CodeParty

CS3219 Group 11 Project Report and Developer Docs

CodeParty: Technical Interview Preparation with Peers

Team Members

Name	Matric Number	NUS Number
Ong Jun Xiong	A0124943E	E0533994
Tay Yi Hsuen	A0214979M	E0534486
Charisma Kausar	A0226593X	E0638879
Gabriel Goh	A0217702M	E0543738
Lee Chun Wei	A0217534H	E0543570

Table of Contents

1. Introduction

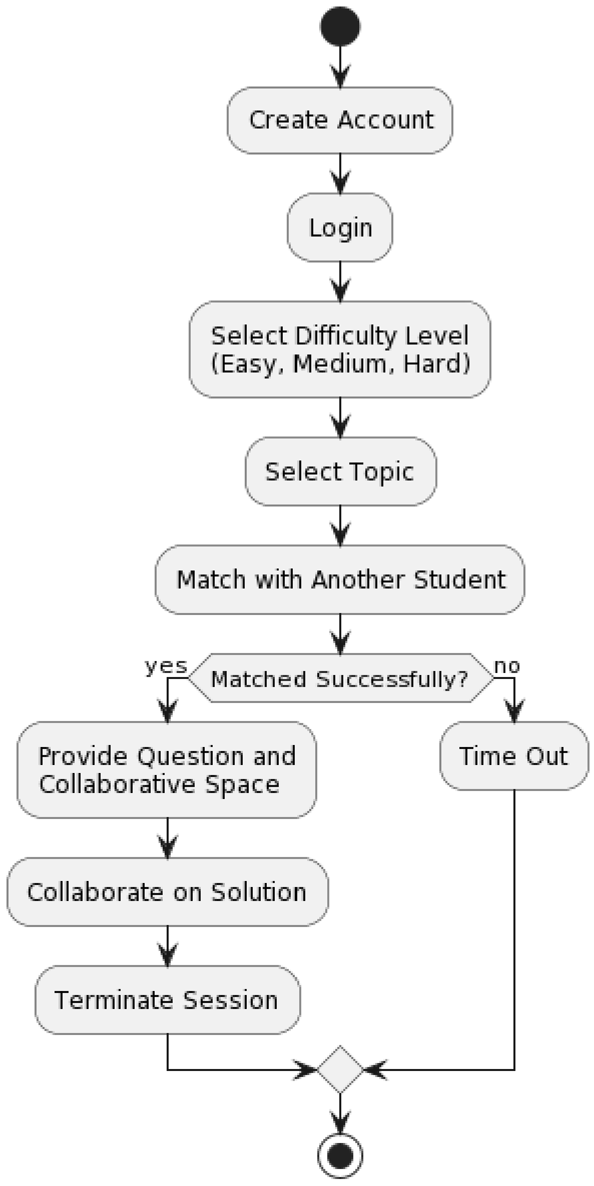
1.1. Background

The CodeParty project aims to create a comprehensive platform that enables users to prepare collaboratively for interviews or other forms of assessments by connecting peers, facilitating real-time collaboration, and providing a rich repository of questions. The platform is founded on a microservices architecture to ensure scalability, maintainability, and agility in rolling out new features.

1.2. Purpose

CodeParty is designed with the primary aim of amalgamating the benefits of collaborative learning with the convenience of digital technology.

1.3 User Flow



Create Account
Login
Select Difficulty Level (Easy, Medium, Hard)
Select Topic
Match with Another Student
yes
Matched Successfully?
no
Provide Question and Collaborative Space
Time Out
Collaborate on Solution
Terminate Session

Diagram 1: User Flow of CodeParty user

The user flow of CodeParty is according to the given requirements:

“A student who is keen to prepare for their technical interviews visits the site. They create an account and then log in. After logging in, the student selects the question difficulty level (easy, medium, or hard) and a topic they want to attempt today. The student then waits until they are matched with another online student who has selected the same difficulty level as them. If they are not successfully matched after a specific duration, they time out. If they are successfully matched, the student is provided with the question and a collaborative space to develop their solution in real-time, allowing both the student and their matched peer to collaborate on the provided question. The application should allow the users to terminate the collaborative session gracefully.”

2. Sub-group Contributions

While the essence of our project's success lay in the cohesive effort of the entire team, with everyone contributing significantly across various facets of the project, we acknowledge the necessity to delineate contributions specific to each sub-group for the purpose of grading. It's imperative to note that the following breakdown does not fully encompass the collective spirit and unified commitment that was ubiquitously present throughout the project's lifecycle. However, for the sake of clarity and fairness in the evaluation process, we present the contributions that can be distinctly attributed to individual sub-groups.

Sub-group	Members	Technical Contributions
Sub-group A	Jun Xiong Yi Hsuen Gabriel	N8: Extensive testing N9: Deployment of the app N11: API gateway N5: Enhance collaboration service (done but not claimed for 3NTHs)
Sub-group B	Charisma Chun Wei	N1: Communication N2: History N4: Enhance question service N10: Scalability (done but not claimed for 3NTHs)

3. Requirements

Below are the list of features we have completed as part of the coursework.

1. M1: User Service: High priority, foundational for user interaction with the platform.
2. M2: Matching Service: Medium priority, enhances user experience but not foundational.
3. M3: Question Service: High priority, crucial for providing users with preparation material.
4. M4: Collaboration Service: High priority, as real-time collaboration is a core feature.
5. M5: Basic UI for User Interaction: Medium priority, necessary for access but can initially function with a basic design.
6. M6: Deployment: Low priority, necessary for the final stages of the development process.
7. N1: Communication: Implement a mechanism to facilitate 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.
8. N2: History: Maintain a record of the questions attempted by the user e.g., maintain a list of questions attempted along with the date-time of attempt, the attempt itself and/or suggested solutions.
9. N4: Enhance question service to enable managing questions, for example, tagging (by topic, popularity, etc.,), retrieving questions on the fly during a session initiation.
10. N5: Enhance collaboration service by providing an improved code editor with code formatting, syntax highlighting for one language, syntax highlighting for multiple languages
11. N8: Extensive (and automated) unit, integration, and system testing using CI. Teams can also use various test frameworks or demonstrate effective usage of CI/CD in the project.
12. N9: Deployment of the app on the production system (AWS/GCP cloud platform).
13. N10: Scalability – the deployed application should demonstrate easy scalability of some form. An example would be using a Kubernetes horizontal pod auto-scaler to scale up the number of application pods when there is a high load.
14. N11: The application should have an API gateway of some kind that redirects requests to the relevant microservices. An example would be using an ingress controller such as NGINX ingress controller if using Kubernetes (https://kubernetes.GitHub.io/ingress-nginx/).

3.1. Requirements Breakdown

3.1.1. Must Haves Requirements

M1. User Service (Quicklink: 4.4.1. M1: User Service)		**Priority**
F1.1	**Unauthenticated user**	
F1.1.1	Whenever an unauthenticated user visits any page in the app, the user should be sent back to the home page	H
F1.1.2	The login screen/button should support GitHub authentication for new users to create an account on the service	H
F1.2	**User registers for account for first time**	
F1.2.1	Users should be able to sign in for the first time with their GitHub account	H
F1.2.2	The supplied GitHub account should be used to initialize an account on an Identity Server	H
F1.2.3	Should the user decide to cancel registration, the user should be brought back to the login screen	M
F1.3	**User logs in**	
F1.3.1	Users should be able to sign in with the same GitHub account	H
F1.3.2	Once a user logs in, the user should be given a JWT token for authentication	H
F1.3.3	Users should be given roles for authorization (admin or normal user)	H
F1.4	**User updates account settings**	
F1.4.1	A user should be able to change their display name in a form.	H
F1.4.2	A user should be able to store match preferences on the user service. E.g. misc details used for matching with other users - preferred question difficulty and preferred language	L
F1.5	**User logs out**	
F1.5.1	Once the user logs out (or the session expires), the user should be redirected to the login screen	H

F1.6	**User deletes account**	
F1.6.1	Once a user deletes the account, the user should be logged out and redirected to the login screen	H
F1.6.2	When a user has not been recently authenticated, then they should be prompted to authenticate again before deleting.	M
F1.6.3	The user should be asked for confirmation if they want to delete their account	M
NF1.7	**Privacy and Security**	
NF1.7.1	Under no circumstances should user passwords be revealed by the application to other users, even the administrators	H
NF1.7.2	User passwords or credentials should be stored securely -> they are never stored as plain text	H
NF1.7.3	Actions made by users should be logged for auditability	M
NF1.7.4	A user should only be able to update his own preferences and account details and not those of other users	H
NF1.7.5	If user chooses to delete account, all data regarding the user should be deleted	H
NF1.8	**Scalability**	
NF1.8.1	The user service should be able to horizontally scale up to match an increase in API calls for production usage	M
NF1.9	**Availability**	
NF1.9.1	User account data should be recoverable in case of a database failure	H
NF1.9.2	User data should be persisted after logging out from the service.	H
NF1.10	**Usability**	
NF1.10.1	Once a new user creates an account, the user should be logged in immediately	M
M2. Matching Service - to match 2 students with similar criteria (Quicklink: 4.4.2. M2: Matching Service)		**Priority**
F2.1	**User submit matching request**	
F2.1.1	User selects the difficulty level (Easy, Medium, Hard) and programming language (Python or Java or C++) and submits the request	H

F2.1.2	If user exits the session, the matching request is terminated	M
F2.2	**User is matched with another suitable user**	
F2.2.1	A waiting user can be matched with a new user who has selected some common difficulty and language	H
F2.3	**User is notified of a match**	
F2.3.1	After a successful match, both users are notified.	H
F2.4	**Usability of System**	
F2.4.1	User can cancel the request before they have been matched	H
F2.4.2	User will wait for up to 30 seconds before their match request is canceled by system	M
NF2.5	**Performance and Availability**	
NF2.5.1	Handle up to 100 users finding matches concurrently	M
NF2.5.2	If a match is possible, both users are notified within 5 seconds	M
NF2.5.3	Be resistant against multiple match requests from the same user (preventing Denial of Service)	L
M3. Question Service - to maintain a question repository indexed by difficulty level (Quicklink: 4.4.3. M3: Question Service)		**Priority**
F3.1 Connect to a database to CRUD questions		
F3.1.1	Authenticated users can retrieve questions from the database to display to the user based on difficulty level	H
F3.1.2	Admin users can create new questions with unique titles	M
F3.1.3	Admin users can update existing questions	M
F3.1.4	Admin users can delete questions	M
F3.1.5	Authenticated users can search for questions based on the title keywords and sorting order	L
F3.2 The system provides a RESTful API		
F3.2.1	System can get a question or a list of questions	H
F3.2.2	User can get one question at random if there are more than 1 questions to be returned based on their selection criteria for practice	H

NF3.3 Performance - Question Service must perform fast		
NF3.3.1	Return questions to display to the user in 1 second or less	H
NF3.4 Reliability		
NF3.4.1	System should be up at least 99% of the time.	H
M4. Collaboration Service - provides a mechanism for real-time communication between users (Quicklink: [4.4.4. M4: Collaboration Service](#))		
F4.1 Set up real-time communication back-end		
F4.1.1	Users should be able to communicate to the server with real-time communication.	H
F4.2 Real-time code editing feature		
F4.2.1	Users should be able to interface with a code editor that supports real-time updates	H
F4.2.2	Users should be allowed to edit the code simultaneously	M
F4.3 Integrate with Matching Service		
F4.3.1	Once users are matched, they should be put into a collaboration room/session for them	H
F4.3.2	Allow users to leave and rejoin sessions even if they disconnect midway	M
F4.4 Provide RESTful API for Collaboration Service		
F4.4.1	The system provides an API to initiate a collaboration session for matched users	H
F4.4.2	The system provides an API to fetch ongoing collaboration session details	H
F4.4.3	The system provides an API to end/terminate a collaboration session	L
NF4.5 Collaboration service should be user-friendly		
NF4.5.1	User receives clear notifications or alerts if connection issues arise during collaboration	M
NF4.6 Collaboration service data should be recoverable		
NF4.6.1	System should store collaboration sessions' data temporarily for any potential recovery needs	M

M5. Basic UI *Most of the requirements for M5 have been mentioned under the rest of the services, corresponding to the backend microservice they are associated with. Some common ones are mentioned here.* **(Quicklink: 4.4.16. Frontend Design)**		
F5.1 Usability		
F5.1.1	Confirm dangerous user actions with the user	H
F5.1.2	Show user the status of the action performed or error states	M
F5.2 Styling		
F5.2.1	Ensure the UI is usable by employing stylesheets	M
F5.2.2	Standardize styles across the app	L
M6. Basic Deployment *Excludes deploying to Google Cloud* **(Quicklink: 4.2.4. Deployment Strategy)**		
F6.1 Basic Deployment		
F6.1.1	Each service should be able to run on its own Docker container	H

3.1.2. Nice to Have Requirements

N1. Communication in collaborative space - provides a mechanism to facilitate communication during collaboration (Quicklink: 4.4.5. N1: Communication)		
F1.1 User interface to turn on audio/video		
F1.1.1	Allow user to join a video call room	H
F1.1.2	Allow user to switch off/on their camera	M
F1.1.3	Allow user to switch off/on their microphone	M
F1.2 User interface to display user video/audio		
F1.2.1	User should be able to see the videos of themselves and other users in the same collaboration room who have turned on their camera	H
F1.2.2	User should be able to hear the audio of other participants in the collaboration room if they are unmuted	H

NF1.3 Reliability		
NF1.3.1	Streaming of audio and video should be reliable by using established communication protocols for video calls	H
NF1.4 Usability/Privacy		
NF1.4.1	User should only be able to join a room if they are assigned to it by the matching/collaboration service	H
NF1.4.2	Users should be asked for permission to access microphone and camera	M
NF1.4.3	Users should be notified with an error message if sufficient permissions are not given to start audio/video	M
NF1.4.4	User should mute and turn off their camera by default	M
NF1.5 Performance		
NF1.5.1	Audio and video quality must be high (min 720p)	M
NF1.5.2	The latency should be small (below 1 second)	M
NF1.6 Security		
NF1.6.1	Only authorized users should be able to send and receive video and audio information in the same video room	H
NF1.6.2	Users should use an authentication token (like JWT) with the video server.	H
N2. History (Quicklink: 4.4.6. N2: History)		
F2.1 History of questions attempted by the user		
F2.1.1	System saves collaboration rooms after all users disconnect, i.e. the question and users' attempts	H
F2.1.2	System saves users questions attempted on their own without collaboration	H
F2.1.3	User can see history of attempted questions, including the question, their code and time attempted	H
NF2.2 Usability		
NF2.2.1	User does not need to click on button to manually save an attempt, but is automatically saved on leaving the collaboration room	L
N4. Enhance Question Service (Quicklink: 4.4.8. N4: Enhance Question Service)		

F4.1 Tagging of questions		
F4.1.1	Allow questions to be tagged based on topic	H
F4.2 Retrieval of questions on the fly		
F4.2.1	Allow matched users to choose a random different question in a collaboration session	M
F4.2.2	Ensure that both users agree to change questions before swapping the question	L
F4.3 Searching, sorting and filtering of questions		
F4.3.1	Allow users to search the entire database of questions by a case-insensitive substring of the title.	M
F4.3.2	Allow users to do a case-sensitive sorting of the table of questions by title	M
F4.3.3	Allow users to filter the table based on difficulty	M
F4.4 Allow markdown formatting		
F4.4.1	Allow users to write a question description with markdown and simple HTML formatting	M
F4.5 Show question solution		
F4.5.1	Display the solution of a question during solo or room practice when requested	M
F4.5.2	The solution displayed should have syntax highlighting	L
NF4.6 Performance of query of questions		
NF4.6.1	The searching and sorting of questions by title should take no longer than 1 second on average	M
NF4.7 Security		
NF4.7.1	Sanitize the user-inputted question description in markdown and HTML format to avoid cross-site scripting (XSS) attacks	H
N5. Enhance Collaboration Service (Quicklink: 4.4.9. N5: Enhance Collaboration Service)		
F5.1 Code editor		
F5.1.1	Users can edit code with syntax highlighting	M
F5.1.2	Users can edit code with automatic indent formatting	H

F5.1.3	Users can edit code with simple code completion	L
F5.1.4	If the user changes the programming language, the syntax highlighting would change	M
F5.1.5	User should be able to switch between more than one language with minimal support for python, c++ and java	M
F5.2 Concurrent edit collision handling		
F5.2.1	Users can edit concurrently without completely removing another user's edit on collision (no data race)	M
F5.2.2	Users should be able to maintain their cursor position after another user makes an edit.	M
NF5.3 Low Latency real time changes		
F5.3.1	Users should not feel like another user is lagging when concurrently editing	M
F5.3.2	Users should be able to make edits even if one of the users internet connection is unstable. (Communication should be hosted by server)	M
NF5.4 Usability		
NF5.4.1	System should have user-friendly interface for the code editor with features like syntax highlighting and line numbers	M
N8. Extensive CI/CD pipeline (Quicklink: 4.4.10. N8: Extensive CI/CD Pipeline)		
F8.1. Continuous Integration Setup		
F8.1.1	Implement automated unit testing for individual files	H
F8.1.2	Set up automated system testing for microservices	M
F8.1.3	Integrate linting of Typescript files as part of the CI process to ensure code quality and consistency.	H
F8.1.4	Configure the CI pipeline to run tests upon push to any branch	H
F8.2. Continuous Deployment Setup		
F8.2.1	Implement automated deployment of the application to production upon changes to code in a specific branch	H
F8.2.2	Notify developers any problems with deployment	H

F8.2.3	Build and push Docker Images to GCS automatically in the CD process.	M
F8.3. Pipeline Maintenance and Optimization		
F8.3.1	Optimize pipeline execution time to ensure quick feedback and efficient use of resources.	M
F8.3.2	Implement monitoring and logging for the CI/CD pipelines to identify and troubleshoot issues promptly.	M
NF8.4. Reliability and Stability		
F8.4.1	Ensure high reliability of the CI/CD pipeline, minimizing failures due to pipeline configuration errors.	H
F8.4.2	Implement automatic recovery mechanisms for pipeline failures to maintain continuous operation.	M
NF8.5. Scalability and Performance		
F8.5.1	Design the CI/CD pipeline to efficiently handle an increasing number of builds and deployments as the application grows.	H
F8.5.2	Ensure the pipeline can support concurrent builds and deployments without significant performance degradation.	M
N9. Deployment of App (Quicklink: 4.4.11. N9: Deployment of App)		
F9.1. Infrastructure Setup and Config		
F9.1.1	Ensure that the database services are set up with high availability and automatic failover in mind.	H
F9.1.2	Services should be accessible with domain names, whether internally within a cluster or from the Internet	M
F9.2. Application Deployment and Orchestration		
F9.2.1	Containerize all microservices for the production environment using Docker.	H
F9.2.2	Utilize a managed orchestration service on a cloud platform for the deployment and management of containerized applications.	H
F9.3. Security and Compliance		
F9.3.1	Ensure all data in transit is secured via SSL encryption.	H
NF9.4. Performance and Scalability		

F9.4.1	Ensure the system can handle a predefined number of concurrent users per microservice without performance degradation. (Related to N10)	M
F9.4.2	Plan for future scalability by allowing easy integration of additional resources and services based on demand.	L
N10. Scalability (Quicklink: 4.4.12. N10: Scalability)		
NF10.1 Scalability		
NF10.1.1	The services should be horizontally scalable when run in production.	H
N11. API Gateway (Quicklink: 4.4.13. N11: API Gateway)		
F11.1 Proxying API requests		
F11.1.1	The API gateway should sit between the frontend and the backend services.	H
F11.1.2	The API gateway should be able to verify authentication and authorization of requests that pass through it.	H
F11.1.3	The API gateway should be reachable from frontend code that is loaded on a browser.	H
NF11.2 Security		
NF11.2.1	Backend services must not be directly exposed to the Internet	H

4. Developer Documentation

4.1. Development Process

4.1.1. Process

We adopted an Agile development approach, characterized by short development cycles or 'sprints'. Each sprint was two weeks long and aimed at delivering a predefined product increment. This approach provided the flexibility to adapt to changes swiftly and break up development into manageable chunks.

