**NATIONAL UNIVERSITY OF SINGAPORE**
SCHOOL OF COMPUTING



AY 21/22, SEMESTER 1
CS3219: Software Engineering Principles and Patterns

Lee Xuan Wei, Jeremy A0205807H
Swa Yong Shen A0199852J
Tan Jun Wei A0206143U
Yeoh Hsin Ying Candice A0206107U

Team G5

Project Report: PeerPrep

# Contributions

| Name | Technical Contributions | Non-technical Contributions |
|------|------------------------|----------------------------|
| Lee Xuan Wei Jeremy | <ul><li>Questions microservice</li><li>Authentication</li><li>DevOps (Kubernetes, autoscaling, Dockerisation, deployment)</li></ul> | <ul><li>Requirements documentation</li><li>Project Report</li></ul> |
| Swa Yong Shen | Implement all frontend pages<ul><li>Login</li><li>Logout</li><li>Interview</li><li>Home page</li><li>CRUD questions page</li><li>Tutorial page</li></ul> | <ul><li>Requirements documentation</li><li>Project Report</li></ul> |
| Tan Jun Wei | <ul><li>Match microservice</li><li>Chat microservice</li></ul> | <ul><li>Requirements documentation</li><li>Project Report</li></ul> |
| Yeoh Hsin Ying Candice | <ul><li>User microservice</li><li>Editor microservice</li></ul> | <ul><li>Requirements documentation</li><li>Project Report</li></ul> |

# Introduction

Many students struggle with technical interviews and may not have a platform to practice effectively. As computer science students, we understand the importance of being well-prepared for technical interviews, as they are an integral part of the hiring process for computer scientists, be it in the software engineering field or beyond.

As such, with PeerPrep, we enable students to collaborate and support each other by empowering them with a platform to practice on coding questions with a range of difficulties. Users are matched on a 1-1 basis, and the app enables students to communicate via an in-app chat system, and collaborate through a real-time text editor. The app also supports basic features like authentication.

# Project Timeline

## Schedule

| | Jeremy | Yong Shen | Candice | Jun Wei |
|---|---|---|---|---|
| Weeks 5 - 6 (6 Sep - 17 Sep) | Requirements Churning Architectural Design of Application | | | |
| Week 7* (1 Oct - 6 Oct) | Set up backend routes for Questions microservice | Design Frontend prototypes | Set up backend routes for User microservice | Set up backend routes for Match microservice |
| Week 8 (7 Oct - 13 Oct) | Set up authentication for microservices | Implement static web pages for Frontend | Set up logic for user registration and login | Set up logic for match queueing and interview |
| Week 9 (14 Oct - 20 Oct) | Set up Kubernetes cluster | Implement interview page for Frontend | Set up backend for Text microservice | Set up backend for Chat microservice |
| | | Implement question layout for Frontend | | |
| Week 10 (21 Oct - 27 Oct) | Set up Kubernetes cluster | Implement tutorial and CRUD questions page | Integrate Redis Pub/Sub | Integrate Redis Pub/Sub |
| | | Integrate Frontend with User component to test user registration and login Integrate Frontend with Match component to test finding match | | |
| Week 11 (28 Oct - 3 Nov) | Deployment on Google Cloud and set up autoscaling | Integrate Frontend with Text component to test simultaneous edits Integrate Frontend with Chat component to test sending messages | | |
| | | Report Writing | | |
| Week 12 (4 Nov - 11Nov) | Bug fix Report Writing | | | |

(* Before working on the individual microservices, we have also discussed and come up with API endpoints for each microservice)

# Software Development Life Cycle

Unlike a real-world project, the requirements for this project were already very well defined, and there will be little to no changes to project requirements. Furthermore, having also seen prior demonstration of another PeerPrep app in lecture, we were also well aware of how the app would be used. Our group hence adopted the *waterfall* model which allows us to develop our application in a highly structured and steady manner, resulting in a timely and systematic development effort that ensures our project is completed on time.

This enables a well defined outcome every stage, leaving some artifacts to be used in the next stage of production. Below is an overall timeline of our project based on the waterfall SDLC model.
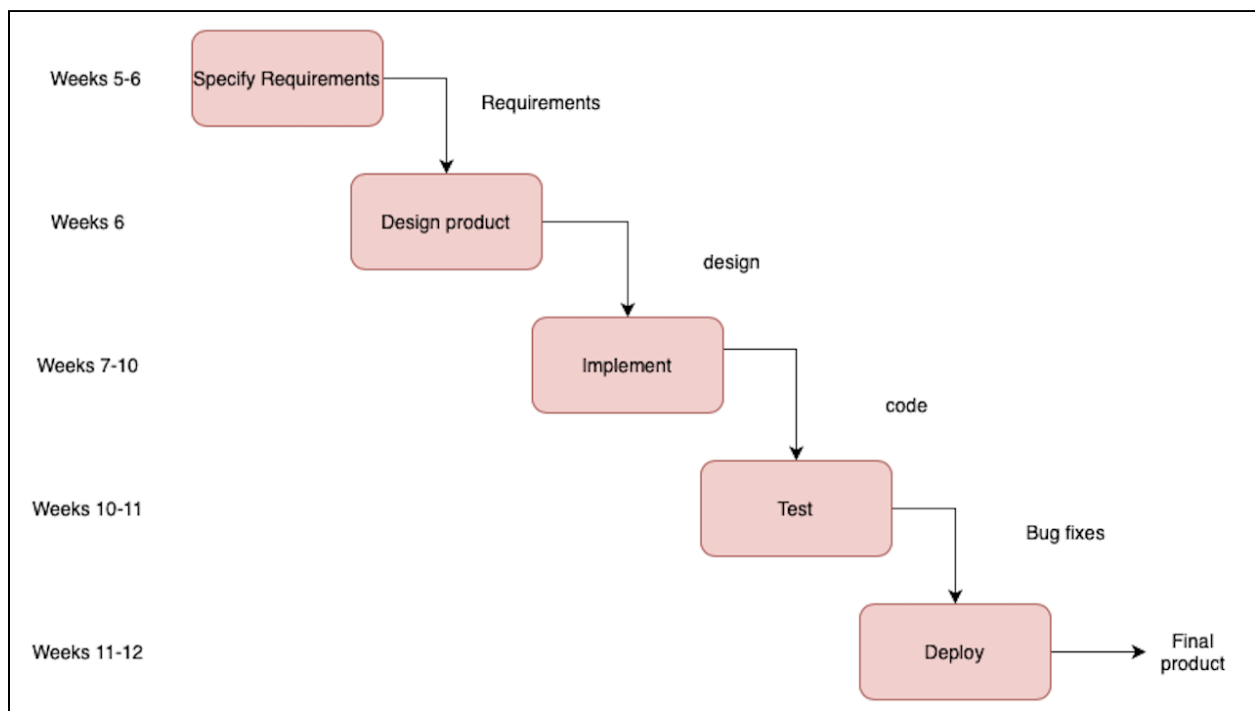


We also held a meeting at the beginning of each week to:
- discuss any changes to the API for each microservices
- reflect on what each member did in the previous week
- allocate tasks for the upcoming week

# API

To ensure that our microservices can communicate and send information to each other, we defined concrete API response formats. Microservices can use this API to communicate with other microservices without knowing the internals of the other microservices.

To provide a consistent format, we adopted the [JSend](#) specification for the JSON responses from our backend microservices. JSend provides a structured response body for the 3 main types of responses.

For a successful response, we adopted the following structure, with a `status` field to indicate success and a `data` field to contain the data associated with the response:

```
response:
{
    status: "success",
    data: {
        ...
    }
}
```

For a failed response, we adopted the following structure, with a `status` field to indicate failure and a `data.message` field to specify the failure message:

```
response:
{
    status: "failed",
    data: {
        message: "failure message"
    }
}
```

For an error response, we adopted the following structure, with a `status` field to indicate error and an `error_message` field to specify the error message:

```
response:
{
    status: "error",
    error_message: "error message"
}
```

# Functional Requirements

**Account Management**

| ID | Functional Requirement | Priority |
|---|---|---|
| AM-1 | Users must be able to register a user account with a name, email address and password. | High |
| AM-2 | Users must be able to login to their user account with a registered email address and password. | High |
| AM-3 | Users should be able to log out. | High |
| AM-4 | The system should be able to authenticate a user account using an email and a password. | High |
| AM-5 | The system should be able to authenticate a user account using a JWT token. | High |
| AM-6 | The system should be able to authorize an admin user account using a JWT token. | Medium |

**Match System**

| ID | Functional Requirement | Priority |
|---|---|---|
| MS-1 | Users should be able to find a match for a practice interview for a chosen difficulty level. | High |
| MS-2 | Users should be able to match with another user who has chosen the same difficulty level. | High |
| MS-3 | System will stop finding a match if no match is found for the user within 30 seconds. | High |
| MS-4 | System should prevent the user from finding more than one different match at any time. | High |
| MS-5 | System should inform the user that he has cancelled finding a match in another browser. | Medium |
| MS-6 | System should inform users to come back later if the limit of concurrent interviews has been | Medium |

| ID | | Priority |
|---|---|---|
| | reached. | |
| MS-7 | System should be able to track the number of concurrent interviews being held. | Medium |

**Interview Process**

| ID | Functional Requirement | Priority |
|---|---|---|
| IP-1 | Users should be able to choose the difficulty level of the question to attempt. | High |
| IP-2 | Users should be able to end a collaboration session. | High |
| IP-3 | System should resume a user's practice interview if the user reloads the page. | High |
| IP-4 | System should allow a user to resume his practice interview if he visits another route in the application. | High |
| IP-5 | Matched users should be able to read the given practice question. | High |
| IP-6 | The given practice question should be rendered in Github markdown style. | Low |
| IP-7 | Matched users should be able to input their responses in a shared editor field at the same time. | High |
| IP-8 | Users should be able to see their match's typing/response in a shared editor field in real time. | High |
| IP-9 | Users should be able to select their preferred programming language for the text editor. | Low |
| IP-10 | The text editor should support Python, C++ and Java. | Low |
| IP-11 | Application should end the user's interview if the user does not have any new keystroke in the text editor within 10 minutes. | Low |

| IP-12 | System should provide a chat widget for two matched users to communicate with each other via instant messaging. | High |
| --- | --- | --- |
| IP-13 | System should inform users that their matched partner has connected to the chat | Low |
| IP-14 | System should inform users via the chat widget that their matched partner has disconnected from the interview session. | Low |

**Administrative**

| ID | Functional Requirement | Priority |
| --- | --- | --- |
| AD-1 | Admin users should be able to create, read, update and delete practice questions. | High |
| AD-2 | Admin users should be able to set a difficulty for practice questions. | Medium |
| AD-3 | Admin users should be able to set a title for practice questions. | Medium |
| AD-4 | Admin users should be able to set a Markdown formatted description for practice questions. | Medium |
| AD-5 | Admin users should be able to preview the formatted practice question before submitting. | Low |

**System**

| ID | Functional Requirement | Priority |
| --- | --- | --- |
| AP-1 | System should provide a tutorial page with steps to use the application. | Low |

# Non-Functional Requirements

**Portability**

| ID | Non-Functional Requirement | Justification | Implementation | Priority |
|----|---------------------------|---------------|----------------|----------|
| P-1 | System should be compatible with Edge, Firefox, Chrome, Safari. | 94% of all web users uses Chrome, Safari, Firefox and Edge (statista June 2021). For greater inclusivity, we want to support these major web browsers | - | High |
| P-2 | System should support devices of different screen sizes. | Students might want to practice algorithms on the go. Hence, the application should also support mobile devices such as tablets and mobile phones. | Bootstrap's grid system. | Low |

**System Performance**

| ID | Non-Functional Requirement | Justification | Implementation | Priority |
|----|---------------------------|---------------|----------------|----------|
| SP-1 | Users should be able to login within 5s. | To provide a responsive user experience. | - | Medium |
| SP-2 | Users with stable internet connection should be able to see their match's typing/response within 2s. | It is important to be able to view each person's response in real time for a seamless collaboration experience. | Socket.io for real time communication combined with a Redis Pubsub system to manage concurrency and reliability | Medium |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

**Security**

| ID | Non-Functional Requirement | Justification | Implementation | Priority |
| --- | --- | --- | --- | --- |
| SC-1 | User account password must be hashed in the database. | In cases where the database is compromised, hashed passwords will not leak sensitive information. Using a one-way hash function makes it computationally infeasible for attackers to calculate the passwords with only the hashed passwords. | The database will store user passwords encrypted with SHA-512 hashing algorithm. | High |
| SC-2 | User's communication with our backend server should be encrypted. | This is to avoid a man in the middle attack where an attacker can intercept the user's web traffic and steal his password. | Our website provides HTTPS which uses TLS encryption to encrypt the user's traffic with our backend servers. | High |

