CS3219 Software Engineering Principles and Patterns

Group 43 Project Report

Member	Student Number
Kwek Min Yih	A0205883Y
Wu Hanhui	A0218237E
Tan Ka Shing	A0218409A
Joven Pua	A0201693J

Github Repository:
https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g43

Deployed app: https://www.kminy.space/

Introduction

Background

Technical assessments are the new normal when applying for internships and jobs as a software engineer in this day and age. With the availability of numerous algorithmic practice platforms out there, we are given a large amount of resources with solutions to improve their problem solving skills. However, programmers are still faced with the problem of articulating their thought process during live interviews and understanding/solving programming problems in general. While collaboration and knowledge sharing has always been part of software engineering, there is currently no notable platform available for real time collaborative solving of coding problems.

Purpose

MeetCode aims to promote peer-programming for programmers by providing a platform where users can practise and solve technical interview questions together in real time. Users of MeetCode will be able to find like minded individuals to enhance their communication skills in articulating technical concepts and to discuss and share knowledge in solving and exploring complex programming problems.

Individual Contributions

The following table details the individual contributions of each member in terms of development, deployment and report writing.

Member	Contribution
Kwek Min Yih	● Implemented Frontend and Learning Pathway Service
	● Deployment and continuous deployment for Frontend ● Domain configuration
	Sections Written: ● Functional Requirements (Frontend & Learning Pathway Service) ● Quality Attributes (Usability & Robustness) ● Development Process ● Overall System (MVA Architecture & Database Design) ● Frontend ● Learning Pathway Service Contributed to: ● Overall System (Microservice Architecture)

Wu HanHui	• Implemented User Service and Collaboration Service
	• Configure deployment environment • Deployed User, Collaboration and Learning Pathway Services • Set up CI/CD for backend services
	Sections Written: • Functional Requirements (User & Collaboration Service) • Quality Attributes (Scalability) • Tech Stack • Overall System (Architecture Diagram) • User Service • Collaboration Service • Deployment
Tan Ka Shing	• Implemented Question Service
	• Deployed Question Service
	Sections Written: • Introduction (Background and Purpose) • Functional Requirements (Question Service) • Future Improvements
Joven Pua	• Implemented Matching Service
	• Deployed Matching Service
	Sections Written: • Functional Requirements (Matching Service) • Matching Service • Learning Points and Reflections Contributed to: • Overall System (Microservice Architecture)

Requirements

Functional Requirements

Frontend

Name	Frontend
Description	The frontend provides an interface for the user to interact with the application and access its functionalities
Capabilities	User Interface
Dependencies	User Service, Matching Service, Question Service, Collaboration Service, Learning Pathway Service

Functional Requirements		
S/N	**Description**	**Priority**
FR1.1	The system should allow the user to register an account with a username and password	High
FR1.2	The system should allow the user to log into their account using their username and password	High
FR1.3	The system should validate the username and password before sending it to the user service for either registration or authentication	High
FR1.4	The system should allow users to log out of their account.	High
FR1.5	The system should allow users to delete their accounts.	High
FR1.6	The system should allow users to change their password.	High
FR1.7	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
FR1.8	The system should emit a match event to the matching service when the user starts a session	High
FR1.9	The system should emit a timeout event to the matching service if the user has not been matched in 30 seconds	High
FR1.10	The system should inform the user if a match has not been found in 30 seconds, allowing them to continue waiting or leave the session	High
FR1.11	The system should redirect the user to the room page once a match has been found	High
FR1.12	The system should allow the user to leave the room page.	High

FR1.13	The system should display a warning when the user leaves the room page.	Medium
FR1.14	The system should inform the user if their partner leaves the room page.	High
FR1.15	The system should show both matched users the same random question of specified difficulty on the room page	High
FR1.16	The system should allow users to write code concurrently and collaboratively using a code editor	High
FR1.17	The system should allow users to select the language being used in the code editor	Medium
FR1.18	The system should allow users to change the theme of the code editor	Low
FR1.19	The system should send the details of the coding attempt (timestamp of when the attempt first started, user ID, username of the partner, question difficulty, question ID, question title, code written) to the learning pathway service when the user leaves the room page.	High
FR1.20	The system should display an error message if the coding attempt could not be sent to the backend, and allow the user to choose if they wish to resend it or cancel.	Medium
FR1.21	The system should display a table of the user's past coding attempts, showing the following fields for each attempt: timestamp of attempt, name of question attempted, the username of partner	High
FR1.22	The system should display the code written as well as the question text as additional details of each attempt	High
FR1.23	The system should display a breakdown of the number of questions of each difficulty attempted by the user	Low
FR1.24	The system should display a breakdown of the recent attempts by the user in the past day, week and month	Low
Non-Functional requirements		
S/N	**Description**	**Priority**
NFR1.1	The system should be intuitive and user-friendly.	High
NFR1.2	The system should have a responsive design and work on smaller screens	Low
NFR1.3	The system timeouts API calls if they take greater than 5 seconds	High

User Service

Name	User Service
Description	Maintains and provide functionalities related to user account details
Capabilities	- Create a new user account - Login user account - Logout user account - Delete user account - Update user password - Create JWT tokens for authentication - Verify JWT tokens - Create refresh token

Functional Requirements		
S/N	**Description**	**Priority**
FR2.1	The system should allow users to create an account with username and password.	High
FR2.2	The system should ensure that every account created has a unique username.	High
FR2.3	The system should validate that the password meets the minimum requirements before creating the account or changing the password.	High
FR2.4	The system should allow users to log into their accounts by entering their username and password.	High
FR2.5	The system should allow users to log out of their accounts	High
FR2.6	The system should allow users to delete their account.	High
FR2.7	The system should allow users to change their password.	High
FR2.8	The system should create an access token after the user login.	High
FR2.9	The system should verify the access token before processing the request, except for creation of account and login.	High
FR2.10	The system should remove the access tokens when the user logs out.	High
FR2.11	The system can retrieve all records belonging to a particular user	Medium
FR2.12	The system should verify the refresh token before renewing the user's access token.	Medium

Non-Functional Requirements		
S/N	**Description**	**Priority**

NFR2.1	Tokens will be implemented with JWT.	High
NFR2.2	Access token will be stored in cookies and be removed from cookies after the user logout.	High
NFR2.3	Access token should expire after 2 hours.	High
NFR2.4	Refresh token should expire after 12 hours.	High
NFR2.5	Minimal requirement for a password is at least 8 characters long and includes both upper and lower case characters.	High
NFR2.6	Only the owner of the account should be able to change the password or delete the account.	High
NFR2.7	Users' passwords should be hashed and salted before storing them in the database.	High

Matching Service

Name	Matching Service
Description	Provide matching service between two users based on the question difficulty selected
Capabilities	- Create a pending match - Create a match - Bringing matched users to a room upon matching - Cancel a pending match
Dependencies	User Service, Frontend

Functional Requirements		
S/N	**Description**	**Priority**
FR3.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
FR3.2	The system should match two waiting users with the same selected difficulty level.	High
FR3.3	The system should generate a room ID and communicate it to the pair of matched users.	High
FR3.4	If there is a valid match, the system should match the users within 30s.	High
FR3.5	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
FR3.6	The system should remove the waiting user from the wait queue upon the user’s request or upon user disconnecting.	High

FR3.7	The system should store the details of both matched users.	High
FR3.8	The system should notify the other matched user if the partner leaves the room.	Medium
FR3.9	The system should notify users of their partner's username.	Low
FR3.10	The system should notify users if the coding language has been changed by their partner	High
Non-Functional requirements		
S/N	**Description**	**Priority**
NFR3.1	Matched users should be able to see the room page within 5s after matching.	High

Collaboration Service

Name	Collaboration Service
Description	Provide real-time updates of the code edits to both the matched users
Capabilities	- Update the users of changes made by one another - Editor provides basic features such as autocomplete and syntax highlighting - Editor provides different colour themes.
Dependencies	Frontend, Matching Service

Functional Requirements		
S/N	**Description**	**Priority**
FR4.1	The system should establish communication between the two matched users.	High
FR4.2	The system should push updates such as code changes and cursor position to the other user.	High
FR4.3	The system should allow users to select language for the purpose of syntax highlighting.	Medium
FR4.4	The system should allow users to pick the colour theme.	Low
Non-Functional Requirements		
S/N	**Description**	**Priority**
NFR4.1	The editor will make Java, Python, and Javascript programming languages available for selection	Medium

NFR4.2	The editor should provide at least 3 colour themes	Low
NFR4.3	If a user disconnects, he/she should be able to get the latest state of the editor after reconnecting.	High

Question Service

Name	Question Service
Description	Provide storage and retrieval of coding questions from a question bank indexed by difficulty level
Capabilities	- Create a new coding question - Retrieve a random coding question - Can add/retrieve/delete coding questions with three levels of difficulty: easy, medium and hard
Dependencies	Frontend, Learning Pathway Service

Functional Requirements		
S/N	**Description**	**Priority**
FR5.1	The system should allow the addition of new questions of specified difficulty (easy, medium or hard).	High
FR5.2	Given the difficulty and a UUID, the system should return a seeded random question of specified difficulty, using the UUID as the seed.	High

Non-Functional Requirements		
S/N	**Description**	**Priority**
NFR5.1	Questions will be taken from Leetcode.	Medium
NFR5.2	The questions stored will consist of id, title, difficulty and text.	High
NFR5.3	Question text will be stored as HTML text - retrieved in NFR5.1, in the database for easy rendering on the frontend.	Medium
NFR5.4	Questions are considered immutable and cannot be deleted or updated	High
NFR5.5	The system should validate all incoming data	High

Learning Pathway Service

Name	Learning Pathway Service
Description	Provide a record of users' past coding attempts for various problems
Capabilities	- Create a record of users' coding attempts - Retrieve users' coding attempt records
Dependencies	Frontend, User Service, Question Service

Functional Requirements		
S/N	**Description**	**Priority**
FR6.1	The system can create a record of users' coding attempts, storing the following fields: the user ID of the user, username of partner, question ID, code written by the user and the datetime when the attempt was started.	High
FR6.2	The system can retrieve all records belonging to a particular user	High
FR6.3	The system can delete all records belonging to a particular user	Medium

Non-Functional Requirements		
S/N	**Description**	**Priority**
NFR6.1	The system should handle requests within 5 seconds.	High
NFR6.2	The user IDs stored correspond to the user IDs in the user service.	High
NFR6.3	The question ID and title corresponds to the question IDs and titles in the question service	High
NFR6.4	The coding attempts stored can only be deleted, not updated	High
NFR6.5	The system should validate all incoming data	High

Quality Attributes

Quality Attributes Prioritization Matrix

Attribute	Score	Availability	Modifiability	Performance	Robustness	Scalability	Security	Usability
Availability	3		<	<	^	^	<	^
Modifiability	0			^	^	^	^	^
Performance	1				^	^	^	^
Robustness	4					^	<	^
Scalability	5						<	^
Security	2							^
Usability	6							

Usability

Usability refers to user-friendliness, ease of use and human engineering. The NFRs related to usability include NFR1.1 and NFR1.2.

NFR1.1: Intuitive and User Friendly System

In order to ensure the frontend is intuitive and user-friendly, and provides a good user experience, we adhered to UI/UX laws and best practices.

In accordance with Jakob's Law[1], which states that users spend most of their time on other sites, and prefer your site to work the same way as all the other sites they already know, we follow familiar conventions in the design of the frontend. For instance, the login button and the user menu are placed in the top right of the screen. In the room page, the layout is similar to LeetCode, a popular coding interview question website.

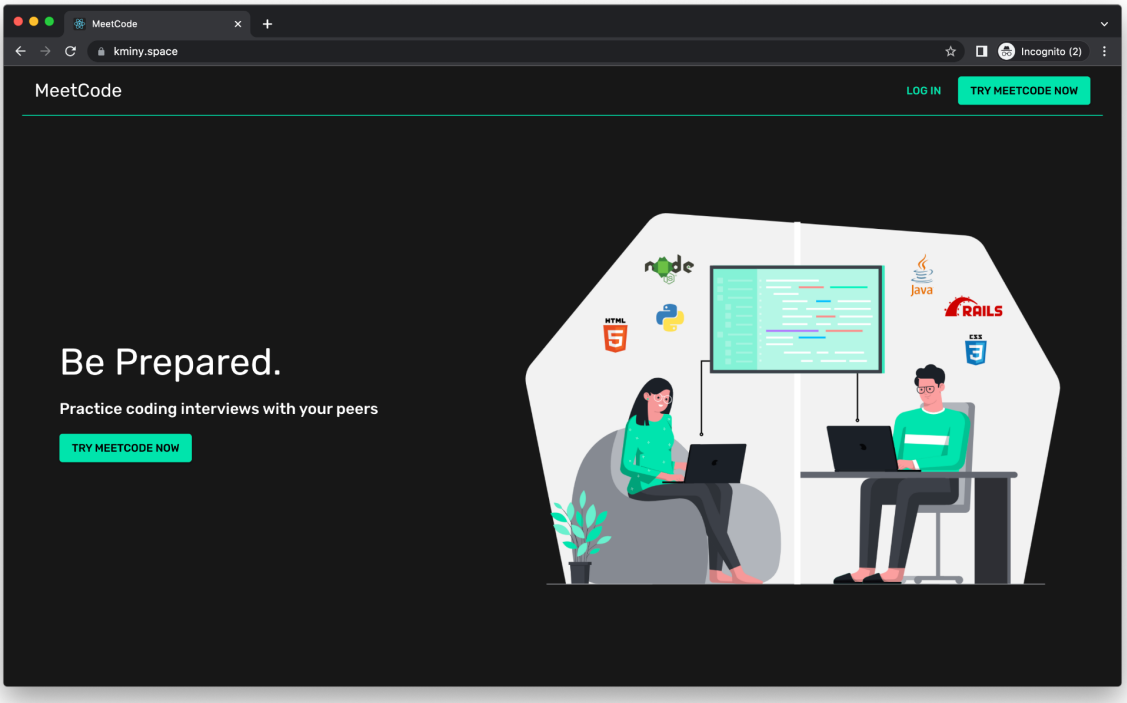

Figure 1 – Landing Page

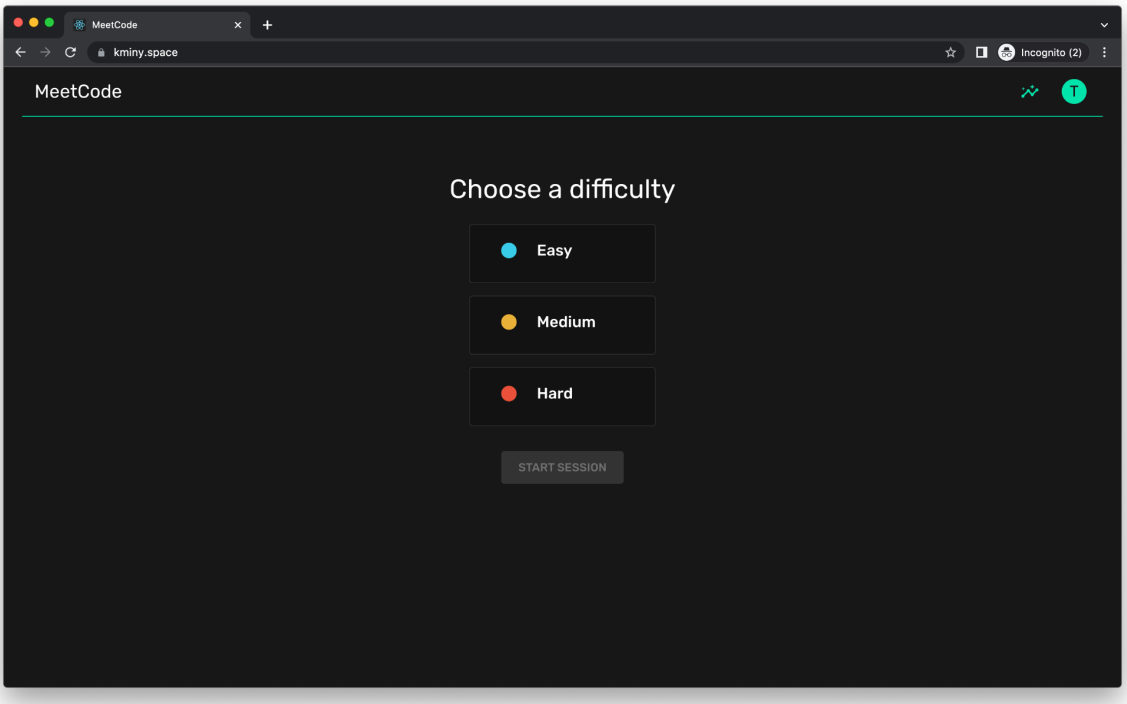

Figure 2 – Home Page

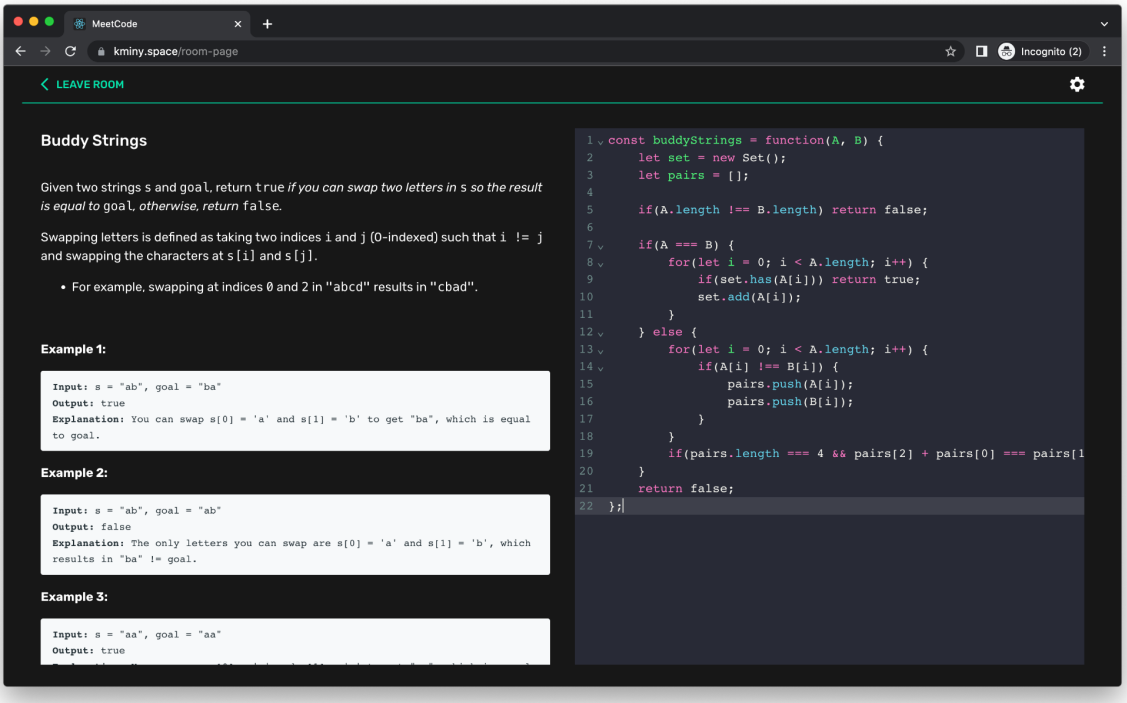

Figure 3 – Room Page

In accordance with the Doherty Threshold[2], which states that productivity soars when a computer and its users interact at a pace (<400ms) that ensures that neither has to wait on the other, we considered the user's perceived performance of the web application in its design. When fetching data, such as the question to be attempted, or the user's previous attempts, we make use of skeletons to denote that the content is being loaded to visually engage the users. When sending data such as a completed user attempt, or when waiting for a match, loaders are also used to assure the user their request is being processed.