To ensure code quality and consistency across different microservices, we established a code review process. Every piece of code was reviewed by at least one other developer before being merged into the main branch.

4.1.2. Development Workflow

We followed a branching Workflow here where:

- Prod Branch: Contains stable code that's ready for production.
- Development/Master Branch: Used for active development; may be unstable.
- Feature Branches: Branches created off the Development branch for developing new features or bug fixes

4.2. Technical Architecture Overview

This section provides a detailed overview of the technical architecture of CodeParty, encompassing the backend and frontend frameworks, communication protocols, database systems, deployment strategies, and CI/CD tools.

4.2.1. Backend Architecture (Microservices)

4.2.1.1. Framework

Our choice of framework for building the backend services is Express.js, known for its minimalism, flexibility, and performance. Each microservice is developed using this framework, enabling us to create a robust set of functionalities with cleaner, maintainable code.

4.2.1.2. Communication

The services communicate with each other using a combination of communication patterns:

- REST API for synchronous HTTP request-response communication.
- Websockets for Real-time Communication (RTC) usage.

4.2.1.3. Chosen microservices and bounded contexts

Based on the requirements of the application, we have split the backend server application into several microservices. These services are chosen based on the bounded contexts of our application. We have chosen the following microservices:

1. User service: This microservice deals with the user profile details, such as their display name, display photo, question attempts and preferred language and difficulty. It allows users to fetch details of themselves or other users, and perform edits on their profile, or even deletions on their own profiles. It also facilitates the creation of a new profile when a new user logs in for the first time.
2. Admin service: This microservice deals with the setting and deleting of admin rights, which can only be accessed by admins. This is separated from the user service, because admin rights are set on the third-party Firebase Authentication provider, which uses custom claims to determine admin rights.
3. Question service: This microservice deals with the CRUD operation of the question repository. This is the only microservice that communicates with the question

database, which decouples the question CRUD logic from the rest of the application.

4. Matching service: This microservice handles the logic of matching 2 users for collaboration. It also ensures that users who disconnect midway through an ongoing match can reconnect back to the same match. In addition, it handles the creation, deletion and even changes in matches (such as when the users want to swap a question in the match).
5. Collaboration service: This microservice handles the deeper logic of collaboration between 2 users in a match. It facilitates high-frequency concurrent editing of code for both users, and constantly stores the user-inputted code to ensure persistence of their attempts. This microservice must be fast and handle concurrent edits well.

4.2.2. Frontend Architecture

4.2.2.1. Framework

The frontend is built with Next.js, a powerful React framework that enables functionality such as server-side rendering and static site generation. TailwindCSS is used for styling, providing utility-first CSS classes that speed up the design process. To understand why we went with this refer to 4.4.4. Frontend Design

4.2.2.2. Communication

The frontend communicates with the backend primarily through HTTP requests to the Backend REST API, using an API Gateway that centralizes and routes the requests. Additionally, we use Websockets for RTC features, which is also routed through our API Gateway.

4.2.3. Database Systems

4.2.3.1. Relational Database

PostgreSQL is our relational database of choice for general use cases, offering advanced functionality, reliability, and a strong emphasis on standards compliance.

4.2.3.2. NoSQL Database

For our question repository, we utilize MongoDB, a NoSQL database known for its scalability and flexibility, ideal for handling varied, unstructured data inherent in such repositories.

4.2.4. Deployment Strategy

We use Docker for containerizing our application, which simplifies dependency management and ensures that our application runs the same way in every environment.

Kubernetes is used for the orchestration of our Docker containers, managing scaling, failover, and deployment patterns among other tasks when in the production environment.

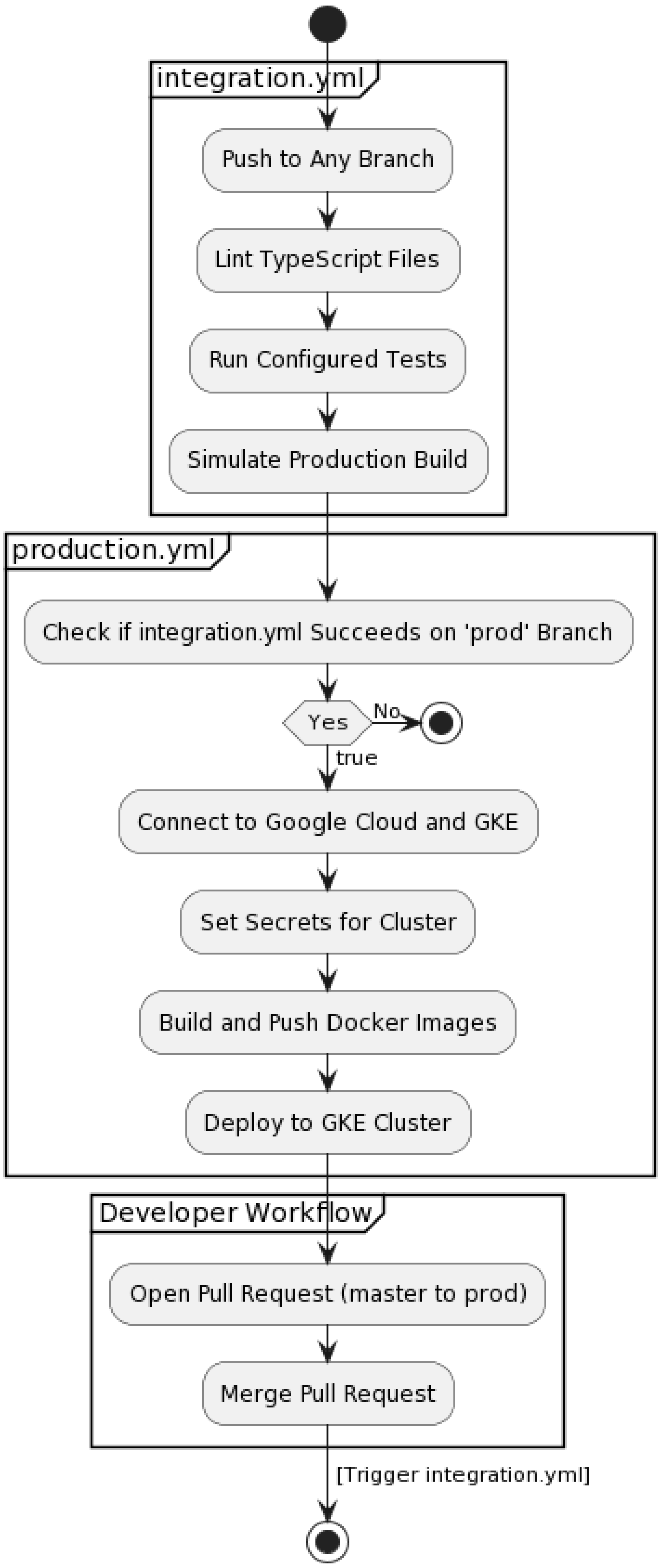
4.2.5. File Structure

Simplified Structure for Microservice + Gateway + Frontend Monorepo

```
/CodeParty
├── /services (express backend)
│   ├── /user-service
│   ├── /matching-service
│   ├── /question-service
│   ├── /admin-service
│   ├── /collaboration-service
│   └── /gateway
├── /frontend
│   └── /src for CodeParty (NextJs application)
├── /deployment
│   ├── /prod-dockerfiles
│   ├── /gke-prod-manifests
│   └── build-export-prod-images.sh (Only needed for prod)
├── /prisma (shared ORM for BE services)
├── /utils (shared files for BE services and FE)
└── README.md (and other root-level files & docs)
```

`/services`: This directory contains all microservices.
`/frontend`: Contains frontend application.
`/deployment`: Contains Dockerfiles, Kubernetes configuration files, etc.

4.2.6. CI/CD Pipeline



integration.yml
Push to Any Branch
Lint TypeScript Files
Run Configured Tests
Simulate Production Build
production.yml
Check if integration.yml Succeeds on 'prod' Branch
Yes
No
true
Connect to Google Cloud and GKE
Set Secrets for Cluster
Build and Push Docker Images
Deploy to GKE Cluster
Developer Workflow
Open Pull Request (master to prod)
Merge Pull Request
[Trigger integration.yml]

Diagram 1.1: CI/CD flow

There are 2 GitHub Actions pipelines: integration.yml (for Continuous Integration) and production.yml (for Continuous Deployment).

Pipeline	**General Steps**	**When the pipeline is run**
integration.yml	1. Lint the TypeScript files 2. Run tests that are configured on certain services 3. Simulate production build (compile the TypeScript code into JavaScript, but won't run the app)	Upon pushing onto any branch of the repo
production.yml	1. Connect to Google Cloud and the Google Kubernetes Engine (GKE) cluster 2. Set the secrets for the cluster 3. Build the Docker images and push them to Google Artifacts Registry 4. Deploy the new Docker images to the GKE cluster based on the manifest files	Upon integration.yml succeeding on the prod branch only

Generally speaking, to push changes to production, a developer needs to open and merge a pull request from the master branch to the prod branch.

4.3. System Design

4.3.1. System-Level

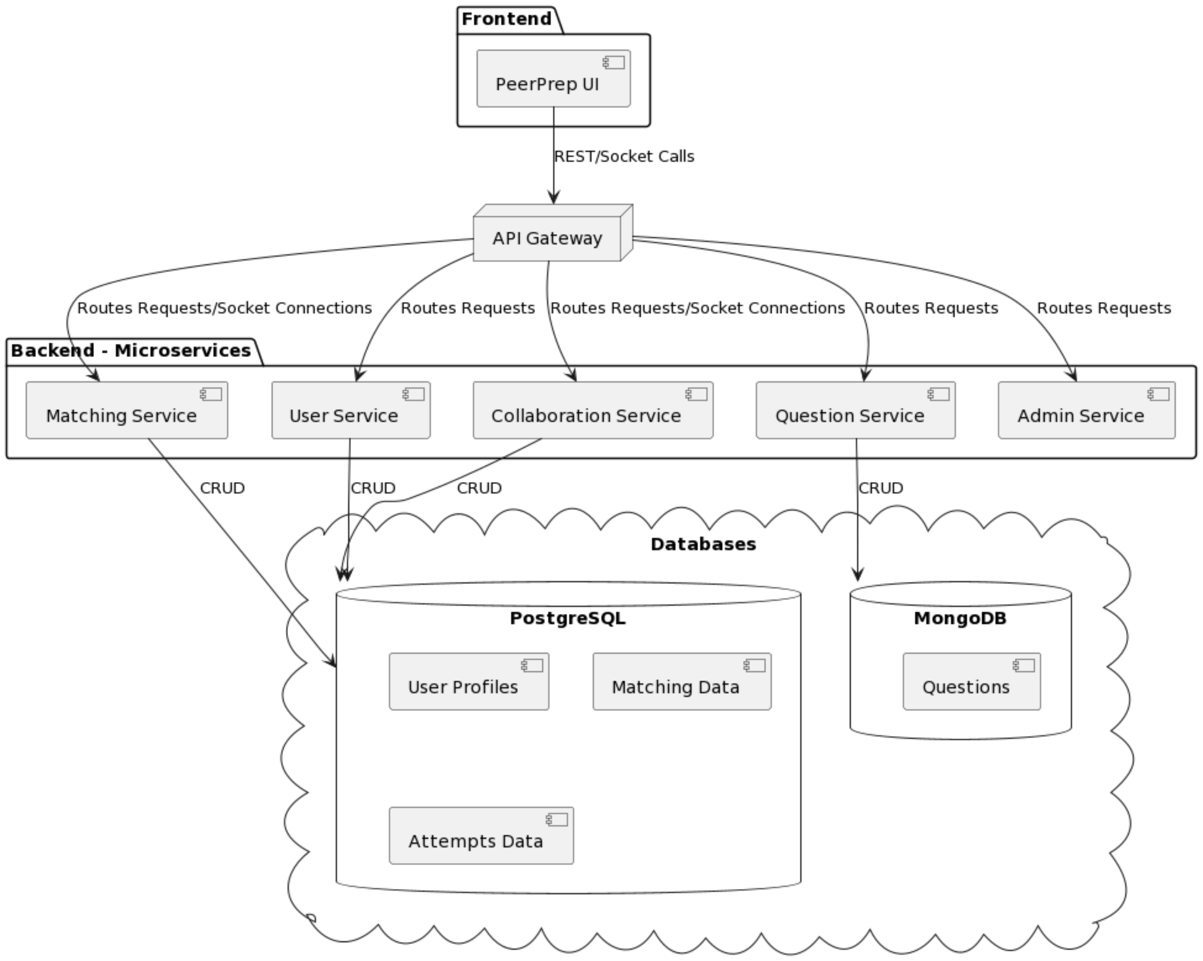


Diagram 2: System Level Design

This diagram provides a high-level view of the system's architecture, illustrating the system's various components and how they interact with each other. It was crucial for understanding the flow of data across the system, identifying potential bottlenecks, and planning for future scalability. More detailed descriptions on each component are as shown below:

- Backend (Microservices):
  - Framework: Express.js - we choose this as its simple, flexible, and performant, it is used to build the backend services.
  - Communication: Services communicate via REST API for complex or internal communications.
- Frontend:
  - Framework: The UI is built using Next.js for its SSR capabilities and TailwindCSS for utility-first styling.

    - Communication: The frontend communicates with the backend primarily through HTTP requests to the REST API and Websockets via the API Gateway, abstracting the complexity of service-to-service communication.
- Database:
    - Relational: PostgreSQL is used for general purposes, such as storing user profiles and matching criteria, due to its robustness, reliability, and feature set.
    - NoSQL: MongoDB is used for the question repository because of its schema flexibility and performance with large volumes of data.
    - Prisma: Was used as our main ORM throughout the application.
- Deployment:
    - Containerization: Docker is used for creating consistent development environments and ensuring the same application behavior across different setups.
    - Orchestration: Kubernetes manages the containers, handling deployment, scaling, and setup in production environments.
- CI/CD:
    - Tools: Github Actions is used for continuous integration and continuous deployment, automating the process from code push to deployment. Workflows in GitHub Actions may also be supported by Bash scripts

Overall, we chose a microservices architecture due to its scalability and resilience. Each service (User, Matching, Question, Collaboration) was developed, deployed, and scaled independently, allowing for faster iterations and releases.

4.3.2. System-Service Level

4.3.2.1. API communication design

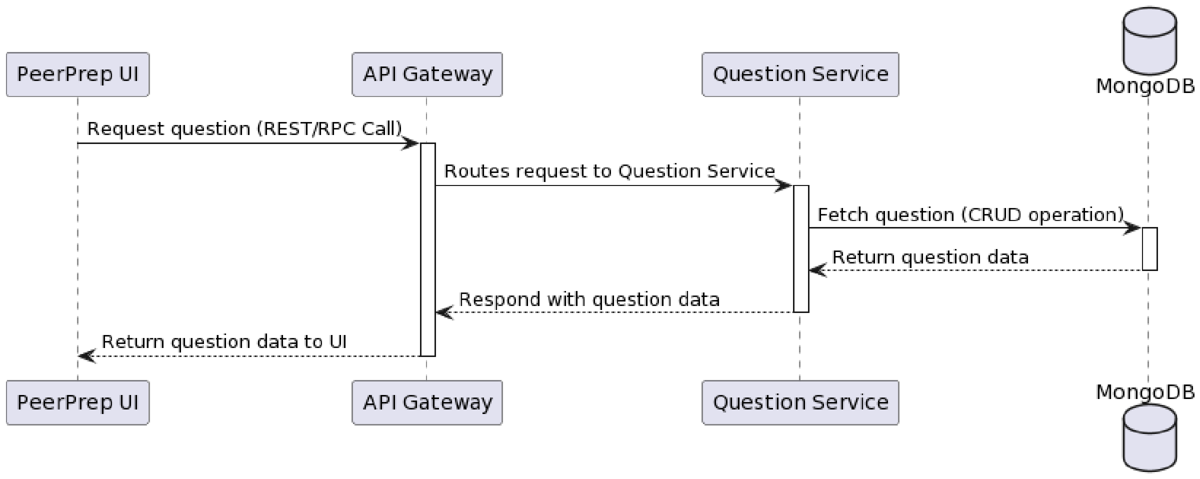

PeerPrep UI
API Gateway
Question Service
MongoDB
Request question (REST/RPC Call)
Routes request to Question Service
Fetch question (CRUD operation)
Return question data
Respond with question data
Return question data to UI
PeerPrep UI
API Gateway
Question Service
MongoDB

Diagram 3: Service-System Level Design - API calls

The above sequence diagram illustrates the sequence of a user request originating from the CodeParty UI, being routed by the API Gateway to the Question Service, which then interacts with the MongoDB to fetch data. The data then travels back up the chain to be presented to the user. This is a simplified view, assuming a happy path where the request is processed without any errors and the system components are always available.

