



School of Computing

CS3219 Software Engineering Principles and Patterns

AY22/23 Semester 1

Final Report

Group 8

Team Members	Student No.
Bu Wen Jin	A0205129M
Foong Siqi	A0205150Y
Li Huankang	A0199799N
Jonas Ng Zuo En	A0201836L

Mentor(s): Foo Kai En

Content Page

Content Page	2
Overview	4
Background and Purpose of the Project	4
Architectural Design	4
Microservices & Tech Stack	5
Technical Report – Implementation Details	6
1 User Service	6
1.1 Requirements	6
1.2 Class Architecture	7
1.3 Database Schema	7
1.4 Development Process	8
2 Matching Service	9
2.1 Requirements	9
2.2 Database Schema	10
2.3 Design Decisions	11
3 Question Service	13
3.1 Requirements	13
3.2 Class Architecture	13
3.3 Database Schema	14
3.4 Design Decisions	15
4 Collaboration Service	17
4.1 Requirements	17
4.2 Development Process	18
5 Communication Service (Nice to have)	20
5.1 Requirements	20
5.2 Design Decisions	21
6 History Service (Nice to have)	22
6.1 Requirements	22
6.2 Database Schema	22
6.3 Design Decisions	23
7 Frontend	25
7.1 Requirements	25
7.2 Development Process	25
7.3 Design Decisions	28
8 Deployment (Nice to have)	29
8.1 Requirements	29
8.2 Documentation	29
8.3 Design Decisions	32

Development Process & Project Management	33
Scrum Methodology	33
Project Management	34
CI/CD Pipeline (Nice to have)	35
Project Schedule	36
Member Contributions	37
Conclusion	38
Suggestions for Improvements and Enhancements	38
Reflections and Learning Points from the Project Process	40
Appendix	41
User Service Documentation	41
Matching Service Documentation	44
Question Service Documentation	46
Communication Service Documentation	53
History Service Documentation	54

Overview

Background and Purpose of the Project

Increasingly, students face challenging technical interviews when applying for jobs which many have difficulty dealing with. Issues range from a lack of communication skills to articulate their thought process out loud to an outright inability to understand and solve the given problem. Moreover, grinding practice questions can be tedious and monotonous.

PeerPrep is a web application to help students or users better prepare themselves for technical interviews. PeerPrep is an interview preparation platform that matches two users to practise whiteboard-style questions.

Architectural Design

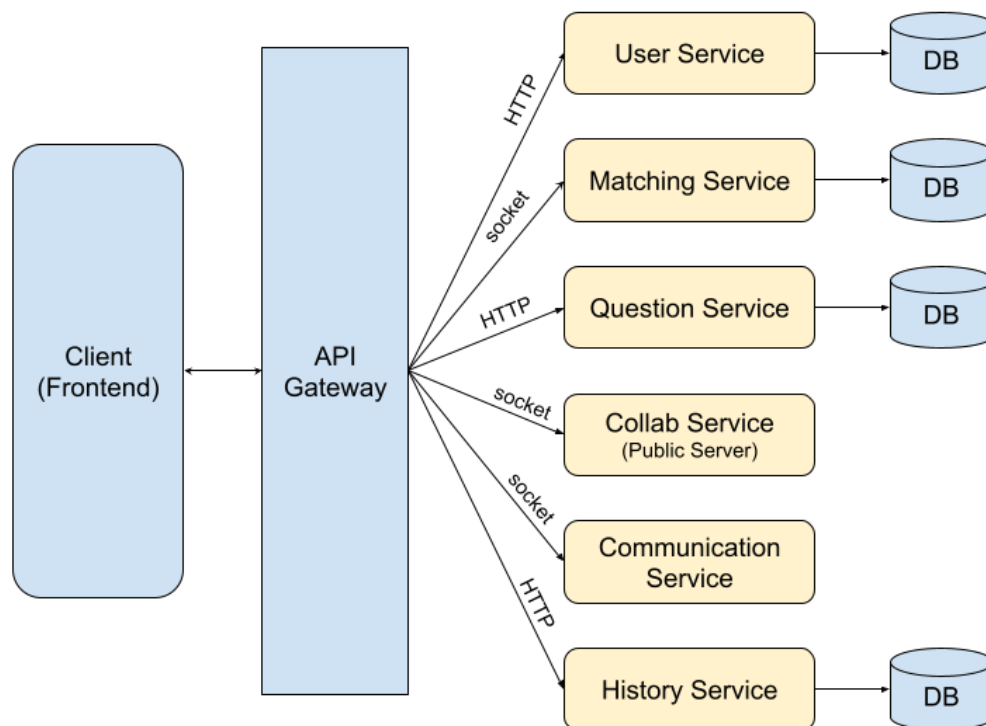


Figure 1: Overall PeerPrep Architecture

Microservices & Tech Stack

Functionality of each service

Microservice	Description
User Service	Handles authentication of users.
Matching Service	Handles matching of users via difficulty level selected.
Question Service	Stores all questions of various difficulty and randomly assigns a question to new matches based on difficulty level.
Collab Service	Handles editor collaboration between matched users.
Communication Service	Enables 2-way communication between matched users in a collaboration room.
History Service	Store user's latest question attempts.

Tech stack of each service

Component/Service	Tech Stack	Local env port number
Frontend	React + Redux, Socket.io, Axios	3000
User Service	NodeJs, PostgreSQL, Express	8000
Matching Service	NodeJs, PostgreSQL, Socket.io	8001
Question Service	NodeJs, PostgreSQL, Express	8002
Collaboration Service	NodeJs	-
Communication Service	NodeJs, Socket.io	8005
History Service	NodeJs, MongoDB, Express	8004
Deployment	Google Kubernetes Engine, Cloud SQL, Mongo Atlas, Terraform	-
CICD	Github Actions + Cloud Build, Terraform	-
Project management	Github Issues + GitHub Project Board	-

Technical Report – Implementation Details

1 User Service

1.1 Requirements

S/N	Functional Requirements	Priority	Status
FR1.1	The system should allow users to create an account with username and password.	High	Done ▾
FR1.2	The system should ensure that every account created has a unique username.	Medium	Done ▾
FR1.3	The system should allow users to log into their accounts by entering their username and password.	High	Done ▾
FR1.4	The system should allow users to log out of their account.	High	Done ▾
FR1.5	The system should allow users to delete their account.	Medium	Done ▾
FR1.6	The system should allow users to change their password.	High	Done ▾

S/N	Non-Functional Requirements	Priority	Status
NFR1.1	Users' passwords should be hashed and salted before storing in the DB.	High	Done ▾
NFR1.2	The system should have a web token to cache user sessions.	High	Done ▾
NFR1.3	Username and passwords should be at least 8 characters.	Medium	Done ▾
NFR1.4	Passwords should have at least one uppercase, one lowercase, one digit and one symbol.	Medium	Done ▾

1.2 Class Architecture

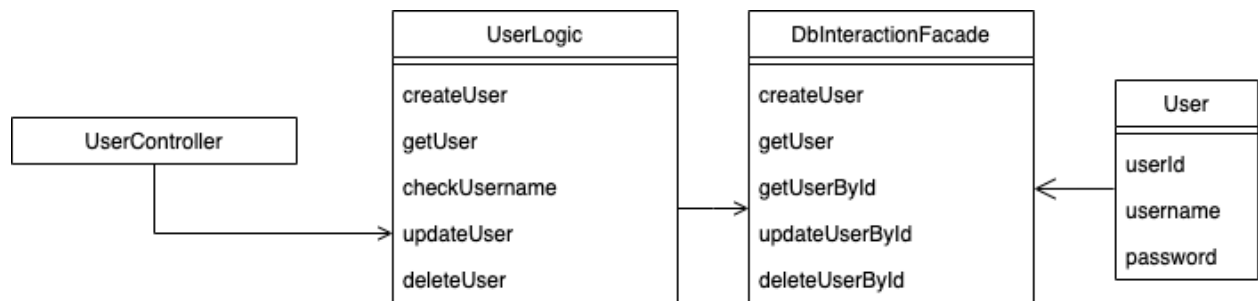


Figure 2: Class diagram for User Service

Facade pattern in User Service

User Service contains 1 database table. We used PostgreSQL to implement this table, hence there is a lot of code specific to PostgreSQL in this service. Those third party database interactions are abstracted out using a Facade pattern, to keep the business logic separate from the code interacting with the third party libraries. **DbInteractionFacade** acts as the Facade that provides a simple interface for the complex PostgreSQL subsystem which is relevant to our User Service.

Principles:

- **Reduced Coupling.** This helps us to reduce coupling between the database and our application.
- **Single Responsibility Principle.** This helps us to ensure each component is only responsible for 1 job. (ie. **UserController** will be responsible for error handling, **UserLogic** will be responsible for implementing business logic and **DbInteractionFacade** for interacting with database)
- **Portability.** If we decide to change the database used for the **User** table, we can easily change the **DbInteractionFacade** to fit the new database, and not have to change business logic.

1.3 Database Schema

User	
PK	<u>userId UUID NOT NULL</u>
	username STRING NOT NULL
	password STRING NOT NULL

Figure 3: User table in User Service

The **User** table stores the auto-generated **userId**, chosen **username** and hashed **password**.

1.4 Development Process

Authentication using JWT and bcrypt

The bcrypt library and the JWT protocol was used to authenticate users in User Service.

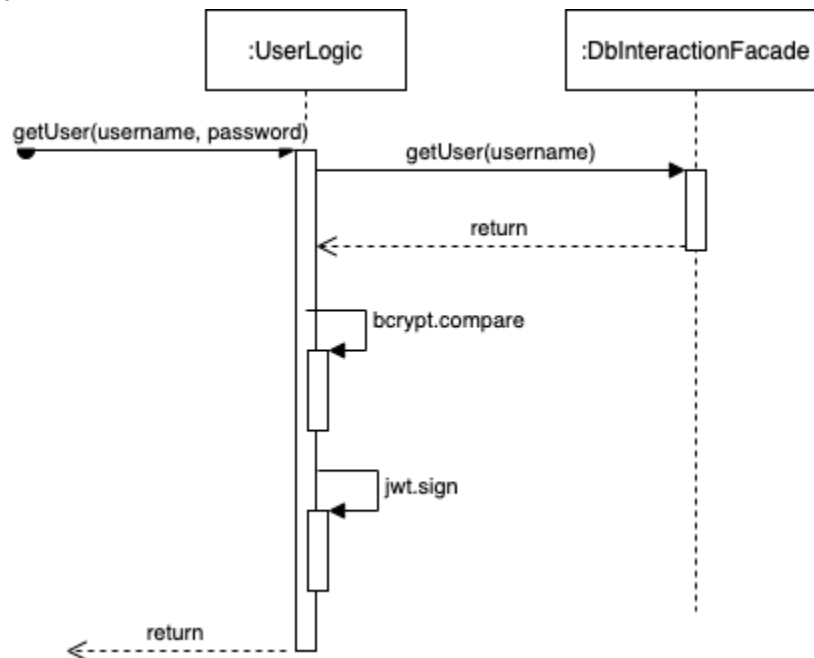


Figure 4: Sequence diagram showing login logic and interaction between UserLogic and DbInteractionFacade

The above sequence diagram shows what happens when a user attempts to login. First, `UserLogic` will call `getUser` to get the stored hashed password in the database. It then calls the `bcrypt.compare` method to check that the password given by the user in the login query matches the hashed password stored in the database. If the password matches, it signs a jwt token with the `userId` of the user attempting to login and returns it. If the password doesn't match, `UserLogic` throws an error that is handled by `UserController`.

