NUS
National University of Singapore

School of Computing

CS3219 Software Engineering Principles and Patterns

AY22/23 Semester 1

Project Report

Group 25

Team Members	**Student No.**	**Email**
Ajjagottu Kaushik Kumar Reddy	A0214112A	e0516181@u.nus.edu
Li KangLi	A0218386W	lkl@u.nus.edu
Lim Wan Ning	A0221393L	e0556585@u.nus.edu
Liong Wen Xuan	A0220924M	e0556116@u.nus.edu

Table of Contents

1. Background and purpose of the project

As students ourselves, we understand the challenges of job applications and preparing for technical interviews. It can be a frustrating experience failing to answer the question in time or being unable to explain your thoughts to the interviewer while coding.

With such a wide variety of interview questions, it is impossible to memorise the solutions to all of them. A more efficient approach would be to regularly practice questions from different categories of questions, to better identify the approach one should take in the interview. However, grinding these questions can be a tiresome and lonely process.

This is where our project, PeerPrep comes in. The purpose of PeerPrep is to tackle the problems identified above by pairing students together in mock technical interviews. This helps students to be better prepared as it simulates a real interview where there is another party observing.

PeerPrep enables users to choose the category and difficulty of questions they want to focus on, allowing them to improve on their weaker areas. Matching them with other students who chose the same level allows them to learn from each other's approach to the same coding question. They are able to view each other's code live through the real-time collaborative rich-text editor, which is often used in real technical interviews over a code editor. Users can also communicate to discuss their approaches or seek clarifications from each other through the real-time chat box.

Pairing users together helps to break the monotony of studying for interviews alone and can be an effective learning tool through the use of PeerPrep's features.

2. Individual contributions

Member	Technical	Non-technical
Ajjagottu Kaushik Kumar Reddy	Implemented chat service and its corresponding frontend.	Contributed equally to the report
Li KangLi	Implemented user service and its corresponding frontend.	
Lim Wan Ning	Implemented matching and question service and its corresponding frontend.	
Liong Wen Xuan	Implemented collaboration service and its corresponding frontend.	

3. Product Requirements

The following table highlights the functional requirements for our product, grouped according to their microservice:

- User Service
- Matching Service
- Question Service
- Collaboration Service
- Chat Service
- UI

S/N	Functional Requirement	Priority
	User Service	
FR1.1	The system should allow users to create an account with an email and password.	High
FR1.2	The system should ensure that every account created has a unique email.	High
FR1.3	The system should allow users to log into their accounts by entering their email and password.	High
FR1.4	The system should allow users to log out of their accounts.	High
FR1.5	The system should allow users to reset their password.	Medium
	Matching Service	
FR2.1	The system should allow users to select the difficulty level and category of the questions they wish to attempt.	High
FR2.2	The system should be able to match two waiting users with similar difficulty levels and categories and put them in the same room.	High
FR2.3	If there is a valid match, the system should match the users within 30s.	High
FR2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
FR2.5	The system should provide a means for the user to leave a room once matched.	Medium
FR2.6	The system should inform the other user if one user leaves the room.	Medium
FR2.7	The system should ensure that users cannot be matched to themselves.	High

FR2.8	The system should ensure that users can only open one window of the application at any time.	Medium
Question Service		
FR3.1	The system should have a question bank that stores all questions.	High
FR3.2	The system should randomly choose a question from the question bank for the user to work on.	High
FR3.3	The system should provide different difficulties (e.g. easy, medium, hard) for the questions.	High
FR3.4	The system should provide different categories (e.g. Algorithms, Databases, etc.) for the questions	Medium
FR3.5	The system should provide a solution to the questions once both students end the session.	Low
Collaboration Service		
FR4.1	The system should allow two users to work on the same set of answers for a particular question.	High
FR4.2	The system should update in real time the work done by the other user.	High
FR4.3	The system should use a rich text editor to allow better formatting of the code.	Medium
FR4.4	The system should save the text editor occasionally to prevent loss of work by the users.	Medium
FR4.5	The system should alert the user if the other user disconnects halfway and allow them to resume themselves.	Medium
Chat Service		
FR5.1	The system should update the chat between two users in real time.	High
FR5.2	The system should allow the users to chat while answering the question.	High
FR5.3	The system should allow the users to save the chat locally.	Medium
FR5.4	The system should not allow users to chat if there is a disconnect.	Medium
FR5.5	The system should inform users when users connect and disconnect.	Medium

UI		
FR6.1	The system should have a UI for users to login.	High
FR6.2	The system should have a UI for users to sign up.	High
FR6.3	The system should have a UI to allow users to choose their desired difficulty level and category.	High
FR6.4	The system should have a UI to allow users to collaborate with the other user and work on the question together.	High
FR6.5	The system should have a UI to allow users to chat while attempting the question.	High
FR6.6	The system should have a UI to allow users to choose the category of questions they want to attempt.	High
FR6.7	The system should have a UI waiting screen with a countdown timer when it is looking for a match.	High
FR6.8	The system should display the questions in a human-readable format (i.e. code in code blocks, etc).	Medium
FR6.9	The system should work on both mobile and web.	Low

The following table highlights the non-functional requirements of our product.

S/N	Quality	Non-Functional Requirement	Priority
NFR1	Security	The system should ensure that the user database is stored securely.	High
NFR2	Security	The system should ensure any communications between the frontend and backend to be secured with TLS.	High
NFR3	Usability	The system shall allow the use of common keyboard shortcuts (ctrl-c, ctrl-v) during the collaboration on answers.	High
NFR4	Modifiability	The code shall be easily modifiable by maintainer programmers with 1 hour or less of development effort.	Medium
NFR5	Portability	The system can run in multiple environments, such as Windows, Mac and Linux.	High
NFR6	Performance	The system shall redirect users to the room within 5 seconds after matching.	Medium

4. Developer Documentation

In this section of the documentation, we will delve into the development and deployment processes, and design decisions.

4.1. Development Process

4.1.1. Tech Stack

Our development tech stack is as follows:

	Technology required	**Utilized by (Services)**
Frontend	React, Material UI	N.A.
Backend	Express.js, Node.js	All
Database	Mongoose (MongoDB)	Question
	SQLite	Matching
	Redis	Question
Deployment	Amazon S3, Amazon CloudFront, Amazon EC2, AWS Lambda	N.A.
Pub-Sub Messaging	Socket.IO	Matching, Chat, Collaboration
	Redis	Matching, Question
Authentication and Authorisation	Auth0	User
Testing	Mocha, Chai, Jest, Postman	Matching, Question, Collaboration, Chat
CI/CD	GitHub Actions	N.A.
Orchestration Service	Docker-Compose	N.A.
Project Management Tools	GitHub Issues	N.A.

