CS3219 - Software Engineering Principles and Patterns

AY2022/2023 Semester 1

Team Project Report

Group 5

Production URL: www.codecollab.codes

Name	Matriculation Number
Chia Wen Ling	A0205983X
Tan Ying Jie, Dexter	A0149802A
Wong Jun Long	A0201889W
Yen Pin Hsuan	A0205520X

Table of Contents

Table of Tables

Table of Figures

1. Introduction

This document outlines the development process and design decisions of CodeCollab, a collaborative technical interview preparation platform.

1.1. Background

Students applying for jobs in the computing industry often face challenging technical interviews. During these interviews, students are expected to solve algorithmic problems and implement solutions in code. However, many students may not be well-equipped to handle the rigor of these interviews. Some pain points experienced may include being unable to solve the given problem or an inability to communicate thought processes to the interviewer. Moreover, repetitive practice of algorithmic problems can be taxing and mundane.

1.2. Purpose

To address the above pain points, the team has created a collaborative interview preparation platform called CodeCollab to improve the technical proficiency and communication skills of users.

1.3. CodeCollab

In CodeCollab, users can select their choice of programming language and match with peers to practice interview questions together using a shared code editor. Users will be able to chat with one another and view each other in real-time over video streaming. This will allow users to practice their communication skills and hone their ability to communicate technical concepts to other parties. Users will also be able to run their code submissions and verify the correctness of their code using CodeCollab's submission feature.

To encourage users to set goals for themselves and track their progress towards their goals, CodeCollab has integrated a dashboard which provides insights into the user's activity on the website. Users will be able to track the number of questions attempted for each difficulty level, time taken for each attempt of the past month, topics and the number of questions related to the topic attempted, previous peer collaborators, and number of code submissions per day of the past year.

2. Individual Contributions

The table below summarizes the individual contributions of team members to the project.

Chia Wen Ling (Frontend / Backend)

Technical

- Implemented question service for selection of question by difficulty and APIs in question controller.
- Implemented room controller.
- Implemented frontend for difficulty selection page and navigation to match waiting page.
- Implemented frontend for existing room check for rejoining or leaving room.
- Implemented frontend room related API hooks.
- Added API endpoints to user controller for username retrieval and password update.
- Added support in user module for case agnostic emails.

Non-Technical

- Reviewed PR for group members.
- Worked on Question Selection and Display section of the report.

Tan Ying Jie, Dexter (Lead Developer / Frontend / Backend / DevOps)

Technical

- Set up pre-commit hooks, code linters, and code formatters.
- Set up continuous integration workflows on GitHub Actions.
- Bootstrapped frontend and backend applications by converting them to TypeScript.
- Designed the database schema and setup database connections.
- Created the local development environment with Docker Compose.
- Implemented user creation services and controllers.
- Implemented email verification and reset password services and external Twilio Verify integration.
- Implemented user authentication controller, service, guards, serializer, and session middleware.
- Implement configuration service for parsing and validating environment variables.
- Implemented application-wide logging service.
- Implemented frontend API service.
- Implemented frontend sign up, login, and password reset forms.
- Implemented frontend routing, protected routes, public routes, and page layouts.
- Implemented frontend navigation bar and logout feature.
- Integrated notification bars into frontend application.
- Integrated Socket.IO and implemented reusable socket context.
- Bootstrapped shared folder and integrated with frontend and backend folders.
- Updated frontend theme to reuse Material UI's default color palette.
- Updated and standardized fonts used in the frontend.
- Implemented reusable generic frontend components including buttons, layouts, and input elements.

- Migrated frontend to use Vite for improved developer experience and reduced development time.
- Implemented room gateway and services to handle room creation, room teardown, and user connection changes.
- Implemented backend support for collaborative editor and integrated frontend editor libraries.
- Integrated authentication middleware Socket.IO using custom adapters and modified editor namespace creation logic.
- Redesigned and implemented the updated room page with user status indicators.
- Added support for input validation using class-validator and custom decorators.
- Implemented textual chat feature and related frontend and backend components using Twilio Conversations.
- Built video chat related components and implemented video chat integration using Twilio Video.
- Implemented submission callback for integration with code execution provider.
- Implemented statistics retrieval controller and service.
- Added support for handling duplicate Socket.IO connections.
- Set up backend test infrastructure for REST API and Socket.IO connections.
- Fixed WebRTC glare bug in y-webrtc library.
- Designed and set up the staging environment on Google Cloud Platform (GCP).
- Set up and configured production infrastructure using GCP.
- Converted question scraping script to TypeScript and developed an automated data cleaning and ingest pipeline.