Principle: Encapsulation and information hiding

In User Service, a `userId` and `token` is returned when the user is successfully authenticated. All the logic behind the authentication and user information is hidden behind these 2 pieces of data. Whenever the frontend client has a valid `userId` and `token` they can be assured that this user has been authenticated. The frontend can then proceed to do other API calls or use this information for the authenticated user without having to be concerned about the details of how this user is authenticated or what information to store about the user.

2 Matching Service

2.1 Requirements

S/N	Functional Requirements	Priority	Status
FR2.1	The system should be able to match 2 active users	High	Done ▾
FR2.2	The system should be able to timeout after 30 seconds if no matching users are found	High	Done ▾
FR2.3	The system should be able to find an appropriate match for users based on difficulty level selected	High	Done ▾
FR2.4	The system should be able to detect when a new user starts matching	High	Done ▾
FR2.5	The system should be able to allow users to cancel matching	Medium	Done ▾
FR2.6	The system should allow users to select the difficulty level of the questions when finding a match.	High	Done ▾
FR2.7	The system should allow users to find a match promptly after leaving a room.	Medium	Done ▾
FR2.8	The system should allow matched users to leave the room at any point in time after being matched	High	Done ▾
FR2.9	The system should inform a user if his/her matched user has left the room	Medium	Done ▾
FR2.10	The system should not allow a user who is currently matched to find another match.	High	Done ▾

S/N	Non-Functional Requirements	Priority	Status
NFR2.1	The system should not match a user to more than one other person	High	Done ▾

2.2 Database Schema

MatchPotential maintains records of users who are currently waiting for a match. This includes information relating to the user's ID, user's socket ID, and level selected for the match.

Matched maintains records of matched users, including information relating to users' ID, users' socket ID, and level selected for the match.

MatchPotential		Matched	
PK	<u>userId</u> STRING NOT NULL	PK	<u>matchedId</u> UUID NOT NULL
	level STRING NOT NULL socketId STRING NOT NULL		userId1 STRING NOT NULL userId2 STRING NOT NULL level STRING NOT NULL socketId1 STRING NOT NULL socketId2 STRING NOT NULL

Figure 5: Database Schemas for Matching Service

The API endpoints and developer guide are specified in the [Appendix section](#) below.

2.3 Design Decisions

Client-server communication using API vs Socket.io

Both API and Socket.io were considered as potential options for the underlying protocol to be used to implement the matching service.

Criteria	RESTful API	Socket.io
Type of Communication	Only facilitates unidirectional communication from the client to the server	Allows for bidirectional and full-duplex communication between client and server through the emitting of events
State Management	A stateless protocol which does not maintain state across different requests	A stateful protocol, maintains state for each socket connection
Performance	Requires a new TCP connection to be established before a request can be sent, resulting in slower performance and transmission of messages between client and server	Much lower latency and higher throughput during transmission of messages since multiple messages can be sent continuously in each TCP connection

Final Decision

In this case, we decided to use Socket.io for communication between the frontend and matching service as we require bidirectional and stateful communication between the services.

Furthermore, with the consideration that the matching service has to perform multiple matchings concurrently, the use of Socket.io would minimise delays during the matching process, which is important towards user experience.

Handling of events after match finding

Certain events during the collaboration stage require the modification and/or deletion of records in the **Matched** table. In view of this, there were 2 approaches that we considered:

1. Collaboration page on the frontend would communicate with collaboration service which would access the **Matched** table
2. Collaboration page on the frontend would communicate with the matching service which would access the **Matched** table.

Final Decision

We chose approach 2 as it reduced coupling between collaboration service and matching service, which is an important consideration in a microservice architecture. This adheres to the separation of concerns principle by ensuring that collaboration service would not have to concern itself with any matching service related logic. While approach 2 might introduce some unwanted interactions between the collaboration and matching stages, it was necessary to ensure the correctness of future matching processes.

Prevention of race conditions

Another design consideration in the implementation of the matching service was the use of mutexes as a solution to potential race conditions during the matching process.

Problem

Since multiple users might be looking for a match concurrently, unsynchronised accesses to the database to look for waiting users might result in race conditions. Such race conditions can result in multiple users being matched to the same user, or a user being matched to a user who has already found a match.

Solution

To ensure synchronised accesses to the database for read and write operations, the use of mutexes as a synchronisation tool was appropriate as it prevented concurrent accesses to the same database. This would ensure that each matching operation was viewed as an atomic operation to maintain the integrity and correctness of the matching process.

Final Decision

However, it was important to note that the use of mutex would decrease the performance of the matching operation, since potentially many processes must wait to acquire the mutex before any database operations can be performed. This can affect scalability when the number of users finding a match increases. Furthermore, it was crucial to implement the use of mutexes correctly to avoid potential deadlocks which would compromise the matching service. Eventually, we decided that the use of mutexes was necessary to ensure the correctness of the matching process, which was of greater priority as opposed to performance. To prevent the occurrence of deadlocks during the matching process, it would suffice to perform adequate testing with multiple concurrently matching users to ensure that the behaviour of the matching process was as expected.

Possible improvements

Currently, the process of finding a match involves sequential queries to both the **MatchPotential** and **Matched** tables. However, a mutex has to be acquired by the process before it can start querying either tables. This can result in instances where the process is currently querying the **MatchPotential** table but no other process is allowed to query the **Matched** table despite the fact that independent accesses to different tables will not trigger any race conditions. We can optimise the performance of the matching service further by creating a distinct mutex for each table, and enforcing the release of a mutex once the table operation has been completed. This can allow for concurrent access to different database tables by different processes.

3 Question Service

3.1 Requirements

S/N	Functional Requirements	Priority	Status
FR3.1	Need store a question bank indexed by difficulty level (and any other indexing criteria – e.g., specific topics)	High	Done ▾
FR3.2	The system needs to be able to retrieve a question of appropriate difficulty level to the shared workspace (room) once the participants are matched	High	Done ▾
FR3.3	Allow database admins to perform CRUD on the question bank	High	Done ▾

3.2 Class Architecture

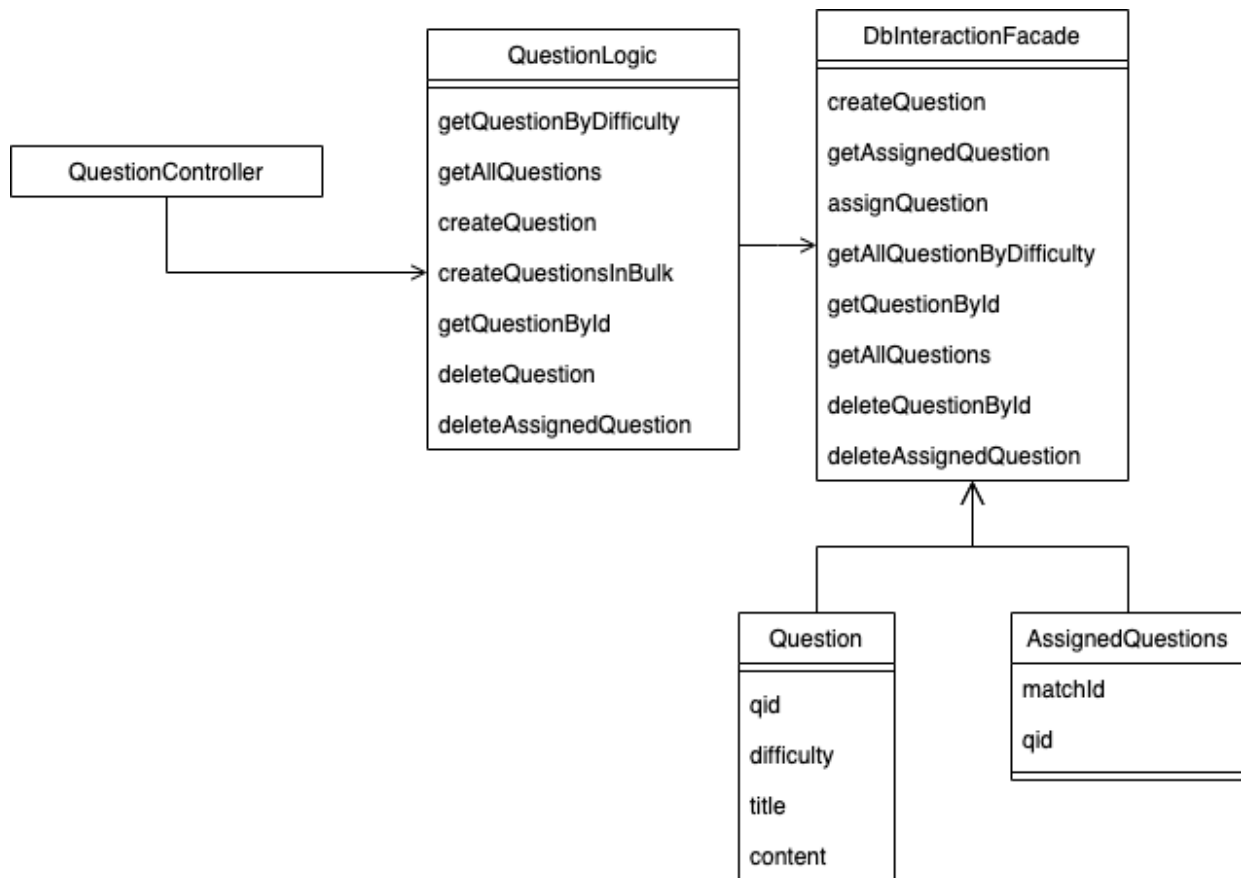


Figure 6: Class diagram for Question Service

Facade pattern in Question Service (Similar to User Service)

Question Service contains 2 database tables, `Question` and `AssignedQuestions`. We used PostgreSQL to implement these 2 tables, hence there is a lot of code specific to PostgreSQL in this service. Those third party database interactions are abstracted out using a Facade pattern, to keep the business logic separate from the code interacting with the third party libraries. `DbInteractionFacade` acts as the Facade that provides a simple interface for the complex PostgreSQL subsystem which is relevant to our Question Service.

Principles:

- **Reduced Coupling.** This helps us to reduce coupling between the database and our application.
- **Single Responsibility Principle.** This helps us to ensure each component is only responsible for 1 job. (ie. `QuestionController` will be responsible for error handling, `QuestionLogic` will be responsible for implementing business logic and `DbInteractionFacade` for interacting with database)
- **Portability.** If we decide to change the database used for the `Question` and `AssignedQuestions`, we can easily change the `DbInteractionFacade` to fit the new database, and not have to change business logic.

3.3 Database Schema

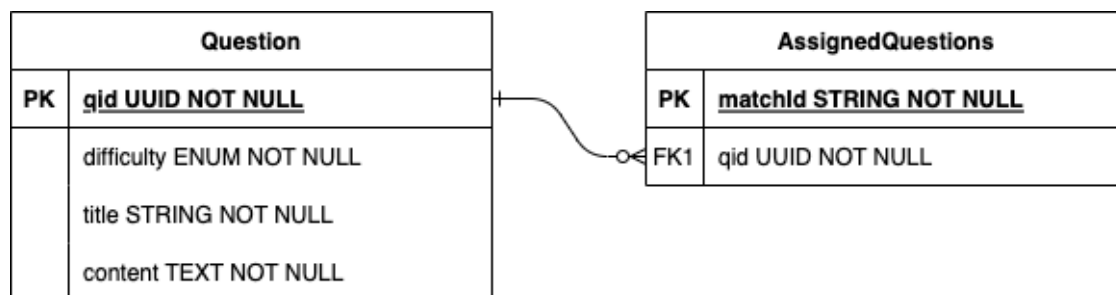


Figure 7: Database schema for Question Service

The `Question` table stores all the information related to a question. The `title` and `content` are information that will be displayed to users so that they can solve the chosen question. The `difficulty` is used to assign each pair of matched users a question that is of the difficulty chosen by them.