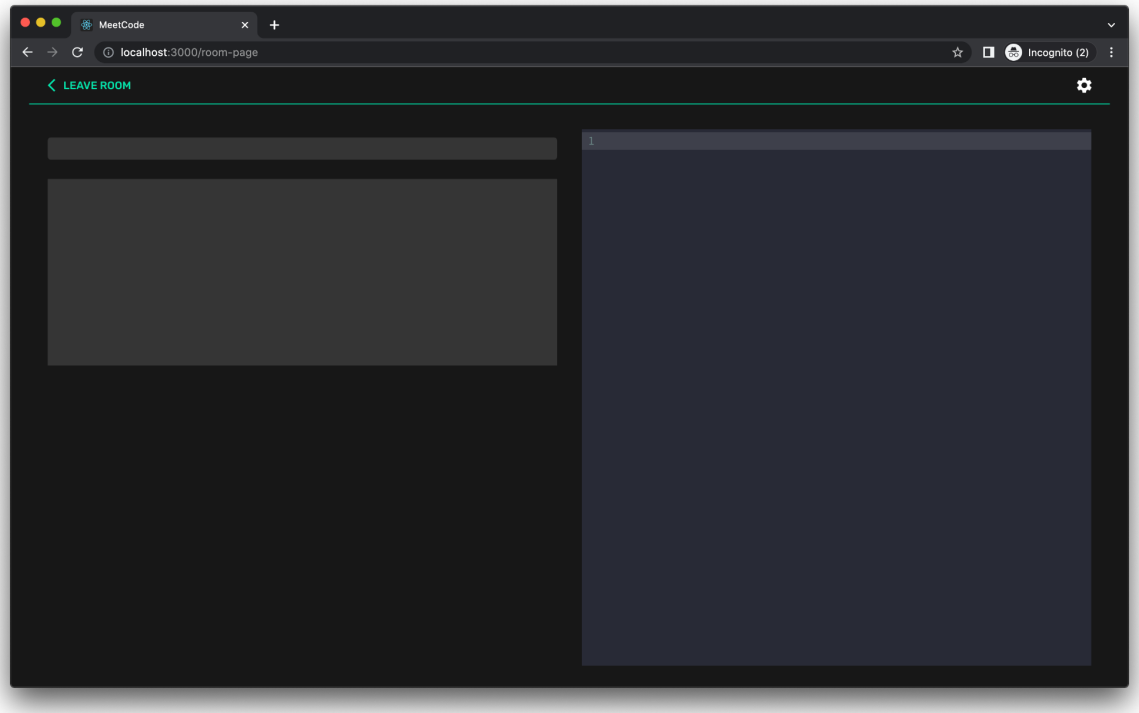

Figure 4 – Question Display Skeleton

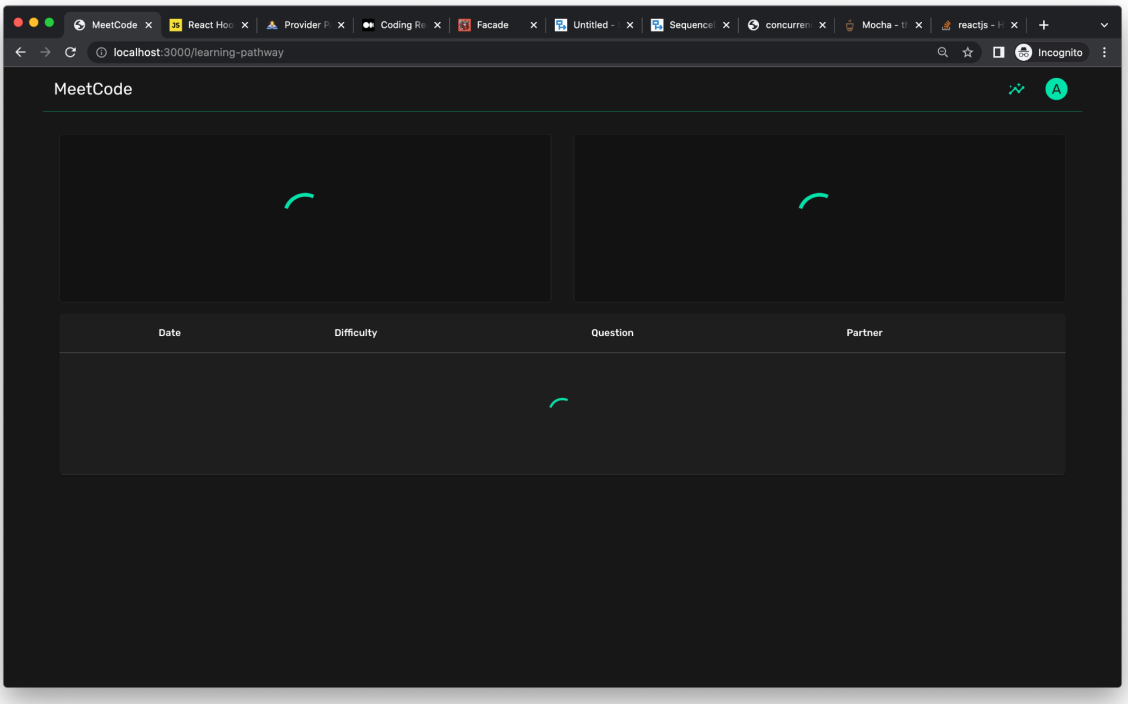

Figure 5 – Loader for Learning Pathway Page

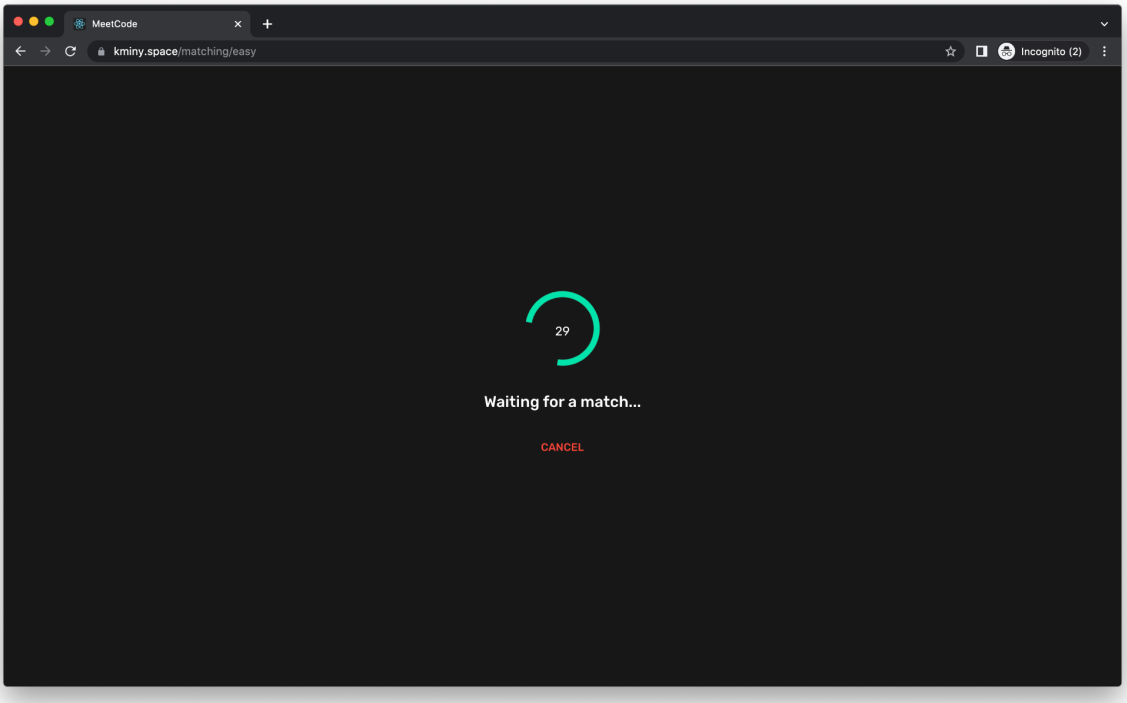

Figure 6 – Loader for Matching Page

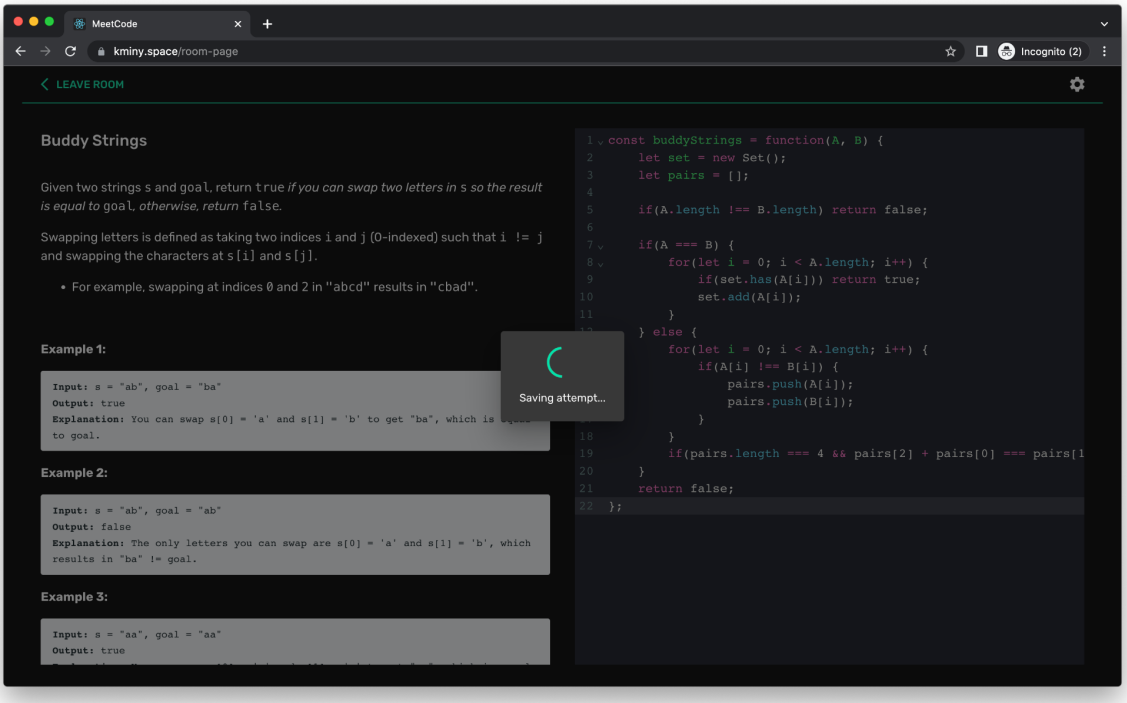

Figure 7 – Loader for sending user attempt

We also took effort to make our frontend aesthetically pleasing, simple and uncluttered in accordance with the Aesthetic-Usability Effect[3], which states that users often perceive aesthetically pleasing design as design that's more usable. We did so by adhering to a consistent theme with a primary accent colour, green.

Other measures taken to improve the usability of our web application and prioritise the user experience include:

1. Warning users when doing actions that are not easily reversible
    a. Users are warned when leaving the room page that their action is irreversible

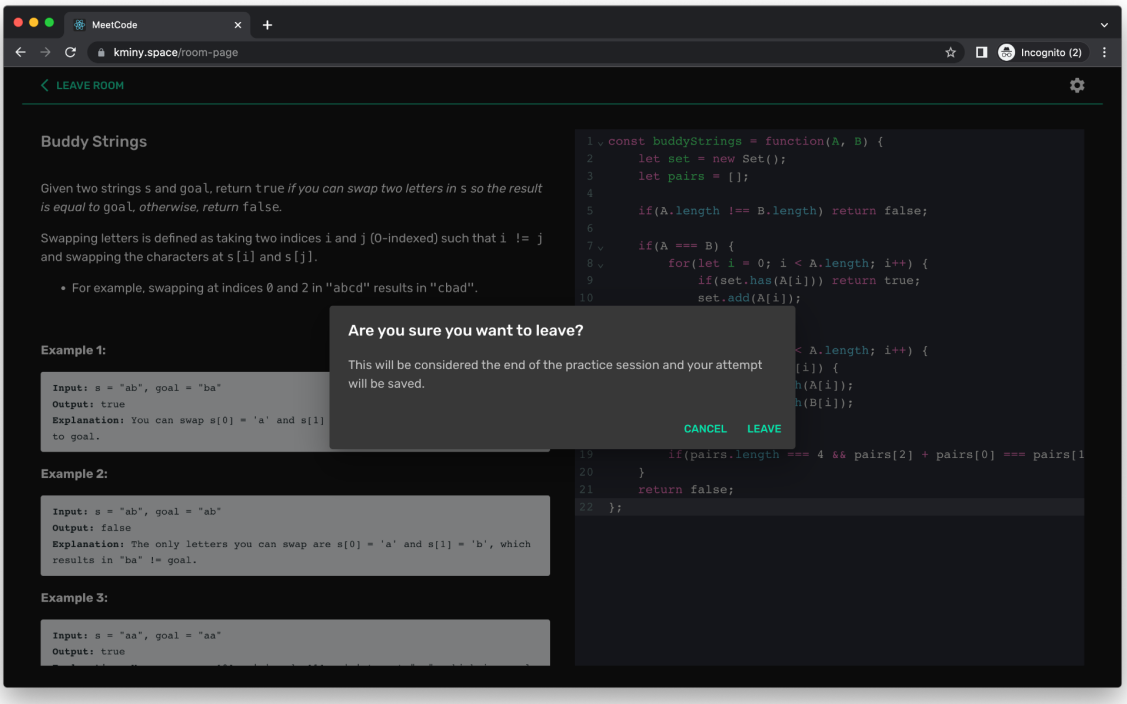

Figure 8 – Warning message for leaving room

b. When the user deletes their account, they need to type in their username to ensure they truly wish to delete their account