**Availability**

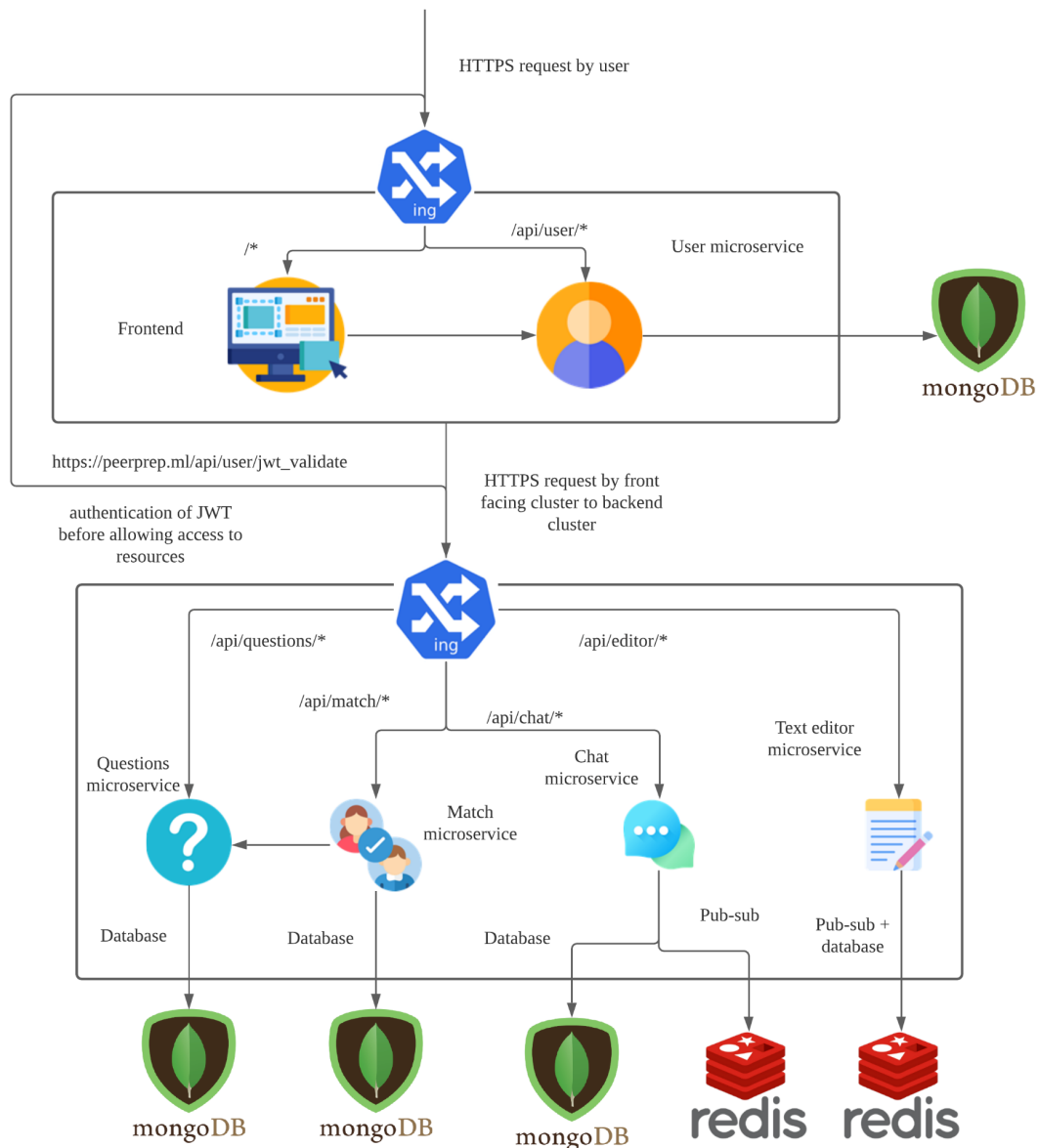| ID | Non-Functional Requirement | Justification | Implementation | Priority |
| --- | --- | --- | --- | --- |
| A-1 | Each microservice should automatically restart in the event that one of its containers shuts down unexpectedly. | As each microservice is critical to the operation of our application, we need all of them to be available at all times. | Deployment is done using Google Kubernetes Engine (GKE), using Kubernetes as an orchestration engine. This will automatically create a new | Medium |

| | | | container in the event that one fails. | |
|---|---|---|---|---|

**Scalability**

| ID | Non-Functional Requirement | Justification | Implementation | Priority |
|---|---|---|---|---|
| S-1 | The application should be able to support at least 50 concurrent users. | To allow our service to benefit a larger pool of users at one time. | Autoscaling of pods and nodes in GKE cluster. | Medium |
| S-2 | Application should be able to host 5 concurrent interviews at any point in time. | Having multiple concurrent interviews will allow a larger group of users to use this feature at the same time. | Autoscaling of pods and nodes in GKE cluster. | Medium |

# System Architecture

## Microservices Architecture

Our team has chosen the microservices architecture of our application, and the design considerations that we have discussed will be in the following sections. Below is the overall system architecture.

Based on the [Functional Requirements](#), we have separated the application into different components to separate the different responsibilities.

The **front-end** will be the main component that the users are interacting with when they visit our site.

The **user microservice** is responsible for login, registration, and JWT validation. It also stores and retrieves user-related information from its database.

The **questions microservice** will be responsible for storing all the algorithmic questions that the match and front-end might require, as well as create, read, update and delete operations for questions.

The **match microservice** handles the matchmaking of users who are looking to find a partner to do a question with. It also stores and retrieves match-related information from its database.

The **chat microservice** acts as a message broker that allows users to send messages in real-time to each other. It also stores and retrieves user messages from its database.

The **text editor microservice** allows users to work on a question collaboratively in real-time. It also stores and retrieves interview text from its database.

We will be doing a more detailed breakdown of the microservices [here](#).

# Design Considerations

Our team has identified 2 main system architectures that could be plausible for this project - monolithic architecture and microservices architecture. We will be comparing these 2 main choices in the following sections.

### 1. Scalability

| Scalability |  |
| --- | --- |
| **Purpose:**<br>● To ensure that our web application is able to handle high loads of users as defined in the requirements section<br>● Our main considerations about scalability are based on CPU and memory consumption of our components |  |
| **Choice 1: Monolithic architecture** | ● Entire system is scaled up/down together |

| | |
|---|---|
| **Choice 2: Microservices architecture** | <ul><li>Allows flexibility in scaling, such that only microservices which consume more resources are scaled up<ul><li>Helps in cost saving</li></ul></li><li>For example, even if there are many users using our application at the same time, the questions microservice, which has minimal processing in the backend, will not be consuming huge amounts of CPU or memory</li><li>Chat and text microservices have to maintain the socket connections which allow all the matched users to communicate with their partners, thus consuming more resources</li></ul> |
| **Justification:** | Microservices architecture allows for scaling that is more specific to the needs of each component. |

## 2. Coupling

| **Coupling** | |
|---|---|
| **Purpose:**<ul><li>Reduce coupling to minimise dependencies on other components in the application</li></ul> | |
| **Choice 1: Monolithic architecture** | <ul><li>Tighter coupling between different components</li><li>Changes to a single component requires redeployment of the entire application, even if the changes are not relevant</li><li>All components to use the same tech stack and language</li></ul> |
| **Choice 2: Microservices architecture** | <ul><li>Less coupling between microservices</li><li>Tests and upgrades can all be done individually without redeployment of the entire application</li><li>Each microservice can choose its own tech stack based on what is suitable e.g. Redis database vs MongoDB database</li></ul> |
| **Justification:** | Microservices architecture, so that each developer can focus more on their own microservice without having to understand the concrete implementations of other components.<br><br>Also allows for faster development, as discussions between components are done through APIs rather than having to nail down the concrete implementations, since there is minimal dependency. |

# Implementation Considerations

## 1. API Gateway

| API Gateway | |
|---|---|
| **Purpose:**<br>● Routing of request endpoints to the different microservices<br>● Set up HTTPS | |
| **Choice 1: Ingress resource in Kubernetes cluster** | ● Allows us to setup and manage HTTPS certificates<br>● Single point of entry for the cluster<br>● Routes requests to individual backend microservices |
| **Choice 2: Backend-for-Frontend (BFF)** | ● Aggregate responses from the various microservices<br>● Removes the need for an explicit API gateway that routes requests |
| **Final Choice:** | Ingress resource in Kubernetes cluster |
| **Justification:** | It is easier to set up as it is simply a configuration yaml file, as opposed to a BFF which is a backend server on its own.<br><br>Microservices can also communicate directly without leaving the cluster, using the defined ingress endpoints. |

## 2. Service Discovery

| Service Discovery | |
|---|---|
| **Purpose:**<br>● Routing of requests to the various instances | |
| **Choice 1: Kubernetes** | ● Has built-in service discovery for the various microservices<br>● GCP has native Kubernetes support (Google Kubernetes Engine (GKE)) |
| **Choice 2: Docker Swarm** | ● Has built-in service discovery for the various microservices<br>● Tighter integration with Docker images |
| **Final Choice:** | Kubernetes |
| **Conclusion:** | As beginners when it comes to orchestration engines, the availability |

| | |
|---|---|
| | of documentation and support online is particularly important. Hence, due to the growing popularity of GCP as a cloud provider, Google Kubernetes Engine (and thus Kubernetes) is our preferred choice.

Additionally, GKE has its own ingress controller, and there is no need to provide additional configurations to set up our own ingress controllers. This native support allows us to also utilise Google's managed certificates to set up HTTPS endpoints for our application. |
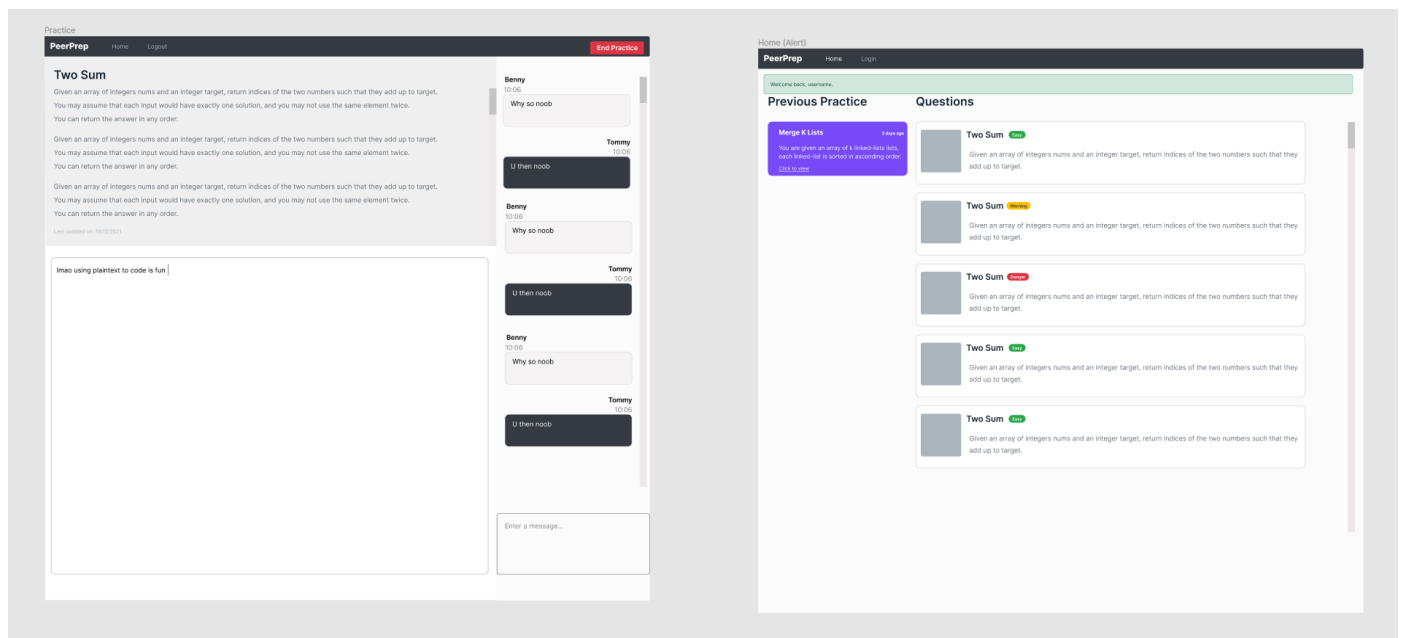
# Front end

## Tech stack

| Component | Library/framework used |
|-----------|------------------------|
| Front end | React |
| UI styling | React Bootstrap |
| Code editor | React ace |
| Chat | React chat widget |
| Question | React markdown (renders markdown as HTML), github-markdown-css (adds GitHub styling to markdown) |

## UI prototypes

The development of the front end component was prefaced with the designing of UI prototypes. We made use of an online tool, Figma to produce the UI prototypes. Following are some screenshots of the prototypes.