4.3.2.2. Websocket communication design

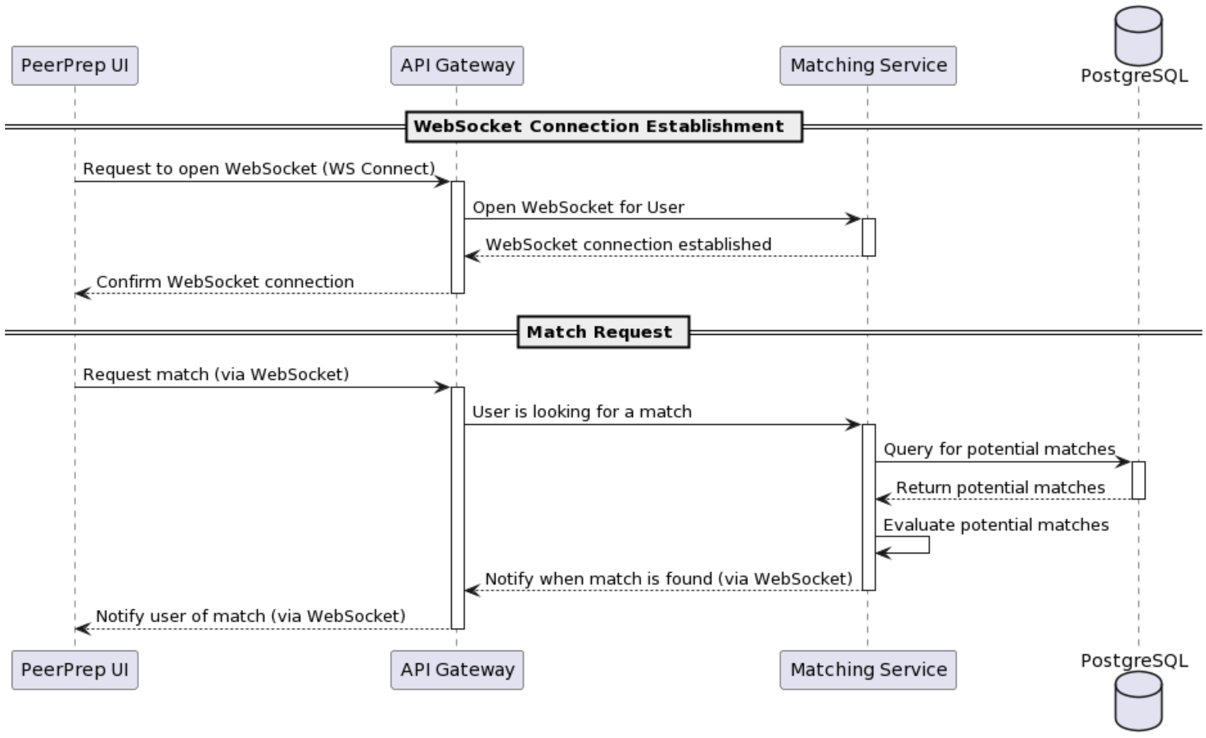

PeerPrep UI
API Gateway
Matching Service
PostgreSQL
WebSocket Connection Establishment
Request to open WebSocket (WS Connect)
Open WebSocket for User
WebSocket connection established
Confirm WebSocket connection
Match Request
Request match (via WebSocket)
User is looking for a match
Query for potential matches
Return potential matches
Evaluate potential matches
Notify when match is found (via WebSocket)
Notify user of match (via WebSocket)

Diagram 4: Service-System Level Design - Web Sockets

In the matching scenario depicted above, the CodeParty UI initiates a real-time communication channel with the backend by requesting the API Gateway to establish a WebSocket connection. This persistent connection is set up between the user's interface and the Matching Service through the API Gateway, allowing for continuous two-way communication. When a user seeks a peer for practice, they send a match request via this WebSocket channel. The request is routed through the API Gateway to the Matching Service, which then queries the PostgreSQL database for suitable peer candidates. The Matching Service applies its internal logic to evaluate and identify a fitting match from the returned candidates. Once a match is found, the service communicates this information back up the chain — it travels via the WebSocket connection through the API Gateway and ultimately to the user's CodeParty UI. This real-time, bidirectional communication stream ensures immediate notification and response, significantly enhancing user experience by minimizing latency and keeping users fully informed of their match status.

4.3.3. Matching Service & Collaboration Service Interactions

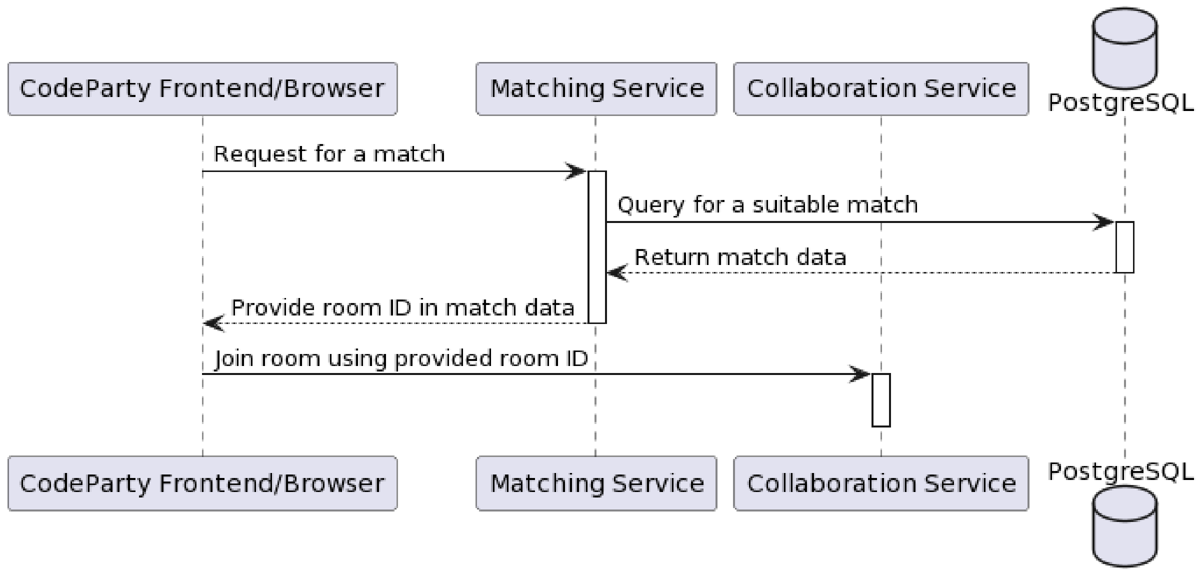

CodeParty Frontend/Browser
Matching Service
Collaboration Service
PostgreSQL
Request for a match
Query for a suitable match
Return match data
Provide room ID in match data
Join room using provided room ID
CodeParty Frontend/Browser
Matching Service
Collaboration Service
PostgreSQL

Diagram 5: Matching Service & Collaboration Service interaction - Client/Server

In this diagram, the CodeParty frontend client initially interacts with the Matching Service to retrieve the match data containing necessary IDs for the collaboration room and question. Once received, the client initiates a real-time session with the Collaboration Service, passing along these IDs. The Collaboration Service then establishes a real-time connection with the user's browser or application, using WebSockets.

Once this connection is established, users can share text in real time, with the Collaboration Service managing this interactive session. All users in the same collaboration room should be able to see updates live, reflecting the real-time nature of the service. This diagram simplifies the interaction for clarity, focusing on the sequence of events related to establishing and using the collaboration room.

4.3.4. Production Environment

For the production environment, we used Google Cloud in the following ways:

- Google Kubernetes Engine: for hosting the Kubernetes cluster
    - Google SSL Managed Certificates for SSL connections
- Google Cloud Domains and Cloud DNS: for registering a domain for the application and setting the DNS records
- Google Artifact Registry: for storing the Docker images for the services

The production workflow generally follows the below steps:

1. Developer makes a pull request from the master branch to the prod branch
    a. If merge conflicts are flagged, the master branch changes have higher priority
2. Upon passing Continuous Integration, the pull request is merged
3. Another run of Continuous Integration is done after the merge
4. Immediately after step 3 passes, a deployment workflow is carried out to apply the Kubernetes manifests on the GKE cluster

The Kubernetes cluster generally looks like this:

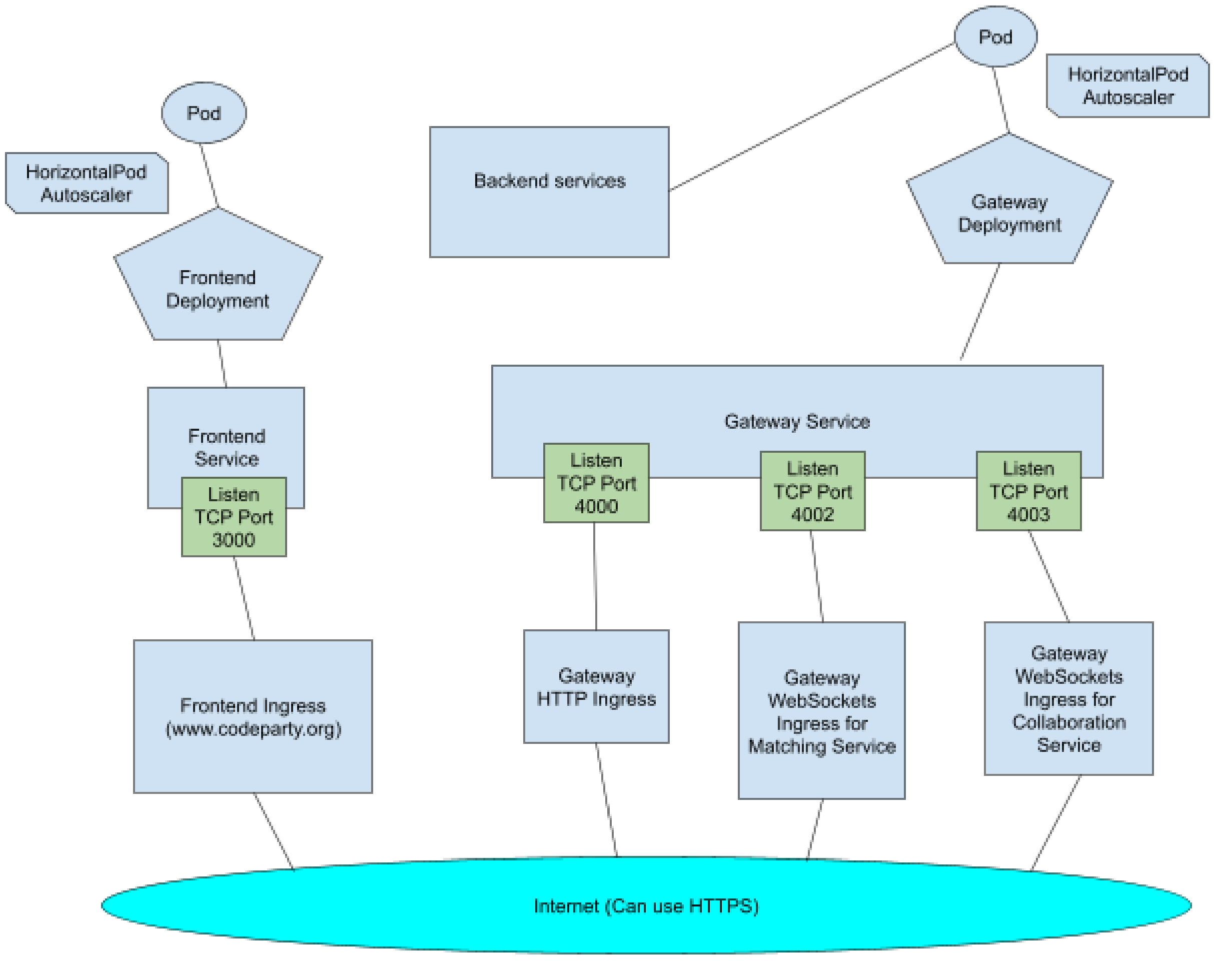


Diagram 6: Production Diagram

The use of Kubernetes Ingress allows the standard HTTP and HTTPS ports to be used (and mapped to the ports we used in development). Since we are using Google-provided Ingress Controllers for GKE, we can also use a Google-managed SSL certificate.

The external IP addresses that are assigned to each Ingress upon creating the cluster are also mapped to our domain and subdomains using DNS records.