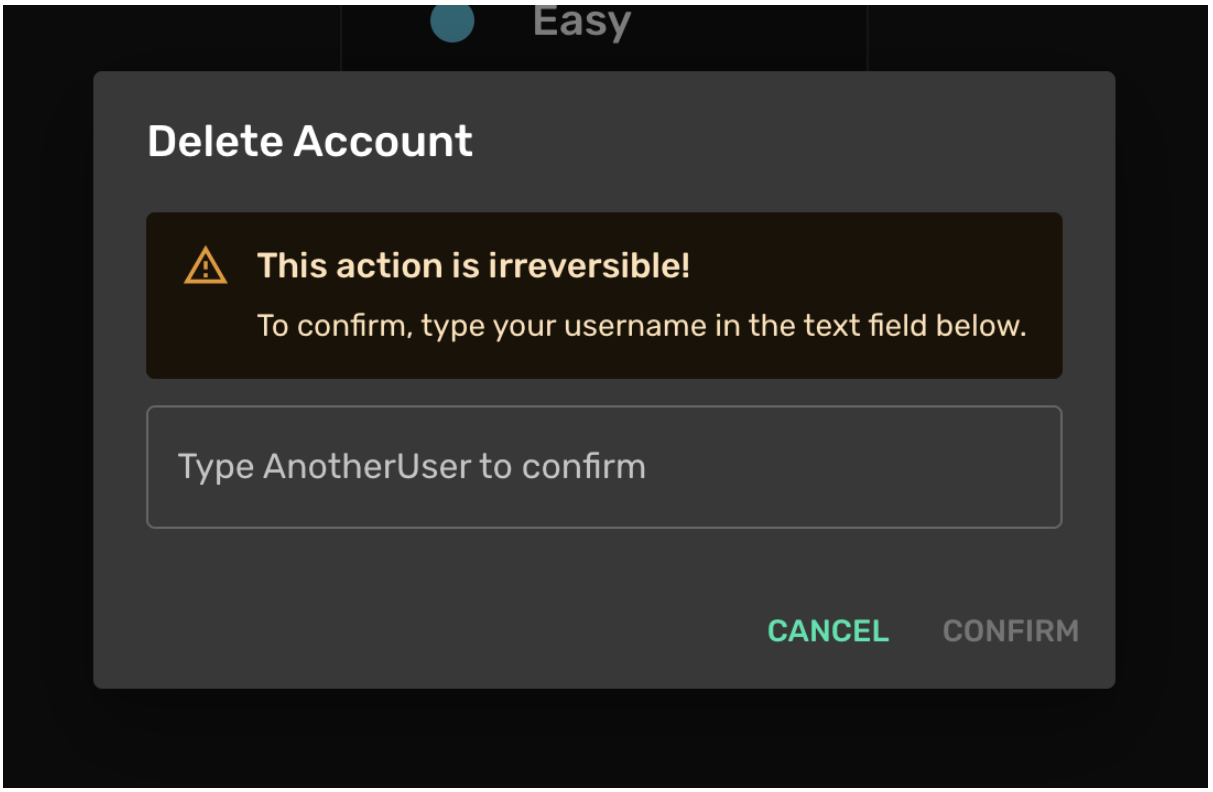

Figure 9 – Warning message when deleting account

2. Displaying clear success, error and notification messages to the user

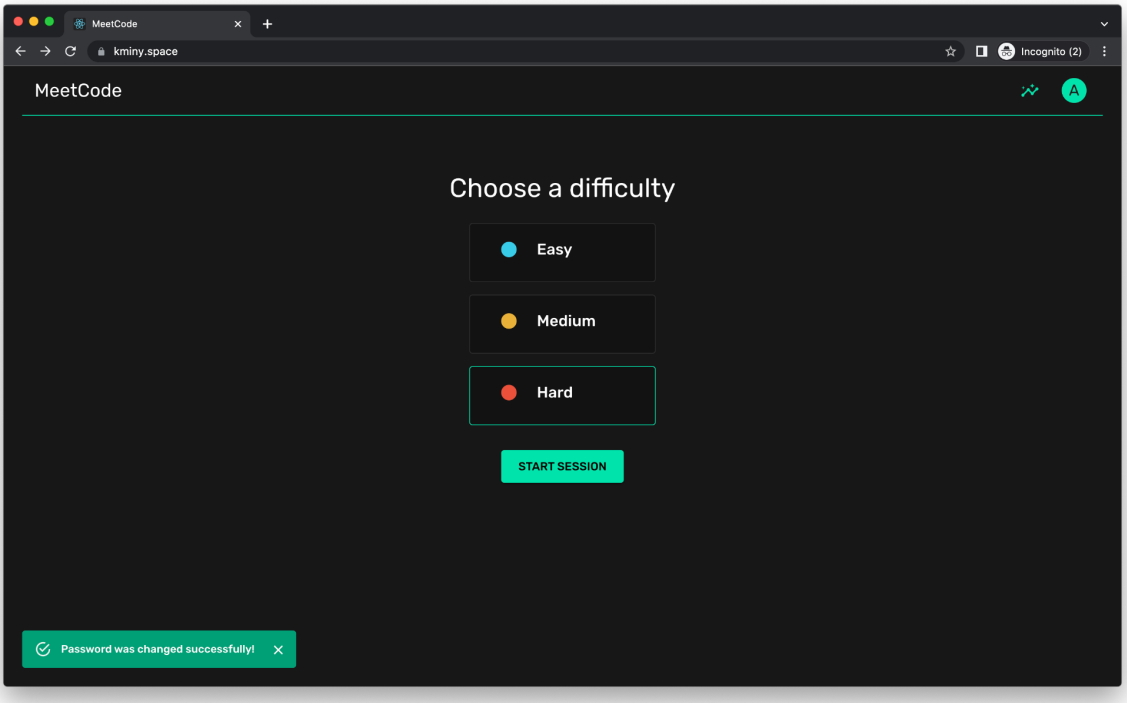

Figure 10 – Notification when Password has been successfully updated

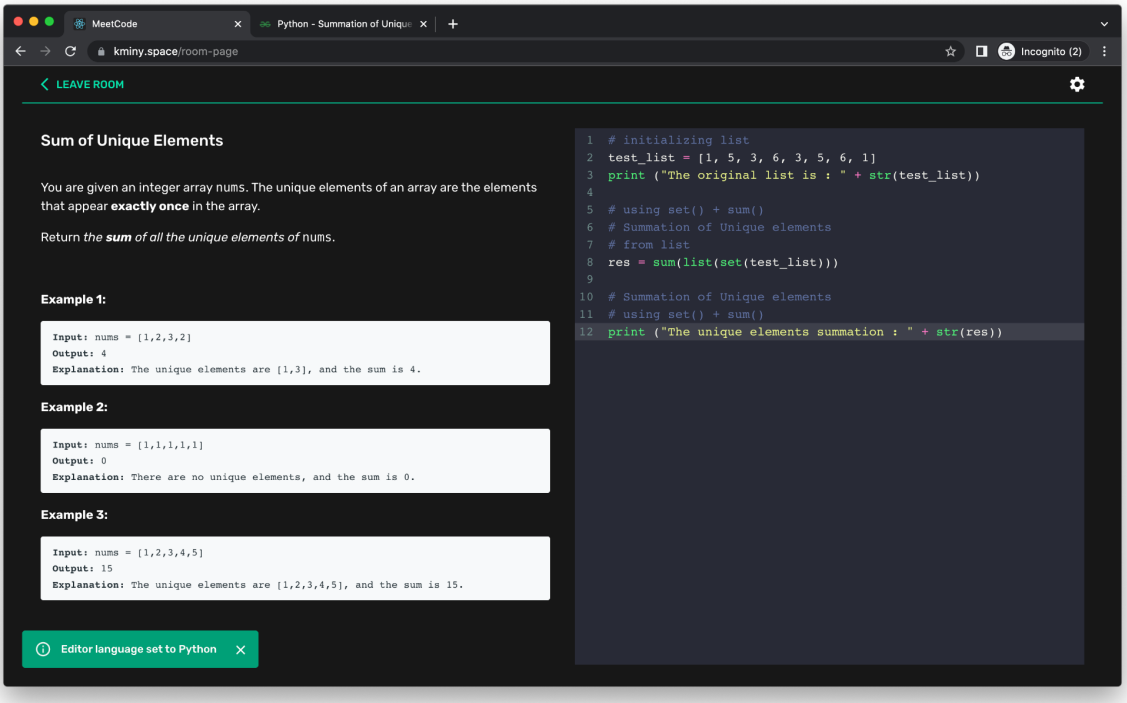

Figure 11 – Notification when editor language has been updated

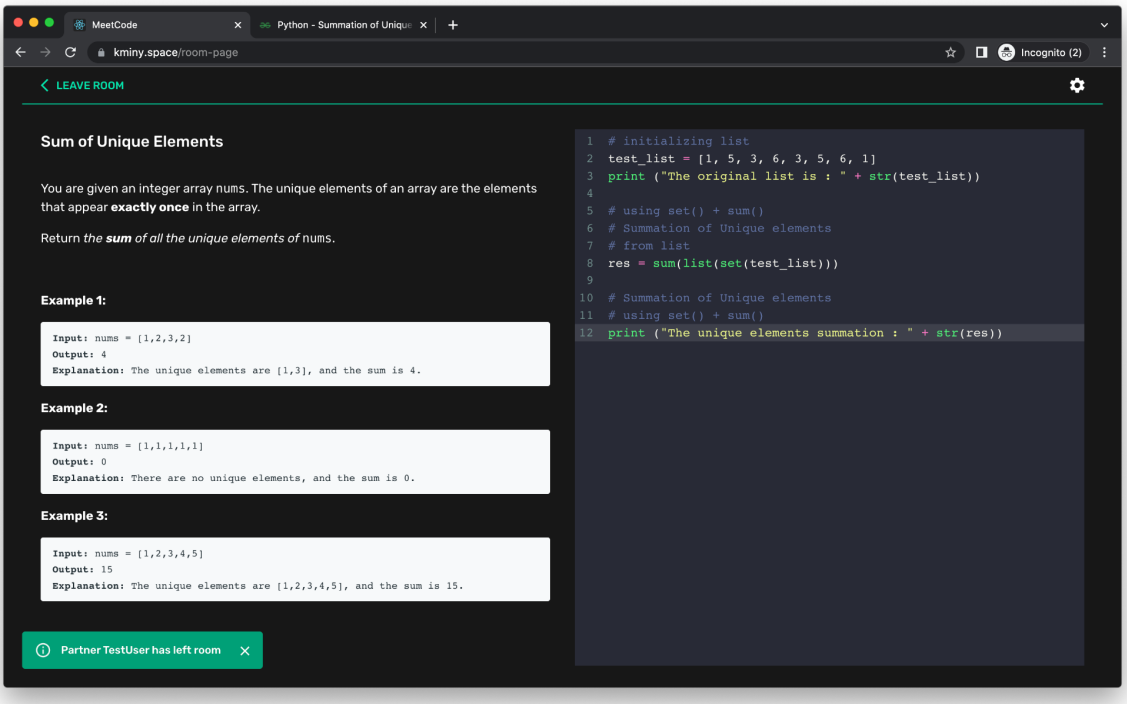

Figure 12 – Notification when partner has left the room

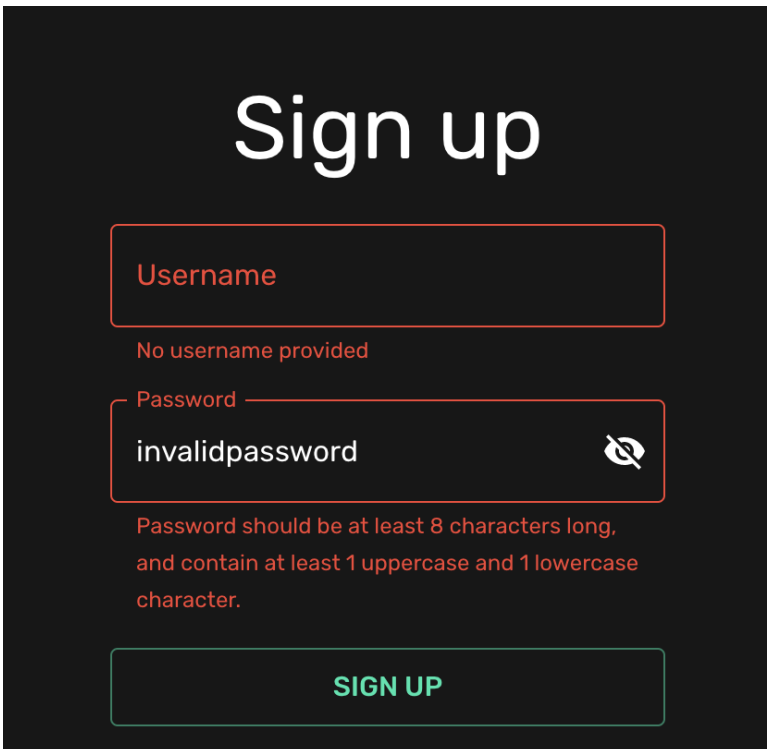

Figure 13 – Error messages for username and password validation

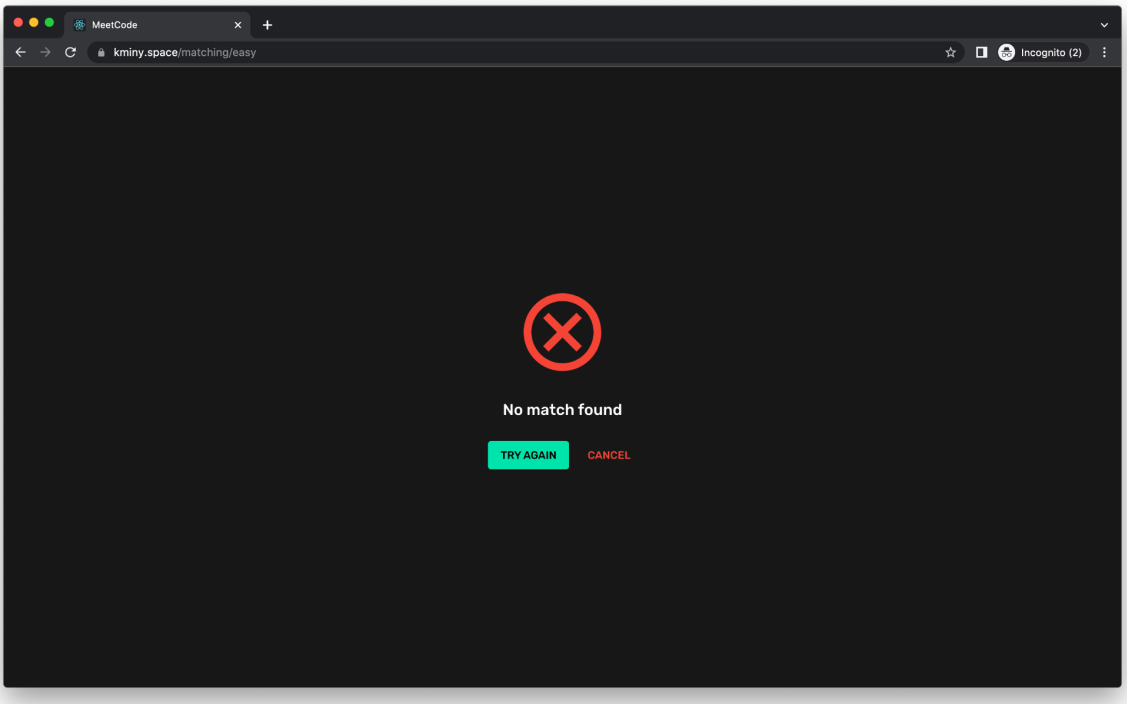

Figure 14 – Message when no match is found

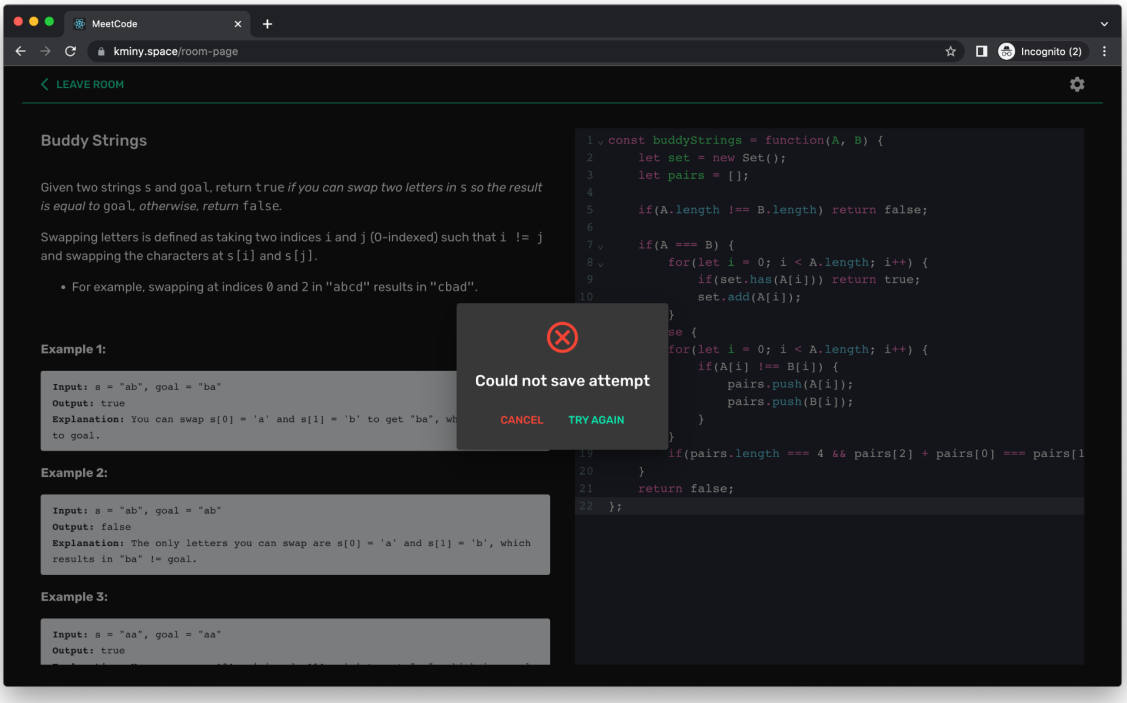

Figure 15 – Error message when user's attempt could not be saved

3. Usage of tooltips to explain icon buttons

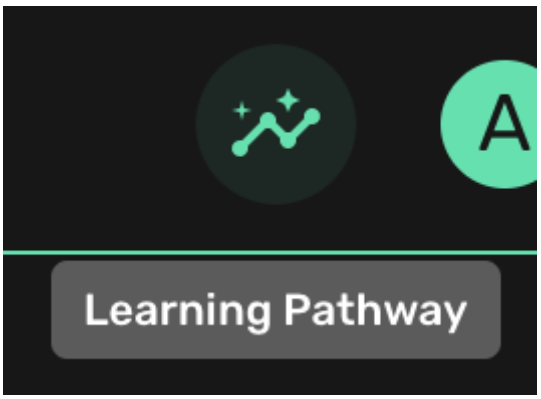

Figure 16 – Tooltip to indicate Learning Pathway Page

NFR1.2: Responsive Design

Though our web application is designed to be used on laptops and desktop computers, mainly devices with larger screen sizes, some users may wish to use our web application on a split screen, possibly to view documentation on another window simultaneously. To account for this, our frontend has a responsive design.

