



NUS
National University
of Singapore

CS3219 AY 2022/2023

Software Engineering Principles and Patterns

Group 3 Final Report: PeerPrep

Group Members: Lim Rui Xiong, Chang Rui Feng, Lim Junxue, Soh Xin Wei

Website: <https://frontend-xkpqea35pq-as.a.run.app>

Code Repository: <https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g3>

Table of Contents

1. Contributions	2
1.1 Individual Contributions	2
1.2 Must Haves Achieved and Good to have targeted	3
2. Introduction	4
2.1 Background and purpose of project	4
3. Functional Requirements	4
4. Non-Functional Requirements (NFRs)	6
5. Software Development Process	7
6. Application Design	8
6.1 Development Tech Stack	8
6.2 Overall Architecture Diagram	9
6.3 Microservice Diagram	10
6.4 Database Schemas and Design	11
6.5 User Flow of Application	12
6.6 Frontend Design	13
6.6.1 Overall Design	13
6.6.2 Form Validation	13
6.6.3 Pagination	14
6.6.4 Flow for Authentication and JWT	14
6.6.5 Authentication and Cookie	15
6.7 User Service	16
6.8 History Service	16
6.9 Matching/Collaboration/Chat Service	18
7. Testing	21
8. Deployment	21
9. Existing bugs	24
10. Suggestions for improvements and enhancements to the delivered application	24
10.1 Timer Feature	24
10.2 Friends list	24
10.3 Video Call	24
10.4 Admin Role	25
11. Reflection and learning points for the project process	25

1. Contributions

1.1 Individual Contributions

Rui Xiong	Code <ul style="list-style-type: none">- F3.1, F3.2, F4.1, F4.2- Setup CI pipeline for Github Actions unit test cases.- Integration of Matching/Collaboration/Chat service with frontend- Set up MongoDB database Report <ul style="list-style-type: none">- [1.2] Targets- [3] Functional Requirements- [4] Non-Functional Requirements- [5] Developmental Process- [5.2] Overall Architecture- [5.5] User Flow- [6.9] Matching/Chat/Collab Service- [7] Testing- [9] Existing bugs
Rui Feng	Code <ul style="list-style-type: none">- Frontend components for all pages associated with the FRs- Integration of User service with frontend- UI design and implementation of app- Private routes authentication Report <ul style="list-style-type: none">- [6.6] Frontend Design- [10] Suggestions for Improvements- [11] Reflections and Learning
Jun Xue	Code <ul style="list-style-type: none">- F3.3, 3.4, F5, F6- Integration of Question and History service with frontend- Testing of Question and History service with frontend- Setup CD pipeline for Github

	Actions Report <ul style="list-style-type: none"> - [3] Functional Requirements - [6.3] Microservice Diagram - [6.4] Database Schemas and Design - [6.8] History Service
Xin Wei	Code <ul style="list-style-type: none"> - F1 - Set up authentication for APIs with JWT - Set up redis cache for JWT blacklist - Deployment of services to Google Cloud Run Report <ul style="list-style-type: none"> - [6.7] User Service - [8] Deployment

1.2 Must Haves Achieved and Good to haves targeted

Must have	Good to have
User Service	History Service
Matching/Collab Service	Communication (Chat) Service
Question Service	CI/CD
Basic UI	Fancy UI
Deployment on Local	Deployment on production system

2. Introduction

2.1 Background and purpose of project

PeerPrep is the project we developed to help students prepare and collaborate with other students for technical algorithmic interviews. Students are able to match up with their peers and solve technical questions filtered by the difficulty level chosen.

It allows 1 to 1 concurrent collaboration between 2 students through a shared editor. Students can also send private messages to one another to discuss the problem, before tackling it.

After completing a problem together, the user can also check the history of questions they have attempted before.

3. Functional Requirements

Functional Requirements		
FR	Description	Priority
User Authentication (F1)		
F1.1	The system should allow users to create an account with username and password.	High
F1.2	The system should ensure that every account created has a unique username.	High
F1.3	The system should allow users to log into their accounts by entering their username and password.	High
F1.4	The system should allow users to log out of their account.	High
F1.5	The system should allow users to reset their password if they forget it.	High
Navigation (F2)		
F2.1	The system should allow users to navigate back to the home page.	High

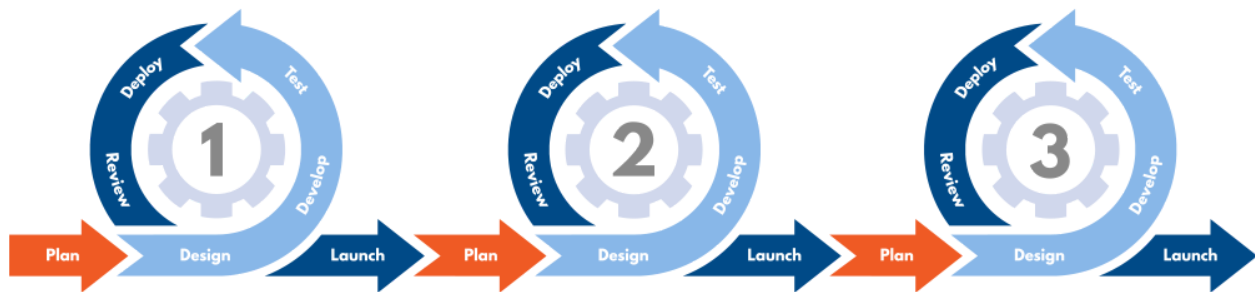
Matching (F3)		
F3.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
F3.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High
F3.3	If there is a valid match, the system should match the users within 30s.	Medium
F3.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	Medium
F3.5	The system should provide a means for the user to leave a room once matched.	Medium
Collaboration/Chat (F4)		
F4.1	The system should allow concurrent editing of the shared space with both users able to see each other's work.	High
F4.2	The system should allow users to send private messages to one another.	High
History (F5)		
F5.1	The system should show users questions they have attempted in the past and the name of their partner.	High
F5.2	The system should retrieve the questions done by a user.	Medium
Question (F6)		
F6.1	The system should provide questions according to the difficulty level requested.	High
F6.2	The system should provide a random question that the user has not attempted before, unless the user has exhausted all other questions in the difficulty level selected.	Medium

