PeerPrep

CS3219 Project Report

Group	G04	
Members	Cheng Jun Yue Jedidiah	A0234928W
	Hing Zi Yang Benedict	A0235187Y
	Irving Alexander De Boer	A0238894J
	Kimberly Barcelon Pontanares	A0240884Y
	Lim Yi Kai Edwin	A0234972X
Main Repo	https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g04	
Sub-Repos	https://github.com/orgs/CS3219-AY2324S1-G04/repositories	

Table of Contents

PeerPrep

Introduction

Background

As the field of computing continues to evolve, the need for highly-skilled, well-trained, and well-equipped computer scientists grows ever stronger. On top of knowing how to write code, many companies now expect developers to possess strong problem solving skills. Said skills are often tested during the hiring phase, in the form of technical interviews.

Like any other skill, the ability to solve coding problems can be developed through practice. However, it is common for individuals to encounter problems that they cannot solve. Although, searching forums and looking at solutions written by others is a possible means of learning, it lacks the thrill of problem solving and fails to develop a sense of confidence in one's own ability.

In such situations, having someone to talk to and bounce ideas off can be helpful in giving an individual a different perspective, thus helping them discover solutions they otherwise would not have been able to.

Value Proposition

PeerPrep is a platform that aims to help developers hone their problem solving skills and ability to tackle technical interviews. PeerPrep matches developers looking to solve problems of the same difficulty and type, and provide them with a question as well as collaborative tools to allow them to collaborate on said question. With PeerPrep, developers can work together to refine their coding prowess.

Contributions

Sub-Group A

Cheng Jun Yue Jedidiah	**Must-Have Features**
	● Editor Service and Docs Service for supporting collaborative code editor [M4] ● Front-end UI and logic for collaborative code editor [M4] [M5] ● Editor Service and Docs Service deployment via Docker compose [M6] ● Kubernetes deployment for Editor Service and Docs Service [M6]
	Nice-to-Have Features
	● Enhanced Editor Service with improved code editor [N5] ● Configure horizontal pod autoscaling for Editor Service and Docs Service [N10]
	Report
	● Documentation for Editor Service ● Documentation for Docs Service ● Requirements ● Development Process (Design) ● Architecture diagram ● Requirement traceability
	Misc
	● Overall architecture design ● Meeting note taker
Hing Zi Yang Benedict	**Must-Have Features**
	● User Service for managing user information [M1] ● Front-end UI for user CRUD [M1] [M5] ● Front-end background task for interacting with User Service (e.g. renewing access tokens) [M1] [M5] ● Room Service for managing matched users (developed with Edwin) [M2] ● User Service deployment via Docker compose [M6] ● Webpage Service (for serving webpage) deployment via Docker compose [M6] ● Kubernetes deployment for all services except for Editor Service and Docs Service [M6]
	Nice-to-Have Features

	● Configure horizontal pod autoscaling for all services except for Editor Service and Docs Service [N10] ● Gateway Service for creating a single point of entry to the back-end services [N11] ● Service discovery via Kubernetes DNS resolver [N12]
	Report
	● Documentation for User Service ● Documentation for Gateway service ● Documentation for Room service ● Dev Documentation for Microservice design ● Dev Documentation for Code Convention
	Misc
	● Overall architecture design ● Meeting facilitator (setup agendas and lead discussions) ● Create Typescript conventions for back-end (code linting, code formatting, and code structure etc.) ● Create GitHub organisation ● Create GitHub main repo structure

Sub-Group B

Irving Alexander De Boer	**Must-Have Features**
	● Question Service for managing question information [M3] ● Question Service deployment via Docker compose [M6]
	Nice-to-Have Features
	● Chat Service for facilitating text-based communication between users in collaborative space [N1] ● Enhance Question Service with admin facing management API [N4] ● Enhance Question Service with question retrieval [N4] ● Enhance Question Service with Redis cache for speeding up retrieval [N4]
	Report
	● Documentation for Question Service ● Documentation for Chat Service
	Misc
	● Standardised REST API endpoints

Kimberly Barcelon Pontanares	**Must-Have Features**
	● Front-end architecture design [M5] ● Front-end UI design [M5] ● Front-end UI for questions CRUD [M3] [M5] ● Front-end UI for user matching [M2] [M5]
	Nice-to-Have Features
	● Front-end UI for sending and receiving chat message [M5] [N1] ● Front-end UI for sorting, filtering, and searching for questions with pagination [N4] [M5] ● Front-end UI for user attempts & rankings [N2]
	Report
	● PeerPrep Features ● Functional Requirements ● Non-Functional Requirements ● Documentation for Webpage Service
	Misc
	● Chief front-end developer ● All things front-end that is not specified in other member's contributions
Lim Yi Kai Edwin	**Must-Have Features**
	● Matching Service for matching users based on specified criterias [M2] ● Room Service for managing matched users (developed with Benedict) [M2] ● Docker Service deployment via Docker compose [M6] ● Matching Service deployment via Docker compose [M6]
	Nice-to-Have Features
	● Attempt History Service for managing user attempts [N2] ● Attempt History Service deployment via Docker compose [M6]
	Report
	● Documentation for Matching Service ● Documentation for Room Service ● Documentation for Attempt History Service ● Requirement traceability
	Misc

	● Project Slides

PeerPrep

Requirements

PeerPrep Features

The following are the features envisioned for PeerPrep.

Must-Have Features (Mx)

M1: User Service
Responsible for user profile and session management.

M2: Matching Service
Matches users based on some criteria (e.g., difficulty level of questions, topics, proficiency level of the users, etc.) This service can potentially be developed by offering multiple matching criteria.

M3: Question Service
Maintains a question repository indexed by difficulty level (and any other indexing criteria – e.g., specific topics).

M4: Collaboration Service
Provides the mechanism for real-time collaboration (e.g. concurrent code editing) between the authenticated and matched users in the collaborative space.

M5: Basic UI
Facilitates user interaction to access the app that you develop.

M6: Deployment
Deployment of the application on Docker.

Nice-To-Have Features (Nx)

N1: Communication

Facilitates communication among the participants in the collaborative space (other than the shared workspace) e.g., text-based chat service and/or video (+voice) calling service.

N2: Attempt History

Maintains a record of the questions attempted by the user.

N4: Enhanced Question Service

Provides additional question management functionality for question service.

N5: Enhanced Collaboration Service

Provides enhancements to the code editor such as syntax highlighting and auto-completion.

N10: Scalability

Provides ability to scale the containerised application when heavy load is experienced.

N11: API Gateway

Provides a way for services to communicate with each other while preventing unauthorised access to certain endpoints.

N12: Service Discovery

Provides a way for containerised services to discover each other.

Functional Requirements (FRs)

Sprint 1 - MVP

See Sprint 1 (Week 4 to 7) for our development process.

F1 User Service

ID	Description	Priority
F1.1	**Database**	
F1.1.1	The system should store user profile information	**H**
F1.1.2	The system should store user login credentials	**H**
F1.1.3	The system should store user session tokens	**H**
F1.2	**Back-end**	
F1.2.1	The system provides an API for registering a user account	**H**
F1.2.2	The system provides an API for logging in	**H**
F1.2.3	The system provides an API for logging out	M
F1.2.4	The system provides an API for fetching the authenticated user's profile	M
F1.2.5	The system provides an API for updating the authenticated user's profile	M
F1.2.6	The system provides an API for changing the authenticated user's password	L
F1.2.7	The system provides an API for deleting a user's account	L
F1.3	**Front-end**	
F1.3.1	The webpage provides a UI for registering an account	**H**
F1.3.2	The webpage provides a UI for logging in and out	**H**
F1.3.3	The webpage provides a UI for deleting an authenticated user’s account	M
F1.3.4	The webpage provides a UI for updating an authenticated user’s profile	M
F1.3.5	The webpage provides a UI for changing an authenticated user’s password	M
F1.3.6	The webpage provides a UI for viewing an authenticated user's profile	M

F2 Matching Service

ID	Description	Priority
F2.1	**Database**	
F2.1.1	The system should store matchmaking requests from users	**H**
F2.2	**Back-end**	
F2.2.1	The system provides an API for adding authenticated users to matching queue	**H**
F2.2.2	The system provides an API for removing authenticated users from matching queue	**H**
F2.2.3	The system provides an API for checking if user is in Queue	**H**
F2.3	**Front-End**	

F2.3.1	The webpage provides a UI for matchmaking based on criteria	**H**

F3 Question Service

ID	Description	Priority
F3.1	**Database**	
F3.1.1	The system should store question information	**H**
F3.2	**Back-end**	
F3.2.1	The system provides an API for retrieving all questions	**H**
F3.2.2	The system provides an API for retrieving specific question by ID	**H**
F3.2.3	The system provides an API for deleting a specific question	**H**
F3.2.4	The system provides an API for adding a question	M
F3.2.5	The system provides an API for updating specific question	M
F3.2.6	The system provides an API for retrieving questions indexed by difficulty	M
F3.2.7	The system provides an API for retrieving questions indexed by topic	M
F3.2.8	The system provides an API for retrieving questions based on criterias	**H**
F3.3	**Front-end**	
F3.3.1	The webpage provides a UI for viewing a list of questions	**H**
F3.3.2	The webpage provides a UI for viewing a question's details	**H**
F3.3.3	The webpage provides a UI for creating a question	M
F3.3.4	The webpage provides a UI for updating a question	M
F3.3.5	The webpage provides a UI for deleting a question	M

Sprint 2 - MVP

Please see Sprint 2 for our development process.

F1 User Service

ID	Description	Priority
F1.2	**Back-end**	
F1.2.9	The system provides an API for other services to retrieve the public key for JWT token authentication	**H**
F1.3	**Front-end**	
F1.3.7	The webpage should ensure access tokens do not expire	**H**

F2 Matching Service

ID	Description	Priority
F2.3	**Front-End**	
F2.3.2	The webpage provides a UI for leaving a matchmaking queue	**H**

F2.3.3	The webpage provides a UI for quick matchmaking	L

F4 Room Service

ID	Description	Priority
F4.1	**Database**	
F4.1.1	The system should store room information	**H**
F4.2	**Back-end**	
F4.2.1	The system provides an API for retrieving a user's room information	**H**
F4.2.2	The system provides an API for adding a user to a room	**H**
F4.2.3	The system provides an API for removing a user from a room	**H**
F4.2.4	The system provides an API for keeping a room alive	M
F4.2.5	The system should remove inactive rooms after a specified duration	M
F4.2.6	The system provides a message queue for listening room events such as room creation and deletion	M
F4.3	**Front-end**	
F4.3.1	The webpage provides a UI for joining the room	**H**
F4.3.2	The webpage provides a UI for leaving the room	**H**

F5 Editor-Docs Service

ID	Description	Priority
F5.1	**Database**	
F5.1.1	The system should store editor code	**H**
F5.2	**Back-end**	
F5.2.4	The system provides an API for retrieving code from a specified editor	**H**
F5.2.5	The system provides a socket endpoint that allows an authenticated user to sync editor data	**H**
F5.3	**Front-end**	
F5.3.1	The webpage provides a real-time collaborative code editor with syntax highlighting	**H**

F6 Chat Service

ID	Description	Priority
F6.1	**Database**	
F6.1.1	The system should store chat information	L
F6.2	**Back-end**	
F6.2.1	The system provides a socket endpoint for authenticated users to connect to a chat room	L

F6.3	**Front-end**	
F6.3.1	The webpage provides a UI for sending messages to a chat room	L
F6.3.2	The webpage provides a UI for receiving messages from a chat room	L

F7 Attempt History Service

ID	Description	Priority
F7.1	**Database**	
F7.1.1	The system should store user attempts	M
F7.2	**Back-end**	
F7.2.1	The system provides an API for retrieving all submissions by a specific user	M
F7.2.2	The system provides an API for retrieving a specific submission of a specific user	M
F7.2.3	The system should record a user attempt as a submission upon room closure	M
F7.2.4	The system provides an API for retrieving the ranking of all users based on attempts.	L
F7.3	**Front-end**	
F7.3.1	The webpage provides a UI for viewing all submissions.	M
F7.3.2	The webpage provides a UI for viewing a specific submission.	M
F7.3.3	The webpage provides a UI for viewing the ranking of all users.	L

Non-Functional Requirements (NFRs)

NF1 All Services

ID	Description
NF1.1	**Compatibility**
NF1.1.1	Must run on Google Chrome, Safari, and Firefox browsers
NF1.1.2	Must run on Windows, MacOS, and Linux platforms
NF1.2	**Accessibility**
NF1.2.1	Users should be able to resize fonts
NF1.2.1	App should be responsive and can accommodate all screen sizes
NF1.2.2	Text should be legible 13 inch monitor (macbook air size)
NF1.2.3	Visuals should be able to support colour-blindness
NF1.2.4	Users should receive timely feedback in the event of error
NF1.2.5	App should support dark-mode
NF1.2.6	App must have a cool favicon
NF1.3	**Development**
NF1.3.1	Error handling logs

NF2 User Service

ID	Description
NF2.1	**Capacity**
NF2.1.1	Support up to 100k user signups
NF2.1.2	Support up to 500 concurrent logged-in users
NF2.2	**Speed**
NF2.2.1	Users should be able to log in within 5 seconds
NF2.3	**Security**
NF2.3.1	Inactive user sessions should be invalidated after 1 week
NF2.3.2	User password must be > 8 characters, alphanumeric, etc