Link to Figma prototype:
https://www.figma.com/file/bTj0BygAlwZa83H23rBUki/CS3219?node-id=0%3A1
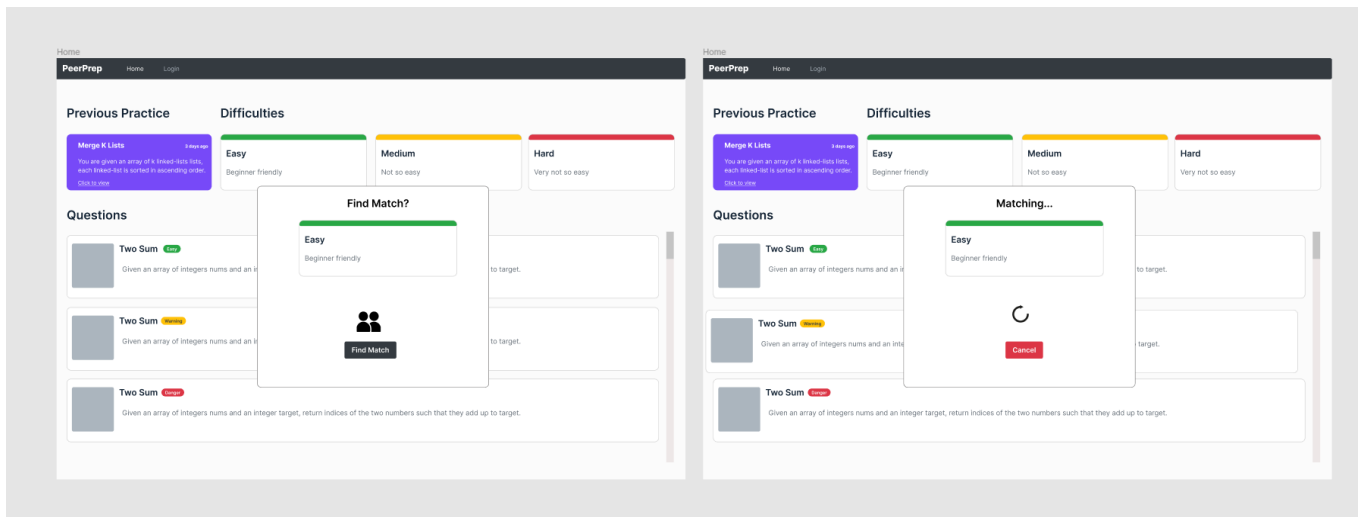
The use of prototypes greatly sped up the development process of the front end component as we did not have to come up with the designs for each page view on the fly and could rely heavily on the guidance provided by the prototypes.

## Architecture

The front end component is split into 5 main components which each represents a view in the application.
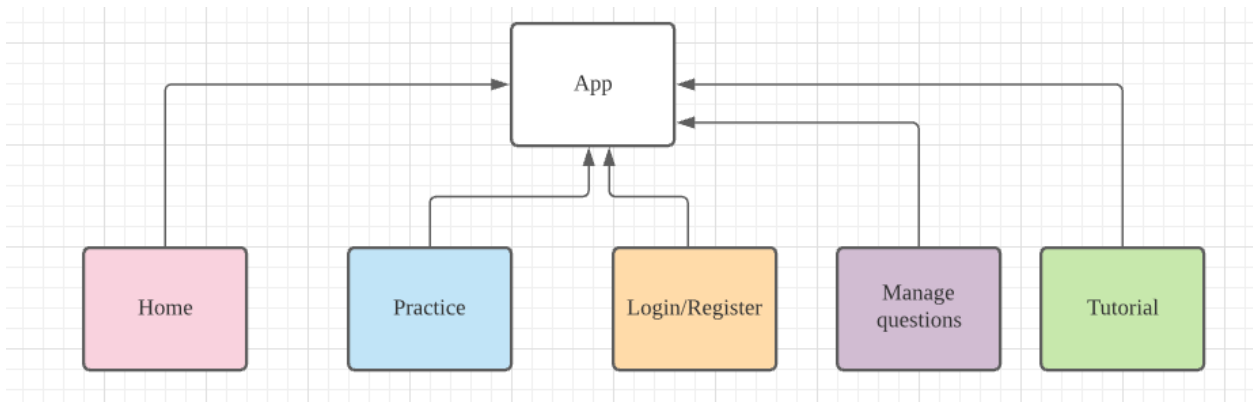


Each of the components can then be further split up into sub-components of the view that they represent. For brevity's sake, we only show how the *Home* component is split into its sub-components. The following screenshots show the three main UI components of the *Home* page.

The *Difficulty* component in the diagram below allows the user to select the difficulty of questions that he wishes to practice. While the *Question* component allows the user to preview questions that he might be assigned to for a particular question difficulty.
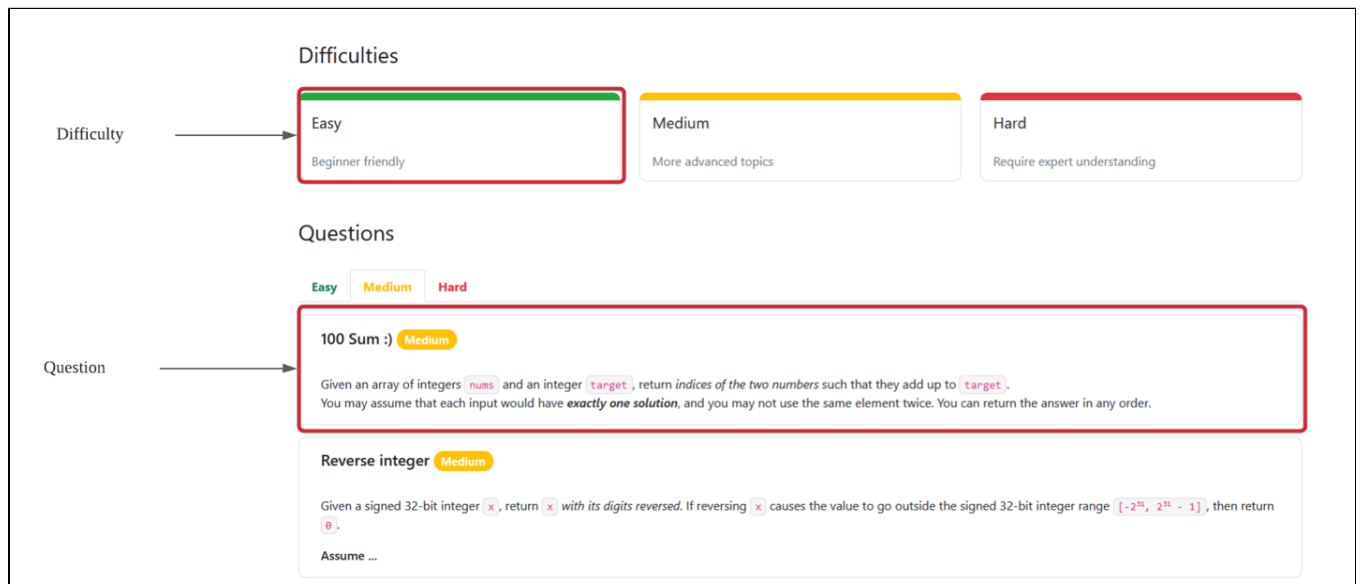
The FindMatchModel prompts the user to confirm his choice of question difficulty before finding a match for the user.
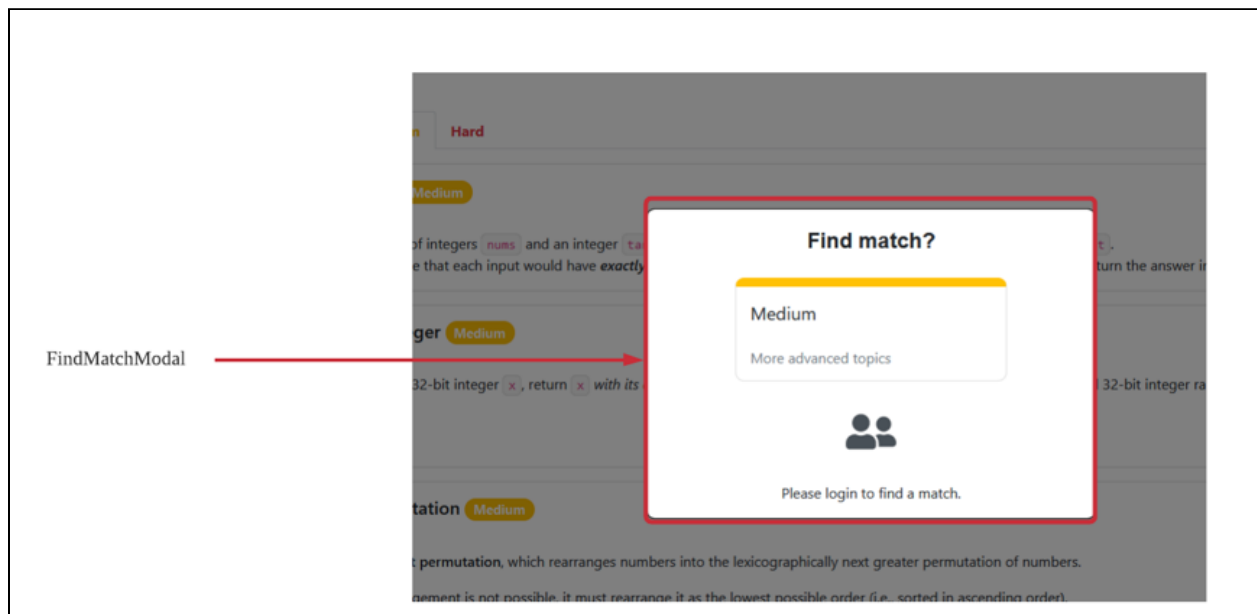


The *Home* component has 3 children components which each can have their own children components. Below is a diagram to show the relationship between the *Home* component and its children components.
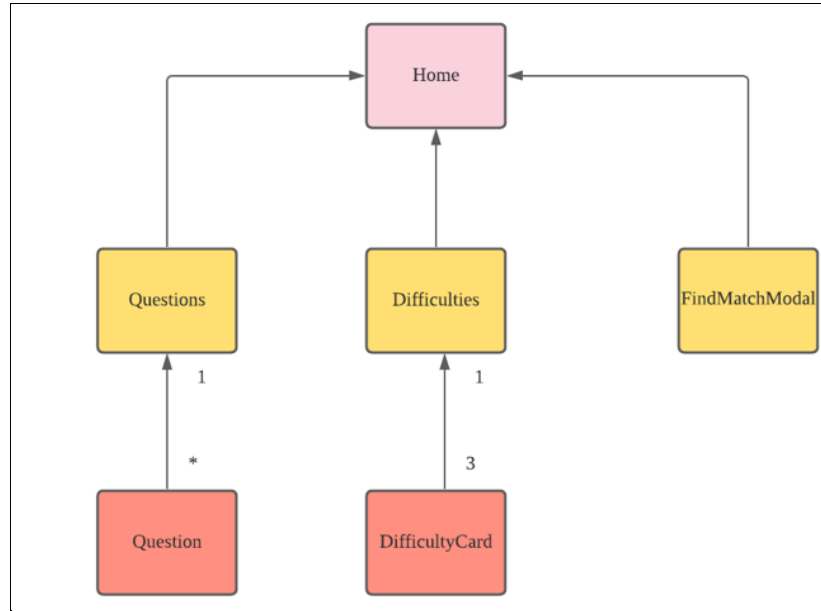
# React components

Documentation of React components, how they are related to other components and their purposes. Each colour in the table below represents components belonging to a particular page. For example, all components that are used in the homepage are colored yellow.

| Page | Component | Child of | Purpose |
|---|---|---|---|
| All | App | - | Manages routing. Contains global states that are passed around using React's *useContext*. Runs checks that are necessary on page load, such as user login check and whether a user is in a match. |
| Home | Home | App | The homepage of the website. |
| | Difficulties | Home | Wrapper component of the question difficulty selection cards. |
| | Difficulty | Difficulties | A difficulty selection card that a user can click to find a match for a practice interview for questions of a particular difficulty. |
| | FindMatchModal | Home | Prompt that is shown when a user |

| | | | selects a difficulty. Asks the user to confirm his intention to find a match.

Handles network calls with the *Match* microservice. |
|---|---|---|---|
| | Questions | Home | Wrapper component of question cards. |
| | QuestionCard | Questions | A card that displays a preview of the description for a particular question. |
| | QuestionModal | Home | A pop up that displays an entire question when a user clicks on a *QuestionCard*. |
| Login, Register | LoginRegister | App | The login or register page of the website. Login and register share a component as they are largely similar.

Handles logic of user authentication including making network calls to the *User* microservice and storing and deleting JWT token from localStorage. |
| Manage questions page (CRUD) | ManageQuestions | App | The page used to create, read, update and delete algorithm questions. |
| | QuestionsTable | ManageQuestions | A table that displays all questions in the application. A user can click on a question to edit it. |
| | QuestionEditor | ManageQuestions | A form that allows the user to edit a question or create a question. Also contains a preview to display the Markdown rendering of the question description entered by the user. |
| Interview page | Practice | App | The mock interview page. |
| | Editor | Practice | Contains a code editor using the React ace library.

