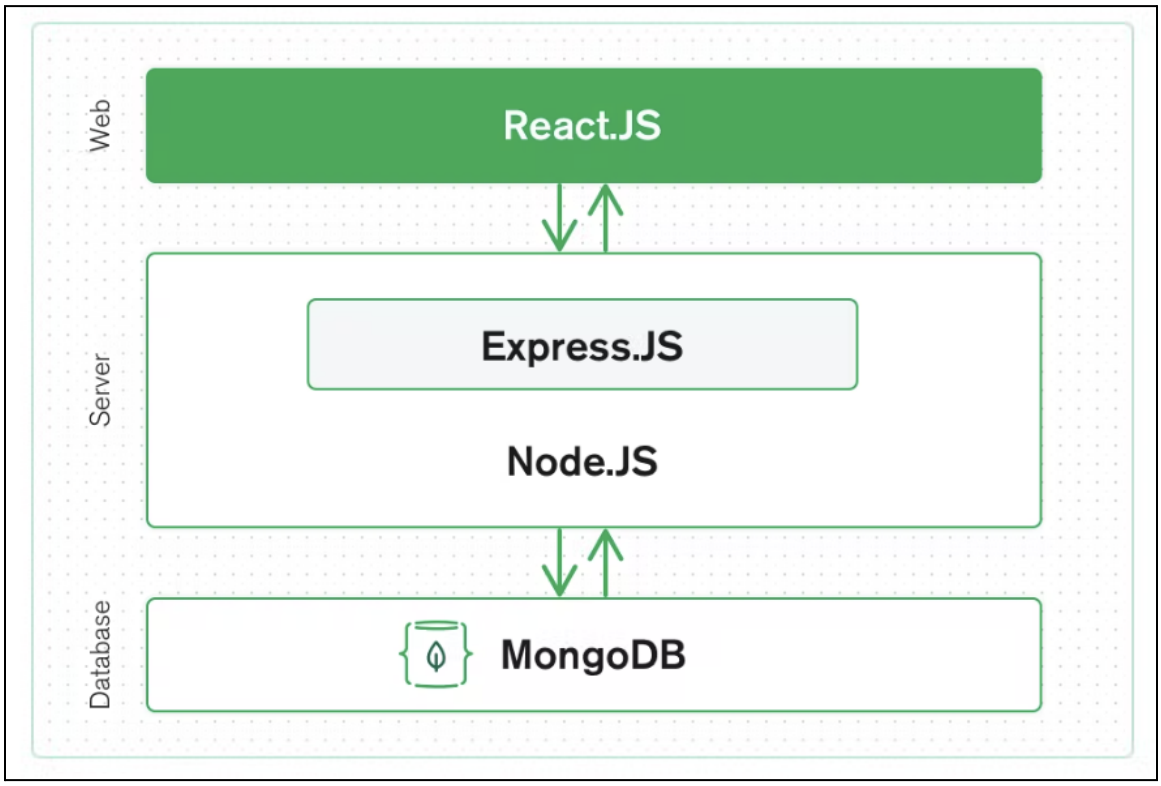
However, we have decided that monolith architecture is not suitable for the following reasons:

1. Any change in the framework or language affects the entire application, making changes are often expensive and time-consuming.
2. Slower development speed due to large and complex codebase.
3. An error in a single module can bring down the entire application. With microservices, we can build systems that handle the total failure of some of the constituent services and degrade functionality accordingly.

To allow us to be able to work independently and efficiently, we decided that it is best that we adopt the microservices architecture so that we can split the work evenly and can work in parallel without having to rely on the implementation of other components.

3-Tier Architecture

Our team also adopted the 3-Tier architecture while implementing our application.



3-Tier Architecture Diagram (taken from https://www.mongodb.com/mern-stack)

The MERN stack is a 3-tier architecture, consisting of the Web tier, the Server tier and the Database tier. The Web tier interacts with the server by making API calls while the server interacts with MongoDB using the mongoose package.

There are several reasons why we chose to work with the MERN stack with a 3-tier architecture:

1. **Loose Coupling**
   The Web is loosely coupled to the server as the only connection between them are the API calls. Therefore, the backend can be changed with minimal effects on the frontend.

2. **JSON Data Structure**
   MongoDB works well with NodeJS because it makes storing, manipulating and retrieving JSON data at all layers very easy. Furthermore, ReactJS uses Javascript, it

is also capable of manipulating the JSON data structure and little additional work is required to translate data between the Web and Server layer.

3. **Better Scalability**
   By running each layer in a different server, each layer can be scaled independently without affecting other layers.

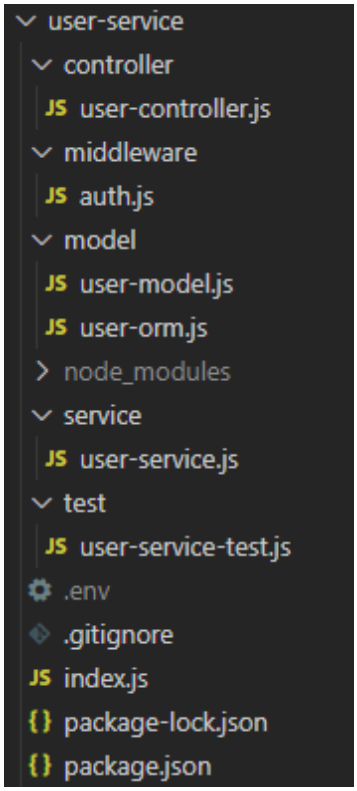
7.2 Design Patterns

Model-View-Controller (MVC) Design Pattern

We primarily followed the MVC design pattern in our application. While our architecture does not strictly follow the MVC design pattern as the application spans across two services – the frontend and the server, MVC principles were still applied.

Our View is written in Javascript, HTML and CSS, which uses ReactJS as the framework. Within the View, users can make API requests to various microservices, which has the Controller and the Model. The Controller will determine what logic to execute based on the actions taken and the Model will run that logic accordingly.

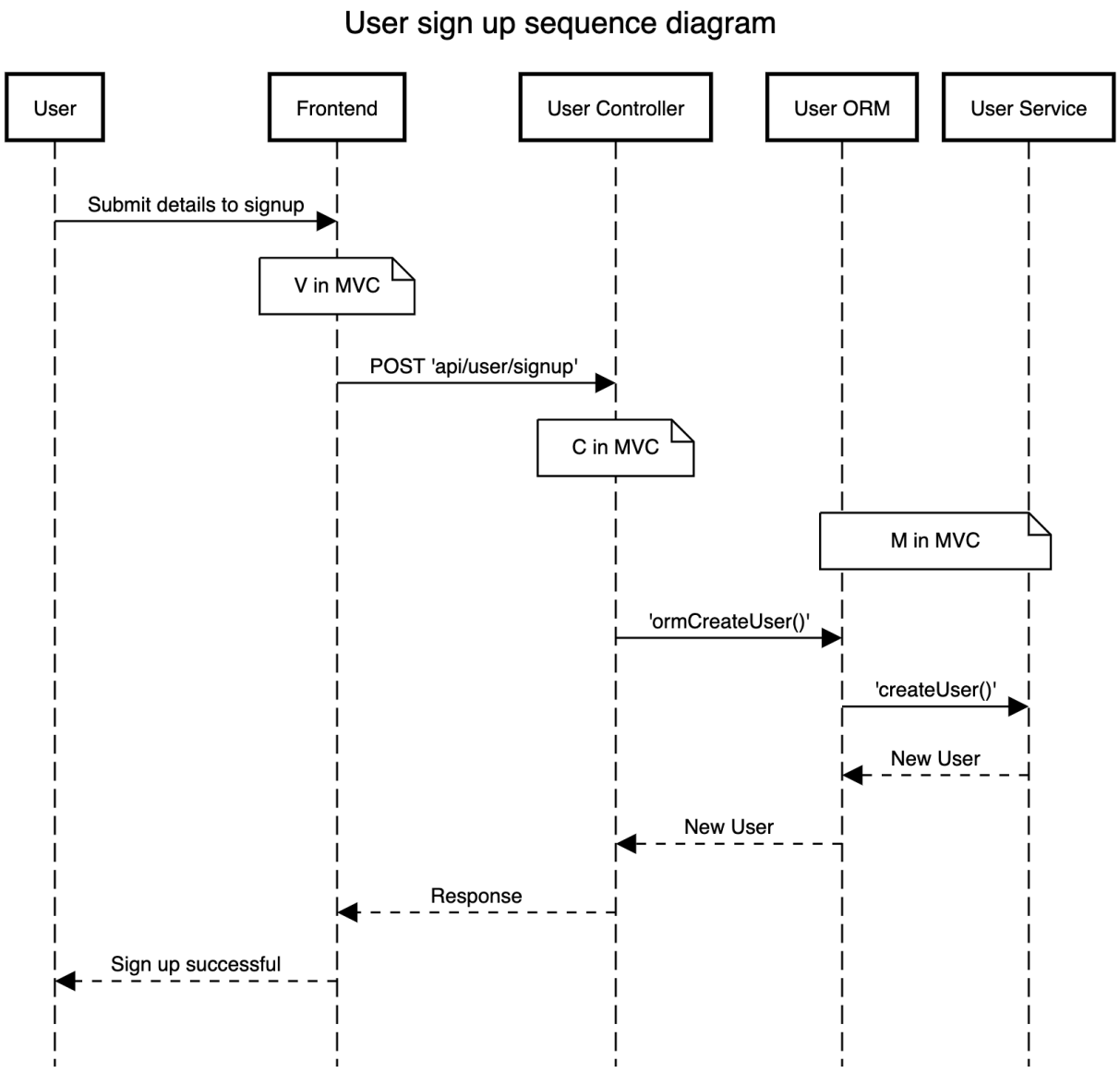
Take the folder structure of the User Service as an example:



User Service Directory

Within the user service, we have the folders: controller, model and service. The model and service would make up the Model component within the MVC. Based on the request received from the view, the controller will choose what logic to run or what data to manipulate by calling functions in the `user-orm`. The model dictates the user schema in the MongoDB database and service manipulates the data in the database.