4.1.2. Architecture Diagrams

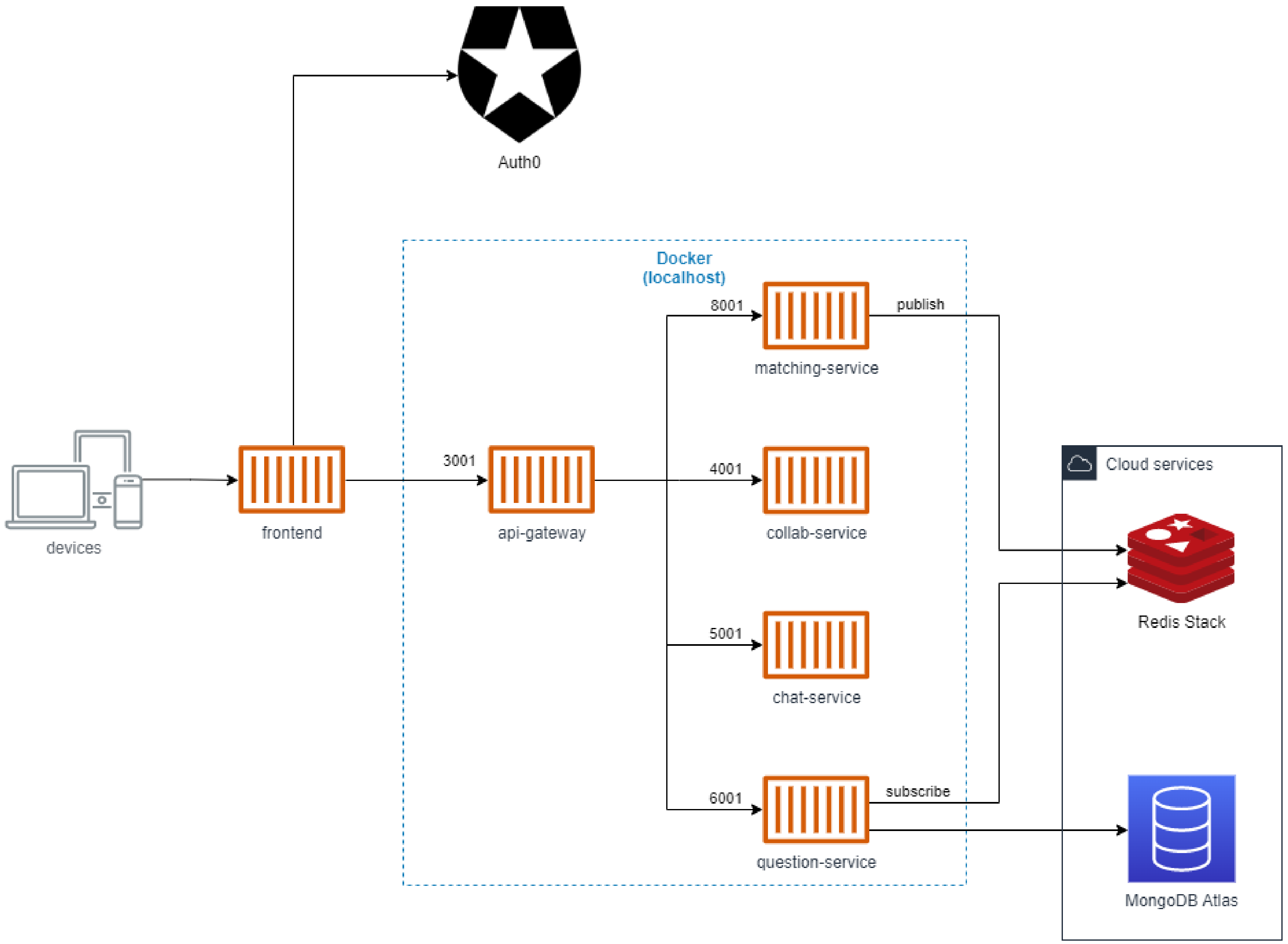


Figure 1: Local Deployment Architecture

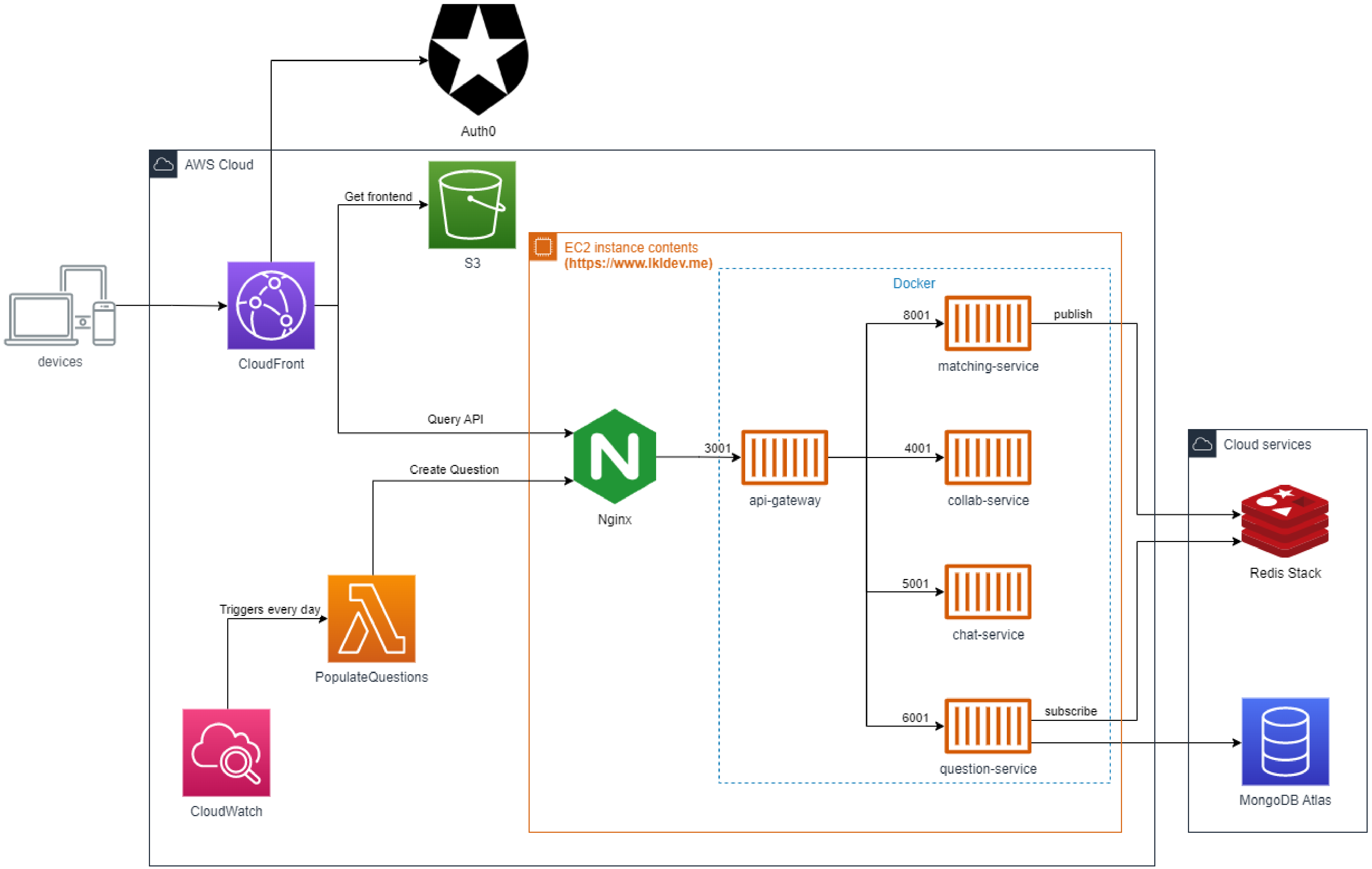


Figure 2: Cloud Deployment Architecture

4.2. Deployment Process

4.2.1. Deployment Requirements

Our team has fulfilled several deployment requirements.

Initially, we deployed the application on our local machine using native technology stack (using Node.js and Express for the server and React as the frontend).

Next, we dockerized each microservice, using Docker-compose as our orchestration service to manage containers on a single machine. Since the frontend will interact with the API gateway separately, we decided to leave the frontend out of our orchestration service.

We built an API gateway that redirects requests to the relevant microservices. This gateway listens on port 3001, and redirects requests to the collaboration service (4001), chat service (5001), question service (6001), and matching service (8001) on different ports. All the services are configured with our orchestration service to start and stop together with the API gateway.

We also deployed the app onto the AWS cloud. To take advantage of the cloud, we separated the hosting of our frontend and our backend. Our frontend is compiled and stored in an Amazon S3 bucket. The frontend is accessible worldwide as it uses Amazon CloudFront, a content delivery network. Amazon CloudFront also provides a secure connection to the frontend, which is required from our user service, Auth0.

Our backend is hosted on Amazon EC2. Since we have previously dockerized our services, moving the backend from local to Amazon EC2 is relatively easy, as we just have to install docker and our backend is good to go. However, to ensure that our service is able to handle extra workload, we placed a nginx server in front of our application to act as a reverse proxy and handle all the requests from the public. We installed a SSL certificate on our EC2 machine, to ensure that the connection from our frontend to our backend is secured. To populate our question database, we created a serverless AWS Lambda function that is triggered every day by Amazon Cloudwatch to scrape the question bank and update our question service via the API gateway.

We also wrote tests for several microservices and demonstrated continuous integration (CI) using GitHub Actions.

4.2.2. Deployment Considerations

When planning our deployment strategy, we realized that there are numerous things to consider as it is not as straightforward as deploying locally.

Our team considered whether we should provide scalability for our solution, but we realized there are a number of hurdles we would have to go through to provide a scalable solution:

- Since our solution utilizes sockets, we have to ensure that each client connects to the correct socket.IO server to establish a room connection, or we provide a pub sub service to allow the socket.IO servers to publish and subscribe the data to act as a collective "database" for all server sockets.
- We will have to create load balancers to balance the traffic sent to all the socket.IO servers available.
- Our matching service currently utilizes a local database, which would require changing in order to scale.

Due to time constraints, we decided to forgo making our solution scalable, since we did not take product scalability into consideration in our initial development of PeerPrep, and it is not a priority requirement that we would like to focus on.

As for our deployment process, our initial plan was to deploy our whole backend using serverless functions, as a serverless architecture would be much less costly. However, we realized that in order to turn our services into serverless functions, we would have to adopt the native web socket in order to be compatible with Amazon Web Socket API Gateway. Our current implementation of using Socket.IO does not use native web socket, as it will always try to connect to the Socket.IO server using HTTP long polling before upgrading the connection into a web socket. We considered using Amazon HTTP API Gateway, but the sheer amount of HTTP requests sent between the Socket.IO server and client renders this approach redundant (the cost will not justify adopting a serverless architecture).

Our next plan was to use Amazon Elastic Container Service (ECS) to run our services, since we have dockerized all our applications. However, we realized that this would still be an issue as our API gateway requires modifications in order to interact with the containers in Amazon ECS. Moreover, to interact with the containers, there needs to be a load balancer in front of each container, and more containers may be created to handle increased workload, which will result in the scalability issue that we mentioned above.

Our final plan is to deploy our whole solution onto Amazon EC2, and to run our local deployment on the cloud virtual machine. However, we realized that such a deployment would not take advantage of cloud services. Hence, we decided to separate the frontend and backend (as described in 4.2.1). While we managed to successfully deploy our solution (based on Figure 2), we encountered a few issues along the way:

- Our deployment needs to provide a secure connection in order for our user service to work. We had to figure out how to provide a secure connection, including generating a SSL certificate and providing it to our EC2 server.
- As our frontend and backend are now essentially on different domains, we had to spend some time trying to understand how CORS works.
- Our serverless function is not idempotent, which resulted in the same question being created multiple times. Our workaround was to ensure that the function would only run once a day, and to disable the retry attempt if the function fails to run.

As mentioned in 4.2.1, we successfully implemented CI with GitHub workflows. However, our current deployment makes it difficult to implement Continuous Deployment (CD), as there are numerous places that the code should be deployed to. This is the CD strategy we planned to deploy our code:

- Our frontend needs to be deployed to Amazon S3 and we need to invalidate the cache in AWS CloudFront.
- Our backend needs to be deployed to Amazon EC2, and rebuild the docker images before restarting the services.
- Our serverless function needs to be deployed to AWS Lambda.

