CS3219

Software Engineering Principles and Patterns Group 38 Final Report: AlgoHike

Taufiq Bin Abdul Rahman	A0218081L
Tay Yan Han	A0216335L
Lim Yu Zher	A0217723H
Kevin Foong Wei Tong	A0217921H

Table of Contents

1. Introduction

1.1 Background

AlgoHike is a web application that helps students and job seekers in the technological industry build up their confidence in tackling technical interviews through interview simulations with randomly matched strangers.

Via various features such as a question generator, collaborative editor, real time chat messaging service and video and voice chat, it provides users with a realistic environment to practise coding interviews.

1.2 Purpose

Coding interviews are a key aspect of job hunting in the technological industry. Employers often evaluate potential employees' capabilities by assessing how well they can solve algorithm questions and explain their solutions in a clear and effective manner.

To aid users in landing their dream technical jobs, AlgoHike not only allows users to practise algorithm questions, but also matches them with other users so he/she can explain their solutions to them and receive feedback on areas they can improve. Through such technical interview simulations, AlgoHike aims to guide users in their algorithmic hike to the summit, and as an added bonus, meet new friends along the way.

2. Individual Contributions

<table>
<tr><td rowspan="2">Taufiq</td><td>Code
Frontend Contributions
- Frontend (Home Page) UI
- Redux Logic for Home Page
- Integration of Match Service APIs with Frontend
- Frontend (Collaboration Page) UI
- Redux Logic for Collab Page
- Integration of Collab Service APIs
- Frontend (User History) UI
- Integration of History Service APIs with User History Page
- Frontend (Questions Bank, Detailed Question Page) UI
- Integration of Questions Page with Question Service APIs.
- Frontend Navigation bar styling

Service Contributions
- Matching Service backend (user matching functionality)
- Collaboration Service (Socket event handlers)

Container Orchestration
- Set up Docker-compose for services and frontend</td></tr>
<tr><td>Report
- User requirements/stories
- Functional requirements
- Non-functional requirements
- Report layout
- Product Backlog
- Design Decisions
- Tech Stack
- Overall Application Architecture
- Application Flow
- Match Service Architecture
- Collab Service Architecture</td></tr>
<tr><td>Kevin</td><td>Code
Frontend Contributions
- Login & Signup Page
- User service Redux Logic
- Real-time text editor</td></tr>
</table>

	- Chat messaging components - Collab Page - Frontend theme design (colour/material UI choice etc) **Service Contributions** - Real-time code editor - Communication service (Real-time texting functionality) **Tooling** - Setting up of linters / pre-commit hooks across services - Trello board setup (configuring with add-ons for epic-story hierarchy, effort estimates, list organisation)
	Report - User requirements/stories - Functional requirements - Non-functional requirements - Quality attributes prioritisation - Chat architecture - Real-time editor architecture - Project management, database, backend & frontend design decisions - Overall architecture design - Project Backlog - Application flow (for chat messages) - Part of reflections
Yan Han	**Code** **Frontend Contributions** - Question Component UI - Integration of Question Service APIs - Collaboration Service Redux Logic, Integration of Collaboration Service APIs and socket event handlers - Login and Signup skeleton UI **Service Contributions** - Collaboration Service APIs - Question Service setting up of question bank & APIs - History Service APIs
	Report - Introduction - User requirements/stories - Functional requirements - Non-functional requirements - Backend Architecture - Question Service Architecture

	- History Service Architecture - User Flow
Yu Zher	**Code** **Frontend Contributions** - Home Page and App Bar UI - Sign up/login validation - Firebase Auth sign up and log in - React Router protected routes - Reset password UI and functionality - User Service Redux Logic, Integration of User Service APIs - Video chat functionality and UI **Service Contributions** - User Service - Video Service
	Report - User requirements/stories - Functional requirements - Non-functional requirements - Development Process - Frontend Architecture - User Service Architecture - Video Service Architecture - Extensions - Challenges - Limitations

3. Requirements

3.1 User requirements

As part of our ideation process, we followed a user-centric approach and focused on identifying user stories and hence obtaining our user requirements.

As a(n)	I want to	so that
Interview preparer	Find a random person to interview me	I can practise interviews without knowing anybody
Interview preparer	Pick a question difficulty	I can practise interviews with questions of different difficulties
Interview preparer	Have an account	I track my progress and see my past solutions
Interview preparer	Change my account password	I can keep my account secure by updating the password.
Interview preparer	Communicate with my interviewer	I can explain my solutions to him/her
Interviewer	Communicate with my interviewee	I can carry on with follow up discussions with him/her and ask him/her more questions about their solution or the question
Interview preparer	Type my solution into an editor	I can solve the problem
Interviewer	View my interviewee's solution in real time	I can discuss/ give feedback in real time
Interviewer	Type the question or comments into an editor	I can give the interviewee the question or write comments on their solution
Interview preparer	Evaluate correctness of my solution	I can see if its correct
Interview preparer	Be able to choose to solve problems alone	I can practise by myself if cannot find someone else
Interview preparer	Leave the room I am currently in	I can proceed to match with another peer
Interview preparer	Practise multiple questions with the same	-

	interviewer	
Interview preparer	Verbally communicate with my interviewer	I can explain my solution verbally
Interviewer	Verbally communicate with my interviewee	I can give my interviewee instructions
Account owner	Choose whether I will stay logged in	I can choose whether I need to log in again next time
Account owner	Reset my password	I can regain access to my account if I forget my login password.
Interview practicer	Be able to see my session history (username of matched user, questions attempted)	I can track my progress
Interview preparer	Be able to see a list of available questions	I can see what questions are available to practise
Interview preparer	Sort and filter the list of available questions by difficulty and name	I can easily find the questions I want to view

3.2 Software Requirements

3.2.1 Functional Requirements

Based on the user requirements identified above, we came up with functional requirements and ranked them in terms of priority to manage team projects and timelines better.

Functional Requirements			
S/N	Functional Requirement	Priority	Additional justification for priorities (as necessary)
User Service			
FR 1.1	The system should allow users to create an account.	High	
FR 1.2	The system should be able to validate users' credentials and allow users to log into their accounts if their credentials are correct.	High	
FR 1.3	The system should inform the user if they entered the wrong username and/or password and prevent logging in.	High	
FR 1.4	The system should disallow passwords which are too weak (FR complies with NFR 1.2).	High	For improved account security
FR 1.5	The system should disallow creating an account with a username that already exists	High	
FR 1.6	The system should allow users to log out of their account.	High	
FR 1.7	The system should allow users to reset their password if they forget their password.	High	
FR 1.8	The system should allow users to remain logged in after exiting the app.	Low	Convenience feature for users
FR 1.9	The system should allow users to change their password while logged in.	Low	
FR 1.10	The system should log the user out after their password has been changed.	Low	

FR 1.11	The system should allow users to delete their account.	Low	
Matching Service			
FR 2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High	
FR 2.2	The system should be able to match two waiting users with the same difficulty levels and put them in the same room.	High	
FR 2.3	The system should inform the users that no match is available if a match cannot be found in the time limit.	High	
FR 2.4	The system should provide a means for the user to leave a room once matched.	High	Allows users to change matched partners once they are done with their current interview session.
FR 2.5	The system should allow the user to start a room alone if no match is found or the user prefers to practise questions without an interviewer.	Low	The user can still view the questions that are available and practise it on their local notepad or coding environment (FR3.3).
Question Service			
FR 3.1	The system should be able to provide a list of coding questions based on the user's selected difficulty.	High	
FR 3.2	The system should be able to exclude certain questions from being returned.	Medium	Used to prevent duplicate questions from being generated for the same mock interview session. Not the highest priority since the user can click next to skip to the next question if they realise that they have encountered this question before.

FR 3.3	The system should be able to show the entire list of available questions.	High	
FR 3.4	The system should be able to sort and filter the entire list of available questions by their name and difficulty.	Medium	Users can easily find a question from the available list of questions.
FR 3.5	The system should be able to return a specifically chosen question.	High	So that users can view particular questions that they have tried in the past or want to try on their own.
Collaboration Service			
FR 4.1	The system should coordinate the questions and ensure the same chat room and editor for the two users.	High	
FR 4.2	The system should allow users in the same room to obtain another question to solve in the same room.	High	Allows matched partners to solve more questions together if they had a good experience
FR 4.3	System should keep track of what questions have already been attempted by a room.	Medium	Ensures that each question received is distinct
FR 4.4	If there are insufficient questions remaining, the system should inform the room participants.	Low	It is unlikely that the users will run out of questions considering the large expected size of our question bank.
Chat Service			
FR 5.1	The system should allow two users to chat with each other via messages in real time.	High	

FR 5.2	The system should be able to persist messages.	Medium	
Video Service			
FR 6.2	The system should allow two users to see each other via video.	Low	Seeing the other party may not be critical for communicating coding ideas during mock interviews
FR 6.3	The system should allow two users to chat with each other via voice chat.	High	Important for interviewer and interviewee to communicate since verbal communication is critical for interview practice
History Service			
FR 7.1	The system should be able to show the history of the user's sessions (time, questions, difficulty of session).	Medium	Serves as a learning pathway for the user to keep track of their progress.
FR 7.2	The system should be able to store the histories of each user.	Medium	
FR 7.3	The system should allow an easy way to view the questions listed on a user's history.	Medium	Makes it more convenient for users to view past questions and remember their mistakes.
Editor Service			
FR 8.1	The system should allow users in the same room to type in a shared editor The system should allow both users to type simultaneously in the editor. The system should update the collaborative editors in real time.	High	

3.2.2 Non-functional requirements prioritisation

NFR trade offs & considerations

In order to choose our most important quality attributes, we looked at the main purpose of our application, which is a online coding mock interview preparation web application.

Since our application does not store much sensitive information on the users, we did not feel security was a key priority. However, we felt that implementing fundamental security practices was still beneficial and a good practice to secure user accounts.

Additionally, since the information transmitted through a mock interview is for practice and not extremely critical, we are able to tolerate some data errors and integrity is not a key priority either (hence lower scores were given to integrity).

On the other hand, mock interviews require real-time communication (in terms of code editor performance and communication), hence we felt that performance is a key focus.

Additionally, since our audience who are interview preparers are usually power users who need to undergo a lot of practice with as much efficiency as possible, usability is a key focus for us (hence the highest scores assigned)

	Score (10)	Security	Integrity	Performance	Usability	Portability	Scalability	Availability	Robustness
Security	4		⇐	⇑	⇑	⇑	⇑	⇑	⇑
Integrity	3			⇑	⇑	⇑	⇑	⇑	⇑
Performance	9				⇐	⇐	⇐	⇐	⇐
Usability	8					⇐	⇐	⇐	⇐
Portability	5						⇑	⇑	⇑
Scalability	6							⇑	⇑
Availability	6								⇑
Robustness	7								