4.3.4.1. Kubernetes Terminology

Each Service in Kubernetes serves as a point of connection for the respective microservice. The Service is itself attached to its respective Kubernetes Deployment, which manages the Pod configuration. Below is the sample code for the User Service as a Kubernetes Service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    io.kompose.service: user-service
  name: user-service
  namespace: default
spec:
  ports:
    - name: "5001"
      port: 5001
      targetPort: 5001
  selector:
    io.kompose.service: user-service
status:
  loadBalancer: {}
```

The line for “name: user-service” in the Service file is what allows the API Gateway to pass any User Service requests to “http://user-service:5001/”, instead of having to store the IP address. Kubernetes will direct the request to a Pod that is associated with the Deployment for the User Service.

4.3.4.2. How Horizontal Scalability is Achieved

Each Deployment has a HorizontalPodAutoscaler which spins up a new Pod when average CPU utilization reaches 50%. The sample code is shown below:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```
metadata:
  name: frontend-autoscaling
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend
  minReplicas: 1
  maxReplicas: 2
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Since running the production application on Google Cloud costs money, we are limiting the maximum number of pods to 2 (maxReplicas). Theoretically speaking, more pods can be created if the maxReplicas is increased.

4.3.5. Design Patterns

4.3.5.1. MVC Pattern

In our project, we have adhered to the Model-View-Controller (MVC) pattern both in our back-end, developed with Express, and on our front-end architecture. This design pattern has been fundamental in organizing our code, enhancing scalability, and improving maintainability.

Backend Implementation with Express:
On the server side, Express facilitated the implementation of the MVC pattern. The breakdown is as follows:

1. **Model**: Our models are responsible for managing the data and logic of the application. They interact with the database and process the data before sending it to the controller. By using Express, we could create models that efficiently handle data transactions, validations, and business logic.
2. **View**: In the context of our Express-based back-end, views are more about the representation of data. They are not traditional views like in front-end frameworks but are concerned with how the data is presented to the client. This could involve formatting data, deciding which data to send, and error handling presentations.
3. **Controller**: Controllers act as an intermediary between models and views. They receive client requests, invoke the appropriate model operations, and send back the response. In our Express application, controllers are responsible for handling

route logic, receiving HTTP requests, and sending the response after processing the data through models.

Frontend Implementation:

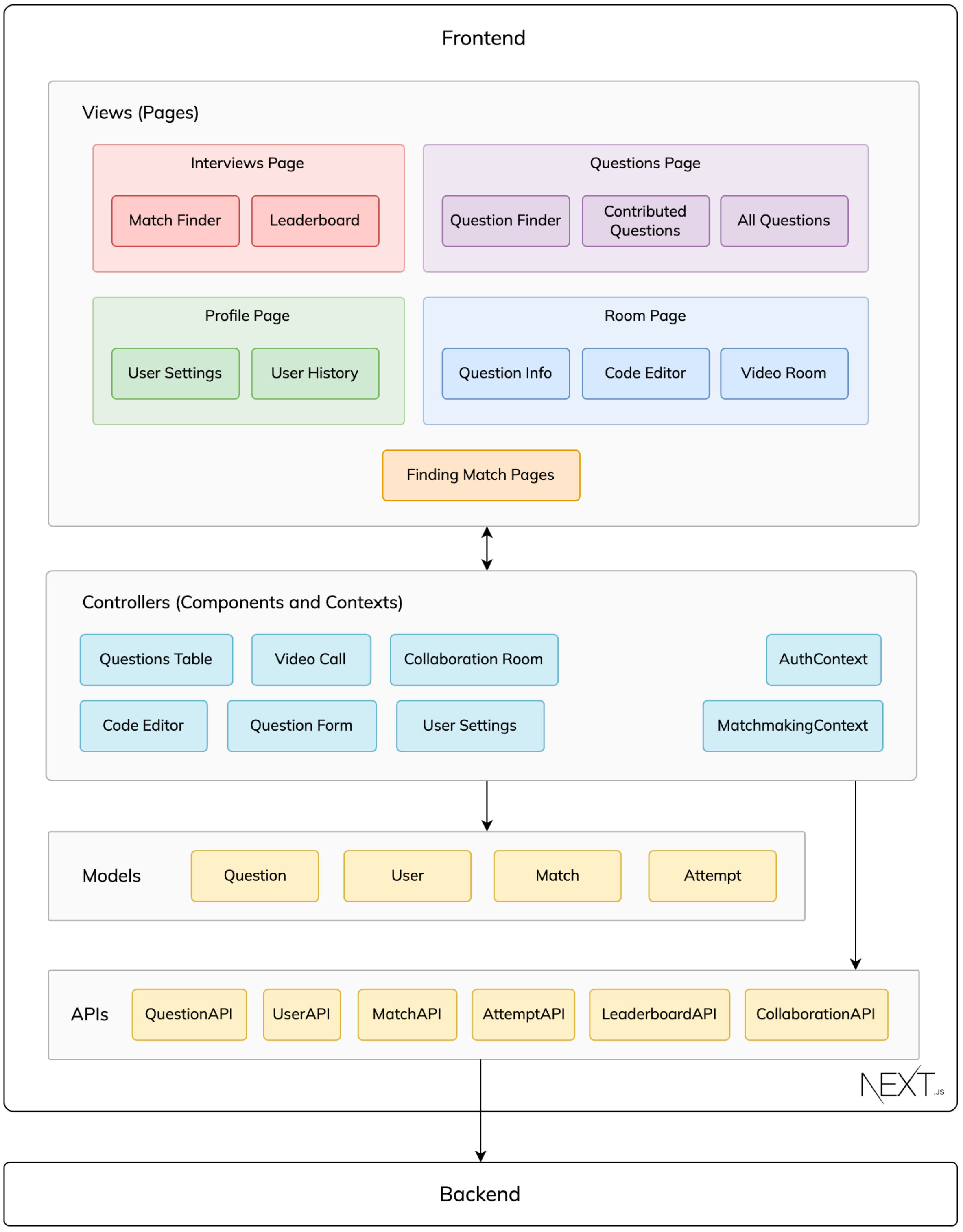


Diagram 7: Frontend/Client Architecture Diagram

On the frontend we have extended the MVC architecture to organize our user interface and user interaction logic:

1. **Model:** Similar to the back-end, the model on the front-end holds the business logic and data. However, in the front-end context, this often interfaces with the providers, managing the state of the application and ensuring that the view is updated when the data changes.
2. **View:** The views in our front-end are the actual components rendered to the users. They are designed to be stateless and are purely concerned with presenting the data to the user. The view interacts with the controller to send user actions (like clicks or input) to be processed.
3. **Controller:** In the front-end, the controller manages the interaction between the view and the model. It handles user inputs, communicates with the back-end (through APIs), processes data, and updates the model, which in turn updates the view.

4.3.5.2. Pub-Sub Pattern

We use web sockets (through socket.io) in our matching and collaboration services. Each user in a match is assigned a room to connect to. We can treat the users as the subscribers and the backend services to be the publishers. Each time the service needs to notify the users of an event, such as the match being found, or the match being deleted, the service would publish the event message on the web socket channel. Each user can receive the event message and proceed to handle the event logic. There is no fixed broker, since messages are directly sent between the users and the services.

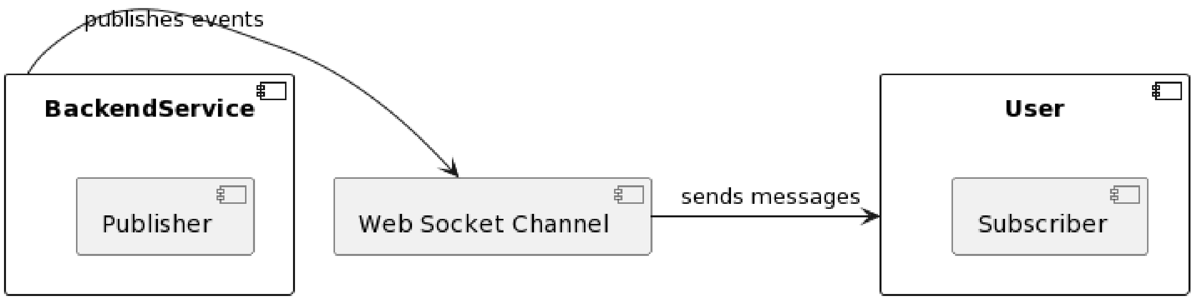


Diagram 8: PubSub Architecture Diagram

If we were to extend our application to support more than 2 users per match, the pub-sub pattern used would be ideal as it would require minimal change in code for the communication of events between the users and the services.

4.3.5.3. Asynchronous Pattern

We have used async patterns to enhance performance, user experience and scalability. These patterns have helped us in handling operations that require waiting for responses without blocking the user interface or other processes. The asynchronous patterns are applied across various modules:

1. **User Service:**

Asynchronous operations are extensively used in user profile management, particularly in handling database queries for user data retrieval and updates. For instance, when a user updates their profile, the request to the database is made asynchronously, allowing the user interface to remain responsive and providing real-time feedback once the operation is completed.

2. **Matching Service:**

The matching service involves complex algorithms to match users based on criteria such as difficulty level, topics, and proficiency. Asynchronous programming here is critical to ensure that these computationally intensive tasks do not block other user actions. This way, users can continue using other features of the app while the matching process is underway in the background.

3. **Question Service:**

Managing the question repository, especially when indexing questions by difficulty level and topics, involves database operations that benefit greatly from asynchronous patterns. For instance, retrieving a set of questions based on specific criteria is performed asynchronously, ensuring that the user interface remains interactive while the data is being fetched.

4. **Collaboration Service:**

Real-time collaboration, like concurrent code editing, requires immediate responsiveness. Asynchronous patterns are employed to handle the continuous data exchange between clients and the server. This allows multiple users to interact with the collaborative space simultaneously without experiencing delays or interruptions.

5. **UI:**

In the user interface, asynchronous calls are crucial for operations like form submissions, data retrieval for dashboards, and updates. This approach ensures that users can navigate through different parts of the UI without waiting for the completion of tasks like data loading or submission confirmations.

6. **Deploying the Application:**

In both local and staging environments, asynchronous operations help in managing deployment tasks. For instance, scripts for setting up the environment, loading initial data, or updating services are run asynchronously to optimize the deployment process and reduce downtime.

4.3.5.4. Synchronous Pattern

In addition to asynchronous patterns, our application also leverages synchronous patterns in specific scenarios where immediate, step-by-step execution is essential. These patterns are particularly important in processes where the outcome of one step directly influences the next.

Key Implementations of Synchronous Patterns:

1. **Sequential Data Processing:** In areas such as initial user setup or data migration, where each step depends on the completion of the previous one, we use synchronous patterns. This ensures data integrity and consistency, as each operation must complete before the next begins.
2. **Transaction Management:** For transactions, especially in the User Service and Question Service, synchronous processing is critical. It ensures that all parts of a transaction are completed successfully before moving on, thereby maintaining database consistency and avoiding partial updates.
3. **Error Handling:** In synchronous operations, error handling becomes more straightforward and predictable. If an error occurs in a sequence, the process halts, allowing for immediate identification and resolution of the issue.
4. **Simplicity in Logic:** For some functionalities, especially in the Basic UI module, synchronous patterns simplify the logic, making the code easier to understand and maintain. This is particularly beneficial for tasks that are inherently linear and do not require background processing.

By integrating synchronous patterns where they are most effective, we have ensured that our application can handle complex operations reliably, maintain data integrity, and provide a straightforward logic flow in certain aspects of the application. This balanced approach, using both synchronous and asynchronous patterns, optimizes performance and user experience.

4.3.5.5. Observer Pattern

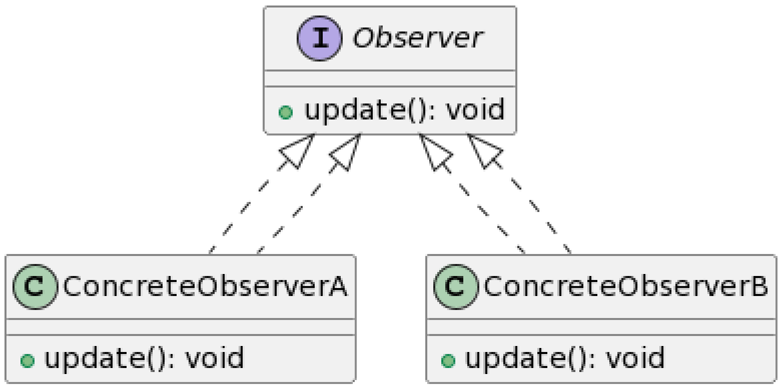


Diagram 9: Observer Pattern Diagram

In our application, the Observer pattern is integral to maintaining synchronization and real-time updates across various components. It's particularly utilized in services like User Service, Matching Service, and Collaboration Service. For instance, in User Service, changes in user profiles are observed and reflected in related components, ensuring up-to-date synchronization. In the Matching Service, this pattern allows for the dynamic updating of matching criteria based on user preferences. Similarly, the Collaboration Service uses the Observer pattern to enable real-time interactive features, such as concurrent editing, by observing and broadcasting changes to all participants. This pattern significantly enhances the responsiveness and consistency of our application, contributing to an efficient and user-friendly experience.

4.3.5.6. Adapter Pattern

The Adapter pattern is a structural design pattern that allows objects with incompatible interfaces to work together. In our application, we use the adapter pattern primarily when we want to work with our PostgreSQL database hosted externally. It would be complicated for the services to make connections to the database directly and issue raw SQL commands to achieve their objectives.
Therefore, we use a Prisma client as the adapter for our application to the database. The Prisma client exposes a simple interface for the services to call to perform common database actions such as the usual CRUD operations. Meanwhile, Prisma would convert the function calls into low level commands compatible with PostgreSQL.

Additionally, our application uses classes to represent each database record on a specific table in the PostgreSQL database. This is to simplify the way our application can interact with the data in the record. However, since the class would not work directly with PostgreSQL, the Prisma client also handles the conversion of the raw SQL output to the desired data type. In essence, it is an adapter for our application to the external PostgreSQL database.

4.3.5.7. Façade Pattern

The Facade design pattern is a structural design pattern that provides a simplified interface to a complex subsystem. It involves creating a facade class that wraps complex functionality to provide a simpler and more readable interface to clients. In our application, we use the Facade pattern in our microservices.

The question service is a facade to the frontend client for performing operations on the MongoDB database of questions. The MongoDB client library provides a wide variety of methods and configurations to use, as it should be to tailor to the needs of different users of MongoDB. However, our application does not need all the functionality of the MongoDB client, and it should not allow the frontend client to perform any operation on our MongoDB server. Furthermore, using the MongoDB client functions on the frontend client would cause unnecessary complexity to the frontend code. Therefore, we provide the question service to the frontend client. The question service contains a limited number of operations that the client can perform. It hides the complex logic of connecting to the database and performing operations. This would simplify the code needed in the frontend client to interact with the MongoDB question database/collection.

We can also see the same pattern used in the user service. The user service is a facade for the frontend to perform operations on application users. It provides limited functionality to the frontend client, thereby simplifying the interface for the frontend client. Within the user service, it will call the more complex logic of accessing the PostgreSQL database and perform other backend business logic.

4.3.6. Database Entity-Relationship Diagram

The following diagram shows the overall ER diagram of the PostgreSQL database used by our application. Note that we have excluded the question document in MongoDB, as MongoDB is not a SQL database and does not fit well in the ER diagram.

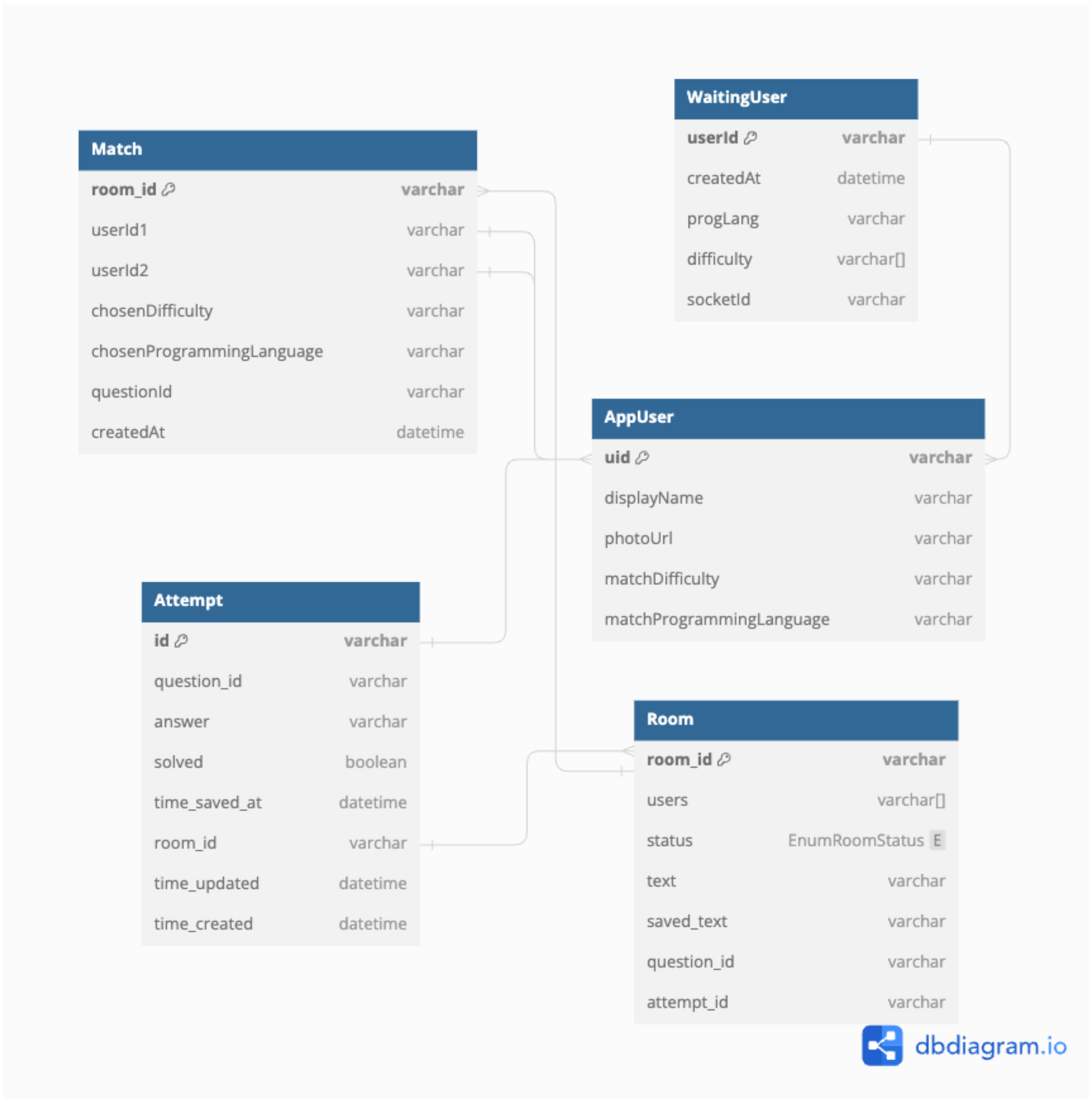


Diagram 10: ER Diagram of PostgreSQL database

4.4. Requirement Specific Design Decisions

4.4.1. M1: User Service

4.4.1.1. Purpose

The User Service is primarily designed to manage user-related data within the application. This includes handling user creation, updates, deletions, and retrieval. Additionally, it is responsible for managing user-specific data like match preferences, which includes difficulty levels and preferred programming languages.

4.4.1.2. Functionality

It provides a range of functionalities such as user account management, storing and retrieving user match preferences, and tracking user attempts in various challenges or tasks.

4.4.1.3. Implementation Details

- Database Interaction: Utilizes Prisma Client for interaction with the database. Prisma Client is an ORM (Object-Relational Mapping) tool that simplifies database operations.
- User Creation and Management:
  - Unique Identification: Each user is uniquely identified by a uid, which is used to create, update, or delete user profiles.
  - Data Validation: The service includes checks for existing users to prevent duplicate entries, ensuring data integrity.
- Match Preferences:
  - Customization: Allows users to set preferences for match difficulty and programming language, enhancing user experience by tailoring challenges to their skills and interests.