Non-Technical

- Reviewed PRs for group members.
- Worked on System Architecture, Deployment Architecture, CI/CD, and Authentication, WebSocket Connections, Collaborative Editor, Communication Services and Question Ingest sections of the report.

Wong Jun Long (Designer / Frontend)

Technical

- Designed wireframes by following an iterative prototyping process of assessing problem, designing by generating ideas, building prototype using subset of ideas generated, and assessing result using user testing until a viable product has been achieved.
- Adopted UI/UX best practices which caters to human visual perception, long/short-term memory and gulf bridging to deliver an intuitive interface to end-users.
- Implemented wireframe designs for all front-end pages in code using React components.
- Wrote a Python script to scrape questions from LeetCode back-end.
- Integrated backend statistics API into the front-end dashboard.
- Implemented end-to-end test cases to validate that users can change their display name and password and that the statistics API renders valid dashboard statistics.

Non-Technical

- Reviewed PR for group members.

• Worked on the Introduction, Project Development Process, Statistics Component, and Enhancement Suggestion portions of the report. • Assisted the team in the selection of a suitable slide deck for presentation.
Yen Pin Hsuan (Frontend / Backend) **Technical** • Implemented a module which serves as an interface for connecting and interacting with Redis. • Implemented user matching service with Socket.IO and Redis. • Implemented a basic queue page with an API endpoint for fetching usernames. • Integrated judge service with Judge0 API and implemented adapters for executing the users' code in 4 languages. • Implemented join room and leave room functionality. • Implemented API endpoint for fetching submission results and displayed the submission results in a table on frontend. • Wrote end-to-end tests for queue and submission related WebSockets. **Non-Technical** • Reviewed PR for group members. • Worked on FR and NFR, User Matching, Room Management, and Code Submission sections of the report.

Table 1: Contributions of group members

3. Specified and Prioritized Requirements

Our functional and non-functional requirements are prioritized using the MoSCoW [1] framework. Table 2 below defines the different priority levels.

Priority Levels	Description
Must-have	Requirements that are fundamental to the project being able to deliver its intended purpose.
Should-have	Requirements that are important to the project being able to achieve its intended outcome but can be deferred to subsequent releases and updates.
Could-have	Requirements that improve the user experience or extend the product functionality.
Won't-have	Requirements that have been deemed as unnecessary for the project at its current stage and will be re-evaluated at a later time when the product matures.

Table 2: Priority levels of requirements

3.1. Functional Requirements

The functional requirements (FR) of CodeCollab have been categorized into the following categories:

- Authentication
- User Profile Management
- User Matching
- Coding Room
- Coding Session: Collaborative Editor
- Coding Session: Code Judging
- Coding Session: Chat Communication
- Dashboard & Statistics

The FRs for each category are provided below. **Rows highlighted in green represent implemented FRs, and rows in red represent those not implemented.** In addition to the specific requirement and priorities, references to the related sections of the developer documentation and files that are significant to the FR's implementation have been included. Beside the name of each file, the component that the file belongs to is indicated in parentheses, with FE representing frontend and BE representing backend.

[1] See https://en.wikipedia.org/wiki/MoSCoW_method

3.1.1. Authentication

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Auth-1	The system should allow users to create a new account with their email, name, and password.	Must-have	Authentication: User Accounts	`SignUpForm` (FE) `UserService` (BE)
FR-Auth-2	The system should ensure that every account created has a unique email.	Must-have	Authentication: User Accounts	Prisma Schema
FR-Auth-3	The system should allow users to log into their accounts by entering their email and password.	Must-have	Authentication: Session Management	`LoginForm` (FE) `LocalGuard` (BE) `LocalStrategy` (BE)
FR-Auth-4	The system should allow logged in users to log out of their accounts.	Must-have	Authentication: Session Management	`AuthContext` (FE) `AuthService` (BE)
FR-Auth-5	The system should allow logged in users to change their password.	Should-have	Authentication: User Accounts	`UpdatePasswordForm` (FE) `UserSerivce` (BE)
FR-Auth-6	The system should allow users to reset their password.	Should-have	Authentication: User Accounts	`ResetPasswordForm` (FE) `ResetPasswordService` (BE)
FR-Auth-7	The system should verify the email provided during account creation before allowing users to log into their accounts.	Should-have	Authentication: User Accounts	`UserController` (BE) `VerificationService` (BE) `TwilioService` (BE)
FR-Auth-8	The system should allow unverified users to resend verification emails.	Should-have	Authentication: User Accounts	`RequestVerificationEmailForm` (FE) `VerificationService` (BE)
FR-Auth-9	The system should allow users to delete their accounts.	Could-have		