4. Non-Functional Requirements (NFRs)

Non-Functional Requirements		
NFR	Description	Priority
Performance (N1)		
N1.1	The system should not appear sluggish, and have any noticeable lag during runtime	High
N1.2	The system should be able to load within a few seconds.	High
Usability (N2)		
N2.1	The system should be viewed properly in different screen sizes	High
N2.2	The system should be easy to navigate between pages.	Medium
N2.3	The system's features should be easy to understand and use.	Medium
Security (N3)		
N3.1	The system should protect user's sensitive information. Passwords should be hashed in database	High
N3.2	The system should authenticate users before calling microservices.	High
N3.3	The system should enable strong password checks when the user signs up or changes their password.	High
Portability (N4)		
F4.1	The system should be able to run in modern web browsers like Google Chrome and Mozilla Firefox.	High
F4.2	The system should be compatible with various operating systems like Windows OS and Mac	High

	OS	
Integrity (N5)		
F5.1	The system should be protected against unauthorized and unauthenticated modification of data.	High

5. Software Development Process



Our team utilized the Agile Development Process while working on our application. Each sprint has a cycle of planning, designing, developing, testing, deployment and review, allowing us to incrementally build the application and also allows for changes and iterations whenever we feel a feature can be improved.

We have Rui Feng working on the frontend, Rui Xiong and Jun Xue working on both the frontend and backend and Xinwei working on the backend. Having 2 people working on both the client and server helps speed up integration testing of our application. We set up weekly sprints and performed scrum meetings every Friday to keep track of tasks, ensuring everyone was aware of the project progress and in sync.

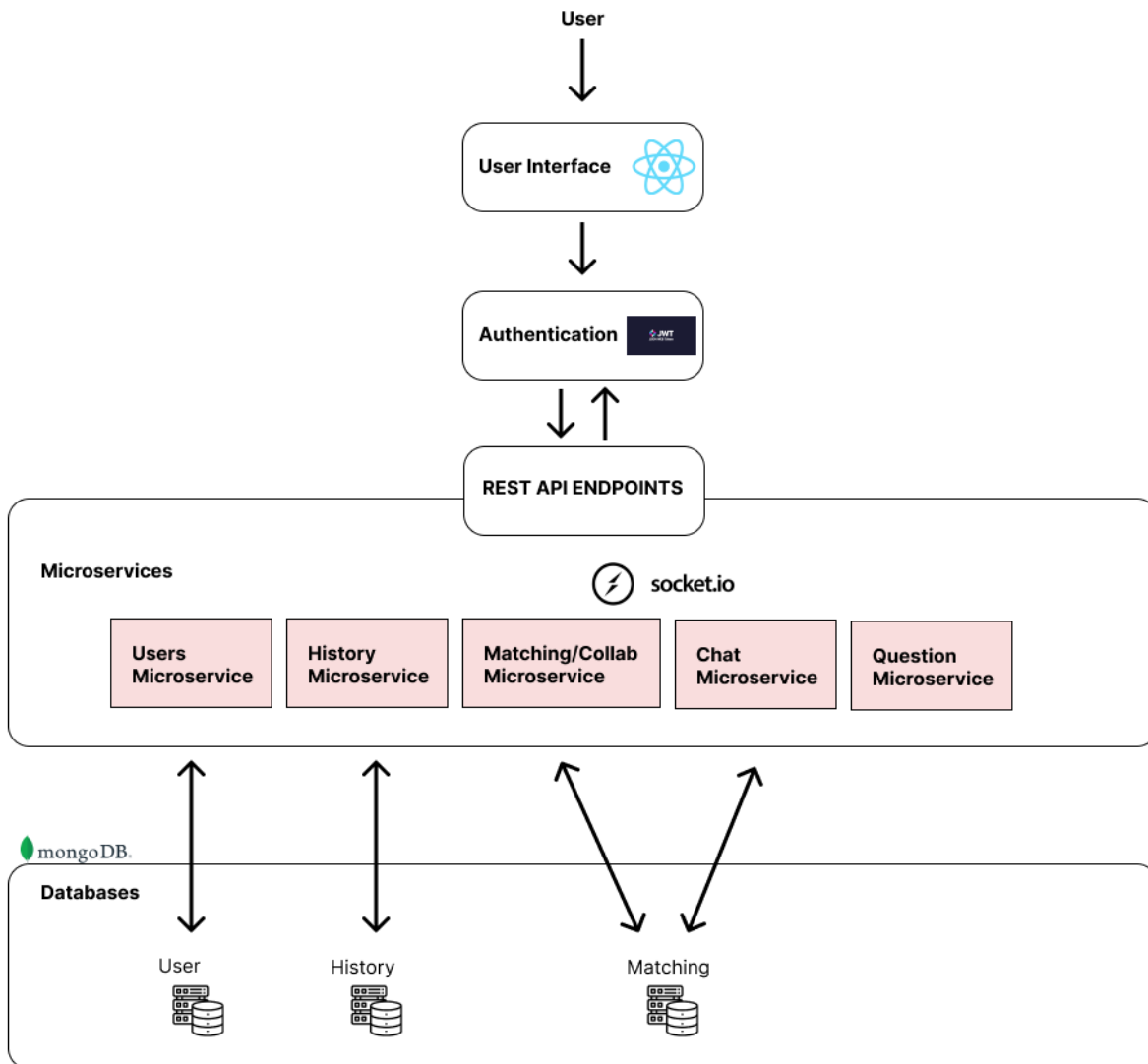
Tasks are allocated according to the priority level of the features in the FRs and NFRs. High priority features are done first, and then subsequently the lower priority ones. We also used Continuous Integration and Continuous Deployment with Github Actions that triggers on merges to the master branch, saving us time from having to manually deploy and perform our unit tests again when we make a change.

