NUS
National University
of Singapore

CS3219 Software Engineering Principles and Patterns:

Group 14

Full Name	Matric Number
Fan Tianhao	A0211218W
Zhang Yubin	A0211244X
Ng Wen Hao Dennis	A0218337A
Sharmaine Cheong Shi Min	A0202219U

Milestone 3 Report

Contents

Goals and Objectives

Goals

Tech interviews can be a daunting task, especially for students either seeking for internships or looking for their first full time job. There are avenues such as leetcodes and kattis that prepare students for technical interviews by solving technical questions. However, some of these questions can be daunting to the students, especially when they are solving it alone. Students wished that they could collaborate with others to solve these questions.

Objective

The objective for code takes two is to allow students to be better prepared for technical interviews. Students will collaborate with other students such that they will not have to think of a solution alone, thus making them solve questions easily.

Team

Name	Responsibilities	Non-technical contributions
Sharmaine Cheong Shi Min	Frontend(UI)	
Dennis	Frontend(Logic such as Socket.io and integration with the backend) Fix bugs	Liaise with the backend team on the implementation and communication between the frontend and the backend components
Fan Tianhao	Backend (REST services, reverse proxy, deployment)	hold weekly meetings
Zhang Yubin	Backend(REST services, socket io)	

Product Requirement

Functional Requirements

User service

S/N	Functional Requirements	Priority
Fr 1.1	The system should allow users to log in with username and password.	High
Fr 1.2	The system should allow users to create a new account.	High
Fr 1.3	The system should present a difficulty page for the users to select difficulty when logged in.	High

Fr 1.4	The system should be able to logout of their account. - After logout, the user needs to key in the credentials correctly again to enter the app.	High
Fr 1.5	The system should allow accounts to be deleted.	High
Fr 1.6	The system should allow users to change password.	High
Fr 1.7	The system should redirect a user to the difficulty page if the user has successfully logged in within 30 days.	Medium

Matching service

S/N	Functional Requirements	Priority
Fr 2.1	The system should allow users to select a difficulty: easy, medium, and hard.	High
Fr 2.2	The users should be on the loading page for 30 seconds to wait for another player. - In the meantime, the system should allow a user to quit waiting.	High
Fr 2.3	Once two users are successfully matched, the system should redirect them to a question page.	High
Fr 2.4	If the user is not able to find a match after 30 sec, redirect them back to the difficulty page	High
Fr 2.5	If a user refreshes the matching page, the timer will restart.	High
Fr 2.6	The system should end a match if both users leave the room/close the tab.	High

Question Service

S/N	Functional Requirements	Priority
Fr 3.1	The system should store questions indexed by difficulty levels: easy, medium, and hard.	High
Fr 3.2	The system should allow a match session to retrieve a list of questions of the same difficulty.	High
Fr 3.3	The system should provide questions indexed by question id.	High
Fr 3.4	The system should store the latest answer submitted.	High
Fr 3.6	The system should guarantee that there is no duplicate question within each match session.	Medium

Chat Service

S/N	Functional Requirements	Priority

Fr 4.1	The chat service should be able to receive the message sent by the other user in the same match	High
Fr 4.2	The chat service should be able to send its message to the other user in the same match.	High
Fr 4.3	The chat service should be able to show the user the messages from the other user, along with who sent the message and the timestamp.	High
Fr 4.4	The chat service should save the messages the user just sent, along with the timestamp.	High
Fr 4.5	The chat service should be able to save the chat history until the match ends. - When the user disconnects and connects back to the match, the user will be able to retrieve the chat history.	High

Justification of Chat Service:

- A live chat box enables users in a room to communicate and interact with each other that improves their experience of collaboration.

Collaboration Service

S/N	Functional Requirements	Priority
Fr 5.1	The system should support a real-time file sharing feature allowing both users to see each other's modification	High

Non-Functional Requirement

Through the progress of our projects, we have to address various non functional requests (NFR) when we are developing our project. However, the following are the top 3 NFR that we decided that it is very important for our project:

- Responsiveness
- Scalability
- Security

Responsiveness is very important for our project, especially if it involves real time interaction between 2 users and the backend service. Firstly, the users must have a smooth interaction from finding matches between 2 users, to having the same questions for the 2 users in the match, little latency on the collaboration service, and finally, chat messages handled by the chat service needs to be sent to the other party without disrupting the chat flow due to internet latency.

The second NFR is Scalability. Being scalable means that the backend services are able to scale according to the incoming connection and usage from the users. As users use our project very often, the initial server setup has to use up more computational power to handle the increasing user connection which will potentially cause the server to slow down and eventually crash. Hence, it is very important that our setup should adjust its own

computational power or even the number of nodes to adapt to the number of connections from users.

And finally, security. For most software developers security is often overlooked as they focus more on the functionality. This however may be disastrous when attacks happen such as data can be stolen or even disruption of service (DOS) attacks which may render our product unworkable for users. In our project for example, we use an SQL database which is susceptible to SQL injection attacks. Therefore, we should ensure that the user inputs are sanitised before querying the database to prevent any devastating effects executing on our database. Furthermore, as we are handling user authentication, we need to ensure that sensitive information such as passwords are stored securely such that if the hacker manages to attack the database, they should not be able to get the password from the database.

Responsiveness:

S/N	Non-Functional Requirements	Priority
Nfr 1.1	Users should not wait for more than 10 seconds to login	High
Nfr 1.2	When two users are within a match and one of the users updates the code section (i.e. collaborating textarea), the change should take less than 2 seconds to be displayed on the other side.	High
Nfr 1.3	When one of the users in the match goes to the next question, the other user needs to receive the next question automatically without delay.	High
Nfr 1.4	The chat function should not break the flow of the conversation between users due to network latency	High

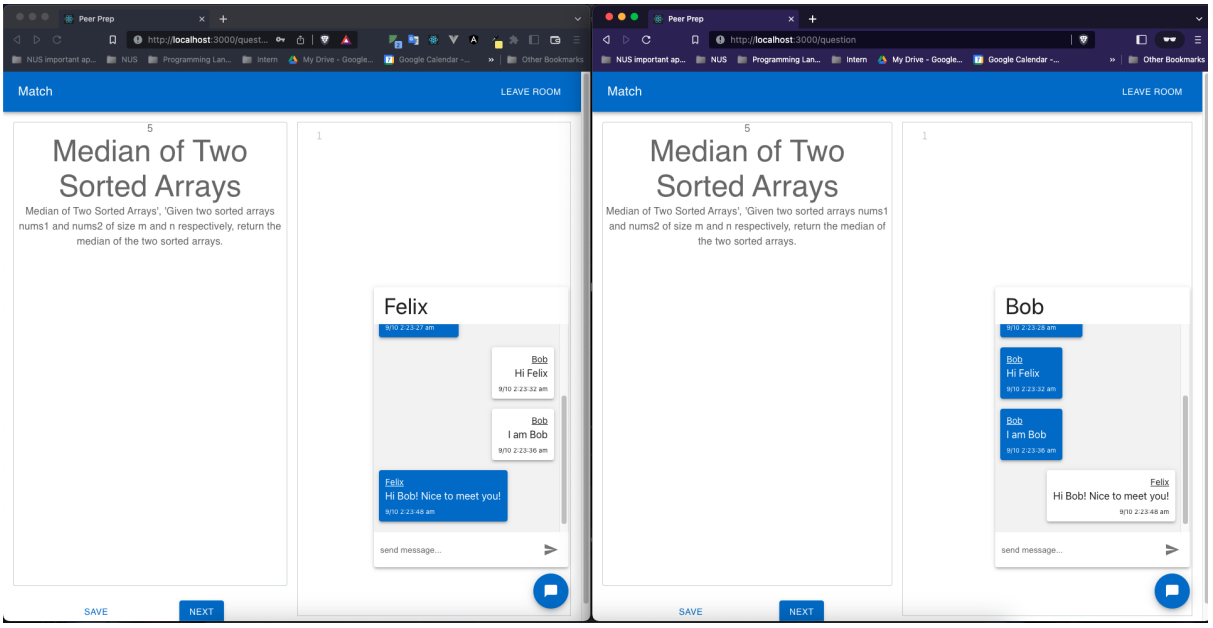


Figure 1: Nfr 1.4 (The chat function should not break the flow of the conversation between users due to network latency)

Scalability:

S/N	Functional Requirements	Priority
Nfr 2.1	The system should scale to match the number of incoming requests	High

```
[ec2-user@ip-172-31-40-148 codetakestwo-k8s]$ kubectl get hpa
NAME                        REFERENCE                              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
auth-service                Deployment/auth-service                0%/80%    1         3         1          35m
frontend                    Deployment/frontend                    0%/80%    1         3         1          35m
matching-service            Deployment/matching-service            0%/80%    1         3         1          109m
question-service            Deployment/question-service            1%/80%    1         3         1          35m
reverse-proxy               Deployment/reverse-proxy               0%/80%    1         3         1          35m
socketio-chat-service       Deployment/socketio-chat-service       0%/80%    1         3         1          35m
socketio-collab-service     Deployment/socketio-collab-service     0%/80%    1         3         1          36m
socketio-matching-service   Deployment/socketio-matching-service   0%/80%    1         3         1          36m
user-service                Deployment/user-service                0%/80%    1         3         1          28m
```

Figure 2: Nfr 2.1 (Configured HPAs)

```
[ec2-user@ip-172-31-40-148 codetakestwo-k8s]$ kubectl get hpa user-service -w
NAME           REFERENCE                 TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
user-service   Deployment/user-service   <unknown>/80%   1         3         0          8s
user-service   Deployment/user-service   0%/80%          1         3         1          15s
user-service   Deployment/user-service   0%/80%          1         3         1          45s
user-service   Deployment/user-service   34%/80%         1         3         1          60s
user-service   Deployment/user-service   100%/80%        1         3         1          75s
user-service   Deployment/user-service   100%/80%        1         3         2          90s
user-service   Deployment/user-service   100%/80%        1         3         2          105s
user-service   Deployment/user-service   87%/80%         1         3         2          2m
user-service   Deployment/user-service   84%/80%         1         3         2          2m15s
user-service   Deployment/user-service   81%/80%         1         3         2          2m30s
user-service   Deployment/user-service   95%/80%         1         3         2          2m45s
user-service   Deployment/user-service   86%/80%         1         3         3          3m
user-service   Deployment/user-service   66%/80%         1         3         3          3m15s
user-service   Deployment/user-service   91%/80%         1         3         3          3m30s
user-service   Deployment/user-service   81%/80%         1         3         3          3m45s
user-service   Deployment/user-service   74%/80%         1         3         3          4m
user-service   Deployment/user-service   70%/80%         1         3         3          4m15s
user-service   Deployment/user-service   88%/80%         1         3         3          4m30s
user-service   Deployment/user-service   91%/80%         1         3         3          4m45s
user-service   Deployment/user-service   74%/80%         1         3         3          5m
user-service   Deployment/user-service   73%/80%         1         3         3          5m15s
user-service   Deployment/user-service   80%/80%         1         3         3          5m30s
user-service   Deployment/user-service   66%/80%         1         3         3          5m45s
user-service   Deployment/user-service   83%/80%         1         3         3          6m
user-service   Deployment/user-service   79%/80%         1         3         3          6m15s
user-service   Deployment/user-service   79%/80%         1         3         3          6m30s
user-service   Deployment/user-service   82%/80%         1         3         3          6m45s
user-service   Deployment/user-service   87%/80%         1         3         3          7m
user-service   Deployment/user-service   51%/80%         1         3         3          7m15s
user-service   Deployment/user-service   0%/80%          1         3         3          7m30s
user-service   Deployment/user-service   0%/80%          1         3         3          9m45s
user-service   Deployment/user-service   0%/80%          1         3         3          12m
user-service   Deployment/user-service   0%/80%          1         3         2          12m
user-service   Deployment/user-service   0%/80%          1         3         1          12m
```

Figure 3: Nfr 2.1 (HPA Scaling during User Service load test.
For details, see Implementations - Infrastructure)

Security:

S/N	Functional Requirements	Priority
Nfr 3.1	Stored Password should be hashed	High
Nfr 3.2	System should be protected from SQL Injection	High
Nfr 3.3	JWT Cookies for each user login in should not be reused	High
Nfr 3.4	Password must follow a format with a mixed of small and capital letters, numbers and special symbol	High

	username [PK] text	password text
1	Bob	$2a$10$WluXDB/sWvvW5AmseUwXKebuaOCHvS3gEERviX9jekX2A9ibQi8nm
2	Felix	$2a$10$2B31v9TDfs3wH85tTHJWUeYqLaRe4V8widBuqTFAcD0a88U1epFyi

Figure 4: Nfr 3.1 (Stored Password hashed in the database)

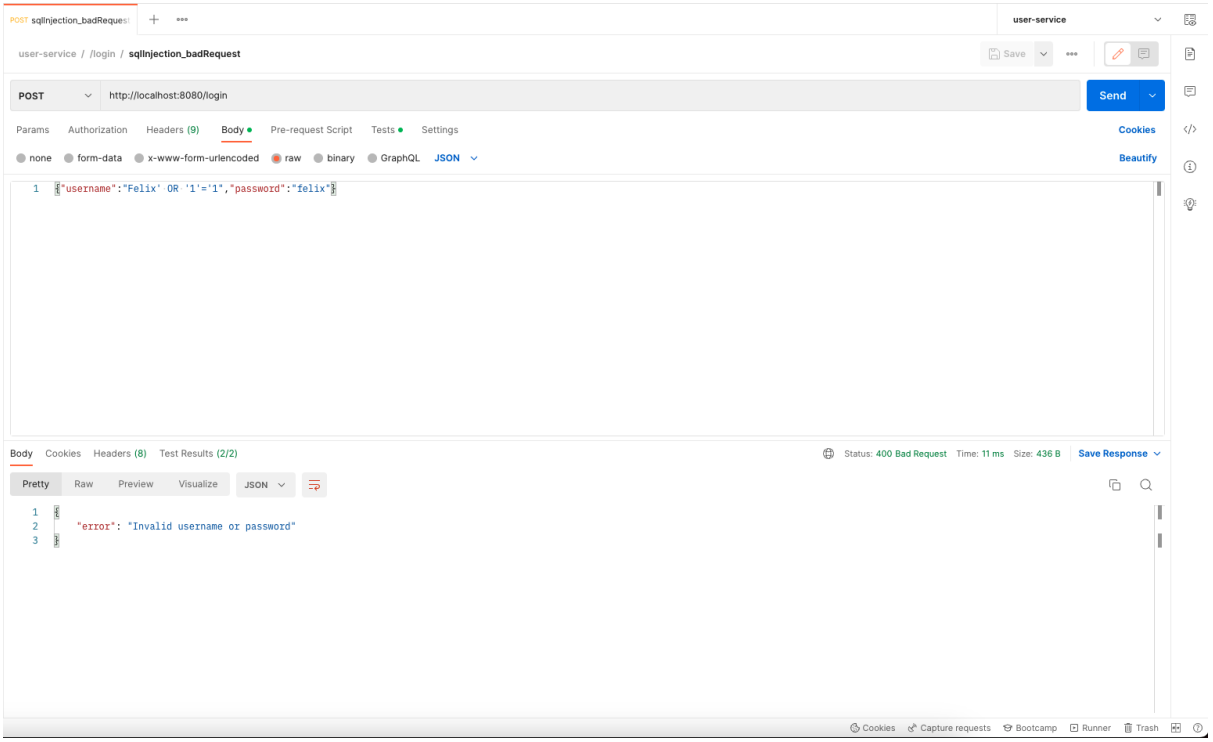


Figure 5: Nfr 3.2 (System should be protected from SQL injection)

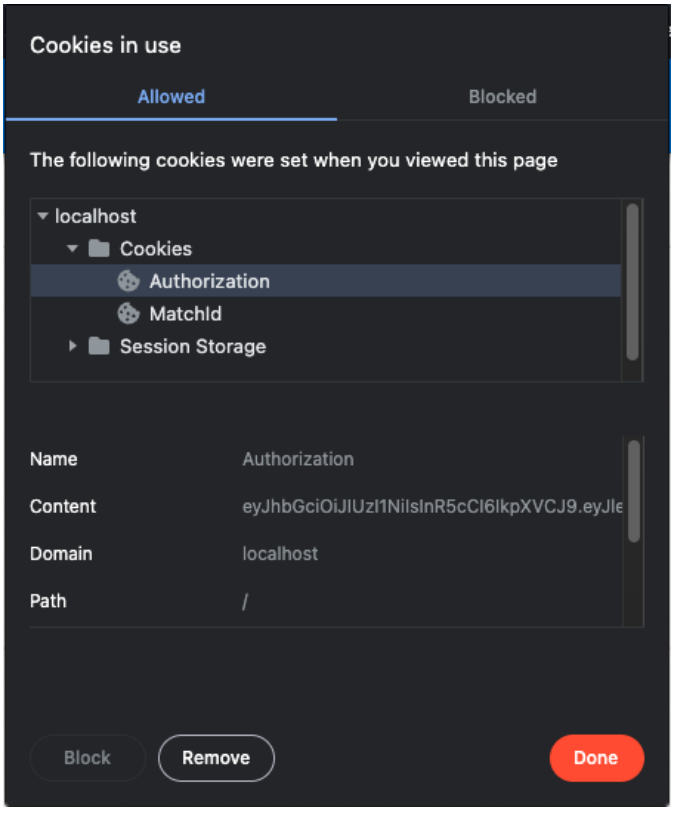


Figure 6: Nfr 3.3 (JWT Cookie used for authentication)