Table 3: Requirements traceability matrix for authentication requirements

3.1.2. User Profile Management

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Profile-1	The system should allow users to change their display name.	Should-have	Authentication: User Accounts	`UpdateDisplayNameForm` (FE) `UserService` (BE)
FR-Profile-2	The system should allow users to add a personal description or biography in their profile page.	Could-have		
FR-Profile-3	The system should allow users to upload a personal profile picture in `jpeg` or `png` format in the profile page.	Could-have		
FR-Profile-4	The profile page should be populated with the user's name, email, profile picture, and personal descriptions.	Could-have		

Table 4: Requirements traceability matrix for user profile management requirements

3.1.3. User Matching

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Match-1	The system should allow users to select the difficulty level of the questions they wish to attempt.	Must-have	User Matching	`SelectionPage` (FE)
FR-Match-2	The system should be able to match two waiting users with the same difficulty level and put them in the same coding room.	Must-have	User Matching	`QueueService` (BE)
FR-Match-3	If there is a valid match, the system should match the users within 30s and place them in the same coding room.	Must-have	User Matching	`QueueService` (BE)
FR-Match-4	The system should inform waiting users that no match is available if a match cannot be found within 30 seconds from the point the user started to wait for a match.	Must-have	User Matching	`SelectionPage` (BE)
FR-Match-5	The system should match only users who have chosen the same programming language and difficulty level.	Should-have	User Matching	`QueueService` (BE)
FR-Match-6	The system should allow users to select the programming language they wish to code in.	Should-have	User Matching	`SelectionPage` (FE)
FR-Match-7	The system should allow users to select multiple programming languages that they wish to code in.	Could-have		
FR-Match-8	The system should allow users to specify a blacklist of users to not match with.	Could-have		
FR-Match-9	The system should allow users to specify a whitelist of users to only match with.	Could-have		

Table 5: Requirements traceability matrix for user matching requirements

3.1.4. Coding Room

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Room-1	The system should randomly assign a question of the chosen difficulty level to a pair of users in the same coding room.	Must-have	Question Selection and Display	`QuestionService` (BE)
FR-Room-2	The system should allow both users in the coding room to view the description and any hints of the question.	Must-have	Question Selection and Display	`Question` (FE)
FR-Room-3	The system should provide a means for the user to leave the coding room once matched.	Must-have	Room Management	`RoomStatusBar` (FE) `RoomService` (BE)
FR-Room-4	The system should allow both users of the coding room to view sample test cases and their expected outputs.	Should-have	Question Selection and Display	`Question` (FE)
FR-Room-5	The system should inform the user if any other user leaves, disconnects from, or reconnects to the coding room.	Should-have	Room Management	`RoomPage` (FE) `RoomStatusBar` (FE) `RoomService` (BE)
FR-Room-6	The system should allow users to rejoin an existing coding room if the user has not explicitly left the coding room.	Should-have	User Matching	`SelectionPage` (FE) `RoomController` (BE) `RoomService` (BE)
FR-Room-7	The system should show users how much time has been spent in the coding room.	Could-have		

Table 6: Requirements traceability matrix for coding room requirements

3.1.5. Coding: Collaborative Editor

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Editor-1	The system should have a code editor which allows all users in a coding room to work on a shared piece of code together in real time.	Must-have	Collaborative Editor	`EditorContext` (FE) `EditorGateway` (BE)
FR-Editor-2	The system should support syntax highlighting in the code editor for the programming language which the users of the coding room have chosen to code in.	Should-have	Collaborative Editor	`Editor` (FE)
FR-Editor-3	The system should provide a solution template in the code editor based on the programming language chosen by the users in the coding room.	Should-have	Collaborative Editor	`RoomService` (BE) `EditorService` (BE)
FR-Editor-4	The system should save and restore the last known state of the code editor when a user connects to the coding room as long as there exists a user who has not explicitly left the coding room.	Could-have	Collaborative Editor	`EditorGateway` (BE) `EditorService` (BE)