The `AssignedQuestions` table stores the question assigned to each active pair of matched users. This is to help maintain data persistence in the sense that when a user refreshes the page or has to query for the question again, the question returned will be the same as the original question he/she was assigned. If no question has been assigned to this match, it will be assigned a random question of matching difficulty.

3.4 Design Decisions

Whether to store **matchId** in the Question Service

matchId is the unique ID generated by the matching service once 2 users have been matched with each other. Each match should be assigned a question and each time the “get random question” endpoint is called, it should return the same question for the same match. We can implement this behaviour using different ways as discussed below.

Criteria	Store matchId in Question Service	Develop an indexing system that allows us to always fetch the same question for the same matchId
Ease of Testing	Simple implementation and minimal edge cases	Additional complexities to handle edge cases such as when a different match has the same matchId as a match that is no longer active match
Coupling	Data coupling with Matching Service as Question Service is no longer fully independent from Matching Service.	Question Service is fully independent from Matching Service

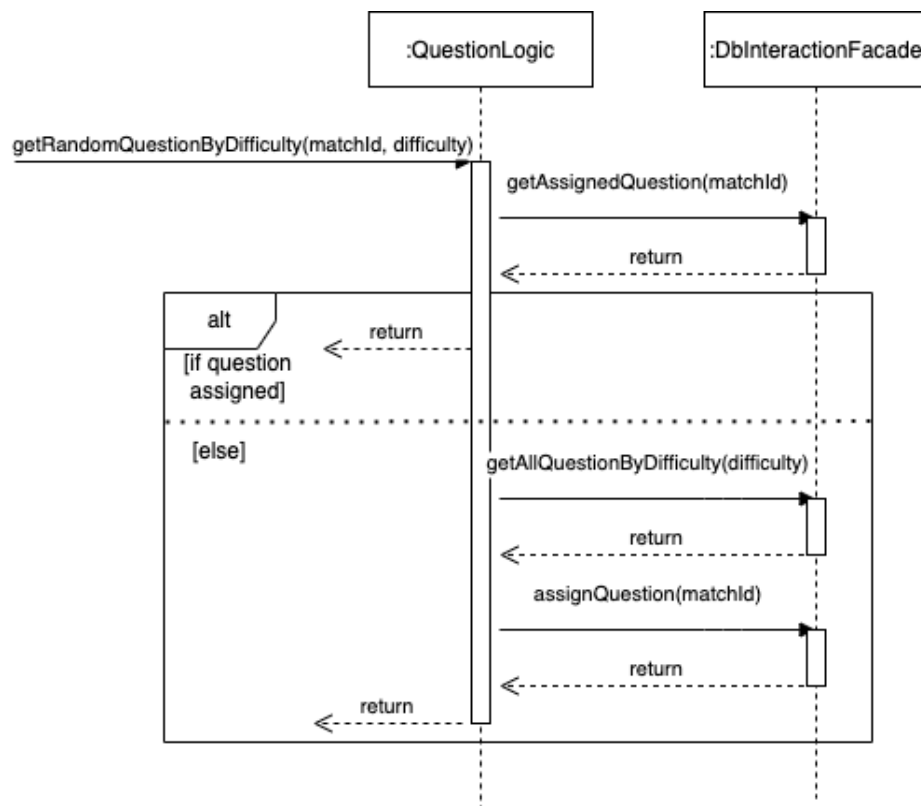


Figure 8: Sequence diagram to show get random question business logic

According to Figure 8, when Question Service receives a query for a random question according to difficulty, it first calls `getAssignedQuestion` to check if a question has already been assigned to this active match. If it has, the Question Service will trivially return that question. If it hasn't, the `QuestionLogic` component will call `getAllQuestionsByDifficulty` to get all the potential questions and pick 1 randomly from the list. It then calls the `assignQuestion` method to store this question into the `AssignedQuestions` database and finally return the chosen question.

Final decision

After considering the pros and cons of both choices, we chose to store `matchId` in Question Service because the data coupling between Matching Service and Question Service is not that strong to justify all the additional implementation complexities required to fully decoupled the 2 services.

4 Collaboration Service

4.1 Requirements

S/N	Functional Requirements	Priority	Status
FR4.1	The system should provide participants with a text box for code editing.	High	Done ▾
FR4.2	The system should allow collaborative code editing between 2 participants in the same room.	High	Done ▾
FR4.3	The system should display the same content on the code editor for all participants in the same room.	High	Done ▾
FR4.4	The system should be able to handle all participants simultaneously typing on the code editor at the same time.	High	Done ▾
FR4.5	The system should allow participants to edit any line in the code editor.	High	Done ▾
FR4.6	The system should create a workspace (room) once 2 users have been matched.	High	Done ▾
FR4.7	The system should close the workspace (room) when both users have left the room.	High	Done ▾

S/N	Non-Functional Requirements	Priority	Status
NFR4.1	The system should allow users to view changes on the code editor made by other participants with a delay of less than 1 second.	Medium	Done ▾
NFR4.2	The system should not exhibit race conditions.	High	Done ▾
NFR4.3	An alert will be shown to confirm the user's action when users want to close or refresh the browser tab.	Medium	Done ▾

4.2 Development Process

Initial Design - Socket.io

Initially, we designed our real-collaborative text editor using socket.io. How we implemented this was to create rooms in our backend collaboration-service, and get the 2 user's sockets to join the room. This means that only those in the same room are able to receive each other's messages.

Firstly, when the 2 users match, we get the `matchId` from the matching-service. We use this `matchId` as the `roomId` in our collaboration-service room making. When the user makes the initial connection to the backend collaboration-service server, both the `userId` and `matchId` are sent over.

When either one of the users first connects to the collaboration-service backend server, the server will check if a room with the provided `matchId`. If such a room already exists, the socket (user) will join the room. If the room does not exist, it will create a room with `matchId` as the name, and the socket will then join this room.

Using this feature of rooms in socket.io, we are able to broadcast messages from one socket to another easily, allowing us to efficiently display changes shared across users. This means that when one user makes a change, this change is sent to all users, allowing them to see the changes being made in real time.

Problem faced

Although using this implementation allows us to broadcast the changes from one user to another, there is a very big drawback to this simple design - there is no state being stored in the server. What this means is that the server does not keep track of the current state of the editor, meaning it will just take any changes from one user and broadcast to the other user. This means that when both users are editing the editor at the same time, there will be a conflict on the state as each user will receive the new changes from the other user. Another issue is the position of the cursor. As we are not maintaining any state, there is only 1 cursor throughout the entire editor, this means that both users will be using the same cursor, preventing them from editing at different locations simultaneously.

The impact can be seen when 2 users simultaneously modify the editor, this will lead to 2 different outcomes.

1. Overwriting of changes - when 2 changes are sent to the server at the same time, the server will broadcast both changes to the other user. This would lead to some changes being lost and a desync in the state of the editor. As user A's changes are not present in the incoming changes of user B, this will lead to the change from user A to be overwritten.
2. Infinite changes loop - as each change is applied to the editor of the other user, due to the overwriting, this leads to a desync of state of the editor. This would result in an infinite propagation of changes from both users, leading to an infinite loop, where each user is constantly sending the "newly" detected changes from the other user.

Iteration 2 - storing the state of the editor in the server

To fix the problems in the initial iteration, we tried to implement a caching system that will store the state of the editor, merge the changes in, then send the resultant state back to each user. This will help to solve the problem of merging multiple changes coming from both users at the same time.

Problem faced

Now we have a new issue of race conditions, where we need to determine which change should be applied if both users modify the same area at the same time.

As the changes from both users will come in at the same time, only the latest change will be applied as the earlier change will be overwritten. This means that when this scenario happens, there is a potential loss of data for one of the users as his/her change will not be reflected correctly.

Final iteration - using a public server for the backend

After speaking to our mentor, and checking with the professor, we decided to rely on the public communication server that is provided with the library that we are using, which is a public websocket server. This server will help to handle all the race conditions, cursor tracking, as well as merge conflicts.

As such, this means that we are no longer using our own backend services for the collaborative editor. However, relying on the public server means that we are adding a new dependency to our application, which is the availability of the public server. If the server goes offline, then the editor will no longer function properly. As such, we have managed to find the source code of the deployed public server, and have the option to host it on our own, which our frontend editor will then be able to connect to it.

The backend is hosted by a public server at "<wss://demos.yjs.dev>".

5 Communication Service (Nice to have)

5.1 Requirements

S/N	Functional Requirements	Priority	Status
FR5.1	The system should allow users to send messages to the other online student who is solving the same question.	High	Done ▾
FR5.2	The system should not allow users in different rooms to be able to communicate.	High	Done ▾
FR5.3	The system should not allow users to view the chat messages belonging to a different room.	High	Done ▾
FR5.4	The system should allow the 2 users to send messages concurrently at the same time.	High	Done ▾
FR5.5	The system should allow users to view past messages sent within the same collaboration session.	High	Done ▾

S/N	Non-Functional Requirements	Priority	Status
NFR5.1	Messages should be sent in real-time and received within 1 second.	High	Done ▾
NFR5.2	Each message should not exceed 1000 characters.	Medium	Done ▾
NFR5.3	Messages sent should not be empty.	Medium	Done ▾

5.2 Design Decisions

Message Queue vs Socket.io

There were 2 approaches that we considered during the implementation of the chat functionality between 2 users in a room:

1. Publisher-Subscriber messaging via a message broker
2. Socket.io

We decided to use approach 2 for the following reasons:

1. The use of Socket.io allowed for easier implementation of the chat functionality since the emitting and listening of events was sufficient to handle the transmission of messages. In contrast, the use of a message broker was more complex since it involved setting up the message broker, creation of message topics and so on. This was inherently more complicated for the relatively simple task of communication between 2 users.
2. There was no need for data (chat message) persistence since messages would only need to persist for as long as the collaboration room was active. As such, storing of messages within the browser state was sufficient. In such a case, the use of a message broker for data persistence was not a crucial consideration.
3. There was no need for asynchronous jobs since the communication service's only purpose is to deliver messages between 2 users. There are no implications in blocking a process while waiting to consume a message. As such, the use of a message broker to allow for asynchronous processing was unnecessary for our use case.
4. There were no complexities in consuming messages since there are only 2 consuming processes at any point in time. In this case, the use of Socket.io was sufficient for our requirements.
5. Despite Socket.io having a much lower performance and throughput than message broker, since the scope of this project only required 2 users in a collaboration room, there would be no significant difference in performance between the use of Socket.io and message queues.
6. Message broker is a relatively newer topic for us and was taught in the second half of the semester. With little experience and a tight timeline, we decided not to explore the use of message brokers but instead stuck to a familiar technology, Socket.io, which was also used in matching service.

While the above section highlights the key reasons for our decision to use Socket.io, there is room for exploration into the use of message brokers for reasons that have been included in the [Suggestions for improvements and enhancements section](#) below.

