School of Computing

CS3219 Software Engineering Principles and Patterns

AY22/23 Semester 1

Project Report

Group 11

Team Members	Student No.	Email
Kumaran S/O Selvvaratnam	A0200132J	e0407113@u.nus.edu
Mohamed Fazil	A0201854L	e0415663@u.nus.edu
Chua Chen Ler	A0200025H	e0407006@u.nus.edu
A H M Thikhina Induwara Weragma Bakmeedeniya	A0221575H	e0556767@u.nus.edu

Table of Contents

1. Introduction

1.1 Background and Purpose of the Project

PeerPrep is a web application where students can help each other in technical interviews by attempting LeetCode-style algorithmic questions. They can play interviewer-interviewee roles and take turns solving a question while collaborating using a live Editor and Chat. The students can select their questions based on difficulty or topic, so they can practice on what they are weaker in. They can mark questions as done, and they can revisit the questions and revise them.

1.2 Individual Contributions to the Project

Kumaran	Question Service, History Service, Doing GitHub Actions and writing the workflows for the CI/CD pipeline
Fazil	User Service, Chat Service, Deployment of microservices on AWS, Setting up of CI/CD Pipeline for microservices using AWS CodePipeline
Chen Ler	Frontend
Thikhina	Matching Service, Collaboration Service and integration of these to frontend.

2. Functional Requirements

Label	Requirement	Priority
FR 1	**User Service**	
FR 1.1	The application should allow users to be able to login with valid credentials	High
FR 1.2	The application should allow users to be able to log out from the account	High
FR 1.3	The application should allow users to be able to reset password	Medium
FR 1.4	The application should allow users to be able to view account details under the profile page	Medium
FR 1.5	The application should allow users to delete their account	Medium
FR 2	**Matching Service**	
FR 2.1	The application is able to provide matching via difficulty level	High
FR 2.2	The application is able to provide matching via a topic	High
FR 2.3	The application is able to match 2 users based on the same filter	High
FR 2.4	The application will create a unique roomID for 2 users that are matched together	High
FR 2.5	The application is able to handle unsuccessful matching after 30 seconds	Medium
FR 3	**Room Service**	
FR 3.1	The application allows two matched users to collaborate on the editor	High
FR 3.2	The application will return either user to the home page if either user exits the session	High

FR 3.3	The application allows either user to end the room session	Medium
FR 4	**Question Service**	
FR 4.1	The application should be able to give two people in the same room the same question	High
FR 4.2	The application should be able to give question based on difficulty level	High
FR 4.3	The application should be able to give question by topic	High
FR 5	**Chat Service**	
FR 5.1	The application should allow two users in the room to message each other	High
FR 5.2	The application should notify the other user when the other joins the room	Medium
FR 5.3	The application should notify the other user when the other leaves the room	Medium
FR 6	**Collaboration Service**	
FR 6.1	The application should allow a user to code on the editor using the following languages: Python, C++, Java, JavaScript	High
FR 6.2	The application should show the same code in the editor to the other matched user	High
FR 6.3	The application should update the editor language to be the same for the matched editors	Medium
FR 6.4	The application should allow users to enable code auto-complete for their selected language.	Medium
FR 6.5	The application should allow users to choose a theme for their editor	Low

FR 7	**History Service**	
FR 7.1	The application allows the user to add a question he has completed to his history of completed questions	High
FR 7.2	The application allows the user to view all the questions he has attempted in the past	High
FR 7.3	The application allows the user to see the history of completed questions in chronological order	High
FR 7.4	The application allows the user to see when the question was last attempted if the user attempts the same question multiple times.	Medium

Table 1. Functional Requirements

3. Non-Functional Requirements

Label	Requirements	Priority
NFR 1	**Performance**	
NFR 1.1	The application should respond within less than 1 second when the user navigates through the PeerPrep	Medium
NFR 1.2	The application should reflect changes in real-time when a user modifies the text field in the editor	Medium
NFR 1.3	Chat in the Collaboration page should be responsive and messages should come within less than 2 seconds	High
NFR 2	**Security**	
NFR 2.1	The application should store user's passwords in a hashed and salted form	High
NFR 3	**Usability**	
NFR 3.1	The application should be supported on Chrome, Safari, Microsoft Edge and Firefox	High
NFR 4	**Availability**	
NFR 4.1	The application should be up and running 98% of the time	High
NFR 4.2	The application should allow at least 1000 users to sign up for PeerPrep	Medium
NFR 5	**User Interface**	
NFR 5.1	The application should allow users to quickly initiate a match from the home page	High
NFR 5.2	During matchmaking, there should be a countdown timer to let the user know that it is finding a match	Medium
NFR 5.3	The user interface should be minimalistic and straightforward to use	Low

Table 2. Non-Functional requirements

3.1 NFR Justification

NFR 1: Performance

NFR 1.1: A response time beyond 1 second is immediately noticeable for users, and would ruin the seamless experience using the PeerPrep. If it takes time to retrieve information from the backend, the User Interface will display a loading icon.

NFR 1.2: The editor should be responsive and update simultaneously for both users so that the interviewer can see how the interviewee codes in real-time, and can provide valuable input and feedback.

NFR 1.3: Communication is vital during a mock interview, where if there are communication issues, it will vastly affect the experience for the user.

NFR 2: Security

NFR 2.1: User's password should be stored in a hashed and salted form instead of plaintext so that in the event of a database breach, the attacker will still need to decipher the hashed password, which acts as another layer of protection and deterrence.

NFR 3: Usability

NFR 3.1: Since Safari, Chrome, Edge, and Firefox are amongst the most popularly used browsers amongst our target audience (university students), it is essential that our application is working on all three of them to ensure that the majority of our users can use the application.

NFR 4: Availability

NFR 4.1: The application should have minimal downtime to ensure users are given a seamless experience in using the application.

NFR 4.2: The application is intended for many users to use, and hence should be able to support at least 1000 users for a start.

NFR 5: User Interface

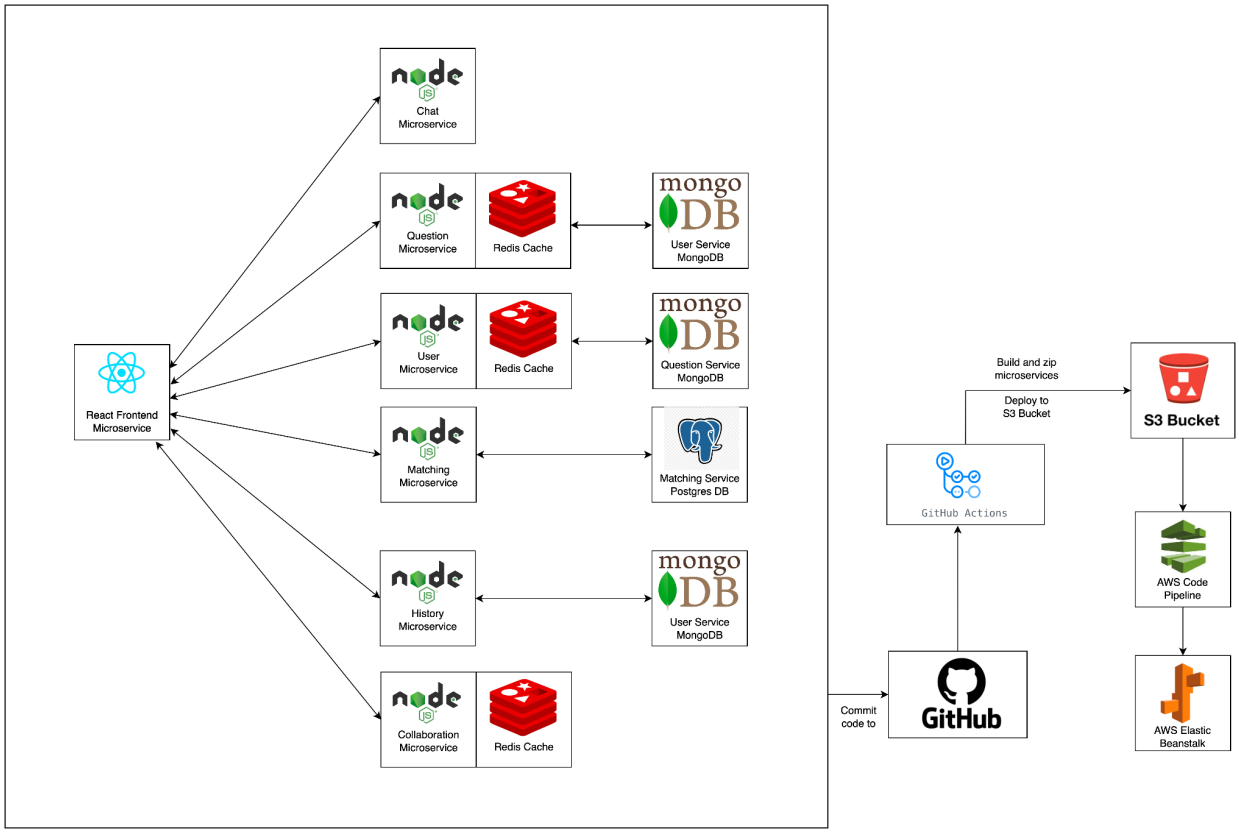
NFR 5.1: The main purpose of PeerPrep is to find a match and give each other a mock interview, thus having the button in the home page to quickly select difficulty and finding a match based on that is crucial in saving the user's time.

NFR 5.2: A dynamic interface will enhance the user experience by giving visual cues and letting the user know that PeerPrep is running in the background to find a match.

NFR 5.3: A minimalistic interface makes it easy for new users to get used to and navigate around.

4. Architectural Design

4.1 Architecture Diagram



4.2 Architecture Decisions

4.2.1 Monolith vs Microservice Architecture

	Microservice Architecture	Monolith Architecture
Benefits	• Easier to scale application when different services have varying loads.	• Easier to set up and deploy

	• Easier for different people to work on the application simultaneously • Easier to decouple different services	
Downside	• Application might get too complicated if too many microservices • Harder to deploy	• Harder to work concurrently for different team members.

Decision: Microservice Architecture

Since we had the initial idea of deploying the application, we decided that microservice architecture would be better apt for deployment as opposed to a monolith. As each service will be deployed independently, it will ensure other microservices will not go down if one microservice is down. On the other hand, in a monolith architecture, as long as one component fails, the entire application goes down.

Additionally, with a microservice architecture, it will be easier for all of us to work on the individual services concurrently, which allows us to have faster progress. Using a monolith architecture would result in slower updates as we would have to wait for each other's code to be pushed in order to incorporate ours.

4.2.2 Kubernetes vs AWS Beanstalk

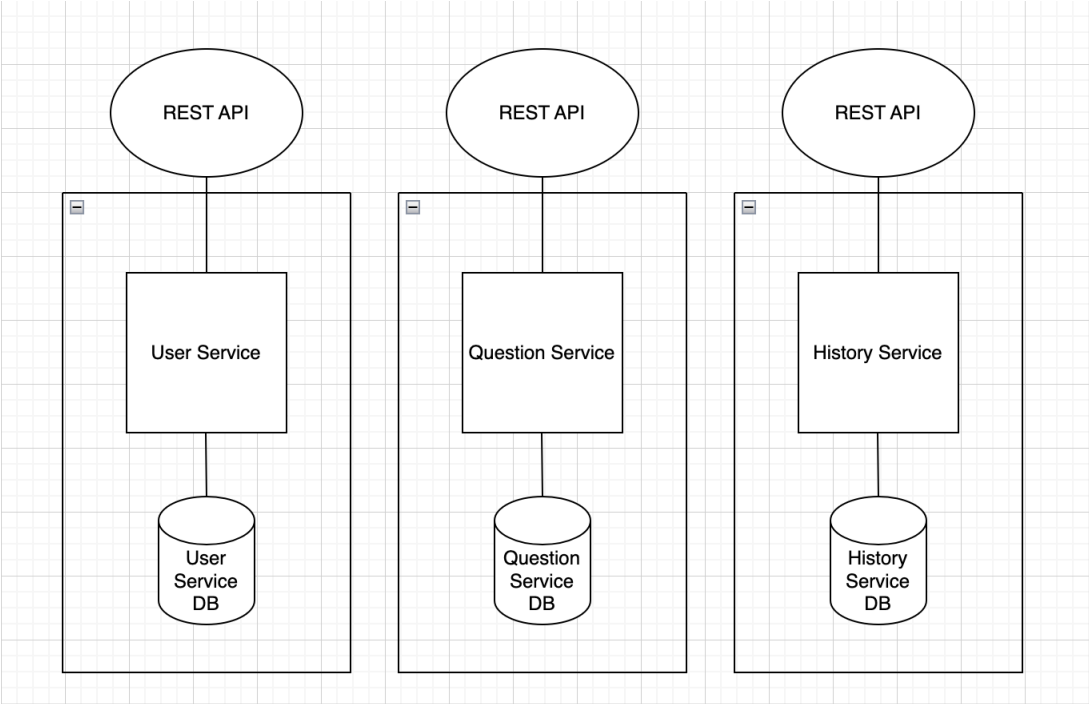
Decision: AWS Beanstalk

Although AWS Beanstalk requires us to pay to use, as beginners, it is easier to deploy the application on AWS Beanstalk as opposed to using Kubernetes. Most of us used AWS Beanstalk for our Task B2.2, which has a simpler learning curve. During the lecture on Kubernetes, our team also faced some difficulties working with Kubernetes. This led to us wanting to use AWS Beanstalk as it is easier to manage the project, with all available on one site.

Additionally, with Elastic Beanstalk, all we have to do is upload a zip of our code and it will automatically handle the deployment, from capacity provisioning, load balancing, and autoscaling to making health checks for the application. As such, we do not have to waste time worrying about configuring the servers and the deployment. Instead, we can focus on improving our code.

5. Design Patterns

5.1 Database-per-Service Pattern



Our microservices architecture followed the Database-per-Service pattern where each individual microservice had its own database. Each service's database is not accessed by the other microservices and each service's persistent data can only be accessed via their REST APIs.

This design decision was made to ensure loose coupling of microservices and to achieve the following benefits:

1. Scalability
    - Data schema changes can be made quickly without impacting other microservices.
    - Each database can scale independently.

2. Reliability
    - The failure of a database will not affect other services.

5.2. Pub-Sub Pattern

Our chat, collaboration, and matching services demonstrate the use of pub-sub pattern through their use of Socket.io for bi-directional communication. For example, the editor component and chat components in the front end make use of client sockets to emit events that are subscribed to by the socket server, and in return consume messages that are published by the socket server.