Table 7: Requirements traceability matrix for collaborative editor requirements

3.1.6. Coding: Code Judging

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Judge-1	The system should allow all users of the coding room to submit the code in the code editor for judging against known test case(s).	Should-have	Code Submission	`SubmitButton` (FE) `RoomPage` (FE) `SubmissionsService` (BE) `JudgeService` (BE)
FR-Judge-2	The system should allow all users of the coding room to resubmit the code in the code editor multiple times for judging.	Could-have	Code Submission	`SubmissionService` (BE)
FR-Judge-3	The system should inform all users of the coding room if their submitted code has passed the test case(s).	Should-have	Code Submission	`SubmissionSerivce` (BE)
FR-Judge-4	The system should inform all users of the coding room which test case(s) the submitted code has failed.	Should-have	Code Submission	`Submissions` (FE) `SubmissionService` (BE)
FR-Judge-5	The system should provide a means for all users of the coding room to view the result of past submissions of the coding room.	Could-have	Code Submission	`Submissions` (FE) `SubmissionDialog` (FE) `SubmissionsService` (BE)
FR-Judge-6	The system should return to all users of the coding room the output of the submitted code when it fails to pass at least one test case.	Could-have		
FR-Judge-7	The system should allow all users of the coding room to submit the code in the code editor for judging against unknown test case(s).	Could-have		

Table 8: Requirements traceability matrix for code judging requirements

3.1.7. Coding: Chat Communication

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Chat-1	The system should support real-time textual chat between all users in a coding room.	Should-have	Communication Services	`ChatContext` (FE) `TextChat` (FE) `ChatService` (BE)
FR-Chat-2	The system should support the formatting (e.g., bold, italics, underline, code) of text messages.	Could-have		
FR-Chat-3	The system should allow all users in a coding room to communicate with each other in real-time through voice chat.	Could-have	Communication Services	`ChatContext` (FE) `VideoChat` (FE) `ChatService` (BE)
FR-Chat-4	The system should allow users to temporarily disable their microphones during a voice chat.	Could-have	Communication Services	`ChatContext` (FE) `VideoChat` (FE)
FR-Chat-5	The system should allow all users in a coding room to communicate with each other in real-time through video chat.	Could-have	Communication Services	`ChatContext` (FE) `VideoChat` (FE) `ChatService` (BE)
FR-Chat-6	The system should allow users to temporarily disable their video camera during a video chat.	Could-have	Communication Services	`ChatContext` (FE) `VideoChat` (FE)

Table 9: Requirements traceability matrix for chat communication requirements

3.1.8. Dashboard and Statistics

ID	Requirement	Priority	Design Doc. Ref.	Key Code Ref.
FR-Statistics-1	The system should count and display the number of questions attempted by the user for the past one year.	Should-have	Statistics	`AttemptSummaryChart` (FE) `StatisticsService` (BE)
FR-Statistics-2	The system should split the number of questions attempted into their difficulty levels.	Could-have	Statistics	`AttemptSummaryChart` (FE) `StatisticsService` (BE)
FR-Statistics-3	The system should show the time taken to attempt each difficulty of questions in the past month.	Could-have	Statistics	`DurationSummaryChart` (FE) `StatisticsService` (BE)
FR-Statistics-4	The system should show the number of questions attempted for each topic.	Could-have	Statistics	`NetworkChart` (FE) `StatisticsService` (BE)
FR-Statistics-5	The system should show the list of attempted questions with the date, question title and collaborated user.	Could-have	Statistics	`DashboardPage` (FE) `StatisticsService` (BE)
FR-Statistics-6	The system should show the questions and company which the question originated from.	Could-have		

Table 10: Requirements traceability matrix for dashboard and statistics requirements

3.2. Non-Functional Requirements

The non-functional requirements (NFR) of CodeCollab have been categorized into the following categories according to descending priority:

- Security
- Usability
- Maintainability
- Performance
- Scalability

The NFRs for each category are provided below along with a brief reason for each selected NFR as well as any supporting references and evidence. **Rows in green represent implemented NFRs, and rows in red are not implemented.**