NF3 Matching Service

ID	Description
NF3.1	**Speed**
NF3.1.1	Users can find a match under 5 minutes

NF4 Question Service

ID	Description
NF4.1	**Availability**
NF4.1.1	Users should be able to view a list of all questions within 10 seconds
NF4.1.2	Users should be able to view question details instantaneously

NF5 Editor-Docs Service

ID	Description
NF5.1	**Capacity**
NF5.1.1	Support up to 100 concurrent collaboration rooms
NF5.1.2	Collaboration service latency should be under 5 seconds

NF5 Chat Service

ID	Description
NF7.1	**Availability**
NF7.1.1	Users must connect to chat server upon opening of chat module
NF7.1.2	Users must be able to receive sent messages instantaneously

NF5 Attempt History Service

ID	Description
NF8.1	**Speed**
NF8.1.1	Users should be able to view a list of all attempts within 30 seconds
NF8.1.2	Users should be able to view the ranking of all users within 30 seconds
NF8.2	**Availability**
NF8.2.1	Users should be able to see latest attempt within 2 minutes of submission

Requirement Traceability

Maps user requirements to functional requirements

Use Cases	Description	Functional Requirement
		F1.1.1
		F1.1.2
		F1.2.1
UC1.1	User registers account	F1.3.1
		F1.1.3
		F1.2.2
UC1.2	User logins	F1.3.2
		F1.2.3
UC1.3	User logouts	F1.3.2
		F1.2.4
UC1.4	User views profile	F1.3.6
		F1.2.5
UC1.5	User updates profile	F1.3.4
		F1.2.6
UC1.6	User changes password	F1.3.5
		F1.2.7
UC1.7	User deletes account	F1.3.3
		F2.1.1
		F2.2.1
		F2.2.3
UC2.1	User joins matchmaking queue based on selected criterias	F2.3.1
		F2.2.1
UC2.2	User quick joins matchmaking queue	F2.3.3
		F2.2.2
UC2.3	User leaves matchmaking queue	F2.3.2
		F3.1.1
		F3.2.1
UC3.1	User views all questions	F3.3.1
		F3.2.2
UC3.2	User view a specific question	F3.3.2

UC3.3	User is able to questions sort based on criterias	F3.2.6
		F3.2.7
UC3.4	Admin can create question	F3.2.4
		F3.3.3
UC3.5	Admin can update question	F3.2.5
		F3.3.4
UC3.6	Admin deletes question	F3.2.3
		F3.3.5
UC4.1	User enters room	F4.1.1
		F4.2.1
		F4.2.2
		F4.2.4
		F4.3.1
UC4.2	User leave room	F4.2.3
		F4.2.5
		F4.3.1
		F4.3.2
UC4.3	User collaborates with users	F4.2.4
		F5.1.1
		F5.2.5
		F5.3.1
UC4.4	User is able to chat with users	F6.1.1
		F6.2.1
		F6.3.1
		F6.3.2
UC4.5	User submits code	F4.2.3
		F7.2.3
UC5.1	User views all of their submissions	F7.2.1
		F7.3.1
UC5.2	User views submitted code	F7.2.2
		F7.3.2
UC5.3	User views ranking	F7.2.4
		F7.3.3

PeerPrep

Development Process

Methodology

Our development process was adapted from the Scrum framework. We adopted sprints as a way of breaking up our development process into manageable chunks. We also decided to host regular meetings.

Although the Scrum framework involves daily meetings, we agreed that doing so would likely prove to be unproductive. As all members have different timetables and are also juggling submissions for other modules, it is unlikely that everyone would make enough progress each day to be worth communicating to the other members. Hence, we decided on a weekly meeting, held at 2pm every wednesday.

Through these regular meetings, members were able to be caught up to speed with the development progress of every other member. The benefits of this includes:

- Detecting misinterpretations and miscommunications regarding feature requirements
- Detecting bottlenecks caused by dependencies between features worked on by different members
- Detecting overlooked features

Design (Week 2 to 3)

The design phase was an essential part of our software development process.

The features, which are also outlined in the *PeerPrep Project Brief,* served as a starting point. To figure out how PeerPrep could deliver actual value to our targets, we sought to understand the needs of our target user.

We looked into how various companies conducted their technical interviews and identified key processes. We also analysed similar coding interview preparation platforms such as LeetCode and HackerRank.

Using the insights from our research, we defined the value proposition of *PeerPrep* to guide the direction of the project.

We then created a product backlog for our application and specified a set of functional and non-functional requirements.

We discussed with our mentor to clarify some of the technical requirements for the project. We then explored various software and tools that could be useful for this application. e.g. Google Cloud, Amazon E2, Discord API for voice chat.

After determining necessary design details such as tech stack and architectural pattern, we drafted an architectural diagram. Adopting domain driven design allowed us to split the workload among team members based on Bounded Contexts.

To ensure a level of consistency in every team member's development environment, we outlined proper guidelines for coding conventions.

Sprint 1 (Week 4 to 7)

Minimum Viable Prototype

We aimed to deliver the core features of PeerPrep, namely
- User Service
- Matching Service
- Question Service
- Webpage Service

These features would satisfy the requirements of a **Minimum Viable Prototype (MVP)**.

Sprint 2 (Week 8 to 10)

After creating the MVP, our team started looking at the nice to haves and additional features that might be useful. Although some were considered (i.e. code judge, AI chat), not all were implemented.

Sprint 3 (Week 11 to 13)

Here quality assurance and developer testing was done to ensure that the applications are integrated together.

Documentation (Week 13)

Here we largely focused on standardising documentation to allow a third party reader to understand how to start up and potentially modify our application.

PeerPrep

Developer Documentation

Git Workflow

Repository Structure

The PeerPrep codebase adopts a multi-repo structure. One repository is created for each microservice, all managed under the CS3219-AY2324S1-G04 GitHub organisation.

The multi-repo structure was chosen over the monorepo structure due to the following benefits:

- Multi-repo structure allows each microservice to have a clean and traceable commit history, independent of the commit history of other microservices. This makes it easy to identify commits where specific changes are made in the event that a revert is necessary.
- Multi-repo structure allows members to easily identify if changes were made to a specific microservice. This lets each member determine if said changes apply to the features they are developing based on whether the feature relies on the microservice that was modified. Members can thus, easily discern whether they need to know what was done in said commit.
- Multi-repo structure allows for a clear division of responsibility. Since each microservice is mainly developed by one member, said member can be placed in charge of the entire repository and enforce the rules and best practices in the repository.

As the project requirement demands that the main repository track all code across all microservices, Git submodules were used to create links from the main repository to the microservice repositories.

Feature Workflow

The PeerPrep codebase adopts a feature branch workflow. Be it a feature, documentation update, or bug fix, a branch is to be created to house all commits related to it. Once the intended changes are complete, only then is a pull request made, followed by the feature branch being merged back into the master branch.

The benefit of this workflow is that:

- Features can be developed in parallel.

- Commits can be easily traced to the feature it is related to or the bug it is intended to fix.
- Multiple members can work on the same repository at the same time.

The last benefit mentioned above is largely only applicable to the Webpage Service which has multiple members working on it. Most of the microservices are developed entirely by a single member with occasional small contributions from other members.

The forking workflow was also considered as it provides the same benefits with the additional strength of allowing for better repository security. However, the additional overhead of having to ensure that one's repository clone is kept up to date with the main repository was considered to be too big of a detriment to be worth adopting. This decision was made taking into account that the team only consists of 5 members with a common goal and thus the possibility of a member going rogue and sabotaging a repository is very low.

Code Convention

Back-end Services

The following code conventions are only applicable to back-end services (i.e. all microservices other than Webpage Service).

A repository was created for defining these conventions. Said repository also doubles as a template that any back-end service can use as a starting point. The back-end code convention repository can be found here.

The following are the developer tools utilised:
- Language: Typescript
- Runtime Environment: NodeJS 18 LTS
- Testing Framework: Jest
- Documentation Language: JSDoc
- Code Formatter: Prettier
- Linter: ESLint

The following are some notable conventions:
- File Naming Convention: Snake case
- Typescript Naming Convention: ESLint's camel case convention described here
- Typescript Class Member Ordering: Default ESLint member ordering described here

For the full list of conventions, please refer to the repository. The typescript conventions are defined via an ".eslintrc" file. The Prettier formatting is defined via a ".prettierrc" file.

Webpage Service

The following code convention is applicable only to the Webpage Service.

The following are the developer tools utilised:

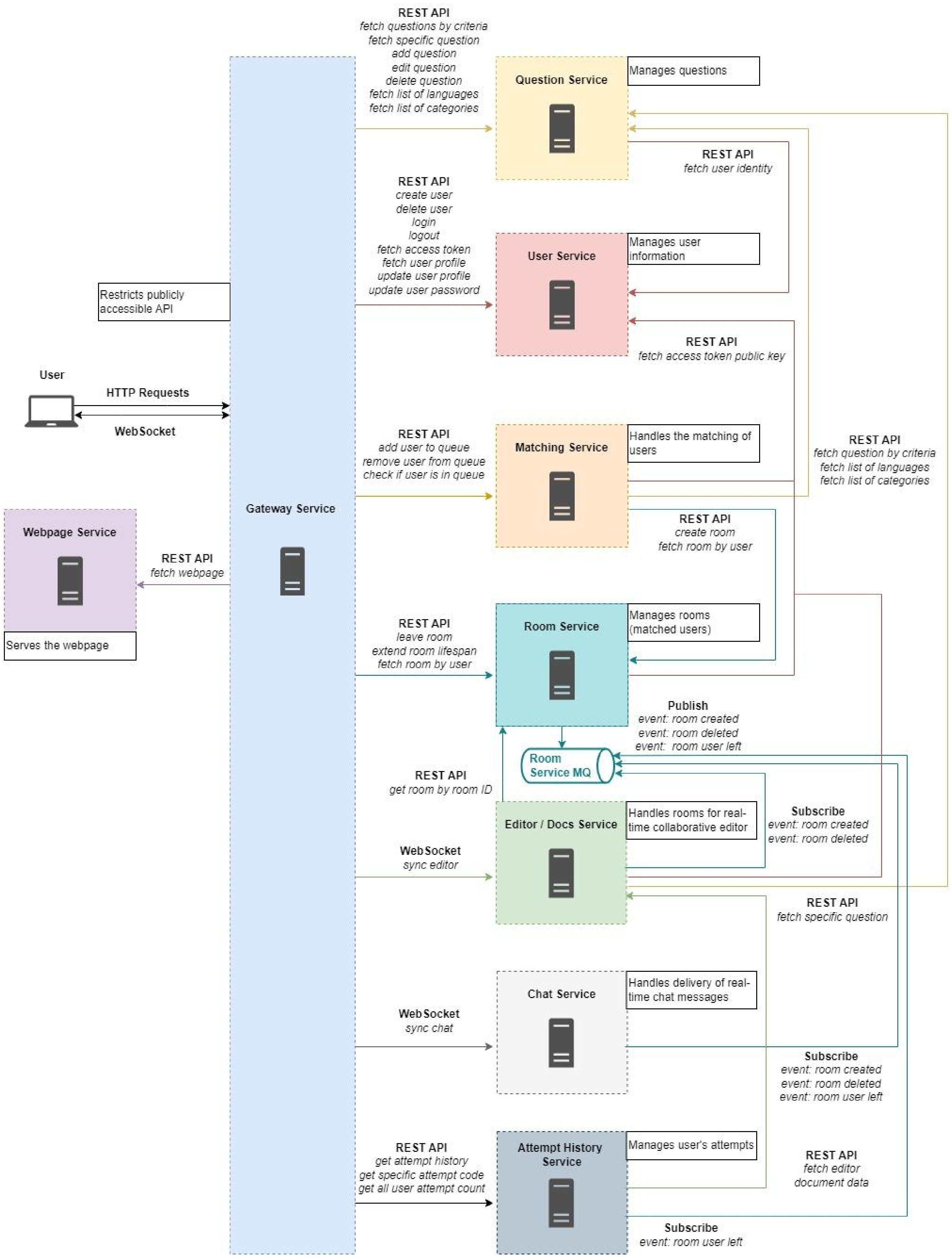
- Language: Typescript
- Runtime Environment: NodeJS 18 LTS
- Bundler: Vite
- Code Formatter: Prettier
- Linter: ESLint

The following are some notable conventions:

- File Naming Convention (for React Components): Pascal case
- File Naming Convention (for others): Camel case

PeerPrep Architecture Overview

Refer to the overall architecture diagram on the next page.