Tech Stack & Justification

Frontend:
ReactJS
- Flexible and easy to maintain
- Reusability
- High performance
- Supports various frontend libraries such as Material UI
- Large user base as compared to other frontend frameworks such as vue.js
- Dependencies used includes:
    - prismjs: syntax highlighting library used in Question page
    - react-countdown-circle-timer: countdown timer component in a circle shape with color and progress animation used in Matching page
    - react-simple-code-editor: code editor used in Question page

Backend:
Golang:
- The code is more readable with less layers of abstractions compared to Object Oriented languages and frameworks, e.g. Java & Spring
- Built-in concurrency support
- Size of executables are small; less overhead compared to JVM
- gin framework:
    - A good encapsulation for the server side functions in http package
    - Simple API for routing and middlewares
- gorm framework:
    - An Object Relational Mapping (ORM) framework in golang
    - Reduce the need for repetitive raw SQL code
    - Provide a robust set of features

Express JS & Socket IO:
- Built in heartbeat mechanism
- Allow server to push messages to client in real time

PostgreSQL:
- A standard SQL database

Redis:
- High-performance in-memory key-value pair storage
- Flexible data structures
- Easy to understand commands

Kafka:
- Distributed message queue; decouple writing from reading
- Easy to scale
- Append-only log: more data does not affect performance

Nginx:
- Low-latency reverse proxy
- Built-in modules for handling authorization and manipulating requests and responses
- Support WebSocket

Packaging Tool:

Docker:

- Provide isolated environment for every microservice
- Platform independent: developers on Windows and Mac OS share the same docker config
- Used along with Dockerhub as image repository

Deployment Tool:

Kubernetes:

- Popular orchestration tool
- Automated healing (restarting dead pods) and scaling
- Easy to migrate from local environment to cloud

Minikube:

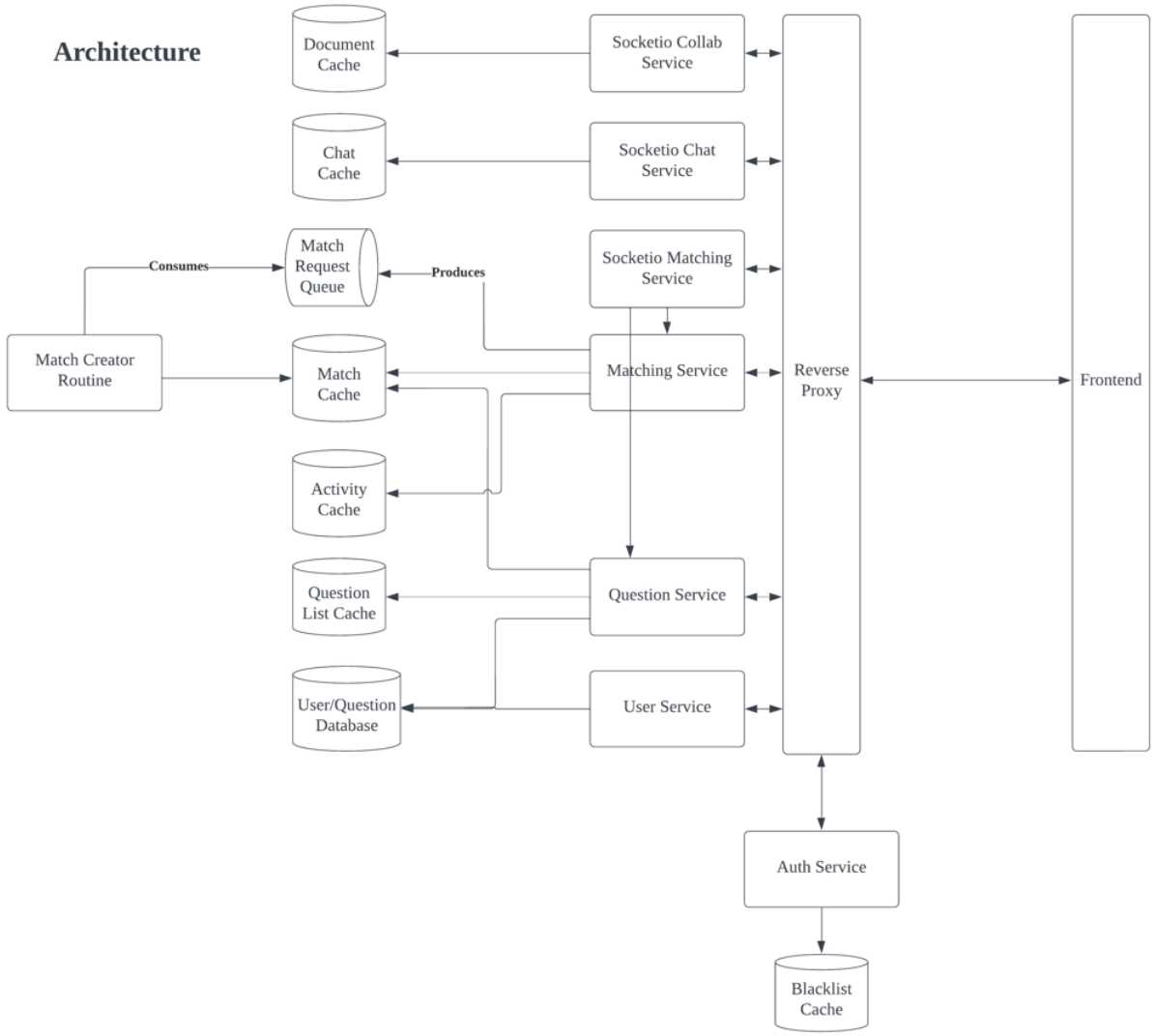
- A local k8s for testing k8s yaml files
- Easy to setup
- Used along with VirtualBox, which provides VM support

kops:

- Automated setup and manage Kubernetes clusters on AWS
- Easy to use

Implementations

Architecture



The main architecture paradigm of our product is microservice architecture. From the diagram above, each entity, drawn in rectangles or cylinders, represents a microservice that runs inside a container. The container can be within either docker environment or k8s environment. With service discovery in either environment, each microservice is assigned a domain name and a virtual IP address, which enable the microservices to communicate with each other via HTTP calls.

There are three main categories among all microservices: **REST services**, **Socketio services, and Persistence**.

REST services:

- Provide a set of APIs on the corresponding domain entities
- Depend on Persistence components
- Implemented in Golang

- Matching Service, Question Service, User Service, and Auth Service fall under this category.

Socketio services:

- Handle events that are transmitted between the server and client using Socket.io
- Depend on both REST services and Persistence components
- Socketio Matching Service, Socketio Chat Service, and Socketio Collab Service fall under this category.

Persistence:

- Store domain objects in memory or disk.
- Can be either a database or a cache
- Document Cache, Chat Cache, Match Cache, Question List Cache, Blacklist Cache, and User/Question Database fall under this category.

The microservices within each category are similar to each other regarding their functionalities, code structures, and dependencies.