Our research suggests that this deployment strategy works with GitHub workflow, but we did not have the time to try out if we will be able to implement it successfully.

4.3. Design Decisions

4.3.1. Monolith vs Microservices

A monolith is a single application with a large codebase whereas microservice architecture is made up of multiple smaller services. The advantages of a monolith include easier deployment and development as there is only one single codebase to maintain. The disadvantages of a monolith, however, include tight coupling between components. This can be a problem as tight coupling makes the system less flexible to changes.

Microservices architecture helps to address these issues. By separating the logic into smaller microservices, the degree of coupling is decreased making the system more open to changes. This helps in the development and deployment process of the application as well, since each microservice can implement its own tests, tech stack, and deployment without being concerned with the other microservices. Maintainability increases with low coupling, and the effort required to fix bugs or add new features (extensibility) is reduced as well. Microservice also provide the additional flexibility to scale, which would have been more difficult with a monolithic architecture. Thus, our team decided to adopt the microservice architecture.

Our initial approach only had three microservices, matching, question and user services. We planned to create chat and collaboration services such that they utilize the same socket that the matching service uses to match two users together. However, we realized that this would result in a very tightly coupled solution, and changes to matching service may break either the chat or collaboration service. Hence, we decided to uncouple them, and have five separate microservices instead.

4.3.2. API Gateway

The use of the façade pattern can be observed from the use of an API-gateway service to provide a unified interface and increase the ease of use of each subsystem.

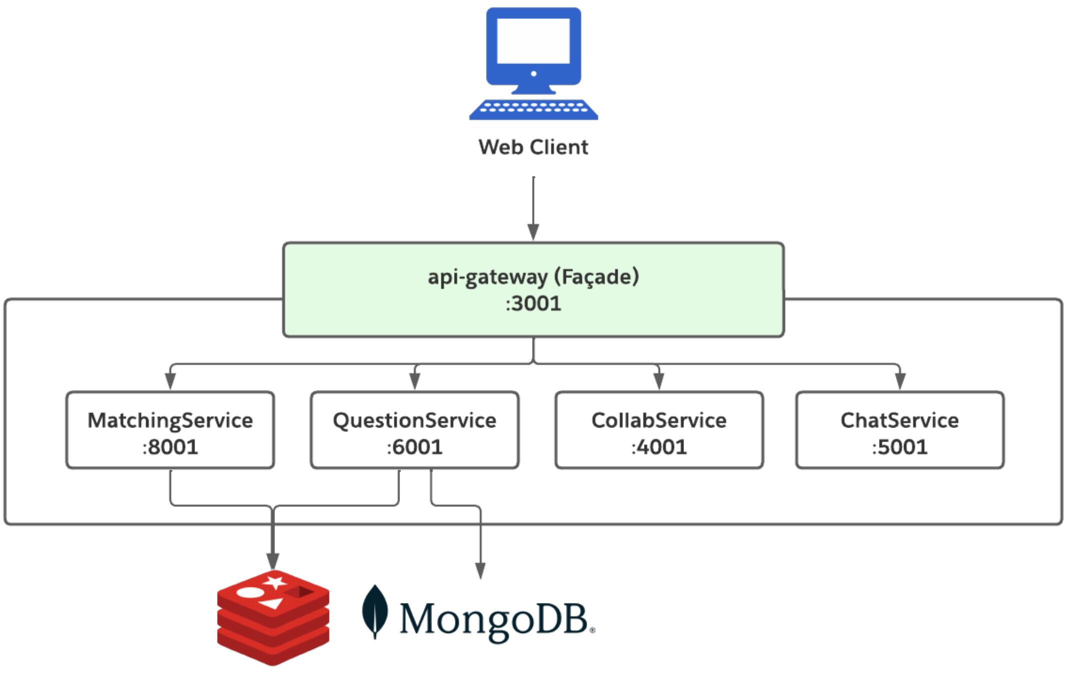


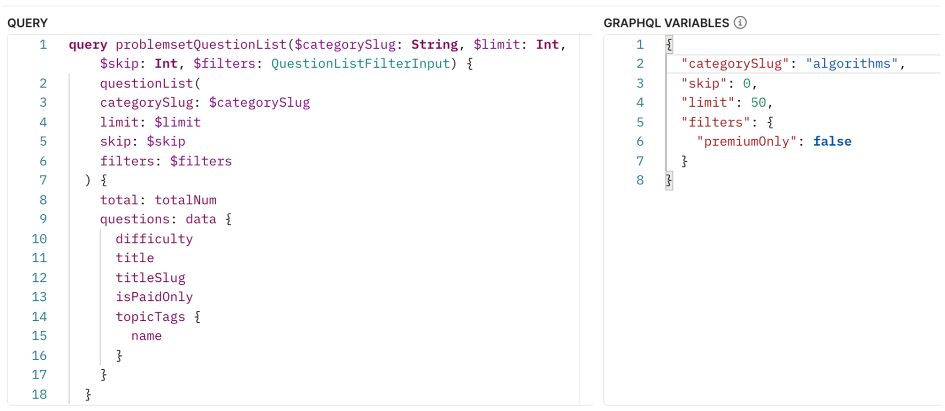
Figure 3: How Facade Pattern is used in API Gateway

The desktop *client* requests the *façade* to perform some action, in which the API-gateway (*façade*) will delegate the request to the appropriate service. This design pattern is useful to isolate our services from the client's perspective. The client would only be required to know one address, which is that of the api-gateway, instead of having to know the address of all different services. Having this API gateway would also make authentication much easier, as the gateway is able to ensure that the request is authenticated before allowing them to access our internal services. We configured our API gateway to only allow authenticated users to access, any request to it will require a JWT that is signed by Auth0.

4.3.3. Question Generation

Our initial plan was to create a frontend for administrators to create the questions that users can work on. However, we realised that there needs to be a question bank to generate initial questions first. Hence, we focused on creating a serverless function that query from question banks available in public and use them to populate our database. This would allow us to provide the users with popular interview questions. Our deployed serverless function queries data from Leetcode's GraphQL API.

Firstly, we query for the problemsetQuestionList, taking in the `categorySlug` variable as one of the four available categories on Leetcode– algorithms, databases, shell and concurrency.

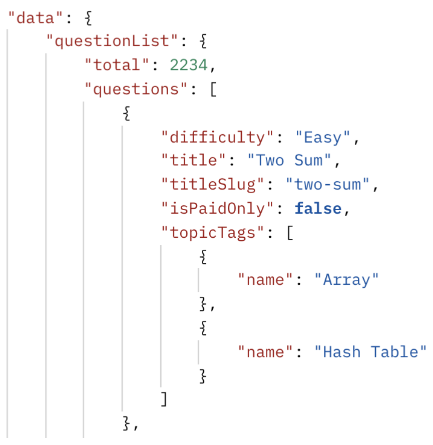

QUERY

```
query problemsetQuestionList($categorySlug: String, $limit: Int,
    $skip: Int, $filters: QuestionListFilterInput) {
    questionList(
    categorySlug: $categorySlug
    limit: $limit
    skip: $skip
    filters: $filters
  ) {
    total: totalNum
    questions: data {
      difficulty
      title
      titleSlug
      isPaidOnly
      topicTags {
        name
      }
    }
  }
```

GRAPHQL VARIABLES

```
{
  "categorySlug": "algorithms",
  "skip": 0,
  "limit": 50,
  "filters": {
    "premiumOnly": false
  }
}
```

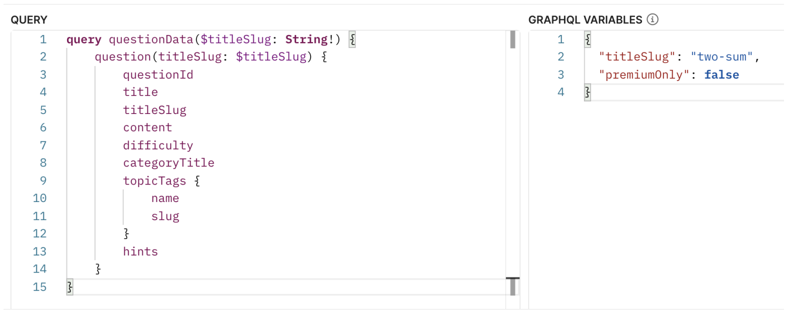
Figure 4: problemsetQuestionList query with 'algorithms' categoryTitle



```
"data": {
    "questionList": {
        "total": 2234,
        "questions": [
            {
                "difficulty": "Easy",
                "title": "Two Sum",
                "titleSlug": "two-sum",
                "isPaidOnly": false,
                "topicTags": [
                    {
                        "name": "Array"
                    },
                    {
                        "name": "Hash Table"
                    }
                ]
            },
```

Figure 5: Output when querying for 'algorithms'

Using the questionList from the previous step, we query each individual question based on their `titleSlug` variable.