Makes network calls to establish a socket connection with the *Text* |

| | | | |
|---|---|---|---|
| | | | microservice and handles other events related to input from the text editor.<br><br>Also handles timeout event when user remains inactive for more than 10 minutes. |
| | Chat | Practice | Contains a chat widget using the [React chat widget](#) library.<br><br>Makes network calls to establish a socket connection with the *Chat* microservice and handles events related to the receiving and sending of text messages. |
| Tutorial | Tutorial | App | The tutorial page. Renders a Markdown document containing the steps to use PeerPrep. |

# Microservices

## User Microservice

The user microservice manages user account related information, it supports user account registration, authentication of user account information like relevant user login and password, JWT validation. It also supports accounts with different permission levels (level 1 for normal users, level 2 for admin user). Furthermore, due to the sensitivity of information the user microservice manages, user passwords are hashed using the SHA-512 algorithm, before being stored in the database. User account related information like name, email, hashed password and permission level are stored in the database of the microservice.

### Tech stack

| Component | Library/framework used |
|-----------|------------------------|
| Backend | Node.js with Express framework |
| Database | MongoDB |
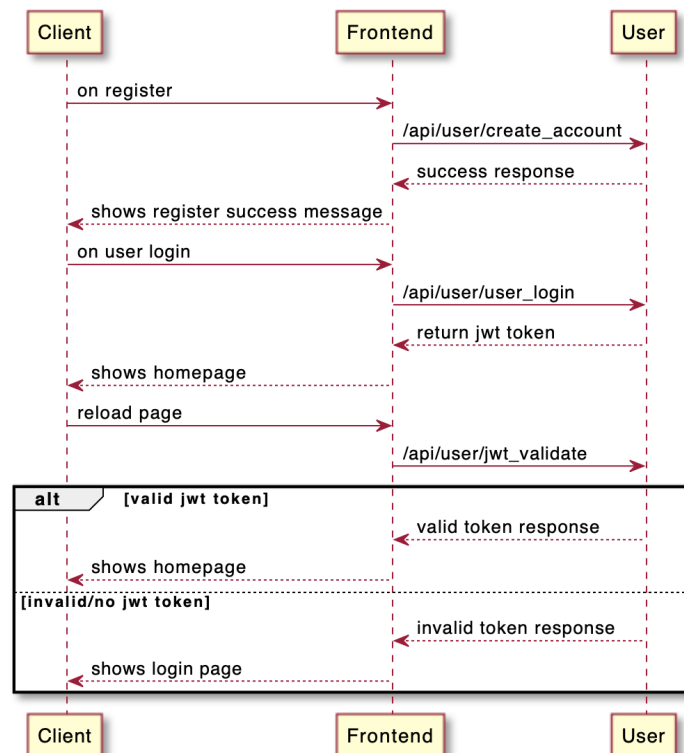


*Sequence diagram for User microservice events*

The sequence diagram above shows a successful user account registration event and subsequent user login that generates a JWT that is returned in the response message. The JWT will be passed to the User-microservice for authentication via the authorization header upon page reload until it expires. If no valid JWT is available, the user will be brought to the login page instead of the homepage.

Similarly, when the front-end makes requests to the backend microservices, the JWT is passed in the authorization header, and is validated by the `/api/user/jwt_validate` endpoint before access is granted to the resources in the backend.

The user-microservice also supports users with 2 different levels of permission(admin and normal users). Admin users can be verified through the `/api/user/validate_admin` endpoint before access is granted to the resources in the backend. `validate_admin` uses the permissionLevel header stored in the JWT to verify the permission level of the user. Level 1 permission level signifies normal user, level 2 permission level is reserved for admin users. Validated admin users will be able to add new algorithm questions into the questions database.

## Design Consideration: JSON Web Token vs Sessions

| Choice of Authentication | |
|---|---|
| **Choice 1: JSON Web Token** | <ul><li>Allows us to store details of the user</li><li>Can be stored locally on client's browser, saving disk usage and thus money</li></ul> |
| **Choice 2: Session-based authentication** | <ul><li>Requires a session identifier to be stored in the database to retrieve the necessary information</li><li>Increases disk usage and cost as we need to run a database to store session information</li></ul> |
| **Justification** | JWT has the benefits of allowing us to authenticate users but without the drawbacks that would be incurred when using a session-based authentication.<br><br>As scalability is a consideration in our requirements, using JWT also allows us to scale our backend database better, as we do not have to store additional information. |
| **Final Choice of authentication** | JWT |

# API Endpoints

Below are the API endpoints provided by the User microservice, with sample success responses and error messages.

## Create Account:

- `POST /api/user/create_account`

```
response:
{
    status:  "success",
    data: {
        message: "Register success"
    }
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 400 | failed | "Missing name, email and password fields" |
| 400 | failed | "Missing name field" |
| 400 | failed | "Missing email field" |
| 400 | failed | "Missing password field" |
| 404 | failed | "A user with this email already exists: <user_email>" |
| 500 | error | "Error writing to database " + err |

## User Login:

- `POST /api/user/user_login`

```
response:
{
    status: "success",
    data: {
        email: <user_email>,
        message: "Valid user login",
        token: <jwt_token>
    }
}
```

| Status Code | status | data.message |
|---|---|---|
| 400 | failed | "Missing email and password field" |
| 400 | failed | "Missing email field" |
| 400 | failed | "Missing password field" |
| 400 | failed | "Invalid email" |
| 400 | failed | "Invalid password" |

**Validate JWT:**

- `POST /api/user/jwt_validate`

```
response:
{
    status: "success",
    data: {
        email: <user_email>,
        name: <user_name>
    }
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 401 | failed | "Missing or invalid JWT!" |
| 500 | error | "JWT check failed: " + err |

**Validate admin account:**

- `POST /api/user/validate_admin`

| Status Code | status | data.message |
|---|---|---|
| 200 | success | - |
| 401 | failed | "Missing or invalid JWT!" |
| 403 | failed | "Invalid admin" |
| 500 | failed | "JWT check failed: " + err |

# Validate admin account:

- `POST /api/user/validate_admin`

| Status Code | status | data.message / error_message |
|---|---|---|
| 200 | success | - |
| 401 | failed | "Missing or invalid JWT!" |
| 403 | failed | "Invalid admin" |
| 500 | error | "JWT check failed: " + err |

# Questions Microservice

The Questions microservice is set up to allow other microservices to fetch and CRUD the various types of questions.

## Tech stack

| Component | Library/framework used |
|---|---|
| Backend | Node.js with Express framework |
| Database | MongoDB |
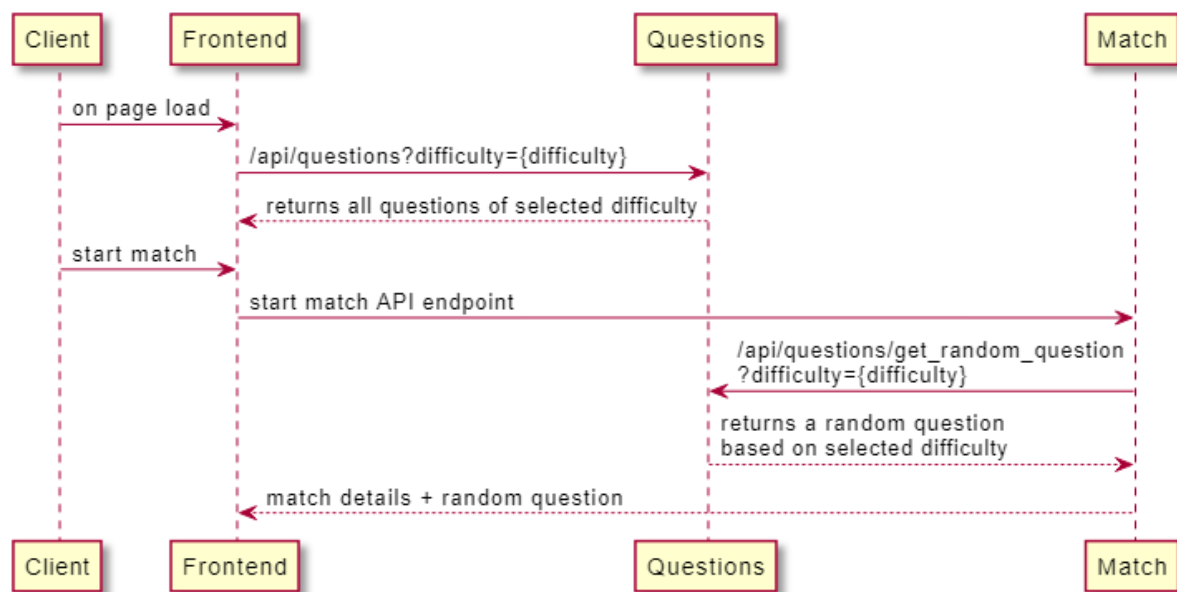


*Sequence diagram of Questions microservice's usage scenarios*

## Design Consideration: Relational vs NoSQL Databases

| Choice of Database | |
|---|---|
| **Choice 1: Relational databases** | <ul><li>Developers are familiar with traditional relational databases (e.g. PostgreSQL)</li><li>Easily scalable through official Docker images e.g. MariaDB Docker image</li><li>Each resource can be represented using rows</li></ul> |
| **Choice 2: NoSQL** | <ul><li>Developers are familiar with NoSQL databases (e.g. MongoDB)</li></ul> |

| databases | <ul><li>Also scalable through official Docker images e.g. MongoDB</li><li>No need to manually scale, can upgrade plan to increase number of servers</li><li>Each resource can be represented using documents</li></ul> |
|---|---|
| Justification | **NoSQL databases store data in the form of documents**, and this can be **easier to work with given the choice of our backend framework** (Node.js), since the **documents are similar to Javascript objects**.<br><br>NoSQL databases like MongoDB are usually horizontally scalable, hence more suited to our use cases. For example, it is cheaper to increase the number of servers used, compared to using more powerful servers through vertical scaling. Given that there is a limit to the resources allocated for our project, we have chosen to go for horizontally scaling of our database, and hence NoSQL. This is a learning point for project management as well. |
| Final Choice of Database | MongoDB, with Mongoose Node.js Library. |

## API Endpoints

Below are the API endpoints provided by the Questions microservice, with sample success responses and error messages.

**Get all questions:**

- `GET /api/questions?difficulty={difficulty}&offset={offset}&limit={limit}`

```
response:
{
    status:  "success",
    data: {
        questions: [
            {
                _id: "61824d0f0164ca949a221a7f",
                title: "what is this number?",
                description: "1",
                difficulty: "easy",
                __v: 0
            }
        ]
    }
}
```

| Status Code | status | error_message |
|---|---|---|
| 500 | error | "Error reading data from MongoDB: " + err |

## Get 1 random question:

- `GET /api/questions/get_random_question?difficulty={difficulty}`

```
response:
{
    status:  "success",
    data: {
        questions: [
            {
                _id: "61824d0f0164ca949a221a7f",
                title: "what is this number?",
                description: "1",
                difficulty: "easy",
                __v: 0
            }
        ]
    }
}
```

| Status Code | status | error_message |
|---|---|---|
| 500 | error | "Error reading data from MongoDB: " + err |

## Create new question:

- `POST /api/questions/`

```
request:
{
    difficulty:"hard",
    description: "what is 5 + 5?",
    title: "summation"
}
```

```
response:
{
    "status": "success",
    "data": null
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 400 | failed | "Missing request body" |
| 400 | failed | "Missing some input" |
| 400 | failed | "Invalid difficulty, not of 'easy', 'medium' or 'hard'" |
| 500 | error | "Error reading data from MongoDB: " + err |
| 500 | error | "Error writing data from MongoDB: " + err |

## Update existing question:

- `PUT /api/questions/:id`

```
request:
{
    "difficulty":"easy",
    "description" : "what is 5 + 5?",
    "title": "summation of 2 numbers"
}
```

```
response:
{
    "status": "success",
    "data": null
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 400 | failed | "Missing request body" |
| 400 | failed | "Missing some input" |
| 400 | failed | "Invalid difficulty, not of 'easy', 'medium' or 'hard'" |
| 400 | failed | "Invalid question id, cannot parse" |
| 404 | failed | "Invalid question id, question does not exist" |
| 500 | error | "Error reading data from MongoDB: " + err |
| 500 | error | "Error writing data from MongoDB: " + err |