6. Application Design

6.1 Development Tech Stack

	Technologies
Frontend	React.js
Backend	Express.js/Node.js
Database	MongoDB
In-memory Database	Redis
Pub-Sub Messaging	Socket.io
Cloud Providers	Google Cloud
CI/CD	Github Actions
Orchestration Service	Docker
Project Management Tools	Github Issues

6.2 Overall Architecture Diagram



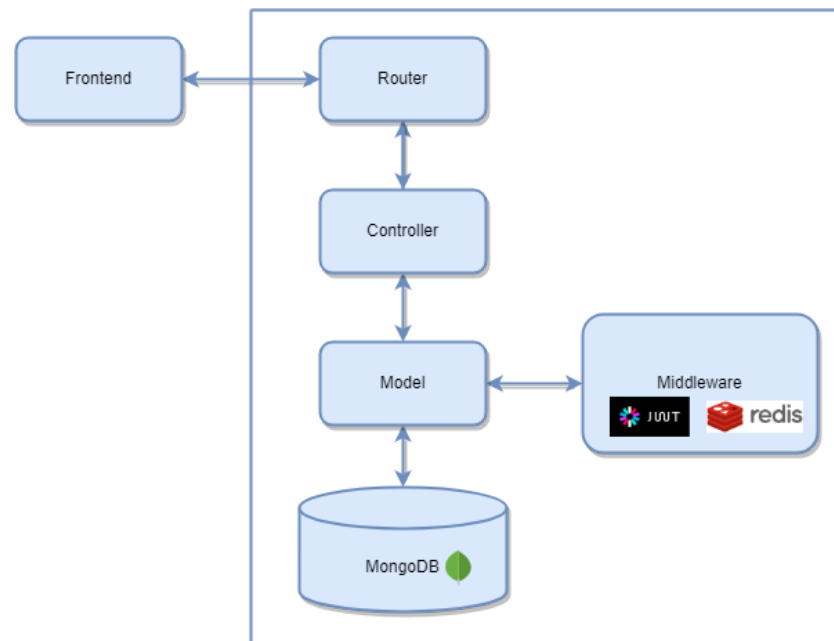
We designed the application using a Monolith Frontend and a Microservices backend after weighing the pros and cons of each architectural design.

Micro-frontends are suitable for large development projects but can hinder and slow down development time for small and medium sized projects. An example is that the visual design style has to stay consistent between the different micro-frontends. Since our application is considered small, we felt a monolith frontend would be easier for developers to collaborate and work on. Hence allowing us to put more focus on developing features and ensuring that our designs remain consistent.

We utilized a Microservice backend architecture as it allows us to separate the different modules into independent services. The benefits to this is that it provides more flexibility, scalability and extensibility than a Monolith Architecture. It also reduces downtime through fault isolation, microservices that are not dependent on another and this allows other services in the system to continue running. The Microservice architecture also allows the team members to individually take charge of a microservice independently, thereby speeding up development.

A Microservice architecture for our backend ultimately reduces tight coupling between the backend components and decreases complexity in testing and deployment, and speeds up our development time.

6.3 Microservice Diagram

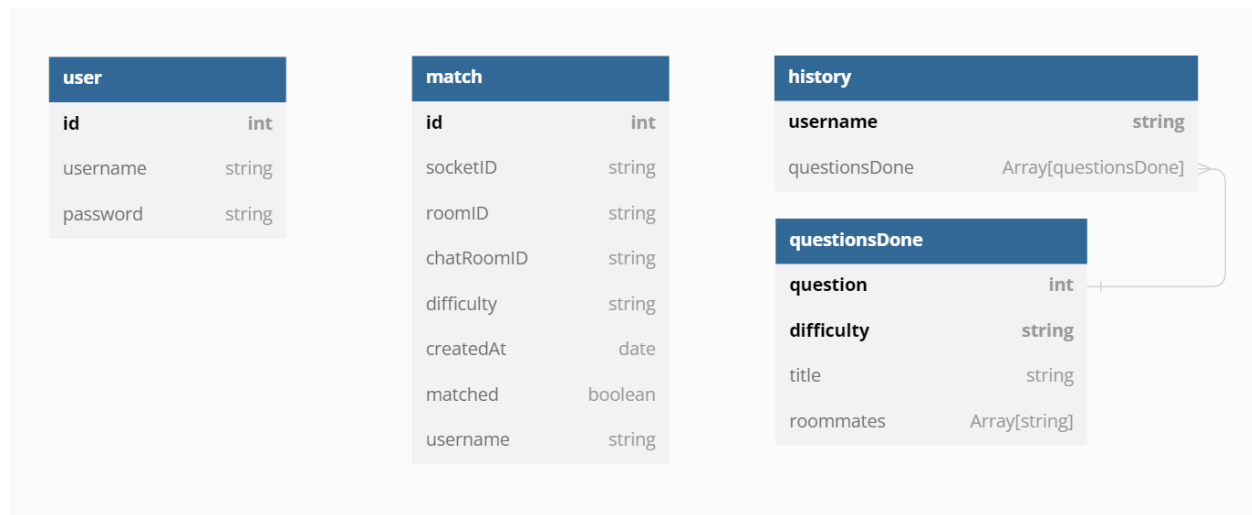


User/Matching/History service

The User/Matching/History services follow the architecture as shown above. When the frontend calls an API of the corresponding service, the router of the service receives it and routes it to the controller which handles the logic of the API. It communicates with the model to carry out persistent data changes. The Object-related Mapping (ORM) functions within the model handle the business logic of the model. Within these functions, the API caller is authenticated by the middleware which leverages on JWT security and Redis for faster caching. At the end of the flow, the data in the model will then be saved in the repository connected to MongoDB, thereby persisting the data of

the relevant service. Data from the repository is pulled by an API call in a similar fashion through the router, controller and model and passed back to the frontend at the top of the stack.