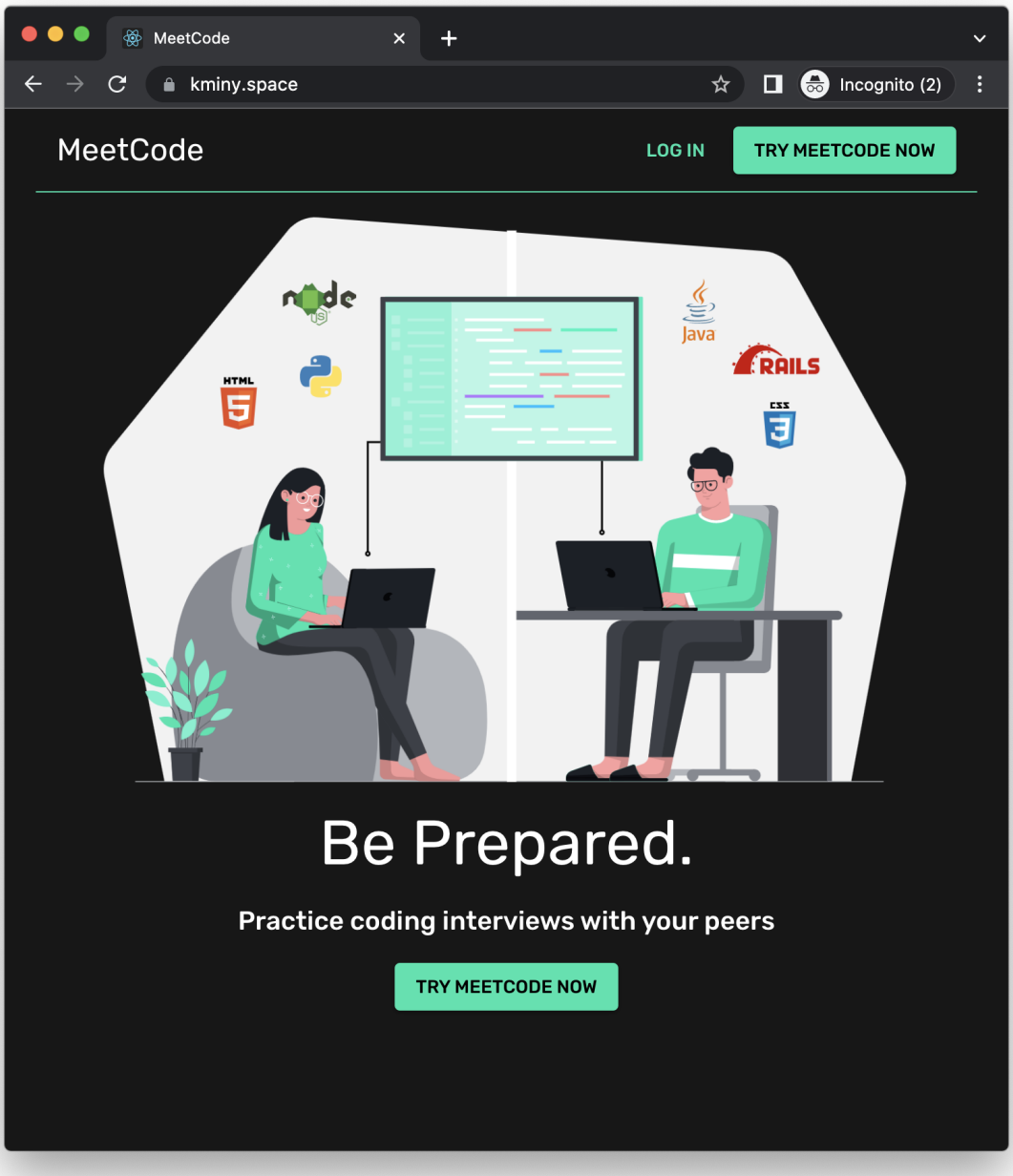

Figure 17 – Responsive Design for Landing Page

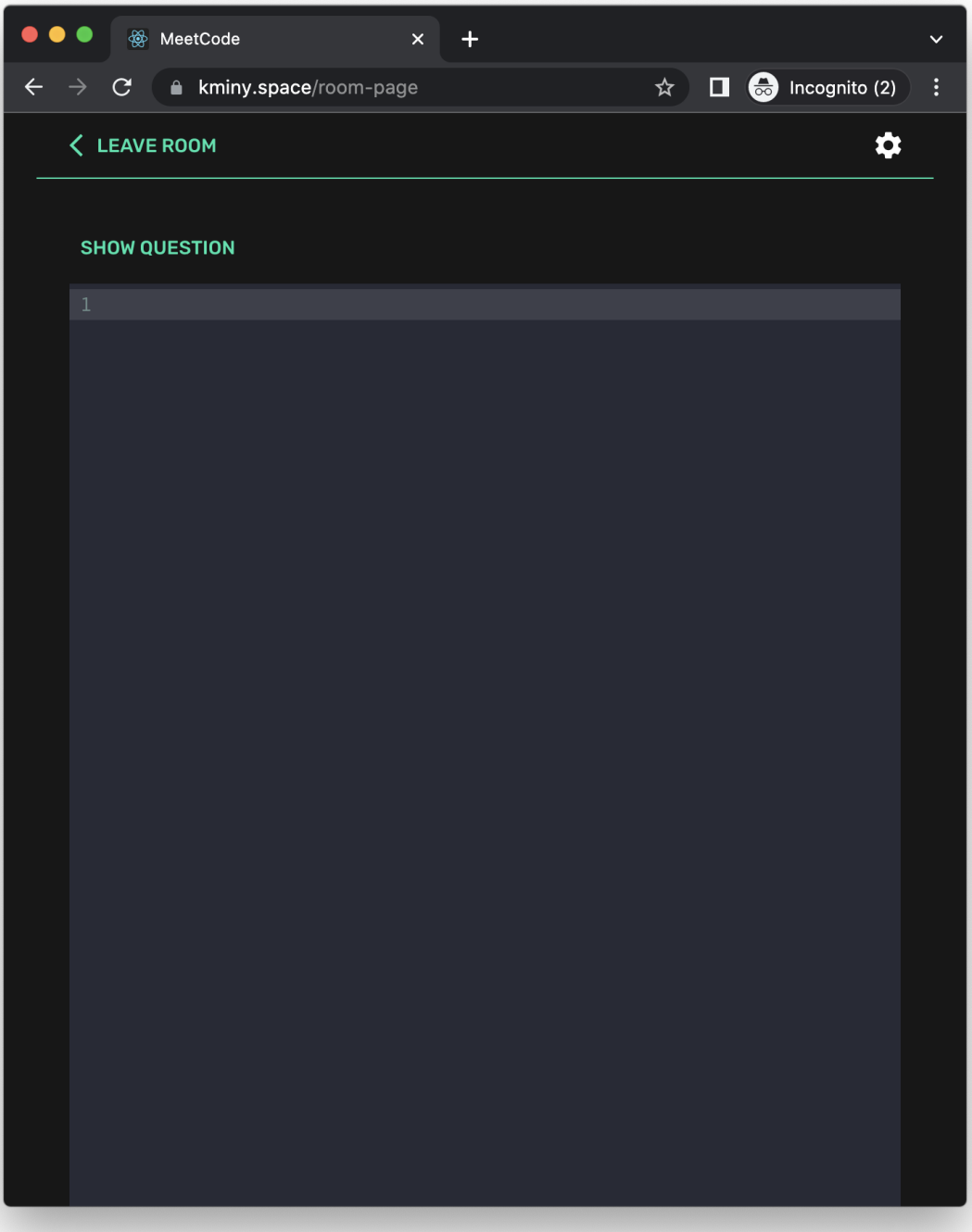

Figure 18 – Responsive Design for Room Page (Editor)

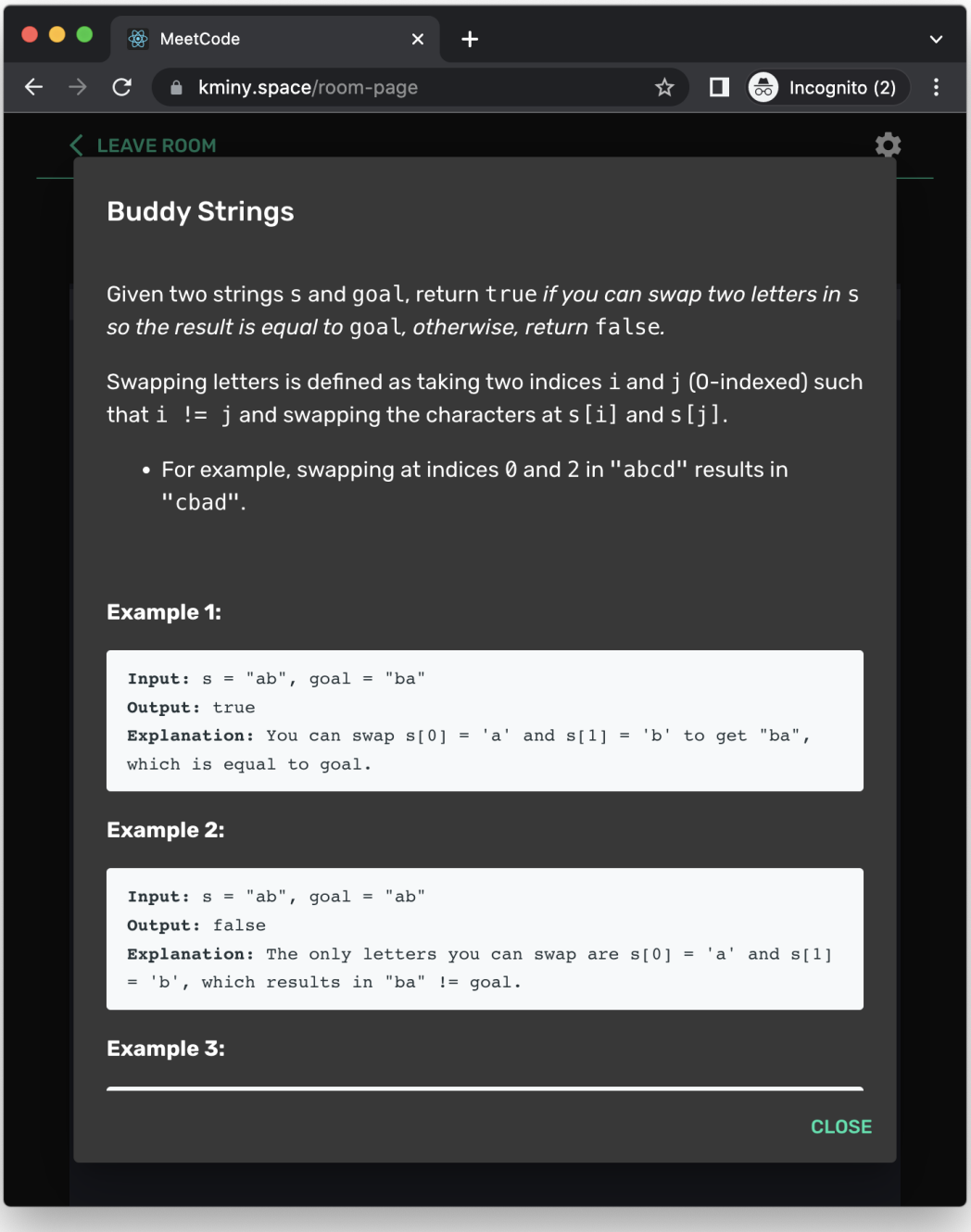

Figure 19 – Responsive Design for Room Page (Question Modal)

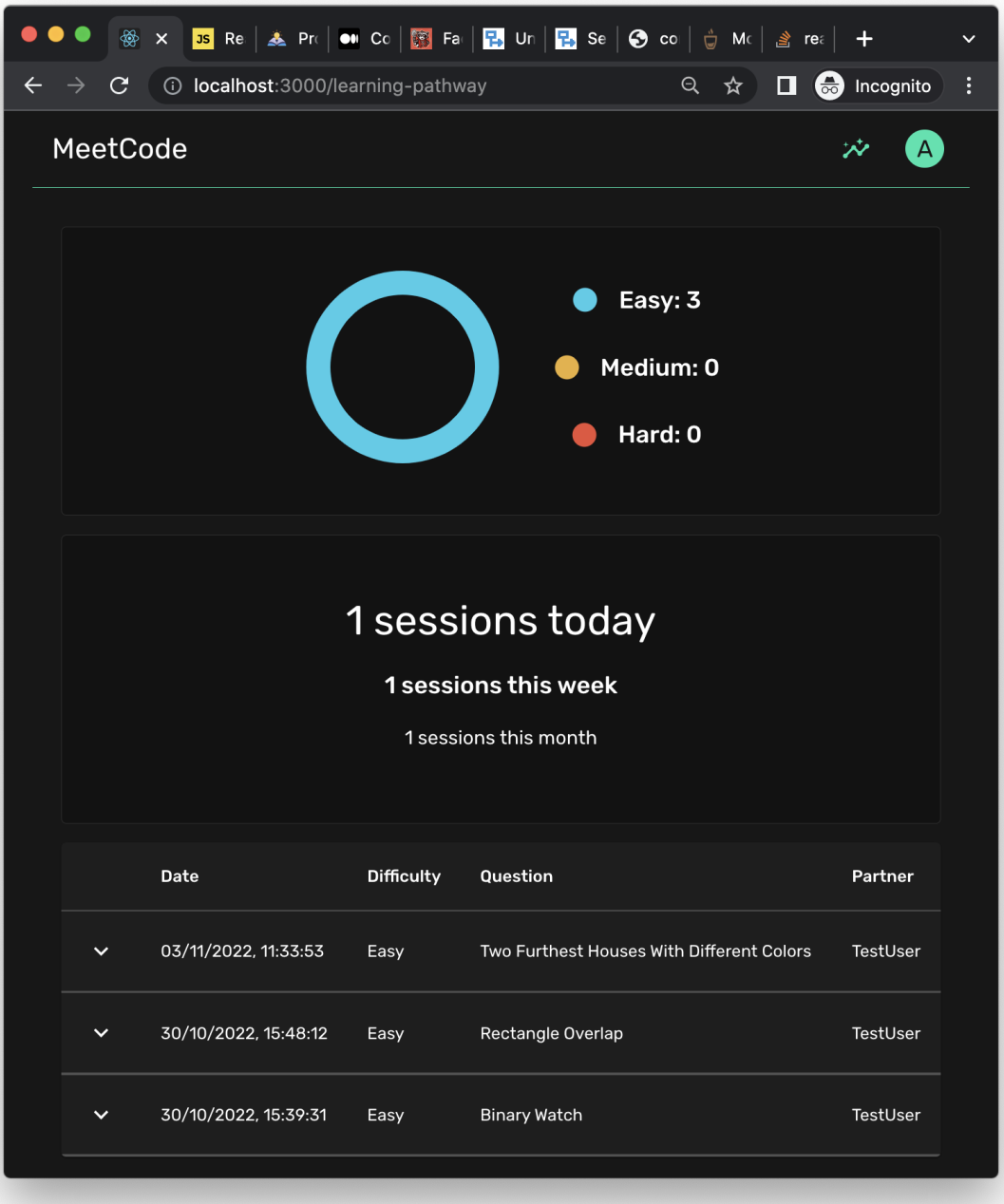

Figure 20 – Responsive Learning Pathway Page

Scalability

In order to ensure that the performance of the system does not deteriorate with increasing network traffic, we need to be able to scale services automatically in those circumstances. With the adoption of microservice architecture, each service can be scaled out independently by adding more instances of that service. This allows the system to support more users while maintaining a relatively low response time for each user request and thus provide better user experience.

As the services are deployed on a distributed infrastructure, a load balancing software is necessary to spread the workload across instances to ensure no single instance gets overwhelmed.

To achieve the above, our application is deployed using Amazon Elastic Container Service(ECS) with an application load balancer to help us distribute incoming application traffic across multiple targets. We can then set up a variety of auto scaling policies that will help us start new instances when the conditions are met. Some examples are scaling when metrics such as CPU utilisation or request count exceeds a certain threshold. These policies can be applied to scale services (to serve the request), EC2 instances (computing resource) and even load balancers (to serve the traffic).