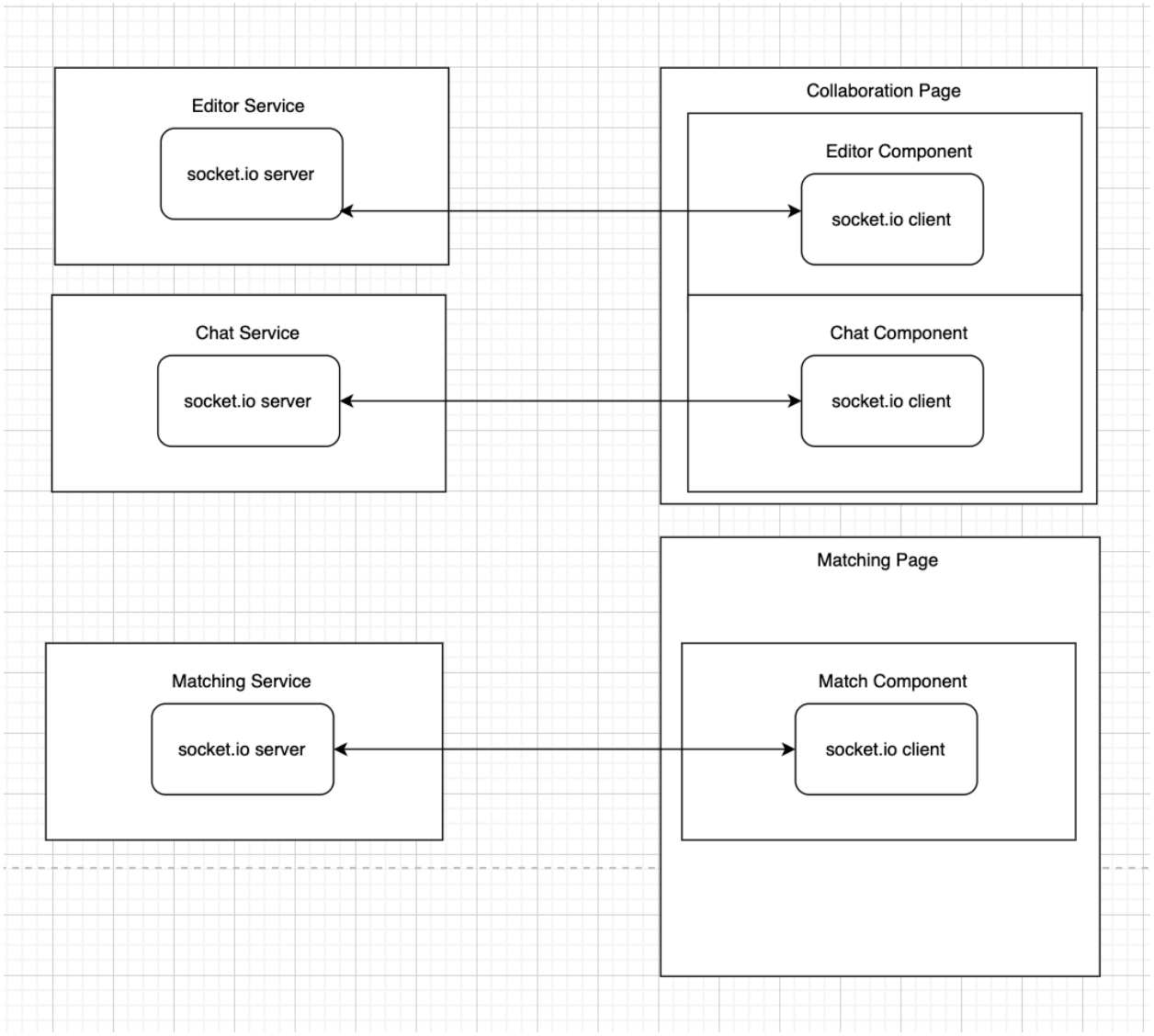
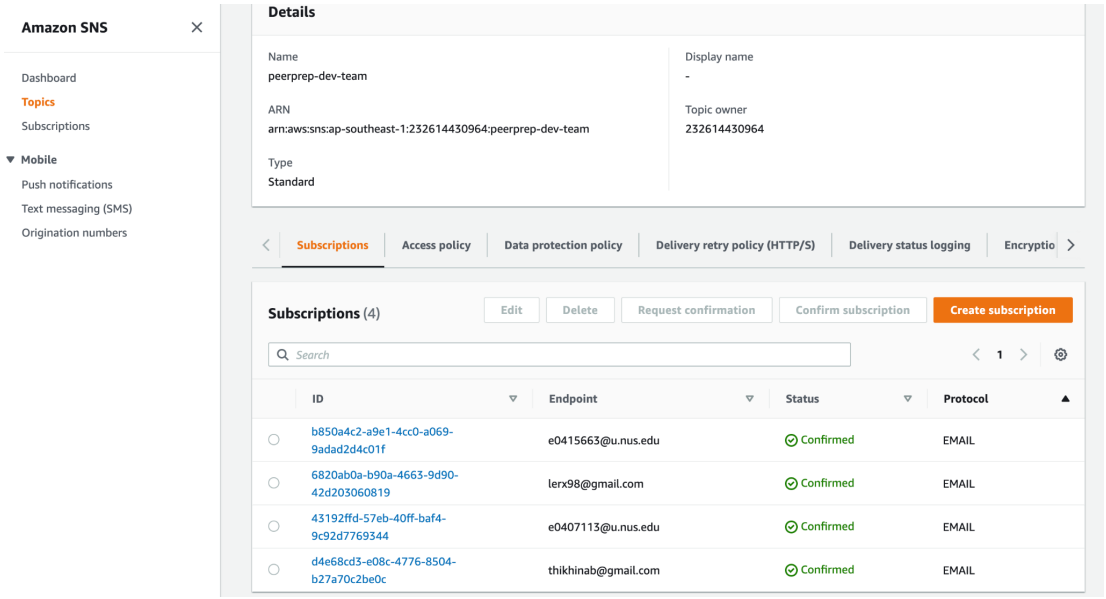


Image: Pub-Sub Pattern Used by PeerPrep

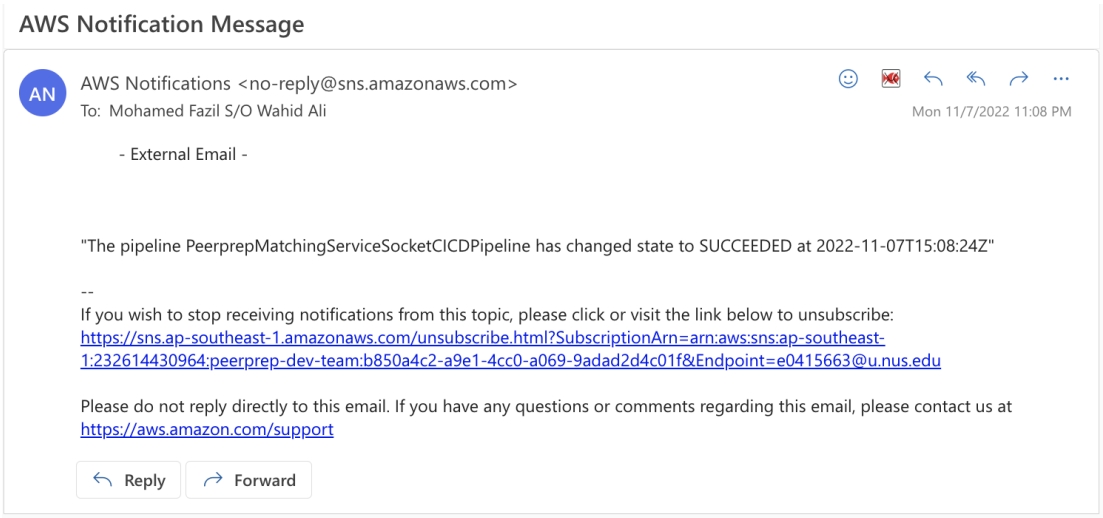
The use of a pub-sub pattern for matching/collaboration/editor services was employed to achieve the following benefits:

1. Faster response time and reduced latency as periodic polling is eliminated and updates can be received asynchronously.
2. Publishers and subscribers are decoupled and can emit events without having to know about the receivers.

In addition, the use of a pub-sub pattern was employed to receive notifications regarding the state of our CICD pipelines.



Using AWS Simple Notification Service, each member of the team subscribed to the peerprep-dev-team topic in order to receive updates regarding our pipeline state. As subscribers of the topic, we received emails whenever our CICD pipelines changed state as follows:



5.3 Model-View-Controller Pattern

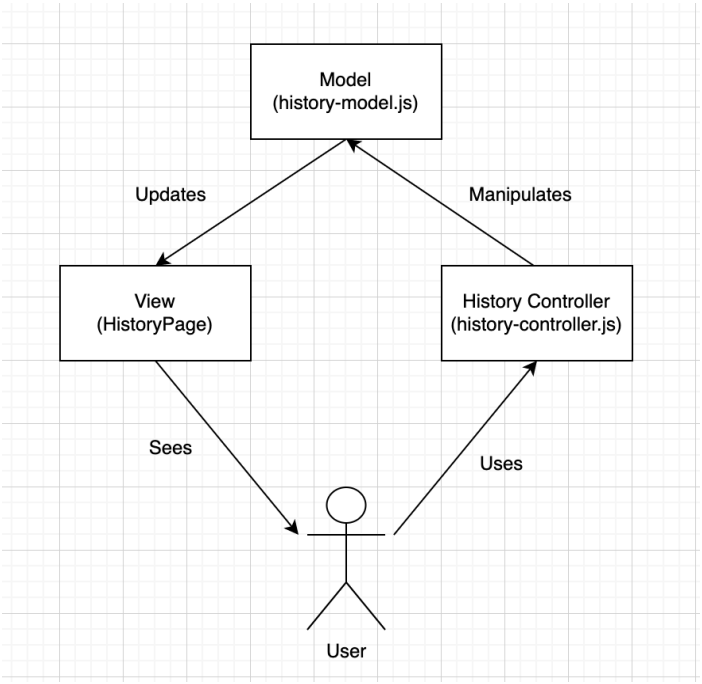
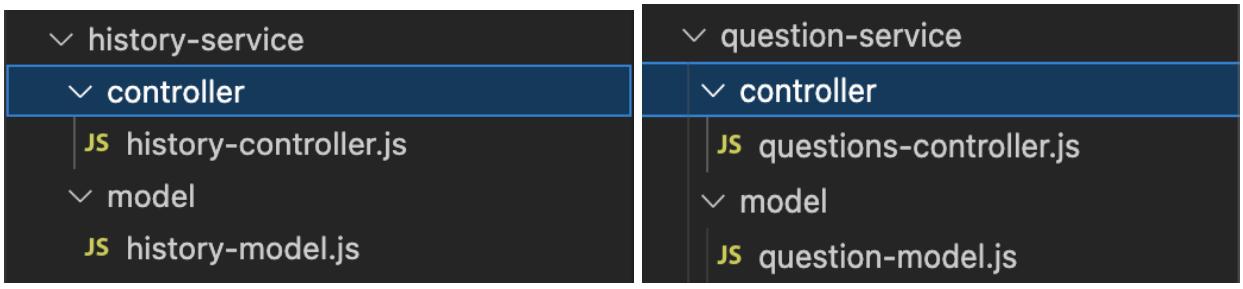


Image: Demonstration of MVC pattern for history-service

The MVC pattern was used extensively in our architecture, namely in question/history/user microservices. The Model layer contains only application data and its schema and does not contain logic describing how the data is presented to the user. The View layer which is implemented in the front end presents the model's data to the user. The Controller layer exists between the view and the model and executes the appropriate manipulation of the model in reaction to events executed in the view layer by the user.



This design pattern allows us to practice separation of concerns and increase modularity in our microservices.

5.4 Decompose By Subdomain Pattern

Our team employed the use of this decomposition pattern to define microservices in line with Domain-Driven Design(DDD). The following subdomains were identified from the problem space:

1. User Management
2. Collaboration Management
3. Match Management
4. Questions Management
5. History Management

The identification of these subdomains led to the development of the following microservices:

1. User service
2. Collaboration service
3. Chat service
4. Matching service
5. Question service
6. History service

5.5 Blue-Green Deployment Pattern

The Blue-Green deployment pattern was used to reduce downtime and ensure the high availability of our backend services. Whenever a change is to be pushed to production, a clone of the current environment is created and URLs are swapped. This allows the green environment to serve traffic while the latest version is being deployed to the original blue environment. Upon successful deployment of the blue environment, the URLs are swapped again.

This deployment pattern was followed to achieve the following benefits:

- Seamless user experience and better usability for our users
- Instant rollbacks to a stable state when the latest change fails

6. DevOps

6.1 Development Process

Agile development process was followed to develop the application iteratively in a short span of time. Following an agile development process allowed our team to be flexible to changes and consistently improve the product based on a continual feedback loop. Specifically, our team utilized Kanban methodologies mainly to implement the agile development process. With the use of Kanban methodologies, our team followed a continual development process throughout the semester and was flexible with regard to tasks and timings. Tasks were reprioritized, reassigned, or updated as needed. However, some elements of scrum methodologies were also adopted such as feature-driven development and imposing time limits on crucial deliverables.

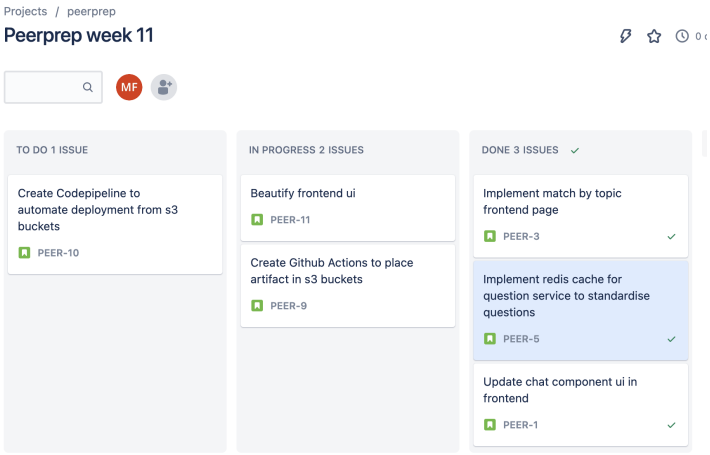


Image: Jira Kanban board utilized by our team that was cleared weekly

6.2 CI/CD

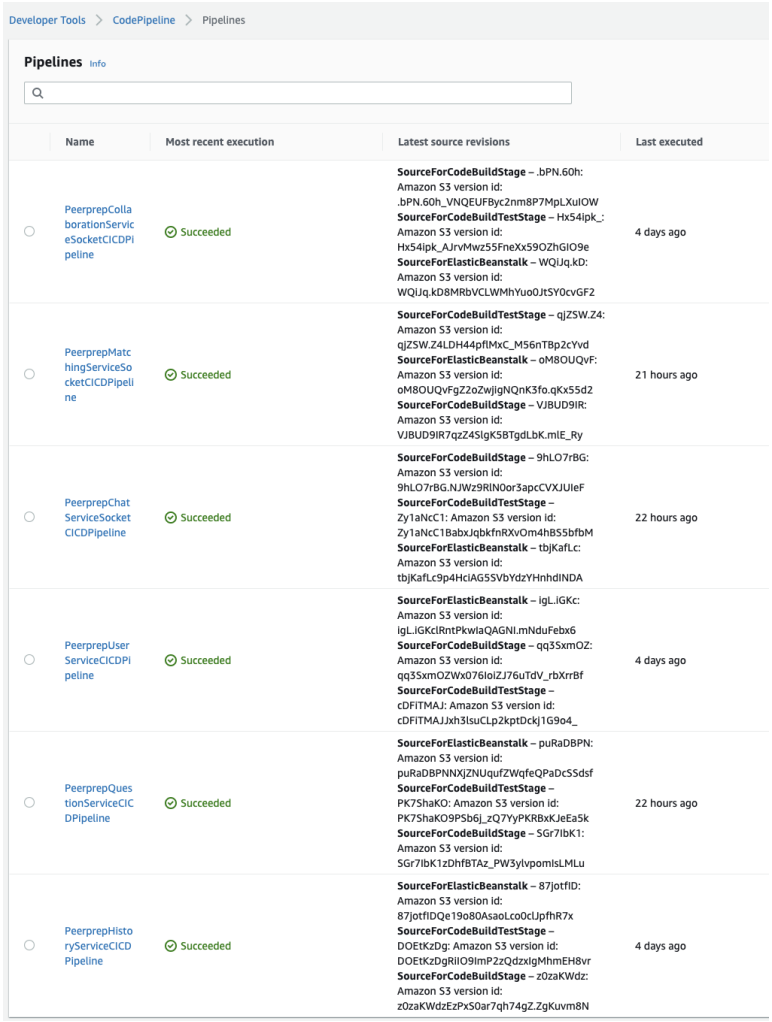


Image: CICD pipelines for each microservice implemented using AWS CodePipeline

Our team set up individual CI/CD Pipelines using AWS CodePipeline for each microservice to perform Blue-Green Deployment on AWS Elastic Beanstalk. Blue-green deployment strategy was adopted as it helps to increase availability and reduce risk. The CI/CD pipeline architecture creates a clone (green) of the live Elastic Beanstalk environment (blue). The pipeline then swaps the URLs between the two environments. While CodePipeline deploys application code to the original environment and testing and maintenance take place, the temporary clone environment handles the live traffic. Suppose deployment to the blue environment fails because of issues with the application code, the green environment serves the live traffic, and there is no downtime. Once deployment to the blue environment is successful, and code review and code testing are completed, the pipeline once again swaps the URLs between the green and blue environments. The blue environment starts serving the live traffic again, and the pipeline terminates the temporarily created green environment.