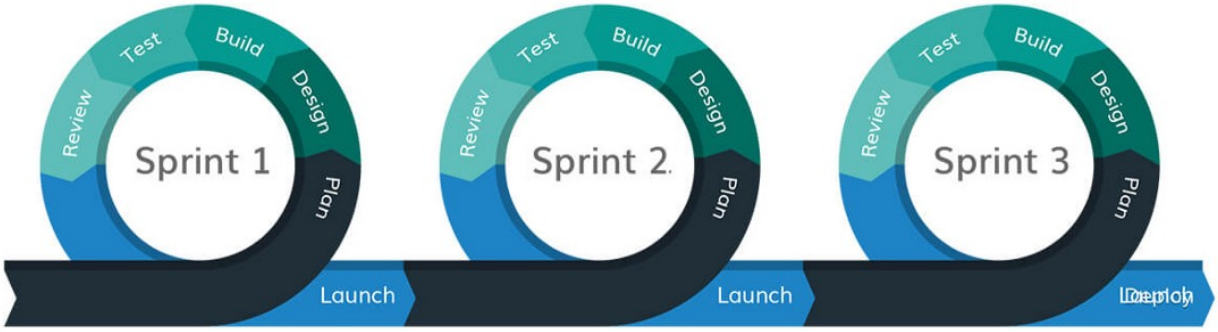
3.2.3 Non-functional requirements

Non Functional Requirements			
S/N	Non Functional Requirement	Priority	Our approach to implementing it
Security			
NFR 1.1	Users' passwords should be hashed and salted before storing in the DB.	Medium	Firebase Authentication hashes all account passwords that are stored.
NFR 1.2	The app should have strong password checks (8 minimum characters, 1 uppercase, 1 lowercase, 1 digit, 1 special character) when the user signs up or changes their password.	Medium	This is checked in the frontend during sign up. Firebase Authentication also performs a check before an account is created.
Performance			
NFR 2.1	A match should be returned within 1s if there's an available one		Usage of SQLite3 database allows for faster performance compared to other databases (10-20x faster than PostgreSQL, 2x faster than MySQL). (*SQLite Database Speed Comparison*, 2014)
NFR 2.2	The matching party should be able to see what the user types within 1 second	High	Usage of a real-time editor with low latency enables this.
NFR 2.3	The users should be able to communicate with each other in real time.	Medium	Provide low latency text messaging, video and voice communication channels.
Usability			
NFR 3.1	The app should be displayed correctly in different laptop/desktop screen sizes.	High	Usage of Material UI responsive grids to allow apps to be presented correctly for different screen sizes.
NFR 3.2	Enable easy navigation between pages (<=2 clicks between pages)	Medium	Implement buttons that allow navigation between pages in a single click (reduce number of interactions to find interview partner)
NFR 3.3	The coding platform should provide a way for interviewee to communicate with interviewer without having to	High	Provide voice/video chat functionality on top of text messaging to allow 'hands-free' communication

	stop coding		
NFR 3.4	The coding platform should be easy to learn to use.	High	Provide similarity to other familiar systems like Leetcode/BinarySearch.com and simplify process to find match/limit number of options displayed (just 3 choices easy, medium, hard)
Scalability			
NFR 4.1	The database should be able to store 10K users.	Medium	MongoDB and Firebase can be scaled easily as needed.
NFR 4.2	The app should be able handle up to 500 messages in the chat within a single session.	High	Code allows storage of up to 500 chat messages within a single session.
Availability			
NFR 5.1	The app should have an uptime of 99% at all times.	High	MongoDB has an availability of 99.995% and Firebase has an uptime of at least 99.95%. (*Reliability | MongoDB*, n.d.)
Portability			
NFR 5.2	The app should be able to run on Mozilla Firefox, Google Chrome, Microsoft Edge, Safari	High	Choose a frontend framework & UI component library (we chose React/MUI) which has known support and continuously well-maintained to work on different web browsers.
Robustness			
NFR 6.1	The code editor should be able to restore a session if a user disconnects and reconnects.	High	Session data is stored in MongoDB and can be restored once a user reconnects. Persist and autosave text written to the text editor to account for potential user failures.

4. Development Process

We made use of the agile development process during the course of our project.



Agile Methodology

Our team decided to divide the timeline into one-week sprints, each with a cycle of planning, designing, building, testing and reviewing. This allowed us to allocate work in small and usable increments, evaluate requirements, plans and results consistently and thus respond to changes swiftly and efficiently. Some of the major steps in our sprint cycles include: Plan, Design, Build, Test and Review, as shown in the figure above. (*Agile Development From a Programmer's Perspective | by Dasith Kuruppu | Level Up Coding*, 2019)

4.1 Planning and Product Backlog Refinement

This was done in our weekly scrum meetings every Tuesday.

In this step we first estimated how many tasks are essential and how long it takes to complete those items. We also refined items to clearer or smaller user stories if needed.

The product backlog is stored on our collaborative Trello board. The tasks are initially written in the Overall Backlog column. At the beginning of each sprint, the tasks for the current sprint are moved into the Current Sprint (Backlog). When a team member begins the task, the task is moved into Current Sprint (In Progress), and when it is ready for review, the task is moved into Current Sprint (For Review). When the task is merged into the main branch, the task is moved into Current Sprint (Completed). At the end of the sprint cycle, the tasks which are completed for the sprint are moved into a new column, for example, a column labelled Sprint 3 (W9) [Completed].

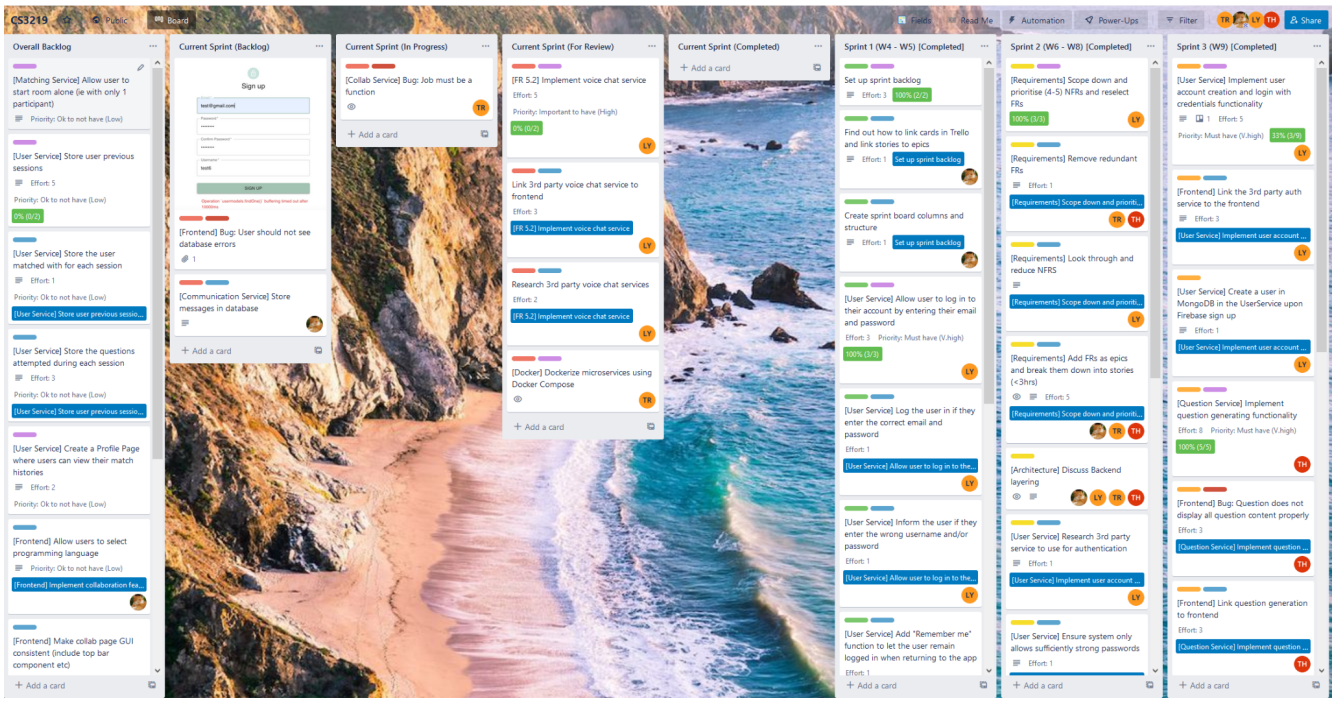
Each task is labelled with a few labels, namely whether it is an Epic, Story or Bug, as well as another label indicating the sprint that the task belongs to. Each sprint is labelled with a unique label and colour. The team member which is pre-assigned to each task at the start of the sprint is assigned the task on Trello.

Link to Trello Board (open to public view): Project Tracker

4.1.1 Project Management

Trello was used for our project management. The possible alternatives were GitHub issues, GitHub project board, JIRA. The reasons for this decision are:

1. Familiarity with Trello
2. Trello provided a user-friendly GUI (such as drag-and-drop features) and flexibility in organising our story board, as compared to Github Issues which had to follow a fixed layout.
3. It provides useful add-ons which allows us to add more information on each card and link of cards (such as effort/card, parent-child epics and stories etc)



Product Backlog on Trello

4.2 Implementation and Review

Our team made use of the Feature Branch Workflow when it came to version control, where all feature development took place inside dedicated branches instead of the main branch. This made it easier for the team to work on a particular feature without disturbing the main codebase. Any feature that was to be merged into the main branch had to have a pull request created for it beforehand, during which other members of the team would do a peer review of the code, before merging it into the main branch.

5. Application Design

5.1 Design Decisions

5.1.1 Frontend

For the frontend, our team used React, Redux, MaterialUI and ESLint. The possible alternatives for these are Angular, Vue.js, Ant Design, Recoil. The reasons for the decision are:

1. The Virtual DOM used by React allows for fast rendering of the UI, as well as supports declarative UI which increases productivity and efficiency of the team while developing the frontend.
2. React is the most popular among the three frameworks and has the largest developer community, making it easier to find resources and solutions to commonly faced problems while developing.
3. Reusable components and modularity increases the ease of development and maintenance.
4. Our entire team has prior experience with React and hence can immediately begin development without learning overhead and are able to better share ideas and help each other.
5. Redux is one of the more popular state management tools and has well-written documentation relative to other frameworks allowing us to easily research solutions to blockers faced and pick it up quickly.
6. We used MaterialUI as it provided good documentation (with project templates) and fit our desired design theme for the UI.
7. MaterialUI has clean interfaces and pre-built components. The clean UI provided by MaterialUI also increases our app's usability, which is one of our key attributes. The pre-built components also save us development time.

5.1.2 Backend

For the backend microservices, our team used Node.js, Express.js. The possible alternatives for these were Django, Ruby on Rails. The reasons for this decision are:

1. Numerous NPM packages are available for use, making the development process easier.
2. Use of Javascript across both the frontend and backend since the team is the most familiar with Javascript for application development.
3. Ease of setting up multiple microservices with Express with minimal configuration as compared to the other frameworks.

For our user management and authentication requirements, we used Firebase authentication for the following reasons:

1. Built-in email and password based authentication increases development speed.
2. Handles and implements security best-practices for us such as hashing/salting passwords under the hood.

5.1.3 Database

For our databases, we used Firebase real-time database (for real-time code editor), Firebase cloud database, MongoDB and SQLite3 across the different microservices. The other possible alternatives were PostgreSQL, MySQL.

For MongoDB:

1. Based on our quality attribute prioritisation, scalability is one of the higher ranked quality attributes. MongoDB atlas provides easy functionality to induce multi-direction scaling and allows useful real-time performance monitoring tools in-built, which allows us to adapt to increasing scale.
2. MongoDB Atlas allows for quick setup and more time dedicated to development. MongoDB also has high availability, which is one of our key quality attributes.
3. Document schemas are flexible and easy to change with changes in business requirements.
4. Since performance is one of our key quality attributes, we chose MongoDB which has high performance due to information stored in a single document, without requiring join operations.

SQLite3 is used for the Matching Service:

1. Lightweight database which is easy to use embedded in the Matching Service.
2. Fast reads and writes which are essential to the time-sensitive nature of Matching Service. This ensures performance, which is one of our key quality attributes (refer to Section 3.2.3 NFR 2.1 for performance comparisons).
3. A high capacity database is not required in the case of Matching Service since the unmatched Match objects are deleted when they expire.

Firebase real-time database is used for the Collaborative coding editor:

1. Since performance is one of our key quality attributes, we chose firebase real-time database which provides Real Time capabilities (receive updates within milliseconds) to provide low-latency code editing between our users.
2. Firebase real-time database was also selected since it abstracted away the complexity of writing networking code and allowed us to focus on improving the application in other ways.

5.1.4 Messaging

Messaging is implemented in the application using Socket.io and RabbitMQ. The possible alternatives were Apache Kafka and Redis.

For Socket.io:

1. Allows bidirectional event-based communication between the frontend and microservices.
2. The rooms in Socket.io make it easier for CollabService to communicate with the users currently in the same collaboration room.