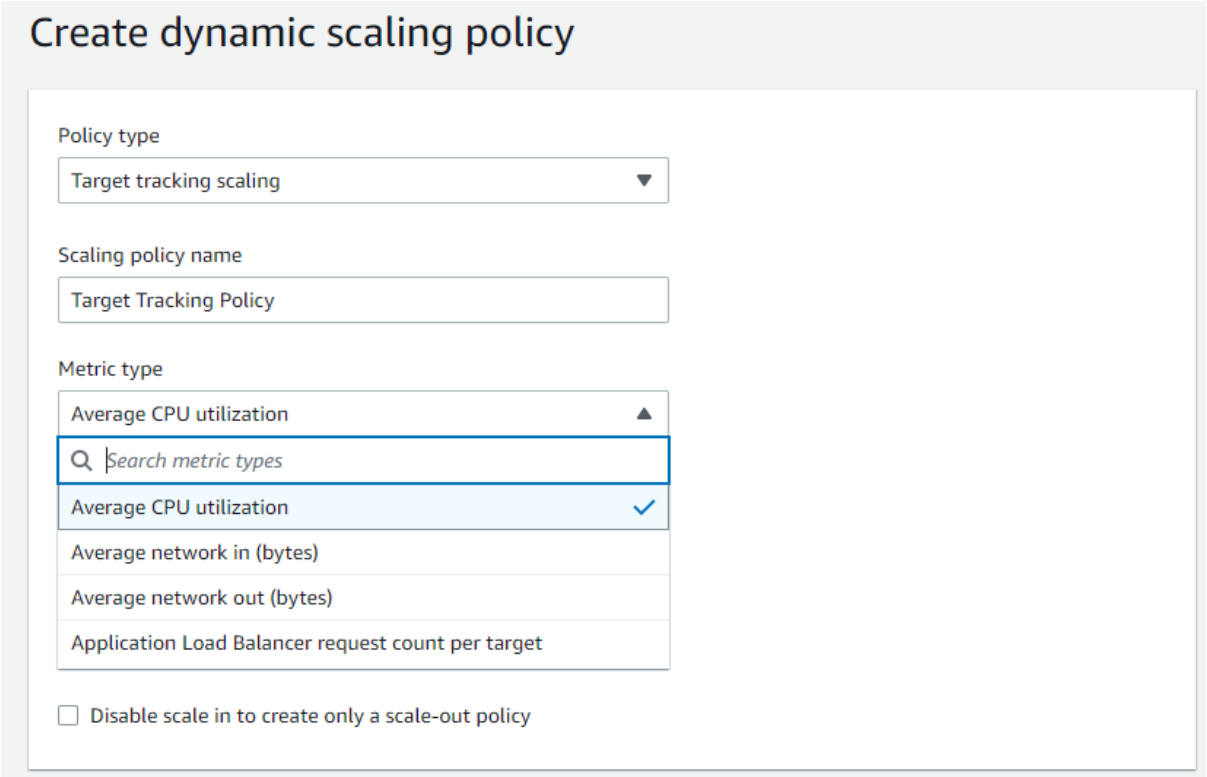

Figure 21 – Setting up of a scaling policy

Robustness

To ensure the system was robust and continued to function when confronted with invalid inputs, input validation in the frontend, as well as request body validation in the backend microservices was conducted extensively.

In the sign up and login screens, validation is performed on the username and password inputs to ensure that they are valid, before being sent to the user service. If invalid, error messages will be displayed to the user as shown in figure 13.

In the backend microservices, validation is once again conducted on the request parameters and body to ensure they are valid. An instance of validation in the Learning Pathway Service, when creating a record of the user's coding attempt, is shown below:

```
export const createRecordValidator = [
  check('user_id').notEmpty().isMongoId().custom(async value => {
    const isUserIdValid = await validateUserId(value)
    if (isUserIdValid) {
      return true
    }
    throw new Error('Invalid User ID');
  }),
  check('partner_username').notEmpty(),
  check('question_difficulty').custom(value => {
    if (value === DIFFICULTIES.EASY || value === DIFFICULTIES.MEDIUM ||
value === DIFFICULTIES.HARD) {
      return true;
    }
    throw new Error('Difficulty must be either \"EASY\", \"MEDIUM\", or
\"DIFFICULT\"');
  }),
  check('question_id').notEmpty().isNumeric().custom(async (value, {req})
=> {
    const questionTitle = await
validateQuestionId(req.body.question_difficulty, value, req)
    if (questionTitle) {
      req.body.question_title = questionTitle;
      return true
    }
    throw new Error('Invalid Question ID');
  }),
  check('code').exists(),
  check('code_language').custom(value => {
    if (value === LANGUAGES.JAVA || value === LANGUAGES.JAVASCRIPT ||
value === LANGUAGES.PYTHON) {
      return true;
    }
    throw new Error('Code language must be either \"JAVA\",
\"JAVASCRIPT\", or \"PYTHON\"');
  }),
  check('timestamp').notEmpty().isISO8601().toDate(),
]
```

Developer Documentation

Development Process

Our team decided on the Agile Methodology in our development process, modified slightly for our purposes. The development process was divided into three sprints in total, with the milestone / final submission marking the end of each sprint.

At the beginning of the project, we set our goals for the project in terms of the services we planned to implement and deployment tasks we intended to complete. Based on these goals, we planned the following outline for each sprint:

Sprint 1	24 Aug – 18 Sept	● User Service ● Matching Service ● Frontend (for the corresponding services)
Sprint 2	19 Sept – 15 Oct	● Question Service ● Collaboration Service ● Frontend (for the corresponding services) ● Continuous Integration
Sprint 3	16 Oct – 9 Nov	● Learning Pathway Service ● Frontend (for the corresponding service) ● Deployment ● Continuous Deployment

At the beginning of each sprint, we decided on the functional requirements that we wanted to complete by the end of each sprint. Functional requirements that were labelled as of high importance, or were prerequisites to the completion of other tasks were prioritised. From there, we plotted a Gantt Chart of the tasks and the people in charge of completing them. Buffer time was also included at the end of each sprint in anticipation of difficulties completing the tasks on time.

Over the course of each sprint, weekly stand-up meetings were held, where each member updated the team with their progress on the tasks they were assigned, and any issues that they faced. Based on the sharing and group discussion, the Gantt Chart was updated, deadlines were shifted or tasks were reassigned.

Tech Stack

React (Frontend)	● Existence of many useful packages / libraries in the React Ecosystem (e.g. component libraries (MUI), routing libraries (React Router), code editor library (CodeMirror) → good developer experience ● Highly performant due to virtual DOM ● Component reusability within the project

	● Existing familiarity with React from the developers
ExpressJS / NodeJS (Backend Microservices)	● Popular web application framework that provides broad features for building web applications ● Good community support
MongoDB (User / Question / Learning Pathway Service Database)	● Built on a scale-out architecture ● Ease of developing scalable applications with evolving data schemas
Socket IO (Matching Service)	● Allows for easy bidirectional communication between client(s) and server(s).
PostgreSQL (Matching Service Database)	● Existing familiarity with developers ● ACID compliance means that concurrent requests will not affect correctness of program behaviour, which can be important for matching service.
CodeMirror (Collaboration Service)	● Very extensible with available extensions for syntax highlighting and line numbers ● Provides handling of conflicting edits when people to make changes at the same time
Github Actions (CI / CD)	● Enables automatic testing on push and pull request ● Enable automatic building and uploading of docker images to ECR
Docker (Deployment)	● Simplify deployment ● Service can be redeployed pulling the latest image from ECR
ECS (Deployment)	● Able to deploy and manage scalable clusters of Docker containers ● Autonomous provisioning of EC2 instances and auto-scaling

Overall System

Architecture Diagram

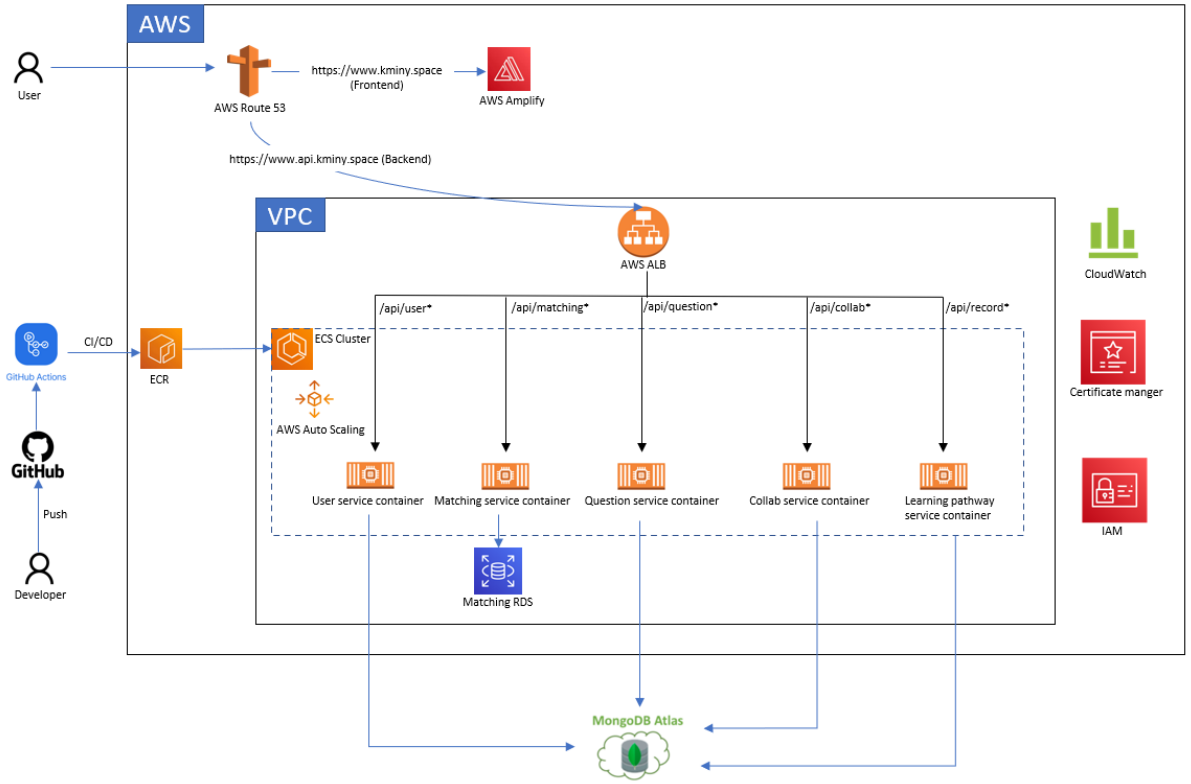

Figure 22 – Architecture Diagram

Add-on services:
AWS CloudWatch - collects and visualises real-time logs, metrics, and event data of resources
AWS IAM - manage identities and access to AWS services and resources
AWS Certificate Manager - provision and manage SSL/TLS certificates

Microservice Architecture

For our application, we chose to use a microservice architecture instead of a monolithic architecture.

The key advantage of using microservices over monolithic architecture is the ability to scale the microservices independently from one another, as compared to scaling the entire backend when using a monolithic architecture. For example, the User Service may only receive a staggered series of login and logout requests at the beginning and end of user sessions, but the Matching Service may be receiving a surge of matching requests from the currently logged in users over a short period of time. In this instance, it is clear that the Matching Service needs to be more scalable than the User Service, and with a microservices approach, we can launch more instances (EC2 instances/containers) running the Matching Service without having to do the same for the User Service.

Another important advantage is the ease of development of microservices compared to a monolith. Having independent and loosely coupled microservices reduces the size of the code base that developers have to work with, reducing the development overhead, as well as reducing the need for coordinating deployment efforts, allowing us to deploy our services independently. Using a monolithic architecture would increase the coupling of the separate aspects of the backend that we would work on, making development and deployment more complex and slowing it down. Therefore, using the microservices architecture would fit well with the Agile Development process that our team chose.

A notable disadvantage of choosing microservices over monolithic is the increased debugging complexity and difficulty of initial deployment, due to the overhead of setting up the many separate microservices. This is true at the initial stage of development and deployment of the application, but with reduced coupling of the application's components, it is easier in future iterations of development to pinpoint where bugs are located, as well as reduce the probability of introducing fresh bugs upon making a small change in the code.

Design Pattern: Model View Adapter (MVA) Architecture

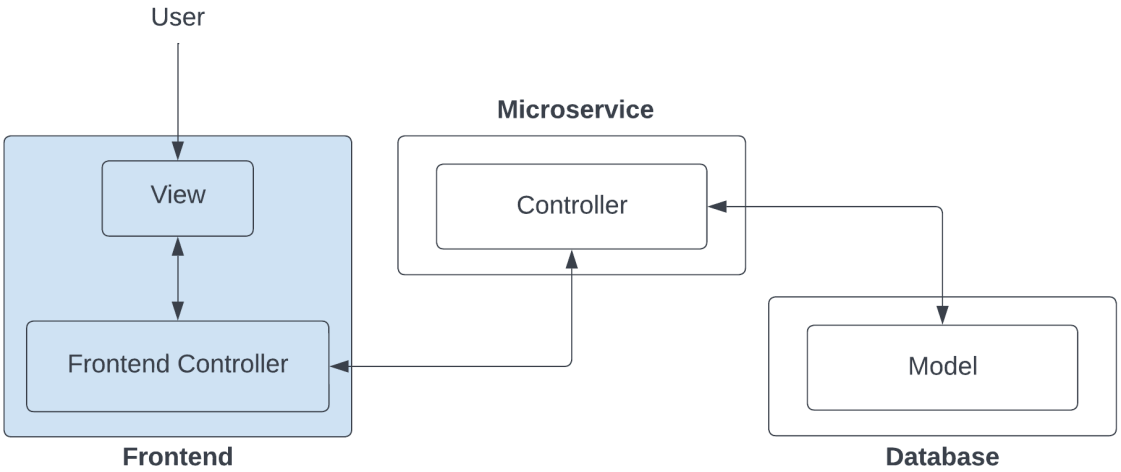

Figure 23 – Model View Adapter (MVA) Architecture

For our application, we followed the Model View Adapter (MVA) architecture, as shown above. All communication between the View and the Model goes through the Controller. This architecture makes use of the Adapter pattern, where the Frontend Controller is an adapter between the Frontend and the Microservice, and the Controller (specifically the ORM layer) is an adapter between the microservice and the database.

An example of the MVA architecture at work would be the Matching Process, as depicted below.

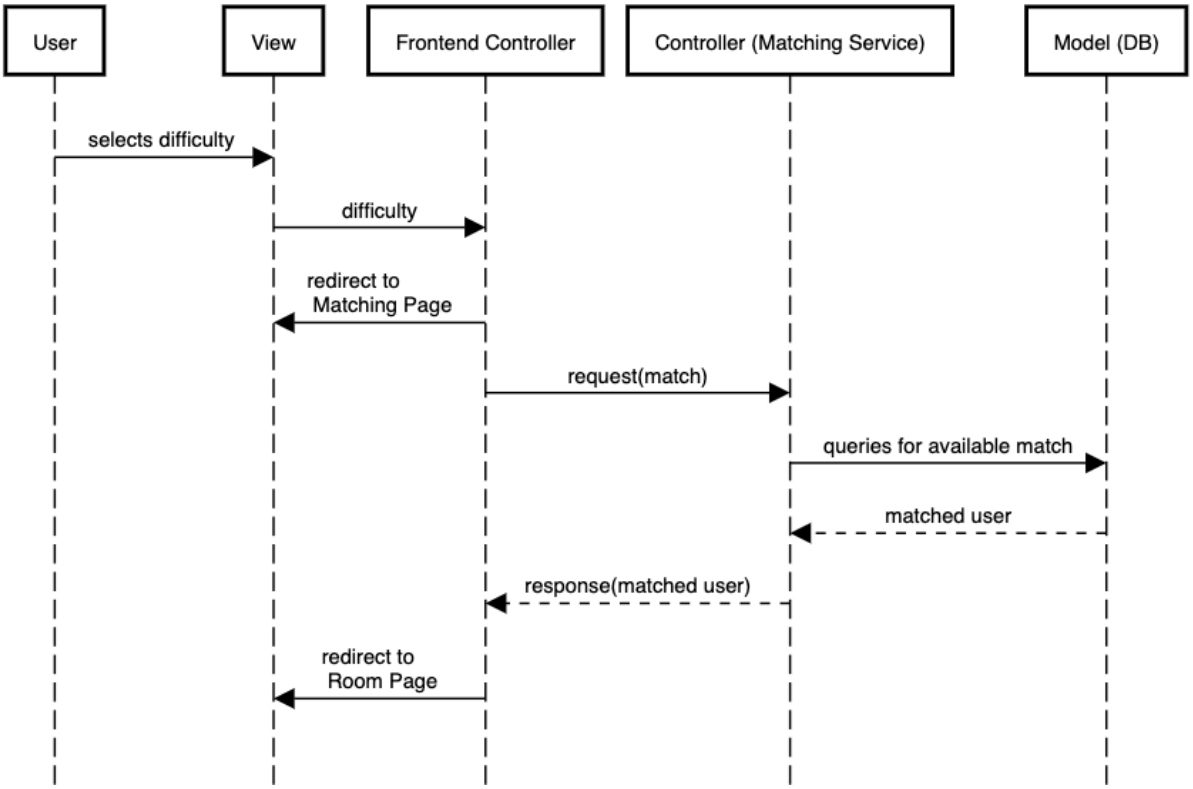

Figure 24 – Matching Process

The User interacts with the View, clicking the difficulty of the question they wish to attempt. The difficulty is communicated to the Frontend Controller, which redirects the user to the Matching Page, and simultaneously emits an event to the Matching Service with the specified difficulty, requesting a match. The Matching Service queries the model in the database for an available match. When one is found, the controller receives the details of the matched user from the database and emits an event to the Frontend Controller. Since a match has been found, the Frontend Controller redirects the user to the Room Page, updating the View.