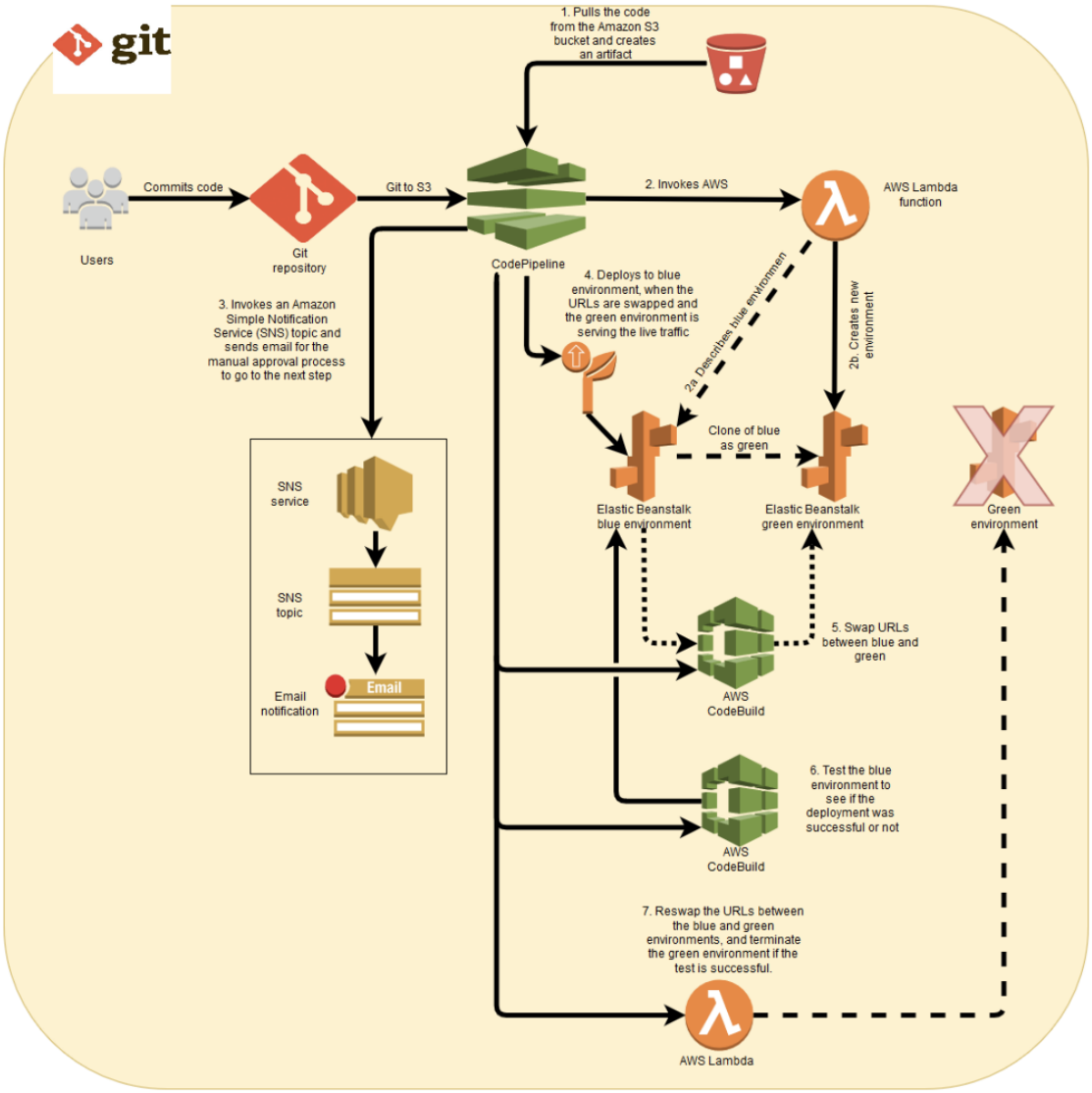


Image: Architecture of our CICD Pipeline

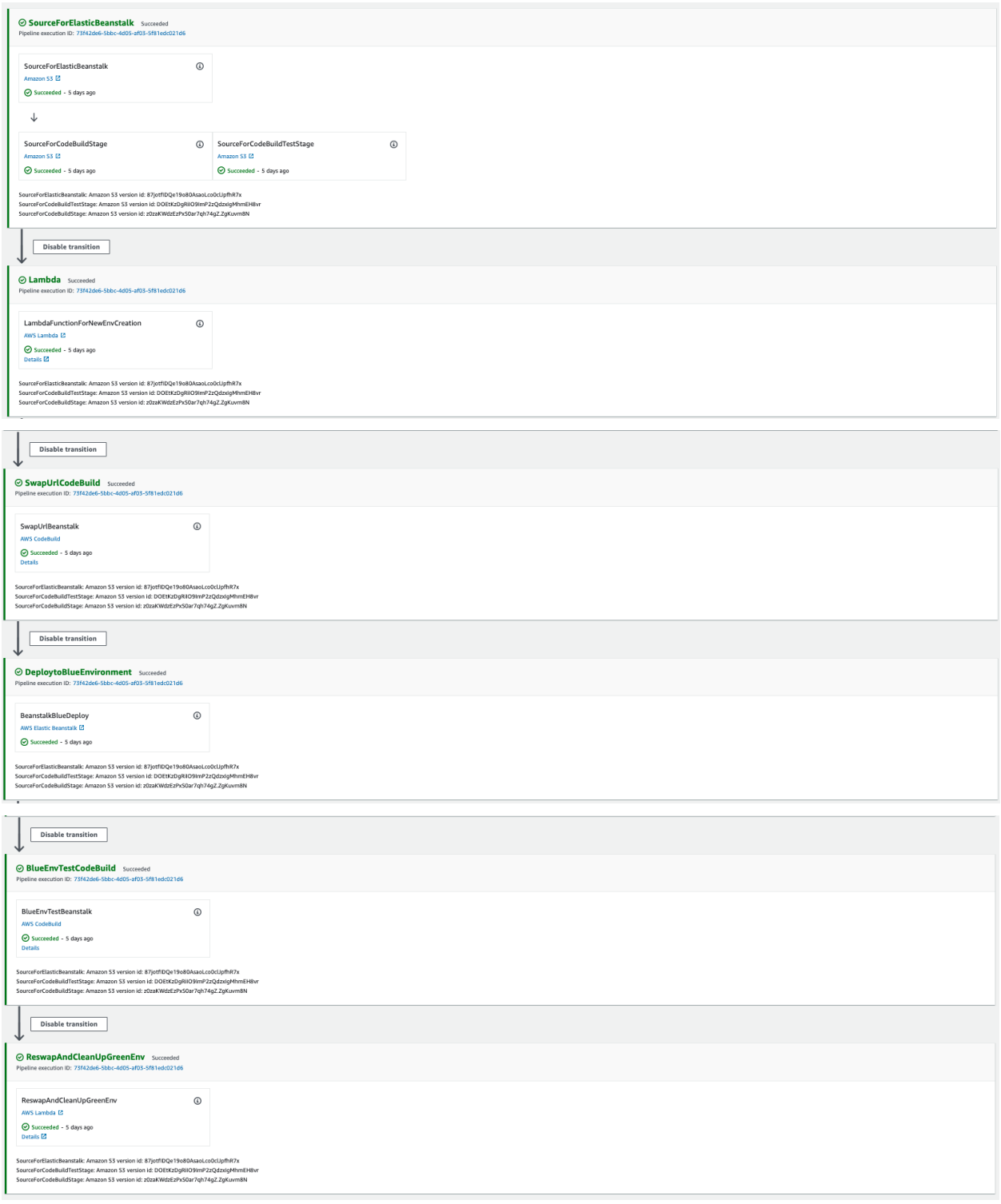


Image: Complete AWS CodePipeline stages

A detailed step-by-step execution of our CICD pipeline will be as follows:

1. Upon merging of a pull request to the main branch in our GitHub repository, we have created 6 Github Actions workflows that are triggered if there are code changes in the respective service directory.

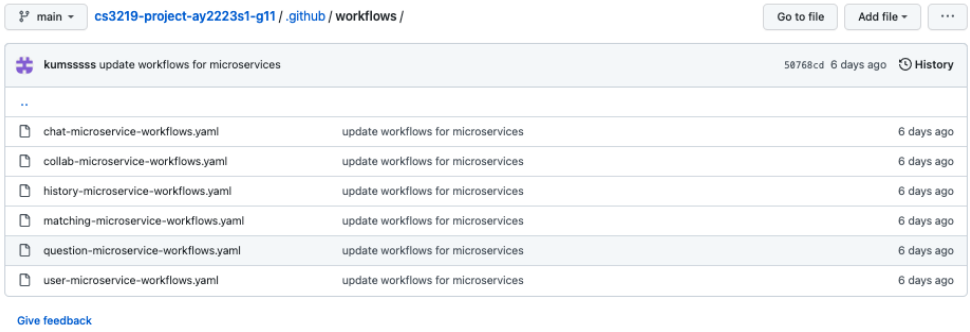


Image: GitHub Actions workflows for each microservice

Each workflow builds the respective microservice and places the zipped artifact file into its corresponding AWS S3 buckets for deployment to AWS Elastic Beanstalk by AWS CodePipeline.

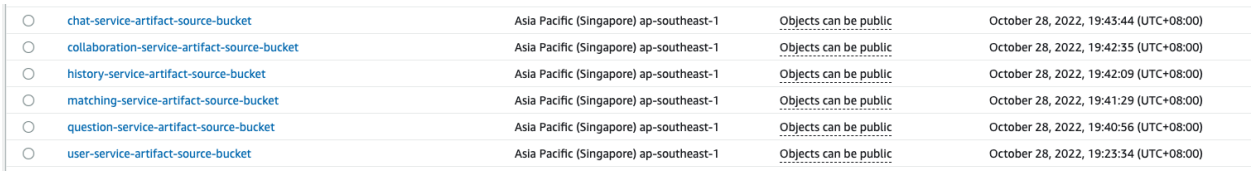


Image: S3 buckets for each microservice artifact

2. Upon detection of change in the bucket, the first stage of the AWS CodePipeline pipeline pulls the artifact from the source bucket and provides it for deployment to the Elastic Beanstalk environment.

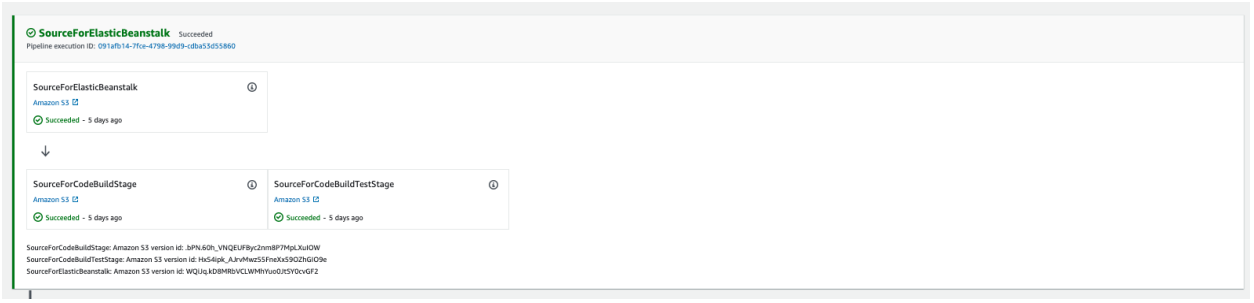


Image: First stage of the AWS CodePipeline which checks for artifact files in S3 bucket

3. The second stage triggers a Lambda function that clones the blue environment, which results in a green environment.

○	Peerprephistoryservice-env	Ok	peerprep-history-service	2022-11-01 22:31:37 UTC+0800	2022-11-03 13:19:33 UTC+0800	Peerprephistoryservice-env.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com	code-pipeline-1667315744637-Adl4fe1vigZZYwl1soPkjCGCK5DwQZui	Node.js 16 running on 64bit Amazon Linux 2
○	Peerprephistoryservice-env-green	Ok	peerprep-history-service	2022-11-04 10:23:09 UTC+0800	2022-11-04 10:25:58 UTC+0800	Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com	code-pipeline-1667315744637-Adl4fe1vigZZYwl1soPkjCGCK5DwQZui	Node.js 16 running on 64bit Amazon Linux 2

Image: Green environment of history-service created during execution of pipeline

```
def handler(event, context):
    timer = threading.Timer((context.get_remaining_time_in_millis() / 1000.00) - 0.5, timeout, args=[event, context])
    timer.start()
    try:
        # Extract the Job ID
        job_id = event['CodePipeline.job']['id']
        # Extract the Job Data
        job_data = event['CodePipeline.job']['data']
        user_parameters = job_data['actionConfiguration']['configuration']['UserParameters']
        print(job_data)
        print(event)
        BlueEnvInfo=GetBlueEnvInfo(EnvName=(json.loads(user_parameters)['BlueEnvName']))
        BlueEnvId=(BlueEnvInfo['Environments'][0]['EnvironmentId'])
        BlueVersionLabel=(BlueEnvInfo['Environments'][0]['VersionLabel'])

        #Calling CreateConfigTemplate API
        ConfigTemplate=CreateConfigTemplateBlue(AppName=(json.loads(user_parameters)['BeanstalkAppName']),BlueEnvId=BlueEnvId,TempName=json.loads(user_parameters)['CreateConfigTempName'])
        ReturnedTempName=ConfigTemplate
        print (ReturnedTempName)
        if not ReturnedTempName:
          #raise Exception if the Config file does not exist
          raise Exception("There were some issue while creating a Configuration Template from the Blue Environment")
        else:
          GreenEnvId=CreateGreenEnvironment(EnvName=(json.loads(user_parameters)['GreenEnvName']),ConfigTemplate=ReturnedTempName,AppVersion=BlueVersionLabel,AppName=(json.loads(user_parameters)['BeanstalkAppName']))
          print (GreenEnvId)
          #print (GreenEnvIddetails)
          if GreenEnvId:
              Status="Success"
              Message="Successfully created the Green Environment/Environment with the provided name already exists"
```

Image: AWS Lambda code segment to create a clone of the blue environment

4. The third stage swaps the URLs between the blue and the green environments using a CodeBuild project. Once this is complete, the green environment serves the live traffic.