For RabbitMQ:

1. Easy to set up - can be set up using Docker.
2. User-friendly interface to monitor RabbitMQ server from web browser.

5.1.5 Orchestration Service

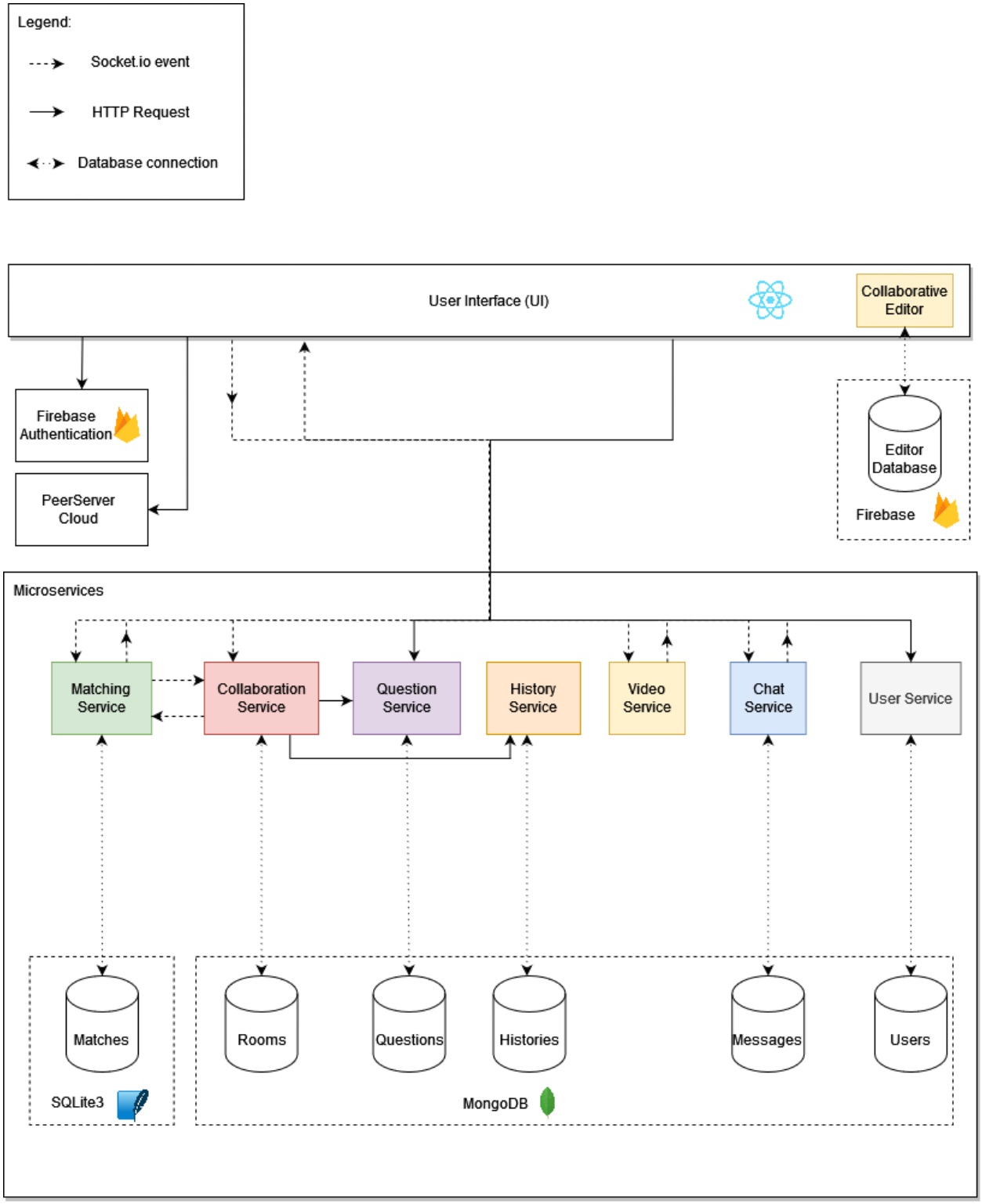
For local orchestration, we used Docker Compose. An alternative would have been Kubernetes. The reasons for choosing Docker Compose are:

1. Familiarity of team with Docker Compose as compared to Kubernetes.
2. Ease of setting up Docker Compose as compared to Kubernetes.

5.2 Tech Stack

Type	Technologies Used
Frontend	• React • Redux • MaterialUI • Firebase Auth • Firepad • Monaco Editor • PeerJS • ESLint
Backend	• Node.js • Express.js • Firebase Admin SDK
Database	• MongoDB • SQLite3 • Firebase
Messaging	• Socket.io • RabbitMQ
Orchestration	• Docker Compose

5.3 Overall Application Architecture



Overall Architecture Diagram

Our application uses a monolithic frontend, and implemented the backend using microservices.

We decided to use a monolithic frontend, due to the small sized nature of our project's frontend. Using micro-frontends could pose several challenges for a smaller sized frontend, such as making it more challenging to set up. We might also incur unnecessary overhead from the communication across the micro-frontends.

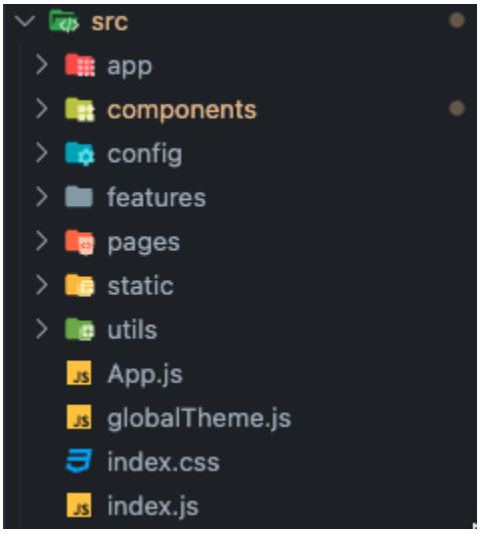
A monolithic frontend on the other hand, is easier to set up, allowing the team to focus more on developing the frontend features. There is less overhead from direct method calls in the monolithic frontend. While it could pose a challenge in collaboration, we did not face any issues due to proper task allocation such that there would be no clashes.

A microservice backend architecture allows us to split the backend functionality into microservices by the different bounded contexts.

Breaking the backend down into the different microservices makes it easier for the team to build and maintain each microservice. Each service can be individually developed, managed and deployed, and use different technologies. The smaller size of each microservices also makes it easier to debug and find out where the issue is present in the codebase.

Each microservice can be scaled individually depending on the load on each microservice, without having to scale the entire backend. Modifying a microservice does not require the redeployment of the entire backend, and each microservice can be deployed independently.

5.4 Frontend Architecture



Src folder of Frontend

As stated in Section 5.3, we implemented a monolithic frontend instead of a micro-frontend. We arranged our frontend code as follows:

1. app folder contains store.js, which combines the different reducers and creates the store for Redux.
2. components folder contains the components and styling used in the frontend.
3. config folder contains configuration files
4. features folder contains the initial state and reducers for the different features in our application.
5. pages folder contains the files for the different pages.
6. static folder contains the static files used in our application.
7. utils folder contains some of the utility code used in different parts of our frontend (e.g. constants and event handlers).
8. App.js contains the main component in the React application, which acts as a container for all other components.
9. globalTheme.js contains our global themes used in conjunction with Material UI.

Our team considered implementing the Atomic Design methodology when it came to implementing our design system. However, we ultimately decided against this as the scale of our frontend is not very large, and most of our components are only used once. However, if the project was of a larger scale, Atomic Design would have been a useful framework for modularizing the reusable components.

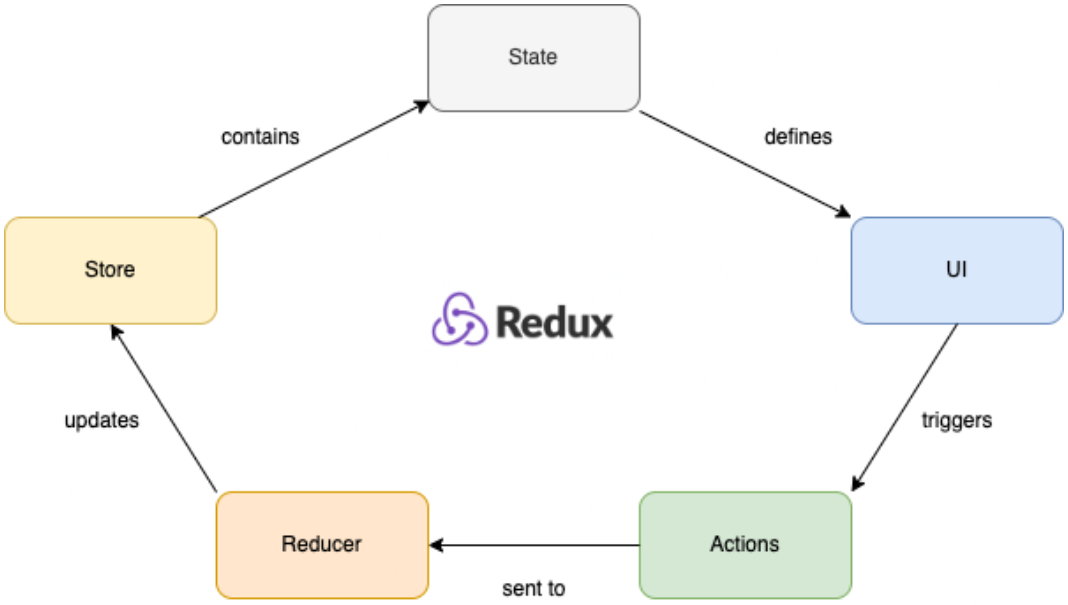
5.4.1 State

The following are the components that make up the state of our frontend application.

1. User state is defined in userSlice.js and handles the following information:
    a. username - name that is shown to other users during the practice sessions.
    b. userId - ID that uniquely identifies the user.
2. Match state is defined in matchSlice.js and handles the following information:
    a. roomId - ID that uniquely identifies the room that the user is in upon finding a match.

3. Session state is defined in sessionSlice.js and handles the following information:
    a. questionId - ID of the current question that is displayed in the room
    b. difficulty - Current difficulty level of questions in the room
    c. questions - The list of questions that have been attempted so far in the room

5.4.2 Redux Architecture



Redux Architecture Diagram

The use of Redux allows us to use the same pieces of application state across multiple components without the added complexity of passing them around as props to nested components (prop-drilling). A frontend monolith architecture of our size would have been increasingly complex to develop and debug if we only made use of prop-drilling.

Furthermore, making use of React and Redux allowed us to closely follow the **Model, View and Controller (MVC)** pattern in our frontend, where the React components (View) are only responsible for presenting the state from the Store, all logic to handle changes in the data will be in the Reducer (Controller), making changes to the Store (Model). This allows us to have a cleaner codebase and also have a **Separation of Concerns** when it comes to separating our business logic from our UI code.

This is also a form of **Shared-Data Pattern**, where multiple components access data through a common data store.

This allows us to have a single source of truth where all the data is centralised and synchronised. Whenever data in the store is modified via actions and reducers, the modification is accessible to all components, which can then update their views accordingly without the fear of data inconsistency. Moreover, this approach also avoids the unnecessary re-rendering of nested components due to changes in props that do not actually concern them, but which they pass to their child components. This optimises performance.

However, one of the drawbacks of Redux is that it adds a large amount of boilerplate code, which can potentially lead to unnecessary complexity in our codebase. In order to partially mitigate this, we made use of Redux Toolkit, which has less boilerplate code and makes the writing of actions and reducers much easier.

5.5 Backend Architecture

We adopted a microservices architecture for our backend. Inspired by a **Domain Driven Design**, we assigned each of our services (User, Match, Collaboration, Question, Chat, Video, History) to solve the needs of a specific subdomain. (eg. Match service is in charge of matching users, Question service is in charge of generating algorithm questions).

Our motivation for using this architecture is that it allows for high cohesion and loose coupling between the logical separations of our application, which is crucial for parallel development, ease of testing and maintenance. This is achieved by ensuring each microservice abides closely to the **Single Responsibility Principle**, and is only required to change if there is a change in its corresponding subdomain's requirements. We also ensured a **Separation of Concerns** between each microservice so as to minimise code changes due to a change in other microservices.

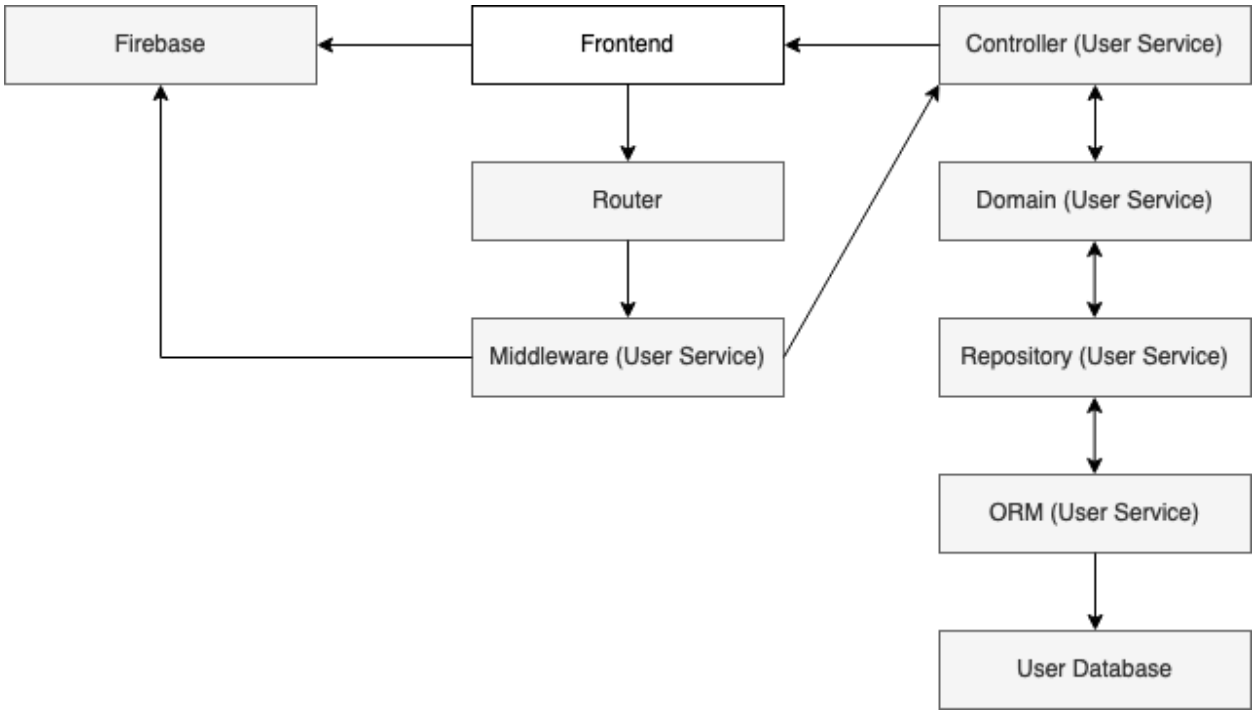
More importantly, the modularisation of our application results in high availability (a crash in one service has minimal impact on the overall application) and high scalability (each service can be scaled independently to meet the needs of our users), which we believe are vital quality attributes of AlgoHike, as illustrated in Section 3.2.2.