- Error Handling: Implements try-catch blocks for robust error handling, especially during database operations.

4.4.1.4. Design Choices

1. Modular Functions: Each functionality (create, read, update, delete, etc.) is encapsulated in separate asynchronous functions, adhering to the Single Responsibility Principle and making the code more maintainable.
2. Asynchronous Operations: The use of async/await for handling asynchronous database operations ensures efficient performance and better handling of concurrent requests.

4.4.1.5. Admin Service

The Admin Service is primarily used as a REST API wrapper for setting and removing admin permissions for the application users. Setting the admin is mainly done by setting a user's custom claim to { admin: true }. For removing admin privileges, this is done by setting the custom claim to null.

The custom claims themselves are stored remotely on Firebase. Hence, the Admin Service needs a FIREBASE_SERVICE_ACCOUNT environment variable to connect to our Firebase authentication service.

Steps:

1. Receive a request for '/setAdmin/{uid}' in routes/index.ts
2. Call setFirebaseUidAsAdmin(), which is in firebase-server/firebaseWrappers.ts
3. Call setUserClaimsWrapper() with { admin: true } as the custom claim
4. Call setCustomUserClaims() in Auth object (Firebase)

4.4.1.6. Authentication

We used Firebase and GitHub OAuth for authenticating users. The procedure for logging in is as follows:

1. User sets up a GitHub account
2. User logs in to the application through GitHub OAuth
3. For a successful login, if user is new, user is added to Firebase Authentication database for the app

For calling backend APIs, the requests pass through the API gateway for checking authentication (discussed in a later section). The Firebase API also allows usage of custom claims to grant administrator access to a user. (https://firebase.google.com/docs/auth/admin/custom-claims).

Data related to user authentication and authorization are stored remotely on Firebase.

4.4.2. M2: Matching Service

4.4.2.1. Purpose

The matching service is to allow the frontend client to look for a match based on a number of criteria, such as the programming language and the difficulties preferred. It is responsible for providing real-time updates to the user if there is a change in the match status.

4.4.2.2. Functionality

It provides a range of functionalities such as notifying both users if a match is found. It also sends messages and events to both users, such as if a user leaves the match, or if a user wants to change the question. It also accepts user input, such as leaving a match, canceling the intention to look for a match, and requesting a change question.

4.4.2.3. Implementation details

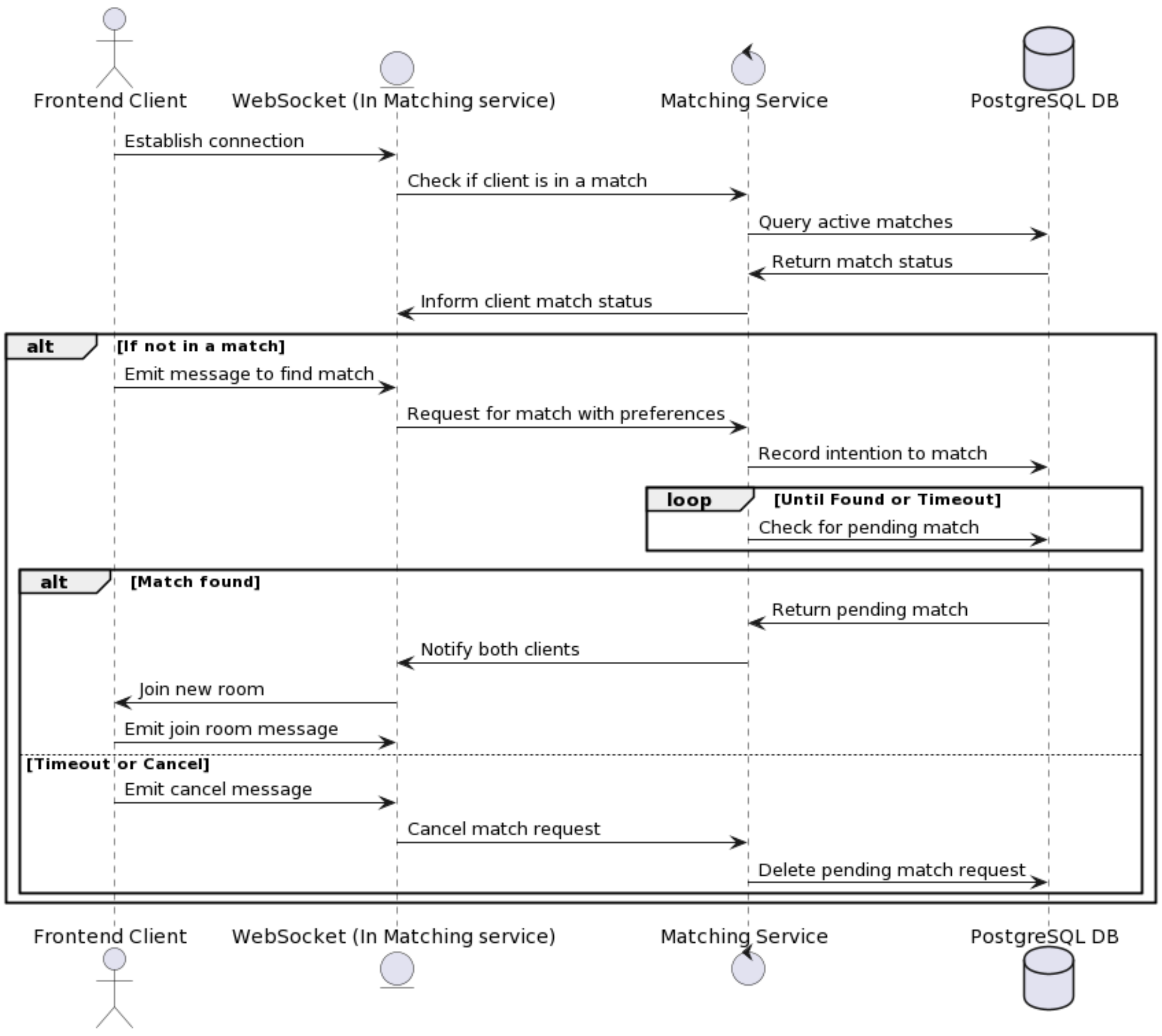


Diagram 11: Matching service sequence diagram

We use websockets to connect the frontend client and our backend server. Once the client establishes a connection, the server checks if the client is already in a match and informs the client if they are in one.

The client can then emit a message to the backend server to start looking for a match, along with their choice of programming language and question difficulties. The matching service will attempt to find a pending match request in the PostgreSQL WaitingUsers table. If no match is found yet, the user's intention to match will be recorded in the database. If a match is found, both users will be notified via the websocket channels and a new room will be created in PostgreSQL. Both users can then join the room by emitting another message to the server.

If a match cannot be found within a time limit (e.g. 30 seconds), or if the user chooses to cancel, the frontend client can emit a cancel message to the server. The server ensures that the record of the pending match request associated with the user will be deleted and no other user will be able to match with that user.

Also refer to 4.3.3 Matching Service & Collaboration Service Interactions for other interactions in detail

4.4.2.4. Design choices

1. Asynchronous Operations: The use of async/await for handling asynchronous database operations ensures efficient performance and better handling of concurrent requests.
2. Command pattern: We separate each command to the matching service socket using a function `handleXXX`. This allows for modularizing of code and makes each operation easier to understand.
3. Usage of database transactions: We use database transactions (through prisma.$transaction) to ensure the ACID properties of our operations on the database. This ensures that there will not be any concurrency bugs for our matching logic such as race conditions due to incorrect interleaving of operations in our server.

4.4.3. M3: Question Service

4.4.3.1. Purpose

The question service is to manage questions in the database and allow users to perform CRUD operations on them.

4.4.3.2. Functionality

It provides functionalities such as allowing users to get a list of questions (and specify filters and sorting criteria), create new questions, edit questions, delete questions, and get a random question.

4.4.3.3. Implementation Details

The question service uses MongoDB to store question documents. MongoDB was chosen because it is a document-oriented database and its schema is flexible. The structure of questions is not highly relational and the schema is constantly evolving during the development. Furthermore, MongoDB scales well with an increasing number of questions. We also create indexes to support efficient queries, such as an index on question difficulty, as it is very common to search questions by difficulty. Of course, we would need more questions in the database to see the benefits of using indexes.

We use MongoDB's new Atlas Search feature to search for questions based on title, so that some fuzzy searching or partial string matching is allowed. Another special operation is retrieving a random question based on difficulty and/or topics. This allows the matching service to choose a question for the match, and allows the user in the frontend to swap to a different question.

The API gateway (See 4.4.13) would filter away unauthenticated users from retrieving questions. Additionally, it would prevent normal users (non-admins) from sending PUT/POST/DELETE requests that would edit the questions on the MongoDB database.

4.4.3.4. Design Decision

Also refer to 4.3.2 System-Service Level for a brief diagram containing the interactions between question service and the rest of the app.

4.4.4. M4: Collaboration Service

4.4.4.1. Purpose

The primary purpose of the Collaboration Service is to enable seamless, real-time collaboration between multiple users, specifically focused on coding together in the same session. It aims to provide a dynamic, synchronized environment where users can engage in simultaneous code editing and interaction, fostering teamwork and enhancing productivity.

4.4.4.2. Functionality

- **Real-Time Code Editing:** Users can edit code in real-time, with changes instantly visible to all participants in the session.
- **Session Management:** The service manages collaborative sessions, ensuring that only authorized users can join and interact.
- **Change Synchronization:** Any change made by a user is quickly propagated to all other users in the session, keeping everyone's view consistent.
- **WebSocket Communication:** Utilizes WebSocket for low-latency, bi-directional communication between clients and the server.

4.4.4.3. Implementation Details

- **WebSocket Connections:** Users (User A and User B) establish a WebSocket connection through the API Gateway to the Collaboration Service. This connection is maintained throughout the session for continuous real-time interaction.
- **Change Propagation:** When a user edits code, the change is sent to the Collaboration Service. The service then immediately forwards this change to the other user, ensuring that both users' interfaces are in sync.
- **Concurrency Control:** The service incorporates mechanisms to handle concurrent edits effectively, minimizing conflicts and loss of data.
- **Session Isolation:** Each collaborative session is isolated to ensure that changes made in one session do not affect others.

Reference diagram 12 for more information.

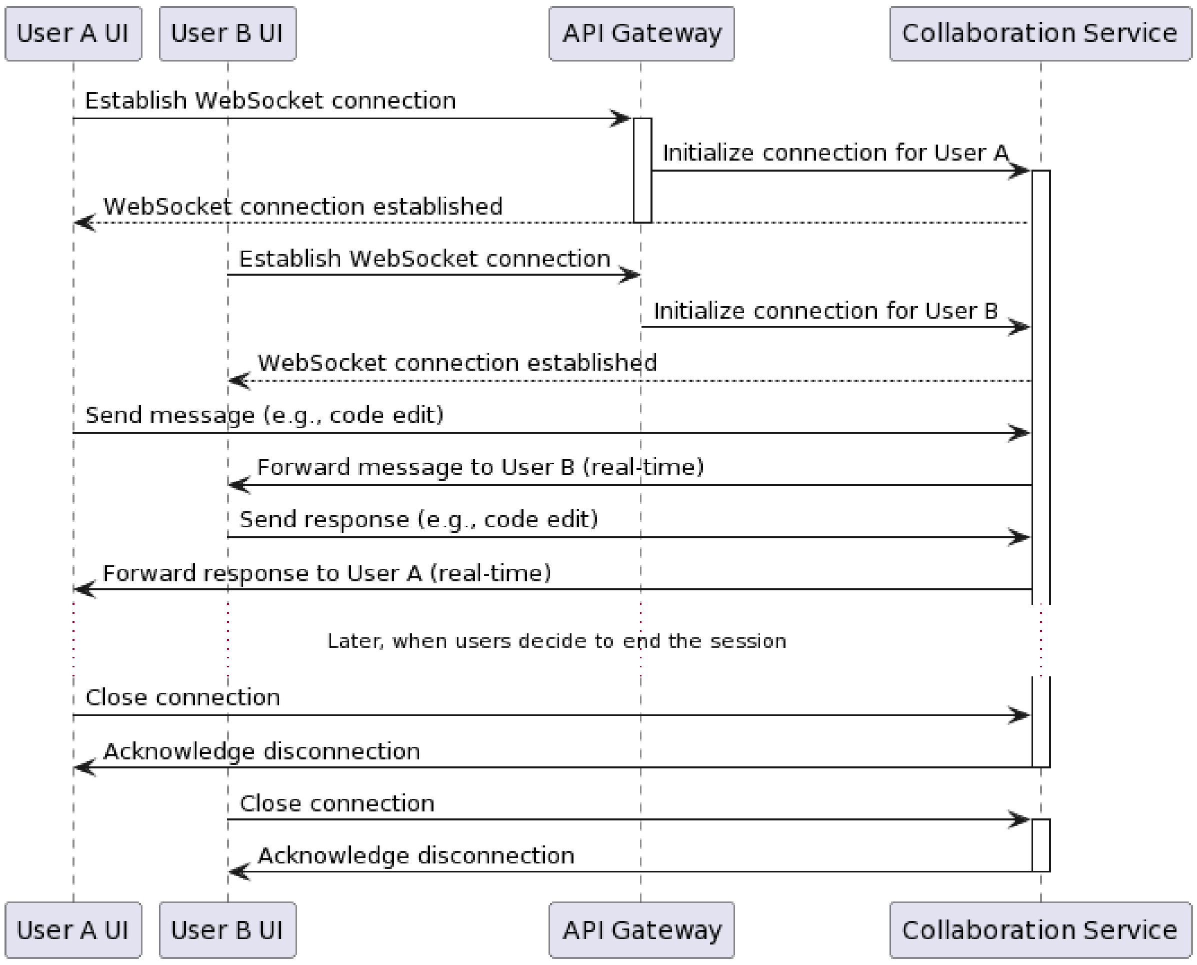

User A UI
User B UI
API Gateway
Collaboration Service
Establish WebSocket connection
Initialize connection for User A
WebSocket connection established
Establish WebSocket connection
Initialize connection for User B
WebSocket connection established
Send message (e.g., code edit)
Forward message to User B (real-time)
Send response (e.g., code edit)
Forward response to User A (real-time)
Later, when users decide to end the session
Close connection
Acknowledge disconnection
Close connection
Acknowledge disconnection

Diagram 12: Matching Service & Collaboration Service interaction - Between 2 users

In the above diagram, "User A" and "User B" represent two different users. Both establish a WebSocket connection through the API Gateway to the Collaboration Service, allowing for real-time communication. When one user makes a change (like editing code), this is sent to the Collaboration Service, which then immediately forwards the change to the other user, ensuring both users' UIs are synchronized.

4.4.4.4. Design Decision

- **Use of WebSockets:** Chosen for their ability to facilitate real-time, two-way communication between the server and clients, essential for a fluid collaborative experience.
- **API Gateway Integration:** This approach centralizes the entry points for the services, simplifying the architecture and enhancing security.
- **Focus on Low Latency:** Ensuring minimal delay in synchronization was paramount to provide a smooth, interruption-free coding experience.
- **Scalability Considerations:** The design supports scaling to accommodate a growing number of simultaneous collaborative sessions.

4.4.5. N1: Communication

4.4.5.1. Purpose

The Communication component within the Collaboration Service is designed to offer an additional layer of interaction among participants beyond the shared coding workspace. Its primary purpose is to facilitate more nuanced and direct communication, such as through text-based chat and video (+voice) calling services.

4.4.5.2. Functionality

Video and Voice Calling Service: Integrates a real-time video and voice communication feature, providing a more personal and immediate way to collaborate and discuss.

4.4.5.3. Implementation Details

Refer to diagram 13 below to learn how the communication via Twilio is integrated into the app. Also refer to 4.4.9.3. Other additional feature: Operational Transform for more details on how twilio fits into collaboration service

- **Use of Twilio:** We have chosen Twilio as the backbone for implementing these communication features due to its robust API, reliability, and ease of integration.
- **Integration with Collaboration Service:** The communication features are embedded within the existing Collaboration Service, ensuring a unified experience.
- **Real-Time Interaction:** Both chat and video/voice services are designed for real-time communication, minimizing latency and enhancing user engagement.
- **Security and Privacy:** Ensuring secure and private channels for communication, keeping the participants' interactions confidential.

4.4.5.4. Design Decision

- **Choosing Twilio:** Selected for its comprehensive suite of communication APIs, offering reliable and scalable solutions for both chat and video/voice services.
- **Focus on Real-Time Communication:** Emphasizing the importance of immediate and fluid communication in collaborative work, especially in coding sessions.
- **User-Friendly Interface:** The design prioritizes ease of use, ensuring that participants can effortlessly access and utilize both chat and video/voice features.
- **Integration with Existing Services:** The communication services are designed to work in tandem with the collaborative coding environment, providing a holistic collaboration experience.

4.4.6. N2: History

4.4.6.1. Purpose

The History component within our Collaboration Service is dedicated to recording and maintaining a comprehensive history of user activities, particularly focusing on the questions attempted by users. This feature is designed to track and store details of each attempt, including the date and time, the attempt itself, and any suggested solutions.

4.4.6.2. Functionality

- **Attempt Tracking:** Captures and logs each question attempt made by users in the collaborative space, recording specifics like attempt details and outcomes.
- **Timestamps:** Every attempt is timestamped, providing a clear historical context for when each question was attempted.
- **Easy Access to History:** Users can easily access and review their historical attempts, facilitating self-assessment and continuous learning.

4.4.6.3. Implementation Details

Refer to diagram 13 below to learn how the history component is integrated into the app.

- **Integration with User Service:** The history tracking is closely integrated with the User Service, ensuring that all user attempts and interactions are accurately captured and stored.
- **Use of the useCollaboration Hook:** This custom hook manages room joining and question context setup, crucial for capturing the history of user attempts.
- **Saving Attempts on Disconnection:** The system automatically triggers a save to the User Service when users disconnect, ensuring no data is lost.

4.4.6.4. Design Decision

- **Comprehensive Tracking:** Decided to maintain an extensive history to support users in revisiting and learning from their past attempts.
- **Seamless Integration with Collaboration Activities:** Ensuring that history tracking is a natural and unobtrusive part of the collaborative experience.
- **Automated Data Capture:** Automated capture of attempts and solutions to reduce manual overhead for users and ensure data accuracy.
- **User-Centric Design**: The design focuses on providing users easy and intuitive access to their history, enhancing their learning and review process.

4.4.7. N1 + N2: Illustration on how they are integrated

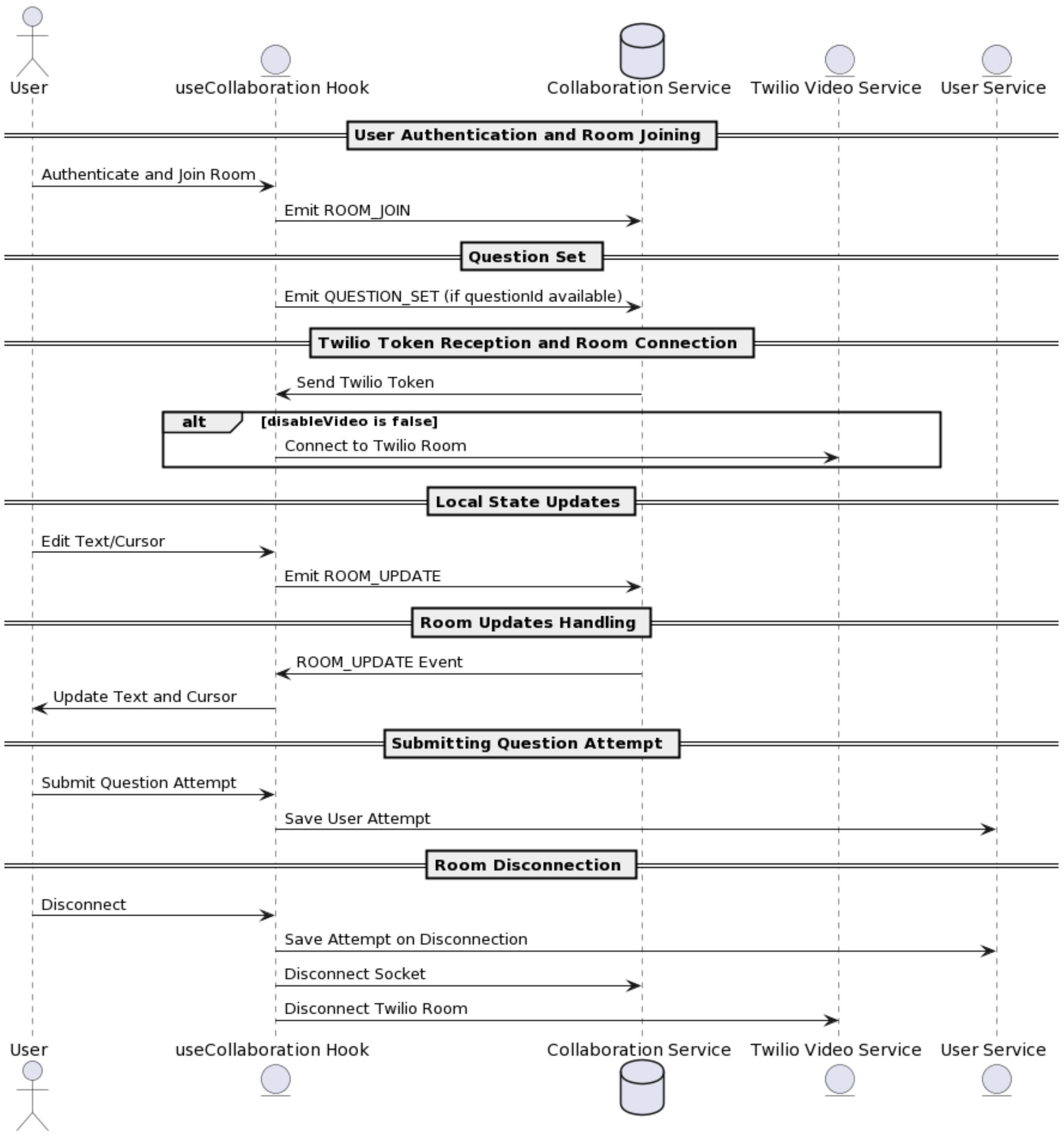


Diagram 13: Real Time Collaboration with History Tracking