6.4 Database Schemas and Design

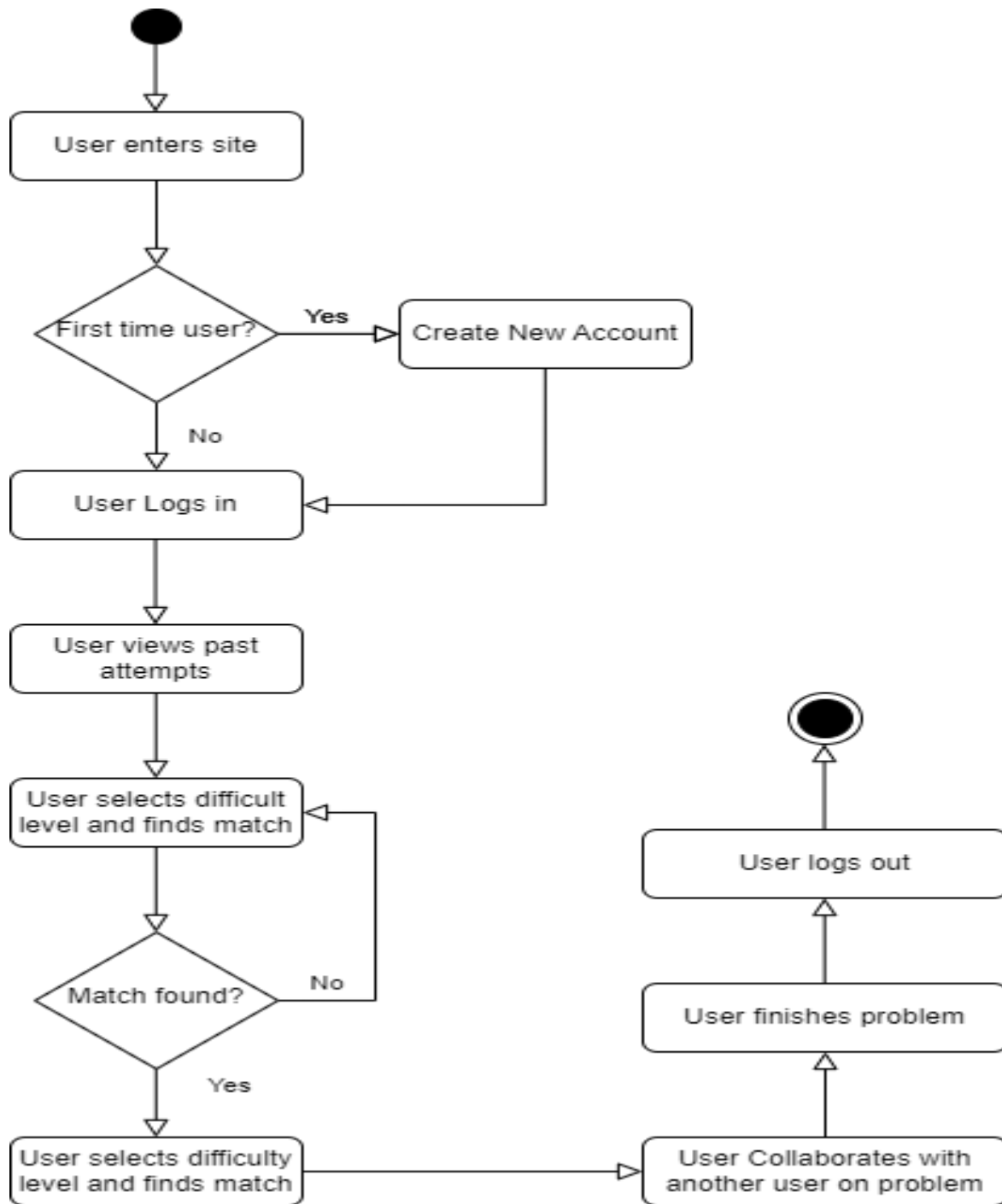


Database Schemas

We have adopted the database-per-Service pattern instead of a shared database. The User, Matching and History services have schemas corresponding to their names as shown above. MongoDB was chosen for us to utilize the fast querying and flexibility of a non-relational database, allowing for storage of diverse types of data. This is optimal as each service does not need relationships between tables and can exploit the fast querying of non-relational databases as they will not have to look through multiple tables compared to a traditional relational database such as MySQL. For example, in MongoDB, an instance/document of history model would keep an array of questionsDone, defined in a separate schema as history. This results in only 1 table of documents to look up. The other schemas do not have relations and thus MongoDB is chosen for its ability to handle unstructured data and speed.

This is also ideal in our microservice architecture as each microservice has its own database, reducing coupling of data within each service. It ensures that data that is related to each service is kept and known only within its bounded context, minimizing responsibility and reducing data pollution, thereby maintaining stricter cohesion and clarity in ownership.

6.5 User Flow of Application



Activity Diagram for User's expected flow of the application

6.6 Frontend Design

6.6.1 Overall Design

For the frontend, we used React as the main library, Material UI (MUI) to style our components, React Router for page navigation, Axios to fetch APIs, Recharts for our bar graph, Socket-io client and React-chat-widget for the chat/collaboration component. We used more of a "VVM" model as we only pulled data directly into components with limited data wrangling of any kind. In terms of state management complexity we did not utilize Redux and its MVC architecture as React Context was more than sufficient to handle our state management needs, though as the app grows in complexity we could start adopting it.