# Delete existing question:

- `DELETE /api/questions/:id`

```
response:
{
    "status": "success",
    "data": null
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 400 | failed | "Invalid question id, cannot parse" |
| 404 | failed | "Invalid question id, question does not exist" |
| 500 | error | "Error deleting data from MongoDB: " + err |
| 500 | error | "Error writing data from MongoDB: " + err |

# Match Microservice

The Match microservice is responsible for storing ongoing interviews and matching two users based on the difficulty selected.

## Tech stack

| Component | Library/framework used |
|-----------|------------------------|
| Backend | Node.js with Express framework |
| Database | MongoDB |

## Matching process

Sequence diagram showing how the Match component handles a `start_find` call:
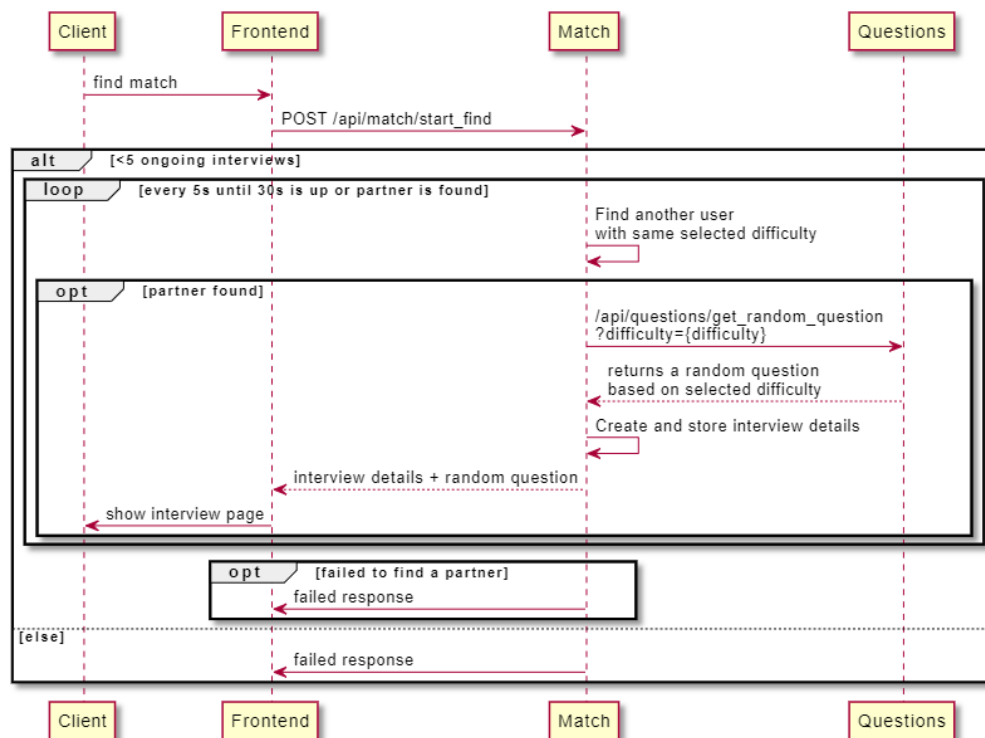


*Sequence diagram showing execution of `POST /start_find`*

When a client tries to find a match, the system will try to search for another client queuing with the same selected difficulty every 5 seconds. If a suitable partner is found, a random question with the selected difficulty will be fetched from the Questions microservice. A random `interview_id` will also be generated that is used to uniquely identify each interview session.

The interview details related to the two clients will then be sent back to the frontend and also stored to a database.

Since we have set the current limit of concurrent interviews to be 5 interviews, a client can only find a match when there are less than 5 concurrent interviews at the moment. Therefore, we have set each interview to have a maximum duration of one hour so that clients cannot stay indefinitely in an interview, allowing other clients to experience the interview feature of our web application as well.

## API Endpoints

Below are the API endpoints provided by the match microservice, with sample success responses and error messages.

### Find match:

- `POST /api/match/start_find`

```
request:
{
    email: "john@gmail.com",
    difficulty: "easy"
}
```

```
response:
{
    status: "success",
    data: {
        partnerEmail: "mary@gmail.com",
        interviewId: "615d93afdb4e069cbdfe114c",
        question: {
            _id: "61824d0f0164ca949a221a7f",
            title: "What is this number?",
            description: 1",
            difficulty: "easy",
            __v: 0
        },
        durationLeft: 3600
    }
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 400 | failed | "Missing some required fields in request body" |
| 400 | failed | "Difficulty should only be one of easy, medium or hard" |
| 404 | failed | "The total number of interviews supported by the application has been reached, please try again later" |
| 404 | failed | "Failed to find a match after 30s, please try again later." |
| 404 | failed | "You have cancelled finding a match in another browser." |
| 404 | failed | "Failed to retrieve question for interview" |
| 500 | error | "Error reading data from MongoDB: " + err |
| 500 | error | "Error writing data to MongoDB: " + err |

## Stop find match:

- `DELETE /api/match/stop_find`

```
request:
{
    email: "john@gmail.com"
}
```

```
response:
{
    status: "success",
    data: null
}
```

| Status Code | status | data.message |
|---|---|---|
| 400 | failed | "Missing some required fields in request body" |

## Get interview details:

- `GET /api/match/get_interview?email={email}`

```
response:
{
    status: "success",
    data: {
        partnerEmail: "mary@gmail.com",
        interviewId: "615d93afdb4e069cbdfe114c",
        question: {
            _id: "61824d0f0164ca949a221a7f",
            title: "What is this number?",
            description: 1",
            difficulty: "easy",
            __v: 0
        },
        durationLeft: 3000
    }
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 404 | failed | "Failed to retrieve interview details" |
| 500 | error | "Error reading data from MongoDB: " + err |

## End interview:

- `DELETE /api/match/end_interview?email={email}`

```
response:
{
    status: "success",
    data: {
        message: "Interview ended for user"
    }
}
```

| Status Code | status | data.message / error_message |
|---|---|---|
| 404 | failed | "Failed to end interview for user" |
| 500 | error | "Error deleting data from MongoDB: " + err |

## Get number of interviews:

- `GET /api/match/interviews`

```
response:
{
    status: "success",
    data: {
        count : 1
    }
}
```

| Status Code | status | error_message |
|:-----------:|:------:|:-------------:|
| 500 | error | "Error deleting data from MongoDB: " + err |

# Chat Microservice

The Chat microservice enables real-time communication between two users in an interview, allowing the participants of the interview to send messages to each other.

## Tech stack

| Component | Library/framework used |
|-----------|------------------------|
| Backend | Node.js with Express framework |
| Database | MongoDB |
| Pub/Sub | Redis |

## Pub/Sub Messaging
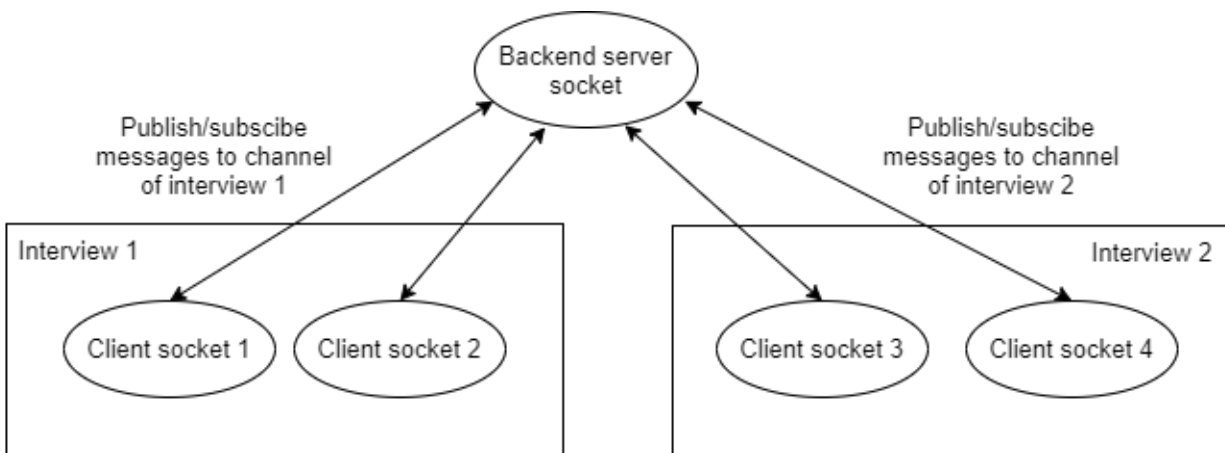


The communication adopts the publish/subscribe messaging paradigm where each client is both a publisher and a subscriber. When a user sends a message, the client socket of the user publishes the message to the channel as a publisher. As a subscriber, each client socket listens for any new messages sent to the same channel.

Using the Redis Pub/Sub system, each client will subscribe to the communication channel of the interview session they are in, using the unique `interview_id` generated by the Match microservice. Since each `interview_id` is unique, only clients of an interview session will know the `interview_id` associated with the interview session and listen in to this channel.
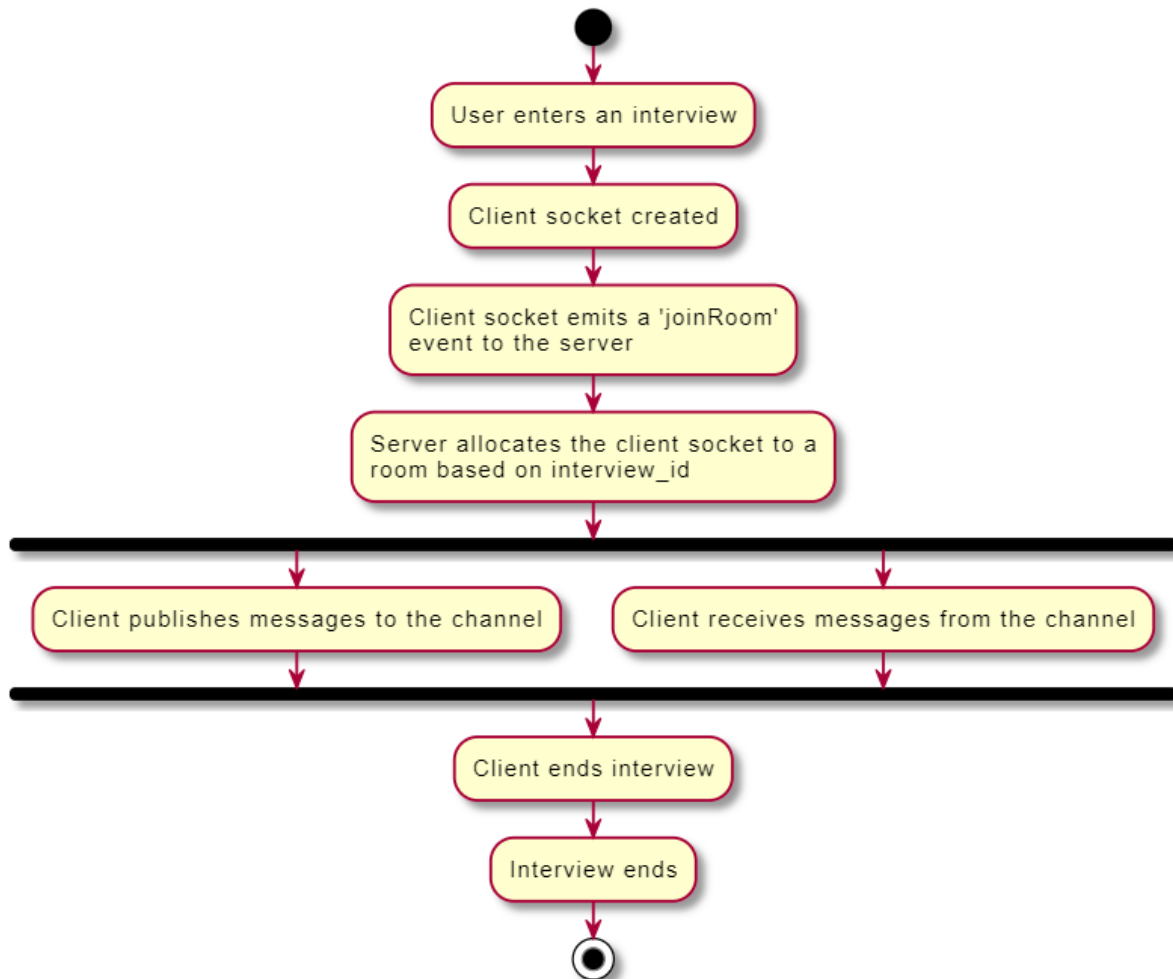
In addition, we decided to store the chat history of the interviews in a database so that a user can recover the past messages even if the user disconnects and reconnects to the interview session. The chat history is stored for an hour after the first chat message is sent from either interviewees, so that the chat history will last for at least the maximum length of the interview, which is an hour, but will also get cleared soon after the interview.

We also made the decision to store the chat history in a database and not a cache mainly due to cost concerns. Since all client messages are stored, a large amount of storage space might be required which is expensive if we were to use a cache to store them. Moreover, the chat messages are unique to an interview session and not shared across different interview sessions. This means that if the chat messages were stored in the cache, they will only be used by two clients at most, which makes the caching of chat messages not RAM-efficient.Furthermore, the chat history will only be fetched when a user reconnects to the webpage, which will not happen frequently, therefore we can afford to use slower but more affordable storage.

## API Endpoints

Below are the API endpoints provided by the Chat microservice, with sample success responses and error messages.

## Get chat history:

- `GET /api/chat/get_messages/:interviewId`