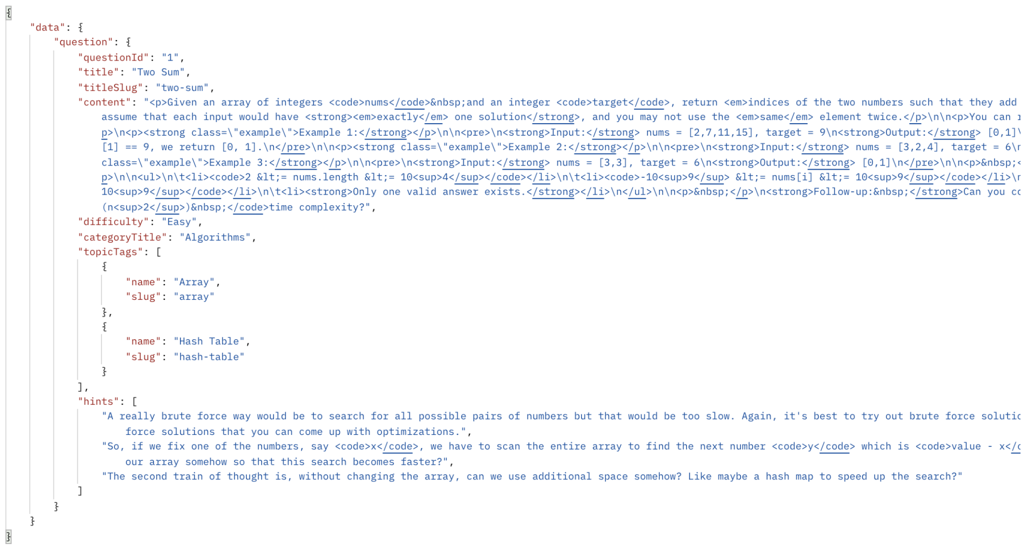
QUERY

```
query questionData($titleSlug: String!) {
    question(titleSlug: $titleSlug) {
        questionId
        title
        titleSlug
        content
        difficulty
        categoryTitle
        topicTags {
            name
            slug
        }
        hints
    }
}
```

GRAPHQL VARIABLES

```
{
  "titleSlug": "two-sum",
  "premiumOnly": false
}
```

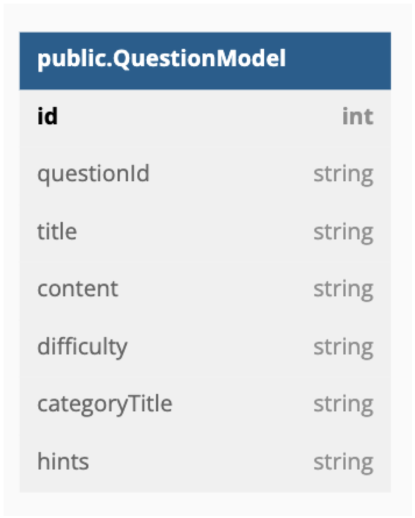
Figure 6: questionData query for 'two-sum' title



```
{
    "data": {
        "question": {
            "questionId": "1",
            "title": "Two Sum",
            "titleSlug": "two-sum",
            "content": "<p>Given an array of integers <code>nums</code>&nbsp;and an integer <code>target</code>, return <em>indices of the two numbers such that they add
                assume that each input would have <strong><em>exactly</em> one solution</strong>, and you may not use the <em>same</em> element twice.</p>\n\n<p>You can
                p>\n<p><strong class=\"example\">Example 1:</strong></p>\n\n<pre>\n<strong>Input:</strong> nums = [2,7,11,15], target = 9\n<strong>Output:</strong> [0,1]\
                [1] == 9, we return [0, 1].\n</pre>\n\n<p><strong class=\"example\">Example 2:</strong></p>\n\n<pre>\n<strong>Input:</strong> nums = [3,2,4], target = 6\
                class=\"example\">Example 3:</strong></p>\n\n<pre>\n<strong>Input:</strong> nums = [3,3], target = 6\n<strong>Output:</strong> [0,1]\n</pre>\n\n<p>&nbsp;<
                p>\n\n<ul>\n\t<li><code>2 &lt;= nums.length &lt;= 10<sup>4</sup></code></li>\n\t<li><code>-10<sup>9</sup> &lt;= nums[i] &lt;= 10<sup>9</sup></code></li>\
                10<sup>9</sup></code></li>\n\t<li><strong>Only one valid answer exists.</strong></li>\n</ul>\n\n<p>&nbsp;</p>\n<strong>Follow-up:&nbsp;</strong>Can you co
                (n<sup>2</sup>)&nbsp;</code>time complexity?",
            "difficulty": "Easy",
            "categoryTitle": "Algorithms",
            "topicTags": [
                {
                    "name": "Array",
                    "slug": "array"
                },
                {
                    "name": "Hash Table",
                    "slug": "hash-table"
                }
            ],
            "hints": [
                "A really brute force way would be to search for all possible pairs of numbers but that would be too slow. Again, it's best to try out brute force solutic
                    force solutions that you can come up with optimizations.",
                "So, if we fix one of the numbers, say <code>x</code>, we have to scan the entire array to find the next number <code>y</code> which is <code>value - x</c
                    our array somehow so that this search becomes faster?",
                "The second train of thought is, without changing the array, can we use additional space somehow? Like maybe a hash map to speed up the search?"
            ]
        }
    }
}
```

Figure 7: Output when querying for 'two-sum'

These questions will then be stored in our database, accessible by our QuestionService's endpoints. Similar to Leetcode, our QuestionModel utilises the same variable names as convention.



public.QuestionModel	
id	int
questionId	string
title	string
content	string
difficulty	string
categoryTitle	string
hints	string

Figure 8: MongoDB model for QuestionService

The above steps to scrape Leetcode's questions are configured to run once everyday. The code compares the scraped question with those in the current database (through the `/getAllQuestions` endpoint), filters all the questions that exist in the database, and adds those questions that do not exist in our database (through the `/createQuestion` endpoint).

The following sequence diagram illustrates how the serverless function populates the Question database.

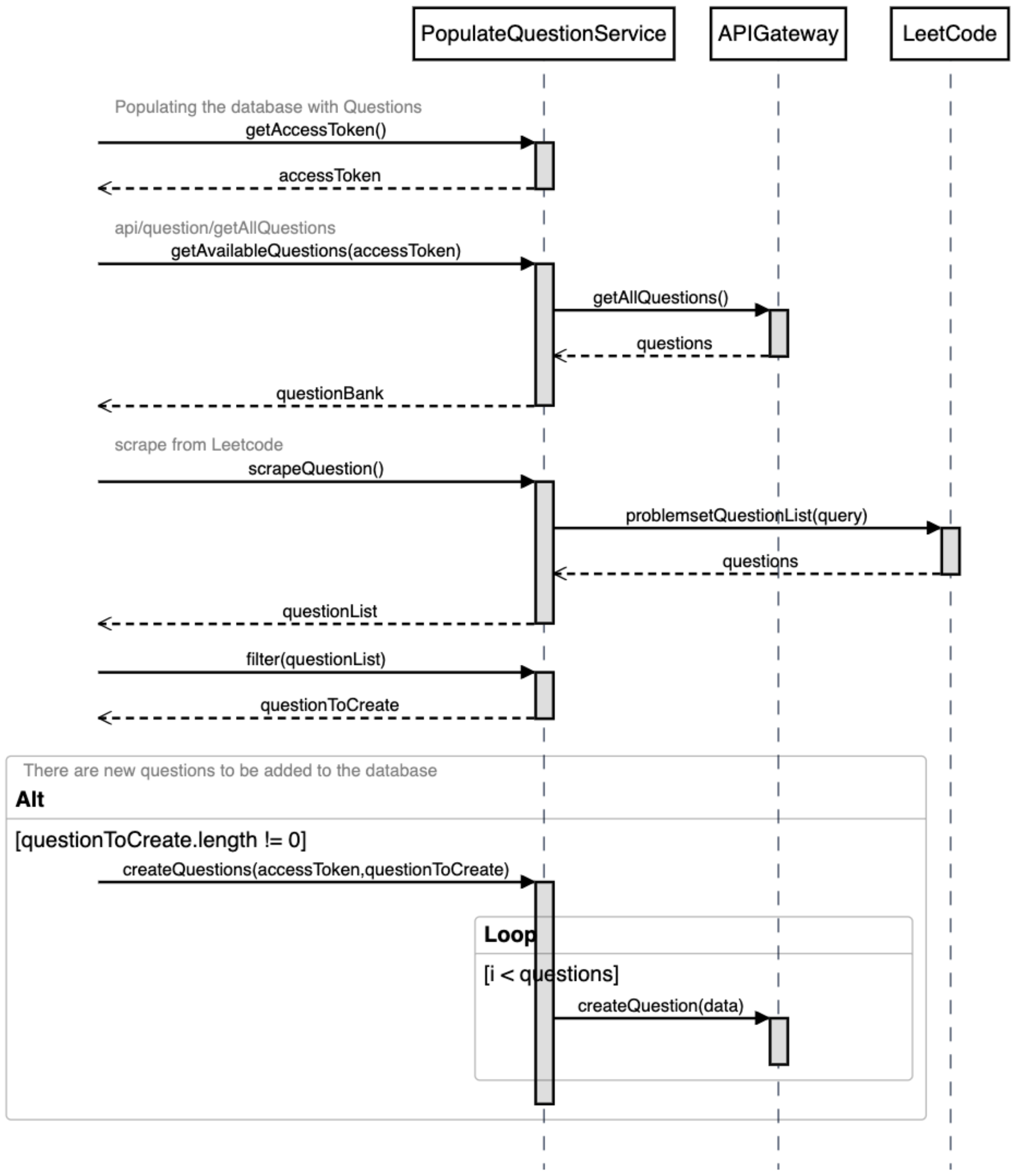
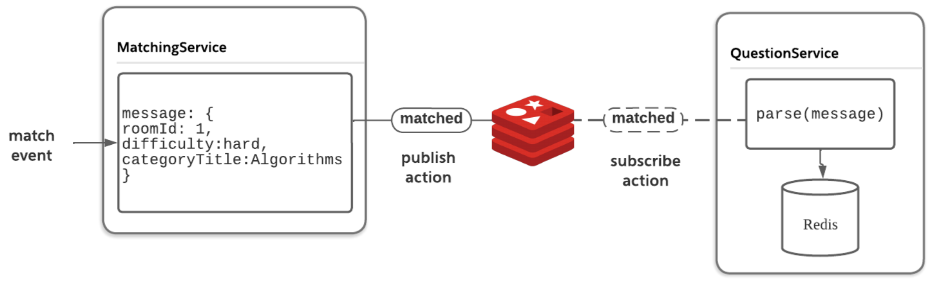