As can be seen, the different aspects of the application have separate responsibilities:

View	● Communicates user input to the Frontend Controller
Frontend Controller	● Updates the View ● Performs API calls / emits events to retrieve / update the Model
Controller (Microservice)	● Retrieves / updates the Model based on API calls / events received ● Sends response / emits event with the Model information to the Frontend Controller
Model (Database)	● Stores application data

This modular architecture was chosen as it adheres to the Separation of Concerns principle, making the code more easily maintainable. In addition, by ensuring the application data flows through the Controller, instead of allowing the View to directly interact with the Model, coupling is reduced and cohesion is increased.

Database Design

When designing the database, we considered two options: (1) having a shared database between microservices, and (2) having separate databases for each microservice. We weighed the following considerations:

Using separate databases for each microservices would lead to higher scalability. Firstly, it reduces coupling amongst different services, allowing them to be developed, scaled and deployed independently. Different services also have different requirements for their data: unlike the other microservices, the Matching Service has strong consistency requirements, thus using a relational database with ACID properties would be more appropriate than using a NoSQL database with BASE properties. The other databases we used are MongoDB, which has the strength of having a more scalable schema due to its document model, which allows for fields to be added freely. Thus, separate databases allow for specific requirements of each service to be met. In addition, it adheres to the Separation of Concerns principle.

Using the same database for all the microservices would increase the ease of performing the following operations:

1. Validation of fields between different microservices
    a. E.g. ensuring the user ID and question ID stored in the Learning Pathway Service are valid
2. Joining fields between different microservices
    a. E.g. Joining the question IDs stored in the Learning Pathway Service with the corresponding questions in the Question Service

However, while more complex to perform, these operations are still possible with different databases for each microservice. For validation of fields, validation APIs were defined and called from the Learning Pathway Service to ensure the data provided is correct. For joining fields between different microservices, it is possible to define a facade class or service to handle this operation, as seen in this section.

Therefore, we chose to use separate databases for each microservice for our application for the following reasons:

1. Scalability is an important non functional requirement for our web application
2. It allows us to meet specific service requirements (scalability for some services, strong consistency for others)
3. Difficulties of using separate databases can be overcome, as described above

Frontend

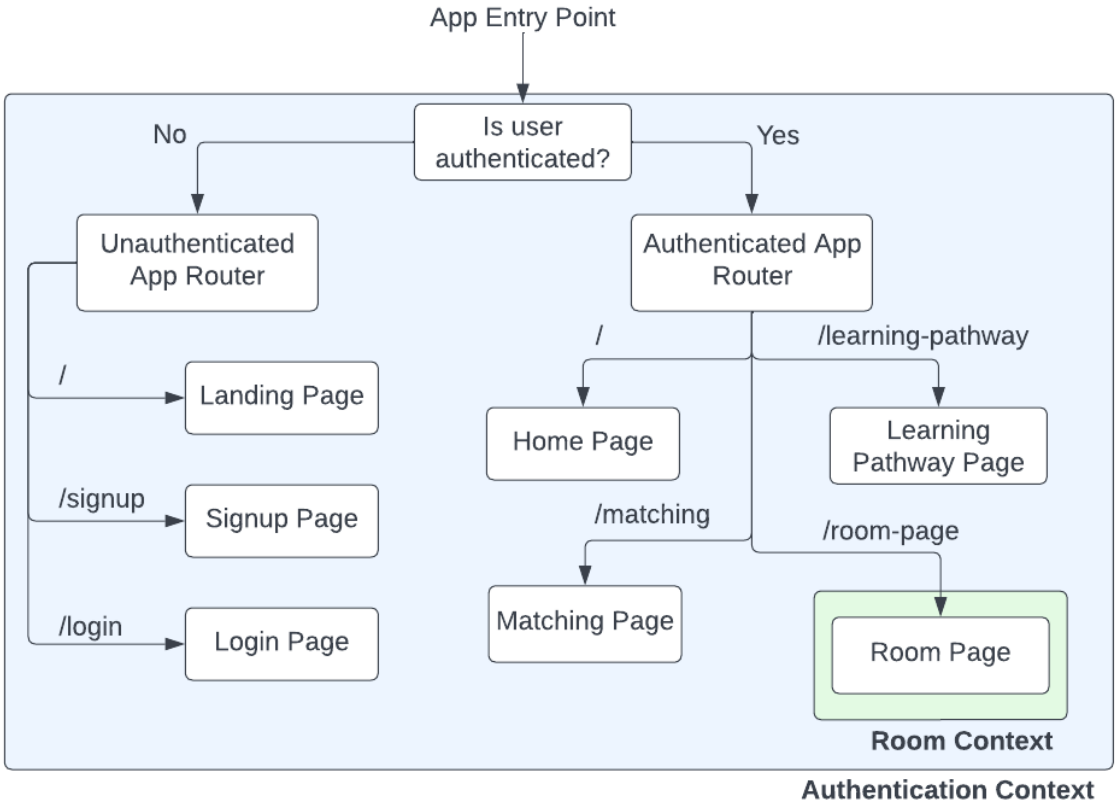

Figure 25 – Frontend Architecture

The figure above depicts the routing architecture of the frontend. First, the application checks if the user is authenticated, by checking if there is a refresh token stored in local storage, and attempting to exchange it for an access token.

If the user is not authenticated, they are directed to a Landing Page, where they can access the Signup Page to create a new account, or visit the Login Page to log into their account.

If the user has been authenticated and the frontend has a valid access token, the user is directed towards the Home Page. From there, they can select a question difficulty and start a session, where they will then be redirected to the Matching Page during the matching process. When they have been successfully matched, they will be redirected to the Room Page, where they can attempt the question of selected difficulty with a partner. They can return back to the Home Page at any point in time, where they can access the Learning Pathway Page to view records of their previous attempts.

Splitting the router into two different routers, one for the unauthenticated pages, and the other for the authenticated pages, follows the Single Responsibility Principle as each router is only responsible for one portion of the application that is not closely related to the other, and also follows the Separation of Concerns Principle.

Separation of Model, View and Controller

Within the frontend, when necessary, the model, view and controller is separated into different modules to differing extents depending on the context.

For example, the Room Page is divided into the

1. View, which consists of the different components that make it up (e.g. `QuestionDisplay`, `Editor`, `Header`, `SettingsModal`)
2. Model, which is the `RoomContext` that the page is wrapped in
3. Controller, the hooks used to modify the model and communicate with the various microservices (e.g. `useRoomContext`, `useQuestion`)

This abstraction of parts down to the Model, View and Controller is most useful for complex pages of the application, such as the Room Page, which has a large amount of data coming from or transmitted to various different microservices. For instance, the partner's username is retrieved from the Matching Service, the question is retrieved from the Question Service, and the attempt details are to be transmitted to the Learning Pathway Service. This is because the modularity allows for separation of concerns, which reduces the complexity and coupling between different components, as well as allows for greater extensibility and addition of new fields and ways to handle inputs, improving the development experience.

However, adhering to this Model – View – Controller architecture also requires a level of overhead that is unnecessary for simpler aspects or pages of the application. In line with the AHA (Avoid Hasty Abstractions) principle, in less complex pages, the model and controller are sometimes condensed into a single hook, such as in the Learning Pathway Page, which uses the `useRecords` hook to fetch data from the Learning Pathway Page, but also reformats and stores the data as its state, which is then passed down to the various components.

Usage of React Context for Data

To handle data in the frontend, two options were considered, React Context and Redux. Even though Redux is more performant as it reduces component re-renders, and works well with both dynamic data and static data, React Context was chosen for the following reasons:

1. Reduces setup overhead
2. More suited to our use case – given that most data we need to handle is static and the application is not data-heavy
3. Minimises bundle size since it ships with React

As depicted in figure 25, the entire application was wrapped with an Authentication Context, which stores the current user's user details, and the Room Page was wrapped with a Room Context, which stores details related to the current coding session. The usage of React Context acts as a single source of truth, ensuring the same data is propagated to all components, especially when the data is updated by other components. It also reduces prop-drilling, as components that require the data can consume it directly.

Despite the advantages of using React Context, we only used it in situations where many components needed to access the context (e.g. Application Context) or when there was a large amount of information specific to a particular context (e.g. Room Context). This is due to the overhead associated with setting the context up, and to avoid hasty abstractions.

Reusable Modularized Components

An advantage of using the React framework is that it promotes highly modularized components that can be easily reused. In writing the frontend code, each page was split into individual components following the Single Responsibility Principle. Components were also designed to be reusable by avoiding context-specific methods following the Single Level of Abstraction Principle. This allowed for the reuse of components such as `AuthForm`, which was used in both the Login and Signup Page, as well as the `QuestionDisplay` component, which was used in both the Room Page and Learning Pathway Page.

Design Pattern: Facade

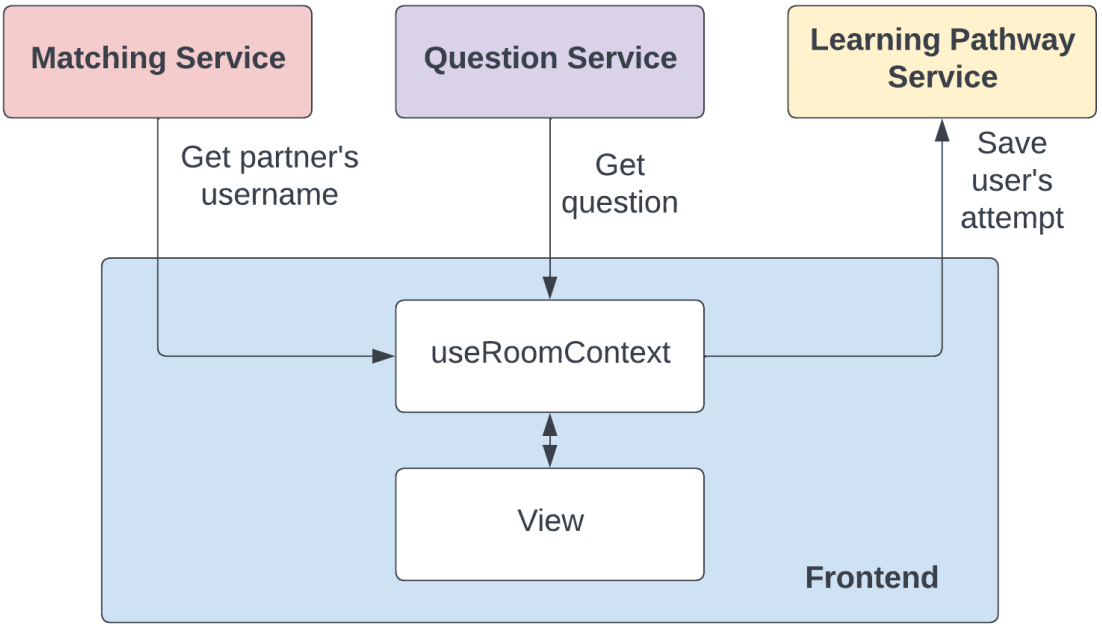

Figure 26 – Facade Pattern

When the user is in the Room Page, the frontend needs to either retrieve data from or send data to different microservices. The useRoomContext hook acts as a facade to interact with the different microservices, and is responsible for (1) retrieving a question from the Question Service, (2) retrieving the partner's username from the Matching Service, and upon teardown, (3) saving a record of the user's attempt to the Learning Pathway Service. Isolating the interactions to a single controller reduces complexity and ensures the other services are coupled to a single portion of the application, rather than many parts.

User Service

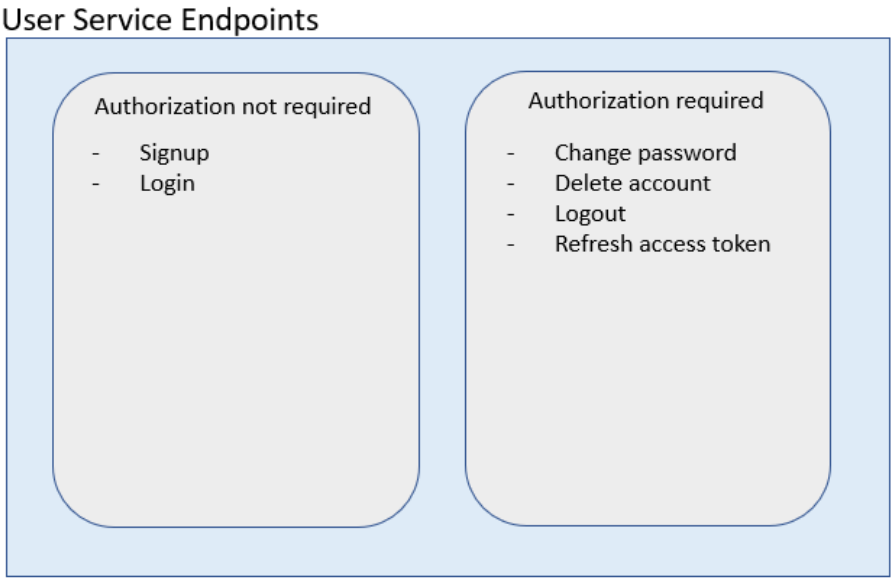

Figure 27 – Some endpoints require JWT token authorization

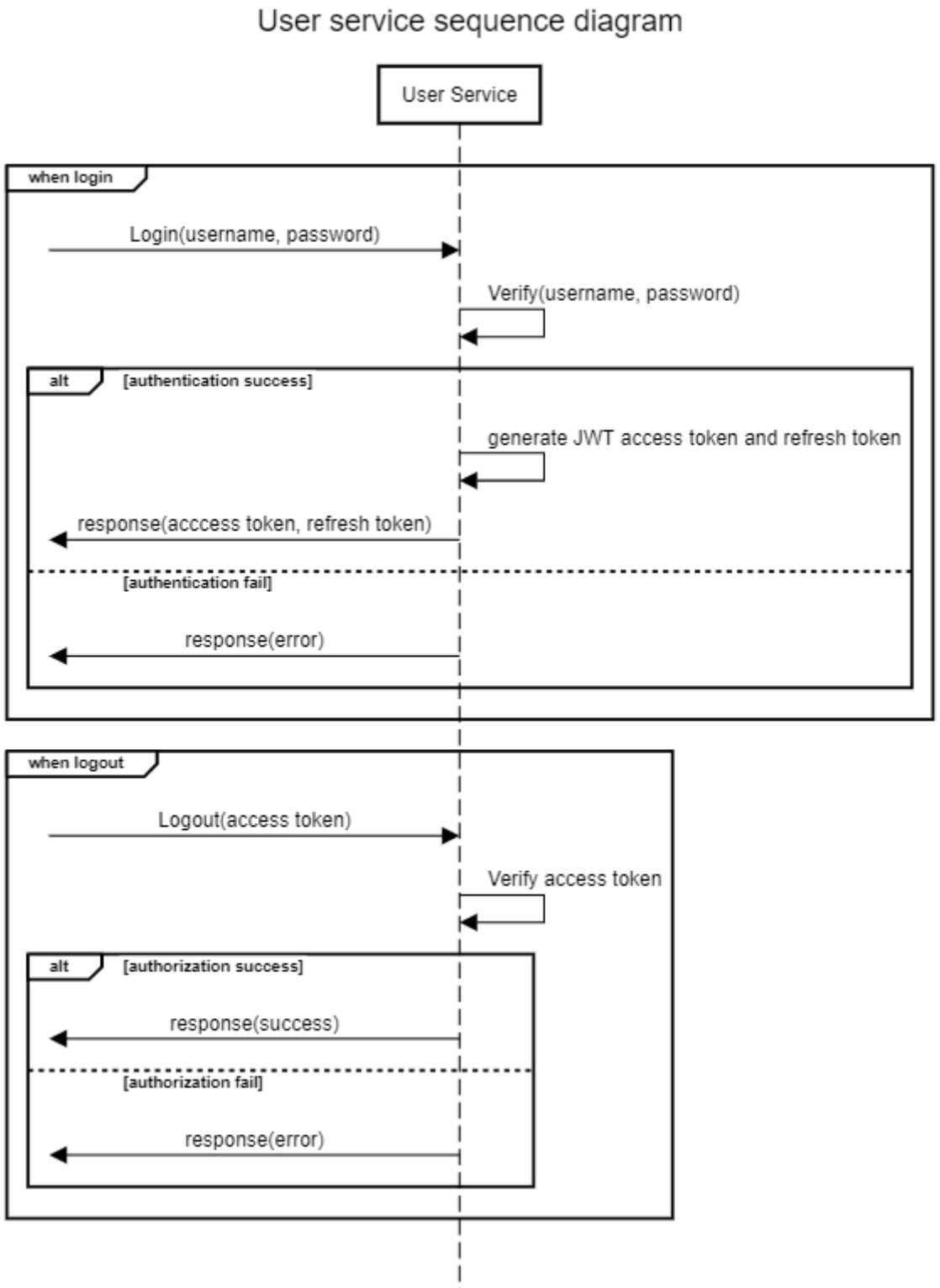

Figure 28 – Sequence diagram showing authentication and authorization process

As shown in the figure above, two JWT tokens are generated and returned - access token and refresh token. Access token is used for authorisation while refresh token is used to get a new access token when it expires. With a refresh token, the user does not need to sign in again before the refresh token expires.