The sequence diagram below depicts how signing a user up would look like:



Sequence Diagram of User Signing Up

The reasons why we chose to use the MVC design pattern are as follows:

1. **Separation of Concerns**
   Using the MVC design patterns allows for better segregation of the I/O of the user from the application state and services. At the same time, this also allows for the design of our View to be changed without affecting the design of the Model and Controller, and vice versa.

2. **Ease of Unit Testing**
   With reduced coupling, testing can be done much more easily through unit tests.

3. **Accelerates Development Process**
   With the MVC structure, development can be done in parallel. In this case, some members were able to work on the frontend (View) of the application while other members were able to work on implementing the microservices (Model and Controller) simultaneously. This allows for a feature to be developed and pushed into production in a much shorter duration.

7.3 Microservices Architectural Decisions

Matching Service

The Matching Service is a microservice that is responsible for matching users and keeping records of matches which have been created upon a successful match. Users are matched based on their difficulty level chosen.

Suppose a user selects the difficulty “Easy”. The user is added to the `waitingRoom` for “Easy”. The matching-service then makes an API call to Question-Service to retrieve a question to be used. To do so, the Matching-Service utilises the JWT Token of the initial user in the `waitingRoom`.

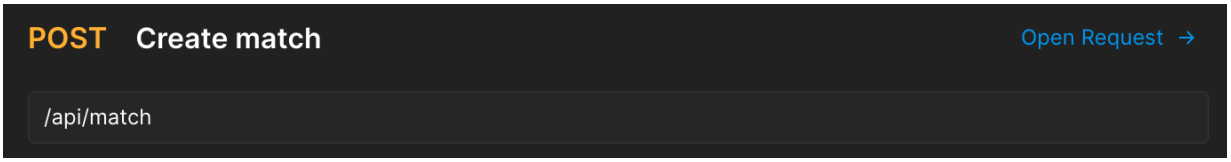
Subsequently, if another user selects the same difficulty, the server will create a room with the following details:

- Usernames of both users
- Question ID of the question selected
- Difficulty of the question selected

If a user is unable to find a match partner within 30 seconds, the match request will timeout and the user will be removed from the waiting room.

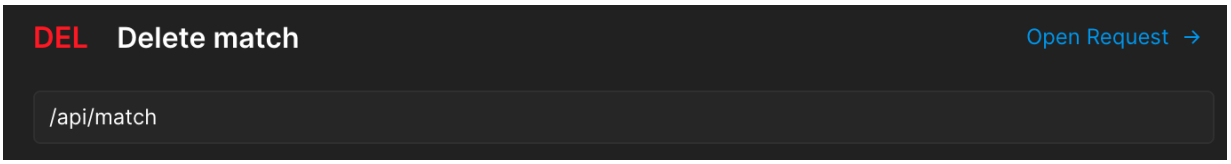
Endpoints

The following are the currently supported endpoints for the matching-service:



body:

```
{
   "userOne": "",
   "userTwo": "",
   "difficulty": "",
   "question": ""
}
```



body:

```
{
   "roomid": ""
}
```

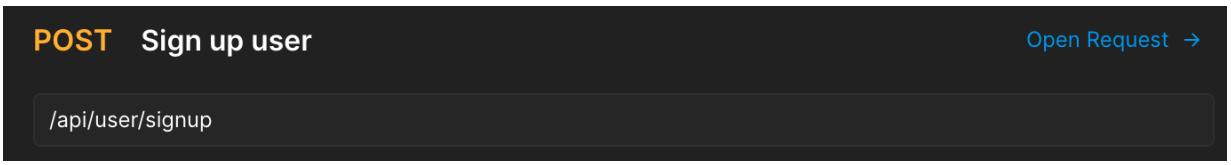
User Service

The User Service is a microservice responsible for authenticating users accessing PeerPrep. Through this service, users can create an account, login, update their passwords and delete their account.

Upon login, a JWT will be sent in the response. The JWT will then be used for authentication of various APIs, including updating passwords and deleting accounts. The same authentication process will be used across different microservices.

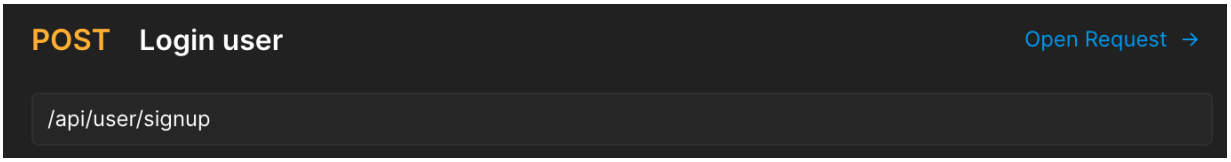
Endpoints

The following are the currently supported endpoints for the user-service:



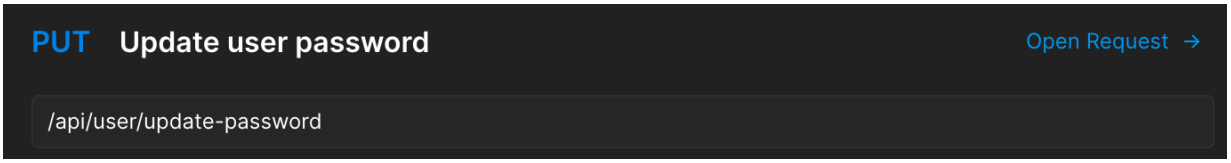
body:

```
{
   "username": "",
   "password": ""
}
```



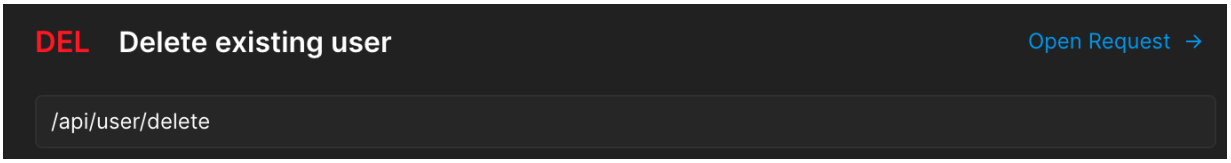
body:

```
{
   "username": "",
   "password": ""
}
```



body:

```
{
   "username": "",
   "oldPassword": "",
   "newPassword": ""
}
```



body:

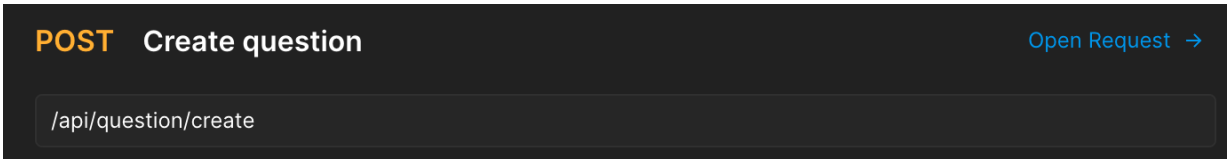
```
{
   "username": "",
   "password": ""
}
```

Question Service

The Question Service is a microservice responsible for creating and retrieving questions from the database. Although the question creation and deletion APIs will not be used by end users, it was more efficient to create the APIs and use them to create and delete questions, instead of directly creating and deleting on MongoDB.

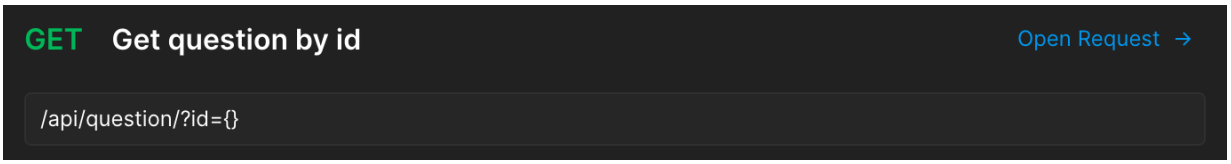
Endpoints

The following are the currently supported endpoints for the question-service:

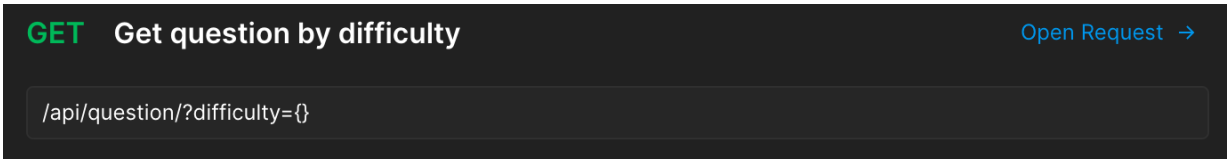


body:

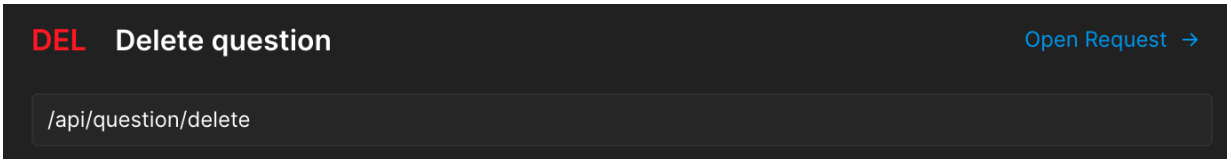
```
{
   "difficulty": "",
   "title":"",
   "examples": [
       [
           ""
       ],
       [
           "",
       ]
   ],
   "question": "",
   "constraints": [
       "",
   ]
}
```



This API call accesses the query instead of the body of the request.