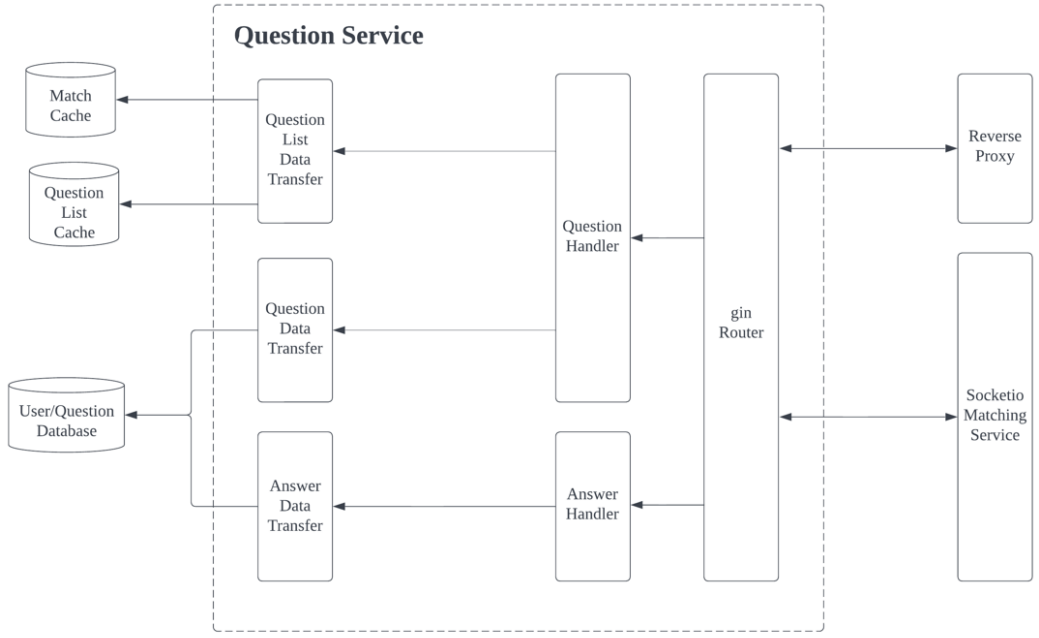
Other components do not belong to any of the three categories.

- Frontend manages the UI.
- Reverse Proxy delegates frontend to the corresponding microservice.
- Match Request Queue is a kafka message queue for match requests.
- Match Creator Routine pulls from Match Request Queue and create match objects

Note that the word "**service**" has a different meaning in this context. A service here refers to **a running process,** (usually) in a container, that performs a concrete task such as serving HTTP requests or handling WebSocket events. (In the "Requirement" section, the word service has a different meaning, which is "a collection of functionalities that serves an end user".)

Microservices

Question Service Diagram



Functionality - Question Service is a rest service that governs question entities. Its main duty is to generate a list of questions based on difficulty at the start of a match. From there, the question service will also record the latest question, and provide the next question in the list when a user asks for one.

Question Service is capable to handle concurrent requests on question objects. For example, when two users ask for the next question at the same time, only one will succeed. This is achieved with a simple lock mechanism using Redis.

It also manages answer objects. However, we did not implement "history service" due to time constraints. Therefore, users can save their latest answer, but they can never retrieve & view it.

Interaction - As a server, Question Service handles requests that come from Reverse Proxy and Socketio Matching Service. On the persistence side, it **reads from** Match Cache, and reads from and writes to Question List Cache and User/Question Database.

Internal Structure - Internally, it conforms to a **three-layer structure**.

The first layer is the gin Router, which is part of the external library. It will parse the HTTP request, extract the URL parameters, headers and request body, then save them into a struct (gin.Context) and pass it to the handlers. The gin Router is also in charge of directing the requests and sending responses back. For more information, see gin's official website.

The second layer is the handler layer. A handler's job is to write HTTP responses. If handling the request requires external data, it will delegate the job to the data transfer(DT) layer to interact with external persistent components. From the parsed HTTP request as well as data from DT layer, handlers will go through a series of logic and eventually writes the response. Then, handlers will pass the response to gin Router.

The third layer is the DT layer. As mentioned before, it interacts with external components and performs data operations. From the basic operations, it could also build helper functions for handlers to use. Despite the helper functions, DT layer functions are meant to be simple and "functional". It usually contains little or no "business logic".

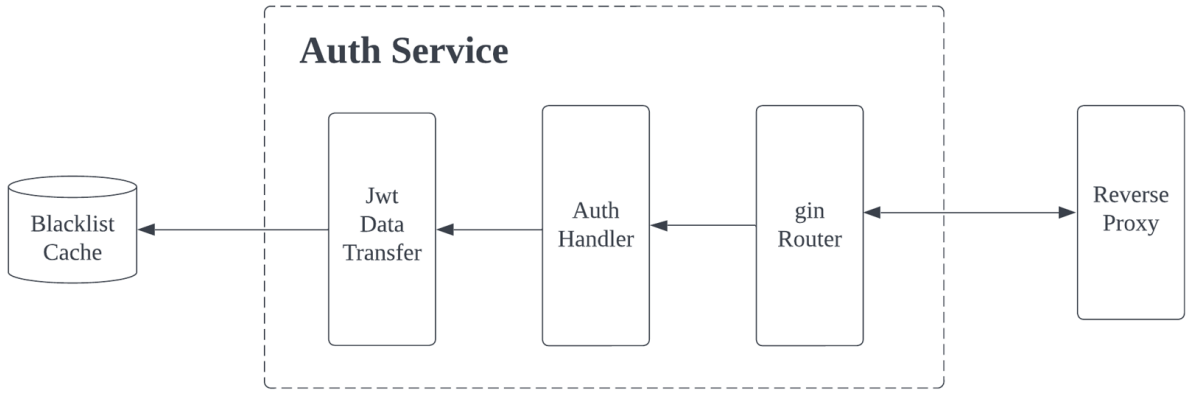
There is also a hidden component, "models". A model is a struct in go, and it defines a domain object. If it is stored in a database, it will also serve as a blueprint of a table for gorm, the ORM library. That is, each struct corresponds to a table in the database.

There are two main benefits of the three-layer structure. The first one is that there are fewer layers of abstractions, making it easy to write new code or understand existing code. Secondly, the business logic is decoupled from the data operations. This means that if we change the data component, we can simply refactor the implementation of the DT layer functions without affecting the other two layers.

The three-layer structure also has some issues. For a complex operation, the business logic is packed into a single function, and the handler function could be quite lengthy. This can make the code harder to maintain.

Other rest services also conform to the three-layer structure.

Auth Service Diagram

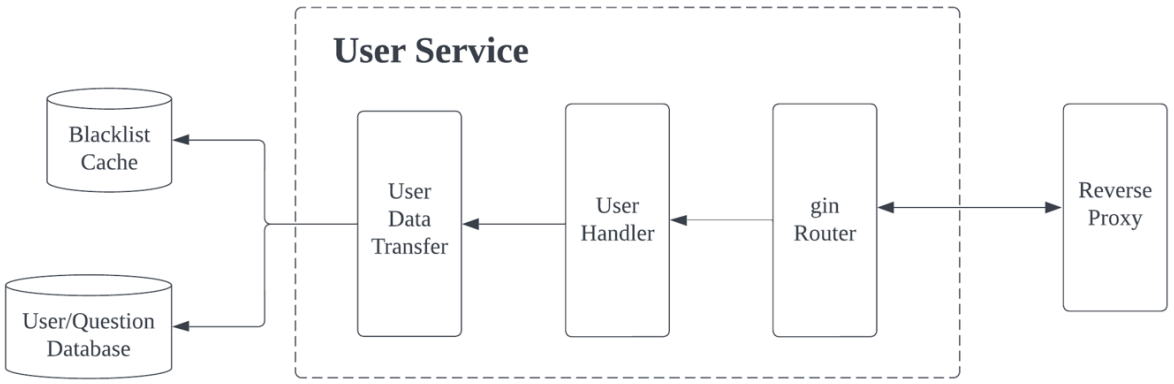


Functionality - Auth Service handles **authorization tasks**. Specifically, it will validate authorization cookies and blocks the requests with outdated or blacklisted cookies. This is crucial for us because most of the endpoints can only be accessed with successful authorization (see Notes section in our README).

For valid cookies, it will also extract the content, save the content as a response header, and pass back Reverse Proxy. Reverse Proxy will then copy the header to the requests to downstream microservices, who can use the cookie values directly.

Interaction **-** As a server, Auth Service accepts requests from Reverse Proxy. On the persistence side, it will **read from** the blacklist cache.