6 History Service (Nice to have)

6.1 Requirements

S/N	Functional Requirements	Priority	Status
FR6.1	The system should be able to save the user's attempt for the question after the user exits the collaboration room.	High	Done ▾
FR6.2	The system should be able to display the user's latest past attempt to a question.	High	Done ▾

S/N	Non Functional Requirements	Priority	Status
NFR6.1	The system should be able to allow users to see their history of questions that they have attempted sorted in reverse chronological order.	Low	Done ▾

6.2 Database Schema

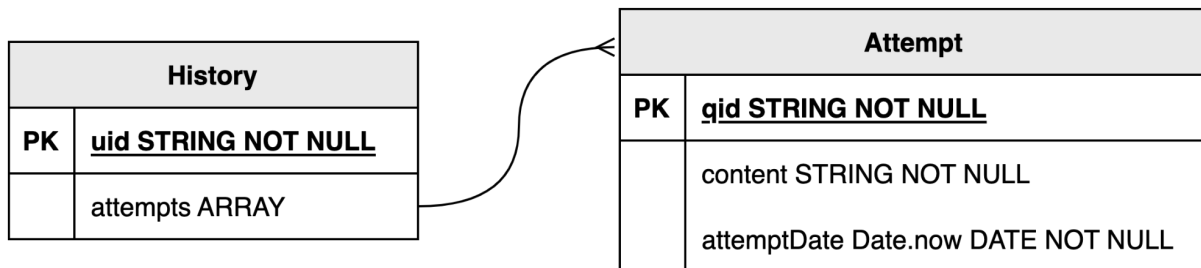


Figure 9: History document in History Service

The **History** document stores the **uid**, given **attempts** array that contains the **qid**, user's attempt as **content** and the **attemptDate**.

The API endpoints and developer guide are specified in the [Appendix section](#) below.

6.3 Design Decisions

Using MongoDB vs PostgreSQL as the database

Criteria	MongoDB	PostgreSQL
Type of database	NoSQL, document-oriented database that uses documents to manage information	Relational database, uses tables to manage and store information
Ease of extensions	Documents can vary in terms of key/value pairs as they are stored as JSON. There is also an option of schema validation to enforce data governance controls over every collection if needed.	The whole schema needs to be designed and configured at creation. Need to alter the table to make any changes to the schema.
Ease of querying nested data	Query performance in MongoDB can be accelerated by creating indexes on fields in documents and subdocuments. MongoDB allows any field of a document, including those deeply nested in arrays and subdocuments, to be indexed and efficiently queried.	Slightly slower to query for nested data

Final Decision

We decided to use MongoDB as the database for the History Service as the data stored does not need to relate to other data therefore the use of tables in PostgreSQL is not required.

Furthermore, using PostgreSQL means that we would need to define its structure in advance. However, we were constantly changing the structure of the data to be stored as we decide between storing all attempts and storing just the latest attempt to the same question.

As we may want to query for a specific question attempted by a user, MongoDB is a better choice as it allows efficient querying for deeply nested subdocuments.

Whether to store the entire question in the History Service

Criteria	Store entire question	Store only the question id
Storage efficiency	Less efficient use of database storage as questions are repetitive in Question Service's database and the history document. Furthermore, each history will store the exact same question.	More efficient use of database storage as a history document only needs to store the question id string.
Ease of use	Frontend only needs to query the History Service to get all the data required to display the users' question history on the website.	Frontend needs to query both the History and Question Service to get all the data required to display the users' question history on the website.
Compliance to Design Principles	Violates OCP as a change in the question's data (level, title or content) in the Question Service will require the same change in all the history documents with the affected question. Violates SRP as the History Service will need to keep track of the changes in question which is not the main responsibility of the service.	Compliance with OCP as a change in the question's data will not affect the history document as we will query from the Question Service to get the latest question. Compliance with SRP as the History Service will only need to track and store the users' attempts to the question only.

Final Decision

By applying OCP and SRP design principles, we decided to store only the question id for each question attempt in the history document. Although there is a tradeoff where the Frontend needs to make an additional HTTP request to the Question Service to get the question data, the additional latency would not be significant. It is also more important that all the questions' data have only one source of truth which should lie in the Question Service.

7 Frontend

7.1 Requirements

S/N	Functional Requirements	Priority	Status
FR7.1	The system should be able to provide users a page for authentication purposes (login, sign up, logout, change password and delete account).	High	Done ▾
FR7.2	The system should be able to display a page for users to choose the question's difficulty level and start matching.	High	Done ▾
FR7.3	The system should be able to route matched users to a page for them to collaborate and work on a given question.	High	Done ▾
FR7.4	The system should be able to allow users to navigate to a page for users to view the history of all the questions they did with the latest attempt shown.	High	Done ▾

7.2 Development Process

Collaboration page design was inspired from Leetcode design

We decided to take inspiration from the Leetcode webpage when designing our collaboration page frontend UI as seen in Figures 10 and 11 below. The webpage is split into 2 halves, the right half with the code editor for users to type and the left half contains the question and the chatbox for matched users to communicate with each other. The reason for following this design is due to the fact that the design is simple and allows users to easily refer to the question while conducting their whiteboard coding practice.

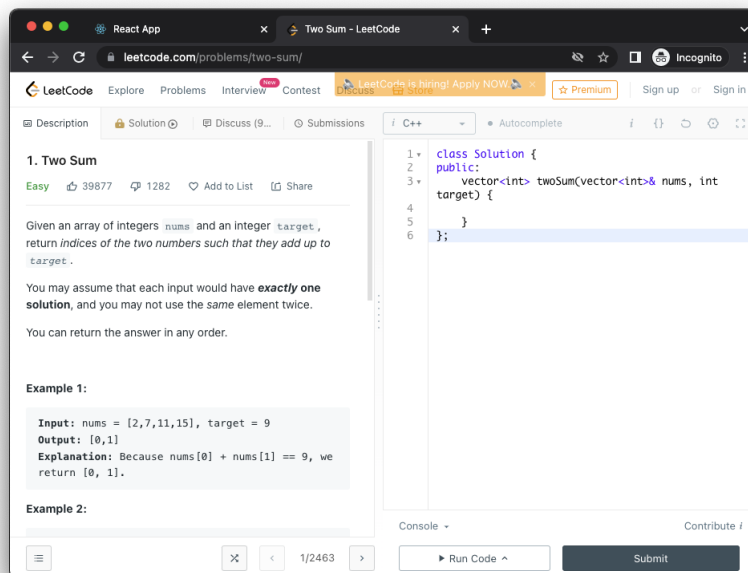


Figure 10: Leetcode coding editor design

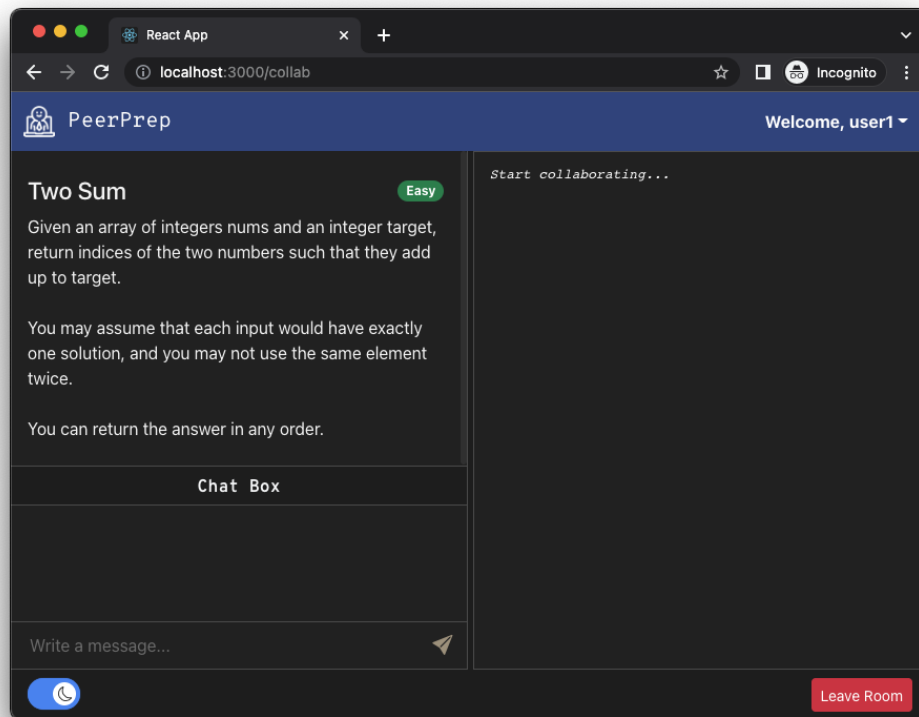
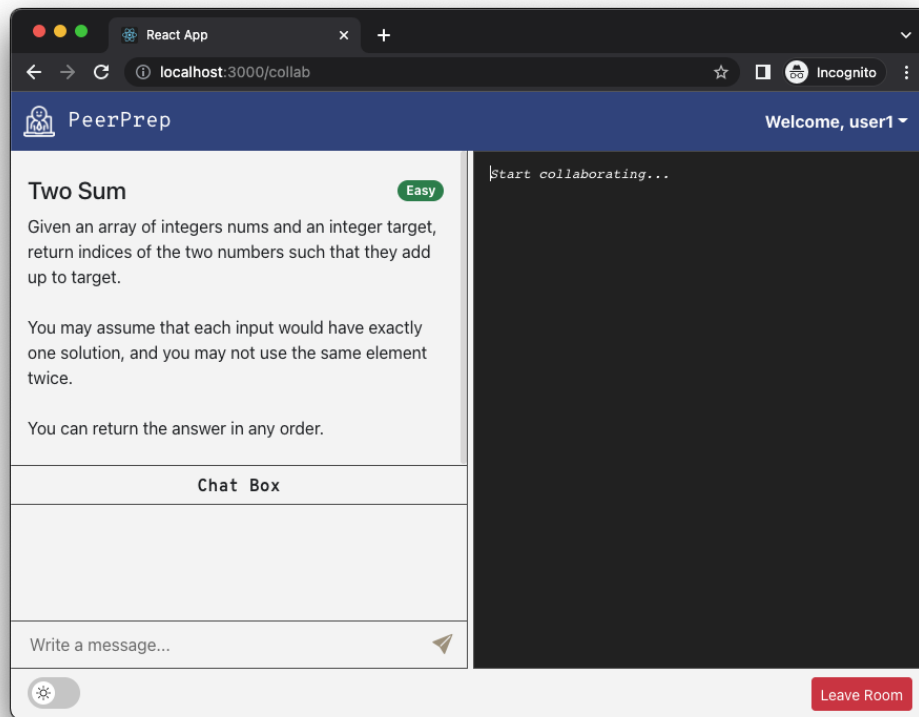


Figure 11: Our collaboration page in light and dark modes

Addition of light and dark mode for the collaboration page

Nearing the end of the project, we decided to implement a function to let users toggle between light and dark mode for the collaboration page as seen in Figure 11 above. This provides an option for users who wish to code in dark mode to change the website UI. We decided to add this feature as most IDEs now have dark themes and our users may want to recreate the same feeling of coding in dark mode.

Frontend as a Mediator between services

Our frontend acts as a controller that handles all communications between microservices, and directs each service to perform the intended function.

In order to reduce the chaotic dependencies between services due to direct interactions, we decided to use frontend as the mediator where all services communicate and collaborate through the frontend.

For example, the question service stores the assigned questions for the matches and would need to interact with the matching service to obtain the `matchId`. This interaction is done through the frontend where the `matchId` is obtained from the matching service before sending over to the question service via a HTTP post request.