This API call accesses the query instead of the body of the request.



body:

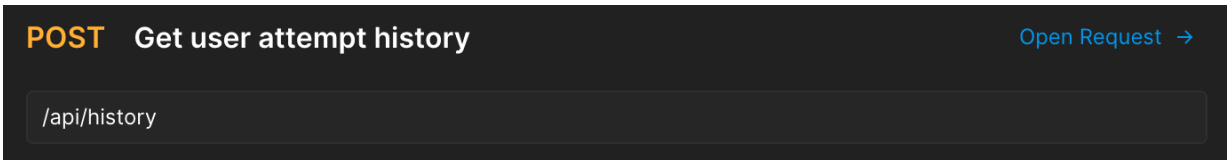
```
{
   "id": ""
}
```

History Service

The History Service is a microservice responsible for retrieving data on a user's history of questions answered and the questions' difficulty levels. This microservice is used when users wish to view the questions they have attempted prior.

Endpoints

The following are the currently supported endpoints for the history-service:

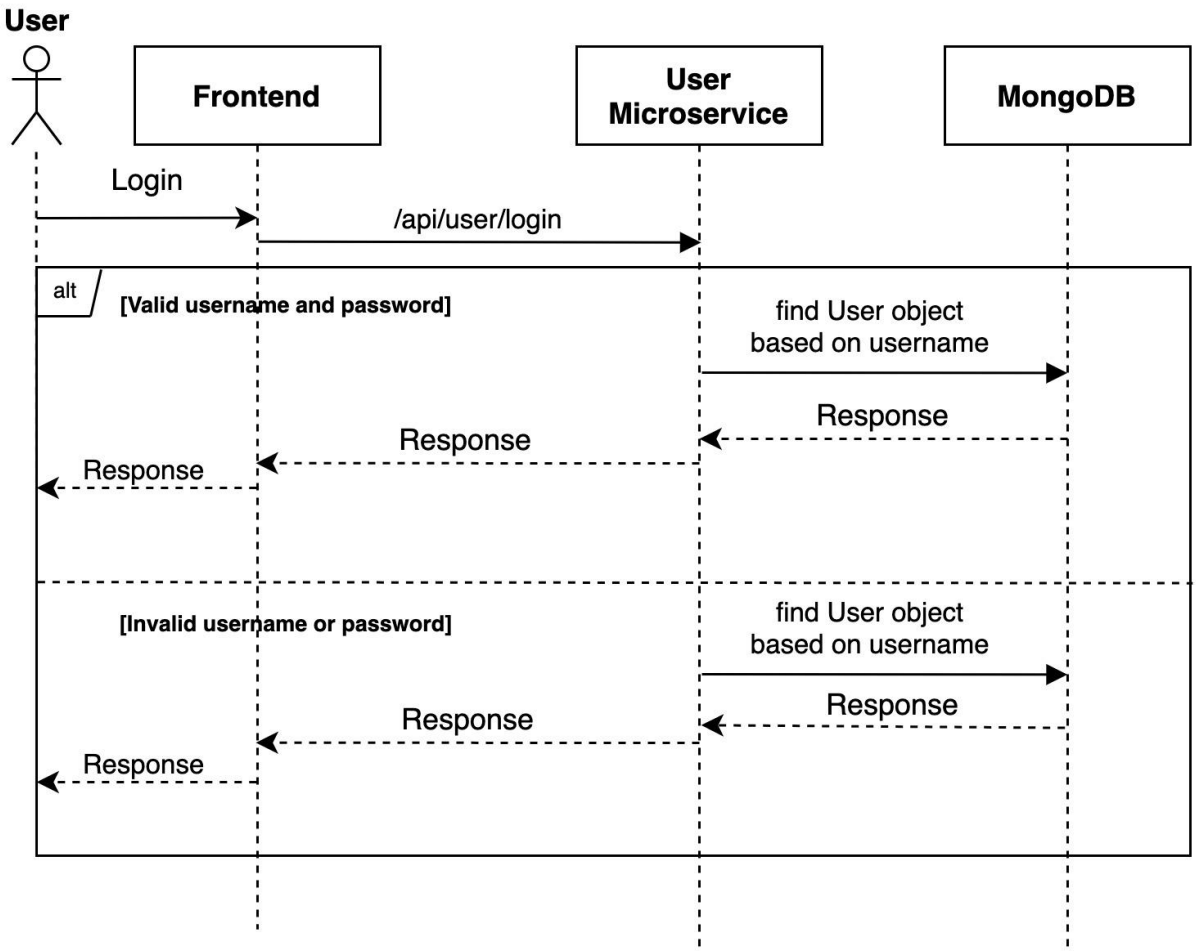


body:

```
{
   "user": ""
}
```

8. Application Flow

8.1 User Service



Sequence Diagram for user login

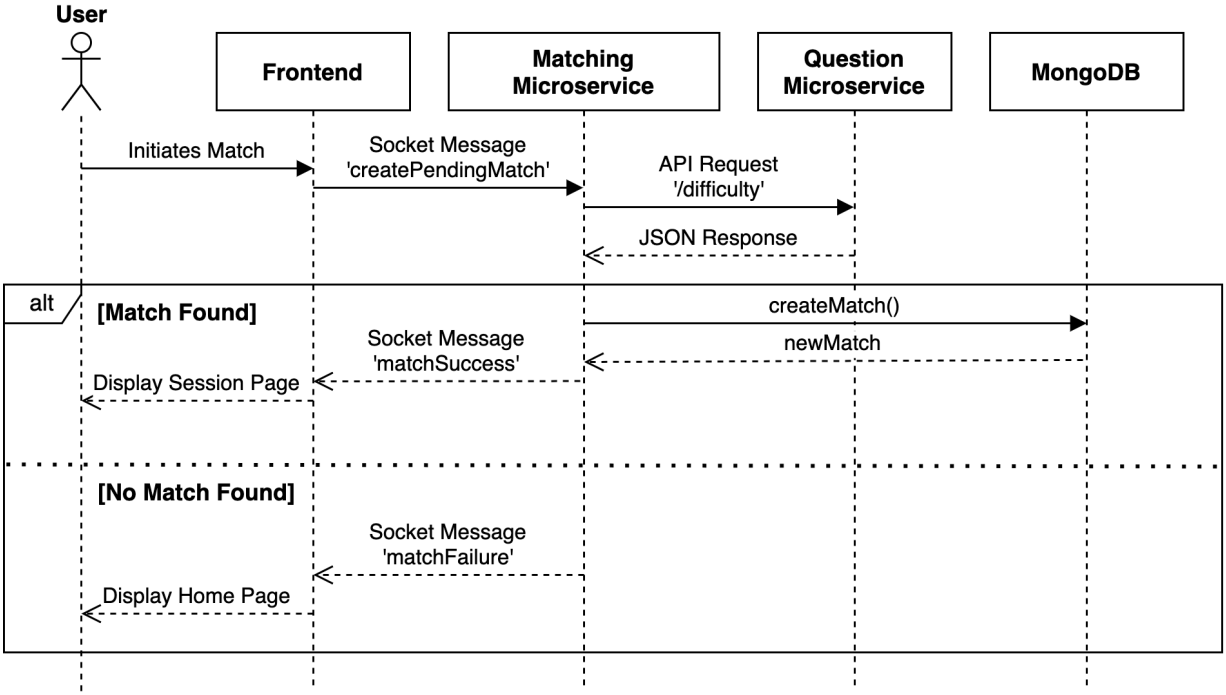
Although this section talks about the User microservice, its application flow can is similar to the Question and History microservices in its interaction with the frontend and the database.

As an example, we will use the flow of a user logging in. When the user fills up his/her username and password and clicks the login button, the frontend will make an HTTP POST request to the User Microservice. The User Microservice will first make a call to MongoDB to find the user based on the username given.

If there is an existing username, the User Microservice will then compare the encrypted passwords to see if it matches. If it matches, the User Microservice will send a 200 OK response along with the JWT that the frontend will use for future authentication during other API calls to other microservices.

This flow is shared by most of our application's features, such as getting questions, or getting the user's history. First, the frontend will make a HTTP request to a microservice. Then, the microservice interacts with MongoDB, either by getting, posting, updating, or deleting data. Finally, a response will be sent back to the frontend.