Below illustrates a more in-depth view of each of our services' architectures and how our services interact with one another.

5.5.1 User Service and Authentication

The User Service is in charge of fetching and handling user information from the User Database.

Architecture



User Service Architecture

Algohike makes use of Firebase Authentication SDK to handle the authentication of our users. This was our preferred choice over implementing our own custom user authentication due to the following reasons:

1. Security and stability: Firebase is built on Google's infrastructure; the hard work of building out secure user storage, email/phone verification etc is done reliably by using Firebase Auth as an auth as a service.
2. Usability: Firebase easily allows the implementation of different methods of signing in. This leaves up with many options to consider in the long-term.

After authenticating, the relevant information will be passed onto the middleware in the User Service, which verifies the token with Firebase Auth once again. Once the verification is successful, the different layers in the User Service will communicate with each other, eventually making queries to our user database. Once all the operations have been executed, the controller will then send a successful or unsuccessful response to the frontend.

User Service listens to the following endpoints:

1) POST <hostname>/api/user/createUser:

Creates a user in the database

2) POST *<hostname>/api/user/login:*

Verifies the token that is passed and returns the user's data if successful

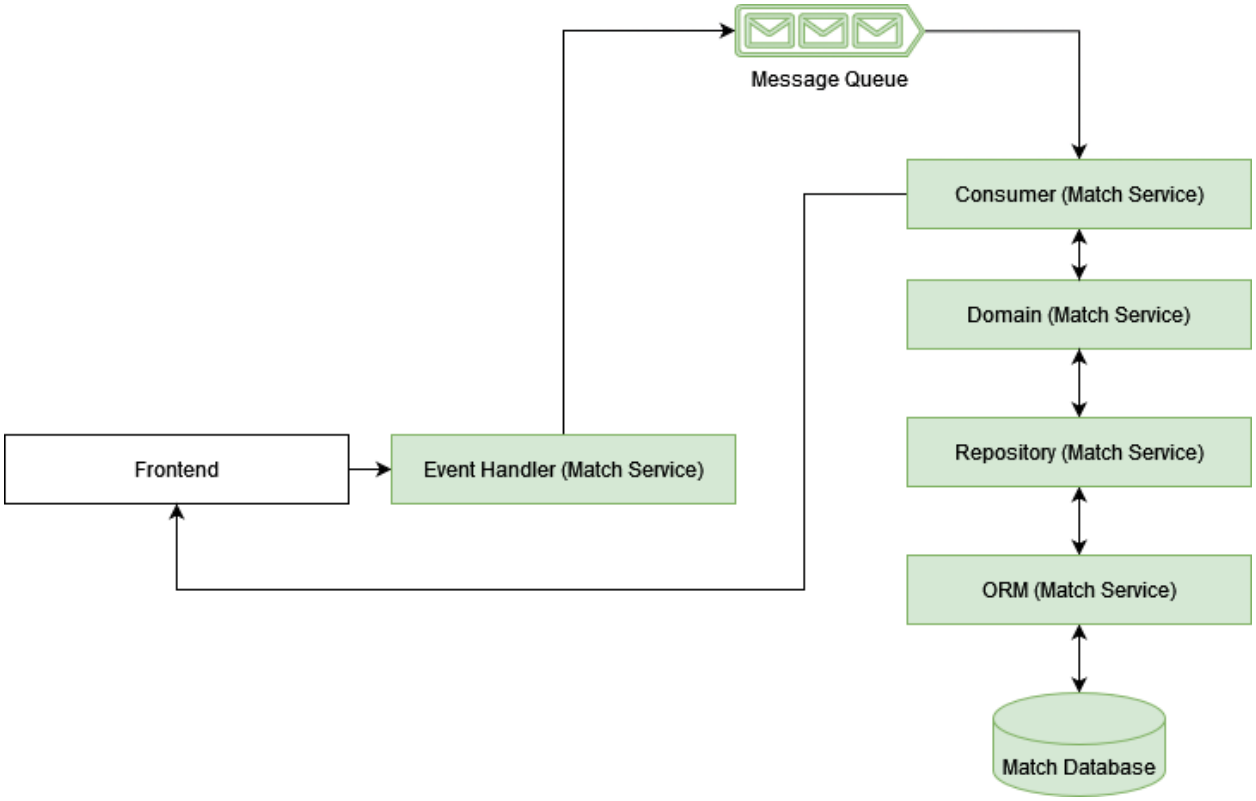
3) *POST <hostname>/api/user/uniqueUsername:*

Checks if the username passed in already exists in the database

5.5.2 Match Service

The Match Service serves to match users who are looking for a partner with another user looking for a match with the same difficulty. When the MatchService finds a match, it sends both users the room ID of the created session.

Architecture



Match Service Architecture

The frontend emits events to the Event Handler in the MatchService. Upon receiving the event, the Event Handler acts as a producer and passes the messages to the message queue. The consumer consumes a message from the message queue, then finds a match by interacting with the MatchService database through the layers indicated in the architecture diagram. When a match is found, the match details are emitted to the frontend directly.

When the frontend emits an event, the producer will produce a message to insert into the request queue. The consumer consumes these messages. For MatchEvent.FIND messages, the consumer will either find a suitable match, and if one is not found, the consumer will create a new Match object in the database. For MatchEvent.CANCEL messages, the consumer will delete the Match object from the database.

When the client disconnects, the producer produces a disconnect message into the request queue. The client consumes this message by deleting the Match object in the database which has the socketId of the recently disconnected user.

For both types of events, the consumer will respond to the client directly.

Match Service Design Decisions

Usage of Web Sockets over HTTP

Web Sockets allow for bidirectional communication between the client and the server by reusing the same communication channel. We can have lower latency communication between the MatchService and the client, upon user requests, so the user can receive faster responses, giving the user a better matchmaking user experience. It also allows implementation of the Pub-Sub model, where the MatchService is subscribed to the MatchEvent.FIND and MatchEvent.CANCEL events from connected clients.

Usage of Message Queue

By using a message queue, we can ensure guaranteed end to end delivery. It is crucial that the request gets processed, so messaging queues help us ensure that all the user's requests get processed. We also can split the responsibilities of communicating to the client and processing the requests and the logic for finding matches into two separate entities, which are independent of each other and communicate with each other through the message queue.

The message queue also acts as a data buffer. If there is suddenly a spike in the number of user requests, the consumer does not need to be aware of the spike. The message queues will buffer the messages for the consumer, and the consumers just have to process the messages one by one. This prevents the consumers from being overloaded with data, making the architecture more robust, which is in line with the quality attribute of robustness (as mentioned in Section 3.2.2).

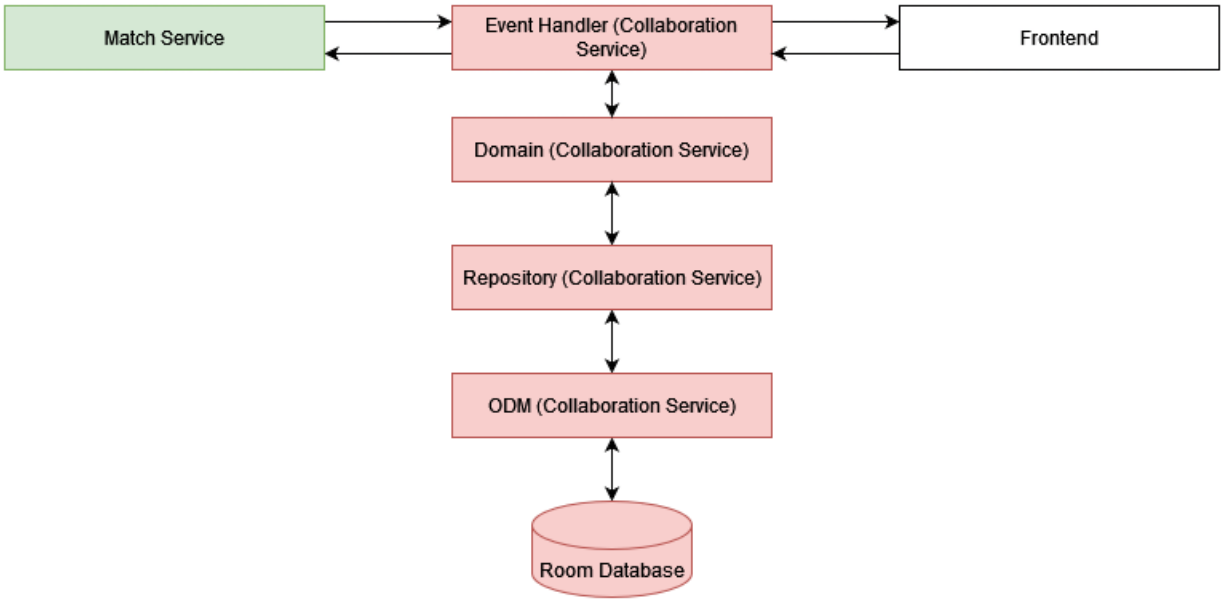
Usage of priority message queue

It is important that the message produced by the producer corresponding to the MatchEvent.CANCEL event must have a higher priority than the MatchEvent.FIND event. Suppose a user is waiting for a match, and the user's Match object is in the database, and a second user emits the MatchEvent.FIND event. Then, while the MatchEvent.FIND message is in the message queue, the first user cancels the match. The MatchEvent.CANCEL message should reach the consumer first, so that the Match object gets deleted from the database and the two users do not get matched. Therefore, we use a priority message queue.

5.5.3 Collaboration Service

The CollabService is responsible for handling the collaboration rooms that store the information about currently running user sessions.

Architecture



Collaboration Service Architecture

For each event that is emitted to the CollabService from either MatchService or frontend, the event handlers (createSessionHandler, joinSessionHandler, updateSessionQuestionHandler, disconnectHandler) handle the respective events and then emit the response back to the client socket.

The event handlers invoke domain layer methods which in turn invoke repository layer methods which interact with the database to read, create, update or delete Room objects from the MongoDB database.

Event Handlers:

1) *createSessionHandler* for SessionEvent.CREATE

Creates a new room for the two users who were just matched. Refer to Section 5.7.3 for more details.

2) *joinSessionHandler* for SessionEvent.JOIN

Handles the event of users joining the room. Refer to Section 5.7.3 for more details.

3) *updateSessionQuestionHandler* for SessionEvent.UPDATE_QUESTION

Handles the event of users requesting for the next question in the room. Refer to Section 5.7.4 for more details.