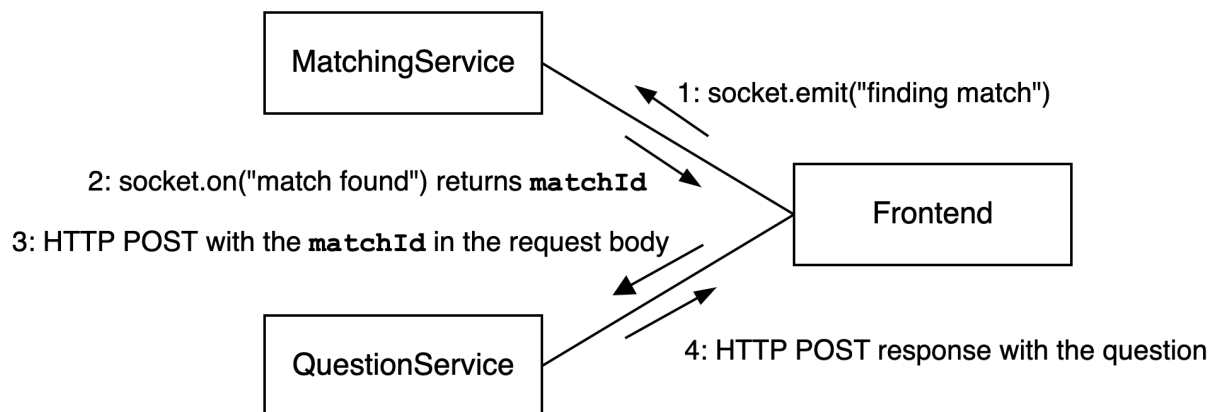


Figure 12: Interactions between services through the Frontend

7.3 Design Decisions

Bootstrap vs Material UI as Styling Toolkit

Criteria	Bootstrap	Material UI
Speed of Development	Relatively faster Bootstrap goes with many UI components, like typography, tables, buttons, navigation, labels, alerts, tabs, etc. It provides enough necessary elements to build a good-looking design with modest efforts and allows developers to concentrate on the functionality of the app.	Material UI is mostly a set of UI components, it doesn't offer such a great boost to the development speed as Bootstrap does.
Animation	Bootstrap uses minimalism in design and boldness in information organisation to provide easy information to users.	Material Design uses eye catching animation, sliders, pop ups, etc in its UI
Documentation	Since Bootstrap is open-sourced, there is ample help in regard with code documentation. Thus, Bootstrap's documentation and support is quite extensive.	Material UI's support is not as great as Bootstrap partly because Material UI is relatively newer.

Final Decision

The skeleton code provided Material UI as the default styling toolkit. However, we decided to switch to Bootstrap as we want to create easy but efficient and highly responsive websites in the shortest amount of time. Bootstrap is relatively consistent and provides a simple, clear interface for developers to use. Material UI focuses more on customizability and animations which are not our main focus for this project.

8 Deployment (Nice to have)

8.1 Requirements

S/N	Functional Requirements	Priority	Status
FR8.1	The system should be able to dockerize each microservice independently	High	Done ▾
FR8.2	The system should be able to run each service as its own container and expose a port externally.	High	Done ▾
FR8.3	The system should be able to deploy on a kubernetes cluster and expose a port internally.	Medium	Done ▾
FR8.4	The system should be deployed onto a production system for access by public users.	High	Done ▾
FR8.5	The system should have an API gateway to decouple client (frontend) from the microservices	High	Done ▾
FR8.6	The system should automatically build, test and deploy when code is pushed to the repository	Medium	Done ▾

8.2 Documentation

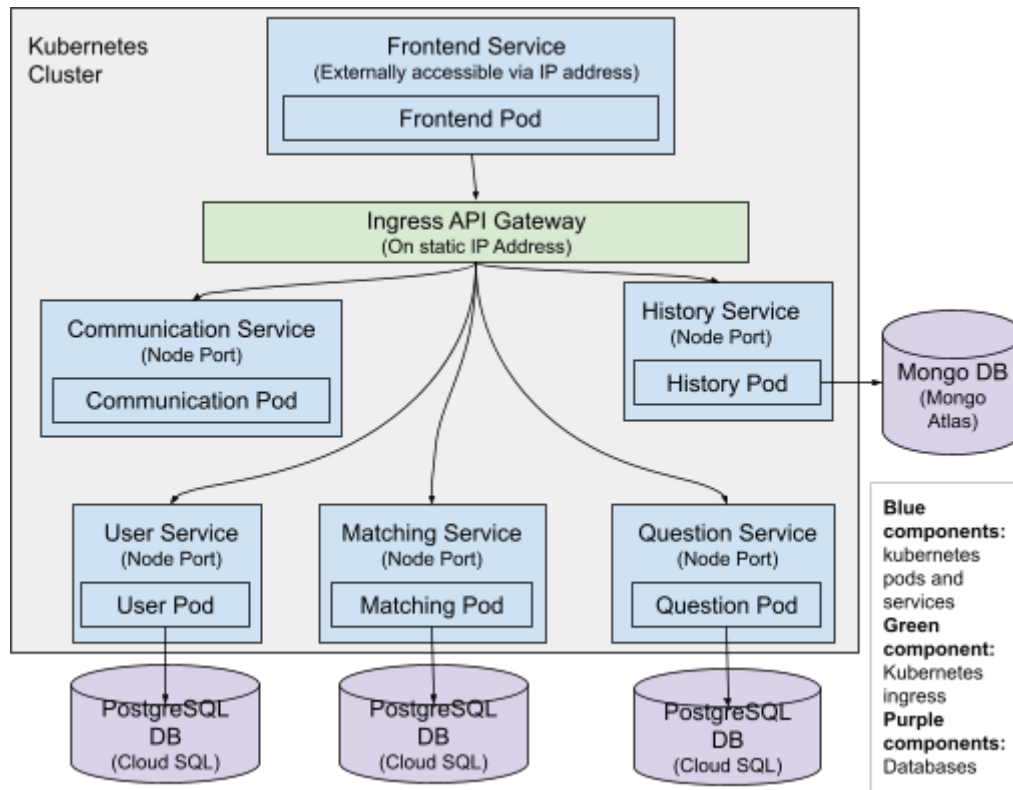


Figure 13: Deployment Architecture

Microservices architecture: Host each microservice on its own pod and service

Since we are using a microservices architecture, we need to ensure each microservice can be deployable individually. This can be easily done with Kubernetes by dockerizing each microservice individually and giving each microservice its own pod and service (Figure 11). Since we designed each microservice to not have any direct communication between each other, we only need to expose each service to the ingress, which acts as the API gateway to our services.

API Gateway

We used an Ingress to act as an API gateway and route API calls to each backend microservice. Each API call sent to the gateway has the service specified in the path name. (Eg. API calls to the question service will be sent to <http://api-gateway-url/question-api/base url>)

This API gateway helps us to encapsulate and hide the internal network structure from the frontend client. This helps us to decouple the frontend from the backend microservices.

Infrastructure as Code: Using Terraform to configure deployment environment

We used Terraform to turn our deployment environment configurations into code. This helped us to collaborate better on the deployment, and also allowed us to reproduce the deployment environment consistently. Reproducing the deployment environment consistently also allowed us to further automate processes such as creating the kubernetes cluster on Google Kubernetes Engine and tearing it down automatically from Github Actions.

Cost Optimizations:

For the SQL databases (hosted on Cloud SQL), we downsized the storage since most of our data is transient (communication, matching, collaboration). User and history service still doesn't need much space currently, so we took the smallest size available. This helped us reduce the projected monthly costs from \$400 to about \$70 for our 3 SQL databases.

For the Kubernetes cluster itself, It is expensive to have it running 24 hours 7 days a week, costing us about \$10 to \$15 each day, which might exceed our budget for the project. Hence we had to take extra steps to optimise the cost for the deployment.

If we take down the cluster each time we test the deployment:

We set up a Github Workflow that will automatically create and set up the cluster according to our terraform configurations that we can run manually when we want to spin up the cluster.

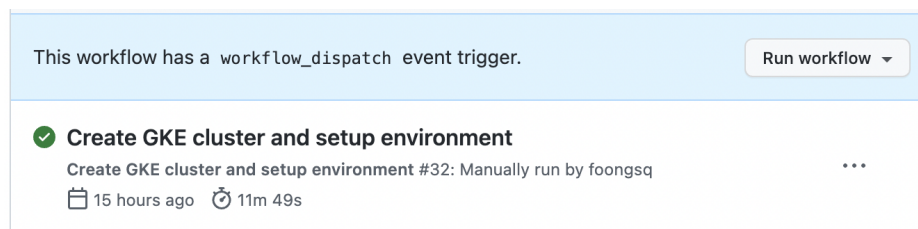


Figure 14: Our Github workflow that can be run manually

This makes it easy such that any member of the team can spin up the cluster and we can be reassured that the configurations will be the same every time.

We have a separate workflow that destroys the cluster that can be manually run or it is also scheduled to run at 2am every night temporarily in case a member forgets to destroy the cluster and prevent us from incurring additional costs.

If we leave the cluster running:

If we want to keep our deployment available, we cannot tear down our cluster each time we are done testing, hence we need to find other ways to optimise cost without taking down the cluster entirely.

We found that our Kubernetes cluster takes the most resources when the deployment pods are running, so we set up cron jobs to scale up (to 1 pod) in the morning each day at 11am and scale down to 0 pods at night at 12am. This will save us almost 12 hours of cost each day, cutting our costs in about a third compared to if we left it running 24 hours.





 Scale up cron job Scale up cron job #10: Scheduled 📅 6 days ago ⌚ 48s	...
 Scale down cron job Scale down cron job #9: Scheduled 📅 7 days ago ⌚ 39s	...
 Scale up cron job Scale up cron job #9: Scheduled 📅 7 days ago ⌚ 44s	...
 Scale down cron job Scale down cron job #8: Scheduled 📅 8 days ago ⌚ 35s	...

Figure 15: Our scaling cron jobs in action

8.3 Design Decisions

Google Kubernetes Engine VS Google App Engine

	Google Kubernetes Engine	Google App Engine
Pros	<ul style="list-style-type: none">• Scalability, easier to scale• Can migrate to other cloud provider easily• Generally lower costs + customizability makes it easier to reduce costs• Faster deployment times	<ul style="list-style-type: none">• Harder to manage, eg. we might need to manually update according to kubernetes versions.
Cons	<ul style="list-style-type: none">• Ease of management• Scales automatically	<ul style="list-style-type: none">• Unable to customise configurations• Higher costs• Slower deployment

Final decision

Considering the requirements for the project, we decided to use Google Kubernetes Engine because of the faster deployment times, greater customizability and better portability. Using kubernetes for our project means that we can easily have a local staging environment to facilitate better testing and we have the option to migrate to a different cloud provider if we have any problems with Google Cloud. This helps us to slightly decouple our deployment from the cloud provider we are using.