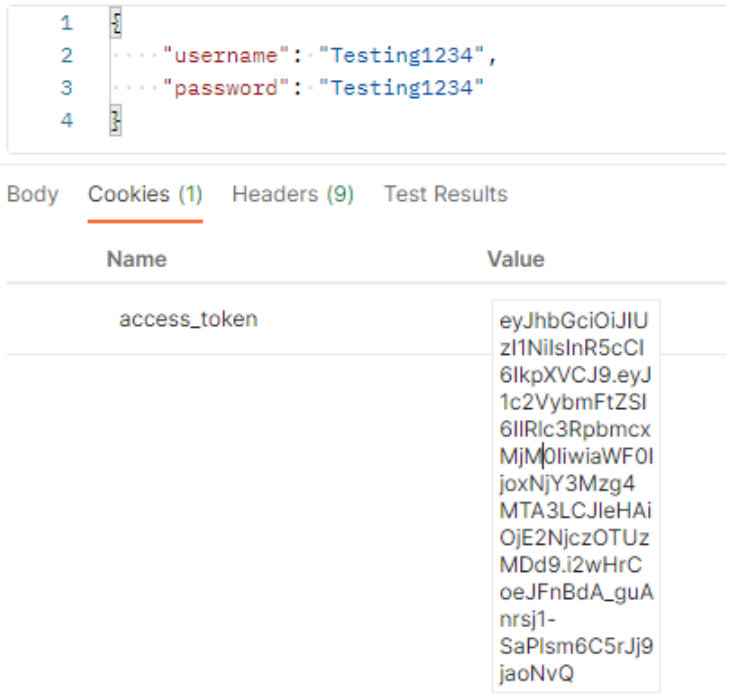

Figure 29 – Example of an access token store in cookie after login

Information encrypted in token

The tokens generated have **username** encrypted in the token payload. Subsequent tasks performed that require authorization will refer to the username stored in the tokens instead of getting the client to supply it in the request. This ensures that the user is only able to perform account related tasks on the account which they log into and not any other accounts.

As the token is encrypted with a digital signature, the information cannot be tempered with without being detected.

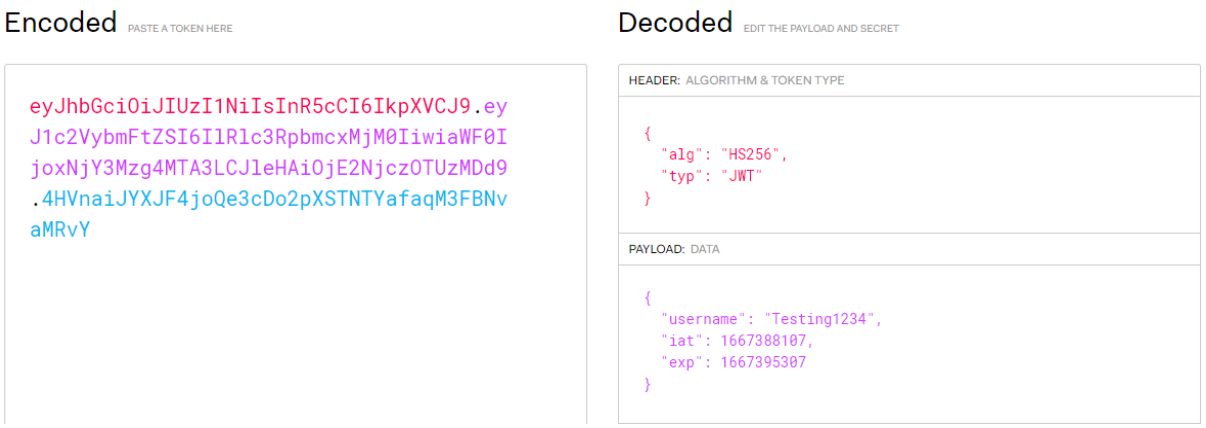

Figure 30 – Username can be seen under the payload section after decoding the token

Design Pattern

The middleware pattern in express can be seen as a variant of pipe and filter pattern, or intercepting filter pattern. This allows us to add new filters easily to our request processing pipeline. Therefore, our authorization check is just another filter before the controller serves the actual request. This pattern is not only used in our user service but also many of our other services to do input validations, such as the Question Service and Learning Pathway Service.

API

POST	**/api/user**	Create a new account with the provided username and password
Request Body:	username	Username must be unique
	password	Passwords need to be at least 8 characters long with both upper and lower case Passwords will be hashed using bcrypt
POST	**/api/user/login**	Authenticate the user with the provided username and password
Request Body:	username	Username of the account to login to.
	password	Password hash should match the one stored in the database.
Response:	accessToken	JWT token used for authorization. Stored in cookies.
	refreshToken	JWT token used for requesting new access token. Stored in local storage.
POST	**/api/user/refresh_access_token**	Get a new access token without needing to login.
Request Body:	refreshToken	JWT token giving after login.
GET	**/api/user/logout**	Logout and remove tokens
PUT	**/api/user/chage_password**	Update user old password with provided new password.
Request Body:	oldPassword	Current user password.
	newPassword	New password should meet the minimum requirement.
DELETE	**/api/user**	Remove user account from database.
POST	**/api/user**	Validates whether the user ID provided is that of an existing user

Request Body:	user_id	User ID to be validated

Matching Service

The Matching Service matches two users who wish to attempt a question of the same difficulty. It works by emitting and listening to events with the Frontend component, and this is achieved by using Socket.IO which allows easy bi-directional communication. PostgreSQL is used as the database, because of its strong consistency guarantees (compared to eventual consistency in MongoDB), which is important because we do not want to have a situation where 2 different users match with the same user because of data inconsistencies. Instead of using other packages to interface between NodeJS and PostgreSQL (such as node-postgres, where raw SQL queries are written), we used an ORM called Sequelize which is generally more readable than raw SQL and has other features which are easy to use, such as more human-readable queries and synchronising "schemas" easily, making the code more maintainable.

Two relational entities, PendingMatch and Match, are used in the implementation. PendingMatch represents a user that is finding a match of a specific difficulty, and thus has 2 fields, the user's name, and the difficulty opted for. Match represents a pair of matched users as well as the difficulty they have both selected, and thus it is similar to PendingMatch but with an additional field for the 2nd user.

To find a match, the service first looks for any PendingMatch of the same difficulty;

- If there is one, then delete that PendingMatch, and create a Match between the current user and the user in the deleted PendingMatch, that is then stored in the database.
- If there is none, a PendingMatch for this user is created and stored in the database

To implement the finding match timeout that occurs 30 seconds after attempting to find a match, the Frontend component will emit an event 30 seconds after (unless a match for this user has been found) it emits a "user finding match" that matching service listens to, so as to delete the relevant PendingMatch. Implementing a timer at the server side instead would not be as feasible since it would noticeably increase server load.

Design Principles/Patterns

Observer pattern was used to notify the frontend of relevant events (also considered state changes). In index.js, an array of server-side sockets (i.e the observers) is stored as a variable, so that when we want to notify the appropriate users (i.e their frontends) of events such as finding a match successfully or their matched user has left the room, we can find the correct server-side socket using that array, and use it to emit the relevant event to the correct frontend (i.e the observer, thru the server-side socket).

Single Responsibility Principle was also employed; All Sequelize calls to modify/query data in the database was placed in a file called "matching-orm.js", all code to handle initialising of database instance was placed in "repository.js", and all the code to handle/emit events was placed in "index.js". By making sure each file has its own responsibility, we reduce coupling

as much as possible, making it easier to pinpoint where a bug could be, and making the code more maintainable.

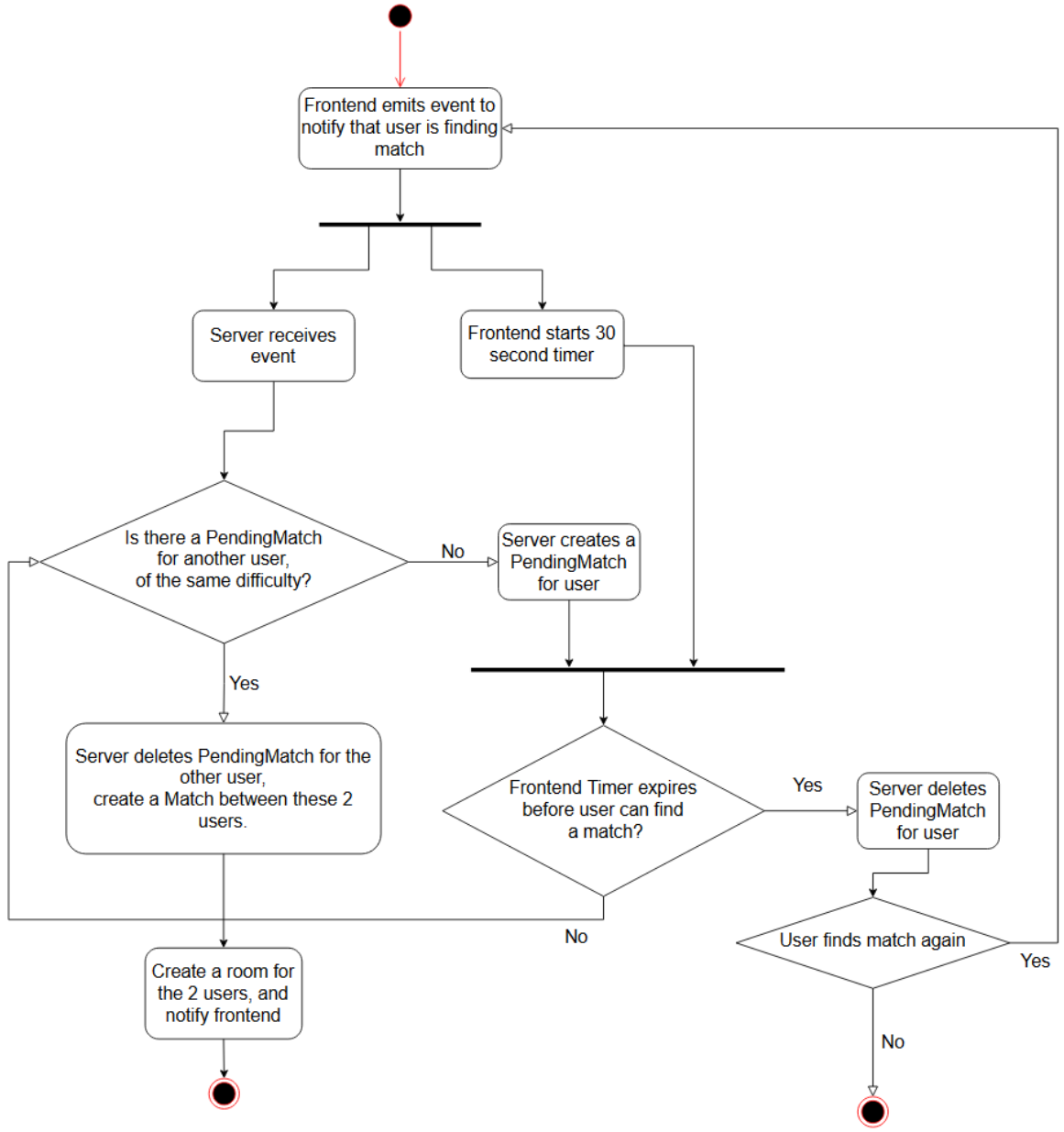

Figure 31 – Activity Diagram of User Matching Process

API

/api/matching	Client-side socket used to listen for events from server and emit events to server

Events

Event name	Object attributes / Callbacks	Description
“user finding match”	*userName, difficulty*	Emitted by frontend when user tries to find a match of a certain difficulty
“matchSuccess”	*matchedSocketId, room*	Emitted by the server for both users that are matched. *matchedSocketId* is the id of the socket of the other user, and *room* is the “shared space” used by both users to communicate with each other
“matching timer expired”	*userName*	Emitted by frontend to inform the server to delete the PendingMatch for this user.
“user disconnected”	*userName*	Emitted by frontend so that server can delete the Match and PendingMatch for this user.
“get partner name”	*userName*	Emitted by frontend to get matched partner’s name
“user leave room”	*userName*	Emitted by frontend when user leaves room, to do all the cleanup operations
“matched user left room”		Emitted by server for relevant frontend updates.

Question Service

Question service is Meetcode’s way of storing its question bank. MongoDB was chosen for the Question Service Database and 3 separate databases, easy, medium and hard are created containing questions with that specified difficulty. question_text was stored as HTML code as it would be easier for the frontend to pull and render.

The database stores the question bank, which records the following fields:

question_id	Number	Question ID of the question
question_title	String	Title of the question
question_text	String	HTML text of the question

```
{
    "_id": "634815f6fb7fe6ef567cbaa9",
    "id": 1025,
    "title": "Divisor Game",
    "body": "<div class=\"content__u3I1 question-content__JfgR\"><div><p>Alice and Bob take turns playing
        a game, with Alice starting first.</p>\n<p>Initially, there is a number <code>n</code> on the
        chalkboard. On each player's turn, that player makes a move consisting of:</p>\n<ul>\n<li>Choosing
        any <code>x</code> with <code>0 &lt; x &lt; n</code> and <code>n % x == 0</code>.</
        li>\n<li>Replacing the number <code>n</code> on the chalkboard with <code>n - x</code>.</li>\n</
        ul>\n<p>Also, if a player cannot make a move, they lose the game.</p>\n<p>Return <code>true</code>
        <em>if and only if Alice wins the game, assuming both players play optimally</em>.</p>\n<p> </
        p>\n<p><strong class=\"example\">Example 1:</strong></p>\n<pre><strong>Input:</strong> n =
        2\n<strong>Output:</strong> true\n<strong>Explanation:</strong> Alice chooses 1, and Bob has no
        more moves.\n</pre>\n<p><strong class=\"example\">Example 2:</strong></p>\n<pre><strong>Input:</
        strong> n = 3\n<strong>Output:</strong> false\n<strong>Explanation:</strong> Alice chooses 1, Bob
        chooses 1, and Alice has no more moves.\n</pre>\n<p> </p>\n<p><strong>Constraints:</strong></
        p>\n<ul>\n<li><code>1 &lt;= n &lt;= 1000</code></li>\n</ul>\n</div></div>",
    "createdAt": "2022-10-13T13:43:18.978Z",
    "updatedAt": "2022-10-13T13:43:18.978Z",
    "__v": 0
}
```


Figure 32 – Example question retrieved in JSON

To ensure the question retrieved by users is the same room is the same, when requesting a randomly chosen question of specified difficulty, the room ID (in UUID format) is sent with the request. This is used to seed a random number generator, which will return the same output when the UUID provided is the same. The index of the question to be returned is calculated based on the random number generated.