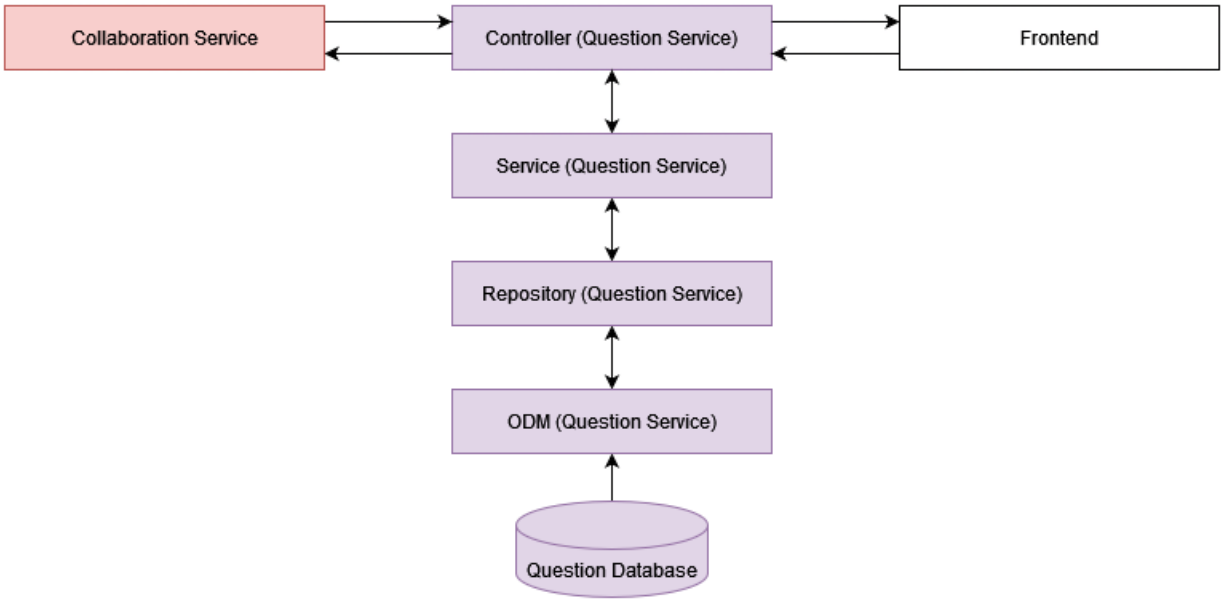
4) *disconnectHandler* for SessionEvent.LEAVE

Handles the event of users leaving the collaboration room. Refer to Section 5.7.7 for more details.

5.5.4 Question Service

The Question Service is in charge of providing a session with questions for users to conduct the interview practice.

Architecture



Question Service Architecture

Question Service listens to the following endpoints:

1) *GET <hostname>/api/question/getQuestion:*

Returns a randomly generated question retrieved from the database.

2) GET *<hostname>/api/question/getAllQuestions:*

Returns the id, title and difficulty of all questions in our database.

3) *POST <hostname>/api/question/getQuestionById:*

Retrieves the question with id specified by the “questionId” parameter. Returns null if the question does not exist in the database. Used by the frontend to retrieve the question content upon receiving a questionId from the CollabService.

4) *POST <hostname>/api/question/getQuestionByDifficulty:*

Retrieves a random question with difficulty specified by the “difficulty” parameter. “difficulty” must be one of “EASY”, “MEDIUM” or “HARD”. Used by CollabService upon creating a new session.

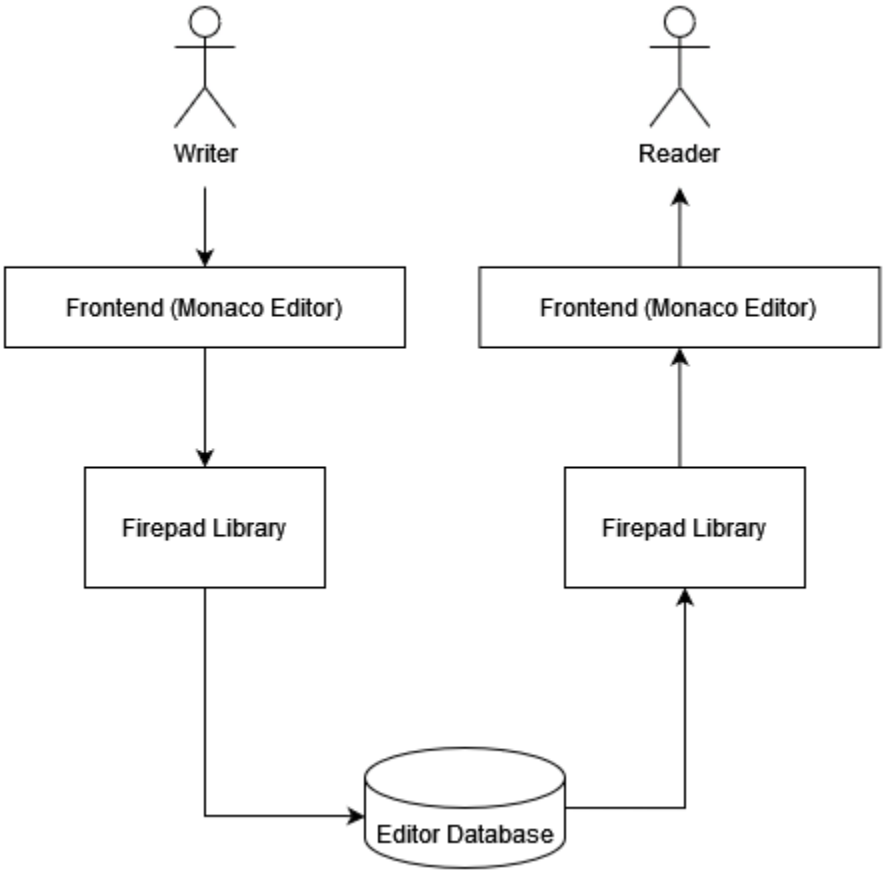
5) *POST <hostname>/api/question/getQuestionWithBlacklist:*

Takes in a “list” of integers and “difficulty”. Returns a random question with the corresponding “difficulty” and id not inside the “list”. “difficulty” must be one of “EASY”, “MEDIUM” or “HARD”. Used by CollabService when a session updates the question.

5.5.5 Editor Service

The editor service provides a real-time code editor for matched users to write code and see changes made to the editor in real-time.

Architecture



Editor Service Architecture

As seen in the architecture diagram, Firepad provides functionality that abstracts the interactions with the Firebase Realtime Database and provides an interface for the Monaco Editor React component to read and write to it.
Hence, this is the main benefit of using firepad, much of the implementation, apart from connecting the components in the frontend, is handled for us and allows us to achieve the desired functionality and meet our requirements without having to handle the implementation and maintenance work for interacting with a real-time database.

Editor Service Design Decisions

For the implementation, we were deciding between using a third-party solution vs implementing our own editor.

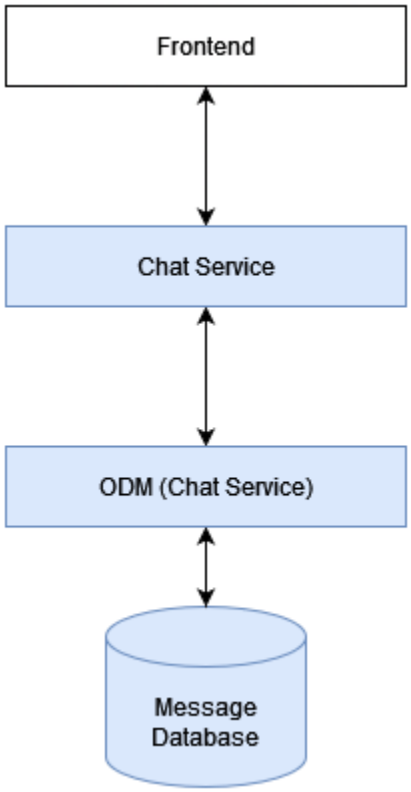
Ultimately, we decided to use a third-party open-source editor for the following reasons:

1. The solution is open-sourced and hence we can customise it to our needs in the future if requirements change. (which is the main benefit of a self-implemented solution)
2. Much of the complexity has been implemented and this allowed us to focus on improving the platform over spending effort developing the editor.
3. It is customizable and we were able to choose our editor of choice, MonacoEditor (which is used in VSCode software)
4. Firepad has no server dependencies and this allows for increased application maintainability as we do not have to maintain a server in the future.
5. We can rely on cloud services such as Firebase Realtime Database to handle the real-time editor updates for us.

5.5.6 Chat Service

The chat messaging service is a microservice which handles the responsibility of enabling matched users to send chat messages to each other during their mock interview session.

Architecture



Chat Service Architecture

The chat service has a relatively simple architecture. We chose to make use of socket.io in order to maintain a bidirectional and event-based communications channel between the chat service server and the frontend client as it handled the complexity of handling real-time event-based communication between the client and server.

A socket connection is established between the server and each chat room frontend client, allowing to keep track of activity participants of each chat room in real time and by listening for events e.g, join room, disconnect events in the event a user leaves the chatroom.

The architecture/application flow for interactions between components is described here (5.7.4 Sending a chat message).

5.5.7 Video Service

The Video Service handles the video chat functionality, enabling live communication through voice and video.

Architecture

Video Service and related components

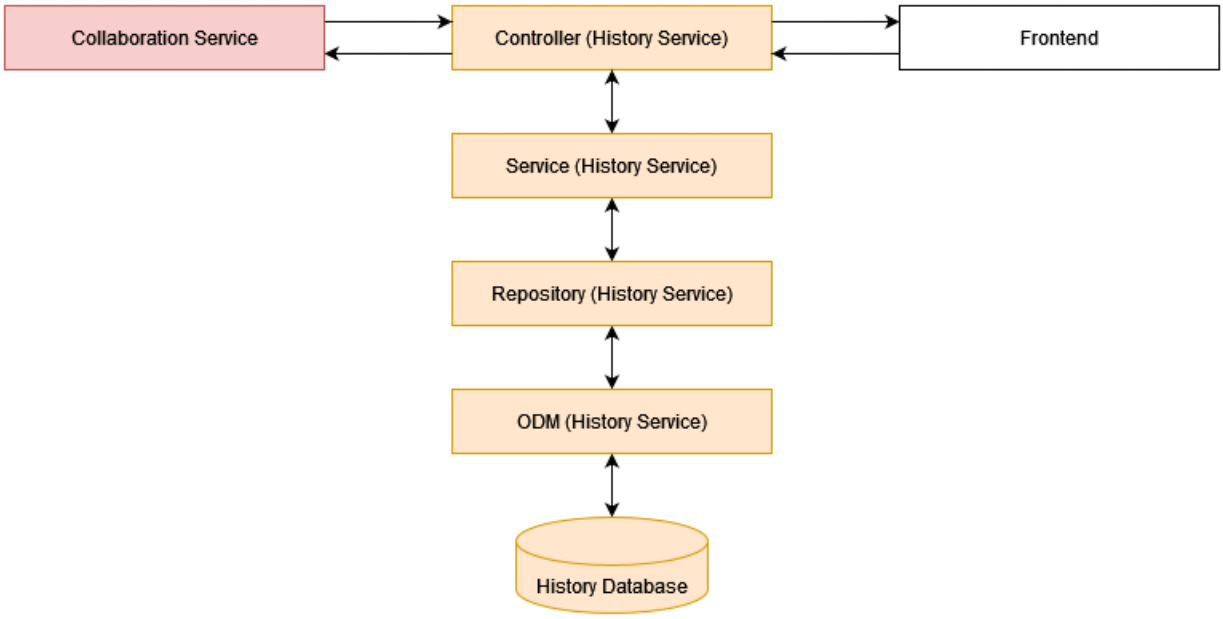
The Video Service has one of the simplest architectures among the microservices as all it does is simply expose a Socket.io endpoint which listens for the VideoEvent.JOIN and VideoEvent.LEAVE events.

The PeerServer Cloud service is a free cloud-hosted version of the PeerServer, which the frontend connects to in order to create a peer instance, which can connect to other peer instances and listen for connections.

5.5.8 History Service

The History Service is in charge of storing a user's session histories so that he/she can revisit questions that he/she has done before.

Architecture



History Service Architecture

History Service listens to the following endpoints:

1) *POST <hostname>/api/history/getHistory:*

Retrieves the session histories of the user with the specified "uid".