8.2 Matching Service



Sequence Diagram for Matching Process

When a user selects the option to find a match, a `createPendingMatch` message is sent to the Matching-Service via sockets, along with their `username`, `token` and `difficulty` chosen.

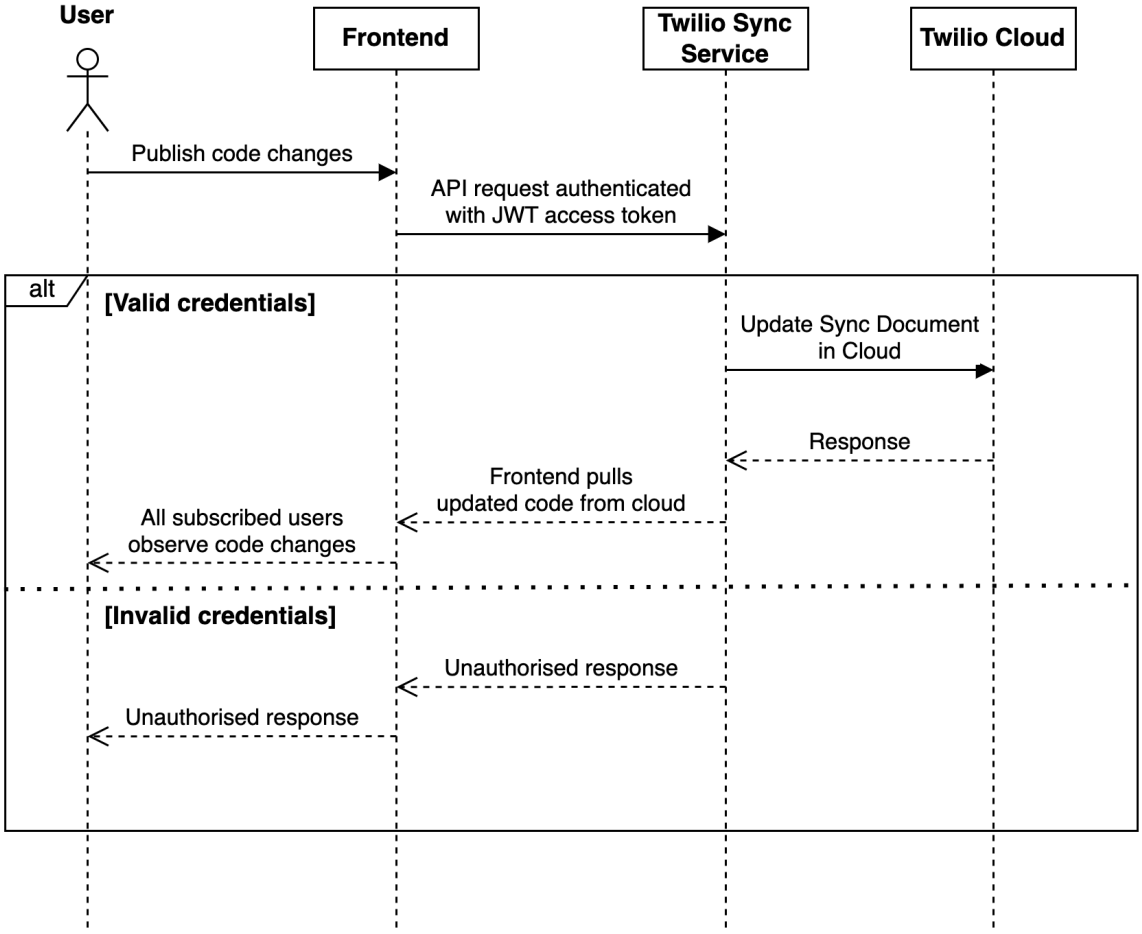
In turn, the matching-service will then make an API request to the question-service using the `token` of the user based on the difficulty the user has chosen. It will then await the JSON Response from the question-service, which is the list of questions of the specified difficulty.

In the case where another user attempts to match and has selected the same difficulty, it would constitute as **[Match Found].** The matching service will send a request to create a new document in its connected MongoDB database. The database in response will return the `newMatch` object, which consists of the data relating to that specific match instance.

The Matching-Service will then send a `matchSuccess` signal to both clients to indicate that the matching was successful. This in turn will cause the frontend to navigate to the Session Page (`/session`).

On the other hand, in the case where no match was found, the Matching-Service will send a `matchFailure` to the client, which in turn will cause the frontend to navigate back to the Home Page (`/home`).

8.3 Code Sync

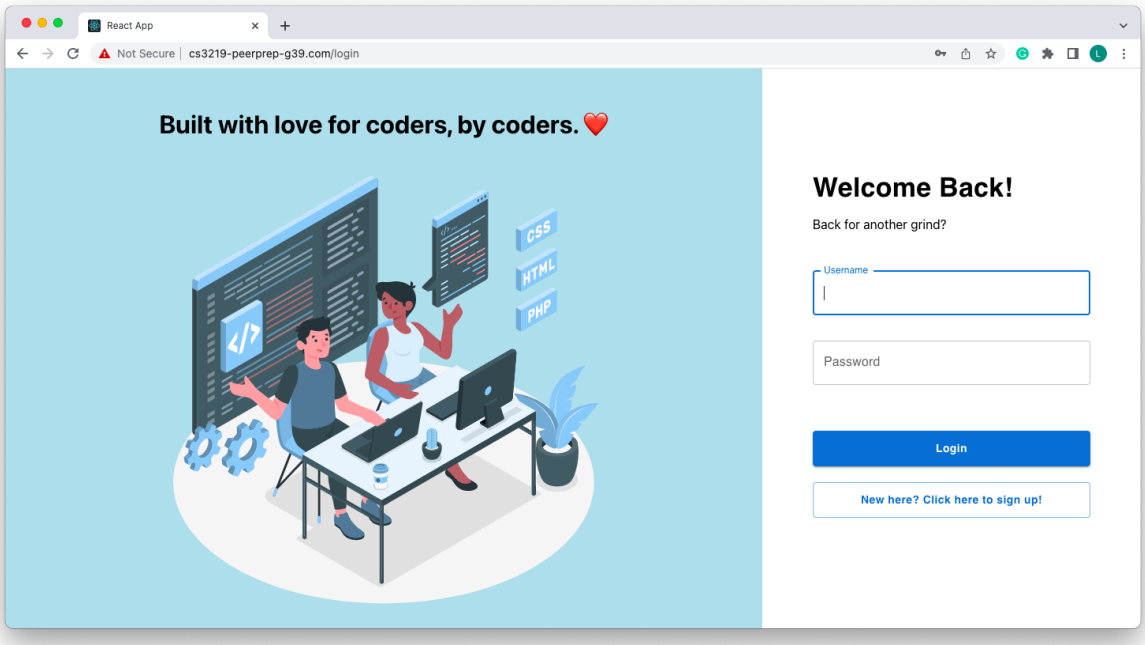


Sequence Diagram for Code Sync via Twilio Sync

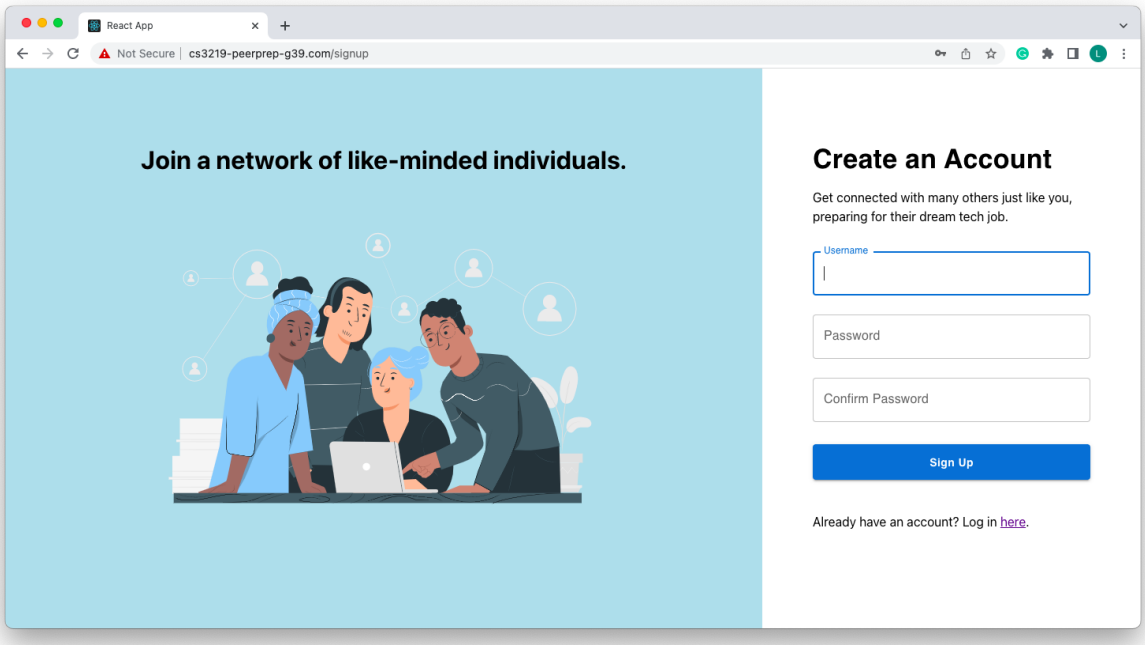
Listeners are subscribed to the same topic, which is the sync document uniquely identified by the room ID on a successful match. On published code changes from either of the users, an API request is sent to the Twilio Sync service. The user's access credentials are then verified, generating a JWT access token to the Twilio Sync service.