5.6.2 Form Validation

Password

- At least 8 characters
- At least one letter and one number
- At least one special character

Confirm Password

Password creation was done in the frontend as we wanted our backend to just store the password in the database. This was done using regex pattern matching being set in the MUI input field. Confirmation of password was also done in the frontend as this was more for the user to check whether there are any typos

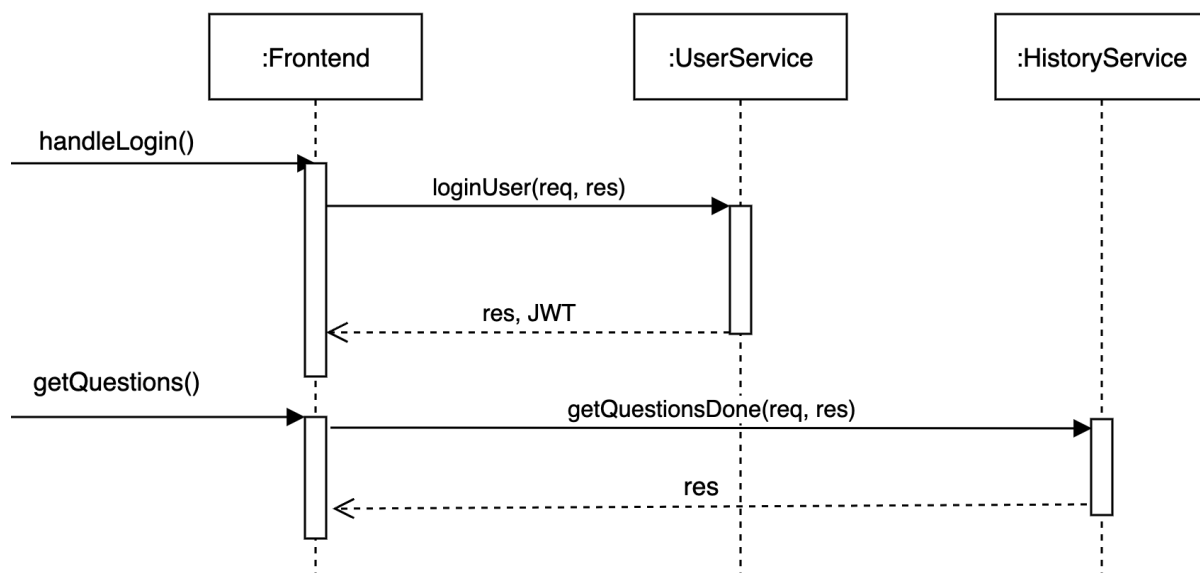
6.6.3 Pagination

Attempted Questions		
Q1	Alice	Easy
Q2	Alice	Easy
Q3	Bob	Easy

< 1 2 >

Pagination was done purely in the frontend as a user's attempted questions would highly unlikely be in the thousands range which did not require complex API call logic to return the specific page or entries. There are potential enhancements to include more parameters for the GET request for pagination to be implemented such as filtering by year or question type as well which can be done in the backend

6.6.4 Flow for Authentication and JWT

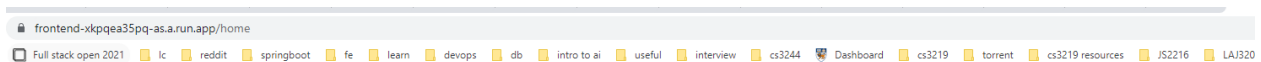


In the sequence diagram, we show a brief overview of the flow for the HistoryService when the getQuestions() API is called. We verify the user's JWT when calling the APIs of the different services, and this JWT is obtained once the user logs in. The JWT is then stored as a cookie (FR1.3) and passed to our various APIs in the request header, which then gets verified in the backend before returning the result.

6.6.5 Authentication and Cookie

```
const ProtectedRoute = ({ children }) => {  
  if (!isLoggedInToken && !window.localStorage.getItem("jwt_token")) {  
    return <Navigate to="/" />;  
  }  
  return children;  
}
```

Apart from authentication for API calls in the backend, private routes had to be set using React Router in order to allow users to view a specific page with a specific URL. This was done using the JWT. This prevents users from being able to access the home matching page without being first logged in and they will be shown a blank page as shown in the screenshot below.



Also, initially React context as global state was used to access the JWT, but we realized we needed persistent storage as users might reload the page causing the JWT to be cleared resulting in them being logged out. So we decided to store JWT in local storage as well to prevent that from happening, and decided global state might be more useful

to store user data information which can be a possible enhancement for all components to access this user data

6.7 User Service

The code structure of this service follows the Microservices architecture diagram under Application Design.



user	
id	int
username	string
password	string

Each user created is stored inside MongoDB as shown above. Instead of storing the password of each user as it is, it is first hashed using the "bcryptjs" Javascript module. This is to prevent leakage of sensitive data in the event of unauthorized access into the MongoDB.

When a user logs in, a JSON Web Token (JWT) is returned in the response body of the HTTP request. The JWT is cached in the user's browser as a cookie and used for authentication of selected APIs within the application. For instance, JWT is required to authenticate retrieval of a user's history from the History Service.