```
response:
{
    status: "success",
    data: {
        history: [
            {senderEmail: "john@gmail.com", message: "Hello!!"},
            {senderEmail: "mary@gmail.com", message: "Hi, I am Mary"}
        ]
    }
}
```

| Status Code | status | data.message / error_message |
|:---:|:---:|:---:|
| 400 | failed | "Missing some required fields in request body" |
| 404 | failed | "Chat history of interview id not found" |
| 500 | error | "Error reading data from MongoDB: " + err |

# Text Microservice

The text editor microservice enables real time communication, allowing users to collaboratively write code in a shared editor. This microservice is implemented with Socket.io, Redis Pub/Sub system and  Redis store, along with Node.js and Express backend.

The [Redis Pub/Sub system](#) enables directional real time communication, and uses Conflict-free replicated data types (CRDTs) that enables two key properties: (i) A user can modify text in the editor  without coordinating with other users (ii) Since two matched users receives the same set of updates from the Pub/Sub channel they are subscribed to, they are able reach the same state deterministically, through a Mathematical framework that guarantees state convergence. This use of Redis Pub/Sub system overcomes the issue of simultaneous writing in the editor.

## Tech stack

| Component | Library/framework used |
|---|---|
| Backend | Node.js with Express framework |
| Database & Pub/Sub | Redis |

Redis Publisher/Subscriber system in the text-microservice

The pubClient is created in the backend, and manages communication channels which are uniquely identified by the `interview_ids` generated by the Match microservice. Since each `interview_id` is unique, only clients of an interview session will know the `interview_id` associated with the interview session and subscribe to this channel.

Keystroke changes in the frontend's editor are emitted to the backend server via a *newMessage* event where the pubClient will then publish the new text change to the channel associated with the `interview_id` attached with the message. Clients subscribed to the channel will then receive the new text change.

The activity diagram below illustrates the interactions of the pubClient and subClient through a *subscription* event, *publish message* event and *receive message* event associated with `interview_id`.
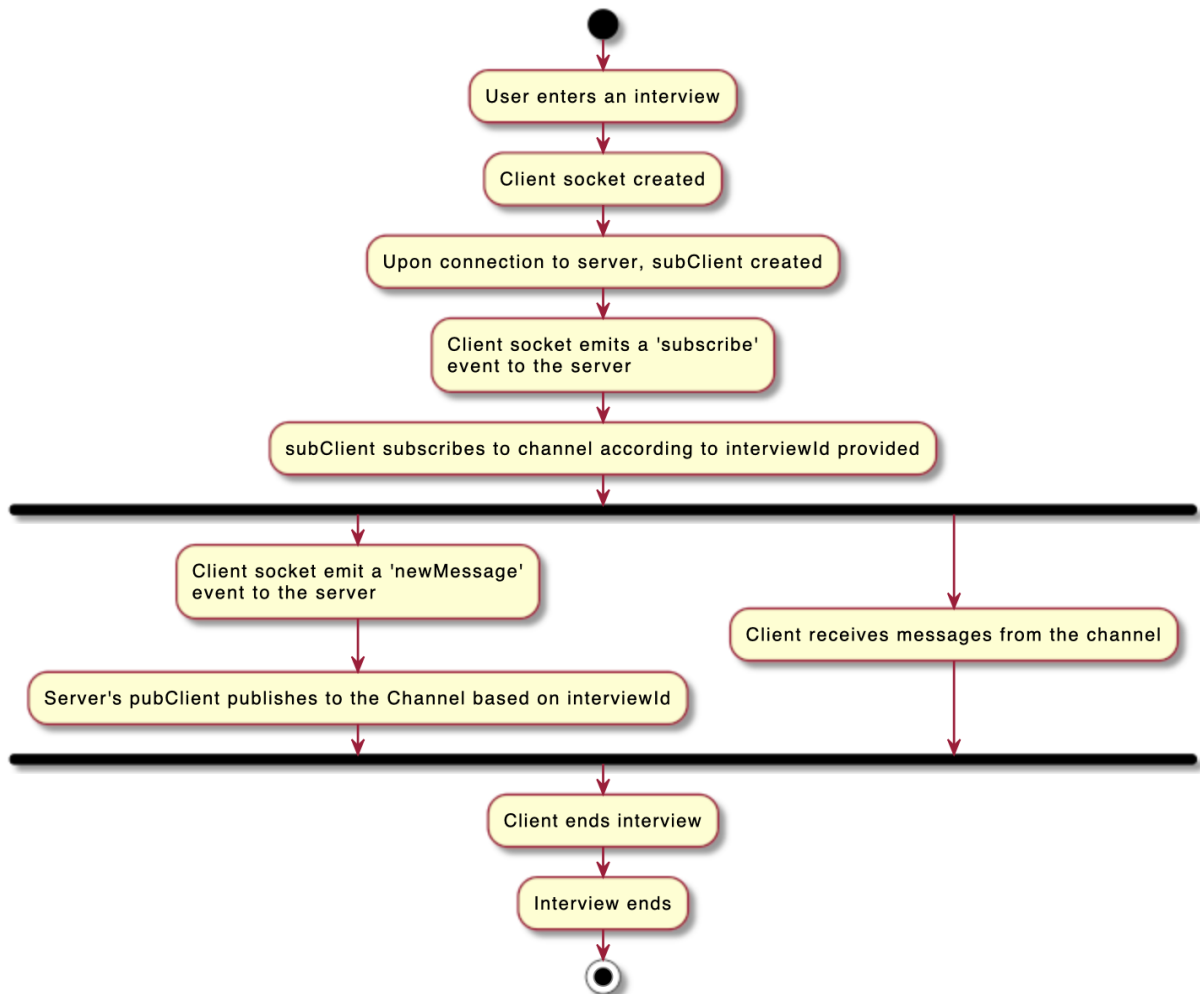
The flowchart below depicts the Pub/Sub system structure where the pubClient publishes messages to channels associated with `interview_id` unidirectionally, and only subClients subscribed to the channel will receive the corresponding messages.
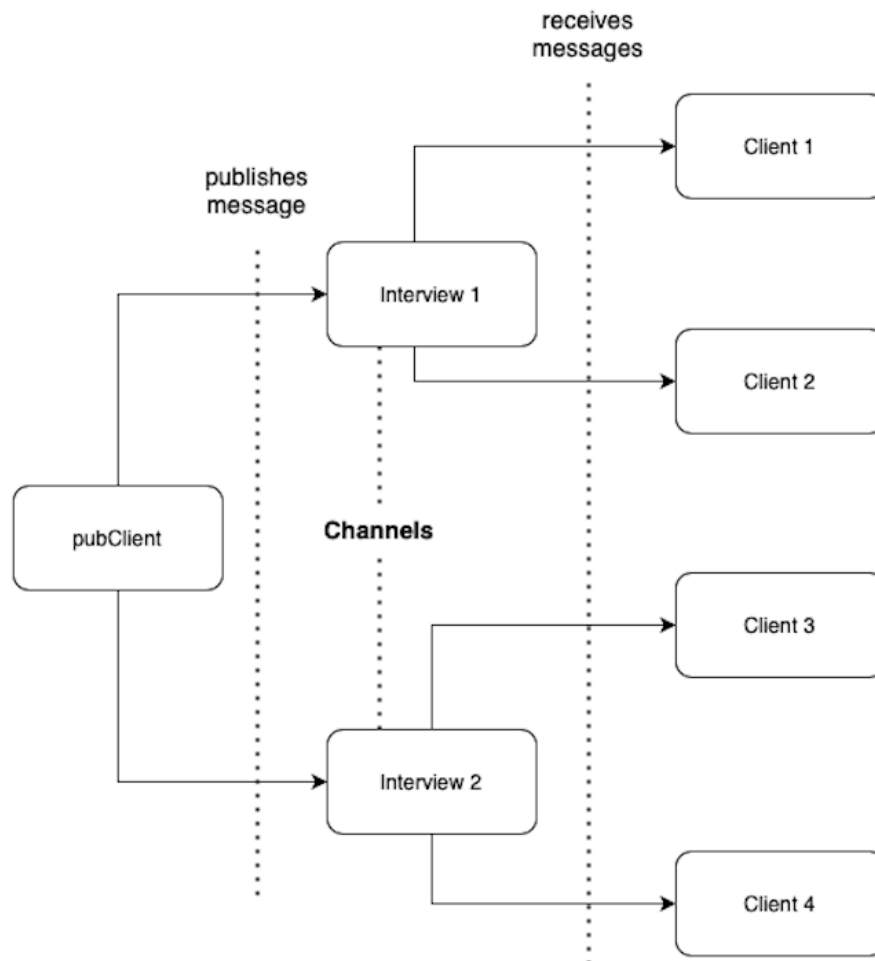


The pubClient in the backend also acts as a Redis store that stores text value associated with `interview_id` as key. This allows the frontend to restore text in the user session on a page reload event, through the `get_text` method in the API below. An end-session API is also provided to delete text associated with an interview id if necessary. Upon receiving a *disconnect* event in the backend, the subClient will also be deleted.

## API endpoints

Below are the API endpoints provided by the Text microservice, with sample success responses
and error messages.

### Get text history:

- `GET /api/editor/get_text?interviewId=<value>`

```
response:
{
    status: "success",
    data: {
        message: <text>
    }

}
```

| Status Code | status | data.message |
|---|---|---|
| 404 | failed | "No prior text history found" |

### End interview session

- `DELETE /api/editor/end-session?interviewId=<value>`

```
    status: "success",
    data: {
        message: "Session successfully deleted:" <value>
    }
```

| Status Code | status | data.message |
|---|---|---|
| 404 | failed | "No such session exists" |

# Design Patterns Used

## Builder Pattern

Our microservices make use of the Node.js and MongoDB library Mongoose, which is an Object Data Modelling library. This library is implemented with a Builder Pattern and abstracts the query logic and interactions with the MongoDB database.

## Pub/Sub Pattern

As explained above, both our Chat and Text microservices use the Pub/Sub paradigm for bi-directional real-time communication between clients.

In our application, the clients are both a publisher and a subscriber. As a publisher, a client publishes messages to the communication channel identified by the allocated `interview_id`, As a subscriber, a client listens for any new messages that are published to the same channel.

# Testing

## Unit Testing

Unit testing was done using the Mocha and Chai frameworks. It was conducted for each individual microservice to test the accuracy of the response of each API endpoint provided.

Some endpoints require authentication and authorization, so we made use of `json-server`, which helps us to create a stub server to simulate the validation endpoints. This allows us to test our microservices in isolation without depending on the Users microservice, which handles the authentication and authorization.

```
const jsonServer = require('json-server')
const server = jsonServer.create()

server.get('/jwt_validate', (req, res) => {
    res.jsonp({})
});
server.get('/validate_admin', (req, res) => {
    res.jsonp({})
});
server.listen(3001, () => {
  console.log('JSON Server is running');
});
```

*`json-server` stub for unit testing*

```
> user-microservice@1.0.0 test
> mocha ./test/test.js --timeout 10000 --exit



  add 2 users
Connected to MongoDB
User microservice listening on port 3600
    POST /user/create_account/
      ✔ should add 1 user (86ms)
      ✔ should add another user

  login 2 users
    POST /user/user_login/
      ✔ should login 1 user
      ✔ should login another user

  logout 2 users
    POST /user/user_logout/
      ✔ should logout 1 user
      ✔ should login another user


  6 passing (607ms)
```

*Unit Testing for User microservice*

```
C:\Users\Jeremy\Desktop\CS3219\Project\cs3219-project-ay2122-2122-s1-g5\questions-microservice>npm run test

> questions-microservice@1.0.0 test
> mocha --timeout 5000 --exit



  add 2 questions
connected to mongodb
Questions microservice listening at http://localhost:3000
    POST /api/questions/
      ✓ should post 1 question (151ms)
      ✓ should post another question

  get all questions
    GET /api/questions/
      ✓ should get all questions
    GET /api/questions/
      ✓ should get all questions with specified difficulty
      ✓ should get all questions with specified offset
      ✓ should get all questions with specified limit

  update question
    PUT /api/questions/:id
      ✓ should update question 5Sum difficulty from easy to hard (75ms)
      ✓ should not return any easy questions
      ✓ should return 1 hard question

  get random question for matchmaking
    ✓ should return 1 random question

  delete all questions
    ✓ should delete 1 question (51ms)
    ✓ should delete another question (43ms)
    ✓ should return 0 questions left


  13 passing (965ms)
```

*Unit Testing for Questions microservice*

```
> match-microservice@1.0.0 test C:\Users\junwe\Desktop\CS3219\Project\cs3219-project-ay2122-2122-s1-g5\match-microservice
> mocha ./test/test.js --timeout 40000 --exit