Upon verification, the sync document in the Twilio Cloud is then updated at real time on publication of code changes from either user. The frontend then pulls the updated code and displays it to all users in the same session.

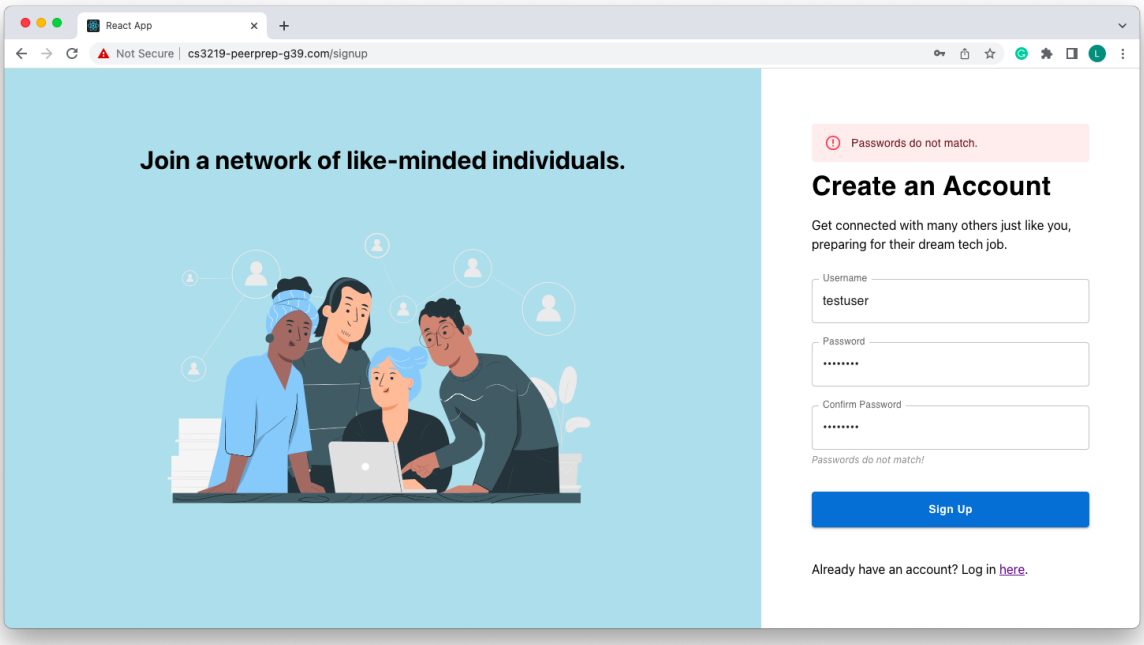
9. Frontend Deployment



Login/Default Landing Page

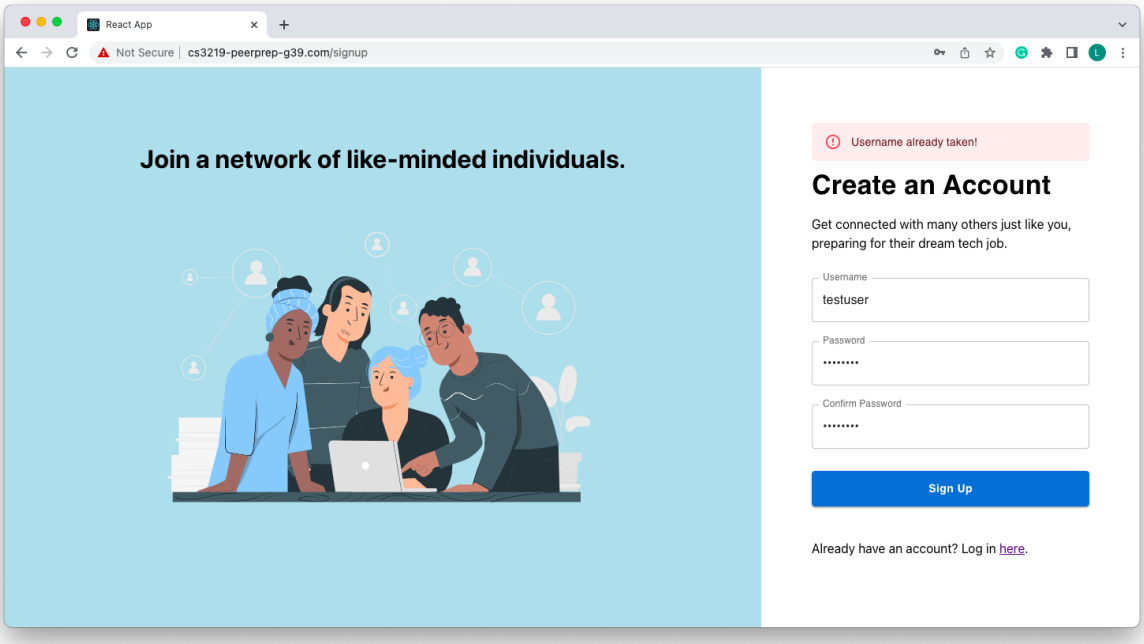


Account Creation Page



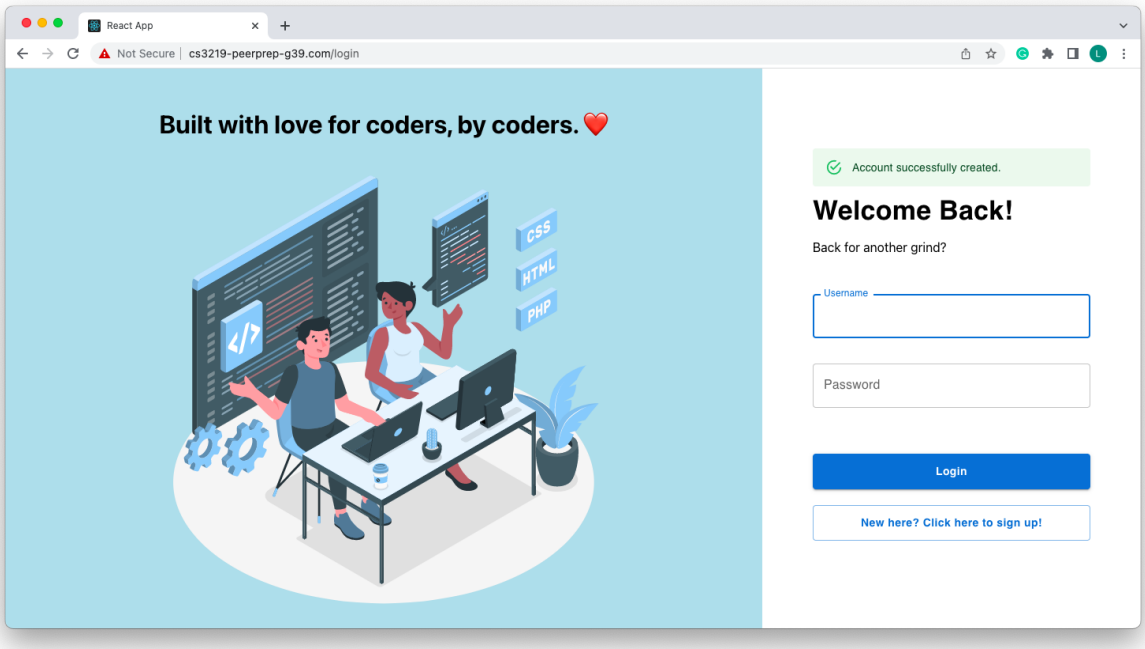
Account Creation with Mismatched Passwords

The application will automatically indicate if a user has entered passwords which do not match. If the user still attempts to sign up despite this warning, the account creation will be unsuccessful and an alert banner will appear to indicate the error.



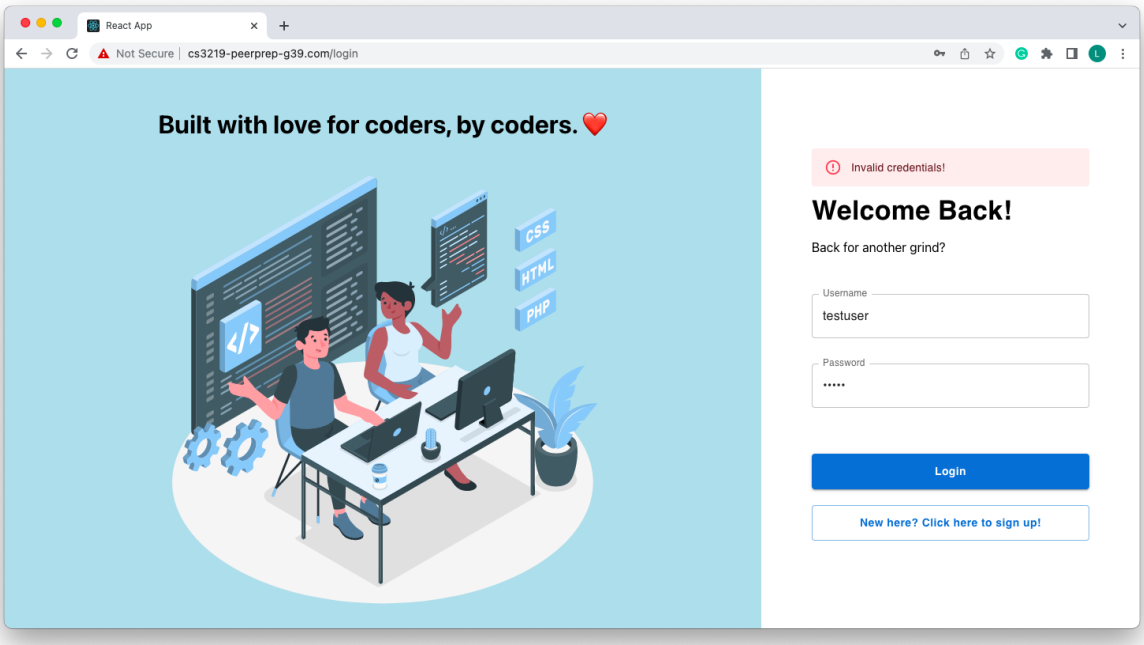
Account Creation with Existing Username

In another instance if the user attempts to create an account with an already existing username, the account creation will still be unsuccessful and an alert banner will appear to indicate the error.