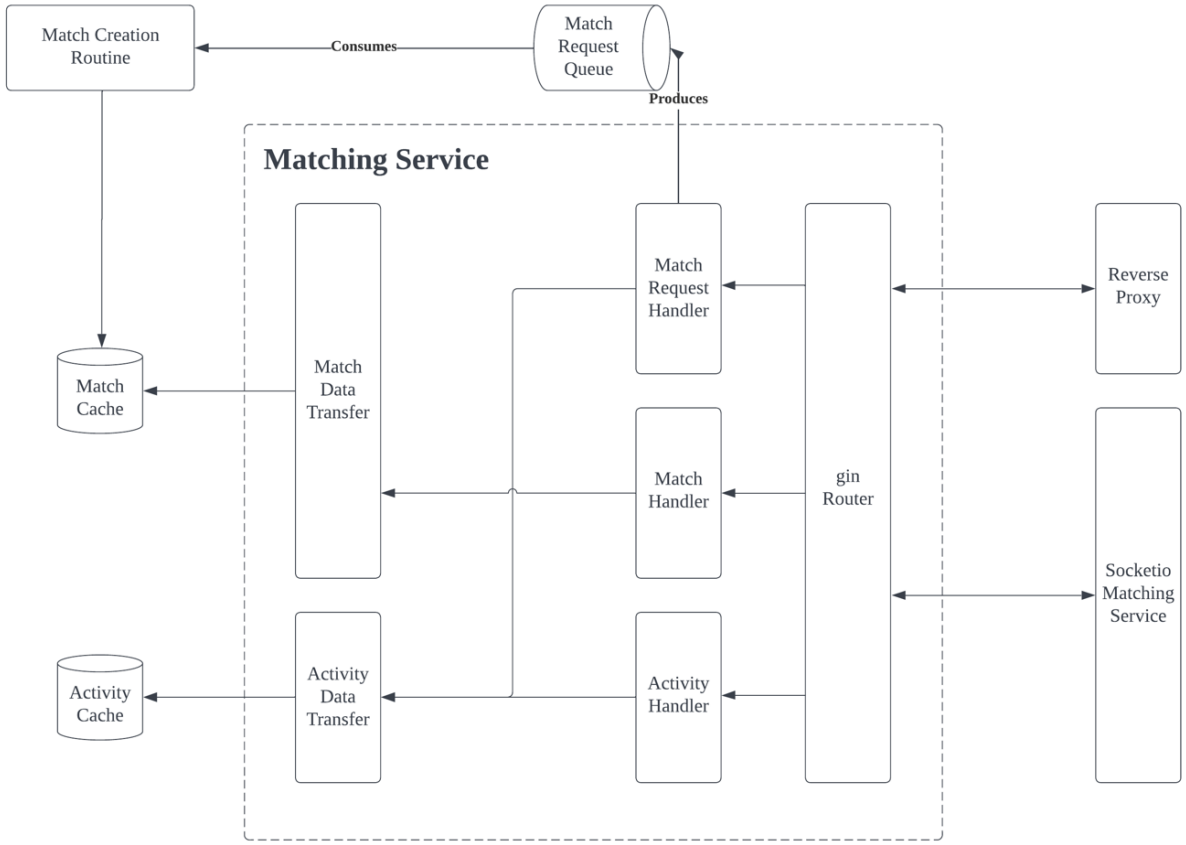
User Service Diagram



Functionality - Each service of our app executes in the security context of a user account. User Service is a rest service that manages user accounts. It allows users to create an account with username and password, to log in to their account, to change password, and to delete accounts. User service performs **authentication** when a user provides the correct credentials.

Interaction **-** As a server, User Service accepts requests from Reverse Proxy. On the persistence side, it will **read from** and **write to** User/Question Database.

Matching Service Diagram



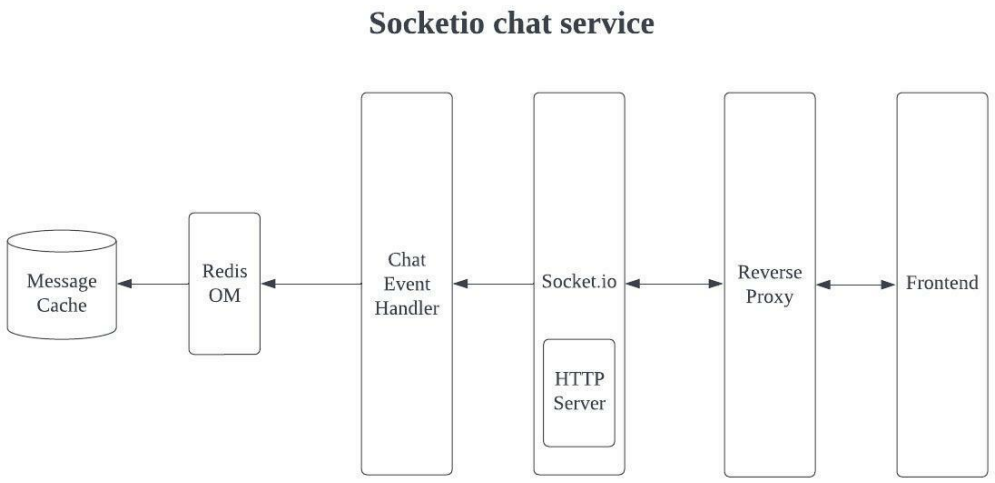
Functionality - Matching Service is a rest service that governs match and activity entities. It provides endpoints for retrieving a match/activity object, as well as updating their status.

It also serves as a producer for Match Creator Queue. Specifically, it creates the required topics (“easy”, “medium”, and “”hard) if not exist. When a user sends a request match, Matching Service will serialize the match request into a json object, and writes it to the match request queue.

Interaction - As a server, Matching Service handles requests that come from Reverse Proxy and Socketio Matching Service. On the persistence side, it reads from and writes to Match Cache and Activity Cache. It also writes to Match Request Queue.

Note - Inside the code, the activity model is called “User”, because it encapsulates the “short-lived” attributes of a user as compared to username and password stored in user service. Here we rename “User” to “Activity” to avoid confusion.

Socketio chat service Diagram

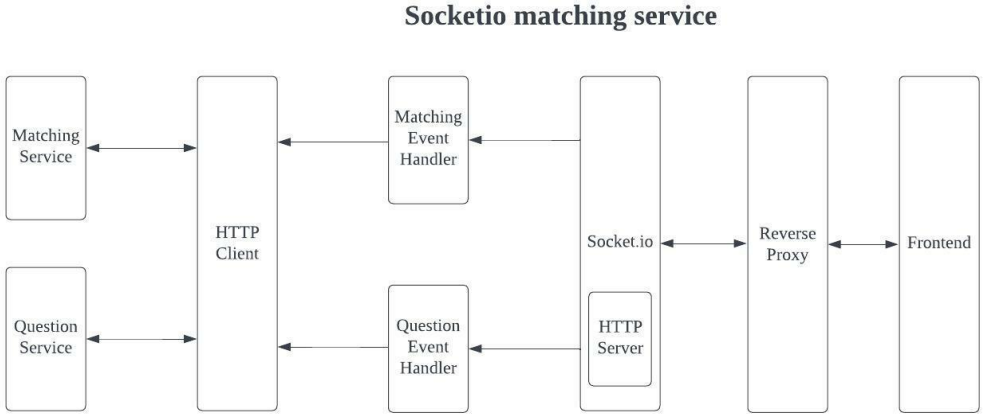


Functionality - Chat service is a socket.io service that implements real-time communication between users in a room. Its main duty is to listen to an event that carries a message from a user and send the message to the other users who are in the same room as the sender. Chat service will also record chat history in a room into a cache and send the chat history to a user when the user reconnects to Chat service.

Interaction - As a server. Chat service handles the events that come from Reverse Proxy. On the persistence side, it reads from and writes to Message Cache.

Internal Structure - Internally, its implementation mainly depends on in-built functions of socket.io (i.e. event listener and emitter). It also uses Redis OM to achieve mapping between Redis data type and JavaScript objects and data retrieval with the help of RedisSearch. More details can be found here.

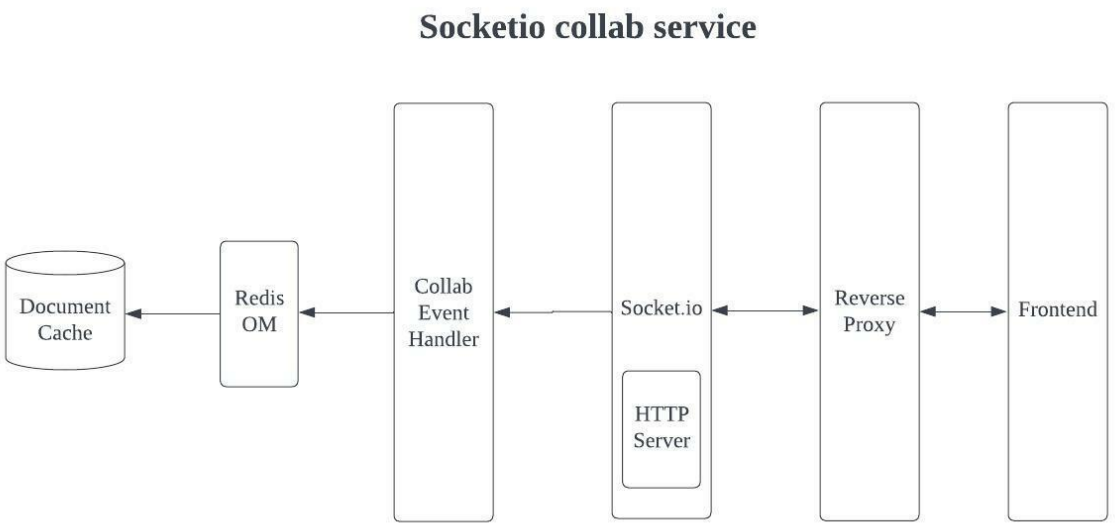
Socketio matching service Diagram



Functionality - Socket.io matching service is a socket.io service that lies between Matching service and frontend. Its main duties include using a long-polling mechanism to retrieve user's matching status and change user's status based on the request from frontend.