○	Peerprephistoryservice-env	Ok	peerprep-history-service	2022-11-01 22:31:37 UTC+0800	2022-11-04 10:29:29 UTC+0800	Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com	code-pipeline-1667528927054-87jotfIDQe19o80AsaoLco0cUpfhR7x	Node.js 16 running on 64bit Amazon Linux 2
○	Peerprephistoryservice-env-green	Ok	peerprep-history-service	2022-11-04 10:23:09 UTC+0800	2022-11-04 10:28:20 UTC+0800	Peerprephistoryservice-env.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com	code-pipeline-1667315744637-Adl4fe1vigZZYwl1soPkjCGCK5DwQZui	Node.js 16 running on 64bit Amazon Linux 2

Image: Swapped URL of green and blue environments during pipeline execution

```
[Container] 2022/11/04 02:27:48 Running command ./swapenvironments.sh
{
    "AcceptRanges": "bytes",
    "LastModified": "Tue, 01 Nov 2022 15:11:33 GMT",
    "ContentLength": 93,
    "ETag": "\"6174a3be4714df9a8efbdc0bdfe9bb2b\"",
    "VersionId": "WQehNEueSRYuegoI9uIfXh8eRoMUV8.i",
    "ContentType": "binary/octet-stream",
    "Metadata": {}
}
Peerprephistoryservice-env.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com
Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com
Ready

Urls are swapped now

[Container] 2022/11/04 02:28:14 Phase complete: INSTALL State: SUCCEEDED
[Container] 2022/11/04 02:28:14 Phase context status code:  Message:
[Container] 2022/11/04 02:28:14 Entering phase PRE_BUILD
[Container] 2022/11/04 02:28:14 Phase complete: PRE_BUILD State: SUCCEEDED
[Container] 2022/11/04 02:28:14 Phase context status code:  Message:
[Container] 2022/11/04 02:28:14 Entering phase BUILD
[Container] 2022/11/04 02:28:14 Running command

[Container] 2022/11/04 02:28:14 Phase complete: BUILD State: SUCCEEDED
[Container] 2022/11/04 02:28:14 Phase context status code:  Message:
[Container] 2022/11/04 02:28:14 Entering phase POST_BUILD
[Container] 2022/11/04 02:28:14 Phase complete: POST_BUILD State: SUCCEEDED
[Container] 2022/11/04 02:28:14 Phase context status code:  Message:
```

Image: Logs of CodeBuild project that swaps URL of blue and green environment

5. The fifth stage performs the deployment to the blue environment.

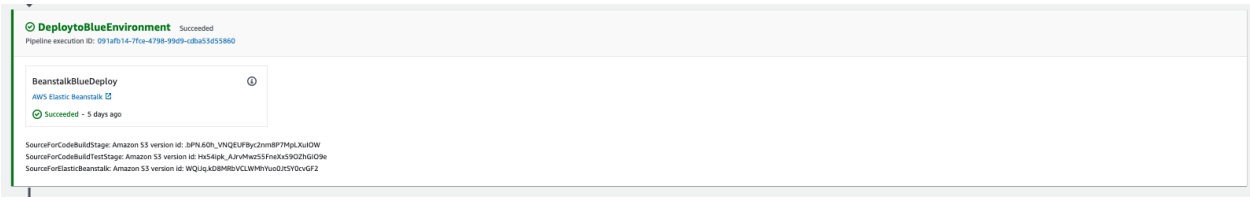


Image: Fifth stage of the AWS CodePipeline which deploys the latest version on elastic beanstalk blue environment

6. If the deployment is successful, the sixth stage triggers a test on the blue environment to see if it can access a 200 OK in the response. If the response is other than 200 OK, the pipeline doesn't proceed and marks this stage as failed.

```
[Container] 2022/11/04 02:32:16 Running command ./testwebsitecodebuild.sh
+ echo Peerprephistoryservice-env
Peerprephistoryservice-env
+ aws elasticbeanstalk describe-environments --environment-names Peerprephistoryservice-env --region ap-southeast-1 --query Environments[0].CNAME --output text
+ cname=Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com
+ echo Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com
Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com
+ [ Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com ]
+ curl -LI http://Peerprephistoryservice-env-green.eba-jgstrgm9.ap-southeast-1.elasticbeanstalk.com -o /dev/null -w %{http_code}\n -s
+ status=200
+ echo 200
200
+ echo Peerprephistoryservice-env
Peerprephistoryservice-env
+ [ 200 = 200 ]
+ exit 0

[Container] 2022/11/04 02:32:30 Phase complete: INSTALL State: SUCCEEDED
[Container] 2022/11/04 02:32:30 Phase context status code:  Message:
[Container] 2022/11/04 02:32:30 Entering phase PRE_BUILD
[Container] 2022/11/04 02:32:30 Phase complete: PRE_BUILD State: SUCCEEDED
[Container] 2022/11/04 02:32:30 Phase context status code:  Message:
[Container] 2022/11/04 02:32:30 Entering phase BUILD
[Container] 2022/11/04 02:32:30 Running command

[Container] 2022/11/04 02:32:30 Phase complete: BUILD State: SUCCEEDED
[Container] 2022/11/04 02:32:30 Phase context status code:  Message:
[Container] 2022/11/04 02:32:30 Entering phase POST_BUILD
[Container] 2022/11/04 02:32:30 Phase complete: POST_BUILD State: SUCCEEDED
[Container] 2022/11/04 02:32:30 Phase context status code:  Message:
```

Image: Sixth stage of the AWS CodePipeline which checks if Blue Environment can serve traffic

7. If the test stage is successful, another Lambda function is triggered in the seventh stage, which again swaps the URLs between the blue and green environments and then terminates the green environment. The blue environment gets back its initial Elastic Beanstalk CNAME that it initially had for serving the live traffic.

7. Application Design

7.1 Tech Stack

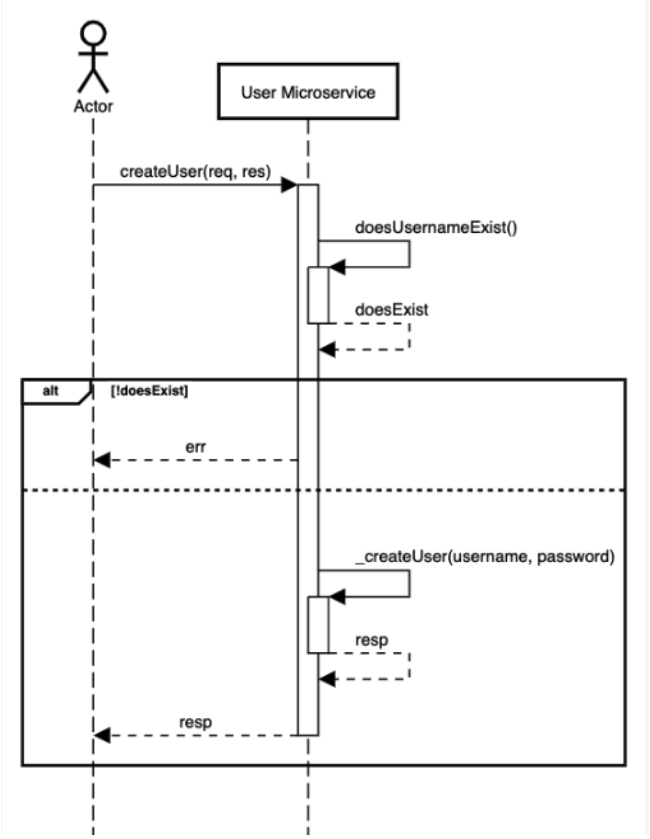
Type	**Used Technologies**
Frontend	React, Material UI
Backend	Express.js, mongoose, ioredis
Database	MongoDB, PostgeSQL, Redis
Deployment	AWS Elastic Beanstalk
Pub-Sub Messaging	Socket.io
Cloud Providers	AWS, Mongo Atlas, RedisCloud
CI/CD	GitHub Actions, AWS CodePipeline
Project Management Tools	GitHub Issues, Jira Kanban Board

8. Backend

8.1 User service

The purpose of the user service is to provide APIs for features such as creating an account, logging in and out of an account, changing account password and deleting an account. It also provides a token when a successful user login is done.

Sequence diagram for User Service SignUp:



There are a number of APIs under the user service:

1. POST / - takes in username and password. Creates an account with that username and password if the username does not exist. Returns status code 201 if successful.
2. POST /login - takes in a username and password. Checks if the username exists and whether the password is the correct one. Returns status code 200 and provides a session token if successful. Otherwise, return code 400.
3. POST /token - takes in a token. Checks if the token is valid. Returns status code 200 if valid, otherwise return code 401.
4. POST /logout - takes in the session token. Checks if the token exists and is valid. Returns status code 200 and logs the user out if token is valid

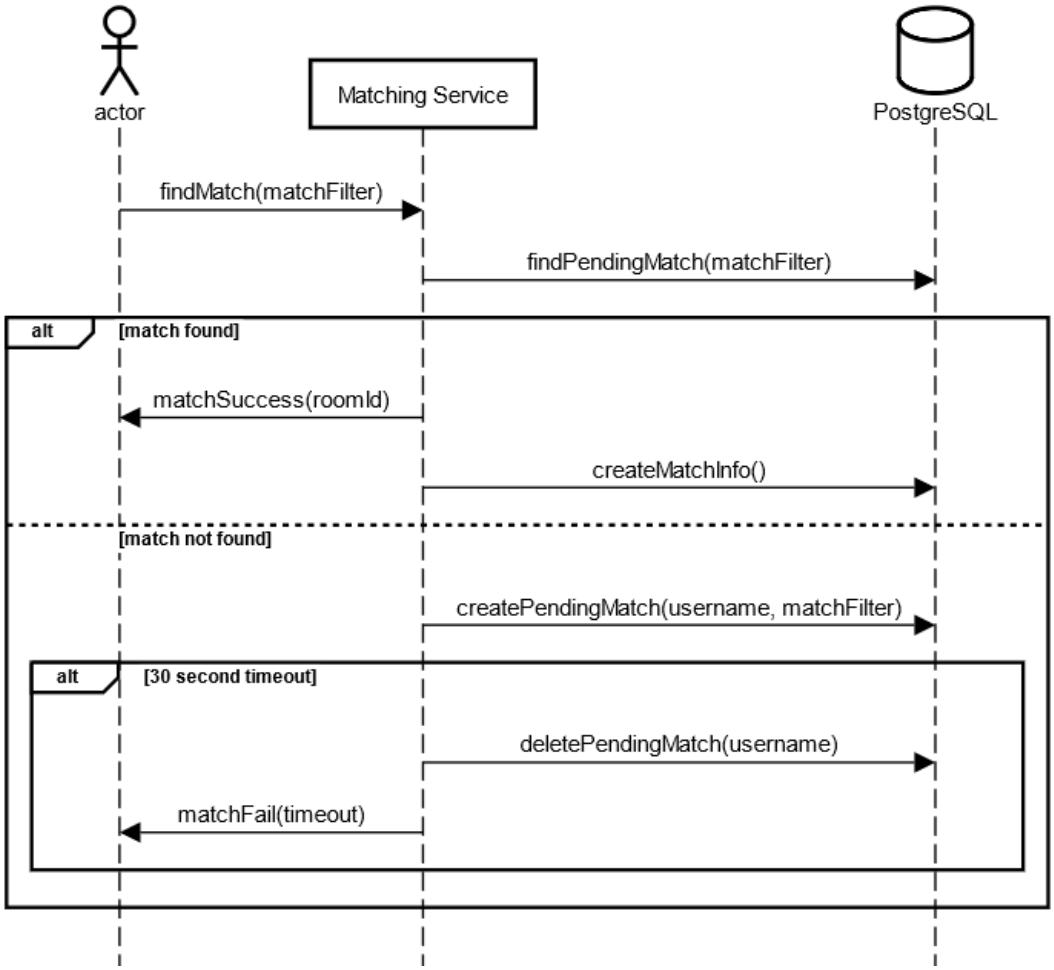
5. POST /change - takes in username, current password and new password. Change the password if both old and new are different and return status code 200 if successful.
6. POST /delete - takes in username and token. Deletes the account if the token is valid.

8.2 Matching service

The responsibility of the Matching service is to match two users based on a requested criteria. The latest version of the application has two match criteria: difficulty level and question topics. For the matching service, we have used Socket.io. The main motivation for using the pub-sub:

- Connection efficiency (Single TCP connection required throughout the lifecycle of the socket)
- The nature of the event-based communication of the matching service

The following sequence diagram below shows how the user and service interact with each other:



We have settled on using Postgres for the following advantages:

- Ability to do transactions especially since ACID properties are valued.
- The records are highly structured and don't change frequently.

PendingMatch Model has been created in a manner that it is extendable to any criteria, thus there is loose coupling. For example app in the future can decide to add more matching criteria/filters (eg. each represented through unique code) and the matching service is extensible to it.

Pending Match Model:

```
/**
 * Stores the users which are pending to match.
 * Primary key is the username.
 */
export const PendingMatch = sequelize.define("PendingMatch", {
    username: {
        type: DataTypes.STRING,
        allowNull: false,
        primaryKey: true,
    },
    socketId: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true,
    },
    filterKey: {
        type: DataTypes.STRING,
        allowNull: false,
    },
});
```

MatchInfo model is used for logging all matches created. This would help developers debug when necessary.
Please note that createdAt and updatedAt properties are initialized by default
MatchInfo Model

```
/**
 * Stores the information about matches made.
 * Property: usernameOne < usernameTwo
 */
export const MatchInfo = sequelize.define("MatchInfo", {
    usernameOne: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    usernameTwo: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    filterKey: {
        type: DataTypes.STRING,
        allowNull: false,
    },
});
```

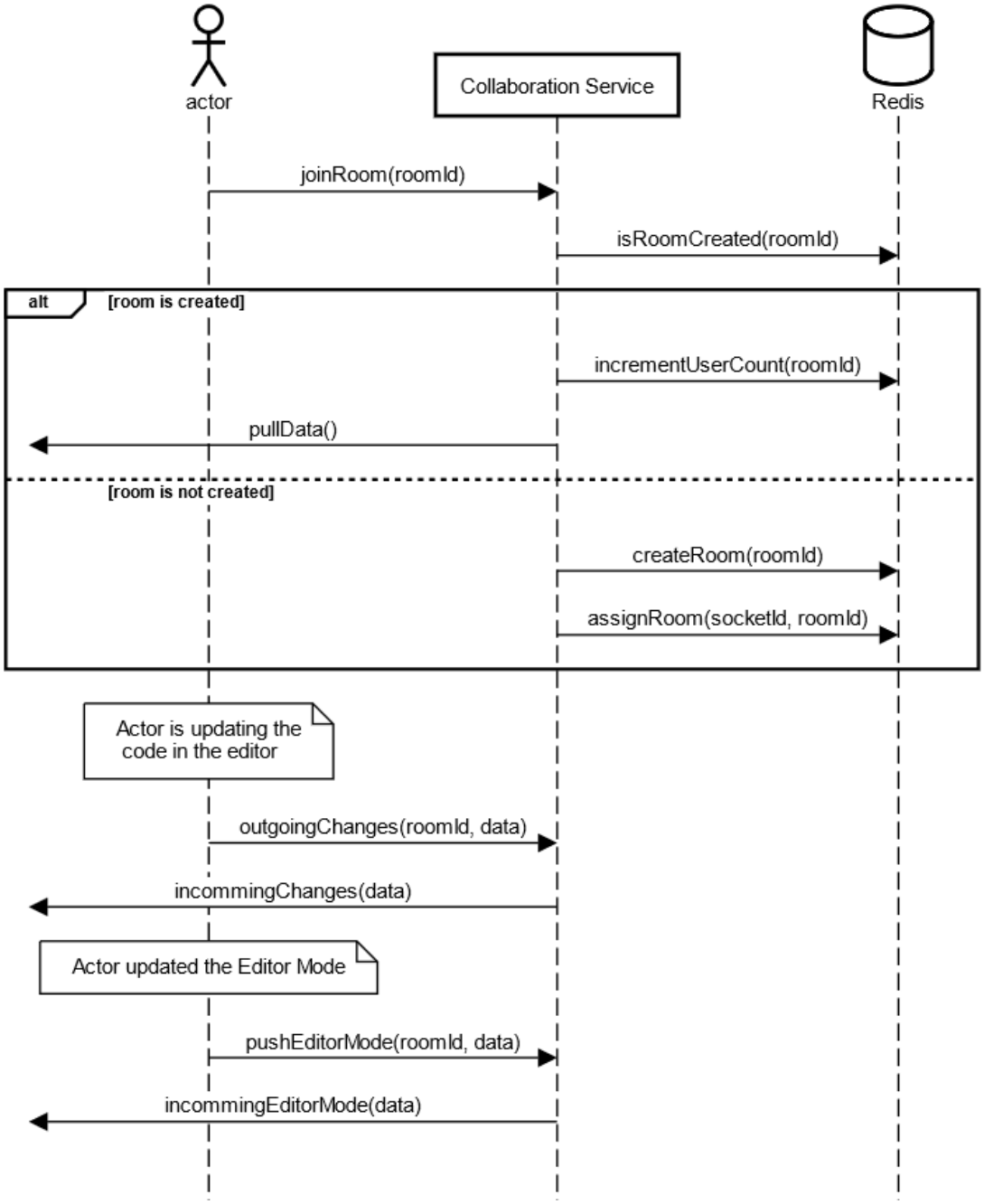
8.3 Collaboration service

The collaboration service is responsible for real time sharing the state of the editor between users who are matched. We have integrated the pub sub messaging pattern using Socket.io. The motivation behind this choice:

- Enable real time communication by reducing the need for polling (polling is used only if the web socket fails)
- Leverage the concept of Socket rooms (Makes the collaboration extensible when developers decide to extend collaboration between more than one user).