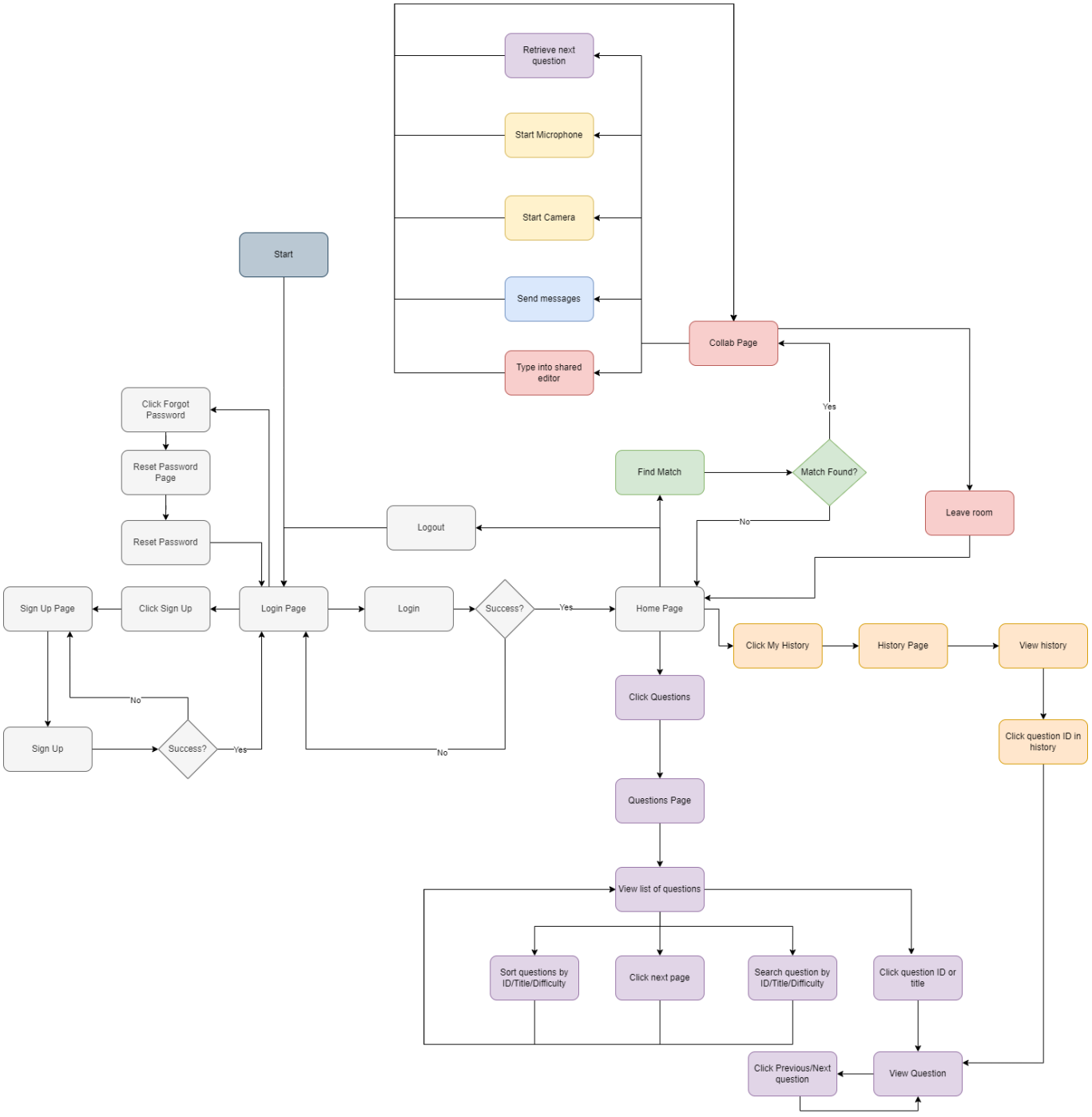
2) *POST <hostname>/api/history/addHistory:*

Creates a new match record with uid1, uid2, question ids, roomId and difficulty. Used by the CollabService upon creating a new session.

3) *POST <hostname>/api/history/updateHistoryByRoomId:*

Takes in a "roomId" and a "qid" and adds the qid to the question list of the session. Used by the CollabService after it retrieves a new question as requested by the user.

5.6 User Flow



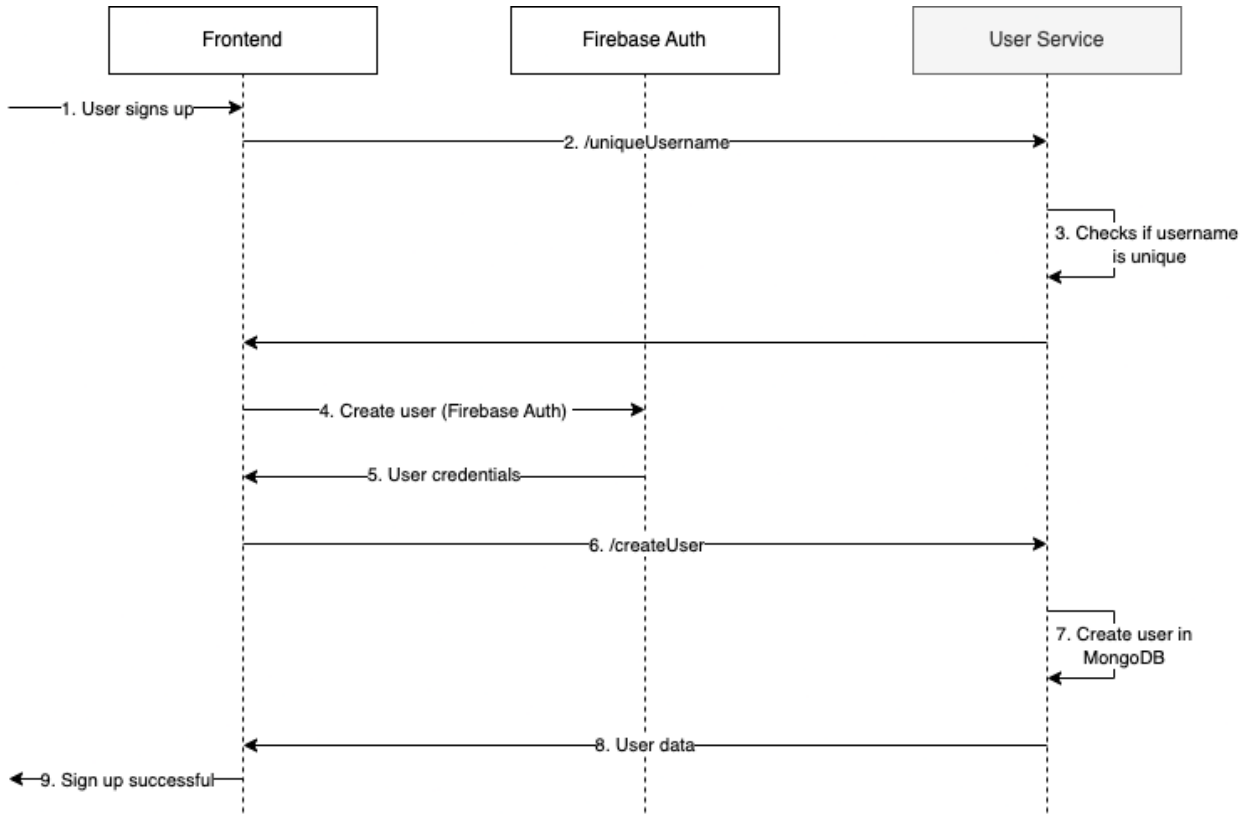
User Flow Diagram

5.7 Application Flow

5.7.1 Sign Up and Sign In

The following occurs when a user signs up/signs in:

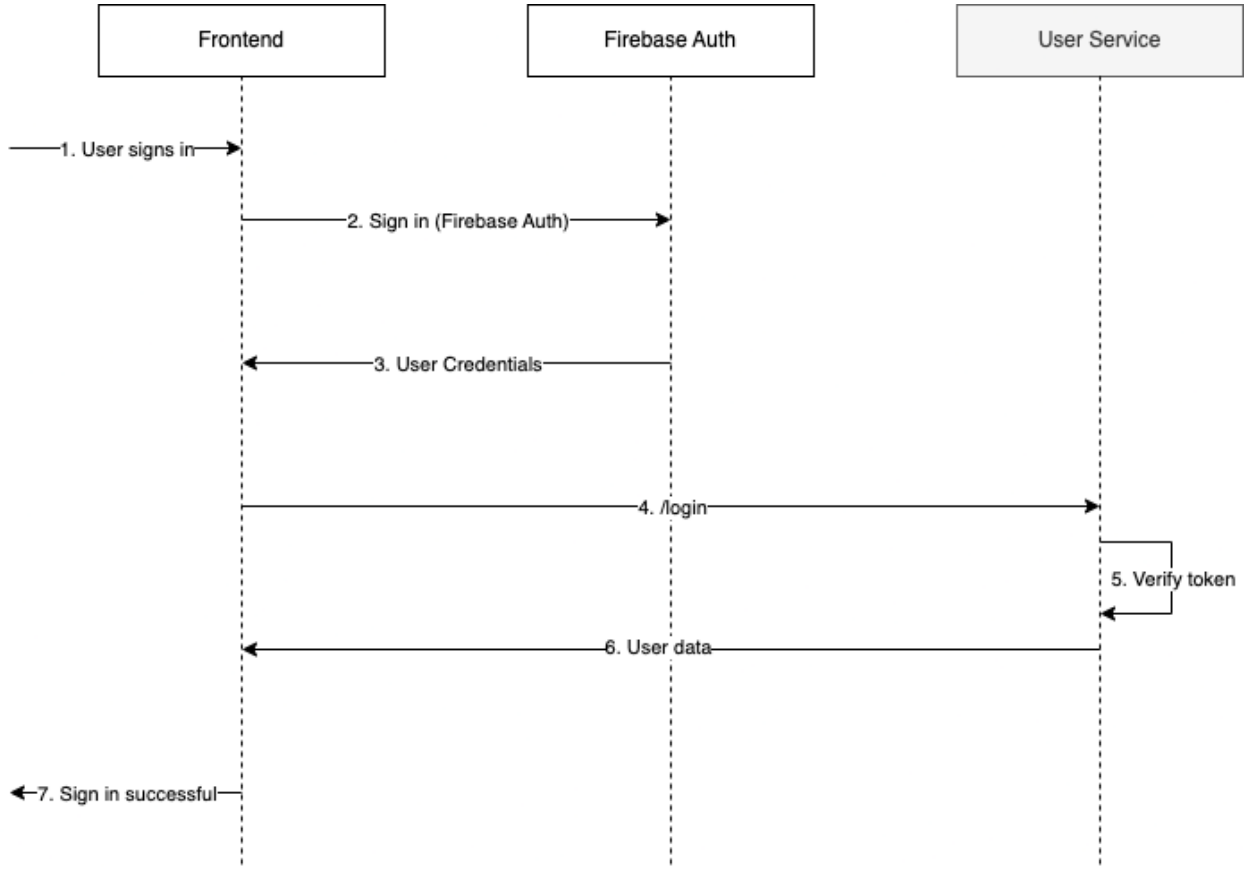
Signing up



Sequence diagram for signing up

1. User clicks on the “SIGN UP” button and passes input validation.
2. Request is sent to the /uniqueUsername endpoint to check if the new username already exists
3. User Service checks whether a user with the same username already exists in the database
4. User is created in Firebase Authentication
5. Firebase Authentication returns the user credentials
6. Request is sent to the /createUser endpoint
7. User Service creates a new entry for the user in MongoDB
8. User data is returned to the frontend
9. Sign up is successful

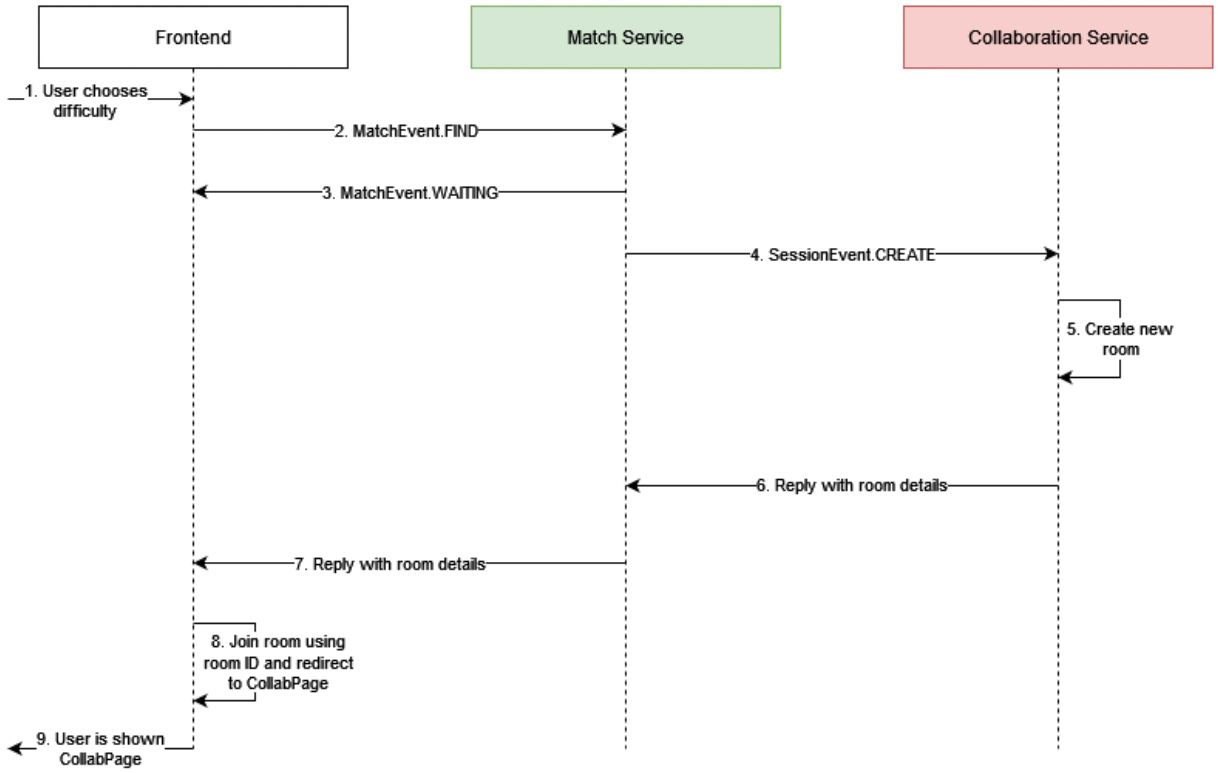
Signing in



Sequence diagram for signing in

1. User signs in from the frontend
2. Email and password input is passed to Firebase Authentication to sign in
3. Firebase Authentication returns the user credentials
4. Request is sent to the /login endpoint, with the token retrieved from firebase passed in the request header
5. Token is verified using the Firebase Admin SDK
6. User data is passed to the frontend
7. Sign in is successful

5.7.2 Finding a match



Sequence Diagram for finding a match

The following sequence occurs when the user chooses a difficulty:

1. User chooses difficulty on the frontend.
2. Frontend emits MatchEvent.FIND to MatchService.
3. MatchService emits MatchEvent.WAITING back to the frontend.