3.2.1. Security

ID	Requirement	Reason	Priority
NFR-Security-1	The system should keep users logged in on a browser for no longer than 7 days after the last activity.	To prevent unauthorized access to the system.	Must-have
NFR-Security-2	User passwords should be required to have at least 8 characters, a special symbol, an uppercase letter, a lowercase letter, and a number.	To prevent the password from being cracked easily.	Must-have
NFR-Security-3	User passwords should be salted and hashed before they are stored in the database.	To prevent attackers from getting access to the plaintext password from the database.	Must-have
NFR-Security-4	Verification and password reset emails should expire after 10 minutes.	To prevent attackers from using the verification links in the future if the user's inbox is compromised.	Should-have

Table 11: Requirements traceability matrix for security requirements

Security is a core requirement in CodeCollab as it stores sensitive user data such as passwords, names, and emails. To ensure that user accounts are secure against simple attacks and are not low-hanging fruits to attackers, CodeCollab has taken the following measures to achieve the NFRs stated in Table 11.

NFR-Security-1: Session cookies are configured by the SessionMiddleware to have a maximum age value of 1 week, ensuring they expire if they are not refreshed.

NFR-Security-2: User passwords are validated by the IsStrongPassword decorator which is based on the default settings of the isStrongPassword validator of validator.js[2] to ensure that user passwords are not susceptible to brute force attacks.

NFR-Security-3: During account creation, password resets, and password updates, user passwords are salted and hashed using the bcrypt[3] library before they are stored in the database.

NFR-Security-4: Twilio Verify is an external service used by CodeCollab for generating and verifying password reset and email verification tokens. According to their documentation, their generated tokens are valid for 10 minutes by default.

3.2.2. Usability

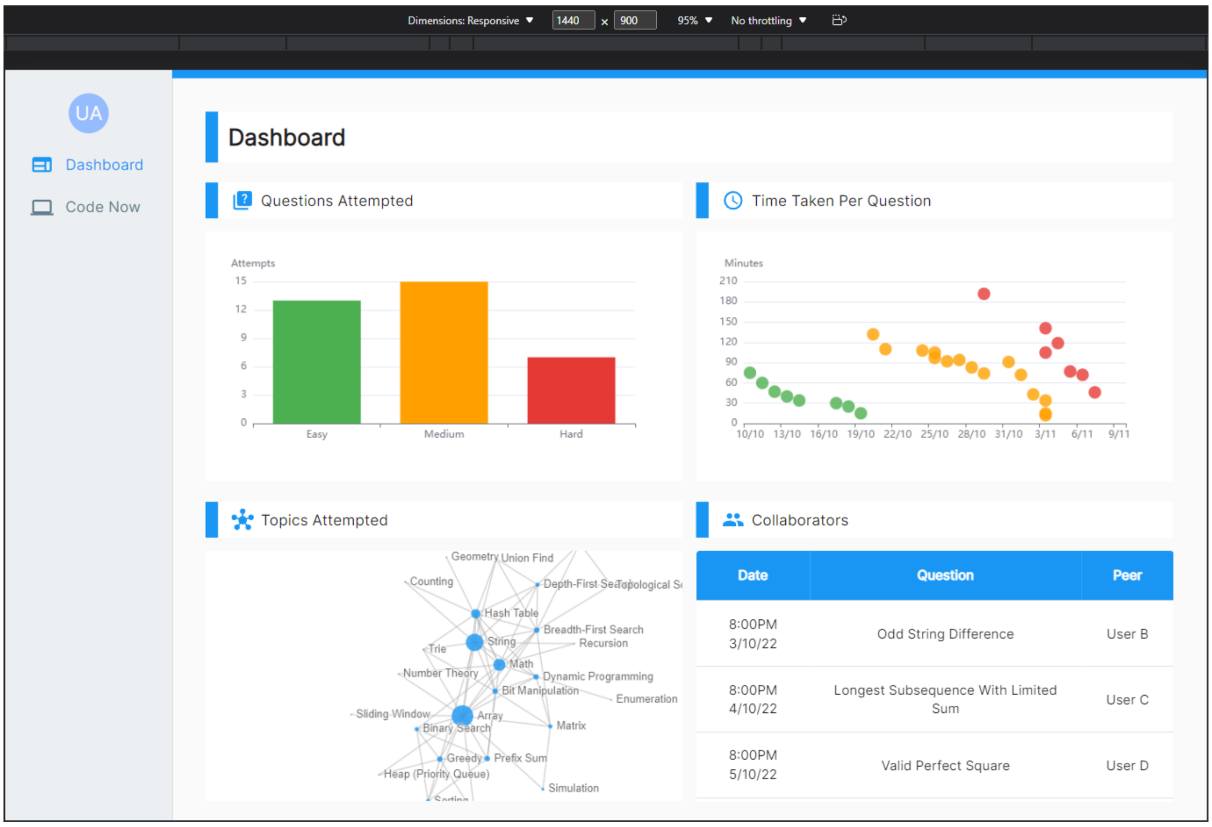
ID	Requirement	Reason	Priority
NFR-Usability-1	The UI should responsively scale to the screen resolution for resolutions between 1440 by 900 to 2560 by 1440 pixels.	To ensure that users with different screen resolutions can have similar experiences.	Should-have
NFR-Usability-2	Navigating to the coding page should not take more than 5 clicks from any page.	This ensures that users can use the application easily.	Should-have