Furthermore, a Redis database is used to keep track of the room IDs and corresponding socket IDs. The primary reason for using Redis is low data access latency and high throughput.

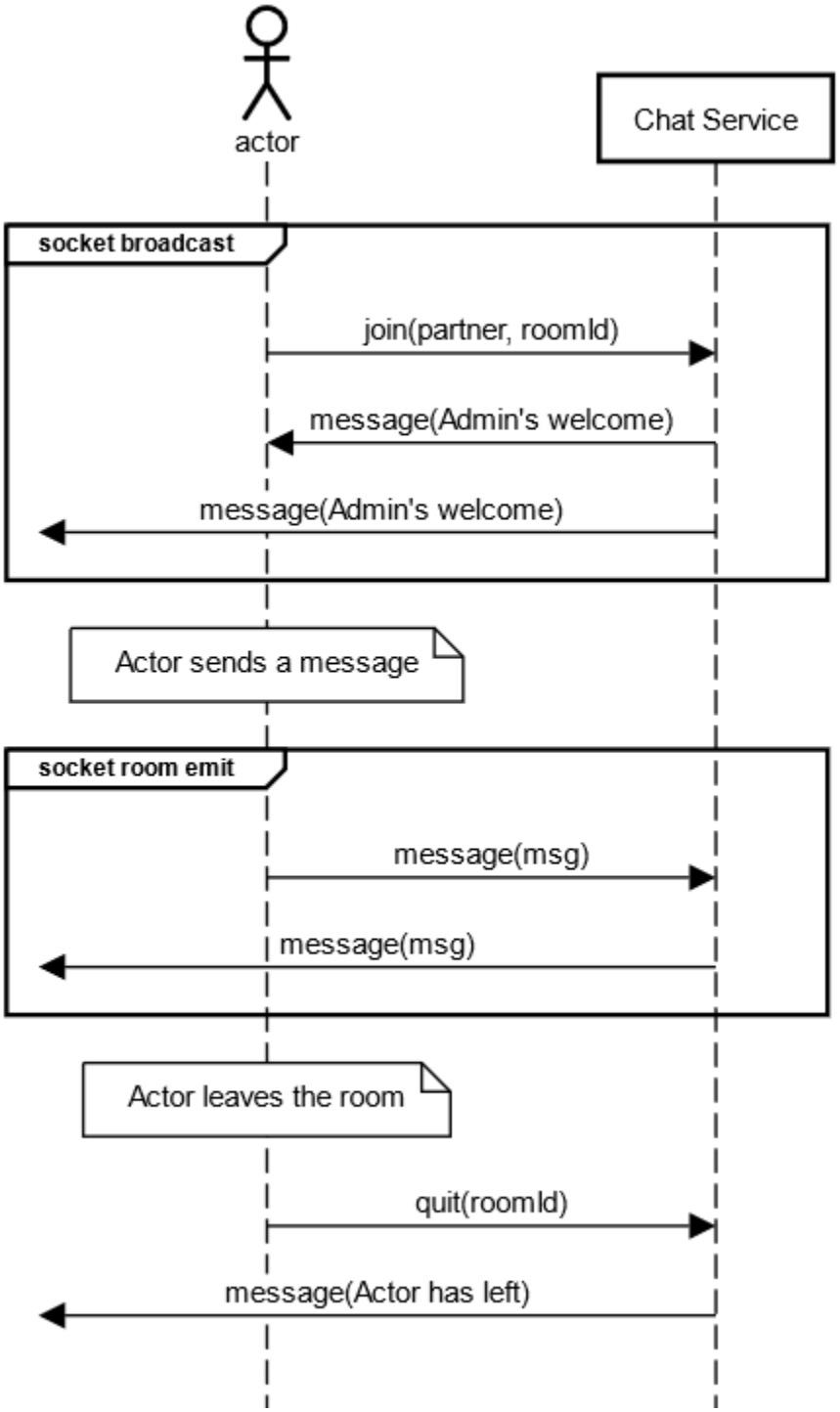
The following sequence diagram shows the interaction between a user and the collaboration service:



8.4 Chat Service

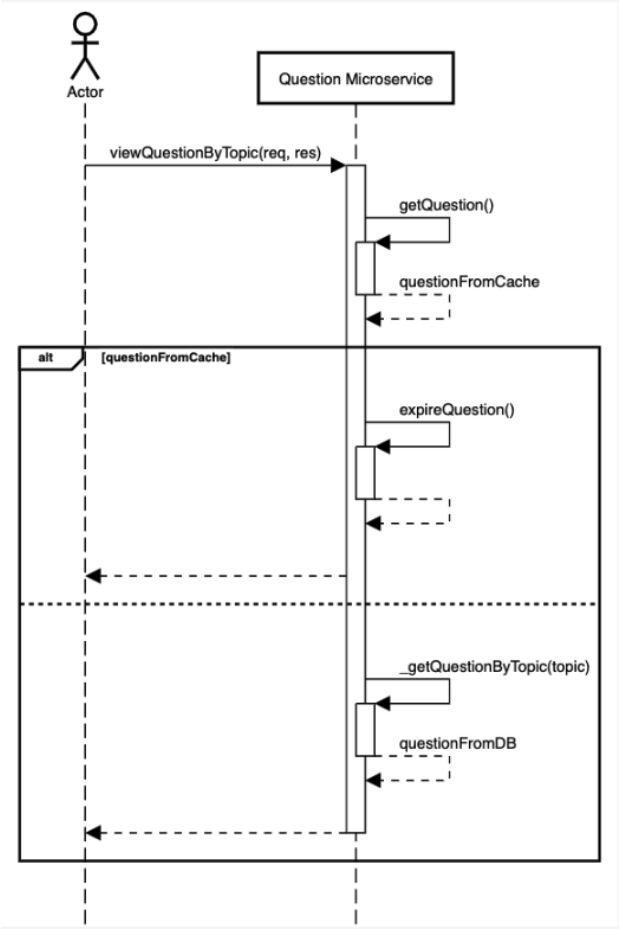
The Chat service enables communication between users who are matched. Similar to the collaboration service, the important facet is real time interaction between the clients. Therefore we settled with Socket.io. Moreover the nature of the chat communications being based on the events triggered and listened confirmed that pub sub messaging is the most appropriate.

The following diagram shows the interaction between a user and collaboration service:



8.5 Question Microservice

The purpose of the Question Microservice is to provide APIs to provide different types of random questions, either by difficulty or by topic. A redis mutex is used to ensure 2 people matched into the same room receive the same question. The first request will retrieve a question and store into the redis cache, which will be retrieved by the second request from the same room. Additionally, a random seed is used to ensure multiple API calls to the same endpoint return different questions.



List of APIs

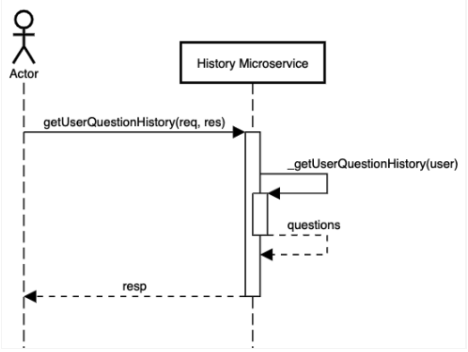
1. GET /questions/difficulty/:difficulty - Checks if a room name is provided. Check the redis cache to see if the room is assigned a question. Returns the question if it exists. Otherwise, query the database to return a question matching the difficulty mentioned in the params.Stores the question to the redis cache for the other member in the room to retrieve.
2. GET /questions/topics - Returns a list of unique topics of all the questions in the MongoDB.

3. GET /questions/topics/:topic - Checks if a room name is provided. Check the redis cache to see if the room is assigned a question. Returns the question if it exists. Otherwise, query the database to return a question matching the topic mentioned in the params. Stores the question to the redis cache for the other member in the room to retrieve.

8.6 History Microservice

The history service is used for user's to view questions they have attempted in the past. In the room page, there is a button to mark a question as done which will trigger an API call to the backend of the History Microservice to add the question they just attempted into the MongoDB. This history of questions can later be retrieved at the history page, where users can view titles of the questions they did, the difficulty, when it was last attempted as well as the question itself.

Sequence Diagram for History Service



List of APIs

1. GET /history/:user - Returns a list of questions completed by the user provided in the params. Returns status code 200 if successful.
2. POST /history/:user - Checks if a question is given and adds the question to the list of questions done by the user that is stored in the database. Returns status code 201 if successful

9. Frontend

We have used **React** and **Material UI (MUI)** for our frontend. React is one of the most widely used JavaScript library for building user interfaces. We have heavily utilised React Hooks, which they are powerful and easy to use, and it provides us with an easy way to reuse state logic without changing our components hierarchy. MUI provides us with an easier way to style our components, where their components are reactive.

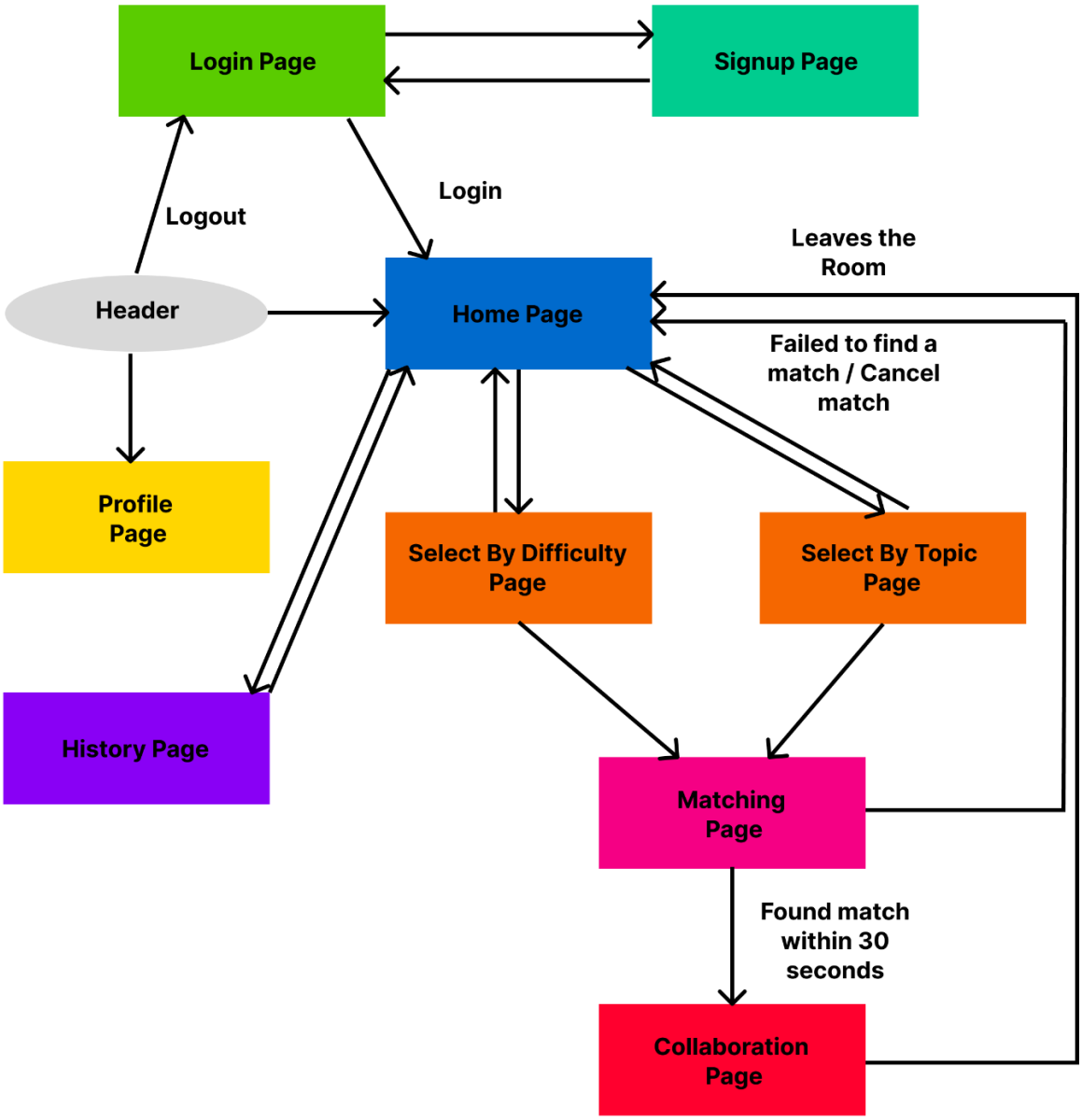


Image: User Interface (UI) flow

Here is our UI flow, which shows all the possible page interactions.

App.js is our parent file, where we incorporate the Header component so that every page has a fixed and consistent Header. For all pages other than Login and Signup, we will check if the user is actually logged in, if not we will redirect the user to the Login page. This is to ensure that the user has to login before accessing other pages.

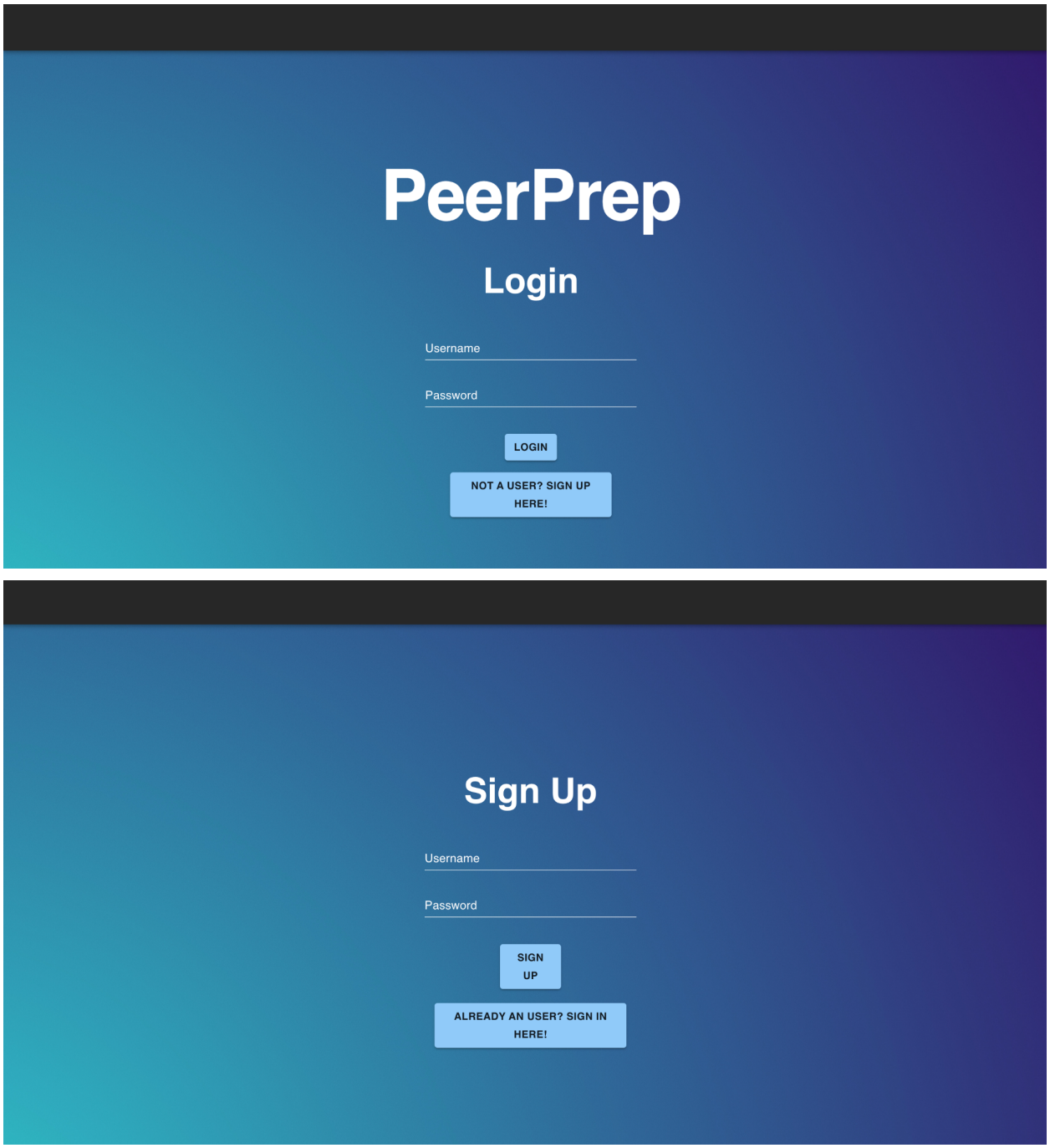
We also utilized localStorage to store the user's information and question (in the Collaboration Page), so if the user refreshes the page, it will still retain the vital information to reload the page. We will clear the localStorage when the user logs out.