The provided sequence diagram provides a comprehensive overview of the interactions within a collaborative system, specifically focusing on real-time communication, question handling, and history tracking. The process begins with user authentication and joining a collaborative room, facilitated by the useCollaboration hook. This hook manages room joining events and sets up question contexts if available. An important aspect of the system is the integration with Twilio's video service, which is conditionally initiated based on the video settings. Users can edit text or cursor positions in real-time, with updates being synchronized via the Collaboration Service. Additionally, the diagram incorporates the User Service, which plays a crucial role in history tracking. When a user submits a question attempt, the useCollaboration hook sends this data to the User Service to save the attempt. Furthermore, when users disconnect from the room, the hook triggers the User Service to save the attempt, ensuring that all collaborative activities and user inputs are accurately recorded and preserved. This sequence diagram thus encapsulates the key functionalities of real-time collaboration, user interaction, and history management in the system.

4.4.8. N4: Enhance Question Service

4.4.8.1. Overview

The enhancement of the question service in the application focuses on enabling dynamic management and retrieval of questions. This includes functionalities like tagging questions by various criteria (topic, popularity, etc.), and efficiently retrieving questions during session initiation.

Our question service enhancement includes the following additions on top of the generic question service:

1. Tagging by topic and difficulty on question creation
2. Filtering by tags while looking for questions
3. Ability to swap to random questions when in solo or collaborative practice modes
4. Fuzzy search for questions in the table
5. Pagination for when the question list gets too long
6. Multi Table management for admins and users.

4.4.8.2. Design Decisions

1. Dynamic Question Retrieval:
   - **Functionality:** Incorporates a feature to fetch a random question based on selected difficulty, enhancing user engagement and providing a diverse learning experience.
   - **Implementation:** Utilizes fetchRandomQuestion function from useQuestions hook, demonstrating effective use of React's custom hooks for encapsulating logic.

2. Difficulty Selection:
   - **Component:** A DifficultySelector component allows users to filter questions based on difficulty levels (e.g., easy, medium, hard), catering to a wide range of user competencies.
   - **User Interaction:** A dropdown selector enables users to choose the difficulty level, enhancing usability and providing a personalized experience.

3. Data Table Integration:
   - **Purpose:** Incorporates DataTable components to display questions, enabling efficient management and browsing of questions.
   - **Customization:** The data table is customizable based on user role (isAdmin), allowing for different views and interactions for administrators and regular users.

In this case, only the admin can edit and make changes to questions while users are only viewers.

4. Contextual Data Handling:
    - **User Context**: Utilizes AuthContext to manage user authentication state, ensuring secure access and role-based features (like contributing new questions).
    - **Routing:** Leverages Next.js' useRouter for navigation, enhancing user experience by seamlessly directing users to specific questions or contribution pages.

4.4.8.3. Addressing Non-Functional Requirements

Usability:

- Intuitive Interface: The UI is designed to be user-friendly, with clear instructions and easy navigation, enhancing the overall user experience.
- Responsive Design: Ensures a seamless experience on different devices, accommodating a diverse user base.

4.4.9. N5: Enhance Collaboration Service

4.4.9.1. Overview

The enhancement of the collaboration service focuses on integrating an advanced code editor. This editor features code formatting and syntax highlighting for multiple programming languages, significantly improving the usability and collaborative experience for users.

4.4.9.2. Design Decisions

1. Choice of Editor Component:
    - Component Used: The Monaco Editor, known for its reliability and extensive feature set, is used as the base for our custom code editor.
    - Rationale: Monaco Editor provides out-of-the-box syntax highlighting and code formatting for various languages, making it an ideal choice for a feature-rich coding environment.
2. Language Support:
    - Supported Languages: Initially, the editor supports Python, Java, and C++, with plans to extend this range based on user demand and usage patterns.
    - Implementation Approach: Language support is modular, allowing for easy extension to include additional programming languages in the future.
3. Code Editor Props:
    - Flexibility: Props such as theme, language, and height are customizable, allowing the editor to be tailored to different user preferences and use cases.
    - Event Handling: Props like onChange and onCursorChange ensure that changes in the editor are captured and processed, facilitating real-time collaboration and data synchronization.
4. State Management:

- Local State: States help to manage the current programming language and submission status, ensuring a responsive and interactive user interface.
- Editor State: The Monaco instance state is managed to handle cursor positions and text changes within the editor efficiently. (This allows it to work together with the current collaboration service)

4.4.9.3. Other additional feature: Operational Transform

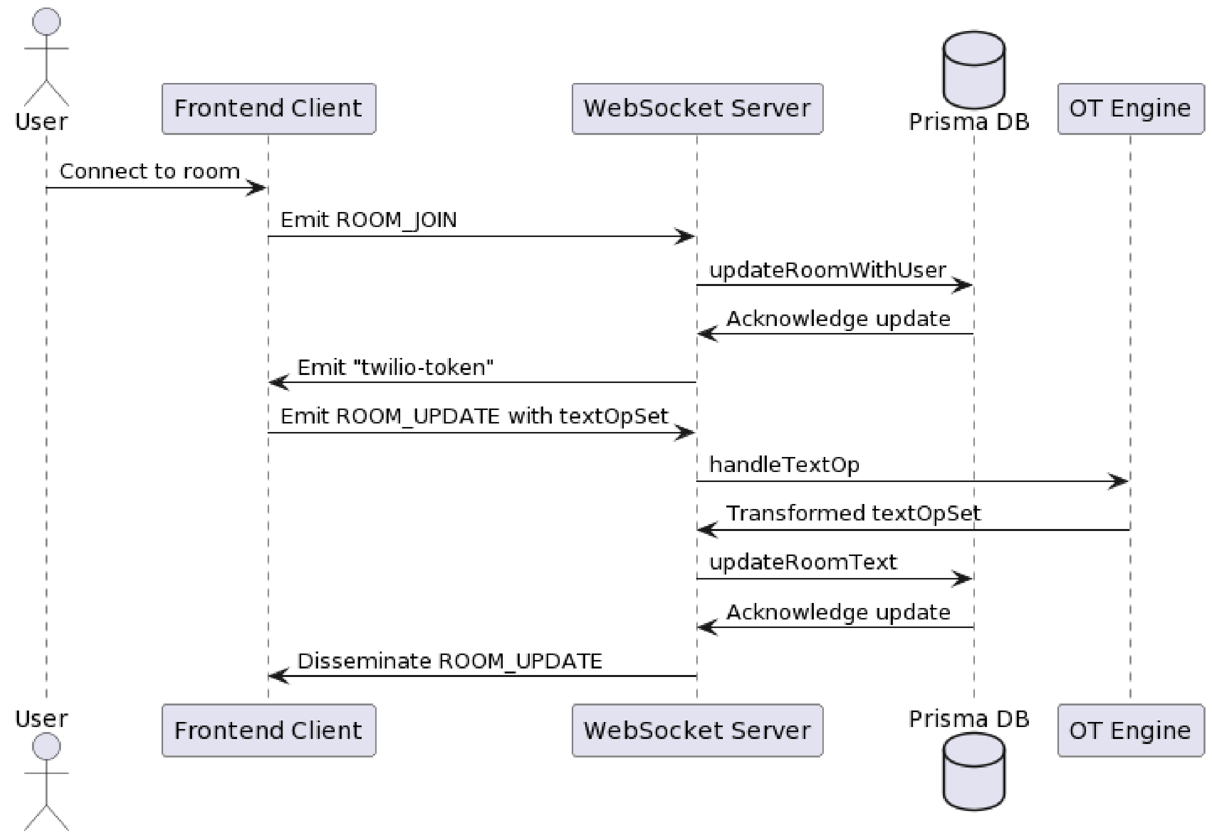


Diagram 14: Code flow for collaboration service

Also reference 4.3.3 Matching Service & Collaboration Service Interactions for more details on collaboration service.

We use WebSockets to create a connection between each user's frontend client and our backend server. We emit events from client to server to update the collaboration room they are in, and disseminate that event to all other users in the same collaboration room to update the text they see on their code editor.

Operational Transforms[1] are used to handle collisions when users' edits conflicts due to latency of any user. To prevent overwriting the previous user's edits, we use operational transforms to combine multiple edits so both users' intentions are preserved and applied.

[1] https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/

We save the collaboration as an attempt linked to all the users involved in the collaboration room, when all users disconnect from the room. If the users reconnect to that room, they will update that existing attempt.

4.4.9.3.1 Design of Operational Transform

Client sends text edits to server as a set of operational transforms and the previous version number given by the server. The client then accumulates further edits until receiving a response from the server. The server checks the version number provided and does an operational transform if the last version number is not the most recent version number. After applying an operational transform if necessary to produce a final text, the version number is incremented by the server, and sent to the other client in the room. The server also syncs to the client to allow the client to send new edits again.

4.4.9.4. Addressing Non-Functional Requirements

Usability:

- Intuitive UI: The layout and design of the editor are user-friendly, with clear icons and easy navigation, making it accessible for users with varying levels of programming expertise.
- Responsive Design: The editor's responsive design ensures a seamless experience across different devices and screen sizes.

Maintainability:

- Component-Based Architecture: The use of modular components like Popover, Button, and Card simplifies maintenance and future enhancements.
- Code Reusability: Common UI elements are reused across the application, reducing code duplication and facilitating easier updates.

4.4.10. N8: Extensive CI/CD Pipeline

Refer to section 4.2.6. CI/CD Pipeline for more information on our CI/CD flow. And refer to sections 4.5.3. Testing for how we performed testing.

4.4.11. N9: Deployment of App

Refer to section 4.3.4. Production Environment for more information on our deployment. We have the app deployed on https://www.codeparty.org/ - feel free to take a look (while we still have google credits)

4.4.12. N10: Scalability

Refer to section 4.3.4.2 How Horizontal Scalability is Achieved for more information on how we achieve scalability.

4.4.13. N11: API Gateway

We used an API Gateway to proxy requests between the frontend and the backend services. This allows us to use a single external-facing domain for the backend services in the production environment.

While we generally followed a tutorial by Janssen (2021) for the basic Gateway structure, we have adapted the design to suit our application.

The API Gateway has custom logic for verifying (also view 4.4.14. Authentication) whether the request is made by a logged in user as well as whether that user has admin rights. This is done by connecting to and verifying against the Firebase Authentication database. The Firebase connection is made using a Service Account on Google.

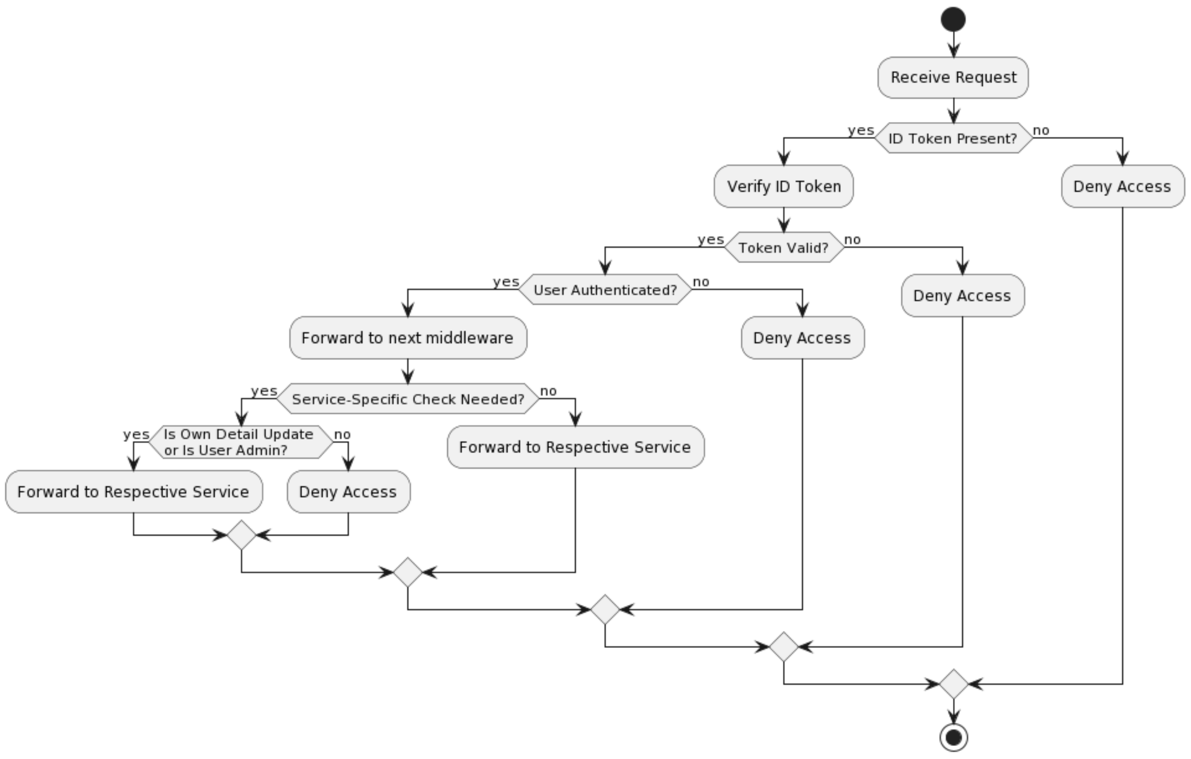

Receive Request
ID Token Present?
yes
no
Verify ID Token
Deny Access
Token Valid?
yes
no
User Authenticated?
yes
no
Deny Access
Forward to next middleware
Deny Access
Service-Specific Check Needed?
yes
no
Is Own Detail Update or Is User Admin?
yes
no
Forward to Respective Service
Forward to Respective Service
Deny Access

Diagram 15: Activity Diagram showing the flow of API gateway

What happens when a request passes through the API Gateway:

1. Request is checked for the presence of an ID token corresponding to the user.
2. ID token is verified - user is considered authenticated if the ID token is valid and not revoked
    a. If user is not properly authenticated, do not pass the request on
    b. The checks in steps 1 and 2 are done using an Express middleware function. If both pass, the request is forwarded to the next middleware
3. Additional checks (depending on the service):
    a. Check if user is updating or deleting their own details (and not those of any other user) on user-service OR

    b. Check if user is admin for creating, updating or deleting questions on the question-service
4. If the additional check passes (or there is no additional check), forward request to respective service

4.4.14. Authentication

We used Firebase and GitHub OAuth for authenticating users. The procedure for logging in is as follows:

1. User sets up a GitHub account
2. User logs in to the application through GitHub OAuth
3. For a successful login, if user is new, user is added to Firebase Authentication database for the app

The setting up of the project on Firebase as well as managing authentication generally followed a tutorial by Purwar (2022).

For calling backend APIs, the requests pass through the API gateway for checking authentication (discussed in a later section). The Firebase API also allows usage of custom claims to grant administrator access to a user. (https://firebase.google.com/docs/auth/admin/custom-claims).

Data related to user authentication and authorization are stored remotely on Firebase. This means that the User Service does not have passwords, admin roles or any sensitive details.

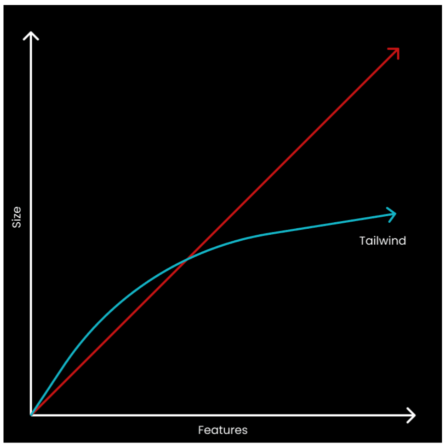
4.4.15. Real-time Collaboration

We used WebSocket protocol for real-time, bidirectional communication. The WebSocket protocol provides a simultaneous two-way communication channel for transmitting text. We establish a connection between each client to the server, such that when the text of each collaboration service is updated, this update can be disseminated to the other client in the same room and update the display on all clients in the same collaboration room. This WebSocket implementation was essential for the Collaboration Service to enable multiple users to edit code simultaneously.

4.4.16. Frontend Design

Our choice of framework is Next.js because it provides built-in support for server-side rendering, which improves page load times and optimizes search engine visibility. This helps in smoother collaboration experience and better question discovery.

For styling, we use Tailwind CSS, as it provides atomic classes that break down styles into small, single-purpose, reusable classes. This speeds up development as it increases readability and maintainability.



Shadcn/ui was our selected component library, as it allows us to pick only the components we need to avoid bloating and can customize it based on our brand colors.

Careful thought was also put in visually designing the frontend with usability in mind, so that page states and errors are made clear to the user, and they can use the app effortlessly.

To provide users with more flexibility in writing question descriptions, we allow them to use plain text, markdown or html formats. However, injecting the html received into the dom directly can make our app vulnerable to cross-site scripting (XSS) attacks. Hence, we sanitize the received description and convert the markdown/html into React JSX components using helpful external libraries such as `sanitize-html` and `react-markdown`.

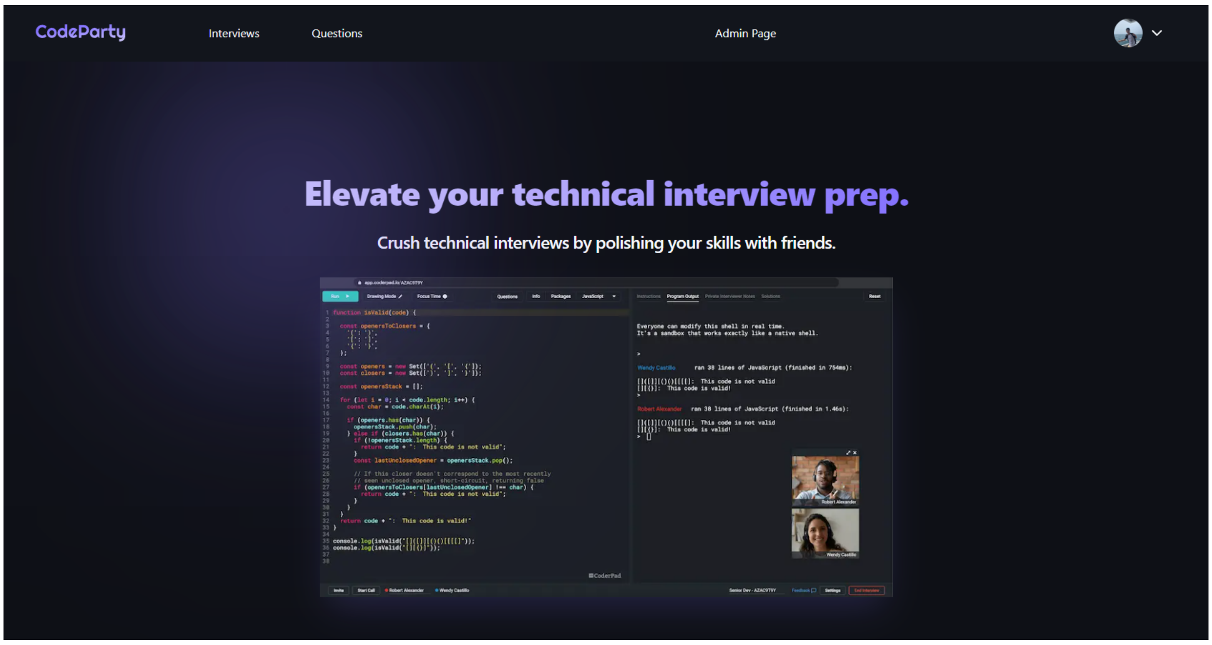
Some other notable helper packages that we used are as follows:

1. @mui/material: This is a UI library that provides React components following Google's Material Design guidelines.
2. @tanstack/react-query: This library is used for fetching, caching, and updating data in React applications. It's significant for managing server-state in a React app, which is crucial for data fetching logic.
3. @tanstack/react-table: A lightweight, flexible, and extendable data grid library for React. It's essential for managing and displaying large datasets in a tabular format with sorting, filtering, and pagination.
4. firebase: Firebase provides a suite of tools for building scalable web and mobile applications. It includes functionalities like authentication, real-time database, analytics, and more.
5. tailwindcss: A utility-first CSS framework used for styling web applications. It allows developers to rapidly build custom designs without leaving the HTML.
6. socket.io-client: Allows the application to have real-time, bidirectional event-based communication functionalities, for features like live updates.
7. react-hook-form: This is a library for managing forms in React. It helps with validation and handling of form states, making form handling easier and more efficient.
8. @monaco-editor/react: A React wrapper for the Monaco Editor, the code editor that powers Visual Studio Code. It provides advanced text editing features, crucial for applications involving code editing or viewing.
9. toastify: A customizable library for showing notifications in a React application. Enhances user experience by providing immediate and engaging feedback for user actions.

4.5. Other Development Artifacts

4.5.1. Screenshot Gallery

Image 1: Landing page



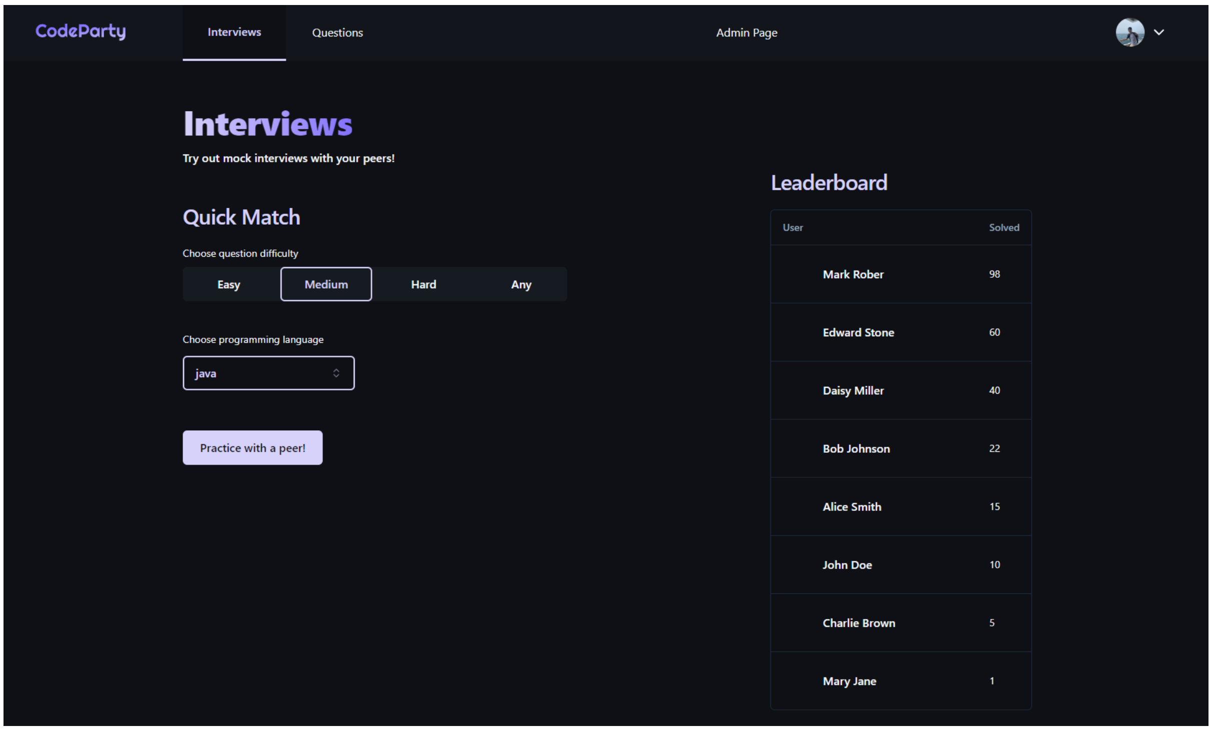
Elevate your technical interview prep.

Crush technical interviews by polishing your skills with friends.