Table 12: Requirements traceability matrix for usability requirement

Usability is a significant factor in attracting users to an application. As such, ensuring that the application is easy to use is a top priority during the development of CodeCollab. To ensure that the application is easier to use for all users, the following analysis and verifications have been performed to ensure that the requirements stated above have been met.

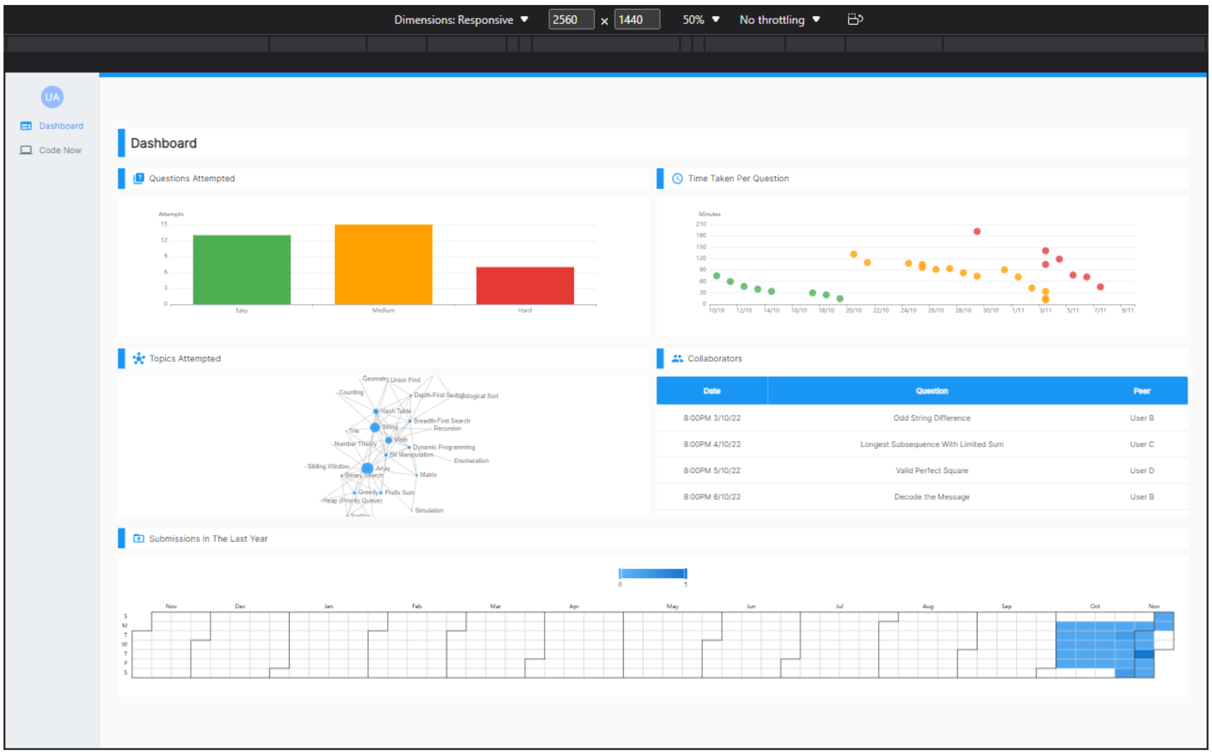
NFR-Usability-1: The main pages of the application, the dashboard and editor pages, have been verified to scale responsively according to different screen resolutions. Figures 1, 2, 3, and 4 verify that the layout of the pages remain relatively consistent across different resolutions.