Deployment Link (up until Demo on 11 Nov 2022): <http://35.224.11.115:3000/>

Development Process & Project Management

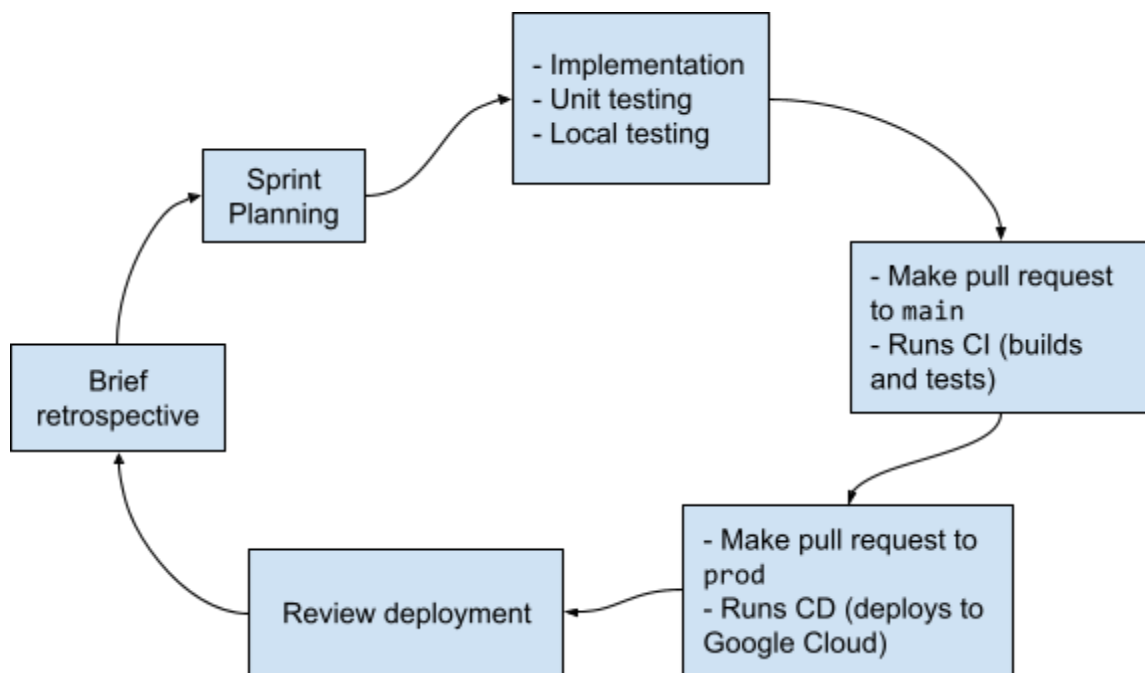


Figure 16: Overview of our weekly workflow

Scrum Methodology

To develop PeerPrep, we modified the Scrum methodology to fit our school schedules. We conducted 1-week sprints instead of the usual longer sprints. Before each sprint, we would prioritise and determine what needs to be done in each sprint and create Github Issues for the tasks. Then we would proceed to do our implementation in 1 week and gather again at the end of the week to do a brief retrospective on the current sprint and planning for the next sprint.

Project Management

We used GitHub's built-in project management tools to aid us in the planning and execution of our project. At the start of the project, we created the 3 milestones and their respective deadlines to track our progress throughout the semester. We also created labels for the respective services and priorities to organise our GitHub issues and pull requests. GitHub Project Board also provides us another view to plan and track our progress with automated workflows to move our cards to the respective columns as we implement our system.

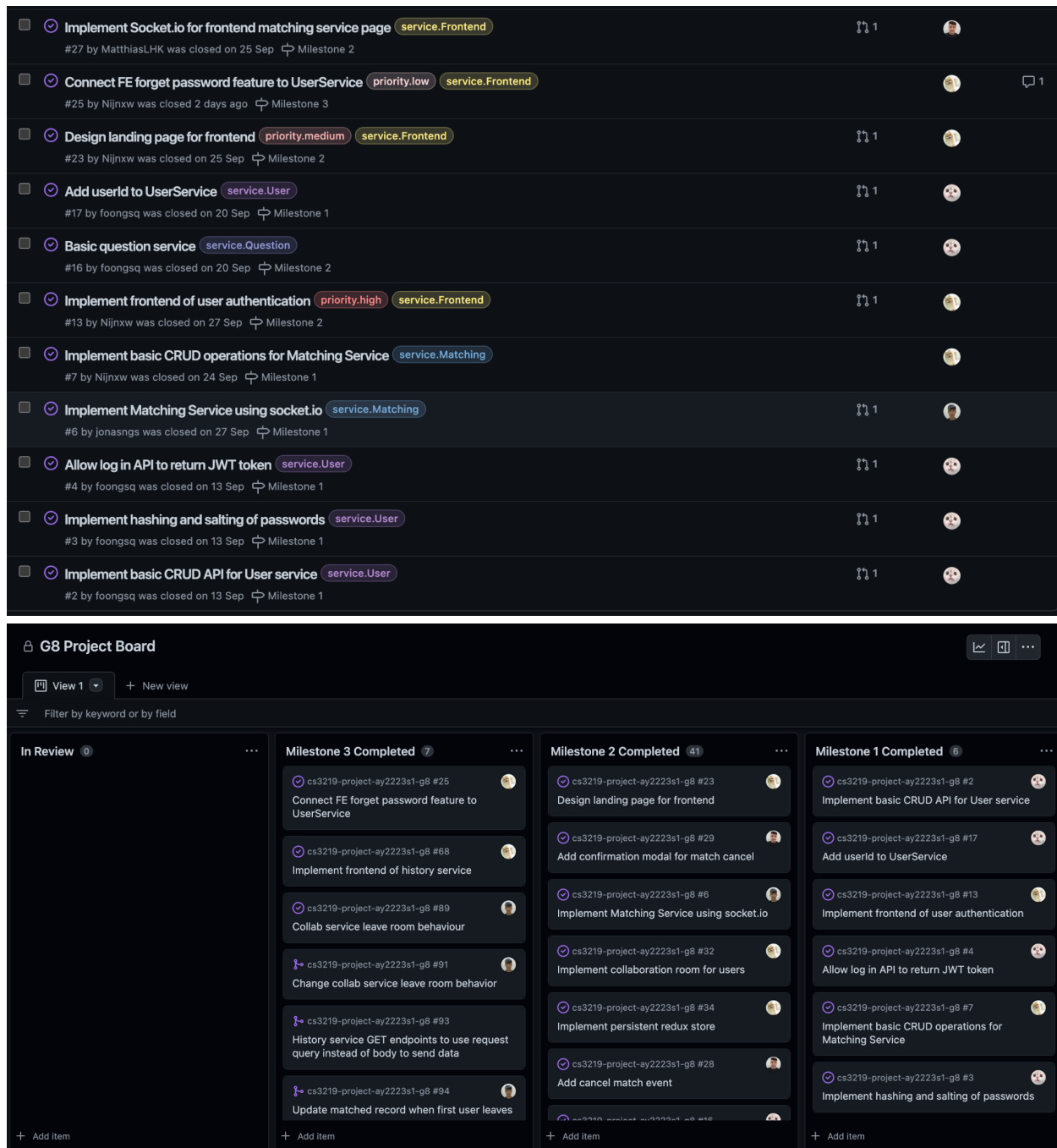


Figure 17: GitHub Issues and Project Board

CICD Pipeline (Nice to have)

We used a CICD pipeline built with GitHub Actions to automatically run our test cases and deploy our application.

Continuous Integration:

In each microservice with code changes, the Github workflow will run the unit test cases for that microservice. Each workflow for the individual microservices are kept separate to adhere to the microservices architecture. If a particular microservice isn't changed in this pull request, it will not be tested. The continuous integration workflow is triggered on pull requests to the branches called **main** or **prod**.

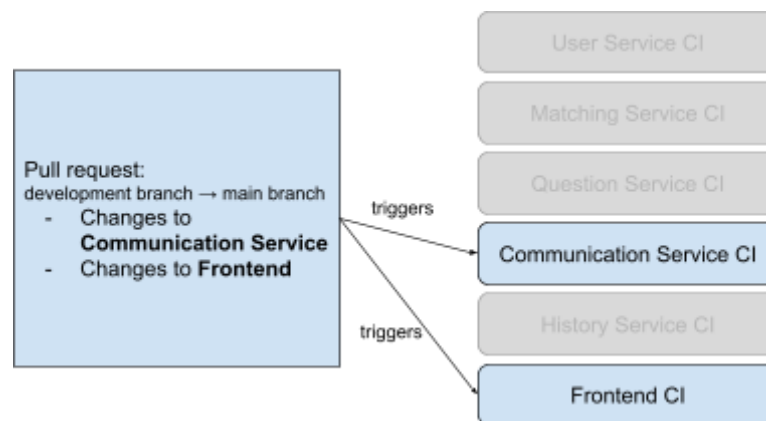


Figure 18: Example of CI triggers on pull request to main branch

Continuous Deployment:

The continuous deployment workflow is triggered on pull request to the **prod** branch, which is the branch we store deployed code. If continuous integration workflows succeed, this workflow will proceed to build the code on Google Cloud using Cloud Build and rollout the updated deployment to our Google Kubernetes Cluster.

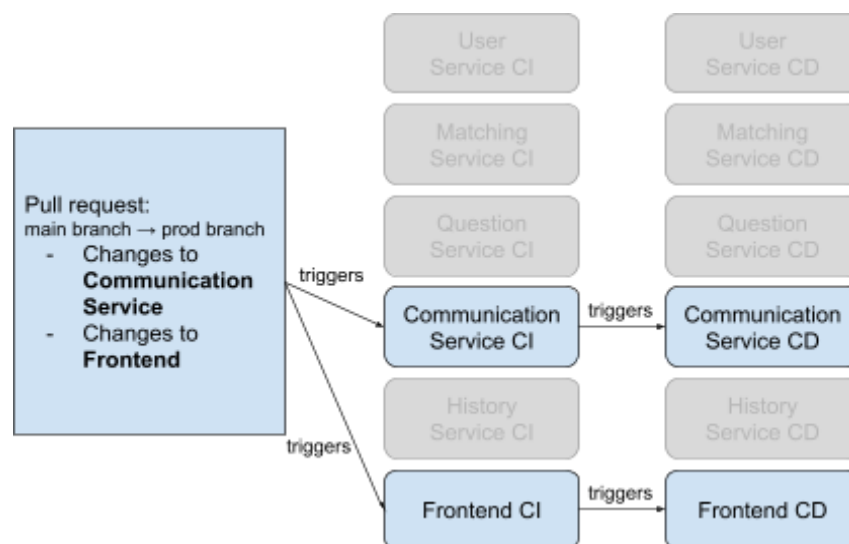


Figure 19: Example of CICD triggers on pull request to prod branch

Project Schedule

Deadlines	Tasks
Milestone 1 (Week 3 - 6)	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> implement MVP: User Service, Matching Service, FE <input checked="" type="checkbox"/> come up with product backlog, FRs and NFRs <input checked="" type="checkbox"/> test integration between services locally <input checked="" type="checkbox"/> deploy frontend, matching and user service to production environment (Docker + Kubernetes)
Milestone 2 (Week 7 - 9)	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> implement Question service together with frontend to go with it <input checked="" type="checkbox"/> implement Collaboration Service together with frontend to go with it <input checked="" type="checkbox"/> implement Communication Service together with frontend to go with it <input checked="" type="checkbox"/> deploy remaining services to production environment (Docker + Kubernetes) <input checked="" type="checkbox"/> Set up CI/CD for running unit test cases and deployment <input checked="" type="checkbox"/> Arrange a technical review with mentor in Week 9
Milestone 3 (Week 10 - 13)	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> implement History Service together with frontend to go with it <input checked="" type="checkbox"/> System testing <input checked="" type="checkbox"/> Write report and README document <input checked="" type="checkbox"/> Prepare for presentation
Deadlines	<p>Code implementation – Wednesday, 9 Nov 2022, 1700 (Wk 13)</p> <p>Project report – Wednesday, 9 Nov 2022, 1700</p> <p>Presentation (20 - 30 mins) – Week 13</p>

Member Contributions

Member	Contributions	
	Technical	Non-Technical
Bu Wen Jin	Frontend	Project Team Leader
	Matching Service	
	History Service	
Li Huankang	Frontend	Team Cohesion IC Devil's Advocate
	Collaboration Service	
Foong Siqi	User Service	Deployment Manager
	Question Service	
	Deployment + CICD	
Jonas Ng Zuo En	Matching Service	Document IC
	Communication Service	

Conclusion

Suggestions for Improvements and Enhancements

Implement our own collaborative backend server

As currently we are relying on the public server for the collaborative text editor to communicate between the different users, it would be good to implement our own service such that it can achieve the same result but with higher reliability. The main reason why we were unable to do so for this project was due to the lack of knowledge on how to properly handle the merge conflicts, as well as the lack of time to experiment with the different algorithms, such as CRDT and OS.