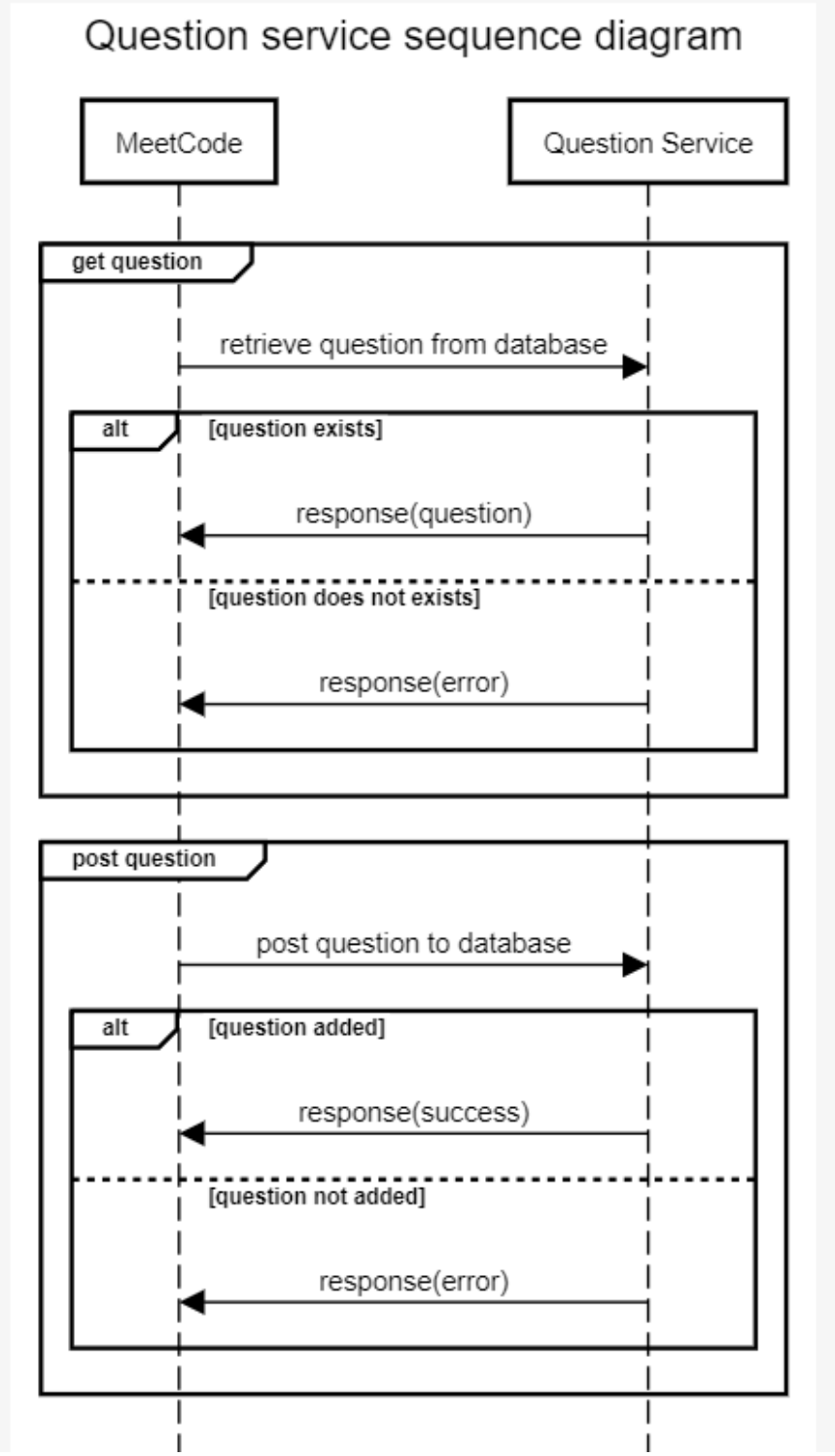

Figure 33 – Sequence Diagram showing retrieving and creating of questions

It is worth noting that the creation of new questions is currently done manually by developers through programming interfaces. There is no frontend UI to streamline the process. It is planned for future iterations when we implement role based access control where only the admin role has the permission to access the create question page.

API

GET	**/api/question/:difficulty/:id**	Retrieve a question with a specified difficulty and id
Response:	id	Question number
	title	Question title
	body	Question text
GET	**/api/question/random/:difficulty/:uuid**	Retrieve a random question with a specified difficulty
Response:	id	Question number
	title	Question title
	body	Question text
POST	**/api/question/:difficulty**	Create a new question with a specified difficulty
Request Body:	id	Question number
	title	Question title
	body	Question text
POST	**/api/question/validate/:difficulty/:id**	Validate whether a question with specified ID and difficulty exists
Response	title	Question title
DELETE	**/api/question/:difficulty/:id**	Delete a question with specified ID and difficulty
Response	message	Deletion message

Collaboration Service

The collaboration service consists of two parts, the frontend code editor and the backend WebSocket server for exchanging document updates.

Code Editor

The code editor makes use of CodeMirror which supports editing features, and has a rich programming interface to allow further extension. Some editing features are syntax highlighting, line numbers and auto complete. CodeMirror also supports many languages such as the popular ones like Python and Java.

Alternatives 1: Basic Text Area
Apart from the feature benefits we can get using CodeMirror as mentioned above, the main reason we use CodeMirror is that it supports real-time collaborative editing. This means that we are saved from the complexity of implementing the logic of handling conflicting edits — since network communication isn't instantaneous, allowing people to make changes at the same time.

Alternatives 2: Ace or Monaco
There are other code editors out there such as Ace and Monaco. The main reason that we choose CodeMirror is its extensibility. All the coding features mentioned above have been implemented as extensions and packages, allowing us to customise the editor easily. The extensions are very generic, this means that we can also write our own extensions to fine tune the features that we so chose to provide.

The above consideration leads to the choice of CodeMirror as our users are mainly going to interact with our code editor most frequently. Therefore, the extensibility and customizability of CodeMirror is a big plus point for our future updates.

Collaboration WebSocket Server

The collaboration WebSocket server makes use of the WebSocket API to open a two-way interactive communication session between the user's browser and the server to exchange document updates.

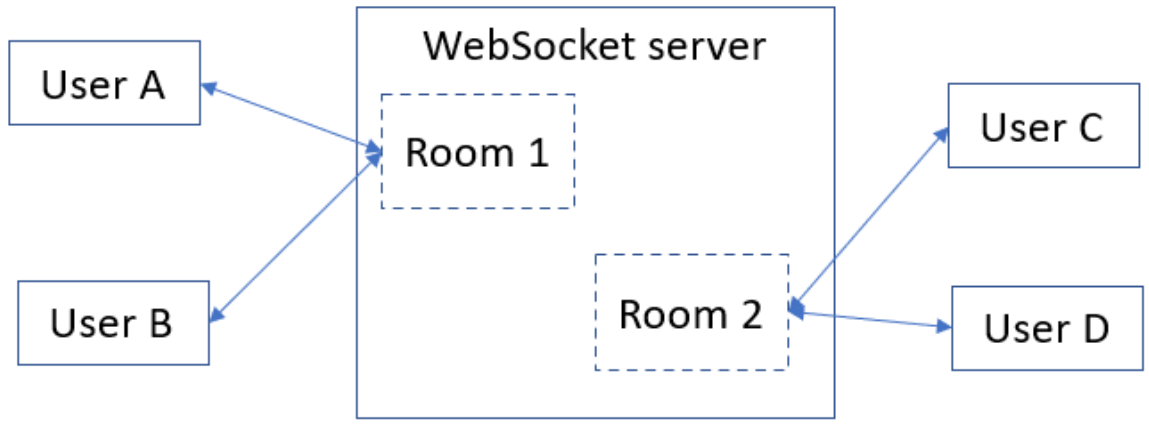

Figure 34 – Collaboration WebSocket Server

As shown in the diagram above, users connect to a single endpoint over Websocket and push any updates to the server through the endpoint. The server then distributes the updates to the other user, this whole process of exchanging updates follows the Publisher-Subscriber pattern. The communication is also event driven, where updates are exchanged through event messages carrying the updates.

As WebSocket provides full-duplex communication, push and receiving update events can take place simultaneously, resulting in lower overhead and better throughput.

Design considerations:

- **Alternative**: Use WebRTC for communication
  Pros: peer to peer communication, does not need a server to sit in between
- **Current:** Websocket
  Reason: WebSocket is better suited for transmitting text/string data whereas webRTC is primarily designed for streaming audio and video content

Learning Pathway Service

Validation

Implementing validation for the Learning Pathway Service was more complex due to NFR6.2 and NFR6.3, since the validation relies on the User and Question Services. Thus, validation APIs were identified in the User and Question Services to be called from the Learning Pathway Service, to verify the validity of the User ID and Question ID. When validating the Question ID, if the provided Question ID is valid, the corresponding Question Title will be retrieved for storage as well.

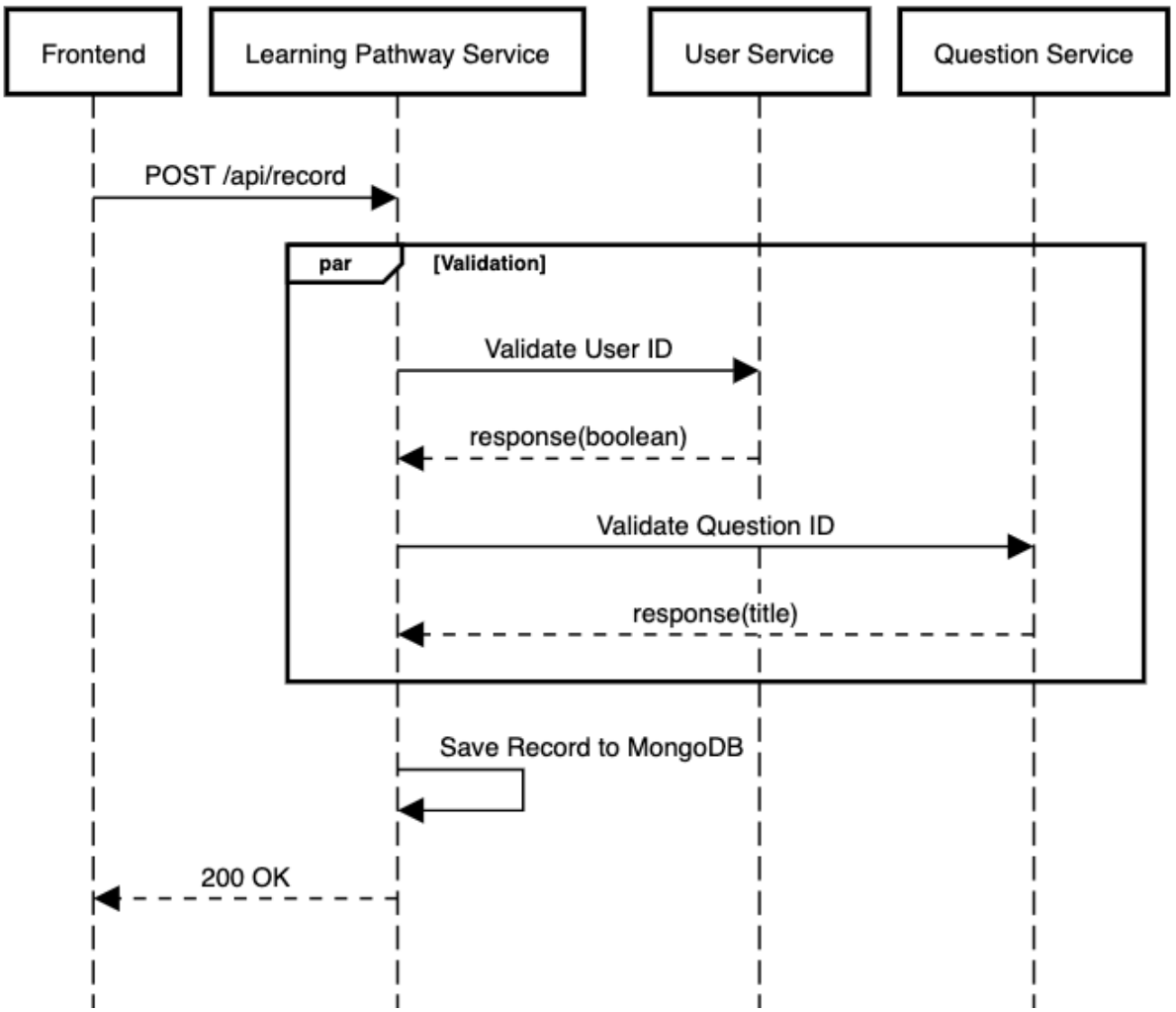

Figure 35 – Sequence Diagram of Saving User Attempt

Database Schema

MongoDB was chosen for the Learning Pathway Database due to the ease of scaling the database schema. This allows for extensibility of adding additional fields when improving the Learning Pathway Service, such as adding a commenting system, which allows users to add a comment reflecting on their attempts for future review.

The database stored documents of each user's attempt, which records the following fields:

_id	ObjectID	ID corresponding to the attempt
user_id (string)	String	User ID of the user that this attempt belongs to
partner_username (string)	String	Username of the partner that the user attempted the question with

question_difficulty	String Enum: “EASY” / “MEDIUM” / “DIFFICULT”	Difficulty of the question attempted
question_id	Number	Question ID of the question attempted
question_title	String	Title of the question attempted
code (string)	String	Code written by the user and their partner
code_language	String Enum: “JAVA” / “JAVASCRIPT” / “PYTHON”	Language that the code written is in
timestamp	ISODate	Datetime of when the user started their attempt

String Enums were used for the question_difficulty and code_language fields to ensure no invalid values are entered.

The question title was also stored as part of the attempt details. Despite the fact that this was duplicated from the Question Service storage, this reduces the complexity of querying for the question titles from the Question Service when the user retrieves all records of their previous attempts. This is particularly due to the short length of the question titles, which means that the cost, both in terms of space and monetary cost, of storing them is relatively low.

API

GET	**/api/record/:userId**	Get all records associated with the provided userId
DELETE	**/api/record/:userId**	Delete all records associated with the provided userId
POST	**/api/record**	Creates a new record with the provided fields
Request Body:	user_id (string)	User ID of the user that this attempt belongs to
	partner_username (string)	Username of the partner that the user attempted the question with
	question_difficulty (string: “EASY” / “MEDIUM” / “DIFFICULT”)	Difficulty of the question attempted
	question_id (number)	Question ID of the question attempted

	code (string)	Code written by the user and their partner
	code_language (string: “JAVA” / “JAVASCRIPT” / “PYTHON”)	Language that the code written is in
	timestamp (string representation of date, using the ISO 8601 format)	Datetime of when the user started their attempt

Deployment

We are using GitHub actions for both our continuous integration and continuous deployment workflows.

Continuous Integration (CI) Workflow

The GitHub action for CI will run test cases for all our services on push to the main branch and any pull requests.

The steps in the workflow:

1. Set up NodeJS
2. Set up database server
3. Run all the test cases for each service

```yaml
name: Test
on:
  push:
    branches:
      - main
  pull_request:
  workflow_dispatch:

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [16.x]
        mongodb-version: [6.0]

    steps:
      - name: Checkout source code
        uses: actions/checkout@v3

      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}

      - name: Start MongoDB
        uses: supercharge/mongodb-github-action@1.8.0
        with:
          mongodb-version: ${{ matrix.mongodb-version }}

      - name: Run All user service Tests
        run: |
          cd user-service
          npm install
          npm run test
```

Figure 36 – Workflow for user service CI

Continuous Deployment (CD) Workflow

CD workflows are only run when changes are committed to the main branch.

We will be pulling the current task definition of the service and updating the image url to the latest one. This means that we will need to have the task definition already defined on the AWS console first. In addition, environment variables should be added or updated through the console as we do not modify them in our workflow.

The steps in the workflow:

1. Build and push image to Amazon ECR
2. Download task definition of the service
3. Updated the AWS ECS task definition with the new image ID
4. Deploy the new AWS ECS task definition

```yaml
- name: Build, tag, and push image to Amazon ECR
  id: build-image
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    ECR_REPOSITORY: user
    IMAGE_TAG: latest
  run: |
    cd user-service/
    docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
    echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> $GITHUB_OUTPUT

- name: Download task def
  run: |
    aws ecs describe-task-definition --task-definition user --query taskDefinition > ./task-definition.json

- name: Fill in the new image ID in the Amazon ECS task definition
  id: task-def
  uses: aws-actions/amazon-ecs-render-task-definition@v1
  with:
    task-definition: ./task-definition.json
    container-name: user
    image: ${{ steps.build-image.outputs.image }}

- name: Deploy Amazon ECS task definition
  uses: aws-actions/amazon-ecs-deploy-task-definition@v1
  env:
    ECS_SERVICE: user
    ECS_CLUSTER: cs3219-project-ECSCluster-mdgWyO8rQXbZ
  with:
    task-definition: ${{ steps.task-def.outputs.task-definition }}
    service: ${{ env.ECS_SERVICE }}
    cluster: ${{ env.ECS_CLUSTER }}
    wait-for-service-stability: true
```

Figure 37 – Workflow to deploy the latest user service

Future Improvements

The learning pathway service can be further enhanced by allowing users to tag their coding attempts (e.g. based on types of question, or whether further review/revision is needed), or add comments to reflect on their attempt.

One enhancement would be an addition of a communication service for users to communicate either via a video feed or text messages to fellow programmers. The ability to communicate within the application instead of using a third party software, will relieve users from the hassle of finding other means for communication.

Another enhancement would be to allow programmers to execute their code within the application so that they can receive feedback on the validity of their solutions.

Lastly, the addition of an experience point system to gamify the application may increase user retention. Currently, MeetCode does not have a rewards system to incentivize users to continue using the application as users can move to another platform anytime without feeling any attachment to MeetCode.

Reflections and Learning Points

A good strategy for deployment of a web application would be to have some basic functionality for each service (do unit tests, and see if they integrate properly by running the whole web application locally), then deploy on the appropriate cloud provider services. After deployment is successful, proceed to update the source code for each service according to all the FRs/NFRs that need to be met.

We came to the conclusion above because deployment on a cloud provider was one of the most difficult parts during development. It is very easy for the application to not run properly with all the network ports and container ports lying around for each service. Once connectivity between the different services is settled, we can continue to add to the codebase without having to worry as much regarding networking/connectivity.

We also learnt about the Importance of writing tests early in deployment. Before tests were written, before merging every PR, manual testing was conducted to make sure the changes made in each PR did not break any existing functionality. Writing tests and setting up continuous integration workflows would have reduced the manual testing workload significantly.

Citations

[1] Yablonski, J. (n.d.). *Jakob's law*. Laws of UX. Retrieved October 30, 2022, from https://lawsofux.com/jakobs-law/

[2] Yablonski, J. (n.d.). *Doherty threshold*. Laws of UX. Retrieved October 30, 2022, from https://lawsofux.com/doherty-threshold/

[3] Yablonski, J. (n.d.). *Aesthetic usability effect*. Laws of UX. Retrieved October 30, 2022, from https://lawsofux.com/aesthetic-usability-effect/