PopulateQuestionService
APIGateway
LeetCode
Populating the database with Questions
getAccessToken()
accessToken
api/question/getAllQuestions
getAvailableQuestions(accessToken)
getAllQuestions()
questions
questionBank
scrape from Leetcode
scrapeQuestion()
problemsetQuestionList(query)
questions
questionList
filter(questionList)
questionToCreate
There are new questions to be added to the database
Alt
[questionToCreate.length != 0]
createQuestions(accessToken,questionToCreate)
Loop
[i < questions]
createQuestion(data)

Figure 9: Sequence Diagram of Question Service Population

4.3.4. Interactions between Matching & Question Service

The use of the observer pattern can be seen from the publisher-subscriber message pattern used in MatchingService and QuestionService.



```
message: {
roomId: 1,
difficulty:hard,
categoryTitle:Algorithms
}
```

Figure 10: Interaction between Matching & Question Service

The *subject* of the pattern now refers to MatchingService. When there is a state change from 'no match' to 'match', MatchingService notifies its *observers*, QuestionService, through Redis's pub-sub function. This can be easily extendable to add more *observers* (subscribers) through a `client.subscribe` action.

The observer pattern follows a push model, where MatchingService (*subject*) pushes a snapshot of its state (parameters) to QuestionService (*observer*).

4.3.5. Interactions between Collaboration, Chat & Matching Service

A room is created whenever:

- a user starts to look for a match.

A room is deleted whenever:

- a user is unable to find a match within 30 seconds or;
- both users exit (disconnect from) the room they are currently in.

To identify each room, a room gets its ID by having the prefix “room_” and digits at the back which comes from the auto-increment ID field of the matching service’s database. An example of a room ID is room_1.

Upon finding a match, the socket of the matching service joins a room with a given unique ID. If the match is successful, the socket will only leave the current room it is in when all clients in the room leave. Otherwise, the socket will leave the room after 30 seconds.

As chat and collaboration services have their own sockets that are separated from the matching service, we have to ensure that their client sockets are connected to the same room.

Therefore, upon matching the clients with the help of the matching service, the sockets of chat and collaboration service will receive data by listening to their `signin` event. This data contains the ID of the room that the client sockets are supposed to be in. ‘socket.join(roomID)’ will then be called to join both sockets to the assigned room.

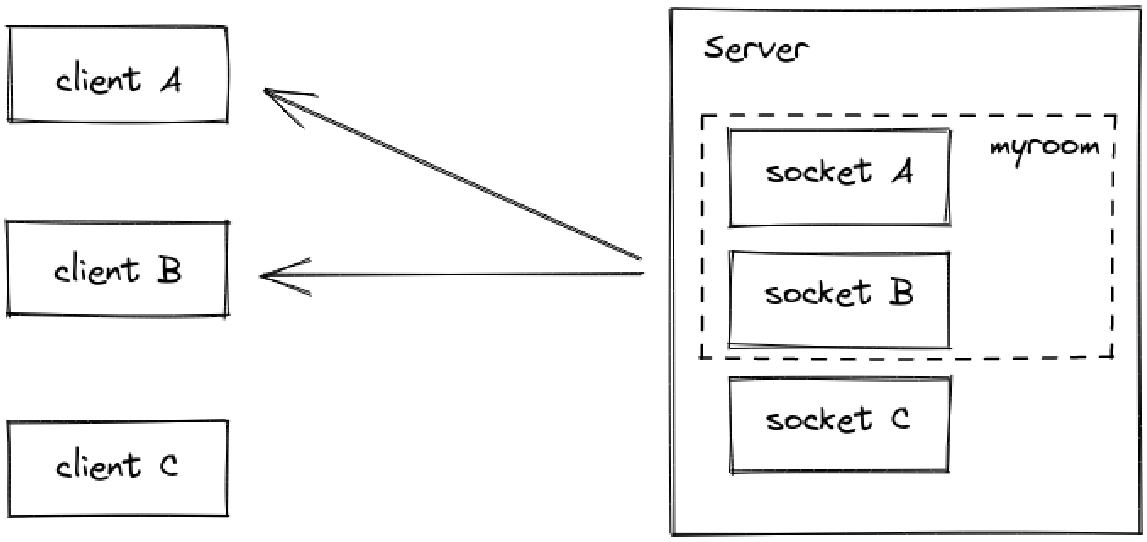


Figure 11: How rooms work in Socket.IO [1]

[1] https://socket.io/docs/v3/rooms/

4.4. Design Diagrams

4.4.1. User Authentication and Authorisation

Our authentication and authorisation are done with the help of Auth0. The following sequence diagram, retrieved from Auth0's documentation, illustrates the process of how the access token of each user is generated.

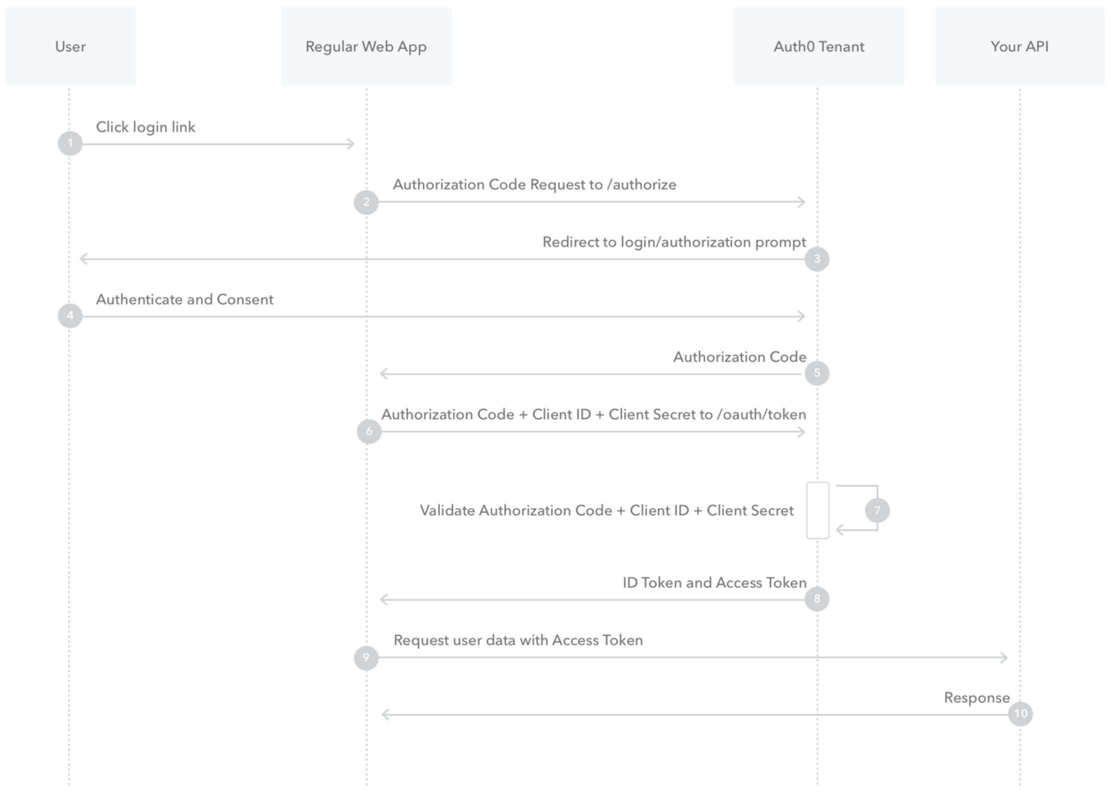

User
Regular Web App
Auth0 Tenant
Your API
1 Click login link
2 Authorization Code Request to /authorize
3 Redirect to login/authorization prompt
4 Authenticate and Consent
5 Authorization Code
6 Authorization Code + Client ID + Client Secret to /oauth/token
7 Validate Authorization Code + Client ID + Client Secret
8 ID Token and Access Token
9 Request user data with Access Token
10 Response

Figure 12: Sequence Diagram of how Authentication/Authorisation work in Auth0 [2]

The access token generated will be used to access the API gateway. Only authenticated users will be able to access our API gateway.

For the rest of the diagrams in this section, we assume that clients are logged in and have a connection to the Matching, Collaboration and Chat sockets.