9.1 User Interface Design Considerations

9.1.1 Header

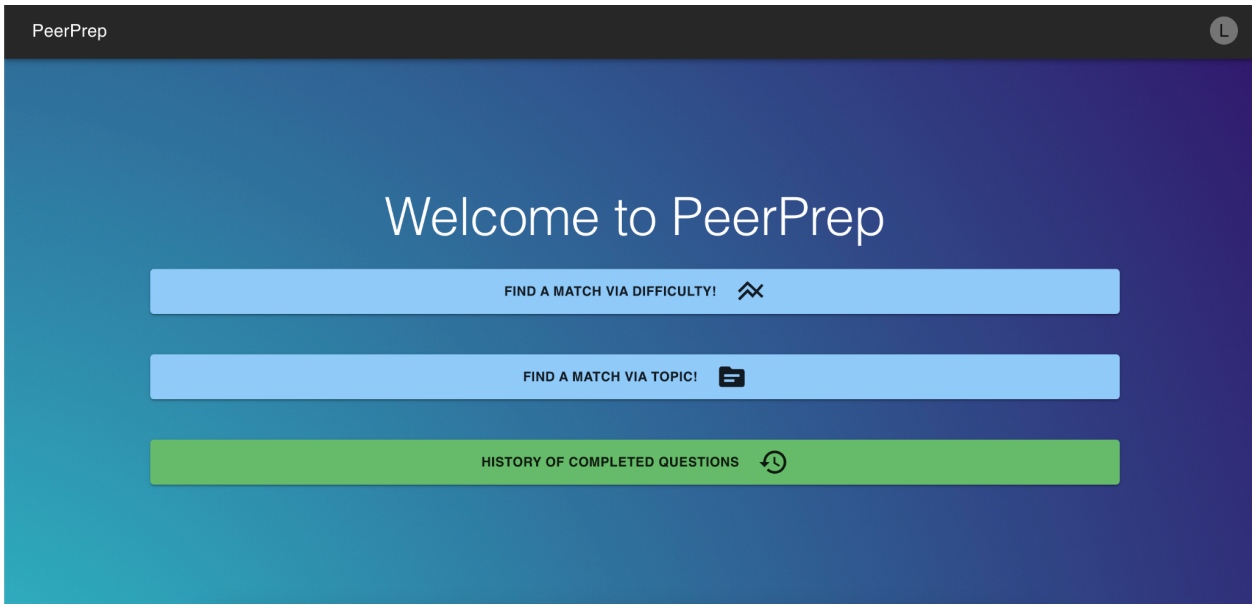
The header provides a consistent user experience between all the pages in PeerPrep, where the user can click on the "PeerPrep" text to go to the Home page, and on the right-hand side, upon clicking the IconButton, it will display a dropdown menu, where the user can choose to go to their Profile page or Logout. The IconButton takes the first letter of the username and uppercase it.

9.1.2 Login / Signup Page



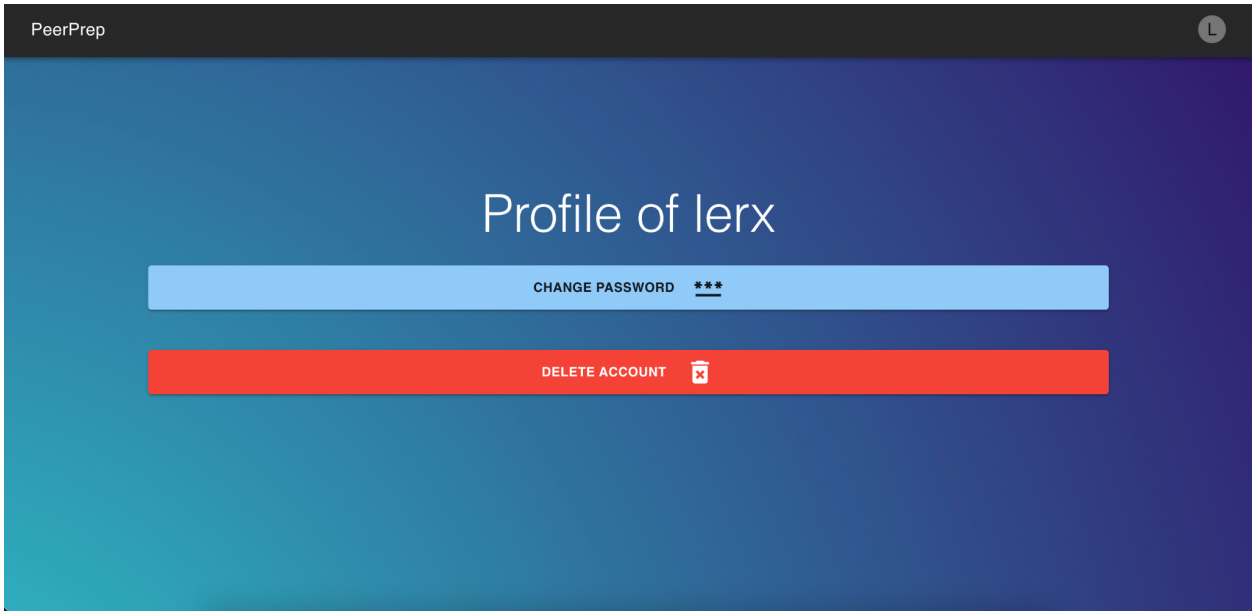
For the Login and Signup page, the header is blank (conditional rendering), this is because the user has to log in to access the features of PeerPrep like the Home and Profile screen. Both pages interact with the User Service.

9.1.3 Home Page

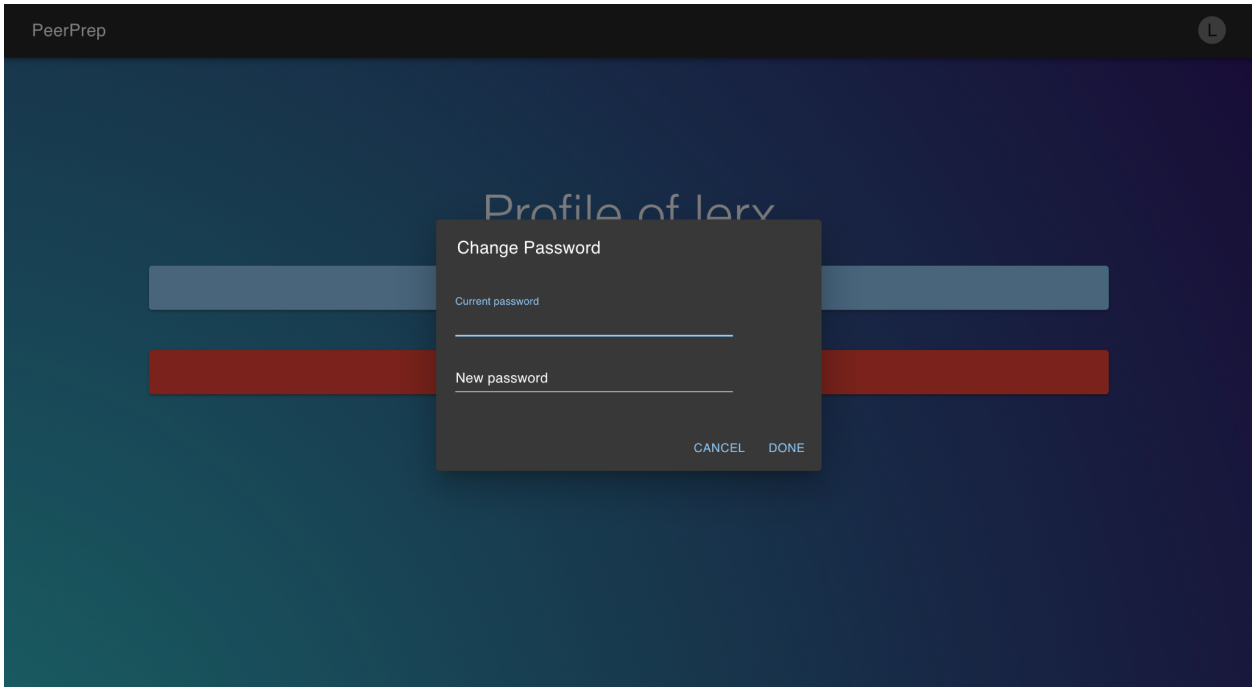


Users will be greeted by the Home page, and the buttons are large and color-coded. Each button has an icon, which is more visually appealing to the user.

9.1.4 Profile Page

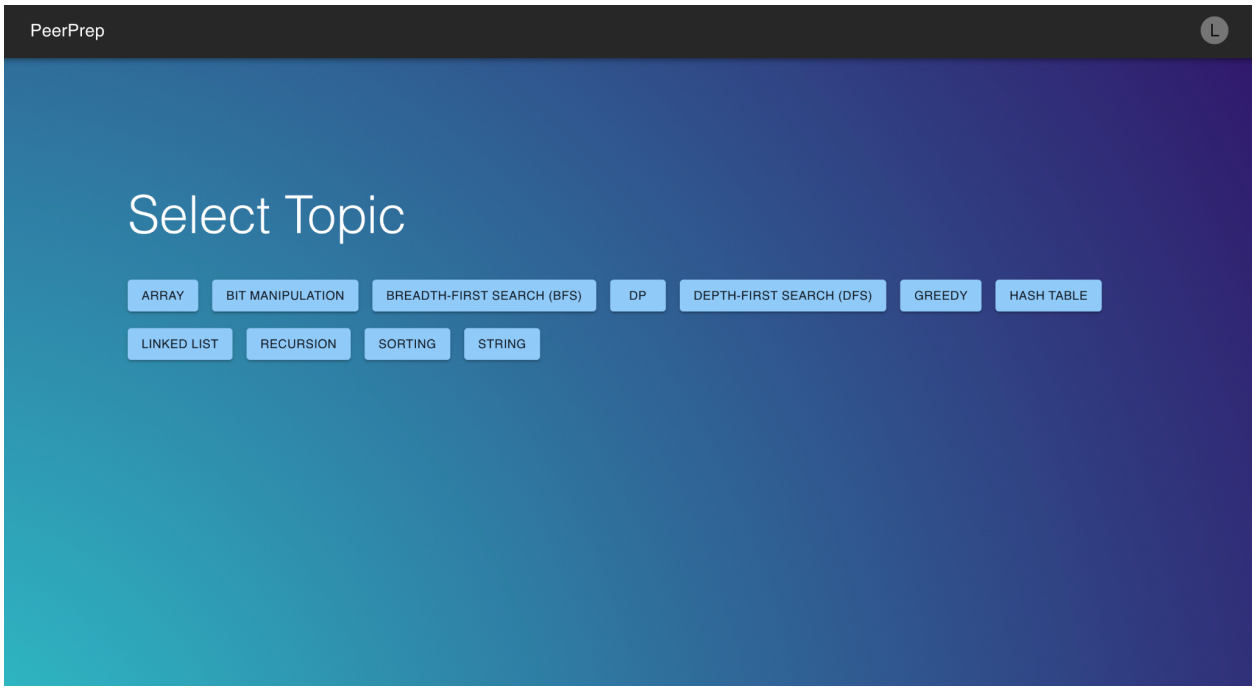


Similar to the Home page, the buttons are large and colored, and they each have a representative icon. The “Delete Account” button is in red as we want the user to be sure that they want to delete their account, and the deletion is not reversible.



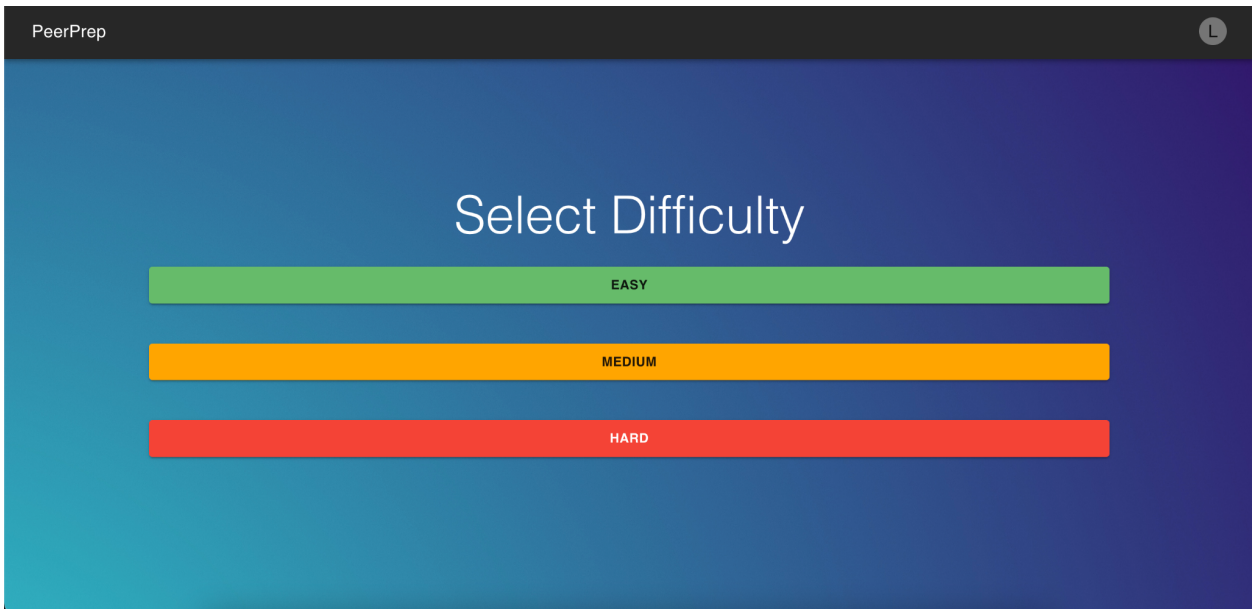
Modal popup when the user click on “Change Password” button.