Interaction - As a server, Socketio Matching Service handles events that come from Reverse Proxy and Matching Service and Question Service.

Socketio collab service Diagram



Functionality - Collab service is a socket.io service that allows users to edit a document together in real-time with easy sharing. Its main duty is to listen to an event that carries a change on the shared document from a user and send the change to the other users who are in the same room as the sender. Collab service will also temporarily save the document being and send the document to a user when the user reconnects to Collab service.

Internal Structure - Internally, it uses Quill, a rich text editor, to record user's change on a document by encapsulating text and operation. More details can be found here.

Frontend

Functionality: The frontend component is a user facing component where the data and state of the various services are reflected to the user. The frontend is compiled using react.js of which it is run using npm (for development mode) or using npm serve dependency (for production mode).

Interactions: Interactions with the backend services are mainly handled via the axios dependency. However, services such as the matching service, chat service and the collaboration service are handled by the socket.io client service.

```
const verifyLogin = async () => {
    const res = await axios.post(URL_USER_SVC_VALIDATE,  data: {},  config: {withCredentials: true})
        .catch((err) => {
            if (err.response.status === STATUS_CODE_NOT_LOGGED_IN) {
                setIsLoggedIn( value: false);
            }
        });
    if (res && res.data.username) {
        setUsername(res.data.username);
        setIsLoggedIn( value: true);
    }
}
```

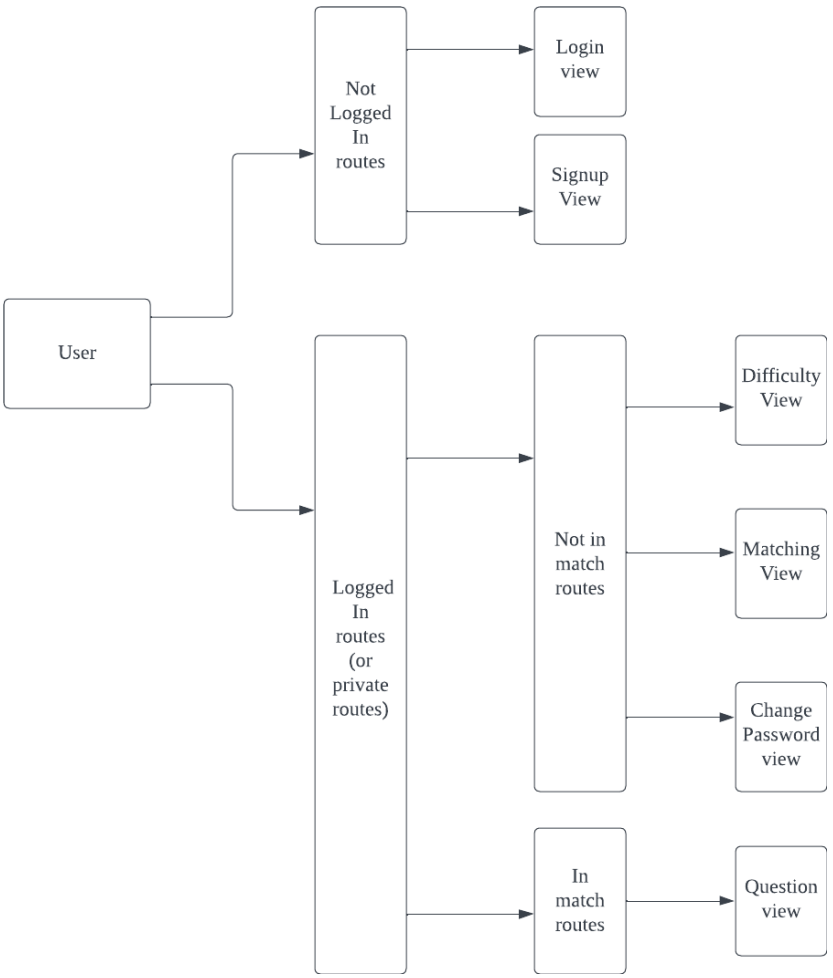
Axios post request used verify whether the user is authenticated

```
useEffect( effect: () => {
    socketMatchingServiceClient = io(BACKEND_URL, socketMatchingServiceConfig);
```

Socket Io client from the frontend connecting to the Socket Io matching service

Internal Structure:

The UI component is done in material design which is compiled from MUI react dependency.



The Frontend consists of the main views:

- User not authenticated:
    - Login view: Prompts the user to log in to peer prep.
    - Signup view: User will create an account for peer perp.
- User authentication needed but user is not in match:
    - Difficulty view: Users will choose the difficulty level of the match.
    - Matching: User will be waiting for another user before initiating a match.
    - Change Password view: Users will be changing password for their account.
- User authentication needed and user is in match:
    - Question view: The actual page where the users will answer questions, collaborate and chat with the other user in the same match.

The views in the frontend are controlled by the following routes using the react router dependency:

- User authentication not needed
    - Handled by the <NotLoggedInRoutes>:
        - Sends a axios post service to the user-service to check whether the user is authenticated
        - If axios receives a 401 unauthorised status from the user-service, the route will direct the user to the desired view.
        - Otherwise, the user will be redirected to the difficulty view (if not in match) or the question view (If in match).
- User authentication needed
    - Handled by the <PrivateRoutes>:
        - Sends a axios post service to the user-service to check whether the user is authenticated
        - If axios receives a 200 ok status from the user-service, the route will direct the user to the desired view.
        - Otherwise, the user will be redirected to the login view.
    - The Routes also contains nested routes which checks whether the user are in match:
        - There are handled by the <InMatchRoutes/> and <NotInMatchRoutes/> routes
            - The <NotInMatchRoutes/> routes use axios requests to the matching service to check whether the user is in match. If the user is not in match, the route will be directed to the desired view, otherwise, it will direct the user to the question view. The <InMatchRoutes/> on the other hand similarly uses axios requests to the matching service to check whether the user is in match as well, however, it directs the user to the question view if the user is in match, otherwise, it will direct the user to the difficulty view.

The frontend has a utility folder which contains the following utilities:

- The routes as mentioned to handle the different scenarios such as whether the user is authenticated or the user is in match.
- The “FormatDate.js” folder that formats the timestamp for the chat feature where it converts the timestamp to the date time format in “dd/mm hh:mm:ss (pm or am)”

The frontend also contains models that replicate the objects interacting around the frontend. Noteworthy objects include:

- Match: information about the match taken from the match-service is stored in the match class. The match class contains methods that will return information about the match such as “getUserInMatch(username)” which will return the other user in the match.
- Chat Info: Information about the chats, either from the sender or received from the chat-service are stored in the ChatInfo class.
- Question: Questions from the question service are stored in the Question chat.

Other noteworthy services:

- **Reverse Proxy:**
  - Proxies the requests to the corresponding microservice
  - Performs authorization check by sending side requests to Auth Service
  - Handles pre-flight requests from brower
  - The underlying component is nginx.
- **Match Creator Routine:**
  - Serves as a consumer for the match request queue
  - For two valid match requests, pair them into a match item and store it into Match Cache
  - Implemented in Golang
- **Match Request Queue:**
  - Serves as a message queue for match requests
  - Has three topics: “easy”, “medium”, and “hard”
  - The underlying component is Kafka along with Zookeeper.

Kubernetes Config

Before deploying to the cloud, we package all of our microservices into docker images, and push them onto docker hub, an image repository.

Then, we created k8s yaml config. Each microservice is encapsulated in a k8s Deployment. Service, PVC, and PV/StorageClass objects are also setup for network and persistence. Additionally, we created a ConfigMap to store the common environment variables, mainly service domain name + port.

Here is a table that summarizes the k8s config for microservices.