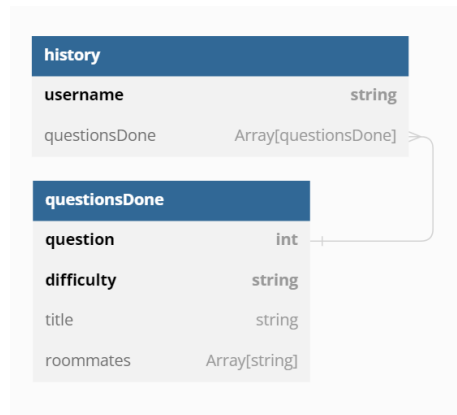
When a user logs out, their JWT is blacklisted. This prevents unauthorized individuals from retrieving other's JWT from cache and using them to access our APIs. JWTs that are blacklisted are stored in both a Redis server and MongoDB. The decision to use the Redis server instead of a database is because it can be used as an in-memory database to facilitate faster retrieval. Fast retrieval is needed since each API that requires authentication will first verify if the JWT provided is in the blacklist. Without the Redis database and with a MongoDB that has many entries, response speed might be slowed and affect user experience. That said, the decision to store the same blacklisted JWT in MongoDB as well is to provide a safety net in the event where the Redis server crashes. The stored blacklist might be lost. In that case, the application attempts to reconnect to the Redis server and populate it with entries from the MongoDB.

We have prioritized the Security (N3) aspect of our Non-Functional Requirements (N3.1, N3.2 and N3.2) during our development work. The User Service and Client works hand in hand in helping us achieve that.

6.8 History Service

The code structure of this service follows the Microservices architecture diagram under Application Design.

The data to be saved in the history of each user is saved within the same document of MongoDB. For example, questions done for a user are saved as shown below.



This is done so to take advantage of the non-relational properties of MongoDB. It would be tedious and slower to perform joins in relational databases. At the same time, currently, the whole history of a user is fetched and used in the frontend so it is easier to retrieve the document from MongoDB.

Another decision made was to separate this history service from the question service. It would have been easy to add this schema to the question service but for cohesion and extensibility, the history service is implemented separately. Therefore, the history of a user can be extended with other types of histories and remain within the bounded context and responsibilities of a history service.

6.9 Matching/Collaboration/Chat Service

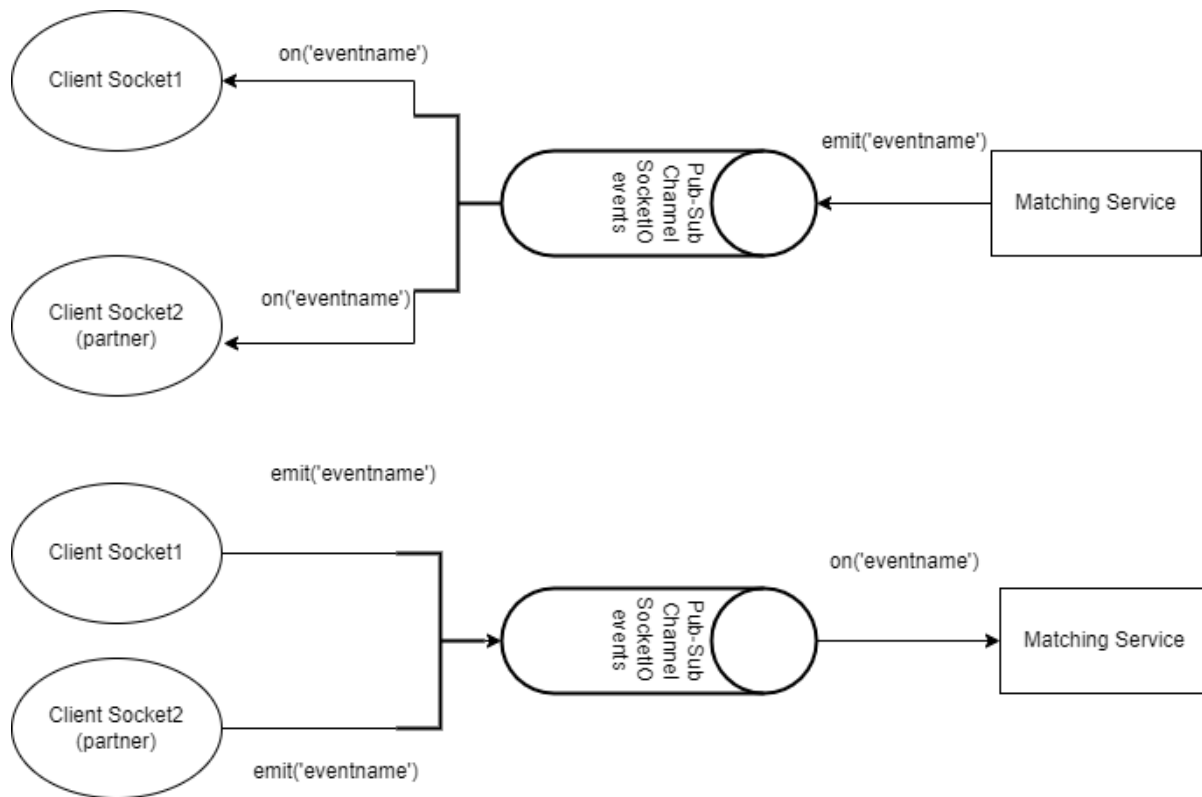
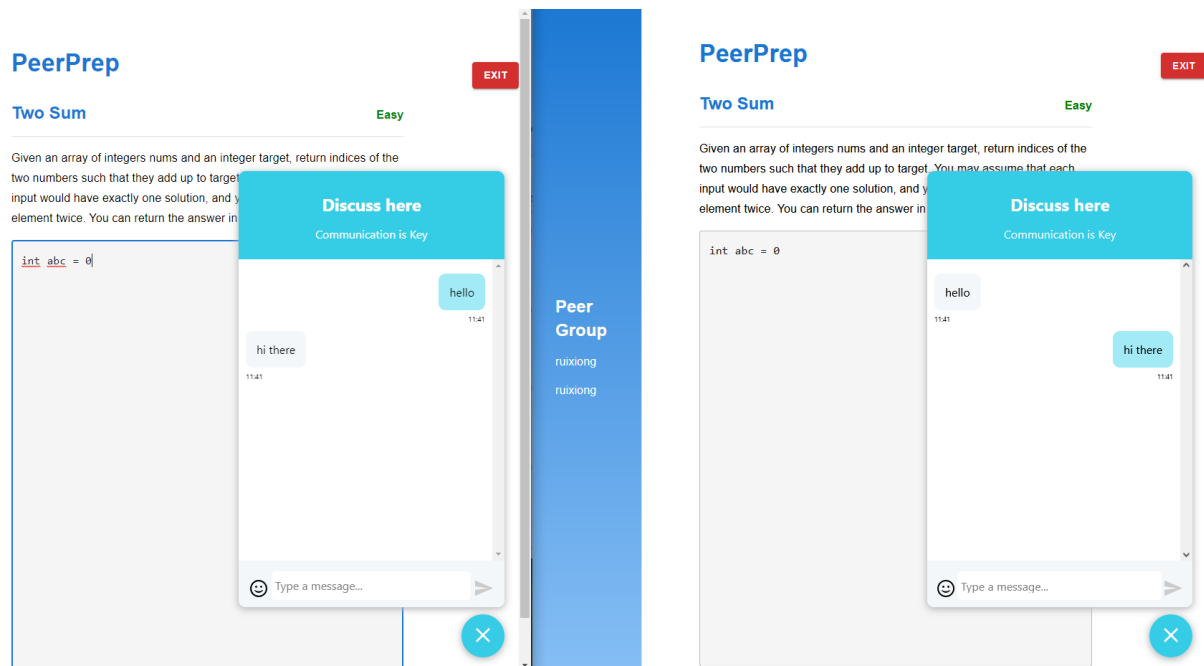