[2]https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow

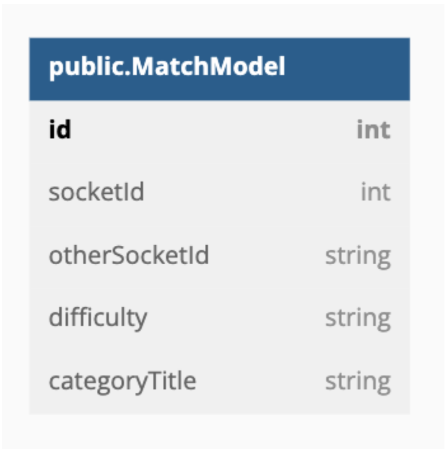
4.4.2. Matching Users by Criteria

On receival of a `match` event, the MatchSocket triggers the MatchOrm::ormFindMatch() function which triggers the MatchRepository::findMatch() function. This function searches the database to find a match based on the difficulty and categoryTitle. If there is a match, it updates the database and returns the match with its otherSocketId field as the current socketId. Otherwise, the otherSocketId field is set to null.

MatchOrm::ormFindMatch() handles the parsing of the match data. If it is a full match, it returns the match as an Object– otherwise, it only returns the matchId.

In MatchSocket, it emits the `matchFound` and the `matchPending` events according to the output mentioned above. If a match is found, the Redis publisher publishes the event `matched` with the question details as its parameter, available for subscription by the Question service (see Section 4.4.1).

The following image shows the schema of the Match model.



public.MatchModel	
id	**int**
socketId	int
otherSocketId	string
difficulty	string
categoryTitle	string

Figure 13: SQLite model for Matching Service

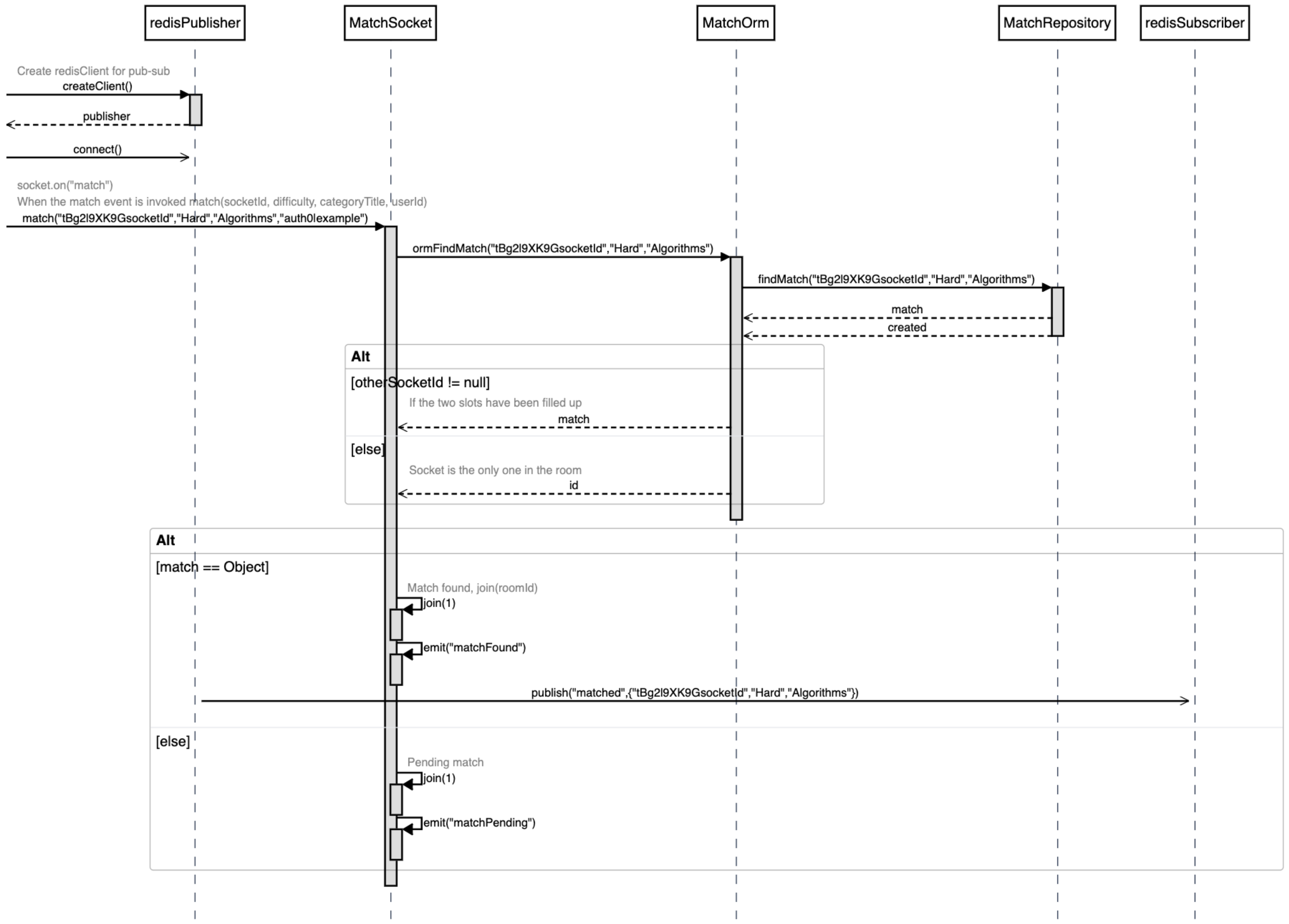

redisPublisher
MatchSocket
MatchOrm
MatchRepository
redisSubscriber
Create redisClient for pub-sub
createClient()
publisher
connect()
socket.on("match")
When the match event is invoked match(socketId, difficulty, categoryTitle, userId)
match("tBg2l9XK9GsocketId","Hard","Algorithms","auth0lexample")
ormFindMatch("tBg2l9XK9GsocketId","Hard","Algorithms")
findMatch("tBg2l9XK9GsocketId","Hard","Algorithms")
match
created
Alt
[otherSocketId != null]
If the two slots have been filled up
match
[else]
Socket is the only one in the room
id
Alt
[match == Object]
Match found, join(roomId)
join(1)
emit("matchFound")
publish("matched",{"tBg2l9XK9GsocketId","Hard","Algorithms"})
[else]
Pending match
join(1)
emit("matchPending")

Figure 14: Sequence Diagram of Matching User by Criteria

4.4.3. Deleting a Match on Timeout and Leave Room

As mentioned in Section 4.3.5, a match is deleted on “timeout” and “leave-room” events. The sequence diagram illustrates the steps the program takes to delete a match.