Microservices	k8s service type	k8s persistent volume claim?	Autoscaling?
User Service	ClusterIP	N	Y

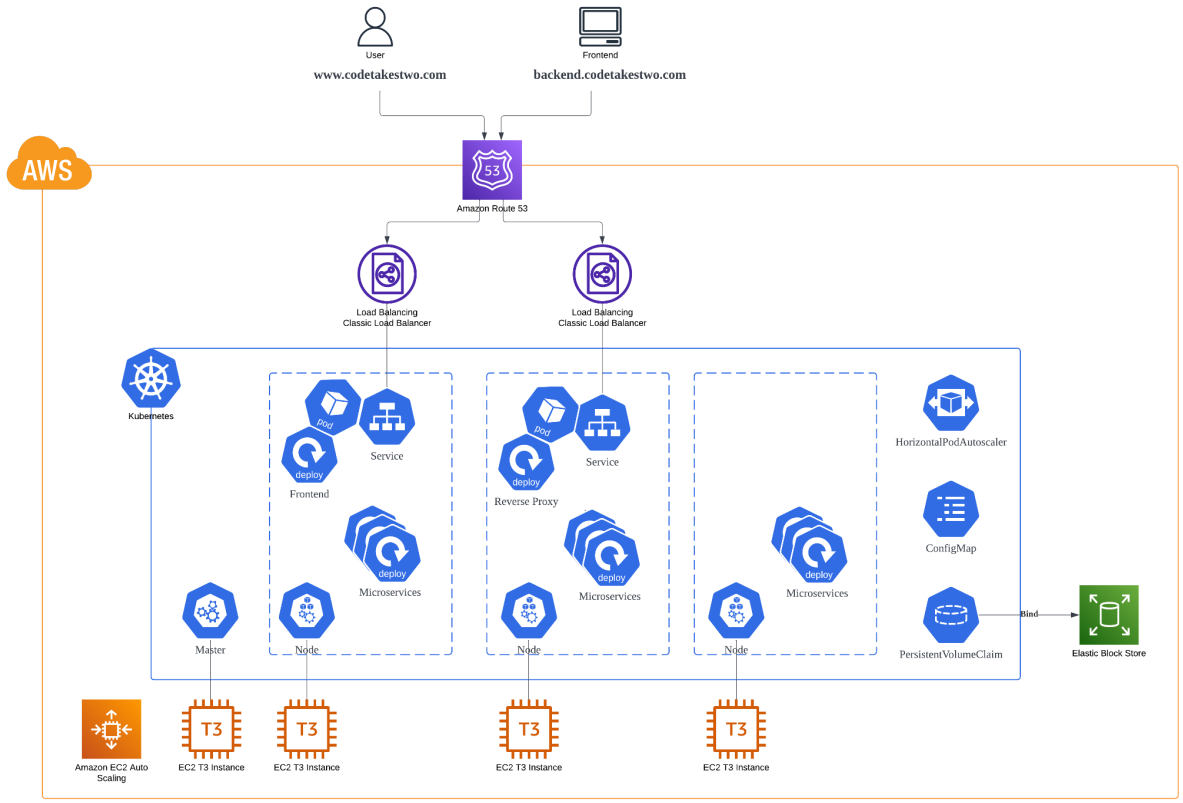
Matching Service	ClusterIP	N	Y
Auth Service	ClusterIP	N	Y
Question Service	ClusterIP	N	Y
Socketio Matching Service	ClusterIP	N	Y
Socketio Chat Service	ClusterIP	N	Y
Socketio Collab Service	ClusterIP	N	Y
Match Creator Routine	-	N	N
Reverse Proxy	NodePort -> LoadBalancer	N	Y
Frontend	NodePort -> LoadBalancer	N	Y
Cache	ClusterIP	Y	N
Database	ClusterIP	Y	N

There are several things to note regarding the table.

1. “Cache” refers to all the cache components, for example, User Cache, Match Cache, etc.
   “Database” refers to User/Question Database
2. On our local staging environment, both Reverse Proxy and Frontend are using NodePort as their service type. That is changed to LoadBalancer when we migrate to cloud.
3. Autoscaling is configured after we migrate to cloud.

With the k8s yaml files, we spinned up a local k8s environment with minikube as our staging environment to test and debug our application.

Cloud Deployment



Our product is deployed to AWS using kops, which automatically creates and validates the k8s cluster. Both master and worker nodes are configured to use EC2 t3.large. The worker nodes are within an EC2 autoscaling group with a minimum of 3 instances and a maximum of 6. All of our nodes are deployed in the Singapore region.

K8s configs are also refactored during our migration to cloud. For Reverse Proxy and Frontend, we changed our Service type to LoadBalancer, exposing the two endpoints. We also changed our PersistenVolumeClaim type from HostPath to EBS StorageClass.

The URIi for our website is www.codetakestwo.com.

Handling a request
When a user visits www.codetakestwo.com, the request will be routed by AWS through the landing zone and Load Balancer, and reach the k8s Service for frontend running on one of the nodes. Then, the frontend (which is a combination of a base html file, javascript, css, and others) will run on the user's browser.

To request data from backend, the frontend (running on the browser) will a send a request to backend.codetakestwo.com, which is routed by AWS and reaches the k8s service for **Reverse Proxy**. The reverse proxy then will pass the request to the corresponding microservice.

AWS Resources:

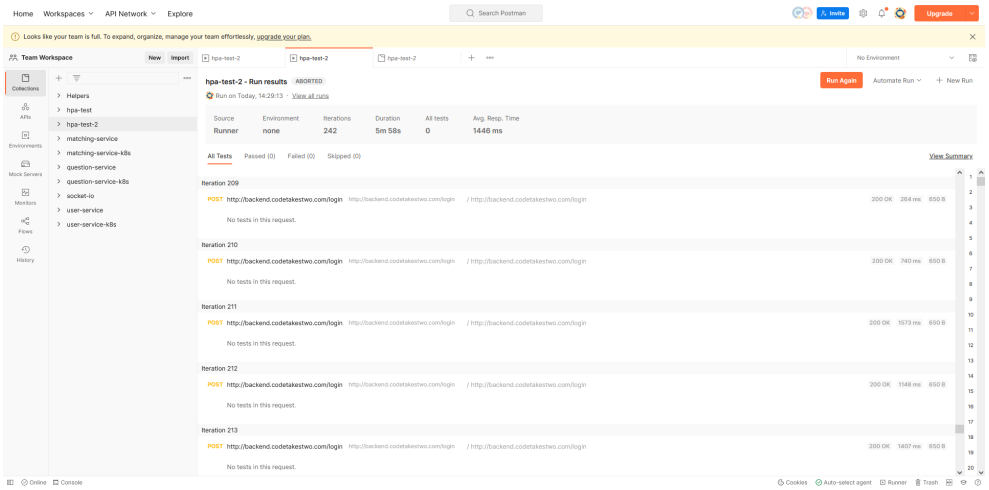
- Router 53 landing zone for the domain name "codetakestwo.com"

- Two classic load balancers
- 4-7 EC2 t3 instances
- EC2 autoscaling groups
- EBS volumes (gp2, gp3) for microservices storage and master node operations
- S3 bucket for kops state store
- Other resources

HPA:
We have deployed Horizontal Pod Autoscaler for most of the service pods. Maximally, we could have 6 worker nodes (EC2 T3.large) running at the same time, which will provide 6000 CPU millicores. Therefore, we set up CPU request &

HPA Target	CPU request & limit / millicore	Min # Pod	Max # Pod	Worst Case CPU Usage / millicore
User Service	100	1	3	300
Matching Service	200	1	3	600
Question Service	100	1	3	300
Auth Service	200	1	3	600
Socketio Matching Service	200	1	3	600
Socketio Chat Service	100	1	3	300
Socketio Collab Service	200	1	3	600
Reverse Proxy	200	1	3	600
Frontend	200	1	3	600

User Service load test:
To test the Autoscaler, we conducted a simple load test on User Service. We used 7 Postman Collection Runners on 4 host machines to send login requests to the backend API. The requests were sent concurrently for 6 minutes. In total, 1686 requests were sent, and all of the requests were successfully handled. The average request rate is 281 requests/min.



One of the seven Postman Collection Runners

On the backend, the request will be passed from Reverse Proxy to User Service, which will then perform a DB query and a cache query. Therefore, we expect that the CPU load of Reverse Proxy, User Service, Blacklist Cache and User/Question Database should increase. In reality, only user service has a significant CPU increase, while other three do not.