REST API
fetch questions by criteria
fetch specific question
add question
edit question
delete question
fetch list of languages
fetch list of categories
Question Service
Manages questions
REST API
fetch user identity
REST API
create user
delete user
login
logout
fetch access token
fetch user profile
update user profile
update user password
User Service
Manages user information
Restricts publicly accessible API
REST API
fetch access token public key
User
HTTP Requests
WebSocket
REST API
add user to queue
remove user from queue
check if user is in queue
Matching Service
Handles the matching of users
REST API
fetch question by criteria
fetch list of languages
fetch list of categories
Gateway Service
Webpage Service
REST API
fetch webpage
Serves the webpage
REST API
create room
fetch room by user
REST API
leave room
extend room lifespan
fetch room by user
Room Service
Manages rooms (matched users)
Publish
event: room created
event: room deleted
event: room user left
Room Service MQ
REST API
get room by room ID
Editor / Docs Service
Handles rooms for real-time collaborative editor
Subscribe
event: room created
event: room deleted
WebSocket
sync editor
REST API
fetch specific question
Chat Service
Handles delivery of real-time chat messages
WebSocket
sync chat
Subscribe
event: room created
event: room deleted
event: room user left
REST API
get attempt history
get specific attempt code
get all user attempt count
Attempt History Service
Manages user's attempts
REST API
fetch editor document data
Subscribe
event: room user left

Microservice Design

The microservices are designed to be deployed via Kubernetes. This is the intended method of deployment for production. Deploying via Docker compose is also provided for development purposes.

The architecture of each microservice is broken up into multiple pieces each corresponding to a Docker image. This is done to ensure that each image has a properly defined set of responsibilities, reducing code complexity and allowing for scaling of different pieces independent of each other. It also allows better utilisation of Kubernetes features such as using "deployments" for images that require replicas, and "jobs" for images that are meant to only run once.

Stateful applications such as database and message brokers / message queues (MQ) are deployed externally outside the Kubernetes cluster. They are hosted on cloud services such as ElephantSQL, MongoDB Atlas, Redis Cloud, and CloudAMQP. Although it is possible for us to deploy these services in Kubernetes via StatefulSets, it was determined that it was not worth the risk. Databases and message brokers are critical to our application. It is difficult to set up a robust database and message broker infrastructure with proper periodic backups and scaling. Should we make a mistake, critical user data may be lost. Hence, it is safer to utilise cloud platforms maintained by industry professionals.

Gateway Service

Note that the Kubernetes method of deployment (for production use) utilises Kubernetes Ingress, while the Docker method of deployment (for development use) utilises the "nginx" Docker image. In this sense, the Kubernetes and Docker methods of deployment are independent of each other.

Tools

Kubernetes Ingress Controller: ingress-nginx

When choosing the Ingress Controller, we needed something that is:

- Well-documented so that solving any issues encountered would be easier.
- Supported by a large community so that solving any issues encountered would be easier.
- Actively maintained as unmaintained software is more likely to break and contain security issues.
- Flexible in terms of deployment as we do not want to be tied down to a specific cloud provider.
- Able to block API endpoints by HTTP path as well as HTTP method.

ingress-controller, the official Ingress controller maintained by the Kubernetes development team, meets all these criterias. Out of all the controllers, it seemed to have the largest community support with many forum posts and tutorials. Being maintained by the Kubernetes development team, we can be sure that it would likely continue to be maintained for a long time.

Docker Image: nginx

Since deployment via Docker compose is meant for development use, the choice of Docker image comes down to whichever image can create an environment as close to production as possible. The official NGINX image, "nginx", was the obvious choice.

Proxy Paths

Proxying is done via path prefix. If none of the following prefixes are a match, the request will be forwarded to Webpage Service.

/user-service

Proxy to User Service.

/question-service

Proxy to Question Service.

/matching-service

Proxy to Matching Service.

/room-service

Proxy to Room Service.

The path "/room-service/rooms/" is blocked. A HTTP 403 error will be returned.

/editor-service

Proxy to Editor Service.

/chat-service

Proxy to Chat Service.

/attempt-history-service

Proxy to Attempt History Service.

The path "/attempt-history-service/add/" is blocked. A HTTP 403 error will be returned.

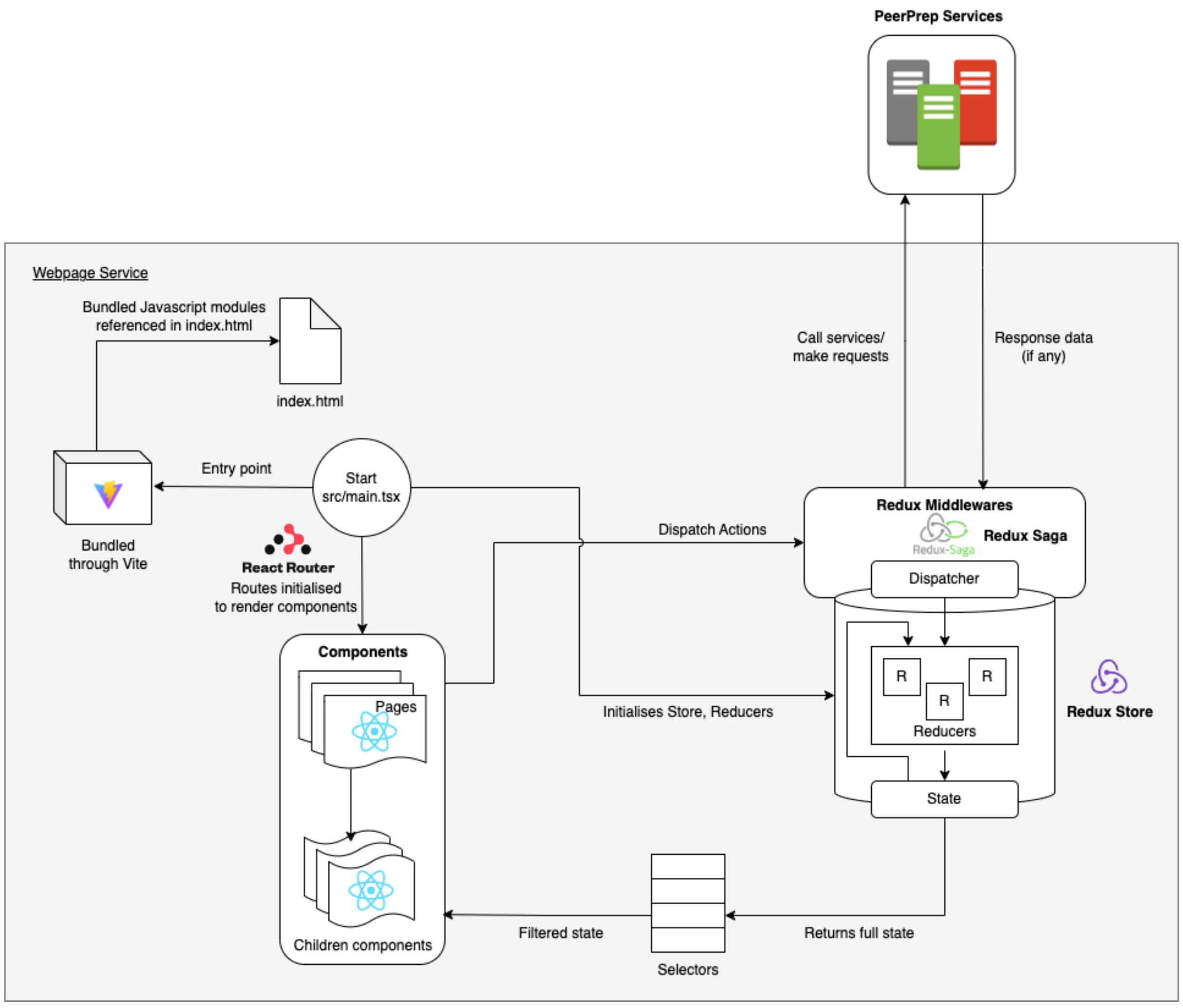
/webpage-service

Proxy to Webpage Service.

Webpage Service

Allows users to interact with PeerPrep.

Architecture



Vite

Vite is a front-end development tool for modern web applications. Vite is responsible for the efficient bundling and transpilation of modular React components during development and production.

When considering the choice of the webpage bundler, we mainly looked at 3 key factors:

- **Performance**: Evaluating bundler's speed, especially during development. A faster bundler can significantly improve the developer experience.
- **Compatibility with React**: React was our chosen front-end framework thus we needed to ensure that the bundler and good compatibility and integration with React.

- **Development Experience**: Checking if a bundler supports Hot Module Replacement (HMR), allowing for real-time updates in the browser without full page reloads during development.

The main candidates were Vite and webpack. While webpack was a popular choice to use with React, we ultimately chose Vite due to its development workflow and performance advantages. Vite's use of esbuild allows for faster module bundling, resulting in quicker build times and a more responsive development server. The built-in support for hot module replacement simplifies the iterative development process, providing real-time updates without the need for full page reloads.

Additionally, Vite's opinionated configuration and seamless integration with React make it well-suited for our project, reducing the overhead of setup and enabling a more straightforward development experience. While webpack remains a robust choice with an extensive ecosystem, Vite's focus on speed, simplicity, and its role as a bundler align well with the specific needs and architecture of the PeerPrep web application.

React

React serves as the core framework for building the PeerPrep Single Page Application (SPA). Its declarative syntax and component-based architecture contribute to a modular and scalable structure, allowing us to design and organise our UI components efficiently.

The decision to choose React was motivated by its robust ecosystem, which includes essential tools such as React Router and React Hooks. This enabled us to implement dynamic client-side routing and manage component logic more effectively. Furthermore, the widespread industry adoption of React, backed by a thriving developer community, ensures ongoing support and a wealth of resources.

Redux Toolkit

Redux Toolkit plays the central role as the state management framework. As shown in the architecture diagram above, Redux efficiently manages the application's global state, providing a predictable and centralised data flow. This is crucial for maintaining a clear separation of concerns with our UI components and facilitates better scalability as our application grows.

The decision to choose Redux is rooted in its ability to streamline complex state management scenarios. It promotes a unidirectional data flow, making it easier to trace and manage state changes across multiple components. Redux’s single immutable state tree enhances predictability, simplifies debugging, and aligns with the principles of our application’s structure.

Furthermore, Redux's extensive ecosystem and compatibility with React through the use of middleware such as react-redux, ensures a seamless integration with our chosen front-end framework, contributing to a cohesive and maintainable architecture.

Redux Saga

With Redux, we were also able to leverage on a powerful middleware called Redux Saga. As showcased in the architecture diagram above, Redux Saga extends the capabilities of Redux by handling side effects and asynchronous actions. This proves instrumental in managing complex workflows, such as asynchronous API calls to PeerPrep services or handling state transitions. Some key examples of how we successfully leveraged Redux Saga was with managing background side effects and API calls during the app initialization phase and while the application was actively running. Additionally, Redux Saga also proved vital in handling WebSocket events.

The addition of Redux Saga aligns seamlessly with our architectural goals, providing a structured and maintainable solution for handling asynchronous logic in the PeerPrep web application. The combination of Redux and middleware such as Redux Saga enhances the application's state management, contributing to a robust and scalable architecture.

User Service

Manages user information. Provides endpoints for CRUD operations with regards to user information.

User Service offers 2 forms of user authentication. When a user is logged in, a user session is created and a unique session token generated. An access token is also generated. Both the session token and access tokens are sent in the HTTP response as cookies. When the front-end makes a request to any PeerPrep service, both the access token and session token are sent. Services can choose to authenticate users via the access token or the session token.

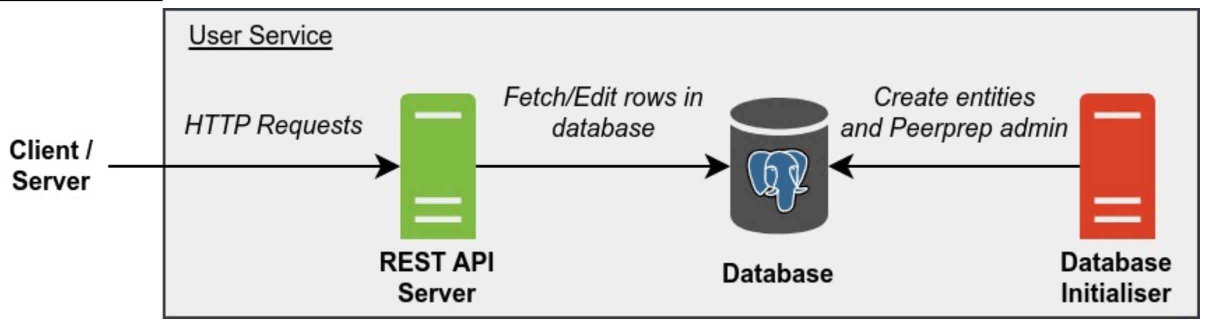
To verify via access token, the service must contact the User Service once in its lifetime to retrieve the needed public key. This puts less strain on the User Service but comes at the cost of security as information in the access token may be out of date (e.g. user role has been changed to user but is still admin according to the access token). Should a session be hijacked, the access token cannot be invalidated immediately. One must wait for it to expire.

To verify via session token, the service must contact the User Service for each verification. The service sends the session token to the User Service and retrieves the user information. This puts more strain on the User Service but is much more secure as the service can be sure that the user information is up to date and a system admin can invalidate a user session by removing it from the database.

The choice of verification is dependent on the operation. In general, session token authentication is use for high risk operations (e.g. editing a question, changing a password) while access token authentication is use for everything else.

The session token also doubles as a refresh token for refreshing the access token when it expires.

Architecture



REST API Server

The REST API server handles REST API requests. This server is stateless allowing it to easily scale up and down horizontally based on the number of REST API requests.

Database Initialiser

Initialises the database by creating the necessary entities and a default PeerPrep admin account whose username is "admin". This is a job that runs once. Upon completion, it shuts down.