9.1.5 Select By Topic Page

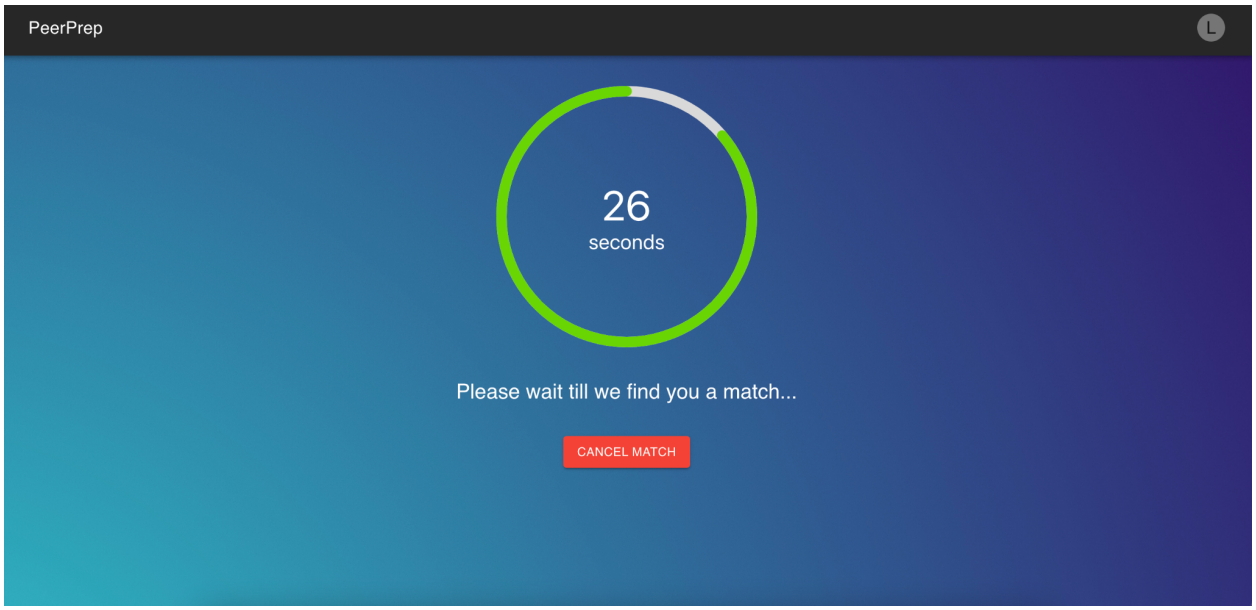


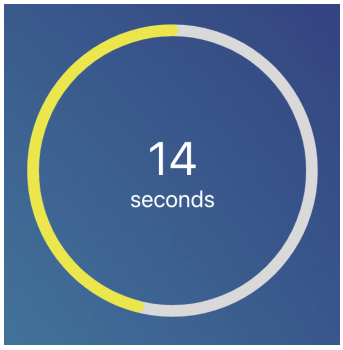
The Select By Topic page interacts with Question Service to retrieve all the topics from the question database.

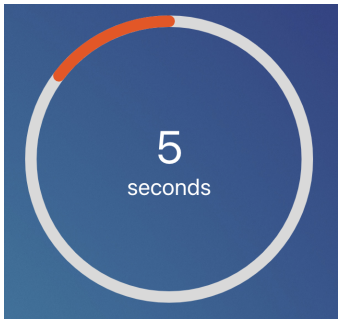
9.1.6 Select By Difficulty Page



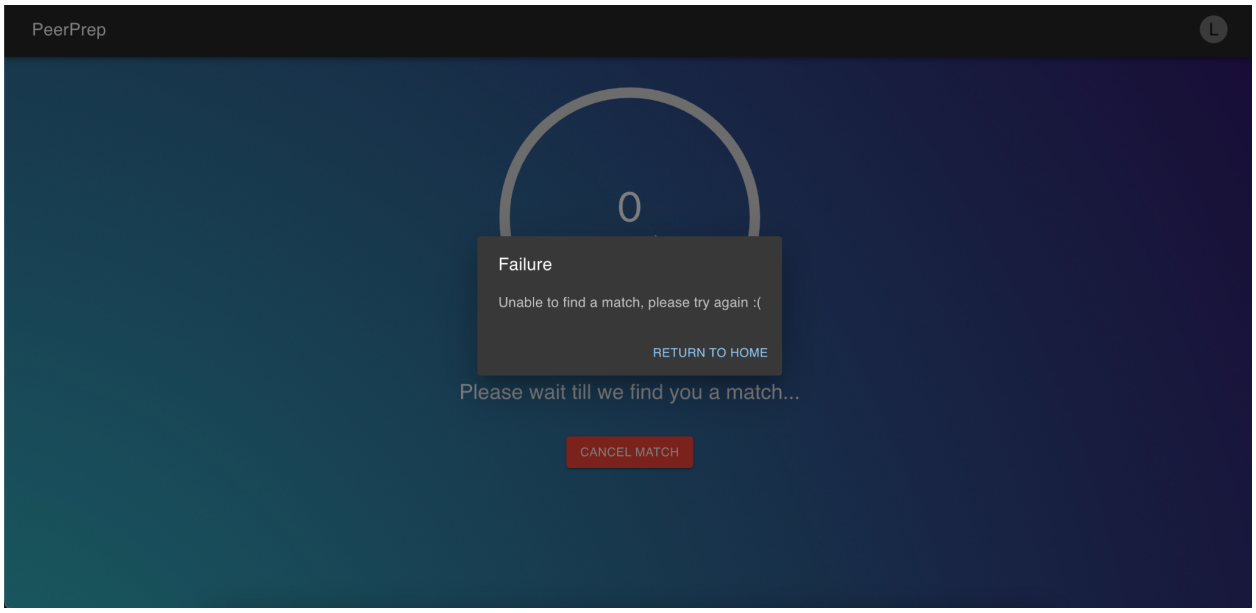
9.1.7 Matching Page





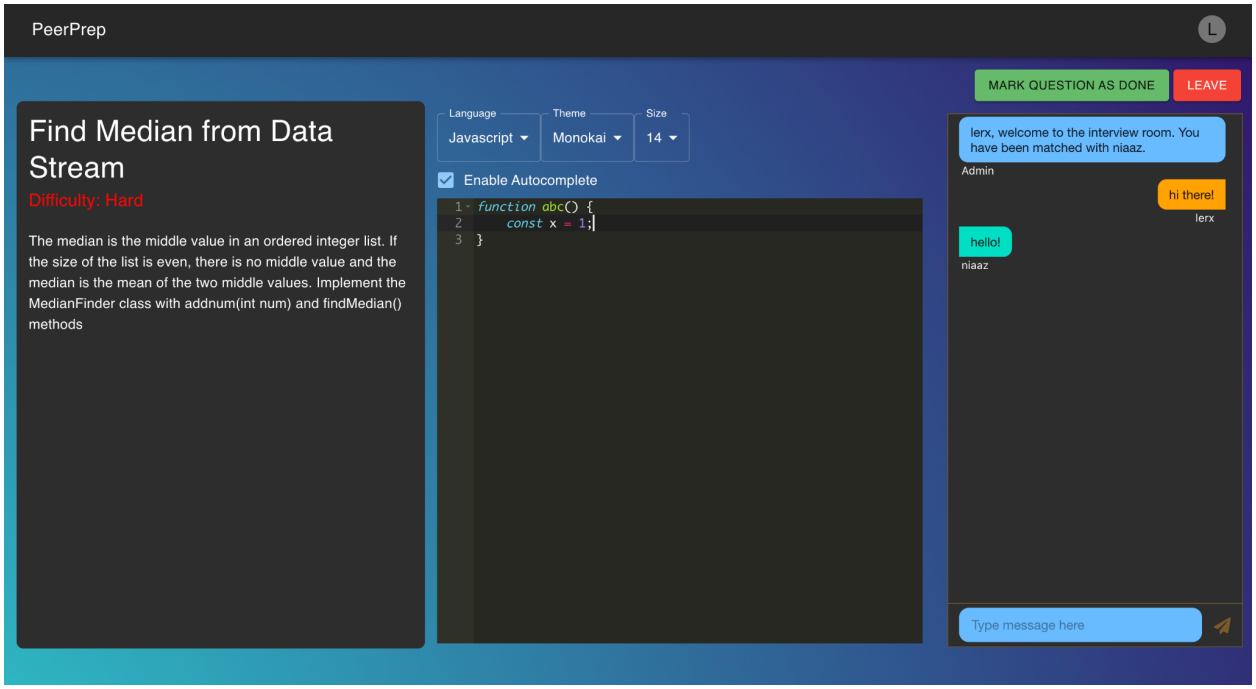


For the matching page, our countdown timer changes color depending on the number of seconds left. Initially, it is green, at the halfway mark, it is yellow, and in the last few seconds, it is red. This gives the user an easy visual indication of how much time is left for the matching. The Matching page interacts with the Matching Service.



A popup will appear if the user did not managed to find a match, and the button will return them to the Home page.

9.1.8 Collaboration Page

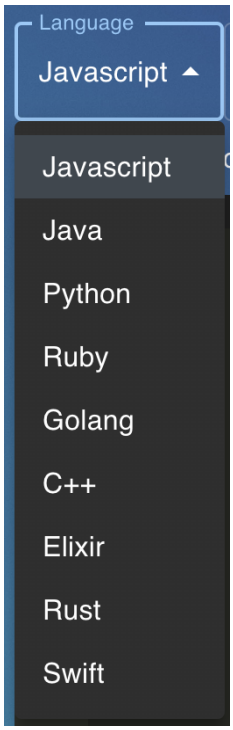


In the Collaboration page, there are 3 main components:

1. Question
2. Live editor
3. Chat

For the question, we have decided not to display the topic, as we want the interviewee to figure out by themselves what approaches they can use to solve the question. Like the buttons in the Select Difficulty page, the difficulty is color coded.

For the editor, it has syntax highlighting and autocompletion (optional), so the user can feel like they are coding in an Integrated Development Environment (IDE). There is also accesibility options like being able to change the Editor's theme and adjust the font size. The user can select their preferred programming language, which the given options are:



For chat, the chat bubbles are colored differently, directional and contains a footer with the sender so that we can easily tell who sent the message.

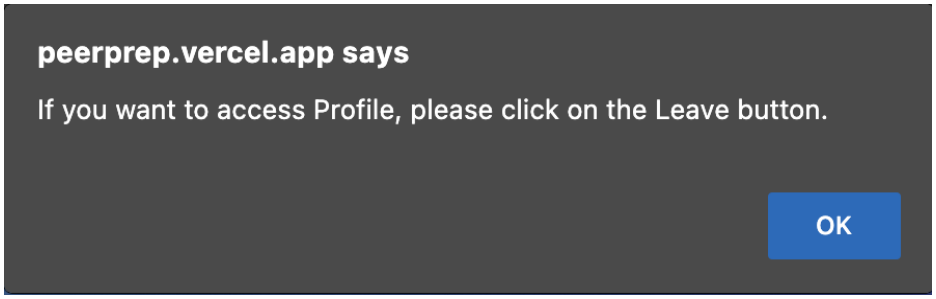
- Admin: Blue
- Sendee: Turquoise and directional (left)
- Sender: Orange and directional (right)

The admin messages is to inform the user the state of the room, like whether the other user joins / leaves the room.

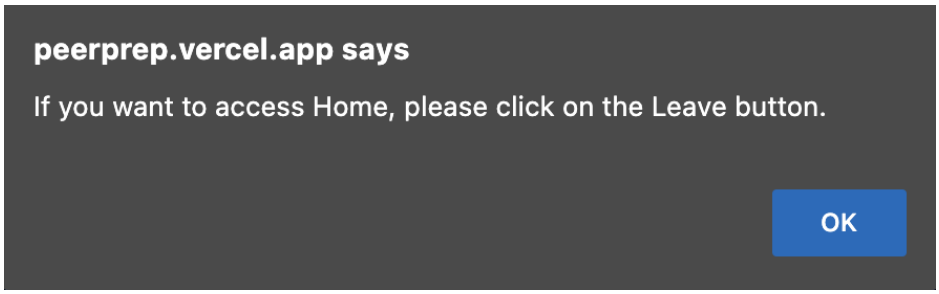
Toast messages

These are the toast messages that will appear on the bottom left-hand side, where it gives the user more context when the joined the room and after clicking on “Mark Question as Done” button respectively.

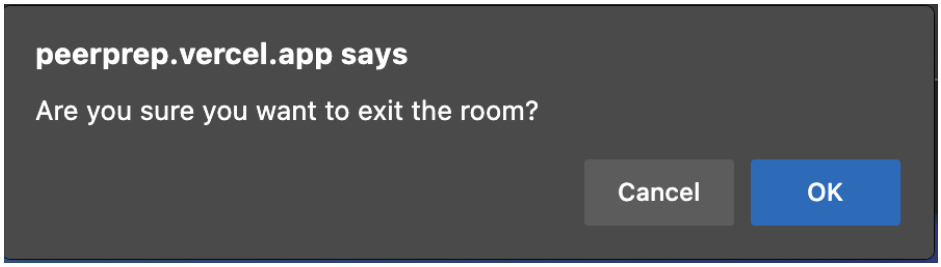
Browser alerts
When the user clicks on the Profile button:



When the user clicks on the Home (PeerPrep) button:



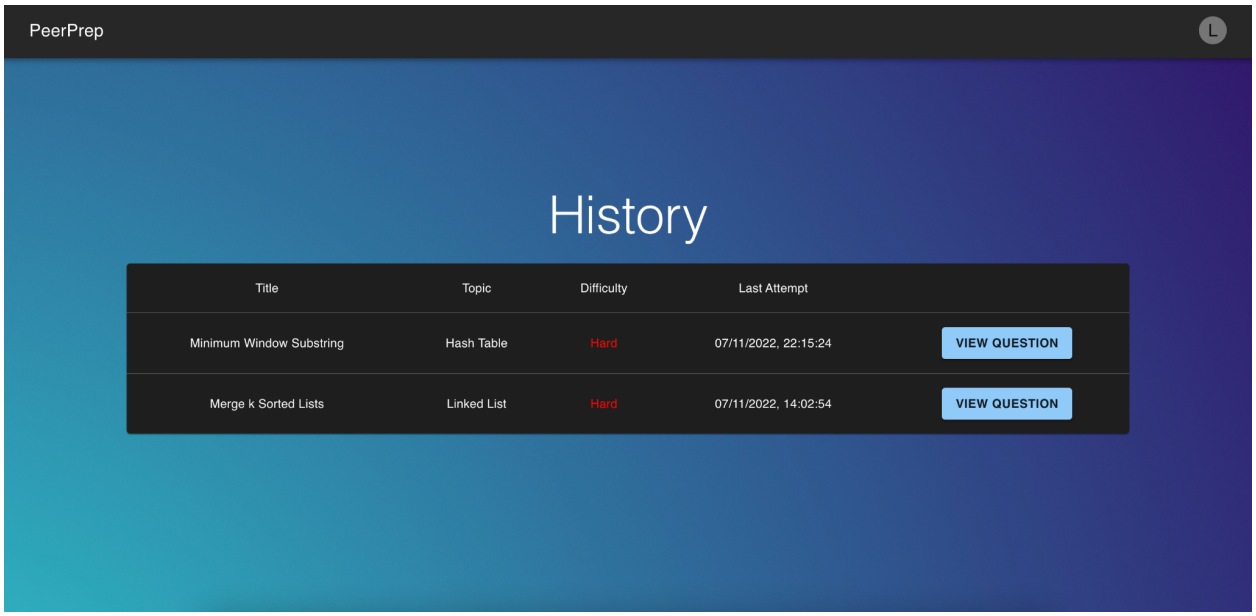
When the user clicks Back:



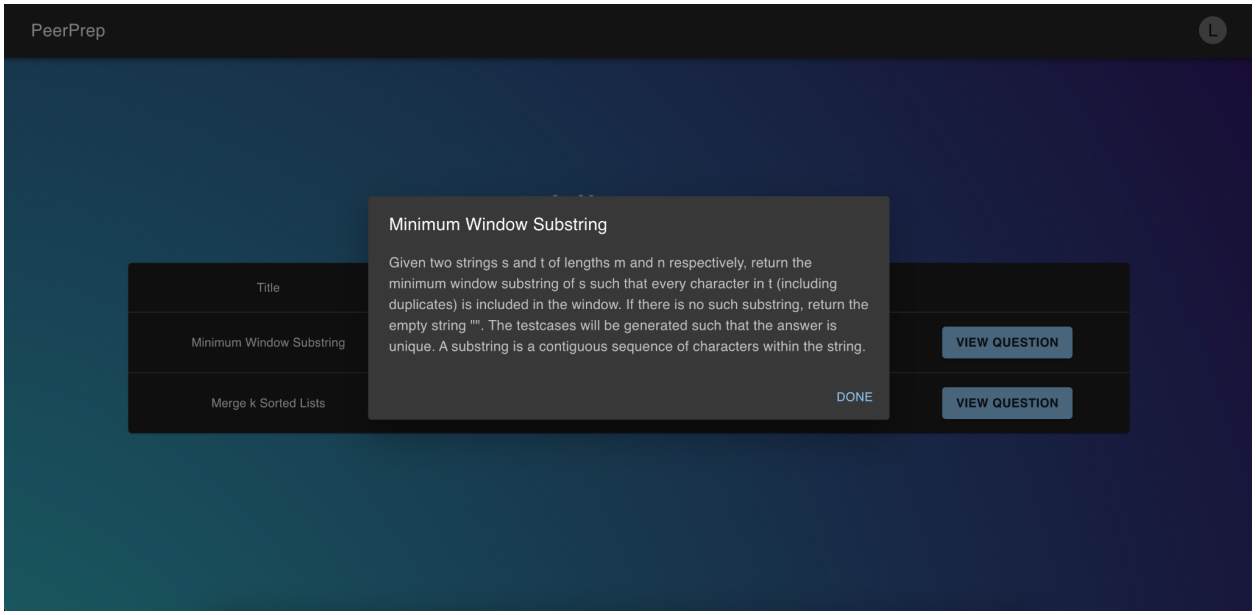
For the back button, we will use an event listener on “popstate” in useEffect, so we can override the default behaviour and create a confirmation popup (as shown above).