  GET /match/status
    API status
      ✓ should get working status of Match microservice API

  GET /match/get_interview
    Get interview of user
      ✓ should get interview of user

  POST /match/start_find
    Unable to find match for a user due to different difficulties
      ✓ should not find match for 2 users queueing with different difficulties (30112ms)

  DELETE /api/match/stop_find
    Stop finding match for a user who is currently finding a match
      ✓ Stop finding match for user

  DELETE /match/end_interview
    End interview
      Successfully delete interview details
        ✓ should delete interview details for user
      Unable to delete interview details due to user not having any interviews at the moment
        ✓ should not delete any interview details

  GET /match/interviews
    Get total number of interviews
      Get 1 total interview
        ✓ should get total number of interviews equals to 1
      Get 3 total interviews
        ✓ should get total number of interviews equals to 3


  8 passing (31s)
```

*Unit Testing for Match microservice*

```
PS C:\Users\junwe\Desktop\CS3219\Project\cs3219-project-ay2122-2122-s1-g5\chat-microservice> npm test

> chat-microservice@1.0.0 test C:\Users\junwe\Desktop\CS3219\Project\cs3219-project-ay2122-2122-s1-g5\chat-microservice
> mocha ./test/test.js --timeout 10000 --exit


  GET /api/chat/status
    API status
      ✓ should get working status of Chat microservice API

  GET /chat/get_messages/:interviewId
    Get chat history for interview id
Received get_messages request
      ✓ should return chat history
    Unable to get chat history due to invalid interviewId
Received get_messages request
      ✓ should not get chat history for invalid interview id


  3 passing (320ms)
```

*Unit Testing for Chat microservice*

```
> text-microservice@1.0.0 test
> mocha ./test/test.js --timeout 10000 --exit

Text microservice is listening on port 3005


  add 2 interviews
    POST /editor/save-text/
pubClient connected to Redis
Reply: OK
      ✔ should save text once (302ms)
Reply: OK
      ✔ should save text again

  get text from interviewId
    GET /editor/get_text/
session1Text
      ✔ should get session 1 text
session2Text
      ✔ should get session 2 text

  delete text stored with interviewId
    DELETE /editor/end-session/
Deleted  1
      ✔ should delete session 1 stored text
Deleted  2
      ✔ should delete session 2 stored text


  6 passing (374ms)
```

*Unit Testing for Text microservice*

# Load Testing

To test scalability, we used Locust as an API load testing tool. We specified 60 users, and 60 to be spawned in 1 second to simulate the event where at least 50 users log in at the same time (as specified in our requirements). Below is the graph showing how our Questions microservice scaled up when there are more users, showing our ability to handle increased loads of users. We successfully managed to handle more than 50 requests per second (RPS).
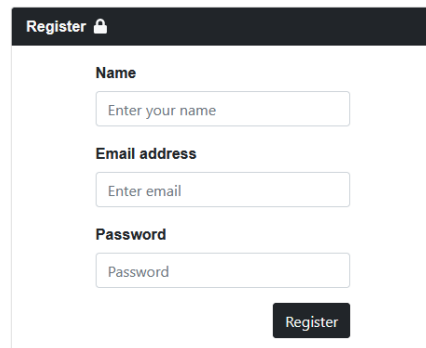


*Load testing for Questions microservice, able to handle > 50 RPS*

Below is the Google Kubernetes Engine event log to show successful scaling when there is increased load due to the Locust API testing.

# Application Screenshots



**PeerPrep**   Home   Login   Register   Tutorial

---

**Register** 🔒

**Name**

Enter your name

**Email address**

Enter email

**Password**

Password

Register

*Register page*

We omit the *Login* page as it is largely similar to the *Register* page.

**PeerPrep**   Home   Logout   Tutorial

## Difficulties

**Easy**

Beginner friendly

**Medium**

More advanced topics

**Hard**

Require expert understanding

## Questions

Easy   Medium   Hard

**100 Sum :)** Medium

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers* such that they add up to `target`.
You may assume that each input would have **exactly one solution**, and you may not use the same element twice. You can return the answer in any order.
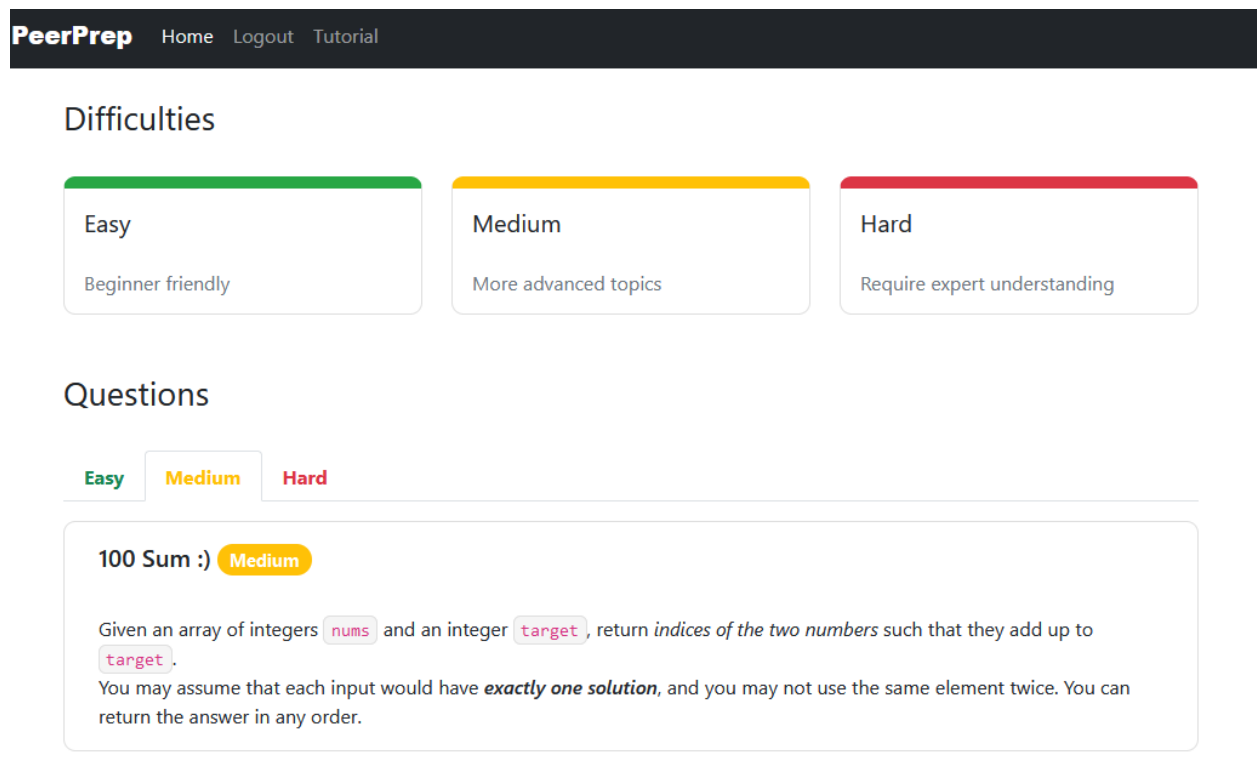
*Home page of PeerPrep which contains difficulty selection cards and question previews*

*Prompt shown when user selects a difficulty to find an interview partner*



*Practice interview page after a match is found*

# Questions   [ Add question ]

| Id | Difficulty | Title | Description |
|---|---|---|---|
| 615d0c8bf9776f95788c5895 | Medium | 100 Sum :) | Given an array of integers nums and an integer target, return *indices of the two numbers* such ... |
| 615d0cb1f9776f95788c5899 | Hard | 3Sum | Given an array of integers nums and an integer target, return *indices of the three numbers* suc... |
| 61824d0f0164ca949a221a7f | Easy | what is this number? | 1<br><br>Ans = 1 |
| 61824d1b0164ca949a221a82 | Hard | what is this number squared? | 2 |
| 61854ecc516e396bb6c441a4 | Medium | Reverse integer | Given a signed 32-bit integer x, return x *with its digits reversed<...* |
| 61854f0a516e396bb6c441aa | Hard | Merge k sorted lists | You are given an array of k linked-lists lists, each linked-list i... |
| 61855583516e396bb6c441c0 | Medium | Next permutation | Implement **next permutation**, which rearranges numbers into the lexicographic... |

*Click a row to edit a question.*

**Title**

[ 100 Sum :) ]

**Difficulty**

[ Medium                                              ▾ ]

**Description**