```
function isValid(code) {

  const openersToClosers = {
    '(': ')',
    '[': ']',
    '{': '}',
  };

  const openers = new Set(['(', '[', '{']);
  const closers = new Set([')', ']', '}']);

  const openersStack = [];

  for (let i = 0; i < code.length; i++) {
    const char = code.charAt(i);

    if (openers.has(char)) {
      openersStack.push(char);
    } else if (closers.has(char)) {
      if (!openersStack.length) {
        return code + ":  This code is not valid";
      }
      const lastUnclosedOpener = openersStack.pop();

      // If this closer doesn't correspond to the most recently
      // seen unclosed opener, short-circuit, returning false
      if (openersToClosers[lastUnclosedOpener] !== char) {
        return code + ":  This code is not valid";
      }
    }
  }
  return code + ":  This code is valid!"
}

console.log(isValid("[]([]][()()[[[[]"));
console.log(isValid("[][{}]"));
```

```
Everyone can modify this shell in real time.
It's a sandbox that works exactly like a native shell.


>

Wendy Castillo     ran 38 lines of JavaScript (finished in 754ms):

[]([]][()()[[[[]:  This code is not valid
[][{}]:  This code is valid!
>

Robert Alexander   ran 38 lines of JavaScript (finished in 1.46s):

[]([]][()()[[[[]:  This code is not valid
[][{}]:  This code is valid!
>
```

Image 2: Interviews page



Interviews

Try out mock interviews with your peers!

Quick Match

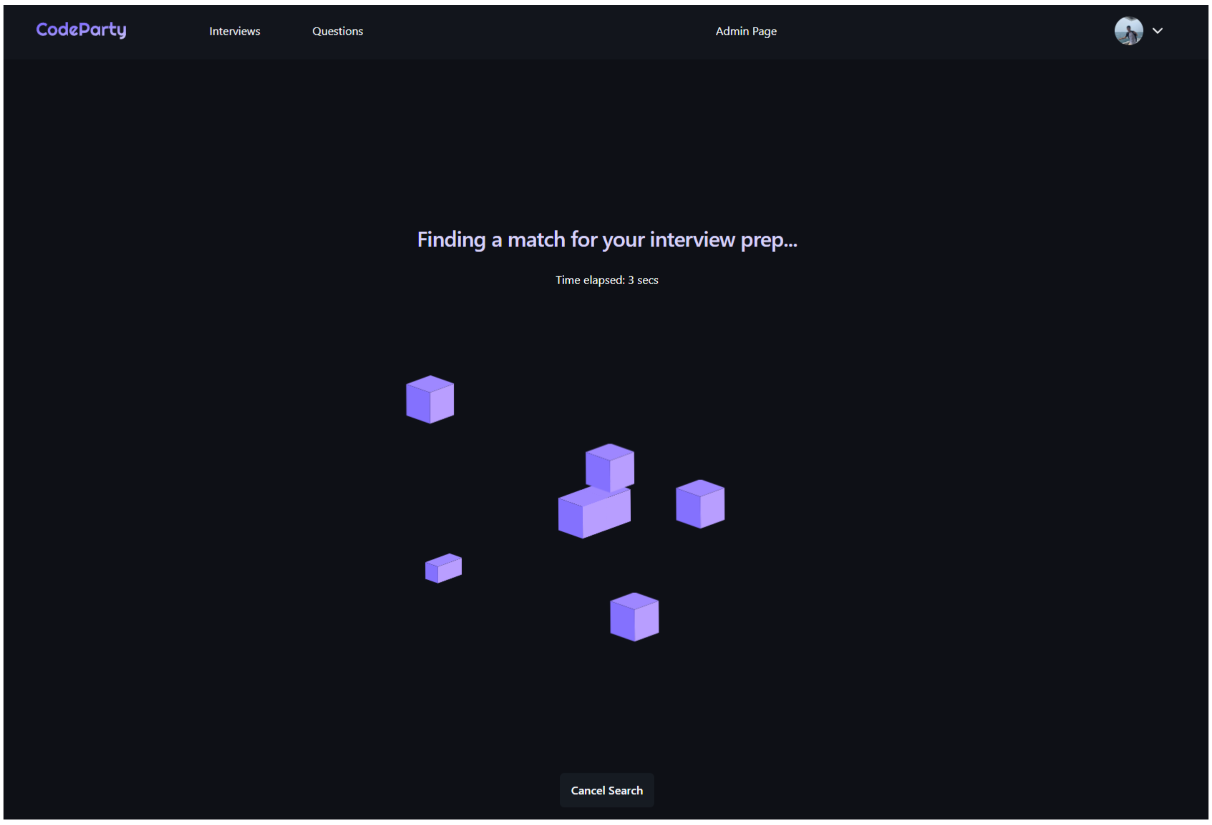
Choose question difficulty: Easy [✓ Medium] Hard Any

Choose programming language: java

Leaderboard

User	Solved
Mark Rober	98
Edward Stone	60
Daisy Miller	40
Bob Johnson	22
Alice Smith	15
John Doe	10
Charlie Brown	5
Mary Jane	1

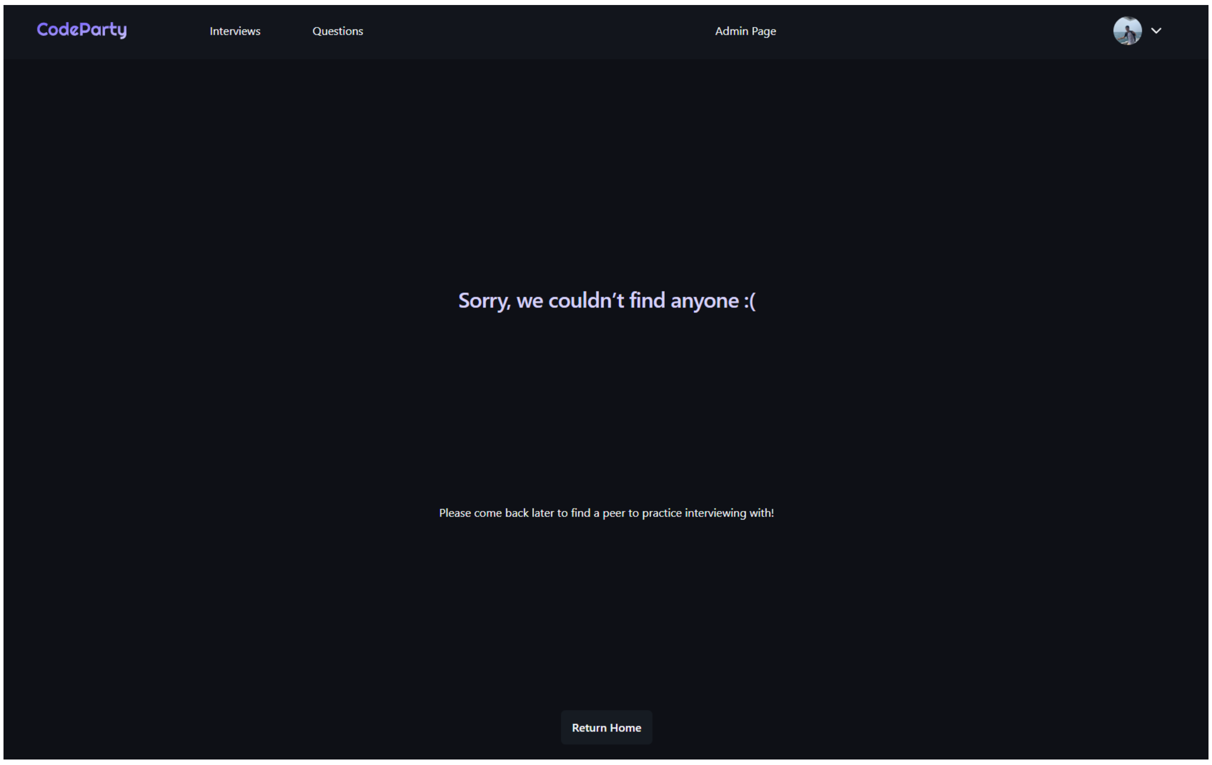
Image 3: Finding Match Page



Finding a match for your interview prep...

Time elapsed: 3 secs

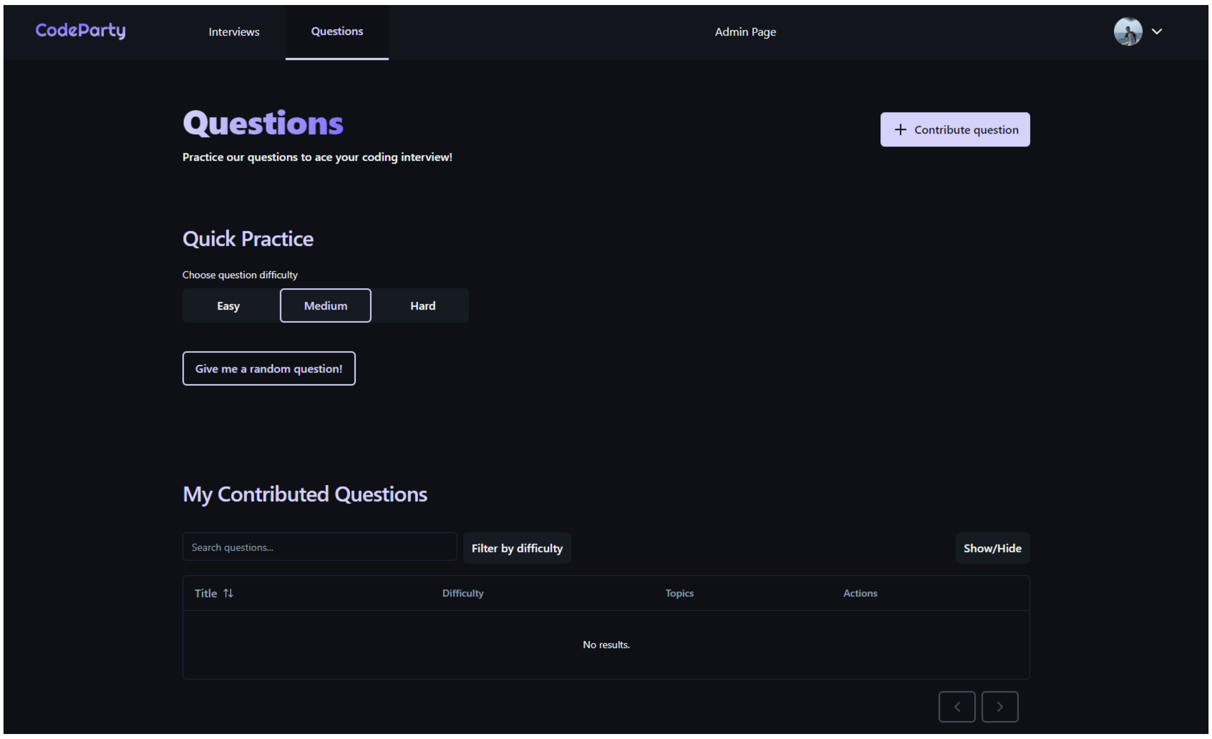
Image 4: Match not found page



Sorry, we couldn’t find anyone :(

Please come back later to find a peer to practice interviewing with!

Image 5-6: Questions Page



Questions

Practice our questions to ace your coding interview!

+ Contribute question

Quick Practice

Choose question difficulty

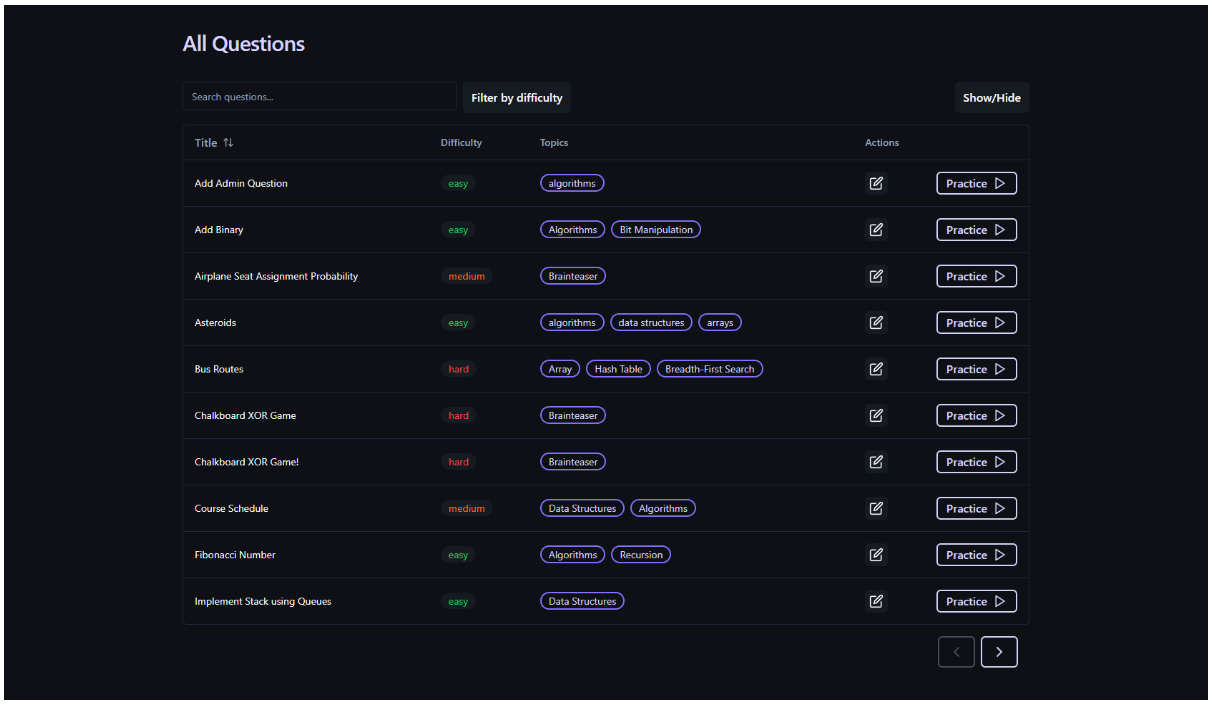
Easy | [✓ Medium] | Hard

Give me a random question!

My Contributed Questions

Search questions...   Filter by difficulty   Show/Hide

Title ⇅	Difficulty	Topics	Actions
No results.			

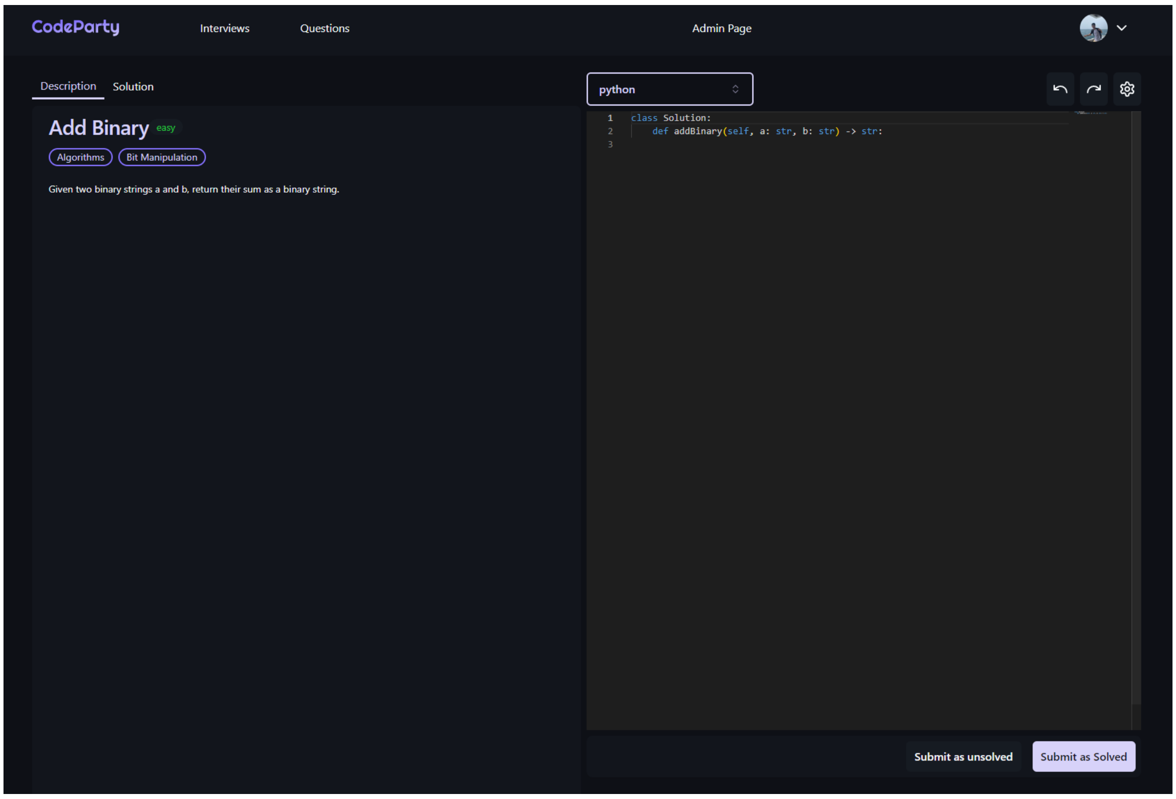


All Questions

Search questions...   Filter by difficulty   Show/Hide

Title ⇅	Difficulty	Topics	Actions	
Add Admin Question	easy	algorithms	✎	Practice ▷
Add Binary	easy	Algorithms, Bit Manipulation	✎	Practice ▷
Airplane Seat Assignment Probability	medium	Brainteaser	✎	Practice ▷
Asteroids	easy	algorithms, data structures, arrays	✎	Practice ▷
Bus Routes	hard	Array, Hash Table, Breadth-First Search	✎	Practice ▷
Chalkboard XOR Game	hard	Brainteaser	✎	Practice ▷
Chalkboard XOR Game!	hard	Brainteaser	✎	Practice ▷
Course Schedule	medium	Data Structures, Algorithms	✎	Practice ▷
Fibonacci Number	easy	Algorithms, Recursion	✎	Practice ▷
Implement Stack using Queues	easy	Data Structures	✎	Practice ▷

Image 7: Questions Code Page



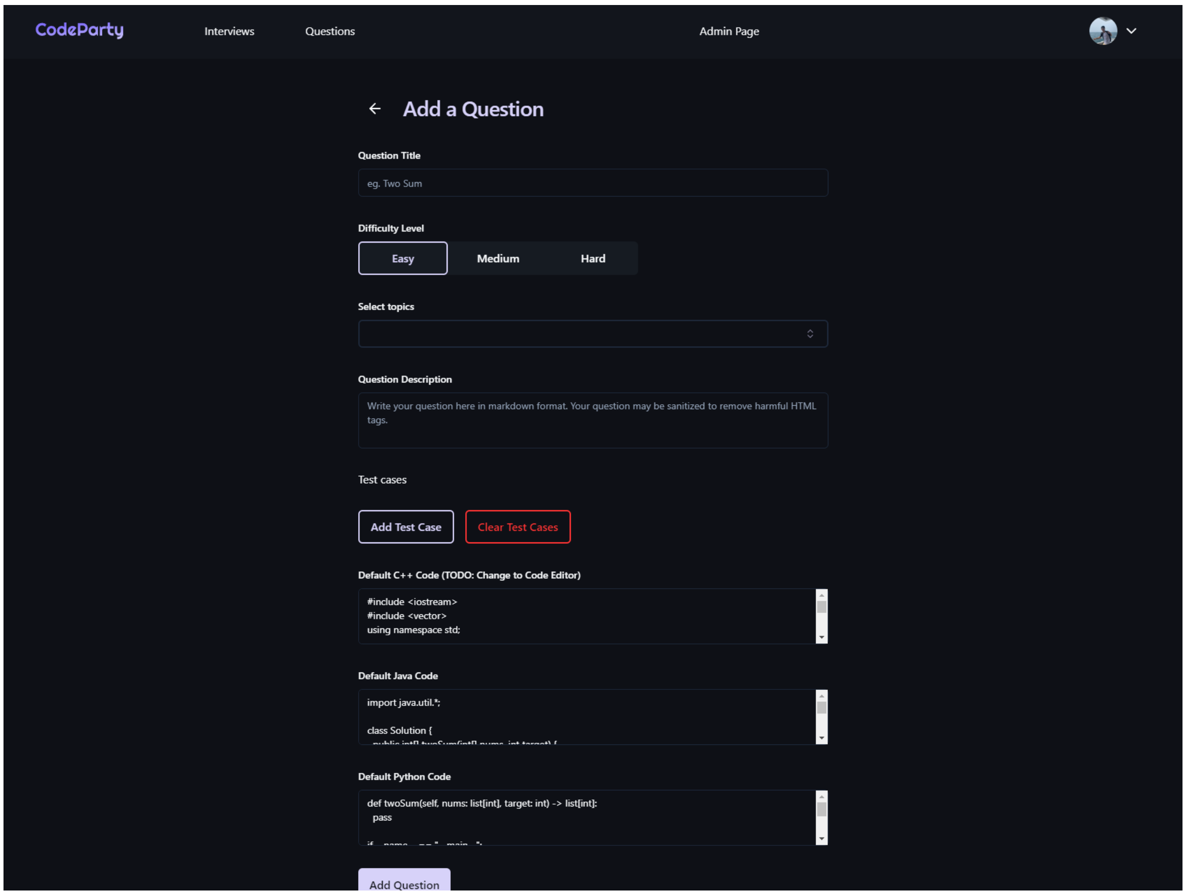
Add Binary easy

Algorithms · Bit Manipulation

Given two binary strings a and b, return their sum as a binary string.

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
```

Image 8: Add Questions Page



Add a Question

Question Title

eg. Two Sum

Difficulty Level

Easy · Medium · Hard

Select topics

Question Description

Write your question here in markdown format. Your question may be sanitized to remove harmful HTML tags.