If the database already contains one or more entities that the initialiser intends to create, the initialisation is aborted to prevent potential loss of data. This behaviour can be changed via an environment variable.

Although the database initialiser could be combined with the REST API server such that the initialisation runs every time a REST API server is created, that would give the REST API server multiple responsibilities which is not ideal.

Database

User information such as username, email, password hash, and user login sessions are stored in this database.

When considering the choice of database, we take into account:

- Schema is fixed.
- One-to-many relationships will exist as a user can have multiple user login sessions.
- Query operations would likely far outnumber the insert operations.

PostgreSQL and MongoDB were both valid candidates for this database. In terms of speed, one can be faster than the other depending on the database operation. MongoDB would be faster with insertions while PostgreSQL would be faster for querying when JOIN operations are not used. A query using PostgreSQL JOIN operation would be slower than a query using MongoDB without JOIN operation. The User Service database will have to perform a mix of different queries, some requiring and some not requiring JOIN operations. Hence, one is not superior to the other in terms of speed.

Ultimately, PostgreSQL was chosen due to its ability to enforce a strict schema and its adherence to the ACID principles. These properties make PostgreSQL easier to work with as it helps to make bugs in the code much easier to detect.

It is worth noting that the REST API server and database initialiser interact with the database via TypeORM which supports multiple different databases. The code is also structured such that all database operations are abstracted behind a DatabaseClient interface. This makes it easy to change to another database should the need arise.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_user_service_api
Description: Runs the REST API.
Environment Variables:
- `DATABASE_USER` - User on the database host.
- `DATABASE_PASSWORD` - Password of the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `HASH_COST` - Cost factor of the password hashing algorithm.
- `ACCESS_TOKEN_PRIVATE_KEY` - Private key for signing access tokens.
- `ACCESS_TOKEN_PUBLIC_KEY` - Public key for verifying access tokens.
- `SESSION_EXPIRE_MILLIS` - Number of milliseconds a user session can live for since the last expiry date and time extension.
- `ACCESS_TOKEN_EXPIRE_MILLIS` - Number of milliseconds an access token can live for.
- `PORT` - Port to listen on.
- `NODE_ENV` - Mode the app is running on ("development" or "production"). "development" mode enables features such as CORS for "localhost".

Database Initialiser

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_user_service_database_initialiser
Description: Initialises the database by creating the necessary entities and a default Peerprep admin account.
Environment Variables:
- `DATABASE_USER` - User on the database host.
- `DATABASE_PASSWORD` - Password of the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `HASH_COST` - Cost factor of the password hashing algorithm.
- `ADMIN_EMAIL_ADDRESS` - Email address of the default PeerPrep admin user.
- `ADMIN_PASSWORD` - Password of the default PeerPrep admin user.
- `SHOULD_FORCE_INITIALISATION` - Should database initialisation be done regardless of whether one or more entities to be created already exist. Set to "true" to enable (may cause data loss).

REST API

Create a User

[POST] `/user-service/users`

Creates a new user.

Query Parameters

- `username` - Username.
- `email-address` - Email address.
- `password` - Password.

Response

- `201` - User created.
- `400` - One or more query parameters are invalid. The reason for the error is provided in the response body.
- `500` - Unexpected error occurred on the server.

Create a Session

[POST] `/user-service/sessions`

Creates a new user session.

Query Parameters

- `username` - Username.
- `password` - Password.

Response

- `201` - Session created.
  - Cookies:
    - `session-token` - Contains session token. This cookie is HTTP-only.
    - `access-token` - Contains access token. This cookie is HTTP-only.
    - `access-token-expiry` - Contains expiry of the access token.
- `400` - One or more query parameters are invalid. The reason for the error is provided in the response body.
- `401` - Username is not in use or the username and password do not match.
- `500` - Unexpected error occurred on the server.

Get an Access Token

[GET] `/user-service/session/access-token`

Gets an access token for the user who owns the specified session token.

A successful request to this endpoint will also extend the expiry of the session whose session token was specified.

Cookies

- `session-token` - Session token.

Response

- `200` - Success.
    - Cookies:
        - `session-token` - Contains session token. This cookie is HTTP-only. This is sent to extend the lifespan of the cookie on the browser.
        - `access-token` - Contains access token. This cookie is HTTP-only.
        - `access-token-expiry` - Contains expiry of the access token.
- `401` - Session token was not provided or is invalid.
- `500` - Unexpected error occurred on the server.

Delete a Session

[DELETE] `/user-service/session`

Deletes the session whose session token is the one specified.

Cookies

- `session-token` - Session token.

Response

- `200` - Success.
    - Cookies:
        - `session-token` - Expired cookie.
        - `access-token` - Expired cookie.
        - `access-token-expiry` - Expired cookie.
- `401` - Session token was not provided or is invalid.
- `500` - Unexpected error occurred on the server.

Get Usernames

[GET] `/user-service/users/all/username`

Gets the usernames of all users whose user ID is specified.

Query Parameters

- `user-ids` - Array of user IDs in JSON format. (e.g. `[7,11,404]`)

Response

- `200` - Success. User IDs and their corresponding usernames are stored as a JSON string in the response body. User IDs which do not belong to any user would not be included in the response.
- `400` - One or more query parameters are invalid. The reason for the error is provided in the response body.
- `500` - Unexpected error occurred on the server.

Get a User Profile

[GET] `/user-service/user/profile`

Gets the profile of the user who owns the specified access token.

Cookies
- `access-token` - Access token.

Response
- `200` - Success. User's profile information is stored as a JSON string in the response body.
- `401` - Access token was not provided or is invalid.
- `500` - Unexpected error occurred on the server.

Update a User Profile

[PUT] `/user-service/user/profile`

Updates the profile of the user who owns the specified session token.

Query Parameters
- `username` - Updated username.
- `email-address` - Updated email address.

Cookies
- `session-token` - Session token.

Response
- `200` - Success.
    - Cookies:
        - `access-token` - Contains access token. This cookie is HTTP-only.
        - `access-token-expiry` - Contains expiry of the access token.
- `400` - One or more query parameters are invalid. The reason for the error is provided in the response body.
- `401` - Access token was not provided or is invalid.
- `500` - Unexpected error occurred on the server.

Update a User's Password

[PUT] `/user-service/user/password`

Updates the password of the user who owns the specified session token.

Query Parameters
- `password` - Password for verifying the user.
- `new-password` - New password to update to.

Cookies
- `session-token` - Session token.

Response
- `200` - Success.
- `400` - New password is not a valid password. The reason for the invalidity is provided in the response body.
- `401` - Session token was not provided, or session token is invalid, or password is incorrect.
- `500` - Unexpected error occurred on the server.

Update a User's Role

[PUT] `/user-service/users/:user-id/user-role`

Updates the user role of a user. The user making the request must have the "admin" user role.

Path Parameters
- `user-id` - User ID of the user whose user role is to be updated.

Query Parameters
- `user-role` - Updated user role.

Cookies
- `session-token` - Session token of the user making the request.

Response
- `200` - Success.
- `400` - One or more query parameters are invalid. The reason for the error is provided in the response body.
- `401` - Session token was not provided, or is invalid, or does not belong to a user with the "admin" user role.
- `404` - User ID does not belong to any existing user.
- `500` - Unexpected error occurred on the server.

Delete a User

[DELETE] `/user-service/user`

Deletes the user who owns the specified session token.

Query Parameters
- `password` - Password for verifying the user.

Cookies
- `session-token` - Session token.

Response

- `200` - Success.
  - Cookies:
    - `session-token` - Expired cookie.
    - `access-token` - Expired cookie.
    - `access-token-expiry` - Expired cookie.
- `401` - Session token was not provided, or session token is invalid, or password is incorrect.
- `500` - Unexpected error occurred on the server.

Get Access Token Public Key

[GET] `/user-service/access-token-public-key`

Gets the public key for verifying access tokens.

Response

- `200` - Success. The response body will contain the public key.
- `500` - Unexpected error occurred on the server.

Get a User Identity

[GET] `/user-service/user/identity`

Gets the user ID and user role of the user who owns the specified session token.

This is only intended to be used by other services when performing high threat model operations, where trusting the information stored in a valid access token is risky.

Query Parameters

- `session-token` - Session token.

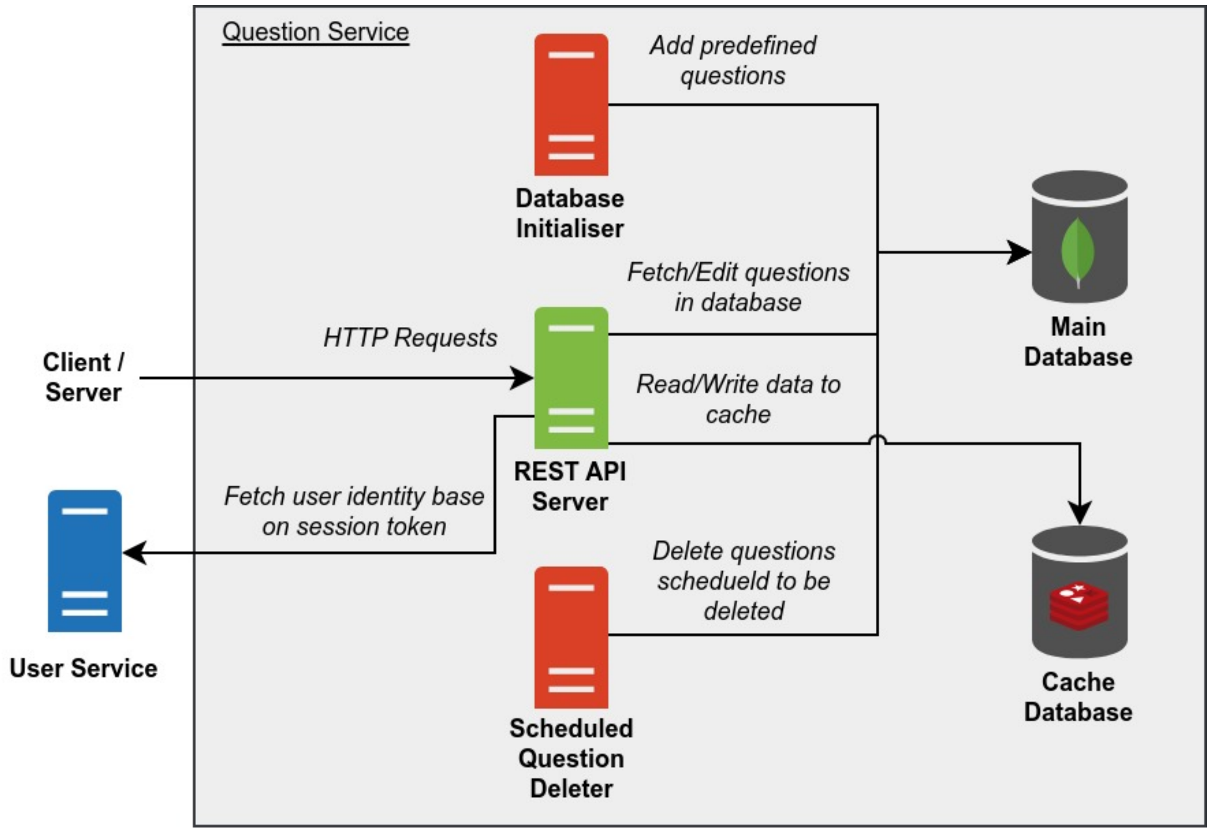
Response

- `200` - Success. User ID and user role are stored in the response body.
  - `user-role` can have the value `admin`, `maintainer`, or `user`
- `401` - Session token was not provided or is invalid.
- `500` - Unexpected error occurred on the server.

Question Service

Manages questions. Provides endpoints for CRUD operations with regards to question information.

Architecture



REST API Server

The REST API server handles REST API requests. This server is stateless allowing it to easily scale up and down horizontally based on the number of REST API requests.

Database Initialiser

Initialises the database by populating it with a predefined list of questions. This is a job that runs once. Upon completion, it shuts down.

If the database already contains one or more questions, the initialisation is aborted.

Scheduled Question Deleter

Periodically deletes all questions that are scheduled to be deleted. This is deployed as a CronJob via Kubernetes.

This container was introduced to allow soft deletion of questions. This was done to avoid returning unexpected errors in a scenario where two users are matched on a question, but before they are able to join the room the question is deleted.

Main Database

Database for storing question information.

MongoDB was chosen due to the fact that as a NoSQL database, it utilises a dynamic schema. This gave the team flexibility to easily add additional metadata to questions stored in the database such as code templates. Additionally, the Question Service lacked a need for relational relations with other tables, a primary advantage of Relational Databases.

Cache Database

Database for caching question information.

Each question can contain a large amount of information. Having to fetch this information from the main database every time it is needed is very slow. To speed up fetch requests, a Redis database was implemented to be used as a cache. The database exploits temporal locality by utilising the cache-aside approach. When question information is to be retrieved, the cache database is checked first. Only upon a cache miss, is the main database checked, after which the cache database is updated to store the fetched question.