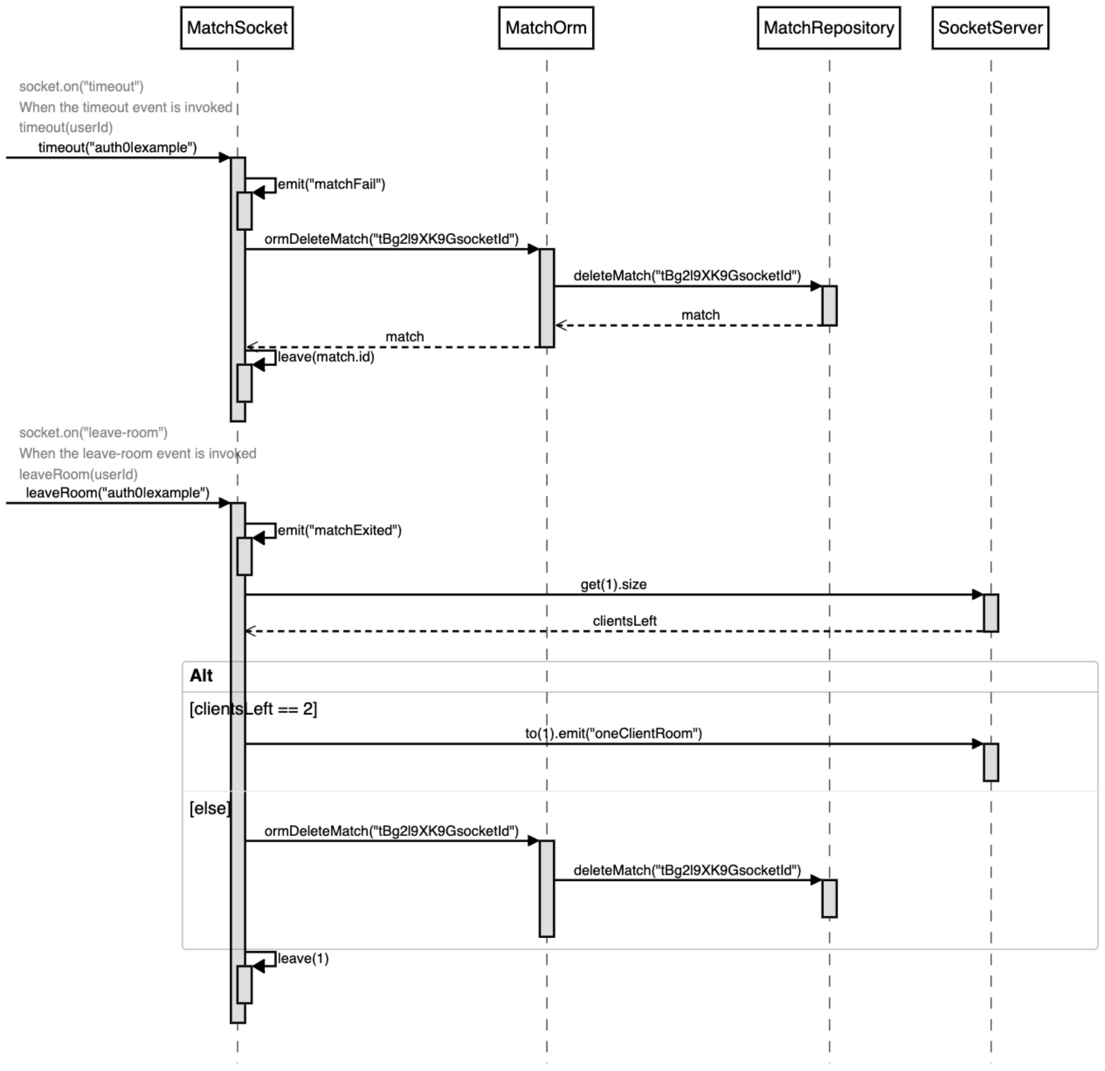
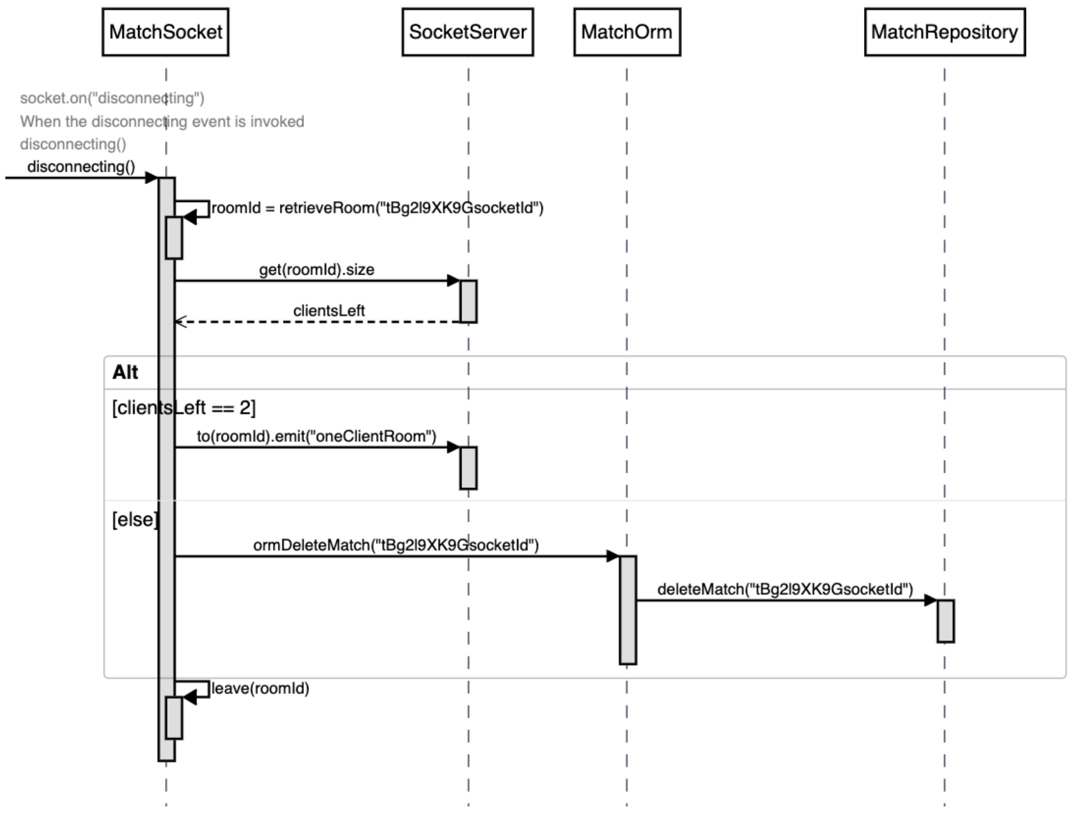


Figure 15: Sequence Diagram of Deleting a Match on Timeout & Leave-Room

4.4.4. Disconnecting from MatchSocket

On the `disconnecting` event of matchSocket, if there is still one user in the room, the `oneClientRoom` is fired and received from the frontend of the other user to inform them that the user has left. If both users have left the room (either through the `disconnect` event or `leave-room` event), the match is deleted.

This event is triggered when the user refreshes the page.



MatchSocket
SocketServer
MatchOrm
MatchRepository
socket.on("disconnecting")
When the disconnecting event is invoked
disconnecting()
disconnecting()
roomId = retrieveRoom("tBg2l9XK9GsocketId")
get(roomId).size
clientsLeft
Alt
[clientsLeft == 2]
to(roomId).emit("oneClientRoom")
[else]
ormDeleteMatch("tBg2l9XK9GsocketId")
deleteMatch("tBg2l9XK9GsocketId")
leave(roomId)

Figure 16: Sequence Diagram of Disconnecting from Match Socket

4.4.5. Pulling data from Question Service

The redis client on the Question Service subscribes to the `matched` event. When a match occurs in the MatchController, the publisher publishes the `match` event, triggering the QuestionController:generateQuestionByDiff() function, which triggers the QuestionOrm::ormGetQuestionByDiff() and then the QuestionRepository::getQuestionByDifficulty() function.

The randomly generated question, based on the difficulty and categoryTitle, is then returned to QuestionController which calls the redisClient::setEx function to set the question in the Redis key-value store with roomId as the key and the question Object as the value.

Redis database schema:

key	value
roomId (string)	question (Object)

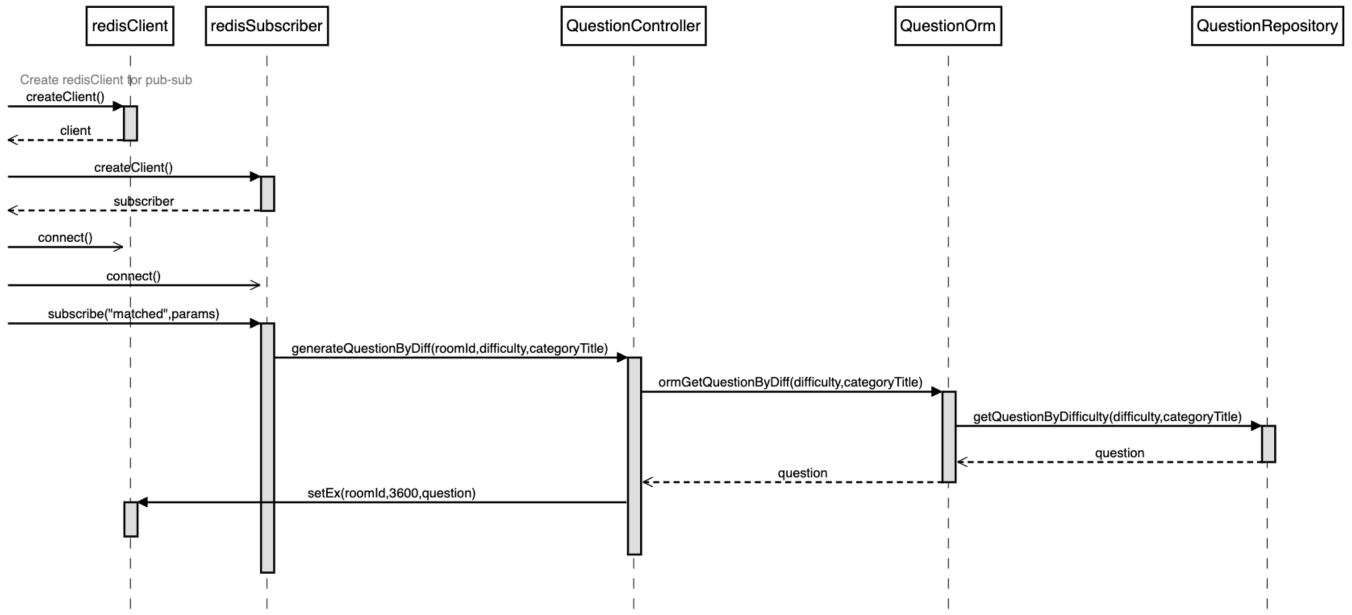

redisClient
redisSubscriber
QuestionController
QuestionOrm
QuestionRepository
Create redisClient for pub-sub
createClient()
client
createClient()
subscriber
connect()
connect()
subscribe("matched",params)
generateQuestionByDiff(roomId,difficulty,categoryTitle)
ormGetQuestionByDiff(difficulty,categoryTitle)
getQuestionByDifficulty(difficulty,categoryTitle)
question
question
setEx(roomId,3600,question)

Figure 17: Sequence Diagram of Pulling Data from Question Service

When the /getQuestionByRoom endpoint is called from the client with a query parameter of room id, QuestionController invokes its get() method to the RedisClient to get the cached question with roomId as its key.

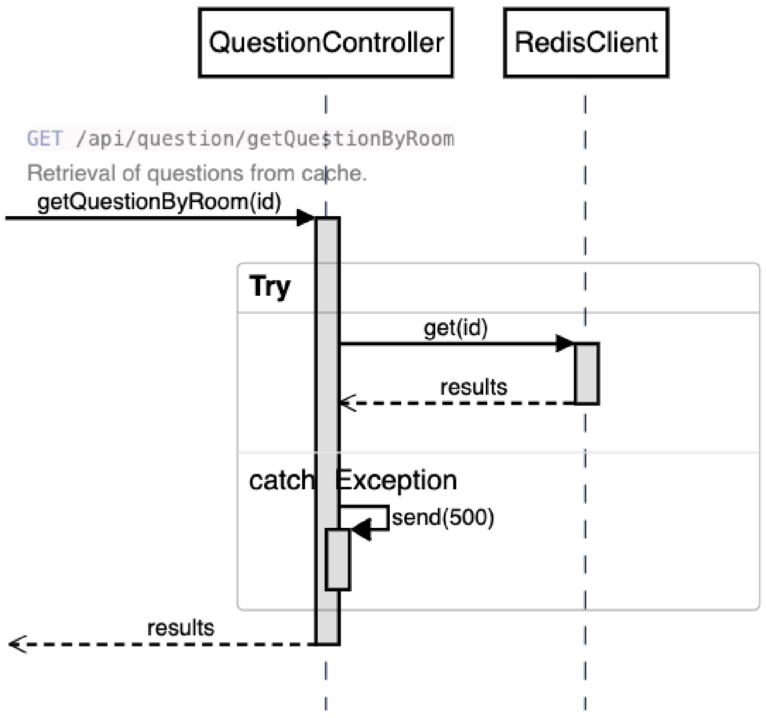


Figure 18: Sequence Diagram of Retrieving Questions from Cache

4.4.6. Real-Time Collaborative Text Editing

Quill[3] is used for our collaborative service.