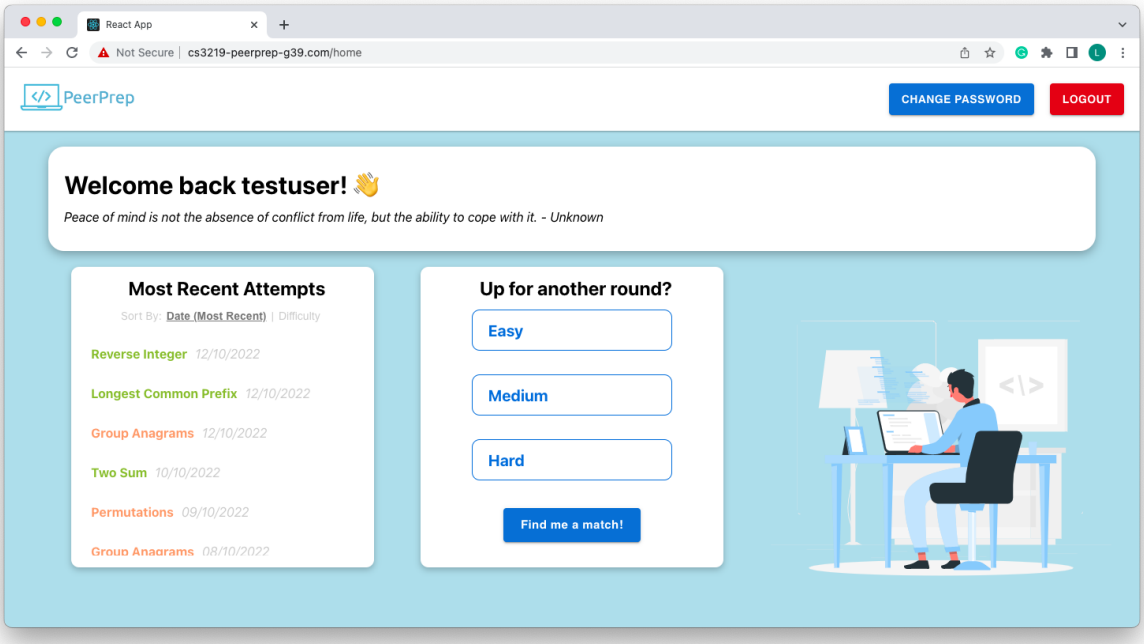
Redirection to Login Page Upon Successful Creation

If the account creation is successful, the user will then be redirected back to the Login Page, with a banner notifying the user of the successful account creation.



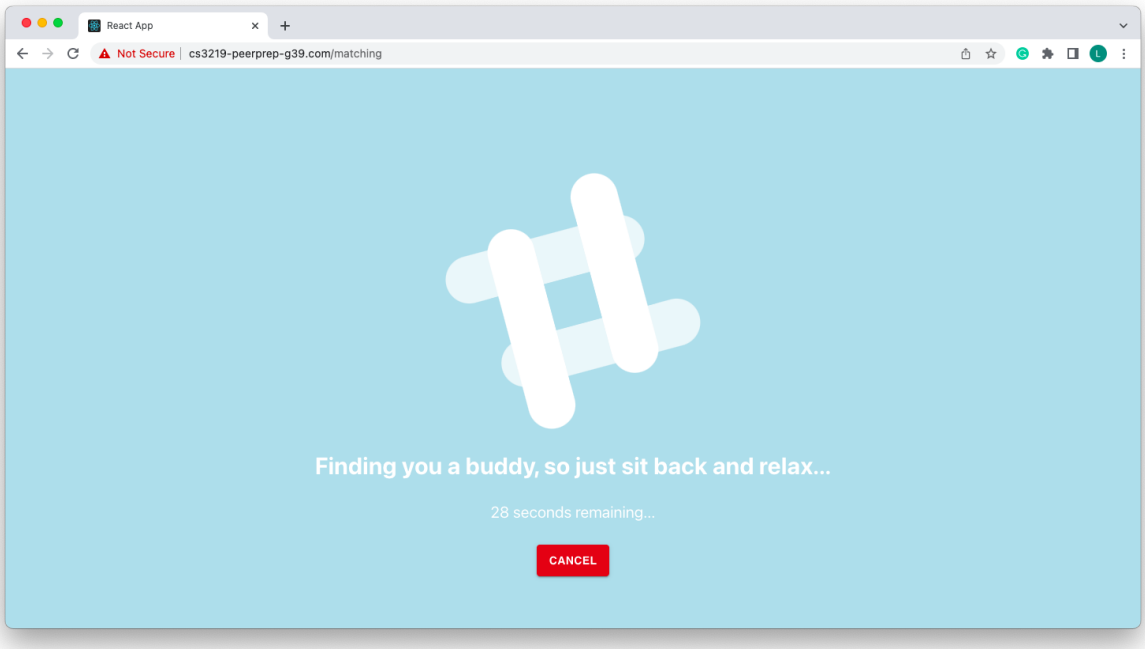
Unsuccessful Login with Invalid Credentials

The client will also alert the user if they have attempted to sign in using invalid credentials, such as a non-existent username, or an invalid password.



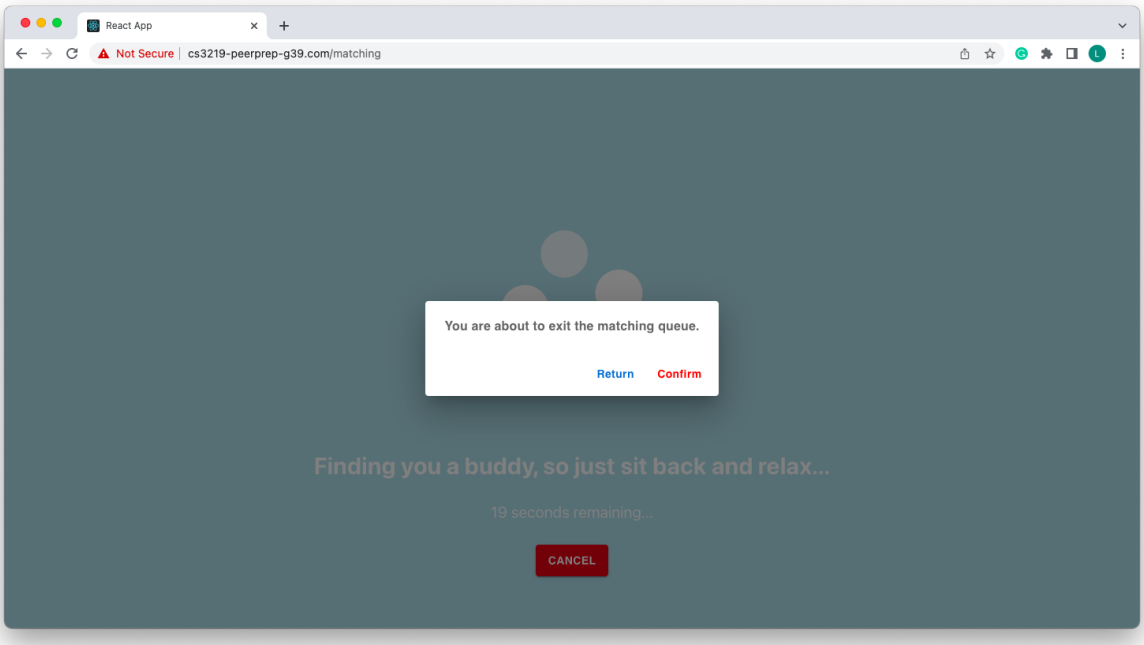
Home Page

On the Home Page, users can view all the previously attempted questions and sort them chronologically, or by difficulty. Users can also select one of the 3 given difficulties and once they click on the “Find me a match!” button, they will be sent to the Matching Page, as shown in the diagram below.