Given more time, we would have liked to find a reliable way to store the intermediate state of the text editor, such that each time a change comes to the server, it will merge with the current stored state and broadcast the new state to all other users.

Using a message broker for communication service

Currently, the communication service leverages Socket.io to implement the chat functionality between 2 users in a room. However, we have also considered the use of message brokers such as Kafka for possible future extensions when the number of users scales. Message brokers can offer the following additional benefits:

1. Message broker would be optimal when the number of users in a collaboration room scales, as message queues would provide better performance during the sending and consuming of a large number of messages.
2. Message broker can allow for decoupling of frontend and communication service.
3. Greater reliability and fault tolerance during the transmission of messages, especially when one part of the system is down.

Refactor Question Logic to improve extensibility using Filter Pattern:

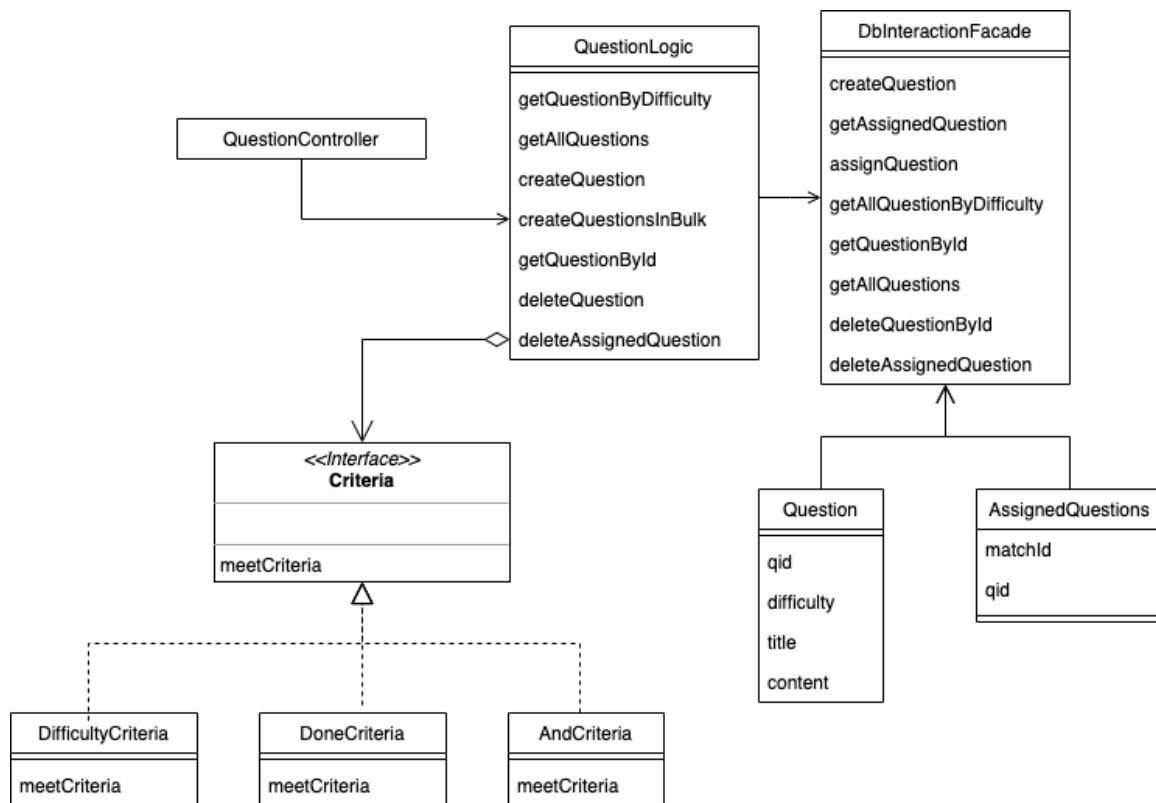


Figure 20: Proposed architecture of Question Service with Facade and Criteria Pattern

Pseudo code for **getRandomQuestion** method:

```
getRandomQuestion() {
    Criteria difficulty = new DifficultyCriteria();
    Criteria done = new DoneCriteria();
    Criteria difficultyAndDone = new AndCriteria(difficulty, done);
    Question[] allQuestions = getAllQuestions();
    Question[] fulfillCriteria = difficultyAndDone.meetCriteria(allQuestions);
    ...
    // choose 1 random question from fulfillCriteria
    return question;
}
```

This filter pattern abstracts out the different criteria that we can filter by into separate classes, and allows us to easily chain more criteria together to get the desired results.

Principles:

- **Open-closed principle.** Easy to extend to filter by more criteria as we can simply add more criteria classes.
- **Single responsibility principle.** Each criteria class is responsible to filter by 1 criteria only and the **QuestionLogic** class no longer has to perform the filtering logic.

Reflections and Learning Points from the Project Process

Reducing dependencies between microservices makes development simpler and faster

Our team managed to develop multiple services in parallel and managed to get an MVP up and running pretty quickly. Subsequent changes that needed to be done to the individual services were also relatively quick and simple. Changes to one service or component also impacted other services very minimally.

Other things to consider for deployment: Cost

One of the bigger issues we faced when deploying was the costs incurred in our production environment. Initially, the space used and compute power we configured our production servers with were way larger and way more than what we needed. We learnt that although we can follow generic guides and tutorials for deployment, we still need to tailor it to our needs in order to optimise for our application better.

Appendix

User Service Documentation

Dev setup

- Set up postgresql on your computer
 - in pgadmin, create a database called `peerprep` and `peerpreptest`
- add `.env` file to `user-service` directory (according to `.env sample`)
- `npm i` to install dependencies
- `npm run dev` to run on localhost
- `npm run test` to run test cases

Tech Stack: PERN

PORT: 8000

Endpoints

Check username: `POST /api/username`

takes	returns
<ul style="list-style-type: none">• username	boolean

- returns true if username is in database and false otherwise

Possible outcomes

Status code	Scenario
200	Success
400	Missing parameters
500	Something wrong with database (likely some connection error)

Create user (Register): `POST /api/user`

takes	returns
<ul style="list-style-type: none">• username• password	<ul style="list-style-type: none">• username• userId• token

- Creates a new user in the database with username and hashed and salted password
- If successful, signs and return a jwt token along with username (jwt token is signed with { id: <USER_ID> })

Possible outcomes

Status code	Scenario
201	Success
409	When username is already in the database
400	Something wrong with username/password (either missing from request or something)
500	Something wrong with database (likely some connection error)

Get user (Login): **POST /api/session**

takes	returns
<ul style="list-style-type: none">• username• password	<ul style="list-style-type: none">• username• userId• token

- Searches for user in database with matching username and password
- if found, signs and return a jwt token

Status code	Scenario
200	Success
400	Something wrong with username/password (either missing from request or incorrect)
500	Something wrong with database (likely some connection error)

Update user (Change password) **PATCH /api/user**

takes	returns
<ul style="list-style-type: none">• token• currPassword• newPassword	-

- Decodes username from token and searches for user with matching currPassword, if all matches, changes password to newPassword

Possible outcomes

Status code	Scenario
200	Success
202	Valid request but no change is made (because old and new password are the same)
400	Something wrong with username/password (either missing from request or incorrect)
500	Something wrong with database (likely some connection error)

Delete user: `DELETE /api/user`

takes	returns
<ul style="list-style-type: none">token	-

- Decodes the username from token and deletes it from database

Possible outcomes

Status code	Scenario
200	Success
400	Something wrong with username (either missing from request or incorrect)
500	Something wrong with database (likely some connection error)

Matching Service Documentation

Dev setup

- Set up postgresql on your computer
 - in pgadmin, create a database called `peerprep` and `peerpreptest`
- add `.env` file to `matching-service` directory (according to `.env sample`)
- In the `matching-service` directory, run:
 - `npm i` to install dependencies
 - `npm run dev` to run on localhost

Stack: PERN

Port: 8001

Database Tables:

1. MatchPotential: Stores records of waiting users
2. Matched: Stores records of matched users

Socket Events

Server Events:

Listen		Emit	
Event Name	Use Case	Event Name	Use Case
<code>find match</code>	Finds a match for the current user	<code>match found</code>	Notifies matched users that both users have been matched
<code>cancel match</code>	Cancels the waiting match for the current user	<code>waiting match</code>	Notifies the current user that match has not been found and is currently waiting for a match
<code>leave room by button</code>	Removes Matched record for the current user when user leaves the collab room	<code>match cancelled</code>	Notifies matched users (if they have been matched) that the match has been cancelled
<code>disconnect</code>	Removes Matched record for the current user when user closes the browser tab	<code>other user left room</code>	Notifies current user that the matched user has left the room
		<code>last user left room</code>	Performs teardown of collab room when all users have left

Client Events:

Listen		Emit	
Event Name	Use Case	Event Name	User Case
match found	Creates new matchId record in redux store	find match	Provides userId and difficulty to server to find a match
waiting match	Notifies user of waiting status	cancel match	Provides userId to server to cancel waiting match
match cancelled	Notifies user that match has been cancelled	leave room by button	Removes Matched record for the current user when user leaves the collab room
other user left room	Notifies current user that the matched user has left the room		
last user left room	Performs teardown of collab room when all users have left		

Question Service Documentation

Dev setup

- Set up postgresql on your computer
 - in pgadmin, create a database called `peerprep` and `peerpreptest`
- add `.env` file to `question-service` directory (according to `.env sample`)
- In the `question-service` directory, run:
 - `npm i` to install dependencies
 - `npm run dev` to run on localhost

Stack: PERN

Port: 8002

Endpoints:

`POST /question-api/random-question` (get question by difficulty)

- this endpoint takes a match ID and difficulty level and returns a question that is of that difficulty level
 - if a question has already been assigned to this `matchId`, the assigned question is returned
 - if a question has not been assigned to this `matchId`, a random question from the question bank that matches the difficulty level is returned
- Example request (Note that the difficulty level must be either `EASY`, `MEDIUM`, `HARD` or `ANY` and in all caps)

```
{
  "matchId": "some-match-id",
  "difficulty": "EASY"
}
```

- Example response

```
{
  "qid": "8f5c49b2-20b6-4545-b04f-8d97a9be66e9",
  "difficulty": "EASY",
  "title": "Two Sum",
  "content": "Given an array of integers nums and an integer target,
return indices of the two numbers such that they add up to target.\n\nYou
may assume that each input would have exactly one solution, and you may not
use the same element twice.\n\nYou can return the answer in any order."
}
```

Status code	Scenario
-------------	----------

200	Success
400	Missing parameters
500	Something wrong with database (likely some connection error)

POST /question-api/question-by-id (get question by ID - for History Service)

- Example request

```
{
  "qid": "some-qid"
}
```

- Example response

```
{
  "qid": "8f5c49b2-20b6-4545-b04f-8d97a9be66e9",
  "difficulty": "EASY",
  "title": "Two Sum",
  "content": "Given an array of integers nums and an integer target,
return indices of the two numbers such that they add up to target.\n\nYou
may assume that each input would have exactly one solution, and you may not
use the same element twice.\n\nYou can return the answer in any order."
}
```

Status code	Scenario
200	Success
400	Missing parameters or invalid qid
500	Something wrong with database (likely some connection error)

GET /question-api/questions (all question)