Diagram for pub-sub channel

We have utilized Socket.io, websockets and the publisher-subscribe design pattern for our matching, collaboration and chat service.

This is an implementation of the Pub-Sub pattern as the client listens to events emitted and the microservice will publish events to the observers (client) with the emit feature of Socket.io. For example, Websockets are used to inform the client in real time that a match has been found and that they have joined a room. The diagram above illustrates how the pub-sub mechanism works in our application.



Our matching algorithm is relatively simple and straightforward, it simply matches according to another person who has selected a similar difficulty level that has not yet been matched.

The matched users will be placed into 2 same socket.io rooms, one for collaboration and another for chat. They can then send and receive messages from one another via the room.

PeerPrep

Select difficulty level

Easy

LET'S GO



When a user selects a difficulty level and clicks the “LET’S GO” button, a match entry with a `isMatched` boolean flag to indicate it is unmatched will be created in the database. When another user with the same difficulty level finds a match, the pair will be matched and the boolean flag of both entries will be set to true.

While we do not allow duplicate usernames in the application, we allow a user who has logged in two different browsers on the same account to be matched to one another. This decision was made as if the site traffic was low and there were no other users present, a user would still be able to attempt the questions without having to make a secondary account.


During the collaboration, when one user leaves the room, both match entries will be deleted from the database and both users will be prompted back to the home page.

7. Testing

All workflows

Showing runs from all workflows

Q Filter workflow runs



Tell us how to make GitHub Actions work better for you with three quick questions.

Give feedback

×

186 workflow runs	Event ▾	Status ▾	Branch ▾	Actor ▾
✓ Merge pull request #44 from CS3219-AY2223S1/fix-pw-change Node.js CI Test (MATCHING) #60: Commit c3e1927 pushed by RuiXiong2211		28 minutes ago 34s	main	...
✓ Merge pull request #44 from CS3219-AY2223S1/fix-pw-change Node.js CI Test (QUESTION) #42: Commit c3e1927 pushed by RuiXiong2211		28 minutes ago 30s	main	...
✓ Merge pull request #44 from CS3219-AY2223S1/fix-pw-change Node.js CI Test (HISTORY) #30: Commit c3e1927 pushed by RuiXiong2211		28 minutes ago 45s	main	...
✓ Merge pull request #44 from CS3219-AY2223S1/fix-pw-change Build and Deploy to Cloud Run #22: Commit c3e1927 pushed by RuiXiong2211		28 minutes ago 6m 10s	main	...
✓ Merge pull request #44 from CS3219-AY2223S1/fix-pw-change Node.js CI Test (USER) #23: Commit c3e1927 pushed by RuiXiong2211		28 minutes ago 38s	main	...

Automated Continuous Integration using Github Actions

We have written unit test cases for the test cases deployed using Mocha and Chai. To optimize more time for development, the test cases we have written mostly cover the code where the critical path of the service is taken.

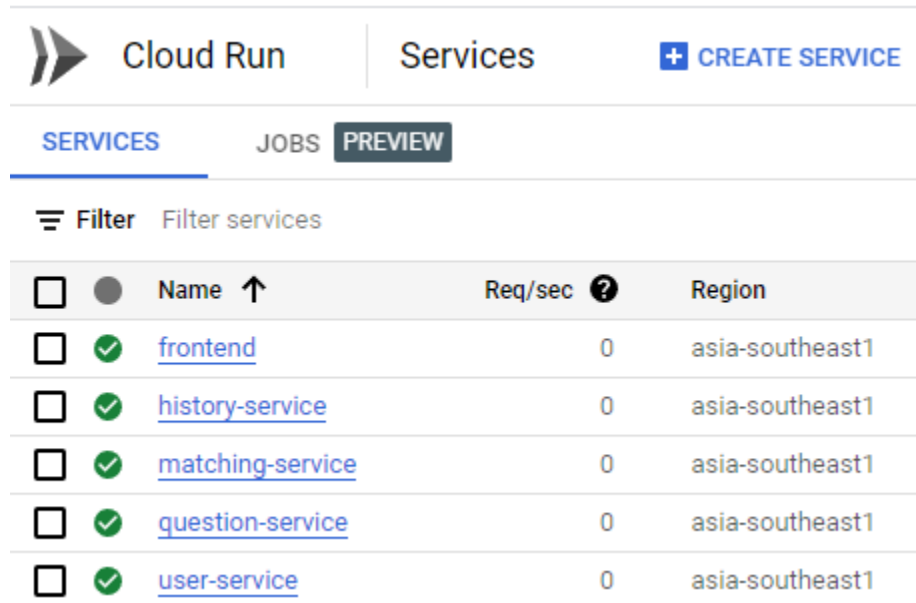
Integration Testing between the server and client is also performed when a microservice is ready to be deployed. System testing of the application is also done to search for potential bugs the user might encounter.