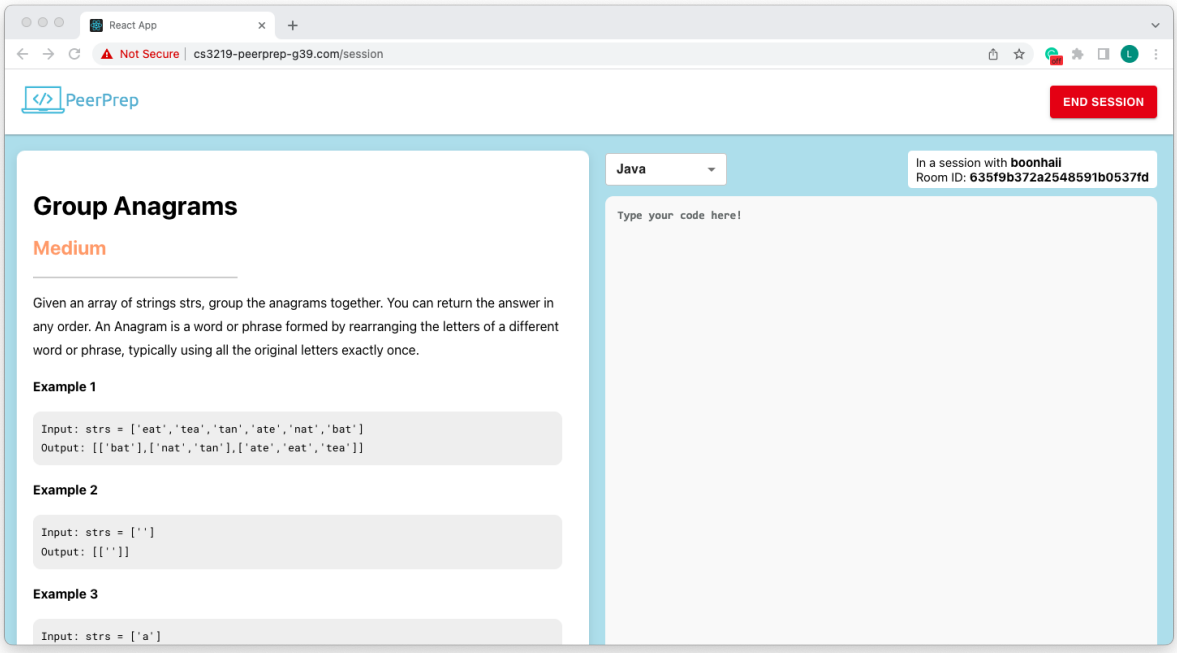
Matching Page

Users will only remain on the matching page for a maximum of 30 seconds, after which it will timeout. If the matching process times out, the user will be redirected back to the home page.



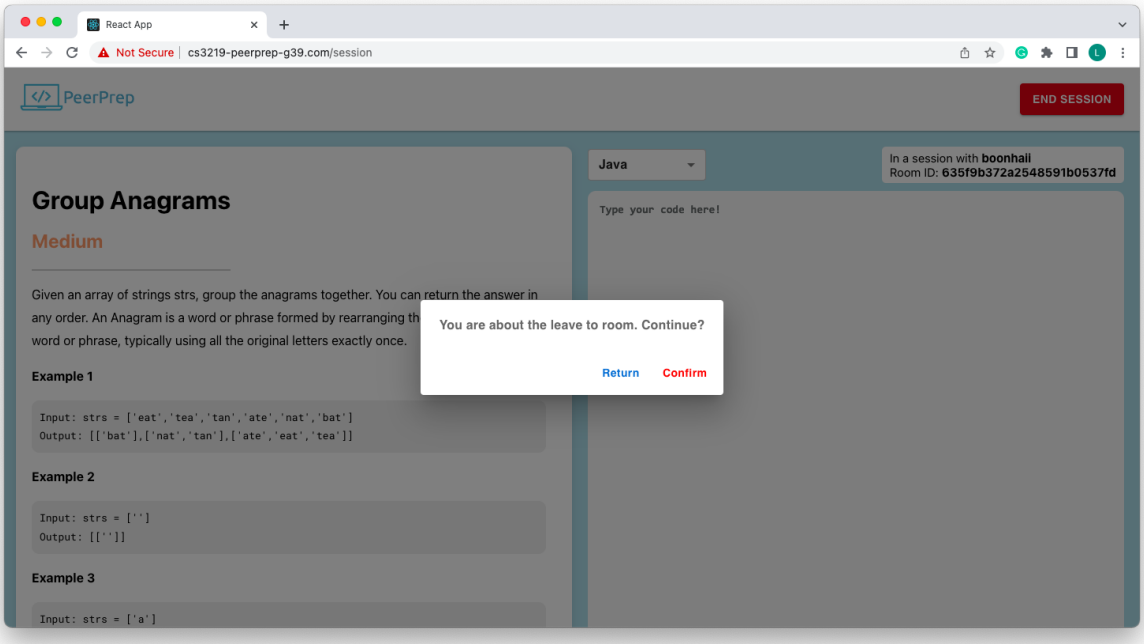
Confirmation Dialog to exit matching queue

To ensure that Users do not accidentally exit the matching queue, a confirmation dialog is presented when the user clicks on the “Cancel” button in an attempt to cancel the matching process. Upon confirmation of cancellation or unsuccessful matching, users will be returned to the home page.



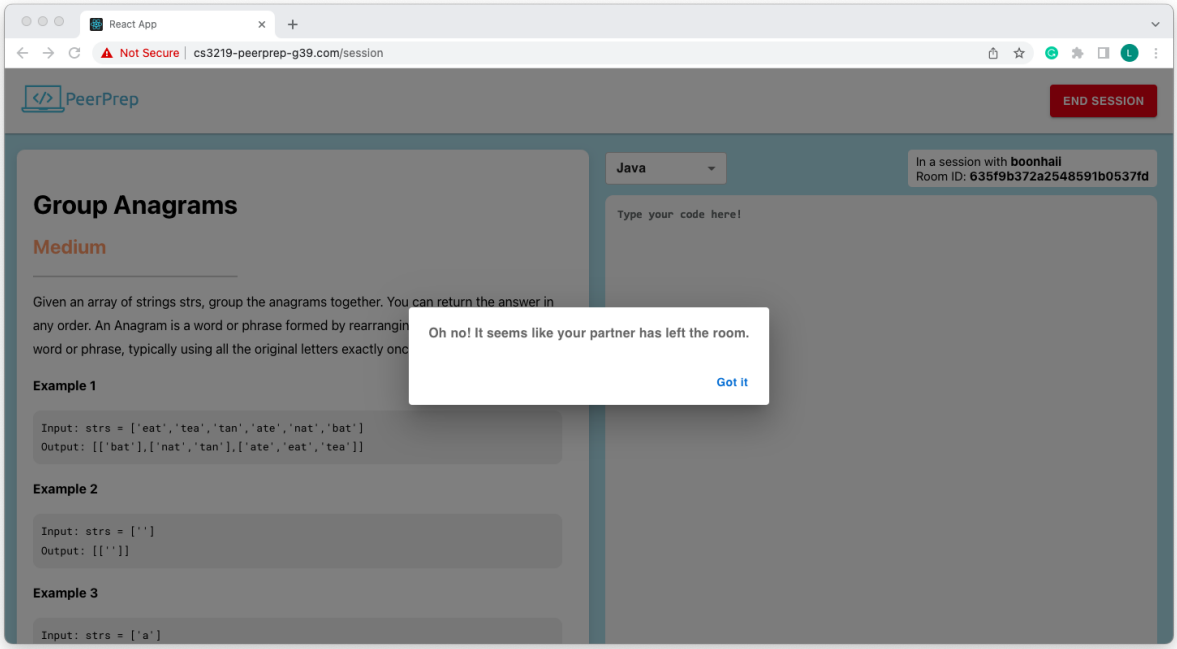
Session Page

If the match was successful, users will be automatically redirected to the session page.



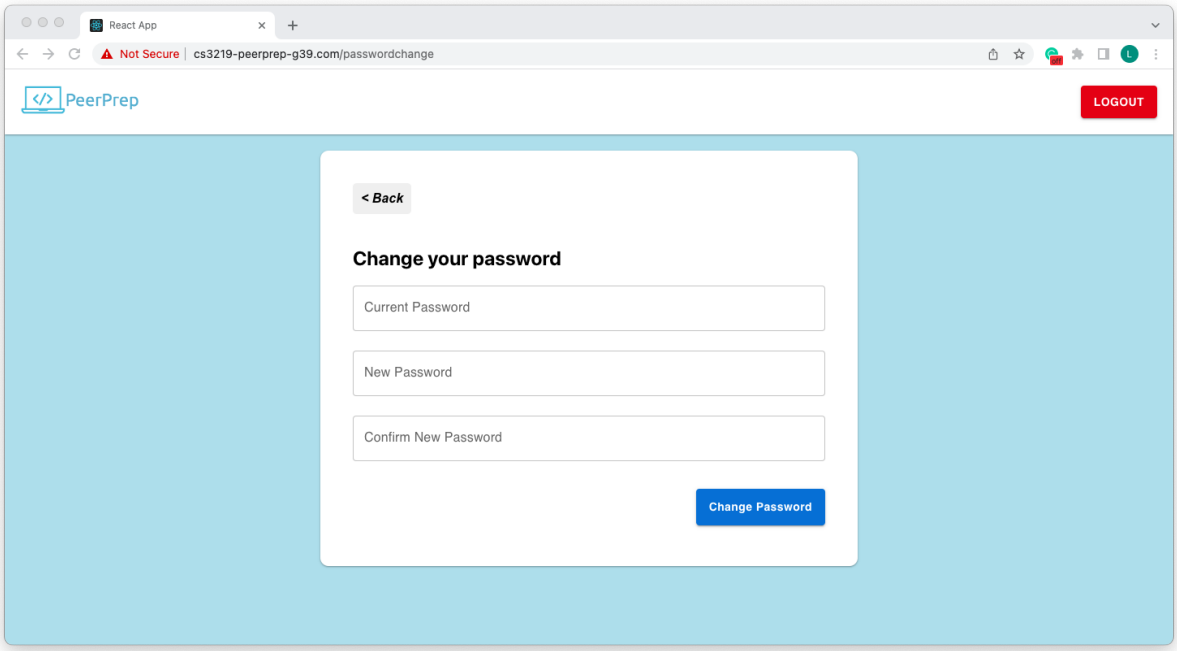
Confirmation Dialog to exit Session

To ensure that Users do not accidentally exit the session, a confirmation dialog is presented when the user clicks on the “END SESSION” button in an attempt to exit the session. Upon confirmation of their intention to exit the room, the initiating user will be returned to the home page.