Additionally, the database cache also helps to reduce the need for repeated expensive queries from the database, replacing them with in memory retrieval. In the case of questions service, the endpoint to retrieve distinct categories and languages is called by multiple features. The associated database query for the endpoint is expensive. By placing the result of the query in the cache, we can relieve pressure on the database and reduce overall latency in the application.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_question_service_api
Description: Runs the REST API.
Environment Variables:
- `MONGO_URI` - URI for connecting to the Mongo database (the main database).
- `REDIS_USERNAME` - Username for the Redis database (the caching database).
- `REDIS_PASSWORD` - Password for the Redis database (the caching database).
- `REDIS_HOST` - Address of the Redis database host (the caching database).
- `REDIS_PORT` - Port the Redis database (the caching database) is listening on.
- `REDIS_SHOULD_USE_TLS` - Should the Redis database connection be secured with TLS. Set to "true" to enable.
- `USER_SERVICE_HOST` - Address and port of the User Service host in the format `ADDRESS:PORT`.

- `API_PORT` - Port to listen on.
- `NODE_ENV` - Mode the app is running on ("development" or "production").

Database Initialiser

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_question_service_database_initialiser
Description: Initialises the database by creating the predefined questions (if the database is empty).
Environment Variables: Same as API.

Scheduled Question Deleter

Name:
ghcr.io/cs3219-ay2324s1-g04/peerprep_question_service_scheduled_question_deleter
Description: Deletes questions from the database that are scheduled to be deleted. This image is intended to be run once every day.
Environment Variables: Same as API.

REST API

Retrieve All Questions

[GET] `/question-service/questions`

Retrieves all questions in the database.

Parameters

- `complexity` - The complexity of the question (Optional)
- `categories` - The categories of the question (Optional) - Can be multiple
- `languages` - The langSlug of the question (Optional)
- `limit` - The max number of questions in response (Optional)
- `offset` - The offset for the first question in response (Optional)

Response

- `200` - Success.
    - Returns questions as an Array of JSON objects
    - Each question has the following fields
        - id
        - title
        - complexity
        - description
- `500` - Error
    - Unexpected error occurred on the server.

Retrieve Question by ID

[GET] `/question-service/questions/:id`

Retrieves a question by its ID.

Response

- `200` - Success.
    - Returns question with matching ID as JSON Object with the following fields:
        - id
        - title
        - description
        - complexity
        - categories
        - template
            - language
            - langSlug
            - code
- `404` - Error
    - Question ID not found in Database.
- `500` - Error
    - Unexpected error occurred on the server.

Retrieve Question by Params

[GET] `/question-service/question-matching/question`

Retrieves a random question by matching params.

Parameters

- `complexity` - The complexity of the question (Required)
- `categories` - The categories of the question (Optional) - Can be multiple
- `language` - The langSlug of the question (Required)

Response

- `200` - Success.
    - Returns question with matching filters as JSON Object with the following fields:
        - id
        - title
- `400` - Error
    - URL is missing complexity parameter.
- `500` - Error
    - Unexpected error occurred on the server.

Update Question (Privileged users only)

[PUT] `/question-service/questions/:id`

Updates a question in the database by ID. Only users of user role admin or maintainer can add questions.

The data for the new question is passed in the request body with the following fields:

- title - The title of the question
- description - The question description
- complexity - The difficulty level of the question
- categories - The different topics applicable to the question
- template - The template code for the question
    - language - The language of the template code
    - langSlug - The language slug of the template code
    - code - The template code

Cookies
- `session_token` - Session Token

Response
- `201` - Success.
    - Question is successfully added to the database.
    - The data field in the response contains the newly added question.
- `403` - Error
    - The user does not have authorization to update questions in the database.
- `404` - Error
    - The question could not be found in the database.
- `407` - Error
    - The client passed an invalid or missing session token.
- `500` - Error
    - Unexpected error occurred on the server.

Add Question (Privileged users only)

[POST] `/question-service/questions`

Adds a question to the database. Only users of user role admin or maintainer can add questions.

The data for the new question is passed in the request body with the following fields:
- title - The title of the question
- description - The question description
- complexity - The difficulty level of the question
- categories - The different topics applicable to the question
- template - The template code for the question
    - language - The language of the template code
    - langSlug - The language slug of the template code
    - code - The template code

Cookies
- `session_token` - Session Token

Response
- `201` - Success.
    - Question is successfully added to the database
    - The data field in the response contains the newly added question

- `403` - Error
  - The user does not have authorization to add questions to the database.
- `407` - Error
  - The client passed an invalid or missing session token.
- `500` - Error
  - Unexpected error occurred on the server.

Delete Question (Privileged users only)

[DELETE] `/question-service/questions/:id`

Deletes a question in the database by ID. Only users of user role admin or maintainer can delete questions.

Cookies
- `session_token` - Session Token

Response
- `201` - Success.
  - Question is successfully deleted from the database.
- `403` - Error
  - The user does not have authorization to delete questions from the database.
- `404` - Error
  - The question could not be found in the database.
- `407` - Error
  - The client passed an invalid or missing session token.
- `500` - Error
  - Unexpected error occurred on the server.

Get Categories

[GET] `/question-service/categories`

Retrieves all unique categories in the database.

Response
- `200` - Success.
  - Returns categories as an Array of strings
- `500` - Error
  - Unexpected error occurred on the server.

Get Languages

[GET] `/question-service/languages`

Retrieves all unique languages in the database.

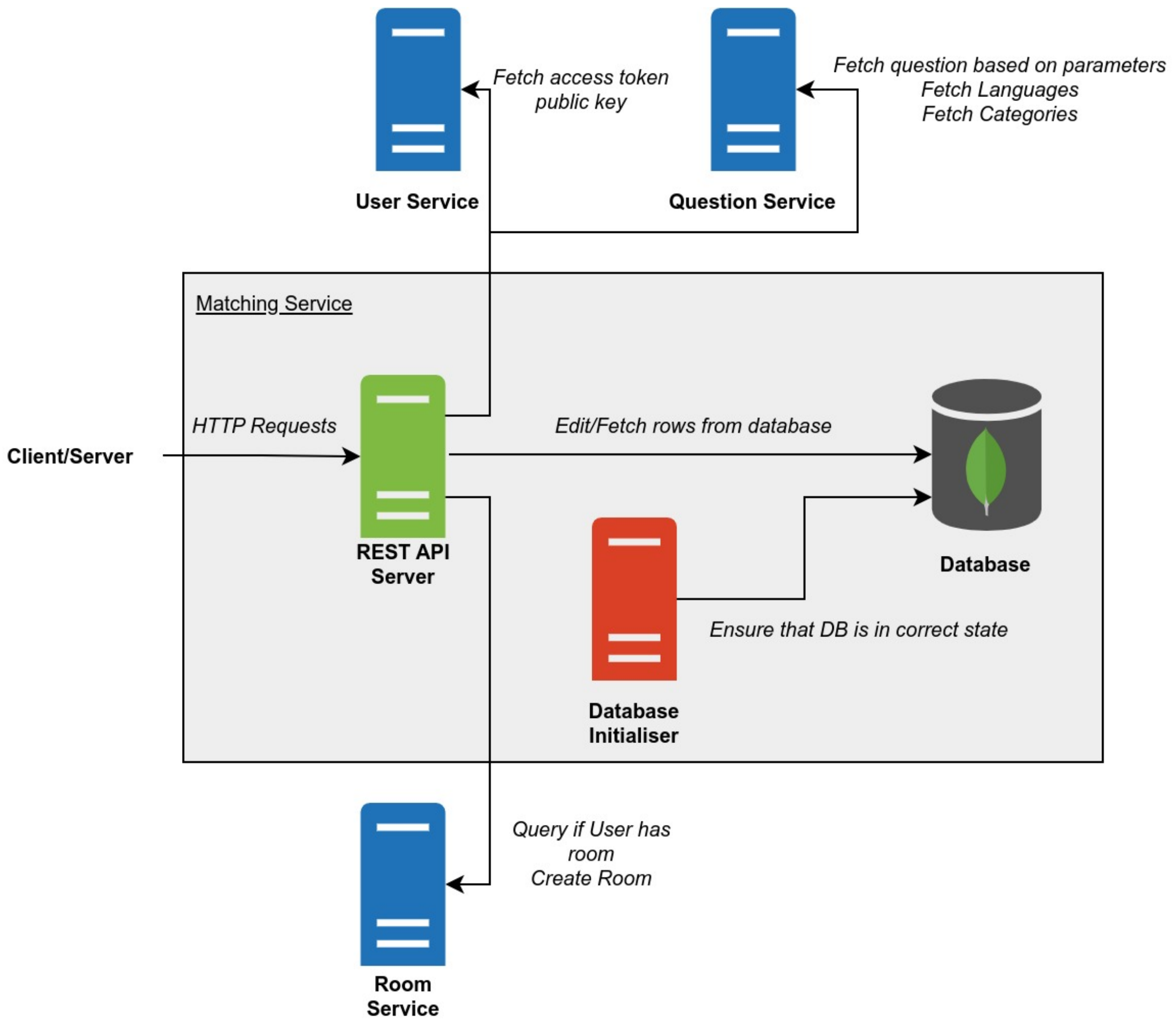
Response
- `200` - Success.

    - Returns categories as an Array of objects with the fields language and langSlug
- `500` - Error
    - Unexpected error occurred on the server.

Matching Service

Handles the queuing and matching of users.

Architecture



Main Database

MongoDB was primarily chosen due to the number of inserts that are done by the API server. In addition, at the point of development, MongoDB was chosen due to the time to live (TTL) functionality which was not present in DBMS such as Postgresql or Mysql. This created a scenario where there was no need to create a new server to listen and perform delete, however this prevents us from exploiting libraries such as TypeORM to allow switching to a different database.

Database Initialiser

Initialises the database by ensuring that the time to live index is set up. This is a job that runs once. Upon completion, it shuts down.

REST API Server

This service was originally written with room service included, but was split apart later on, and hence message queues are not utilised in this portion. For a non first come first serve system, the developers were not able to find a solution to utilise message queues as a queue itself (as described by our stakeholders). Creating publishers and consumers for easy, medium, hard queues is trivial, but handling cases where the user's preferences are different, meant that there is a need to store it somewhere outside of the queue. In addition, matching needed to be atomic, that is to say the match either happens or not at all.

However, since the only other service that needed to listen to matching service's messages was room service, and that essentially the team was treating matching and room service to be a tightly coupled system (e.g. the combining the two actually made up one service). It was important for us to allow the user to be in a room or a queue but not both at the same time, and hence we utilised synchronous communication between the two services.

One of the considerations for getting all questions and categories from the question service was to perform validation of the user's inputs. Since the question service will come equipped with the ability to source questions from third party sources, storing a hard copy of the categories would not work and would require constant checking if the copy is up to date.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_matching_service_api
Description: Runs the REST API.
Environment Variables:

- `NODE_ENV` - Mode the app is running on ("development" or "production").
- `MS_EXPRESS_PORT` - Port to listen on.
- `MS_MONGO_URI` - URI for connecting to the Mongo database.
  - Example `mongodb://<user>:<pass>@<address>:<port>/<database>`
- `MS_MONGO_COLLECTION` - Name of the Mongo collection.
- `QUEUE_EXPIRY` - Number of milliseconds a user's matching request will remain in the queue before timing out. Due to this using Mongo's auto delete, collection may take up to one additional minute to get deleted.
- `SERVICE_USER_HOST` - Address of the User Service host.
- `SERVICE_USER_PORT` - Port the User Service is listening on.
- `SERVICE_QUESTION_HOST` - Address of the Question Service host.
- `SERVICE_QUESTION_PORT` - Port the Question Service is listening on.
- `SERVICE_ROOM_HOST` - Address of the Room Service host.
- `SERVICE_ROOM_PORT` - Port the Room Service is listening on.

Database Initialiser

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_matching_service_database_initialiser
Description: Initialises the database by creating and setting up the necessary collections.

Environment Variables:

- `MS_MONGO_URI` - URI for connecting to the Mongo database.
    - Example mongodb://<user>:<pass>@<address>:<port>/<database>
- `MS_MONGO_COLLECTION` - Name of the Mongo collection.
- `QUEUE_EXPIRY` - Number of milliseconds a user's matching request will remain in the queue before timing out. Due to this using Mongo's auto delete, collection may take up to one additional minute to get deleted.

REST API

Check if a User is in the Queue

[GET] `/matching-service/queue/`

Checks if the user, who owns the specified access token, is in the queue.

Cookies

- `access-token` - Access token.

Returns

- `200` - `{ message: "In queue" }`
- `303` - `{ message: "In room" }`
- `401` - `{ message: "Not authorized" }`
- `404` - `{ message: "Not in queue", data : { difficulty : string[], categories : string[], language : string[] } }`
- `500` - `{ message: "Sever Error" }`

Add a User to the Queue

[POST] `/matching-service/queue/join`
Adds the user and their matching criterias to the queue. Malformed parameters will be randomised.

Cookies

- `access_token` - Access token.

Parameters

- `difficulty` - The complexity of the question.
- `categories[]` - The categories of the question - Can be multiple
- `language` - The programming language of the question

Returns

- `200` - `{ message: "Joined queue" }.`
- `303` - `{ message: "In room" }`
- `401` - `{ message: "Not authorized" }`
- `409` - `{ message: "Already in queue" }`
- `500` - `{ message: "Sever Error" }`

Remove a User from the Queue

Removes the user, who owns the specified access token, from the queue.

[DELETE] `/matching-service/queue/`

Cookies

- `access-token` - Access token.