If a match is found by Match Service after Step 3, the following occur:

4. MatchService emits SessionEvent.CREATE to the CollabService.
5. CollabService creates the room which will store the session details. Details on how CollabService creates the room are listed in Section 5.7.3.
6. CollabService replies to Match Service with the room details.
7. The Match Service then checks whether both users are still connected, if both are still connected then the Match Service then sends the frontend the room ID to the frontend. Otherwise Match Service tries to find another match for the connected user.
8. The front end will join the room using the room ID.
9. User is shown CollabPage and is inside the collaborative room with the other matched user

Scenario 1 (No suitable Match object is found)

If there is no suitable Match object, then a new Match object is created in the database, with the attributes userId, difficulty and socketId. The consumer will then emit a MatchEvent.WAITING to the user who is

searching for a match. The consumer also begins a cron job which will delete the Match object from the database if the item is not removed from the database in 30 seconds.

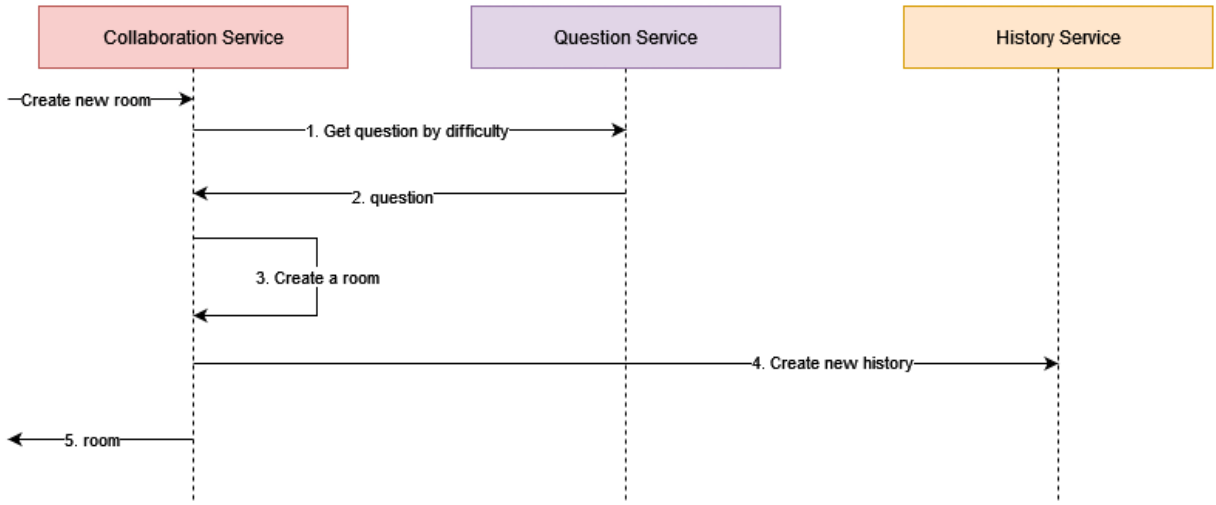
Scenario 2 (A suitable Match object is found in the database)
If the Matching Service finds a suitable Match object for the user searching for a match, the cron job that is set to remove the Match object from the database is cancelled, and the Match object is deleted from the database. A check is done to see if the original user identified by the userId in the Match object is still connected to the Matching Service. If the original user is not connected, then the process of finding a match is repeated.

If the original user is still connected, Matching Service emits SessionEvent.CREATE event to the CollabService with the payload containing userId1, userId2 and difficulty. The CollabService then sends Question Service a request to the getQuestionByDifficulty endpoint for a question with the specified difficulty. CollabService then creates the Room object containing the userId1, userId2, difficulty, questionId and questions attributes. When the collaboration room is created, a cron job is created which will delete the collaboration room in one hour if no users join the collaboration room. Once the collaboration room has been created for users to join, the CollabService then replies the Matching Service with the room details, most importantly the roomId, which is used by the frontend to join the collaboration room. Finally, Matching Service emits the MatchEvent.FOUND event to the two users who were matched using their socket IDs.

5.7.3 Creating and joining a Collaboration Room

After a match is found, the MatchService emits the room details including the room ID to the frontend The frontend then uses this room ID and emits SessionEvent.JOIN to CollabService. When the users join the collaboration room, the cron job which is set to delete the room in one hour is cancelled. The CollabService uses Socket.io rooms to add the two users into a Socket.io room identified by the roomId. This allows all users in the Socket.io room to get updated when there are changes to the state of the collaboration room.

Room Creation



Sequence Diagram for room creation

When CollabService needs to create a room, the following occur:

1. CollabService obtains a question from Question Service with the specified room's difficulty.
2. The Question Service responds to the request with a question.
3. CollabService creates a new Room with the two user IDs, room ID and question ID.
4. CollabService sends a request to History Service to add a new history.
5. CollabService sends the front-end the newly created room's details.

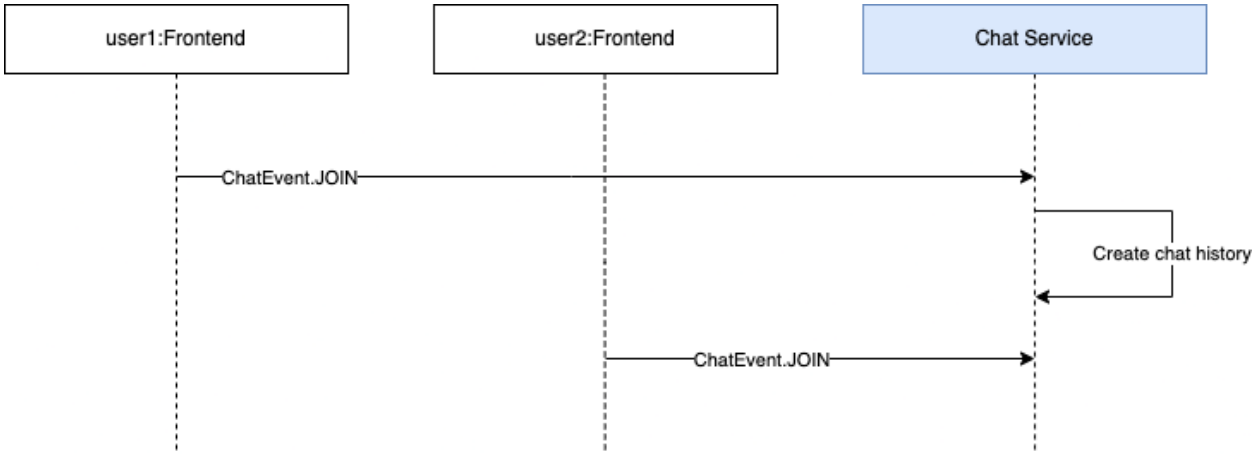
5.7.4 Update Question

When one of the users sends a request to update the question displayed in the collaboration room question, the frontend emits a SessionEvent:UPDATE_QUESTION event to the CollabService, with the roomId and difficulty of the room as the payload. The CollabService sends questions list stored in the Room object to the getQuestionByBlacklist endpoint in Question Service, so that a question that has not been encountered is set as the new question for the collaboration room. The CollaborationService then updates the questionId of the room and emits the updated room details to both users through the Socket.io room.

5.7.5 Joining a chatroom and sending a chat message

The interactions involved with sending a chat message are explained below.

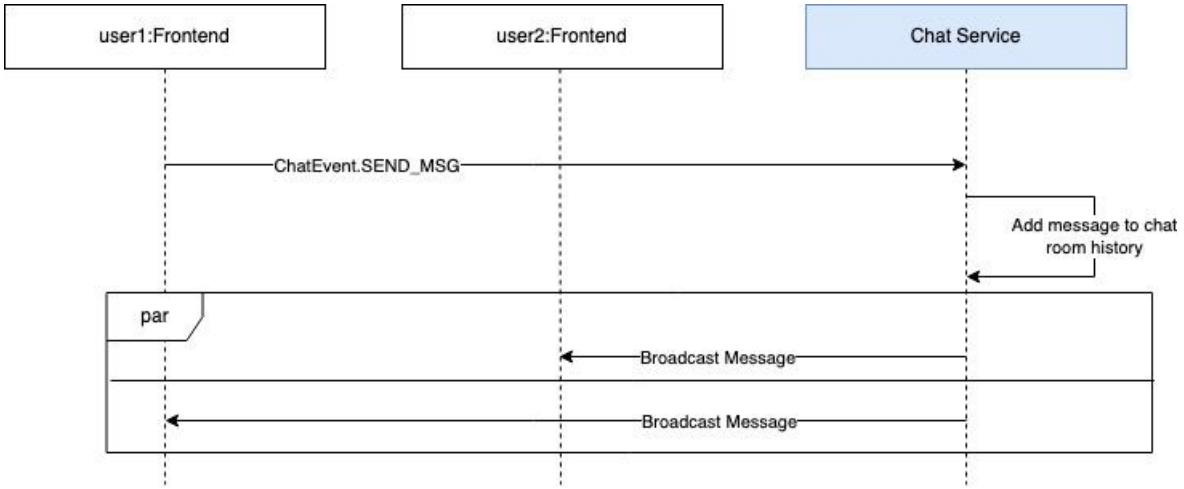
Firstly, when a pair of matched users joins a new room:



Sequence Diagram for joining chat room

1. Both user frontend clients send ChatEvent.JOIN events to the chat messaging server via socket events.
2. A chat room is created in the chat messaging service in Socket.io
3. Both frontend clients will share the same roomId generated by the match service, this is used to create a unique room in Socket.io for the 2 users.
4. If this roomId is new and no message history exists in the database, then a new collection for the roomId is created in the database to persist the latest 50 messages for the chat session.