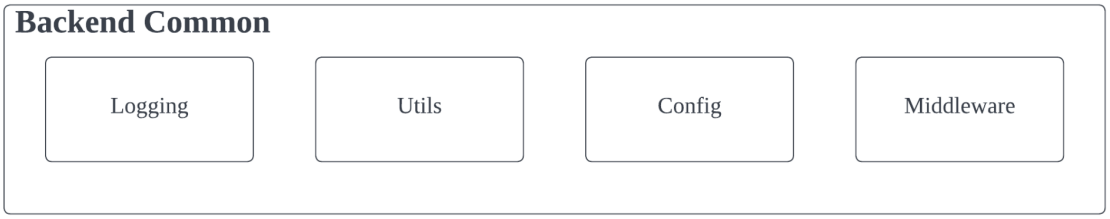
```
[ec2-user@ip-172-31-40-148 codetakestwo-k8s]$ kubectl get hpa user-service -w
NAME           REFERENCE                     TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
user-service   Deployment/user-service       <unknown>/80%   1         3         0          8s
user-service   Deployment/user-service       0%/80%          1         3         1          15s
user-service   Deployment/user-service       0%/80%          1         3         1          45s
user-service   Deployment/user-service       34%/80%         1         3         1          60s
user-service   Deployment/user-service       100%/80%        1         3         1          75s
user-service   Deployment/user-service       100%/80%        1         3         2          90s
user-service   Deployment/user-service       100%/80%        1         3         2          105s
user-service   Deployment/user-service       87%/80%         1         3         2          2m
user-service   Deployment/user-service       84%/80%         1         3         2          2m15s
user-service   Deployment/user-service       81%/80%         1         3         2          2m30s
user-service   Deployment/user-service       95%/80%         1         3         2          2m45s
user-service   Deployment/user-service       86%/80%         1         3         3          3m
user-service   Deployment/user-service       66%/80%         1         3         3          3m15s
user-service   Deployment/user-service       91%/80%         1         3         3          3m30s
user-service   Deployment/user-service       81%/80%         1         3         3          3m45s
user-service   Deployment/user-service       74%/80%         1         3         3          4m
user-service   Deployment/user-service       70%/80%         1         3         3          4m15s
user-service   Deployment/user-service       88%/80%         1         3         3          4m30s
user-service   Deployment/user-service       91%/80%         1         3         3          4m45s
user-service   Deployment/user-service       74%/80%         1         3         3          5m
user-service   Deployment/user-service       73%/80%         1         3         3          5m15s
user-service   Deployment/user-service       80%/80%         1         3         3          5m30s
user-service   Deployment/user-service       66%/80%         1         3         3          5m45s
user-service   Deployment/user-service       83%/80%         1         3         3          6m
user-service   Deployment/user-service       79%/80%         1         3         3          6m15s
user-service   Deployment/user-service       79%/80%         1         3         3          6m30s
user-service   Deployment/user-service       82%/80%         1         3         3          6m45s
user-service   Deployment/user-service       87%/80%         1         3         3          7m
user-service   Deployment/user-service       51%/80%         1         3         3          7m15s
user-service   Deployment/user-service       0%/80%          1         3         3          7m30s
user-service   Deployment/user-service       0%/80%          1         3         3          9m45s
user-service   Deployment/user-service       0%/80%          1         3         3          12m
user-service   Deployment/user-service       0%/80%          1         3         2          12m
user-service   Deployment/user-service       0%/80%          1         3         1          12m
```

Design Decisions & Patterns

Socketio v.s. HTTP

- Bidirectional communication: The HTTP protocol is a unidirectional protocol that only allows the server to make a response to the request from a client. However, some services require the server-to-client data transmission without a request from clients which is not supported by HTTP protocol. By using socketio, either client or server can send a message to the other party without any request required.
- Real-time communication: Some services, like chat service and collab service, require a real-time communication and continuous stream of data transmission. In the context of real-time, socketio is usually a better choice because of its extremely lightweight footprint on the server side.
- Room broadcasting: Socketio provides a feature called room, an arbitrary channel, that can be used to broadcast events to a specific subset of clients. For example, in question service, when one of the users in a room emits the event to move on to the next question, the new question will be updated to every user in the room without any other request from clients. More details can be found here.

REST Service Common Package



While developing the second REST service (matching service), we realised that actually many of the functions and logic from the previous service can be reuse. Following the “DRP” (Do not Repeat Yourself) principle, we extract the commonly used component in our code and created the common package, which is shared by all the Golang microservices.

The common package can be divided to 4 parts:

- Logging provides logging for errors and messages customized by ourselves. It’s generally more fruitful than the plain fmt.Printf. For example, it can log the file and line number that calls the log function, which is helpful for debugging.
- Utils provides utility functions, e.g. transitions between string and unsigned integers.
- Config stores commonly used constants, e.g. environment variable keys.
- Middleware provides gin Handlers. They can be used within the series of handlers for an URL to, e.g. extract header values, since we are storing the cookie contents into headers.

Project Management

Timeline

Time	Frontend	Backend
Week 3	Prepare UI for log in and sign up pages	Build persistence layer for user service. Create Dockerfile and docker-compose files for user service, database and cache.
Week 4	Prepare UI for difficulty page	Create API handlers for CRUD operations on users. Implement login functionality with JWT. Hash and salt password.
Week 5	Integrate the API of user service with the frontend Check if the cookie is in before using the service, otherwise redirect them to the login page	Add logout user and change password functionalities in user service. Create Postman tests API Create schema for matching service. Implement and test Kafka producer and consumer in golang.
Week 6	Integrate the API of matching service with the frontend	Implement user matching with Kafka. Implement cache services. Implement socket io.
Week 7	Add the page where users in a match will collaborate - Questions - Collaboration	Add reconnection mechanism to matching service. Create reverse proxy for authorization.
Week 8	Add the page where users in a match will collaborate - Chat popup	Create question service. Add chatting functionality to socket io server.
Week 9	Add test cases for frontend logic.	Update question service APIs.
Week 10	Merge with backend	Merge with frontend.

Week 11	Implement chat service on frontend.	Add chat history in chat service. Add award system to user service. Deploy to local staging environment. Set up CI.
Week 12	Implement collab service on frontend.	Deploy to AWS.
Week 13	Style frontend pages. Fix bugs and test. Write report	Fix bugs and test. Write reports.

SDLC

The SDLC life cycle was done using the Agile process. We conduct sprints where we discuss our current progress on what we have done so far, as well as what we were lacking. The process that we were lacking will be considered for this sprint's backlog, as well as the new features that will be developed. However, sprints are conducted irregularly due to school commitments on our side.

Our main communication channel is done using zoom, whereas any ad hoc communication such as queries about the backend apis and planning for sprints.

Continuous Integration
After Milestone 2, we set up our Continuous Integration using github actions. Upon any PR or push on the dev branch, the CI will build all the images from the code, tag images, and push them to the image repository. In this way, our images on image repo are always aligned with the latest progress of development. The CI also saves us much time and effort (to manually update the images).

Challenges

Time Management

Our team had difficulties meeting the requirements for our milestone 1 and 2 presentation, due to the delayed integration between frontend and backend. We could have spent more time before the deadline integrating and testing out the different components, instead of only integrating it the day before the presentation. There are also times where we would fall slightly behind the scheduled timeline and then have troubles trying to catch up and rush to meet the deadline.

Communications

Since our team did not have regular stand ups, we would only meet whenever there's a need to (e.g. deadline approaching, new feature needed). This did not workout well as we would

often not be able to plan our time ahead for the stand ups itself. Additionally, we also did not use our group chat very often to communicate with one another and often worked in silo.

Unfamirality with the technology used

Most of us were not familiar with the infrastructure needed to construct our project. For example, the frontend team do not have even the basic knowledge on react.js to implement the websites. Hence, a lot of googling on the react features and tricks such as states, hooks and effects are needed to better understand them. Furthermore, npm has a lot of dependencies which most of them we were not familiar off and as such, we have to constantly google to determine the best dependency for our needs. Backend faces the same situation. In order to implement the functionality we want, we spent a lot of time learning technologies, such as kafka and socketio etc which we never touched before, during our project.

CORS issue

When we integrate the backend services with the frontend, we will always have the CORS issue that prevents the frontend from communicating with the backend services. Especially with the reverse proxy involved, we then have to learn how to allow cors in the nginx configuration file.

References

- Distributed lock mechanism: https://redis.io/commands/setnx/
- Redis OM: https://redis.io/docs/stack/get-started/tutorials/stack-node/
- Socket io: https://socket.io/
- Quill: https://quilljs.com/