Returns

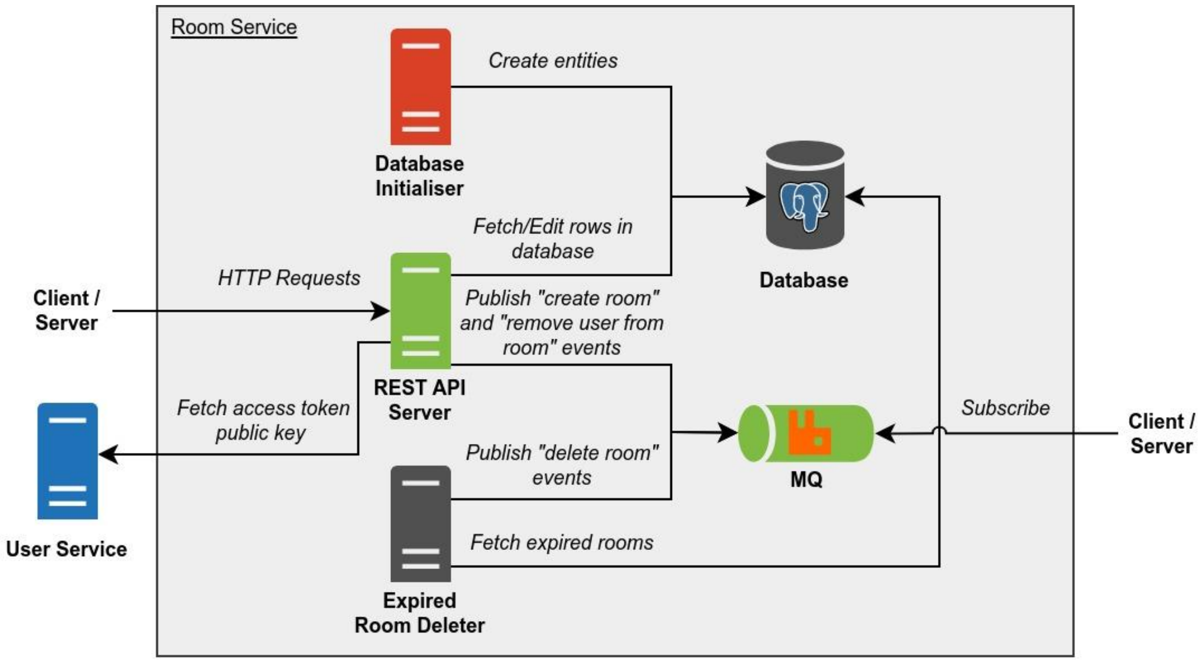
- `200` - `{ message: "Received message" }`.
- `401` - `{ message: "Not authorized" }`

Room Service

Manages rooms. Provides endpoints for CRUD operations with regards to user information.

Note that Room Service does not match users together. Said task is handled by Matching Service.

Architecture



REST API Server

The REST API server handles REST API requests. This server is stateless allowing it to easily scale up and down horizontally based on the number of REST API requests.

Database Initializer

Initialises the database by creating the necessary entries. This is a job that runs once. Upon completion, it shuts down.

Expired Room Deleter

Check the database every few minutes to delete entries that are expired, and publishes the delete event type to the message queue.

Database

Database for storing room information.

Each room has a fixed set of information which can be represented as a single row in a table. Considering that schema flexibility is not needed, read operations far outnumber write operations, and all query can be handled without the use of any JOIN operations, PostgreSQL would perform much better in terms of speed as well as reducing the chance of bugs in the code due to the adherence to the ACID principles.

Message Queue

MQ for publishing room events which other services can subscribe to be notified of said events.

As multiple services (docs service, chat service, and history attempt service) needs to be notified of changes in a room, an MQ is a good fit to decouple the interactions between these services. This also allows us to introduce new services in the future which also rely on room events by just having them subscribe to the MQ. There is no need to modify Room Service in this case.

As for choosing RabbitMQ over other brokers such as Apache Kafka, it was primarily due to the simplicity of RabbitMQ making it easier to setup and deploy. Kafka does provide some advantages such as being faster at transferring large amount of data, and being able to scale better. However, in the case of this MQ, we do not expect such features to produce a strong enough benefit to justify the additional time that would have to be spent.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_room_service_api
Description: Runs the REST API.
Environment Variables:

- `DATABASE_USER` - User on the database host.
- `DATABASE_PASSWORD` - Password of the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `MQ_USER` - User on the MQ host.
- `MQ_PASSWORD` - Password of the MQ.
- `MQ_HOST` - Address of the MQ host.
- `MQ_PORT` - Port the MQ is listening on.
- `MQ_VHOST` - Vhost of the MQ.
- `MQ_SHOULD_USE_TLS` - Should MQ connection be secured with TLS. Set to "true" to enable.
- `MQ_EXCHANGE_NAME` - Name of the MQ exchange.
- `USER_SERVICE_HOST` - Address of the User Service host.
- `USER_SERVICE_PORT` - Port the User Service is listening on.

- `ROOM_EXPIRE_MILLIS` - Number of milliseconds a room can live for since the last expiry date and time extension.
- `PORT` - Port to listen on.
- `NODE_ENV` - Mode the app is running on ("development" or "production"). "development" mode enables features such as CORS for "localhost".

Database Initialiser

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_room_service_database_initialiser
Description: Initialises the database by creating the necessary entities.
Environment Variables:

- `DATABASE_USER` - User on the database host.
- `DATABASE_PASSWORD` - Password of the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `SHOULD_FORCE_INITIALISATION` - Should database initialisation be done regardless of whether one or more entities to be created already exist. Set to "true" to enable (may cause data loss).

Expired Room Deleter

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_room_service_expired_room_deleter
Description: Periodically deletes expired rooms from the database and publishes the appropriate event on the MQ.
Environment Variables:

- `DATABASE_USER` - User on the database host.
- `DATABASE_PASSWORD` - Password of the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `MQ_USER` - User on the MQ host.
- `MQ_PASSWORD` - Password of the MQ.
- `MQ_HOST` - Address of the MQ host.
- `MQ_PORT` - Port the MQ is listening on.
- `MQ_VHOST` - Vhost of the MQ.
- `MQ_SHOULD_USE_TLS` - Should MQ connection be secured with TLS. Set to "true" to enable.
- `MQ_EXCHANGE_NAME` - Name of the MQ exchange.

- `ROOM_DELETION_INTERVAL_MILLIS` - Number of milliseconds between database searches for expired rooms.

REST API

Create a Room

[POST] `/room-service/rooms`

Creates a new room.

WARNING: This endpoint is potentially abusable. It should be inaccessible by the front-end.

Body
The body must be of JSON format, containing the following fields:
- `user-ids` - Array of user IDs which corresponds to the users in the room.
- `question-id` - ID of the question assigned to the room.
- `question-lang-slug` - Language slug of the programming language chosen for the room.

Response
- **201** - Room created. The response body contains the ID of the room.
- **400** - Body is not a valid JSON object or one or more parameters are invalid. The reason for the error is provided in the response body.
- **500** - Unexpected error occurred on the server.

Get a Room by Room ID

[GET] `/room-service/rooms/:room-id`

Gets information about the room whose room ID was specified.

Path Parameters
- `room-id` - Room ID.

Response
- `200` - Success. The response will contain information about the room.
- `400` - One or more path parameters are invalid. The reason for the error is provided in the response body.
- `404` - No room was found that has the specified room ID.
- `500` - Unexpected error occurred on the server.

Get a Room by User ID

[GET] `/room-service/room`
Gets information about the room which contains the user who owns the specified access token.

Cookies

- `access-token` - Access token.

Response

- `200` - Success. The response will contain information about the room.
- `401` - Access token was not provided or is invalid.
- `404` - User who owns the access token does not belong in any room.
- `500` - Unexpected error occurred on the server.

Extend the Lifespan of a Room

[PATCH] `/room-service/room/keep-alive`

Extends the lifespan of the room which contains the user who owns the specified access token.

Cookies

- `access-token` - Access token.

Response

- `200` - Success. The response will contain the updated expiry of the room.
- `401` - Access token was not provided or is invalid.
- `404` - User who owns the access token does not belong in any room.
- `500` - Unexpected error occurred on the server.

Remove a User from a Room

[DELETE] `/room-service/room/user`

Removes the user who owns the specified access token from the room said user is in.

Cookies

- `access-token` - Access token.

Response

- `200` - Success.
- `401` - Access token was not provided or is invalid.
- `404` - User who owns the access token does not belong in any room.
- `500` - Unexpected error occurred on the server.

Delete a Room (Development mode only)

[DELETE] `/room-service/rooms/:room-id`

Deletes the room whose room ID was specified.

Path Parameters

- `room-id` - ID of the room.

Response

- `200` - Success.
- `400` - One or more path parameters are invalid. The reason for the error is provided in the response body.
- `404` - No room was found that has the specified room ID.
- `500` - Unexpected error occurred on the server.

Message Queue Events

Create Room

Event Type '`create`' is published when a room is created, containing information about the room that was created.

Delete Room

Event Type '`delete`' is published when a room is deleted, containing information about the room that was deleted.

Remove User from Room

Event Type '`remove-user`' is published when a user leaves a room, containing information about the updated room as well as the user ID of the user that was removed.

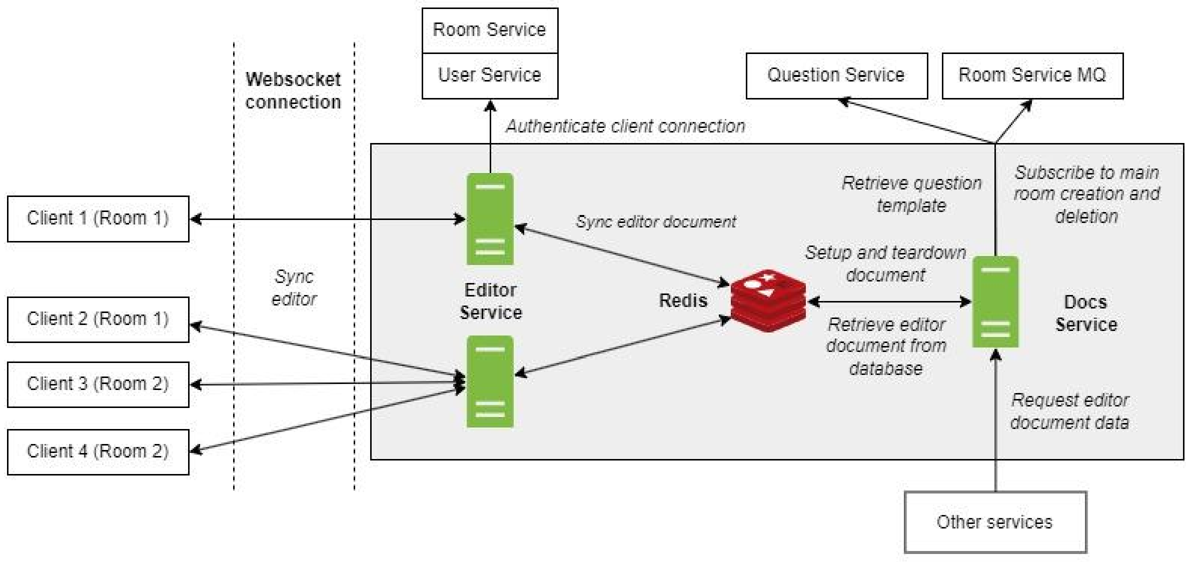
Editor Service

The Editor Service provides the backend mechanism for real-time collaboration (e.g. concurrent code editing) between the authenticated and matched users in the collaborative space.

- Providing document synchronisation that allows users who belong to the same room to view and edit the same code using the editor in real-time.
- Authenticates users based on their access to a given room.

Architecture

Editor-Docs Service Architecture Diagram



The collaborative backend ecosystem comprises of 3 key components:

1. Editor Service instances that establish and authenticate WebSocket connections with clients.
2. A Redis instance that primarily serves as a message broker between Editor Service instances.
3. Docs Service instances serve as mediators between other PeerPrep services and Editor Service instances.

Solving Real-time Collaboration

We define real-time collaboration as the simultaneous live-editing of the same code by different users on different devices.

Real-time collaboration is **NOT**
- Locking the document such that only one user can edit at any given time.
- Updating the document ONLY when the user refreshes the page.
- Freezing the document to resolve merge conflicts.

Hence, if two users Alice and Bob are using two separate devices but belong to the same room, they must see any changes made to the document in real-time. Both users must also see identical states of the document after these changes. i.e. If Bob sees "lorem ipsum", Alice must see "lorem ipsum" and not "lor ipsumem".

However, due to network latency, users making concurrent changes to the document can result in conflicting states of the document. Hence, we cannot simply propagate the user's raw input events through the network and expect both users to always see identical states of the document.

To manage such conflicts, we represent editor documents as conflict-free replicated data types (CRDTs). A CRDT represents document changes as commutative operations. Hence, a user would eventually be able to reconstruct an identical state of the document after receiving all updates regardless of the order they are received.

To implement CRDTs, we adopted the Yjs framework. Yjs is a relatively mature and battle-tested library that has been applied in many collaborative applications. It provides a synchronisation protocol for exchanging updates, and providers that simplify communication between clients.