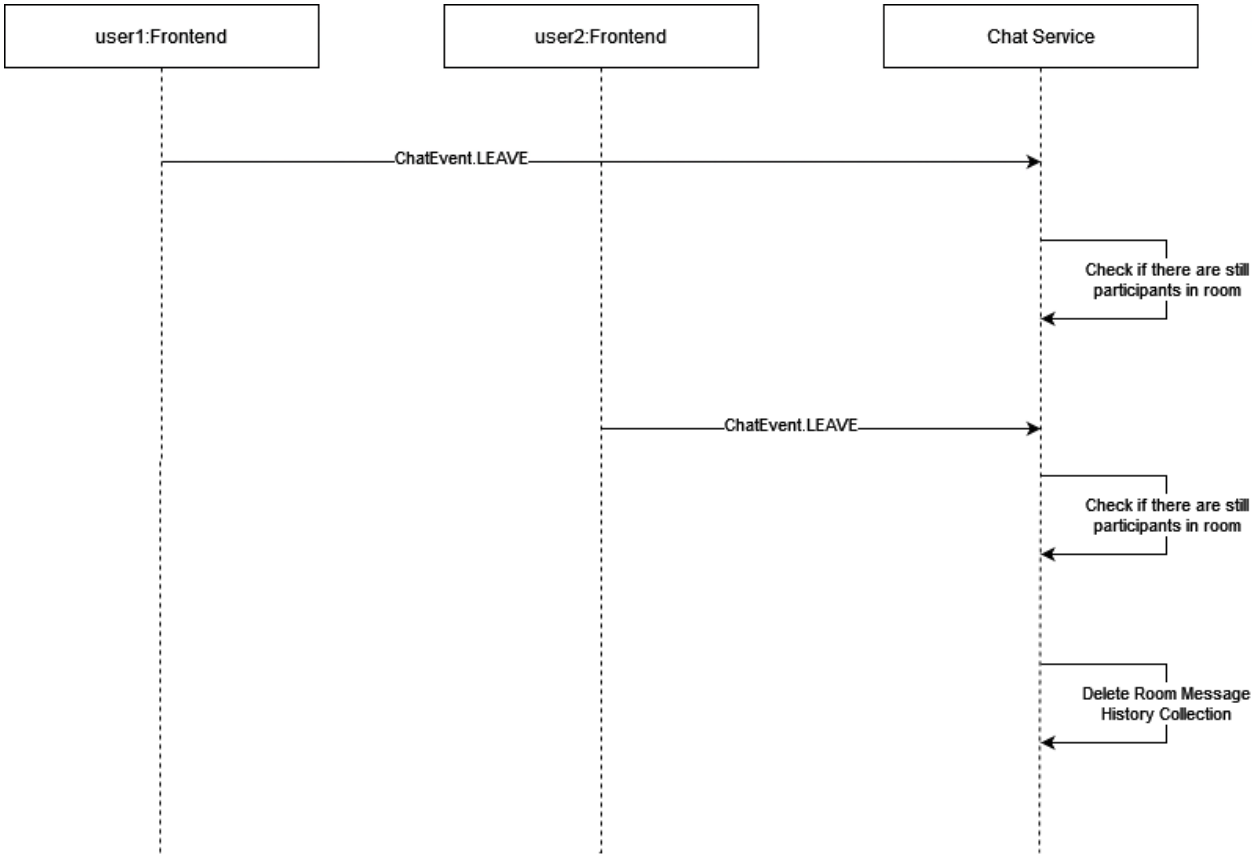
Next, when a user sends a message to the room:



Sequence Diagram for sending a chat message

1. A socket event named ChatEvent.SEND_MSG is sent by the sender client and received by the chat messaging server
2. The chat messaging server then broadcasts to all sockets listening to the socket room (with the same roomId) and allows the clients in the room to receive the message details.
3. At the same time, it also saves the latest message to the database collection for the room to allow for persistence of messages if user's leave and join the room.

When users leave the room and cleaning up:



Sequence Diagram for user leaving chat room

1. Whenever a user leaves the room, the chat messaging server checks the number of remaining participants in the room.
2. Once there are no more participants in the room, it invokes a cleanup operation to remove the room collection and its records from the message history database.

5.7.6 Communicating through Video and Voice Chat

Upon entering a room, the frontend obtains the necessary permission from the user's web browser in order to access his/her webcam and microphone. Once this is successfully done, it emits a VideoEvent.JOIN event to the VideoService, passing along the roomId and userId as arguments. Once the other party joins the room, the VideoService emits a VideoEvent.CONNECTED event to the room, passing along the userId of the party who just joined. A peer is created and connected to the PeerServer Cloud service. A call is then established with the other party using the PeerJS library and the peer instance created earlier, which passes the video/audio stream of the user and also receives the video/audio stream of the other party. When one party leaves the room, the frontend emits a VideoEvent.LEAVE event to the VideoService, which in turns emits a VideoEvent.DISCONNECTED event to the other party.

5.7.7 Leave Collaboration Room

When a user leaves the collaboration room, the frontend emits the SessionEvent.LEAVE event to the CollabService. If the leaving user is the last user in the collaboration room, then a cron job is started, which will delete the Room object from the database in one hour.

6. Extensions

6.1 Features

These are some of the possible features we are considering to implement in the future.

6.1.1 Code Execution

Our code editor currently does not allow code to be run. One possible extension is to allow this functionality, so that users can run their own test cases and check if their solution is correct. Furthermore, we could also then include a wide range of test cases for each problem which users can run their code against. Passing all these test cases would mean that a user has successfully solved the problem.

This will result in a better user experience as users can submit their solutions and guarantee that they've provided the correct/wrong solution.

6.1.2 Leaderboard

A leaderboard feature will enable users to see how they rank among the pool of application users. A user can climb the leaderboard by accumulating points. Points can be claimed once users have successfully solved a question (as mentioned above). Solving questions of higher difficulty/solving questions in less time rewards more points.

Leaderboards are a design element of gamification and can improve productivity, learning and retain user engagement as an element of competition and achievement is introduced. (*How to Improve Engagement With Leaderboards in Gamification?*, 2020)

6.1.3 Friend System

Our last feature is to implement a friend system. This allows users to add other users as friends; this can be other users that they matched previously or other users that they find via usernames. Users can then see if their friends are online and easily match with them with a request. Furthermore, they can also see where their friends are ranked in the global leaderboard and also filter the leaderboard to show only their friends in order to find out where they rank among their friends.

6.2 Deployment and CI/CD

To make our application available for the general public, we can deploy it on a cloud provider such as Amazon Web Services (AWS). A rough overview of how the deployment architecture is as follows:

- Elastic Container Registry (ECR) to store the Docker images of the different microservices.
- Elastic Container Service (ECS) to manage, deploy and scale the Docker containers.
- Application Load Balancer (ALB) to accept incoming traffic from clients and route requests to the microservices.
- Elastic Computing Cloud (EC2), which is a virtual server where the running containers of the ECS services are instantiated.
- Route 53, which is a highly available and scalable Domain Name System (DNS) web service which connects user requests to the ALB.
- Certificate Manager to provision, manage and deploy public and private SSL/TLS certificates, allowing our website links to be accessible via a secure HTTPS connection.
- Virtual Private Cloud (VPC), which allows us to launch our resources into a virtual network.
- A public and private subnet for resources accessible and not accessible from the internet, respectively.

To speed up our deployment and development process, we can also provision a continuous integration and continuous deployment (CI/CD) pipeline using Github Actions. The workflow will be triggered whenever we merge changes to the main branch at the end of each sprint cycle.

7. Reflection

7.1 Challenges

One of the challenges we faced was the planning of the overall software architecture and how our different microservices should interact with one another. This was mainly because none of our team members had a lot of experience designing software architecture and we were also new to the concept and implementation of microservices architecture. As a result, we had to do a great deal of research and ask for advice from our tutors in order to get a feasible architecture that resulted in a functional application. We also faced some challenges with planning our project timeline in the early phases of the project, as we did not fully understand the scale of some of the features that we had set out to build.

7.2 Key Takeaways

7.2.1 Project management

One of the key takeaways we learnt was the importance of project management in our software projects. Since software teams are usually composed of multiple members, it is important to ensure that everyone is on the same page and is constantly updated on what is happening.
One of the key things we learnt was helpful in improving efficiency between developers is the adoption of an Agile practice such as Scrum which we used in our project.

For example, we followed Scrum cycles every week and created an overall product backlog before assigning sprint backlogs to members for the week (using Trello). Using such a backlog allowed our team members to be able to know what the objectives in the week were and the expected effort assigned to each team member.

Additionally, we practised retrospectives where we talked about the challenges we faced and how we can improve in subsequent sprints. For example, we realised in one of the earlier weeks that we needed a lot of time to implement services such as the user service from scratch in one of the earlier weeks. Through our retrospectives, we were able to brainstorm for a better solution which was to research and adopt the use of 3rd party services such as Firebase Authentication & Firepad etc.

7.2.2 Introduction to new technologies

This application exposed our team to using sockets in our application with Socket.io and RabbitMQ. We learnt how to use sockets in our application for bidirectional communication between the frontend and the different microservices, such as ChatService, MatchService and CollabService.

Our team also discovered authentication tools such as Firebase authentication which has built-in authentication functionality. In creating the UserService portion of the application, we were exposed to how to use Firebase authentication in our application.

Doing this project also led us to use an orchestration service such as Docker Compose to manage the large number of microservices. We learnt how to set up Docker Compose files and Dockerfiles and using docker-compose for our container orchestration for faster start up of our application during development.

We also were exposed to niche technologies such as Firepad, Monaco Editor and PeerJS which are the collaborative editor and the video and voice chat implemented in the VideoService respectively. Using these technologies allowed us to learn how to implement them into the frontend and backend services respectively.

7.2.3 Benefits of working on a microservices architecture

During the first 2 sprints, our team only worked on the user service, match service and the frontend. Since more than 1 team member worked on each service at the same time, there was often significant overlap in the development process. This resulted in bottlenecks where we had to wait on each other before starting on our part.

In subsequent sprints, we decided to each work on separate services, and this sped up the development process significantly. A key takeaway we took away as a result was the importance of parallel development, and how we could split the workload by bounded contexts when using a microservices architecture to optimise the software development process.

8. Appendix

8.1 Glossary

Term	Definition
power user	User of computers, software and other electronic devices, who uses advanced features which are not used by the average user.
MatchService	MatchService
MatchEvent.FIND	Name of the socket event emitted by the frontend to MatchService when a user selects a difficulty to find a match.
MatchEvent.CANCEL	Name of the socket event emitted by the frontend to MatchService when a user cancels the search for a match.
MatchEvent.WAITING	Name of the socket event emitted by the MatchService to frontend when it has received the request to find a match.
MatchEvent.FOUND	Name of the socket event emitted by the MatchService to frontend when it has found a match.
Match object	A database object which is stored in the MatchService database. An object represents a user's request for a match, and stores the user ID, socket ID, and difficulty.
CollabService	Collaboration Service
SessionEvent.CREATE	Name of the socket event emitted by the MatchService to CollabService to create a new room in the CollabService.
SessionEvent.JOIN	Name of the socket event emitted by the frontend to MatchService when a user joins the room.
SessionEvent.UPDATE_QUESTION	Name of the socket event emitted by the frontend to MatchService when a user wants to get the next question while in the room.
SessionEvent.LEAVE	Name of the socket event emitted by the frontend to MatchService when a user leaves the room.

Room object	A database object stored in the CollabService database. An object represents a session that is ongoing, and stores the user IDs, current question ID, past question IDs.
VideoService	Video Service
VideoEvent.LEAVE	Name of the socket event emitted by the frontend to the VideoService when a user leaves the room.
VideoEvent.JOIN	Name of the socket event emitted by the frontend to the VideoService when a user enters the room.
VideoEvent.DISCONNECT	Name of the socket event emitted by the VideoService to the frontend when a user disconnects from the room.
VideoEvent.CONNECT	Name of the socket event emitted by the VideoService to the frontend when a user connects to the room.
ChatService	Chat Service
ChatEvent.LEAVE	Name of the socket event emitted by the frontend to the ChatService when a user leaves the room.
ChatEvent.JOIN	Name of the socket event emitted by the frontend to the ChatService when a user enters the room.
ChatEvent.LOAD_HISTORY	Name of the socket event emitted by the ChatService to the frontend when a user joins a room. It loads the history of the previous messages in the chat whenever a user rejoins his previous room.
ChatEvent.SEND_MSG	Name of the socket event emitted by the frontend to the ChatService when one party sends a chat message.
ChatEvent.RECEIVE_MSG	Name of the socket event emitted by the ChatService to the frontend to let a user receive a message that has been sent into the room.

8.2 Environment Variables

For the purposes of grading, obtain the environment variables needed to run the project here.

9. References

Agile development from a programmer's perspective | by Dasith Kuruppu | Level Up Coding. (2019, January 2). Level Up Coding. Retrieved November 8, 2022, from https://levelup.gitconnected.com/agile-from-a-developers-perspective-27b23ea665f0

How to improve engagement with leaderboards in gamification? (2020, March 4). Grendel Games. Retrieved November 8, 2022, from https://grendelgames.com/how-to-improve-engagement-with-leaderboards-in-gamification/

Reliability | MongoDB. (n.d.). MongoDB. Retrieved November 7, 2022, from https://www.mongodb.com/cloud/atlas/reliability

Service Level Agreement for Hosting and Realtime Database | Firebase. (2020, April 9). Firebase. Retrieved November 7, 2022, from https://firebase.google.com/terms/service-level-agreement

SQLite Database Speed Comparison. (2014, April 1). SQLite. Retrieved November 7, 2022, from https://www.sqlite.org/speed.html