We have set in place the Continuous Integration (CI) pipeline of our unit tests in Github Action whenever we make a pull request to the master branch. This helps us automatically perform regression testing to ensure that the new changes in our code do not affect the current deployed services. The diagram above illustrates how a successful auto deployment looks like in Github Actions when merges are made to the master branch.

8. Deployment

After creating a project in Google Cloud and a service account with required permissions, the services are dockerized and built. The resulting docker images are then pushed to the Google Cloud Registry. Specifically, they are pushed to the Asia one (asia.gcr.io). We then use these images to deploy our Frontend and Backend

microservices to Google Cloud Run. They are deployed with the "asia-southeast1" region tag to facilitate faster response time.






















The screenshot shows the Google Cloud Run 'Services' page. At the top, there are tabs for 'Cloud Run' and 'Services', with a '+ CREATE SERVICE' button. Below the tabs are three sub-tabs: 'SERVICES' (selected), 'JOBS', and 'PREVIEW'. A 'Filter' button is visible. The main content is a table listing five services: 'frontend', 'history-service', 'matching-service', 'question-service', and 'user-service'. Each service has a checkbox, a status icon (green checkmark), a name with a link, a 'Req/sec' value of 0, and a 'Region' of 'asia-southeast1'.

<input type="checkbox"/>	<input checked="" type="radio"/>	Name ↑	Req/sec ?	Region
<input type="checkbox"/>	<input checked="" type="radio"/>	frontend	0	asia-southeast1
<input type="checkbox"/>	<input checked="" type="radio"/>	history-service	0	asia-southeast1
<input type="checkbox"/>	<input checked="" type="radio"/>	matching-service	0	asia-southeast1
<input type="checkbox"/>	<input checked="" type="radio"/>	question-service	0	asia-southeast1
<input type="checkbox"/>	<input checked="" type="radio"/>	user-service	0	asia-southeast1

After successfully deploying the services locally, continuous deployment is set up using Github Actions. Whenever there is a change in the main branch, the frontend and backend microservices will automatically be redeployed with the changes. Automating out deployment helps optimize more time for development work.

deploy

succeeded 3 hours ago in 7m 2s

- >  Set up job
- >  Checkout
- >  Google Auth
- >  Docker Auth
- >  Build and Push User Service Container
- >  Build and Push Matching Service Container
- >  Build and Push Question Service Container
- >  Build and Push History Service Container
- >  Build and Push Frontend Container
- >  Deploy User Service to Cloud Run
- >  Deploy Matching Service to Cloud Run
- >  Deploy Question Service to Cloud Run
- >  Deploy History Service to Cloud Run
- >  Deploy Frontend to Cloud Run
- >  Show Output
- >  Post Docker Auth
- >  Post Google Auth
- >  Post Checkout
- >  Complete job

9. Existing bugs

There are some existing bugs in our application that we have been unable to fix due to a lack of development time. We acknowledge the bugs below.

No.	Bug	How to replicate
1	Match is lost when the browser manually refreshes, but not both users are prompted back to the home page.	In a matched room with another user, manually refresh your browser.
2	If testing on the same computer, using the same type of internet browser will not match the users due to overriding of cookie values.	On the same browser, open up two instances of the application and click find match.

10. Suggestions for improvements and enhancements to the delivered application

10.1 Timer Feature

We can allow users to select the time duration for the question they attempt so that there will be a countdown timer while attempting the question. This is useful as it can replicate a real life interview environment whereby the interviewees are expected to complete technical questions within a certain amount of time.

10.2 Friends list

Creating a friends list for the user so they would be able to add other users and collaborate with them more frequently. This feature is also very extensible as a friends list can open more features such as having online statuses to check whether a friend is available to collaborate with. It can also allow users to have a better time while attempting the questions as they would be doing the questions with someone they are familiar with, and have more fun while preparing for the interview

10.3 Video Call

In the match room, we can allow users to video call each other so that they can have a more realistic setting of communicating with an interviewer. This can help users gain

more confidence so that they feel comfortable while attempting the questions while simulating being observed by an interviewer

10.4 Admin Role

Creating a new admin role can be beneficial for the app in the long run as admin access to certain things such as populating the question bank, or setting the matching duration helps make modifications to the app without having to constantly tamper with the code.

11. Reflection and learning points for the project process

Throughout the development process, delegation of tasks went well and everyone was able to complete their tasks on time and we also learnt how to collaborate with team members when integrating frontend and backend aspects of tasks. We made sure to Maintain good practices such as pulling from github frequently so we did not have to resolve any merge conflicts at all, and also communicated frequently by having weekly sync ups with the team, which helped all of us get on track with each other's progress and offer help whenever needed. It was definitely a challenging project as we had to quickly pick up new technologies, but overall it was a fruitful experience as we were able to learn new practical and relevant skills.