Facilitating Client Communication

Here is a simple scenario to demonstrate how document changes are propagated to clients through Editor Service instances.

Consider two clients (C1, C2) who belong to the same room (R1) and two Editor Service instances (ES1, ES2). The editor document of R1 is blank.

1. C1 authenticates and establishes a WebSocket connection with ES1.
2. ES1 binds to room R1 and begins listening for R1 updates from Redis.
3. C1 user types "Hello World :3" in the editor.
4. The Yjs updates that comprise "Hello World :3" are then sent to ES1 over the WebSocket connection.

5. ES1 updates its document state and pushes the Yjs updates to Redis.
6. Redis now contains the list of Yjs updates for R1.
7. C2 authenticates and establishes a WebSocket connection with ES2.
8. ES2 binds to room R1 and begins listening for R1 updates from Redis.
9. ES2 pulls the missing R1 updates from Redis and updates its document state.
10. ES2 then pushes these updates to C2 over the WebSocket connection.
11. C2 now has an editor document with the text "Hello World :3".

Scalability

This mechanism of propagating updates between Editor Service instances allows Editor Service instances to **scale horizontally** as two users in the same room do not need to connect to the same Editor Service instance to communicate.

Consistency

As mentioned in Solving Real-time Collaboration, Using CRDTs eliminates the need to worry about **conflict management** between document states as long as we ensure that updates eventually propagate to their recipients.

Efficiency

Updates will be frequently generated by the user as they edit the document. Using WebSockets to maintain a **persistent connection** would be efficient as it would eliminate the need to establish a new TCP connection and handshake for every single update (potentially every keystroke).

Redis

We want to **minimise latency** between clients to ensure a smooth collaborative editing experience between multiple users. Hence, we chose Redis as the message broker for its low-latency read and writes. Since Redis stores a list of Yjs updates for each room for propagation, it also persists document data. The entire document can be reconstructed from this list of Yjs updates. Hence, Redis is also used as a primary database.

Overall, our solution for synchronising editor documents allows for scalability, high availability, partition tolerance as well as the eventual consistency of document states.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_editor_service_api
Description: Runs the Editor Service container.
Environmental Variables:

- `EDITOR_SERVICE_PORT` - The port used to bind the WebSocket server.
- `REDIS_USERNAME` - Username for the Redis database.
- `REDIS_PASSWORD` - Password for the Redis database.
- `REDIS_HOST` - Address of the Redis database host.
- `REDIS_PORT` - Port of the Redis database.
- `REDIS_DB` - Database index used for Redis database.

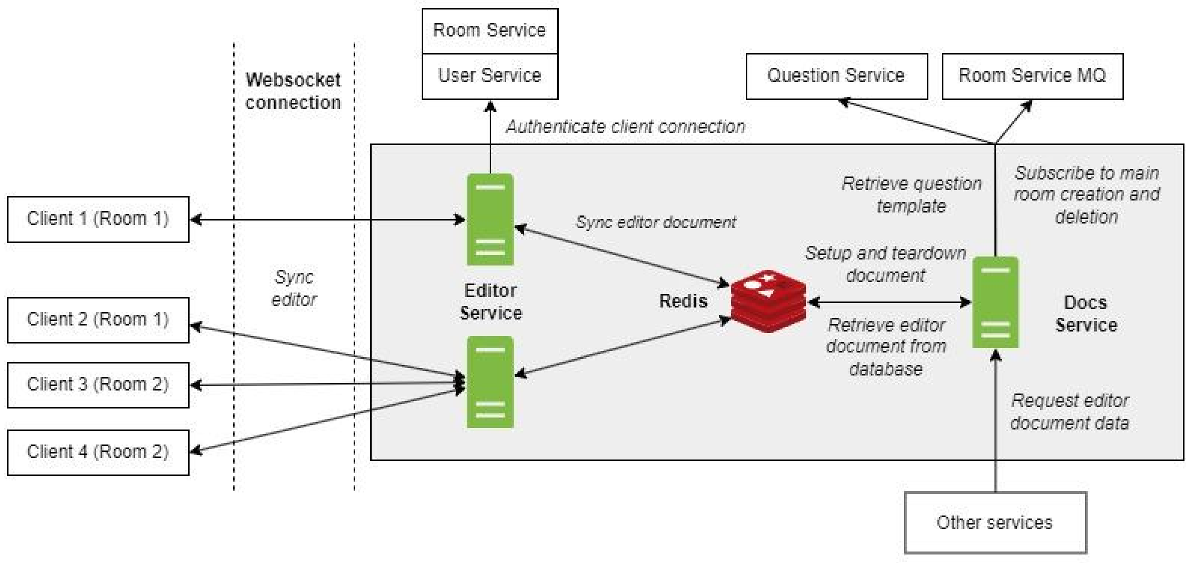
Docs Service

While Editor Service communicates directly with clients to facilitate the synchronisation of editor documents for real-time collaboration, Docs Service performs several key functions to support Editor Service.

- **Document Setup:** Insertion of question code template on room creation.
- **Document Teardown:** Propagates the room deletion event to Editor Service instances through Redis and clears the document data from Redis.
- **Document Retrieval:** Provides a REST API endpoint for other PeerPrep services to retrieve document data.

Architecture

Editor-Docs Service Architecture Diagram



The collaborative backend ecosystem comprises of 3 key components:

4. Editor Service instances that establish and authenticate WebSocket connections with clients.
5. A Redis instance that primarily serves as a message broker between Editor Service instances.
6. Docs Service instances serve as mediators between other PeerPrep services and Editor Service instances.

Separation of Concerns

To insert a code template into the document upon room creation, we depend on two other PeerPrep services.

- Room Service MQ: To subscribe to room creation events.
- Question Service MQ: To obtain the initial code template for the editor document based on the question chosen for the room.

By delegating the responsibility of handling messages from these services, we reduce Editor Service dependencies. Editor Service only relies on the services that are required to effectively facilitate communication between multiple clients.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_docs_service_api
Description: Runs the Docs Service container.
Environmental Variables:

- `DOCS_SERVICE_PORT` - The port used to bind the WebSocket server.
- `REDIS_USERNAME` - Username for the Redis database.
- `REDIS_PASSWORD` - Password for the Redis database.
- `REDIS_HOST` - Address of the Redis database host.
- `REDIS_PORT` - Port the Redis database.
- `REDIS_DB` - Database index used for Redis database.
- `MQ_USER` - Username for the Room Service MQ.
- `MQ_PASSWORD` - Password for the Room Service MQ.
- `MQ_HOST` - Address of the Room Service MQ.
- `MQ_PORT` - Port of the Room Service MQ.
- `MQ_EXCHANGE_NAME` - Exchange name for the Room Service MQ.
- `MQ_QUEUE_NAME` - Queue name for the Room Service MQ.

REST API

Retrieve editor document text

[GET] `/docs-service/docs/:room-id`

Retrieve the editor document text corresponding to a specified room.

This is only intended to be used by other services.

Path Parameters

- `room-id` - Room ID of the room that owns the editor document.

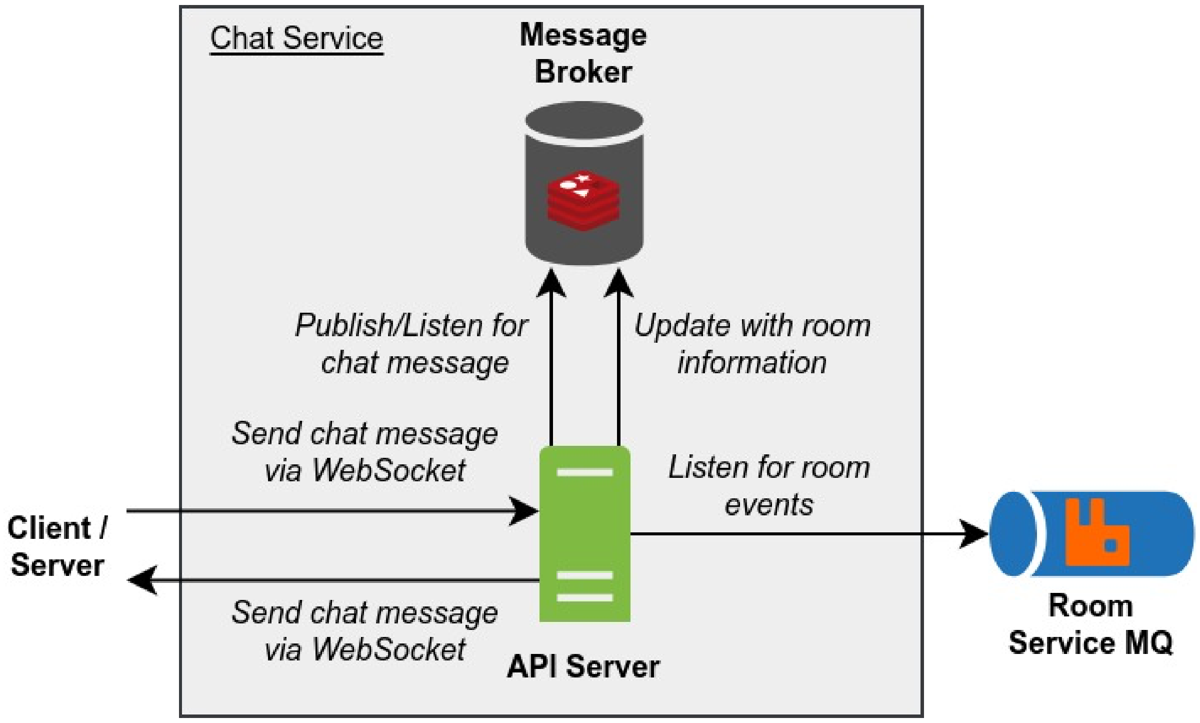
Response

- `200` - Success. The response will contain the editor document text.
- `400` - One or more query parameters are invalid.
- `404` - The requested editor document does not exist.
- `500` - Unexpected error occurred on the server.

Chat Service

Handles the delivery of chat messages among users in the same room.

Architecture


Chat Service
Message Broker
Publish/Listen for chat message
Update with room information
Send chat message via WebSocket
Listen for room events
Client / Server
Send chat message via WebSocket
API Server
Room Service MQ

Message Broker

To ensure chat service is scalable, we employ the use of Redis streams as a form of data synchronisation utilised for horizontal scaling. This ensures that regardless of which server pod the client is connected to, events are able to be sent to another client connected to another server pod.

API Server

Additionally, the team decided to utilise the Message Queue for inter-service communication between the Room Service and the Chat Service. This is primarily due to the fact there is no need for Chat Service to respond to the information received from the Room Service. Hence, by listening to the Events sent out by room service, this enables asynchronous communication between the two services which helps to reduce coupling and latency, making the application more scalable.

To enable bidirectional, low-latency, event driven communication between the client and the server, the team decided on the use of Socket.io which creates a websocket between the client and server. Should the websocket fail, Socket.io falls back to HTTP long polling. To ensure socket connections are synced amongst all server pods in the event of horizontal scaling, Redis streams are used as mentioned above.

The Chat Service listens to room events and stores and updates valid rooms and its associated users in a Redis key-value database, using the roomId as the key and a set to store valid users for that room. This information is populated by listening to events from the Room Service via the RabbiMQ message queue which subsequently populates the Redis Database.

To ensure a user opens a connection and only joins his permitted room, the userId and roomId are passed to the socket server during the handshake connection. The userId and roomId is then checked against the stored values in the database to ensure the connection is valid.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_chat_service_api
Description: Runs the API.
Environmental Variables:

- `REDIS_USERNAME` - Username for the Redis database (the caching database).
- `REDIS_PASSWORD` - Password for the Redis database (the caching database).
- `REDIS_HOST` - Address of the Redis database host (the caching database).
- `REDIS_PORT` - Port the Redis database (the caching database) is listening on.
- `REDIS_SHOULD_USE_TLS` - Should the Redis database connection be secured with TLS. Set to "true" to enable.
- `ROOM_SERVICE_MQ_USER` - User on the MQ host.
- `ROOM_SERVICE_MQ_PASSWORD` - Password of the MQ.
- `ROOM_SERVICE_MQ_HOST` - Address of the MQ host.
- `ROOM_SERVICE_MQ_PORT` - Port the MQ is listening on.
- `ROOM_SERVICE_MQ_VHOST` - Vhost of the MQ.
- `ROOM_SERVICE_MQ_SHOULD_USE_TLS` - Should MQ connection be secured with TLS. Set to "true" to enable.
- `ROOM_SERVICE_MQ_EXCHANGE_NAME` - Name of the MQ exchange.
- `ROOM_SERVICE_MQ_QUEUE_NAME` - Name of the MQ Queue.
- `API_PORT` - Port to listen on.

API - Events

Connecting to Server

Event Name: `connect`
Query Params:
- roomId: The room ID of the match
- userId: The User ID of the associated client connection

The userId is queried against all valid users for roomId. If the roomId is invalid or the user is not a valid user for the room, the socket is closed by the server. Else a websocket connection is initiated.