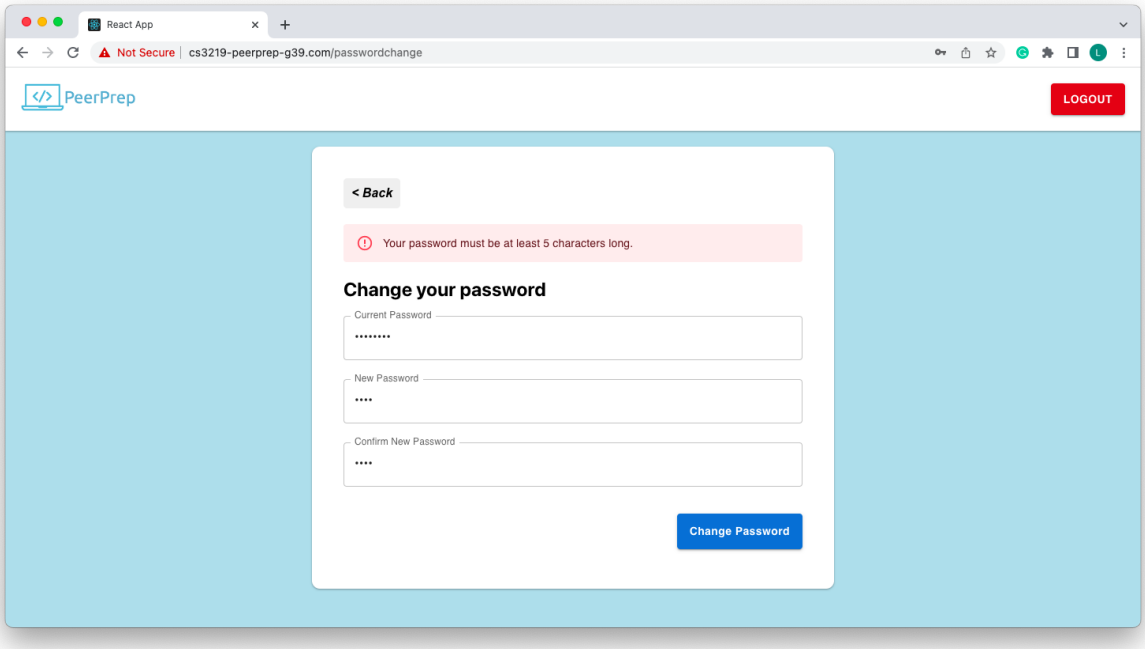
Alert Dialog In Instance of Partner Leaving Room

On the other hand, the remaining user in the session will have a dialog to notify the user that their partner has left the room.



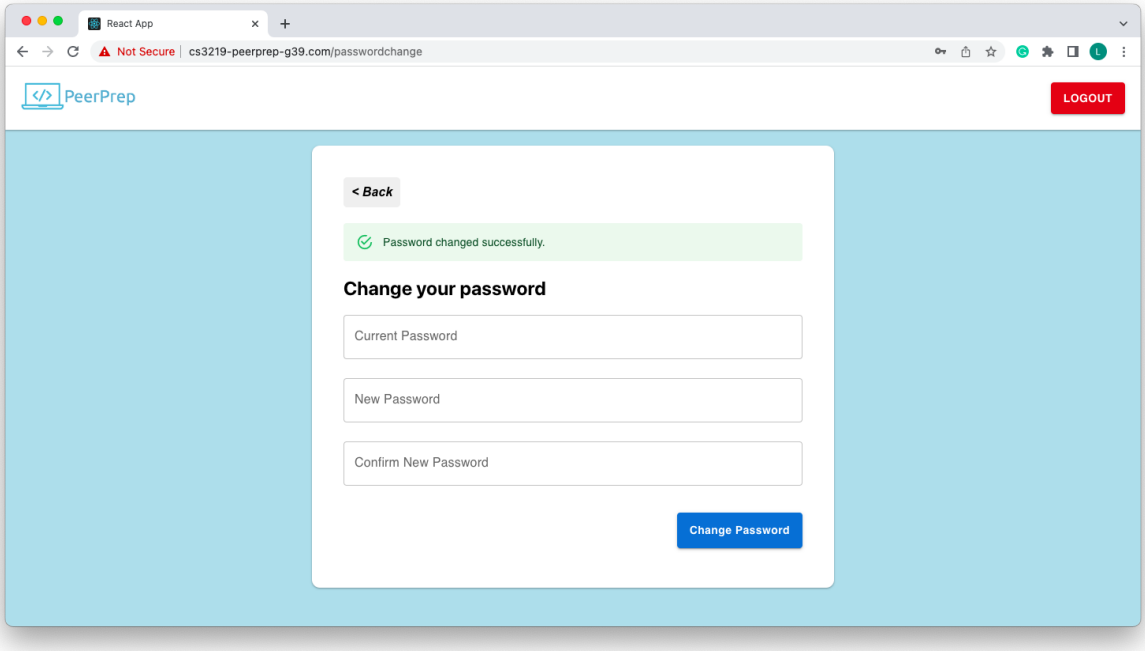
Password Change Page

On the home page, the User can also decide to change their password. This can be done by clicking the “Change Password” button on the top right, which would lead to the page as shown above.



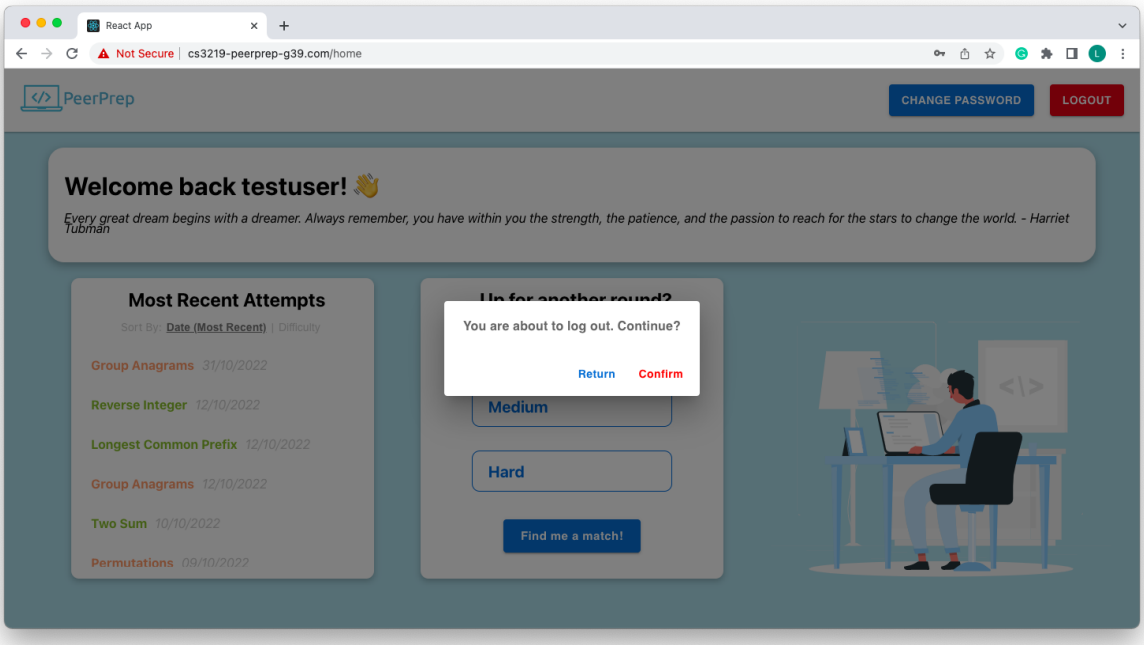
Unsuccessful Password Change due to Password Length

To be able to change password, the User is required to enter their current password for authentication, as well as their new password. Similar to User Creation, the same requirements for password apply. If the User enters a password which does not meet the length requirements, an error banner would be shown, similar to the above.



Successful Password Change

On the other hand, if the password meets the requirements, the password change would be successful, and a successful notification banner would be shown, similar to the above.



Confirmation Dialog For Logging Out

Finally, once a user has completed their session and wishes to log out, the User can return to the home page and click on the logout button. A confirmation dialog will appear to confirm that the user wishes to log out. Once the user has logged out, they will be redirected back to the login page.