Test cases

Default C++ Code (TODO: Change to Code Editor)

```
#include <iostream>
#include <vector>
using namespace std;
```

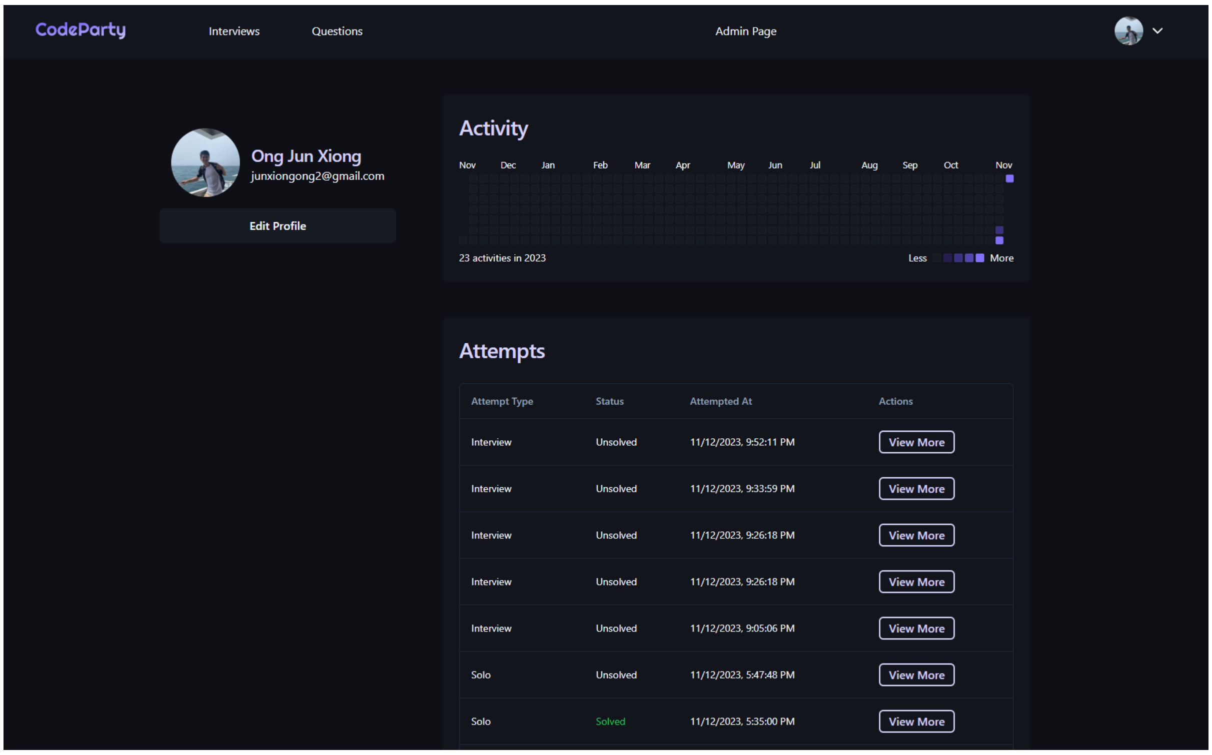
Default Java Code

```
import java.util.*;

class Solution {
  public int[] twoSum(int[] nums, int target) {
```

Default Python Code

```
def twoSum(self, nums: list[int], target: int) -> list[int]:
  pass

if __name__ == "__main__":
```

Image 9: Profile Page



Ong Jun Xiong
junxiongong2@gmail.com

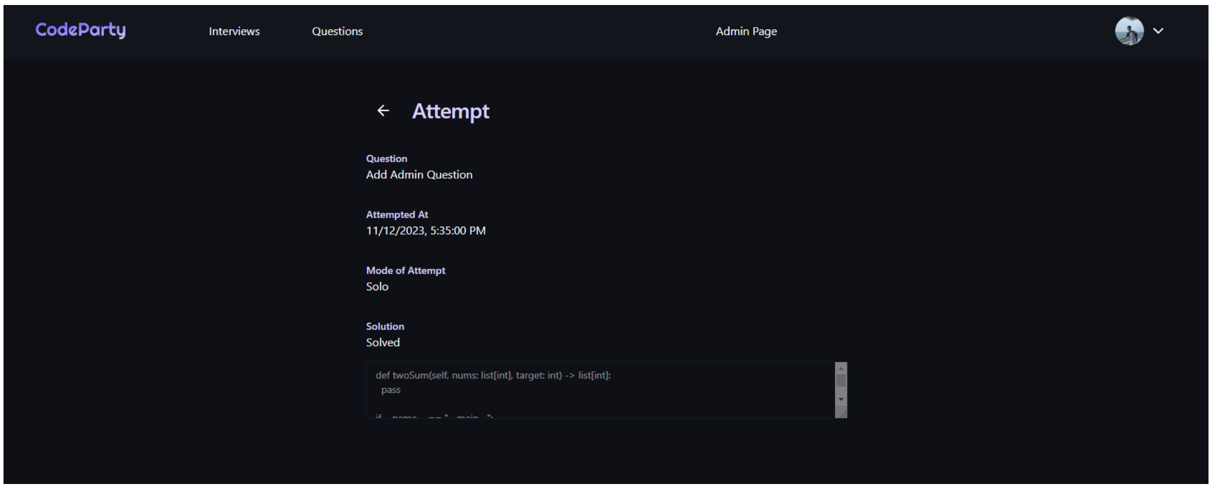
Activity

23 activities in 2023

Attempts

Attempt Type	Status	Attempted At	Actions
Interview	Unsolved	11/12/2023, 9:52:11 PM	View More
Interview	Unsolved	11/12/2023, 9:33:59 PM	View More
Interview	Unsolved	11/12/2023, 9:26:18 PM	View More
Interview	Unsolved	11/12/2023, 9:26:18 PM	View More
Interview	Unsolved	11/12/2023, 9:05:06 PM	View More
Solo	Unsolved	11/12/2023, 5:47:48 PM	View More
Solo	Solved	11/12/2023, 5:35:00 PM	View More

Image 10: Attempt History Page



Attempt

Question
Add Admin Question

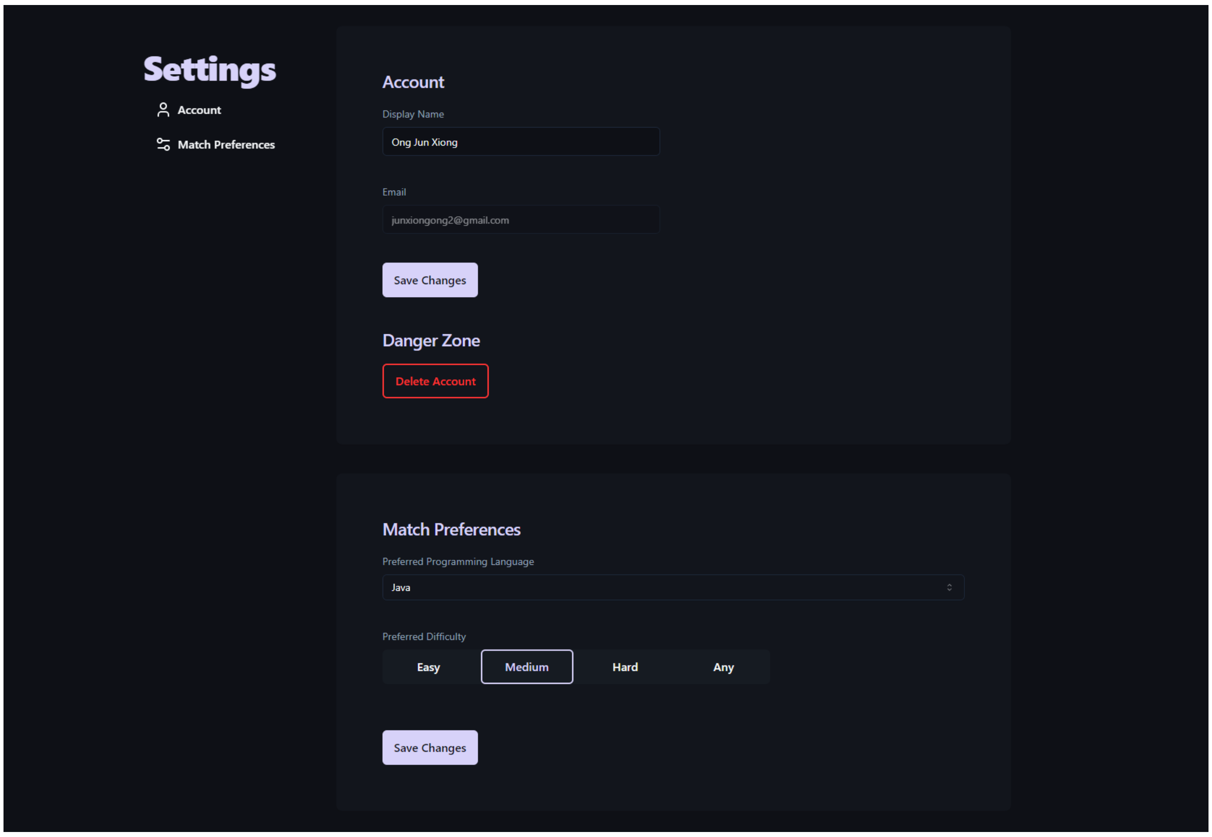
Attempted At
11/12/2023, 5:35:00 PM

Mode of Attempt
Solo

Solution
Solved

```
def twoSum(self, nums: list[int], target: int) -> list[int]:
  pass

if __name__ == "__main__":
```

Image 11: Settings Page



Settings

Account

Match Preferences

Account

Display Name

Ong Jun Xiong

Email

junxiongong2@gmail.com

Save Changes

Danger Zone

Delete Account

Match Preferences

Preferred Programming Language

Java

Preferred Difficulty

Easy [✓ Medium] Hard Any

Save Changes

4.5.2. API Documentation

Comprehensive API documentation was created using tools like Swagger. Using swagger-autogen, we standardized the format of our API documentation across services and unified them to be hosted on the `/docs` url for each service. This interactive documentation provided clear instructions on how to utilize the APIs, making it easier for front-end and back-end teams to collaborate and for future developers to understand the system's capabilities.

Example swagger output:
https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g11/blob/master/services/question-service/src/swagger-output.json

4.5.3. Testing

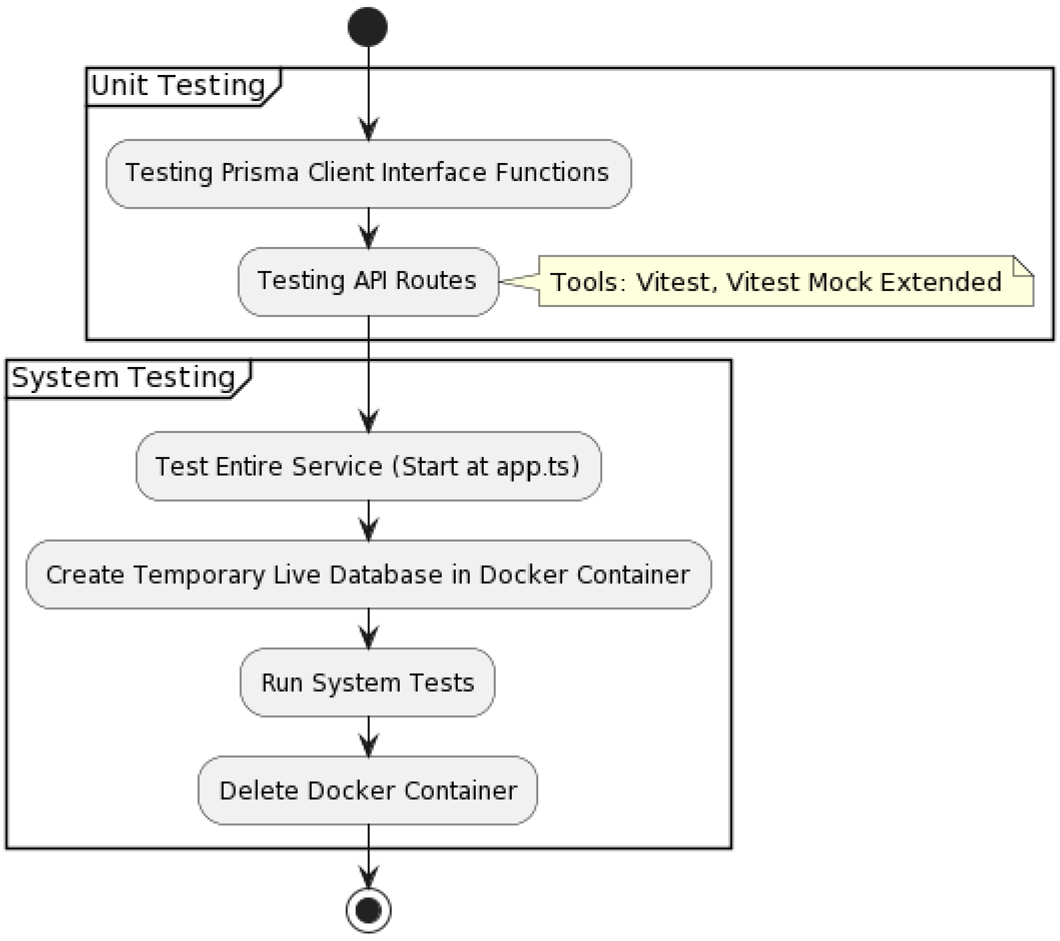

Unit Testing
Testing Prisma Client Interface Functions
Testing API Routes
Tools: Vitest, Vitest Mock Extended
System Testing
Test Entire Service (Start at app.ts)
Create Temporary Live Database in Docker Container
Run System Tests
Delete Docker Container

Diagram 16: Testing Pipeline

Two levels of tests are defined:
- Unit test: testing an individual file
- System test: testing the entirety of a microservice

An example for the User Service is shown below:

Level	Tests run
Unit testing	• Testing the Prisma Client interface functions • Testing the API routes In both cases, Vitest(https://vitest.dev/) and Vitest Mock Extended(https://www.npmjs.com/package/vitest-mock-extended) were used. Vitest Mock Extended is used for mocking the table for storing custom user data like match preferences.
System testing	• The entire service with app.ts as the starting point Here, a live temporary database is created on a temporary Docker container. The container is deleted as soon as the test completes

5. Suggestions for Improvements

To elevate the platform's capabilities and user engagement, several enhancements are proposed. Firstly, integrating smart matching algorithms could significantly refine the accuracy of user pairings, ensuring more relevant and fruitful interactions. Introducing gamification elements, such as rewards or leaderboards, can incentivize users to engage more deeply with the platform. Improving the UI/UX design is also crucial; a more intuitive and visually appealing interface would enhance the overall user experience, making the platform more accessible and enjoyable.

Further technical enhancements include employing smart solutions to streamline the question service, optimizing how inquiries are processed and addressed. This could involve smart categorization of questions or predictive text features to assist users in framing their queries. Additionally, the platform has incorporated functionality that allows users to execute code snippets, providing immediate feedback on whether the code solves the problem at hand—a feature that could dramatically expedite the learning process. With that functionality, we can evaluate the attempts by users and store statistics such as number of successful attempts.

Lastly, implementing an introductory tutorial would be instrumental in onboarding new users. This tutorial would guide them through the platform's features and suggest strategies to tackle challenges, setting the stage for a supportive learning environment right from the start. These suggestions aim to create a more engaging, efficient, and user-friendly platform, catering to the diverse needs of its user base.

6. Reflections and Learning Points

This project was an immense learning curve, especially in understanding the complexities of a microservices architecture and the intricate balance required in team coordination. We learned the importance of clear communication, especially when different services are developed by different sub-groups. Agile methodology played a significant role in keeping the project on track and adapting to changes.

The technical challenges, especially around the real-time collaboration feature, were substantial but rewarding. They provided deep insights into real-time data synchronization, conflict resolution, and network latency.

In retrospect, spending more time during the initial planning phase on designing the interactions between microservices could have saved troubleshooting time later on. Overall, the project highlighted the multifaceted nature of software development, where technical prowess, user-centric design, and team collaboration go hand in hand.

7. References

Janssen, B. (2021, April 19). *Create an API gateway using NodeJS and express*. Medium. https://medium.com/geekculture/create-an-api-gateway-using-nodejs-and-express-933d1ca23322

Purwar, R. (2022, July 6). *How to set up GitHub user authentication using Firebase and react (with hooks)*. freeCodeCamp.org. https://www.freecodecamp.org/news/github-user-authentication-using-firebase-and-reactjs-with-hooks/