- (For internal use only)
- Returns all questions in the database, to allow for deletion of questions by id and also for debugging.
- Example request: -
- Example response:

```
[
  {
    "qid": "8f5c49b2-20b6-4545-b04f-8d97a9be66e9",
```

```

    "difficulty": "EASY",
    "title": "Two Sum",
    "content": "Given an array of integers nums and an integer target,
return indices of the two numbers such that they add up to target.\n\nYou
may assume that each input would have exactly one solution, and you may not
use the same element twice.\n\nYou can return the answer in any order."
  },
  {
    "qid": "d6d88e07-32e0-411a-bd85-86a7a557248b",
    "difficulty": "MEDIUM",
    "title": "Add Two Numbers",
    "content": "You are given two non-empty linked lists representing
two non-negative integers. The digits are stored in reverse order, and each
of their nodes contains a single digit. Add the two numbers and return the
sum as a linked list.\n\nYou may assume the two numbers do not contain any
leading zero, except the number 0 itself."
  },
  {
    "qid": "e4b7089e-b7e9-4770-88f0-a6c405ea37d5",
    "difficulty": "HARD",
    "title": "Median of Two Sorted Arrays",
    "content": "Given two sorted arrays nums1 and nums2 of size m and n
respectively, return the median of the two sorted arrays.\n\nThe overall run
time complexity should be O(log (m+n))."
  }
]

```

Status code	Scenario
200	Success
500	Something wrong with database (likely some connection error)

DELETE /question-api/assigned-question (to delete question assigned to match)

- Deletes a question that has been assigned to a match
- Example request:

```

{
  "matchId": "some-match-id"
}

```

- Example response:


```
{
  "message": "Deleted assigned question successfully!"
}
```

Status code	Scenario
200	Success
400	Missing parameters or invalid matchId
500	Something wrong with database (likely some connection error)

POST /question-api/question (to add new question)

- (For internal use only)
- This endpoint allows us to add a question with difficulty and question details.
- Example request:

```
{
  "difficulty": "HARD",
  "title": "Median of Two Sorted Arrays2",
  "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
}
```

- Example response:

```
{
  "qid": "78aaabdf-a5b6-4c7c-9253-229e8f89aba3",
  "difficulty": "HARD",
  "title": "Median of Two Sorted Arrays2",
  "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
}
```

POST /question-api/questions (to add new questions in bulk)

- (For internal use only)
- This endpoint allows us to add questions with difficulty, title and content in bulk.
- If one of the questions causes an error, the rest of the questions that don't cause an error will be added to the database. The first error will be thrown as an error.
- Example request:

```
{
  questions: [
    {
      "difficulty": "HARD",
      "title": "Median of Two Sorted Arrays1",
      "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
    },
    {
      "difficulty": "HARD",
      "title": "Median of Two Sorted Arrays2",
      "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
    }
  ]
}
```

- Example response:

```
[
  {
    "qid": "78aaabdf-a5b6-4c7c-9253-229e8f89aba3",
    "difficulty": "HARD",
    "title": "Median of Two Sorted Arrays2",
    "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
  },
  {
    "qid": "78aaabdf-a5b6-4c7c-9253-229e8f89aba3",
    "difficulty": "HARD",
    "title": "Median of Two Sorted Arrays2",
    "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
  }
]
```

DELETE /question-api/question (to delete question by id)

- (For internal use only)
- Deletes a question
- Example request:

```
{
  "qid": "1c984669-4ed1-4bcb-8492-ad53193e8ebf"
}
```

- Example response:

```
{
  "message": "Deleted question successfully!"
}
```

Question Bank

```
{
  "difficulty": "EASY",
  "title": "Two Sum",
  "content": "Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.\n\nYou may assume that each input would have exactly one solution, and you may not use the same element twice.\n\nYou can return the answer in any order."
}
```

```
{
  "difficulty": "EASY",
  "title": "Merge Two Sorted Lists",
  "content": "You are given the heads of two sorted linked lists list1 and list2.\nMerge the two lists in a one sorted list. The list should be made by splicing together the nodes of the first two lists.\n\nReturn the head of the merged linked list."
}
```

```
{
  "difficulty": "MEDIUM",
  "title": "3Sum",
  "content": "Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.\n\nNotice that the solution set must not contain duplicate triplets.\n\nExample 1:\nInput: nums = [-1,0,1,2,-1,-4]\nOutput: [[-1,-1,2],[-1,0,1]]\nExplanation:\nnums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.\nnums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.\nnums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.\nThe distinct triplets are [-1,0,1] and [-1,-1,2].\nNotice that the order of the output and the order of the triplets does not matter.\n\nExample 2:\nInput: nums = [0,1,1]\nOutput: []\nExplanation: The only possible triplet does not sum up to 0.\n\nExample 3:\nInput: nums = [0,0,0]\nOutput: [[0,0,0]]\nExplanation: The only possible triplet sums up to 0."
}
```

```
{
```

```
    "difficulty": "MEDIUM",
    "title": "Divide Two Integers",
    "content": "Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator.\n\nThe integer division should truncate toward zero, which means losing its fractional part. For example, 8.345 would be truncated to 8, and -2.7335 would be truncated to -2.\n\nReturn the quotient after dividing dividend by divisor.\n\nNote: Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: [-231, 231 - 1]. For this problem, if the quotient is strictly greater than 231 - 1, then return 231 - 1, and if the quotient is strictly less than -231, then return -231."
}
```

```
{
    "difficulty": "HARD",
    "title": "Median of Two Sorted Arrays2",
    "content": "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.\n\nThe overall run time complexity should be O(log (m+n))."
}
```

```
{
    "difficulty": "HARD",
    "title": "Longest Valid Parentheses",
    "content": "Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.\n\n\nExample 1:\n\nInput: s = \"(()\"\nOutput: 2\nExplanation: The longest valid parentheses substring is \"()\".\n\nExample 2:\n\nInput: s = \"()()\"\nOutput: 4\nExplanation: The longest valid parentheses substring is \"()()\".\n\nExample 3:\n\nInput: s = \"\"\nOutput: 0\n\nConstraints:\n\n0 <= s.length <= 3 * 104\ns[i] is '(', or ')\"."
}
```

Communication Service Documentation

Dev setup:

- In the communication-service directory, run:
 - `npm i` to install dependencies
 - `npm run dev` to run on localhost

Port: 8005

Communication service leverages Socket.io for client-server communication during the use of PeerPrep's chat functionality.

Only matched users in the Collaboration room will be able to send and receive messages to each other.

The following describes all the events that the client and server emits and listens for:

Socket Events:

Server Events:

Listen		Emit	
Event Name	Use Case	Event Name	Use Case
join chat	Creates a new socket room which matched user's socket will join, enabling communication between matched users	receive message	Notifies the client side that the message has been emitted to matched users and delivers the message to both matched users in the room
send message	Receives a message from the client side and emits the message to the socket room		

Client Events:

Listen		Emit	
Event Name	Use Case	Event Name	User Case
receive message	Receives a message from the server and appends the new message to the list of all messages in the chat box for subsequent rendering	send message	Delivers message to server for subsequent transmission to all matched users in the room
		join chat	Provides matchId to server for creation of socket room to enable transmission of messages

History Service Documentation

Dev setup:

- add `.env` file to `history-service` directory (according to `.env-sample`)
 - the values for the variables are as stated below
- In the `history-service` directory, run:
 - `npm i` to install dependencies
 - `npm run dev` to run on localhost

Tech Stack: MERN

PORT: 8004

Mongo Atlas: group gmail account

MongoDB Uri for development:

`mongodb+srv://peerprep:cs3219-group8@cluster0.letyk1a.mongodb.net/HistoryDB?retryWrites=true&w=majority`

MongoDB Uri for production:

`mongodb+srv://peerprep:cs3219-group8@cluster0.letyk1a.mongodb.net/HistoryDB-Prod?retryWrites=true&w=majority`

Data structure:

```
{
  uid: String,
  attempts: [
    {
      qid: String,
      content: String,
      attemptDate: Date,
    }
  ],
}
```

Status code meanings

Status code	Scenario
200	Success
400	Missing parameters / invalid attempt structure
409	Duplicate history (duplicate uid)
500	Something wrong with database (likely some connection error)

Endpoints

GET /history-api/history/all (get all history)

- Returns all the histories in the database (Internal use)
- Example response

```
{
  "message": "All history retrieved successfully!",
  "data": [
    {
      "uid": "user-1",
      "attempts": [
        {
          "qid": "question-1",
          "content": "console.log(\"Hello world!\");",
          "attemptDate": "2022-10-09T15:10:36.244Z"
        }
      ]
    },
    {
      "uid": "user-2",
      "attempts": [
        {
          "qid": "question-1",
          "content": "console.log(\"Hello world!\");",
          "attemptDate": "2022-10-09T15:10:40.480Z"
        }
      ]
    }
  ]
}
```

GET /history-api/history (get history by uid)

- Returns a user's history
 - if no user's history, creates a history with no attempts and returns it.
- Example request

```
{
  "uid": "user-1"
}
```

- Example response

```
{
```

```

    "message": "User history retrieved successfully!",
    "data": {
      "uid": "user-1",
      "attempts": [
        {
          "qid": "question-1",
          "content": "console.log(\"Hello world!\");",
          "attemptDate": "2022-10-10T06:05:45.454Z"
        }
      ]
    }
  }
}

```

GET /history-api/history/attempt (get an attempt by uid and qid)

- Returns a user's **latest** attempt to a specific question
- Example request

```

{
  "uid": "user-1",
  "qid": "question-1"
}

```

- Example response

```

{
  "message": "User attempt retrieved successfully!",
  "data": {
    "qid": "question-1",
    "content": "console.log(\"Hello world!\");",
    "attemptDate": "2022-10-10T06:05:45.454Z"
  }
}

```

POST /history-api/history (create a new history)

- Creates a new user's history
- Example request

```

{
  "uid": "user-1",
  "attempts": [
    {
      "qid": "question-1",
      "content": "console.log(\"Hello world!\");"
    }
  ]
}

```



```
    }  
  ]  
}
```

- Example response

```
{  
  "message": "User history created successfully!",  
  "data": {  
    "uid": "user-1",  
    "attempts": [  
      {  
        "qid": "question-1",  
        "content": "console.log(\"Hello world!\");",  
        "attemptDate": "2022-10-10T06:05:45.454Z"  
      }  
    ]  
  }  
}
```

POST /history-api/history/attempt (add a question attempt for a user)

- Returns a user's **latest** attempt to a specific question
- Example request

```
{  
  "uid": "user-1",  
  "attempt": {  
    "qid": "qn1",  
    "content": "// NEW ATTEMPT console.log(\"Hello world!\");"  
  }  
}
```

- Example response

```
{  
  "message": "User attempt added successfully!",  
  "data": {  
    "uid": "user-1",  
    "attempts": [  
      {  
        "qid": "question-1",  
        "content": "// NEW ATTEMPT console.log(\"Hello world!\");",  
        "attemptDate": "2022-10-10T06:23:33.579Z"  
      }  
    ]  
  }  
}
```

```
}
  ]
}
}
```

`DELETE /history-api/history/all` (delete all history)

- Deletes all histories in the database (Internal use)
- Example response

```
{
  "message": "All user history deleted successfully!",
  "data": {
    "acknowledged": true,
    "deletedCount": 1
  }
}
```

`DELETE /history-api/history` (deletes a history by uid)

- Deletes a user's history (Internal use)
- Example request

```
{
  "uid": "user-1"
}
```

- Example response

```
{
  "message": "User history deleted successfully!",
  "data": {
    "uid": "user-1",
    "attempts": [
      {
        "qid": "question-1",
        "content": "// NEW ATTEMPT console.log(\"Hello world!\");",
        "attemptDate": "2022-10-10T06:23:33.579Z"
      }
    ]
  }
}
```