As Quill requires a container where the editor will be appended, a CSS selector is passed to create a Quill object. Thus, whenever changes like typing or deleting words are made on the editor by the users, events such as `send-changes` and `receive-changes` are called. This is done with the help of React's State Hook which allows us to track the state in a function component.

Upon calling `send-changes` by the client's socket, the changes made on the editor are emitted to the server. These changes are then picked up by the server as it is listening to the `send-changes` event.

Lastly, the data is then emitted to `receive-changes` from the server to the other client in the same room which is also listening to the `receive-changes` event. This data will then be displayed on their editor. Most importantly, the server ensures these changes are only sent to the other client. This is to prevent duplicate changes on the editor.

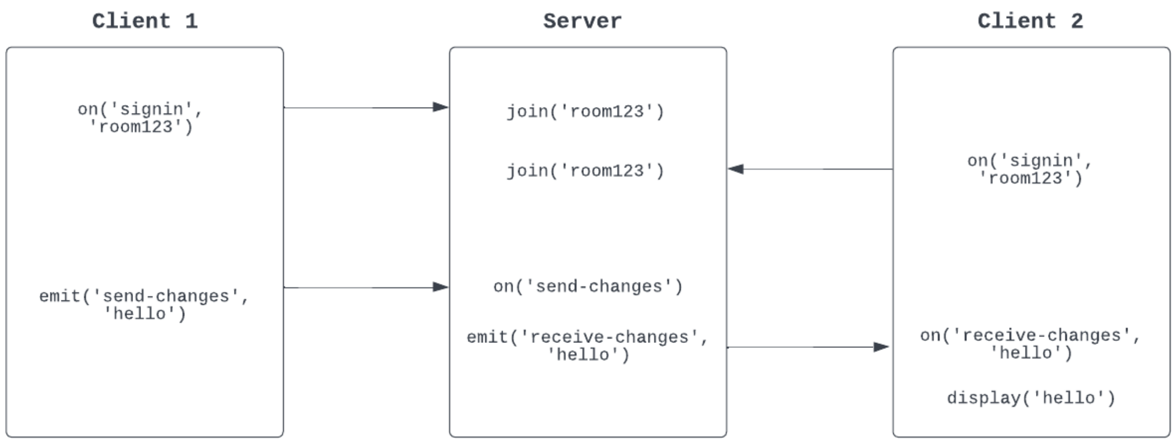


Figure 19: Interaction between Client and Server for Collab Service

[3] https://www.npmjs.com/package/quill

4.4.7. Real-Time Communication through Text Chat

When a user types a message and presses Enter or clicks the Send button, the client's socket sends the message by emitting the `send-msg` event. The message will be saved into the list of messages displayed to client 1 using React's state hook.

The message will be emitted to the server that will be listening to the `send-msg` event. The server will then emit the `got-msg` event to the specific roomId that both clients are in, and will send the message it has received through this event. The roomId will be available as both clients will be logged in and will receive the roomId when they listen to the `signin` event. It is important to note that the server will only emit the `got-msg` event to the second client, and not the client that it received the message from.

The second client will then receive the message as it will be listening to the `got-msg` event. The message will then be saved into the list of messages that will be displayed to client 2 using React's state hook. The chatbox for both clients will therefore be updated in real time.

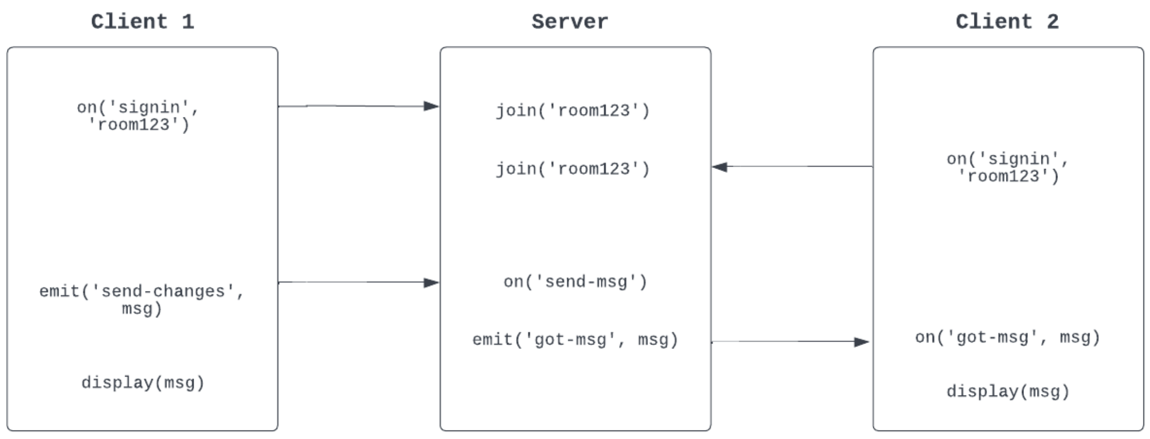


Figure 20: Interaction between Client and Server for Chat Service

5. Suggestions for Improvement and Enhancements

Due to time constraints, our team has decided to limit the number of features we develop and focus on enhancing the basic features of our application. However, given enough time, we have considered the following enhancements that might benefit the users of our application.

5.1. Voice Chat and Video Conferencing

As most technical interviews require both voice chat and video conferencing functionality, we feel that this would be an essential addition to accurately simulate such environments and enhance the experience of both users. A possible example would be to use Socket.IO and WebRTC.

5.2. Countdown Timer

A countdown timer where users can set a fixed timing is essential to enable fairer handovers from one user to another. Without this feature, users may have to agree on a certain amount of time to allocate for each user, dragging on the session unnecessarily.

5.3. Code Compiler

An in-built code compiler would greatly benefit users to check for any syntax errors or check for the correctness of their code. A possible example would be to embed an online code compiler such as Replit or JDoodle.

5.4. Advanced Matching Search by Tags

On top of searching by categories, it might be even more practical to allow search for specific topics like Array, Hash Table, etc. This is so as some users may be weaker in more specific topics.

5.5. Active Session

If a user disconnects from a room due to unpredictable issues, they have the option to join back their current active session to continue their progress. Changes will be made to the matching, chat, and collaboration services. Matching service will have to tie each user to a session in order to search for its active session when needed. The chat service will need to restore chat data. Lastly, the collaboration service will need to adopt a pub-sub model in order to get its latest changes from the other user in the room.

6. Reflections and Learning Points

The experience of developing this project taught our team a lot and they can be categorized into two main learning points.

6.1. Project Management

Initially we were overwhelmed with the number of features we had to implement into the project and were not sure of how to start. We overcame this obstacle by listing out the functional and non-functional requirements that we wanted the project to abide by. After categorizing the requirements according to the respective microservice, we used Github Issues to keep track of the issues that needed to be addressed. This made it easier to get an idea of what had been done and was left.

We had an informal scrum process where we would set internal deadlines to implement certain features before milestones, and would have meetings to discuss any obstacles that we faced in our development. Once we did finish our tasks, we had a manual check in place to ensure that at least one other member reviewed our work before it was integrated into the main branch. In hindsight, we could have enforced our scrum process more strictly to increase our efficiency and is something that we will consider for the future.

6.2. Development and Deployment

Development was challenging as we did not have much experience with web development before this project. Frameworks and tools like Redis, MongoDB, Socket.IO were new to us and we had to find online tutorials to understand how to use them in implementing our features. For instance, using Socket.IO was essential for getting the chat and collaboration microservices to work as they worked on clients interacting with the socket through the frontend.

We also had to keep in mind that deployment is part of the process and had to learn how to deploy our project as we did not have any experience in doing so either. Fortunately we had some practice through our OTOT assignment, and after trying different deployment options, the project was ultimately deployed with the frontend hosted on Amazon S3, using Amazon CloudFront to distribute, and the backend hosted on Amazon EC2. A serverless function is deployed on AWS Lambda as well to automate our question database population.

The main takeaway from the development and deployment process was learning how to find answers to our problems, as we were expected to learn how to use these frameworks and tools on our own. This was a good learning experience to prepare us for the future as we cannot always go to others for help when we run into an obstacle. More often than not, we have to be able to identify the problem and find resources we need to solve the problem independently.

This project exposed us to a lot of different aspects of software engineering that we had not experienced before, from soft skills such as project management and communication, to technical skills like web dev, both frontend and backend. We also had the opportunity to put into practice some of the software engineering theory we learnt in our lectures and tutorials, such as the different principles and patterns. These are invaluable skills that we will definitely use in future software engineering projects.