9.1.9 History Page

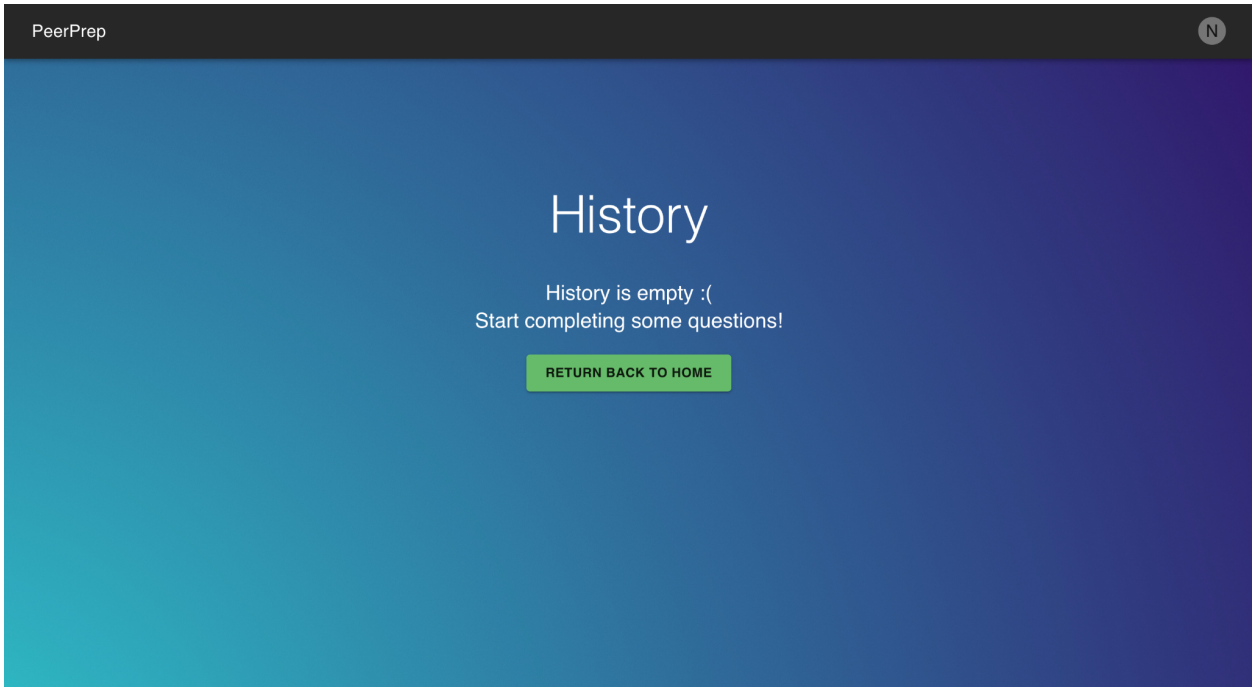
If the user marked >= 1 question as completed:



When the user clicks on “View Question” button:



If the user’s history is empty:

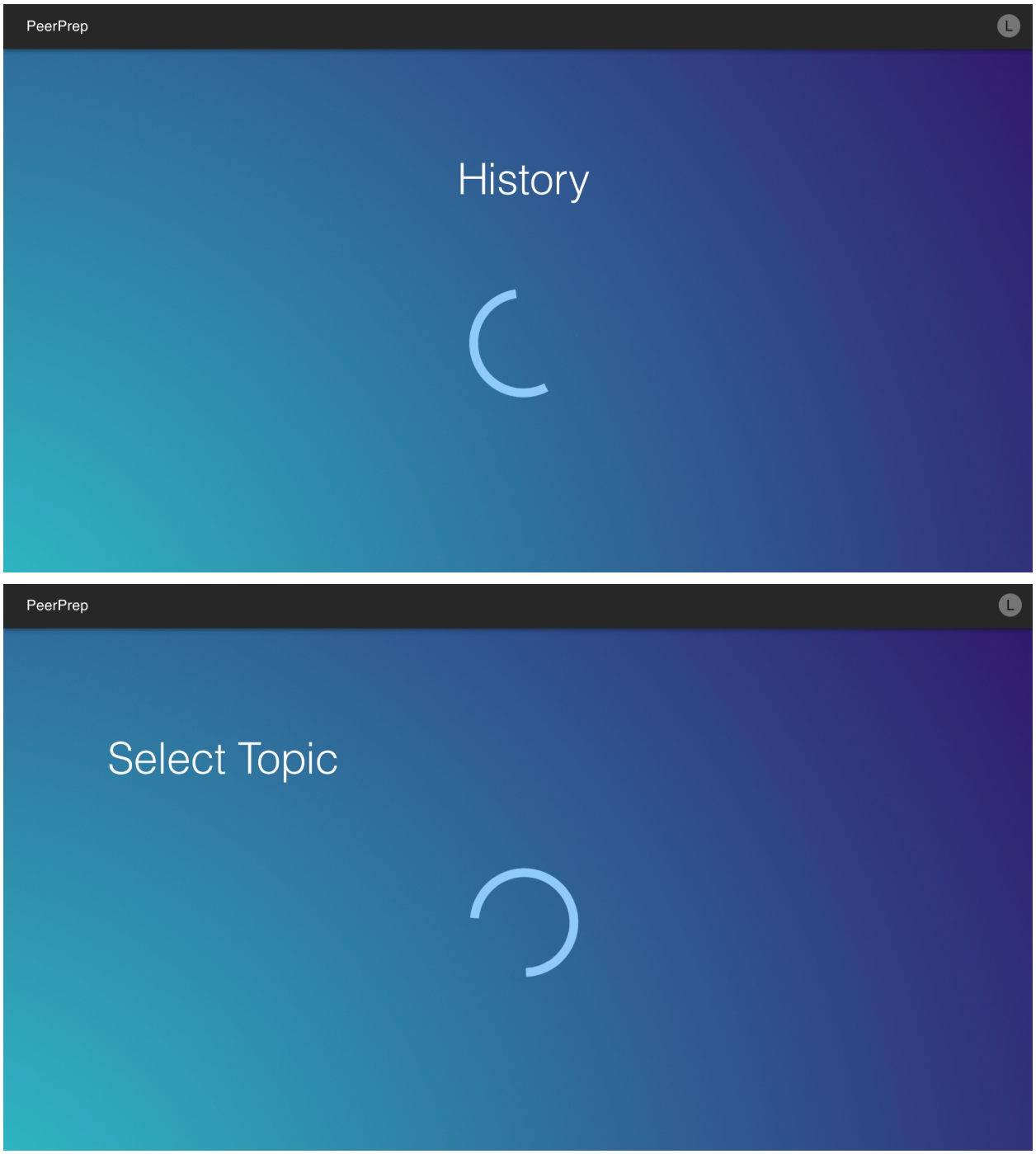


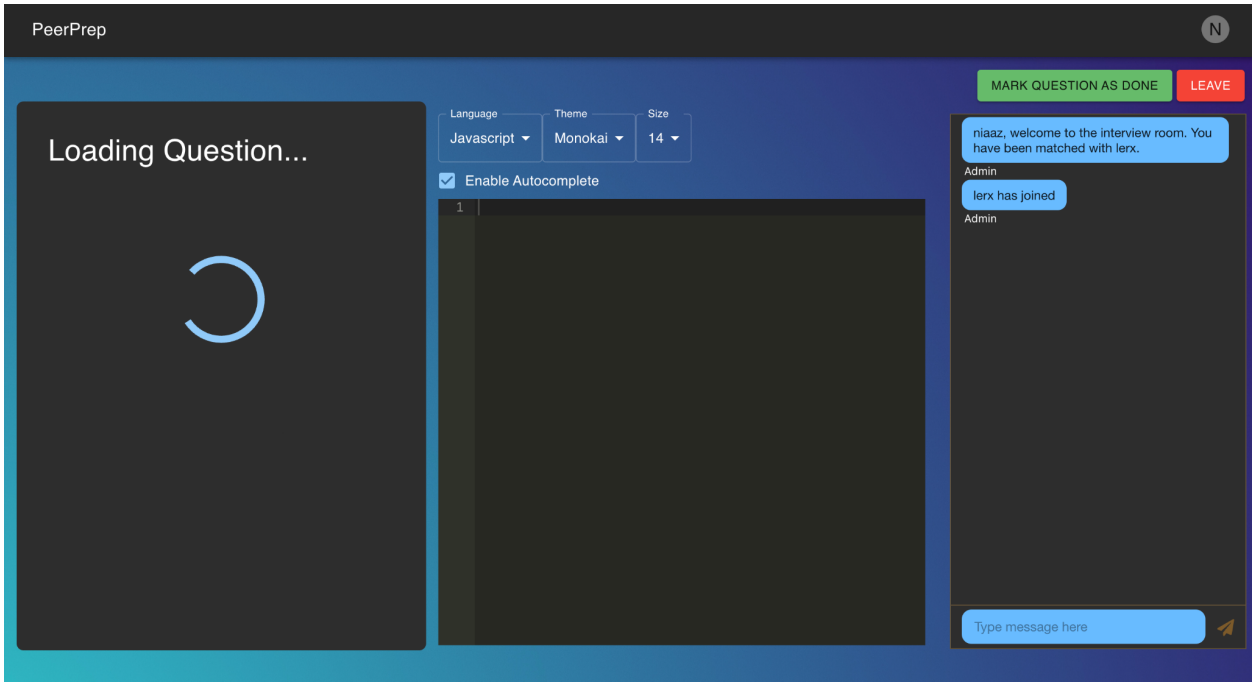
There are 2 scenarios a user could have in the History page:

- Completed >= 1 question
- Completed no questions

The history page retrieves the data from History Service, and based on the scenario does a conditional rendering.

9.1.10 Loading element





Sometimes, it takes a while to retrieve information from our backend (like Question Service), which is why we have incorporated a “loading” circle to visually indicate to the users that the data is being retrieved. After the data is retrieved, we will display the content to the user.

10. Remarks

10.1 Challenges Faced

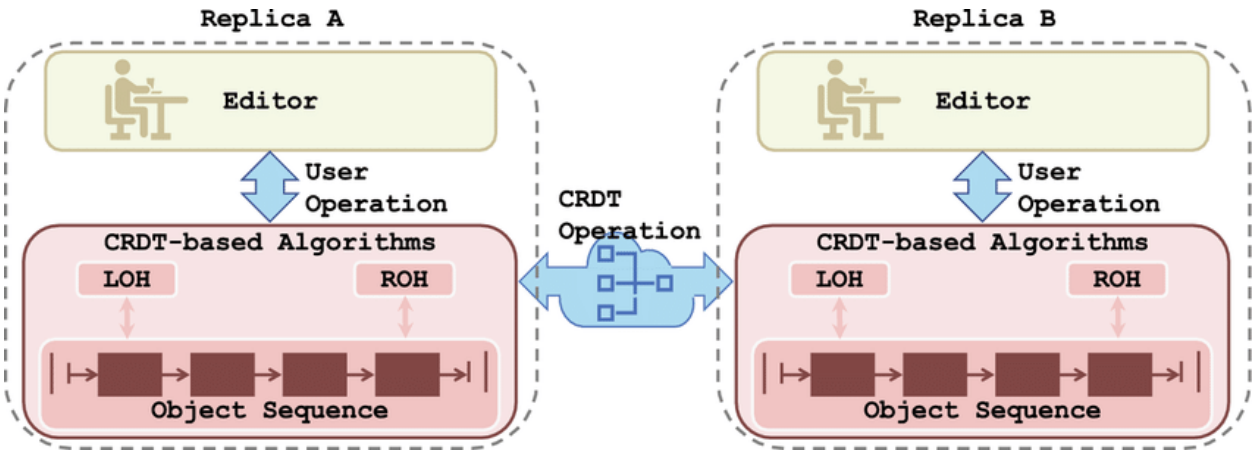
The following were the challenges faced during the course of our project:

1. Unfamiliar technologies that had to be picked up within a short span of time
2. Difficulty in deciding the most appropriate application architecture and communication mechanisms between services
3. Difficulty implementing matching logic
4. Difficulty following best practices during deployment

10.2 Potential Extensions

The following are some potential extensions that my team would have implemented if time was not limited:

1. Simultaneous Editing
   We wish to extend our collaboration service to support simultaneous real-time editing. Our research reveals that we are required to build a service that supports Conflict-free replicated Data Type (CRDT). The need for CRDT arises because of the following reasons:
   1. The editor should be able to update the code independently as well as concurrently independent of the state of the editor used by other users
   2. Each editor of matched users might have a different state, but must eventually converge.



Source: DOI:10.1007/s11227-022-04308-7

An implementation of the CRDT that took our interest was Yjs. It allows greater flexibility in terms of choosing how to communicate with the clients. Few of the communication methods are as follows:

1. WebRTC: This allows us to provide peer-to-peer communication
2. Websocket: This allows us to have a centralized point to manage the logic

However, we might have to take note that we might have to switch to the Frontend editor implementation to implement operations that can collaborate with Yjs. Such editors are ProseMirror, CodeMirror, and Moncao.

2. Integrated Code compilers and interpreters

Allowing users to execute the written code and test against custom test cases would elevate the user experience and benefits of the Peerprep platform. This would allow users to simulate real-world interview scenarios and enhance their learning further such that language-specific syntax errors are caught and users can verify the accuracy of their algorithms immediately.

10.3 Reflections and Learning Points

For our frontend, it is initially challenging as we have to learn React and React Hooks to properly render our information and interact with our backend microservices. After making our frontend work, we have to “beautify” our User Interface (UI), which we have to take advantage of all the components available in MUI (even some external libraries), and use CSS styles to align items and make it match our theme. Furthermore, we have to ensure our UI is reactive, as users may have various kinds of screen sizes, and initially, when we developed it on the laptop screen it looked fine, but on a big external monitor, it looked out of place. Thus, it is vital to test our application on different devices/screens.

As most of our experiences with CI/CD pipeline were what we learned in this module, we faced additional difficulties trying to deploy our application. This took up quite a lot of our time as we had to google a lot of things out during the numerous problems we encountered. However, it was fulfilling when we managed to set up the pipeline to see our application deployed and running.

For the backend, setting up the microservices was a lot easier as compared to the frontend or deployment as we were all familiar with setting up a Node.js application with the Express framework and how to connect to a database as we also learned this during the OTOT Task B.

All in all, we feel that this project turned out to be slightly more difficult than expected. We should have set aside more time for deployment so that we would have managed our time better.