The room ID and associated user IDs are determined by Room Events published by Room Services

Sending a Message

Event Name: `sendMessage`
Object Payload:
- Message : string - Message sent by client

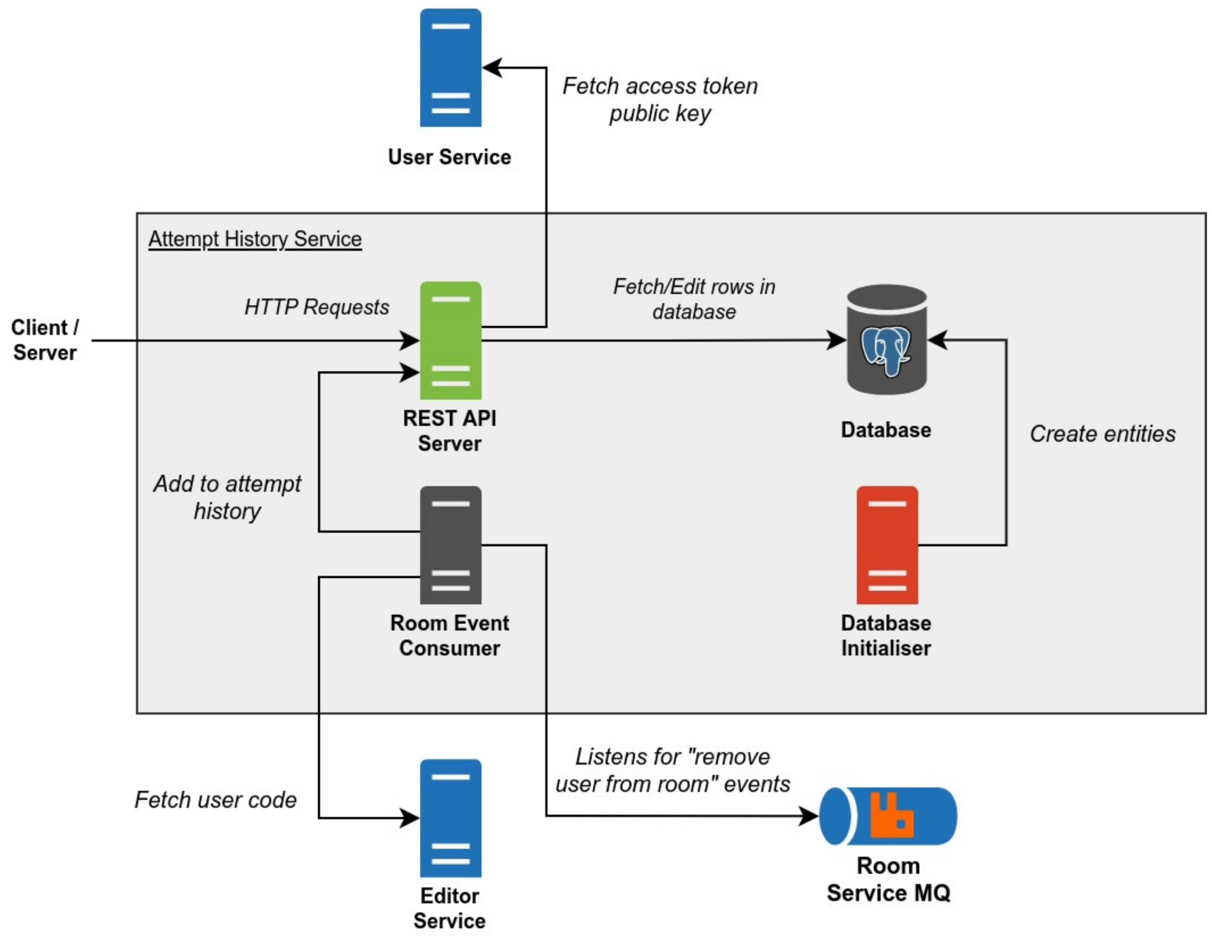
Sending a Message

Event Name: `receiveMessage`
Object Payload:
- MessageObject - MessagePayload for client receiving message
    - Message: string
    - UserName : string

Attempt History Service

Keeps track of the user's attempts.

Architecture



REST API Server

The REST API server handles REST API requests. This server is stateless allowing it to easily scale up and down horizontally based on the number of REST API requests.

There are some limitations to this history service as questions can be deleted. In hindsight, it would have been better for the question service to perform soft update and soft deletion rather than hard deletion for attempt history's consistency sake. For example, currently a user may be able to complete question alpha version 1, but two days later the question may be updated to version 2, currently, attempt service would show that the user has attempted version 2.

Database Initialiser

Initialises the database by creating the necessary entities. This is a job that runs once. Upon completion, it shuts down.

If the database already contains one or more entities that the initialiser intends to create, the initialisation is aborted to prevent potential loss of data. This behaviour can be changed via an environment variable.

Although the database initialiser could be combined with the REST API server such that the initialisation runs every time a REST API server is created, that would give the REST API server multiple responsibilities which is not ideal.

Database

Here, consideration for the database was whether the data would eventually be structured into a relational model. This is because currently there is no code judge in PeerPrep, and perhaps in the future there would be a desire to evolve the matching system. In addition, to give future developers the ability to choose their preferred DBMS, TypeORM library is used to manage the databases.

Room Event Consumer

Room Event Consumer is split from the API server to allow scaling at different levels. That is to say, if the API service is facing higher traffic and is made to scale, then there would not be redundant code that is running in the background.

Currently, Editor Service does not broadcast the code via a message MQ, and thus there is a need to retrieve it via an API call. However, this may have its own limitations as there might be a limitation to how much data can be sent at one go. The developers believe that there could be improvements made to this architecture.

Docker Images

API

Name: ghcr.io/cs3219-ay2324s1-g04/peerprep_attempt_history_service_api
Description: Runs the REST API.
Environment Variables:

- `DATABASE_USER` - Username of the user in the database.
- `DATABASE_PASSWORD` - Password of the user in the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `USER_SERVICE_HOST` - Address of the User Service host.
- `USER_SERVICE_PORT` - Port the User Service is listening on.
- `API_PORT` - Port to listen on.
- `NODE_ENV` - Mode the app is running in ("development" or "production").

Database Initialiser

Name:
ghcr.io/cs3219-ay2324s1-g04/peerprep_attempt_history_service_database_initialiser
Description: Initialises the database by creating the necessary entities.
Environment Variables:

- `DATABASE_USER` - Username of the user in the database.
- `DATABASE_PASSWORD` - Password of the user in the database.
- `DATABASE_HOST` - Address of the database host.
- `DATABASE_PORT` - Port the database is listening on.
- `DATABASE_SHOULD_USE_TLS` - Should database connection be secured with TLS. Set to "true" to enable.
- `DATABASE_NAME` - Name of the database.
- `DATABASE_CONNECTION_TIMEOUT_MILLIS` - Number of milliseconds for a database client to connect to the database before timing out.
- `DATABASE_MAX_CLIENT_COUNT` - Max number of database clients.
- `SHOULD_FORCE_INITIALISATION` - Should database initialisation be done regardless of whether one or more entities to be created already exist. Set to "true" to enable (may cause data loss).

Room Event Consumer

Name:
ghcr.io/cs3219-ay2324s1-g04/peerprep_attempt_history_service_room_event_consumer
Description: Listens for "remove user from room event" from the Room Service MQ and adds an attempt history when such events are consumed.
Environment Variables:

- `API_HOST` - Address of Attempt History Service API Server.
- `API_PORT` - Port of Attempt History Service API Server.
- `ROOM_SERVICE_MQ_USER` - User on the Room Service MQ host.
- `ROOM_SERVICE_MQ_PASSWORD` - Password of the Room Service MQ.
- `ROOM_SERVICE_MQ_HOST` - Address of the Room Service MQ host.
- `ROOM_SERVICE_MQ_PORT` - Port the Room Service MQ is listening on.
- `ROOM_SERVICE_MQ_VHOST` - Vhost of the Room Service MQ.
- `ROOM_SERVICE_MQ_SHOULD_USE_TLS` - Should Room Service MQ connection be secured with TLS. Set to "true" to enable.
- `ROOM_SERVICE_MQ_EXCHANGE_NAME` - Name of the Room Service MQ exchange.
- `ROOM_SERVICE_MQ_QUEUE_NAME` - Name of the Room Service MQ queue.
- `USER_SERVICE_HOST` - Address of the User Service host.
- `USER_SERVICE_PORT` - Port the User Service is listening on.
- `DOCS_SERVICE_HOST` - Address of the Docs Service host.
- `DOCS_SERVICE_PORT` - Port the Docs Service is listening on.

REST API

Get user attempt history

[GET] `/attempt-history-service/`

Cookies

- `access-token` - Access token.

Response

- `200` - `{ message: "Success", data : [{"attemptId":string,"questionId":string,"language":string,"date":date}] }`
- `401` - `{ message: "Not authorized" }`
- `500` - `{ message: "Sever Error" }`

Get user attempt code

[GET] `/attempt-history-service/:aid`

This returns a particular user's attempt, in particular their code.

Params

- `aid` - The attempt id.

Cookies

- `access-token` - Access token.

Response

- `200` - `{ message: "Success", data : user's code }`
- `401` - `{ message: "Not authorized" }`
- `500` - `{ message: "Sever Error" }`

Add to attempt history

[POST] `/attempt-history-service/add`

Note that (user id, room id) is considered primary keys. That is to say, their combination is what makes an entry unique.

Parameters

- `user` - The user id
- `question` - The question id
- `room` - The room id
- `language` - The language id

Body

- Json with the key being 'code' and the value being the user's code.

Response

- `200` - `{ message: "Success" }.`
- `500` - `{ message: "Sever Error" }`

Rank all users by attempt count

[GET] `/attempt-history-service/all`

Rank all users by attempt count and sort them by user id and count.

Response

- `200` - `{ message: "Received message", {"data": [ {"user-id": number, "attempt_count": string}]}}`
- `500` - `{ message: "Server Error" }`

PeerPrep

Feature Enhancements

If the team had more time or perhaps more experience, the team would like to add the following improvement and enchantments.

1. Code Judge

   The team was interested in creating a code judge for the project, however was ultimately unable to do so due to time constraints and limited knowhow.

2. More caching and logging.

   Some services receive lots of calls, specifically things like question service and history service. While question service was able to implement a caching solution, history service was not able to within the time frame.

   In addition, the team would have liked to implement a message queue system that logs events into a database. This would have allowed us to keep track of changes to the system (i.e. changes to questions, credentials, logging crashing servers), and also allow us to perform a system rollback if needed.

3. Catering for more types of users

   One of the nice to have features was to cater to users with some form of colour blindness. This would have improved the experience of these users as they can choose the colours of their editors to suit their needs and preferences.

4. Automated testing

   This was something that we were hoping to carry over from 2103, however, it was a less critical component in the entire big picture.

5. Gamification

   Adding gamification elements would help us to better match users. For example, matching users who have the same relative score, or match users who have obtained certain badges.

6. Password Recovery

   Providing password recovery would be an essential error recovery mechanism in the case a user forgets their password.

7. Code Editor enhancements

   Currently the code template for an editor document is inserted once when a room is created. We should allow the user to regenerate the code template if they accidentally delete it.

   We should allow users to change the coding language while they are collaborating.

PeerPrep

Reflections

Planning

Overall the team found the experience of building this application a fulfilling one as we managed to learn more about the tools that the industry is using. Our initial hunch of breaking the project up into what we were specialised in helped to get us started quickly, generating ideas of what should and should not be in our project backlog, recommending the type of technological stacks to be used and segregating the tasks to be done.

Solving problems as a team

However, the team very quickly hit the situation of differentiated learning and waiting on another team member to complete their task. For the large part, we attempted to circumvent this large gap in knowledge by sharing with each other as many learning resources as possible, and telling each other the interesting findings that we had encountered. For example, specifying capital letters in TypeORM entities causes postgresql to become case sensitive, or what is the best way to structure our folder and files.

United we fall, divided we stand

Despite deploying an extreme form of separation of concerns by having each member develop a particular service, there was the occasional overlap of tasks that cost our team some time. This largely occurred because two members are working on different branches and realised that they needed the same thing, but is mostly resolved through agreeing who takes precedence in setting the expectation for the required shared element.

The importance of defining bounded contexts

Occasionally, our team reviewed each other's code and pointed out areas where code can be reused or simplified. Something that comes to mind would be how we standardised variable names to make it easier for an external developer to relate how components interact with each other. For example, for questions, front-end and matching service, initially we used different terms to relay the same idea of 'difficulty', which led to some confusion when integrating the three components together but was eventually resolved after some discussion as to viewing it from a user's point of view.

Conclusion

In all, this project was an interesting step up from CS2103, where the program was a single user non-concurrent monolith system. This project provided various learning points and challenges to the team, from learning to deal with concurrency, to learning to create expectations and promises as a developer.

PeerPrep

Appendix

Glossary

Match

A match represents a pairing between two users and a question.

- *e.g. (Alice, Bob, Merge k Sorted Lists - javascript)*

Matchmaking

Matchmaking is the process in which we obtain an optimal match based on the criteria specified by two users.

Room

Room is a unique collaborative space shared by multiple users.

- i.e. Matchmaking generates a room for users based on a match.

Question

Question represents a technical interview style problem.

Document

A document is a text document that can be edited concurrently by users who belong to the same room.

Editor

The editor refers to the collaborative real-time editor that enables users to edit documents in real-time.

Session

Session refers to an instance of a user’s login session.

- This login session consists of a session ID and user ID.

Access Token

Access token is a mechanism for clients to authenticate with the server.