```
Given an array of integers `nums` and an integer `target`,
return *indices of the two numbers* such that they add up to
`target`. <br> You may assume that each input would have
***exactly one solution***, and you may not use the same
element twice. You can return the answer in any order.
```

**Preview**

Given an array of integers nums and an integer target, return *indices of the two numbers* such that they add up to target. You may assume that each input would have **exactly one solution**, and you may not use the same element twice. You can return the answer in any order.

[ Submit ]   [ Cancel ]

*Page that allows an admin user to edit questions*
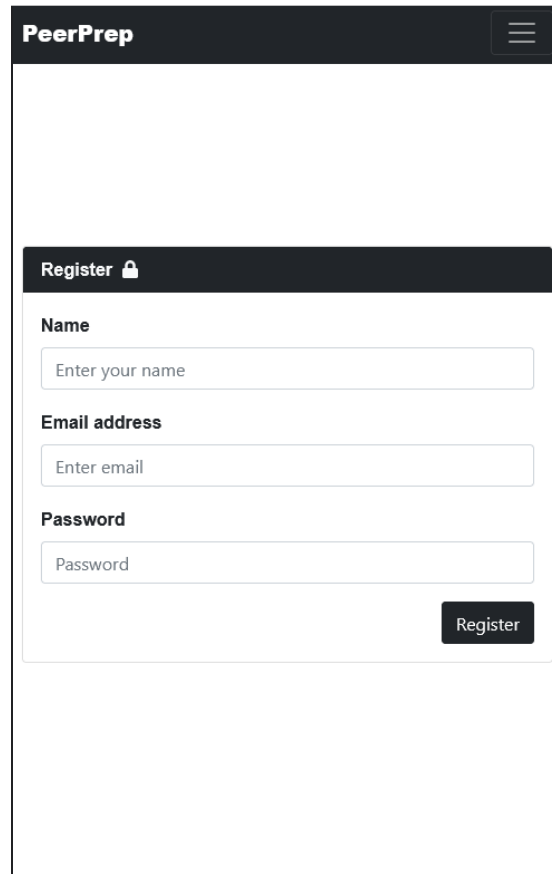
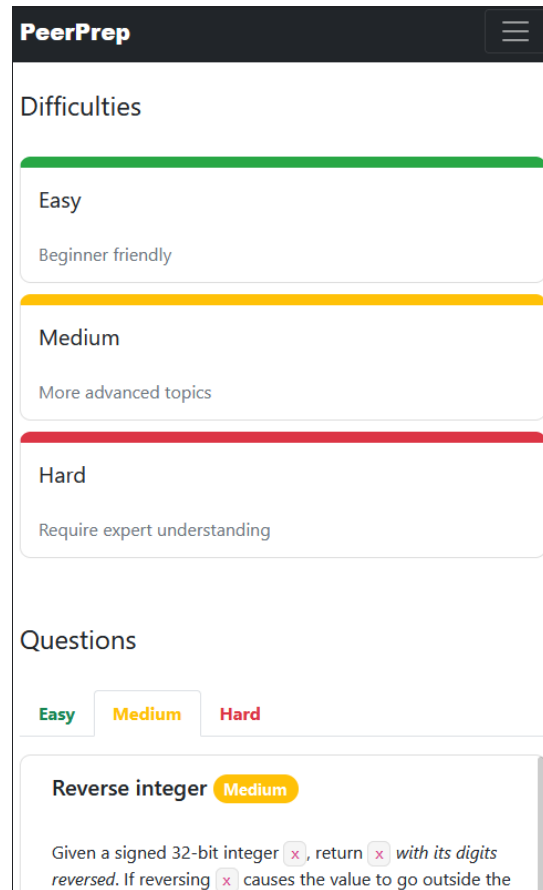*Tutorial page which guides users with steps to use PeerPrep*

# Application Screenshots (Mobile)

To allow greater flexibility to our users, our application is supported on mobile devices as well. Users can browse through questions or even enter an interview while travelling.
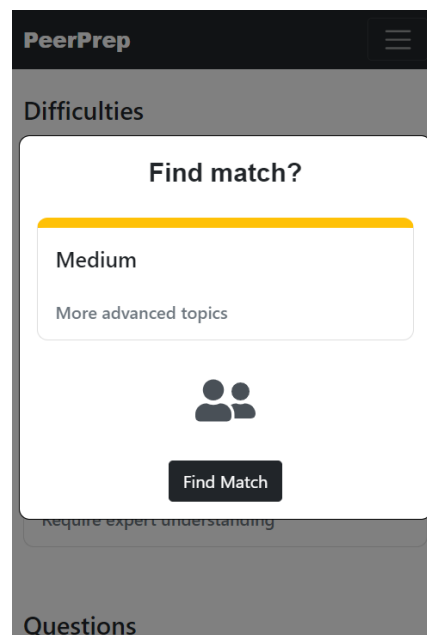


*Register page*

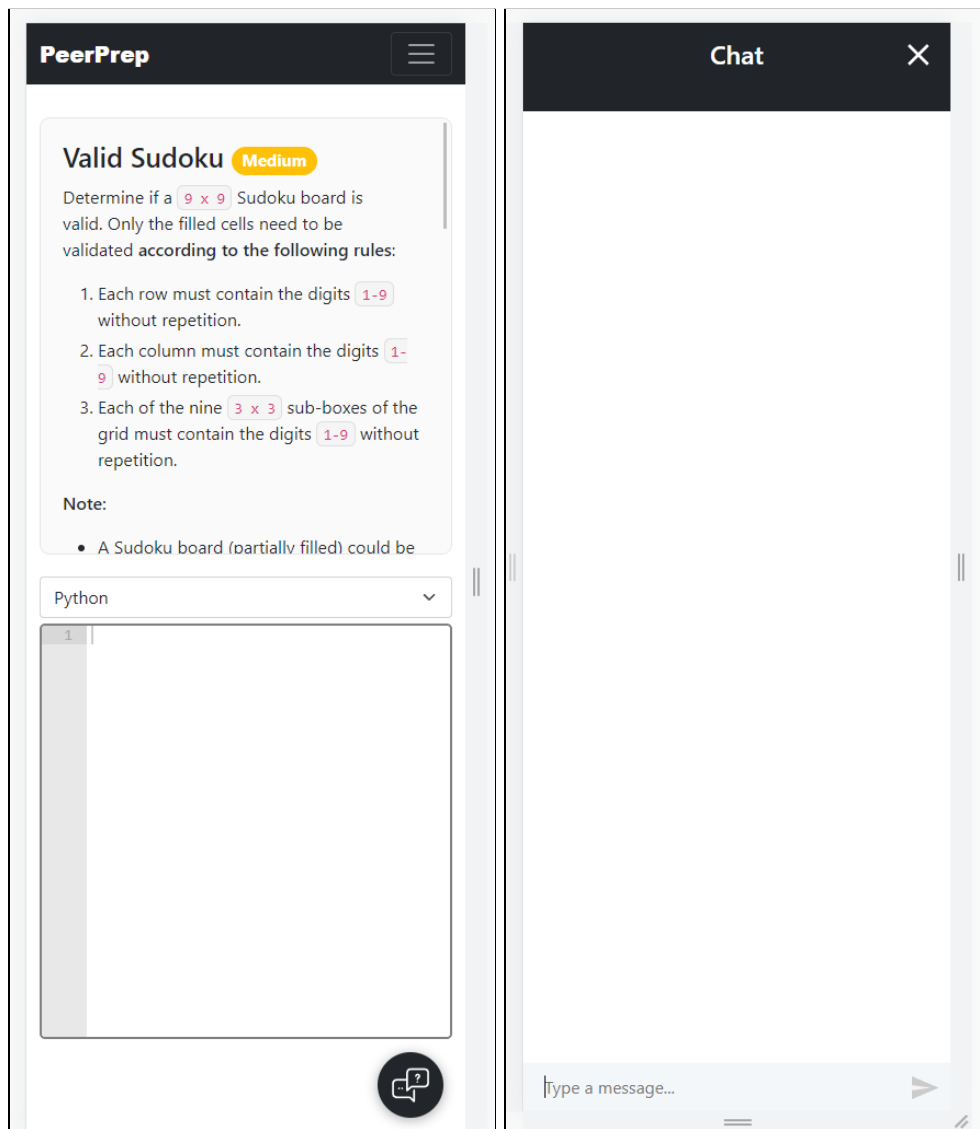We omit the *Login* page as it is largely similar to the *Register* page.

*Home page of PeerPrep which contains difficulty selection cards and question previews*



*Prompt shown when user selects a difficulty to find an interview partner*

## PeerPrep

≡

### Valid Sudoku  Medium

Determine if a  9 x 9  Sudoku board is
valid. Only the filled cells need to be
validated **according to the following rules:**

1. Each row must contain the digits  1-9
   without repetition.
2. Each column must contain the digits  1-
   9  without repetition.
3. Each of the nine  3 x 3  sub-boxes of the
   grid must contain the digits  1-9  without
   repetition.

**Note:**

- A Sudoku board (partially filled) could be

Python  ⌄

1 |

---

## Chat  ✕

Type a message...  ➤

---

*Practice interview page after a match is found, with the chat widget popup*

## PeerPrep

## Questions   Add question

| | | Prefix |
|---|---|---|
| | | |
| 615d0c8bf9776f95788c5895 | Easy | Two Su |
| 6189ecea5bb4c205de03b097 | Medium | Integer Roman |

*Click a row to edit a question.*

**Title**

Longest Common Prefix

**Difficulty**

Easy ⌄

**Description**

```
<div><p>Write a function to find the longest
common prefix string amongst an array of
strings.</p>

<p>If there is no common prefix, return an
```

---

**Title**

Longest Common Prefix

**Difficulty**

Easy ⌄

**Description**

```
<strong>Output:</strong> "fl"
</pre>

<p><strong>Example 2:</strong></p>

<pre><strong>Input:</strong> strs =
["dog","racecar","car"]
<strong>Output:</strong> ""
<strong>Explanation:</strong> There is no
common prefix among the input strings.
</pre>

<p> </p>
```

**Preview**

Example 1:

```
Input: strs = ["flower","flow","flight"]
Output: "fl"
```
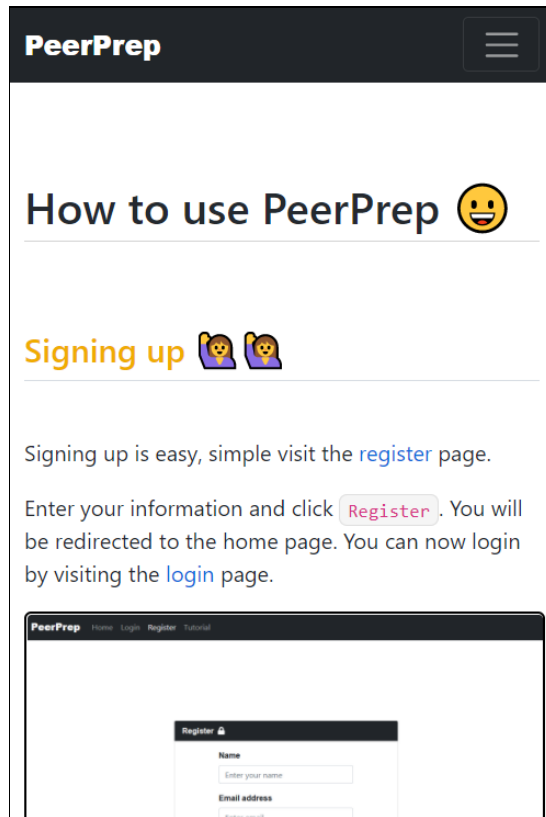
Example 2:

```
Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among
```

Submit   Cancel

*Page that allows an admin user to edit questions*

*Tutorial page which guides users with steps to use PeerPrep*

# Potential Extensions

## Discussion Forum

Currently, we do not have a platform for users to discuss algorithm questions post interview. As such, another useful feature that we can implement is a forum feature, where users can discuss problem solutions or code optimisation methods. Additionally, more experienced users can help those who are less experienced by providing interview tips, or tips on how to better prepare for coding tests on this forum. The forum also increases the interactivity of our app, and makes it more engaging for the users.

### Front end

New components need to be added, such as a *Forum* component to present the forum main page, a *ThreadCard* on the main page that can be clicked to view a particular thread. Each component would also need to do relevant API calls such as `/api/forum/threads` to get the list of all threads that are in the forum.

### Back end

We would implement this using a new Forum microservice, which handles all the logic related to forum activities like creating a new post, fetching new posts, updating posts or deleting posts. We also have to allow functionality to CRUD for comments for each post too.

# Finding match in background

Following up on the Discussion Forum extension, another new feature would be to enable users to find a match in the background while the user navigates through other pages such as the tutorial page or new forum. This reduces mundane wait time that users undergo when waiting for a match to be found, allowing users to engage in other activities before a match is found. We will also increase the match finding duration that is currently set to 30 seconds, so the user will not have to continuously refresh the match find.

## Front end

A find match status will be added to the navigation bar. It would display a 'Cancel find match' button as well as a timer that shows the time left during find match. The main network calls and matchmaking logic would then be transferred from *FindMatchModal* to the new *FindMatchStatus* component. Since the *FindMatchStatus* is a global component, it will be able to handle matchmaking logic regardless of which page the user is visiting.
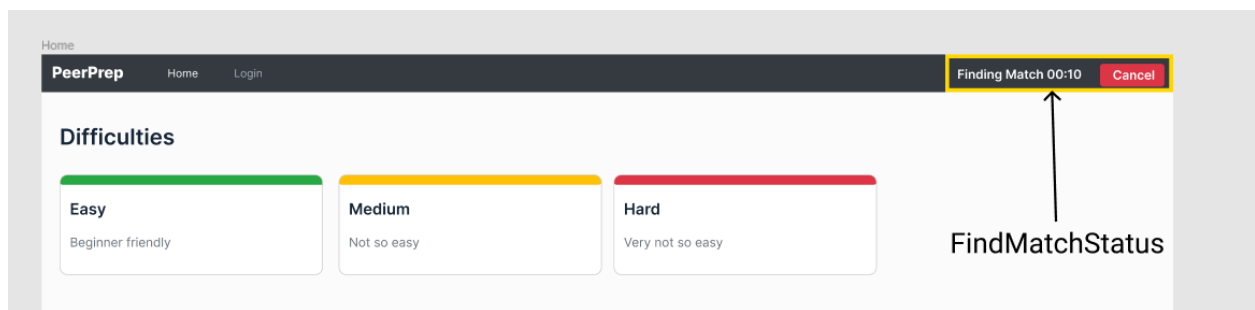


*UI prototype showing FindMatchStatus*

# Reflections, Learning Points

## Project Management

We had to manage the entire project from the start to the end by ourselves, and this required some project management skills. We had to set sprint schedules and manage our schedule well. While those are not new to us, having been exposed to them in CS2103T and CS3203, it was particularly important here as we were unfamiliar with the technologies being used here. It was our first time developing a scalable web application, and we had to ensure that ample time was set aside for deployment and testing after deployment.

And this brings us to the next learning point: integration testing. As with every application, integration testing is important. It is even more so for us in this project due to our initial unfamiliarity with the tech stack used in the other microservices. Fortunately, we set aside time each week to do our integration testing, and to further discuss if there needs to be any changes to the tech stack or the APIs used or defined by each component.

## Technical Skills

### Web Development Technologies

#### Orchestration engines - Kubernetes, GKE

Apart from the assignment in this module, we all had little to no experience with the technologies introduced in the module. There was a very steep learning curve involved when we had to set up a Kubernetes cluster on the cloud (GKE), as we had to do lots of trial and error locally and ensure that the Kubernetes configurations were correct before we could try it on GCP. Even so, due to GKE's native ingress controller, we once again had to change our configurations to suit that. Thankfully, GCP's documentation is up to date and useful, and we managed to debug our configuration issues and even tweak them to suit our application through trial and error.

#### Real-time technologies - Redis, pub-sub mechanism

We also learnt more about how real-time technologies work, and how to develop applications with real-time capabilities. For example, the text editor and chat microservices required us to implement pub-sub mechanisms using Redis as a broker, and also to use web socket technologies to facilitate the real-time transfer of information. It was difficult to debug issues related to these as we had little experience working with these tools, and did not know what to look out for.

As we were not conversant with web security, we had to read up a lot on what were the good practices to implement. For example, when deciding how we should authenticate the user, we used JSON Web Tokens (JWT), which stored the data locally on the client's local storage. This was a design consideration such that we did not need to store the corresponding information on our database; we simply had to ensure that the JWT was valid. Another security issue that we faced was having to set up HTTPS. Fortunately, GCP provided managed certificates that we could use to set up SSL termination at our ingress resources.

## Microservices Architecture

Additionally, it was difficult to envision how the overall system architecture would look like. It was one thing to learn about the microservices architecture in lecture and tutorial, but it was another to have to implement it on our own. There were many other considerations that we had to consider, like managing CPU and memory consumption of the various pods to identify the optimal resource limit to set in the Kubernetes yaml file.

# Conclusion

This module and project has been an eye-opener for all of us, in terms of developing web applications. There is so much more to developing a web application than simply writing code that just works. There are many things that we have to consider, like security issues, scalability, portability and availability, apart from just meeting the functional requirements.