[2] https://www.npmjs.com/package/validator
[3] https://www.npmjs.com/package/bcrypt



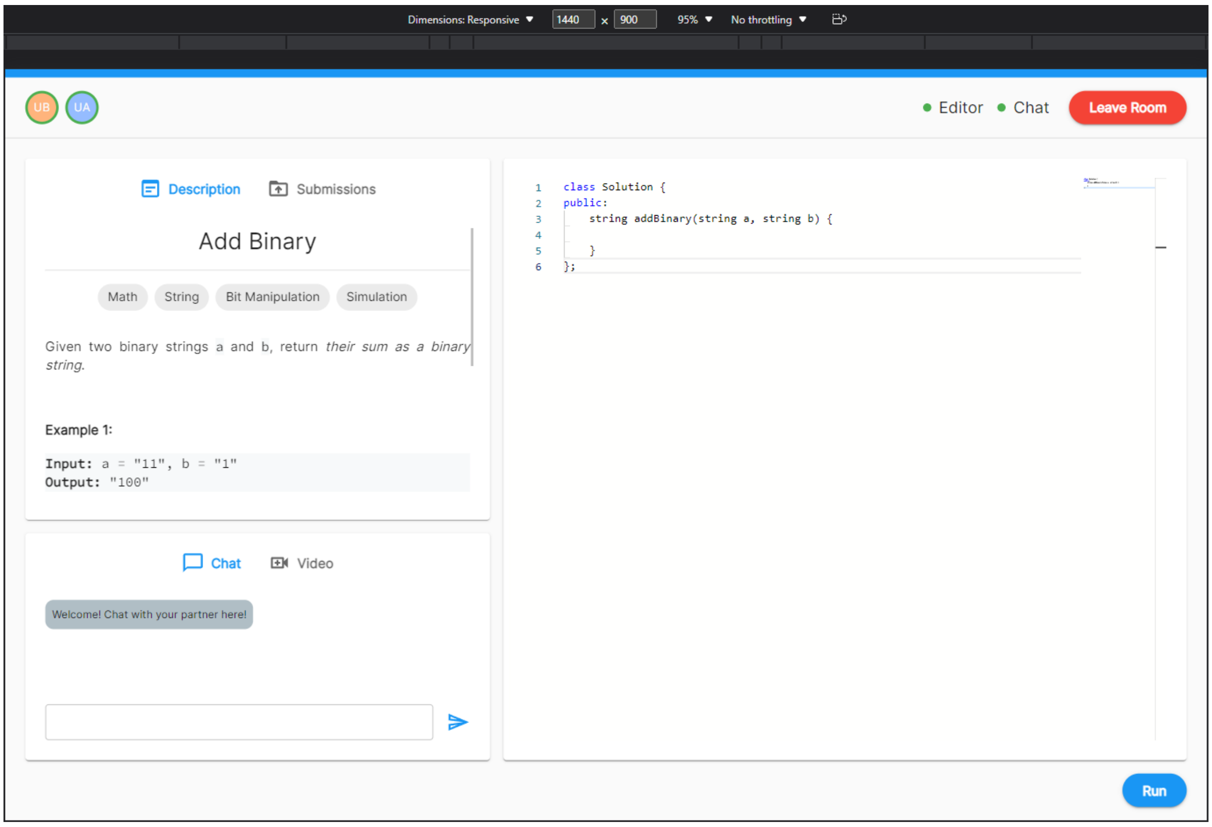
Date	Question	Peer
8:00PM 3/10/22	Odd String Difference	User B
8:00PM 4/10/22	Longest Subsequence With Limited Sum	User C
8:00PM 5/10/22	Valid Perfect Square	User D

Figure 1: Screenshot of dashboard page with resolution of 1440 pixels by 900 pixels



Date	Question	Peer
8:00PM 3/10/22	Odd String Difference	User B
8:00PM 4/10/22	Longest Subsequence With Limited Sum	User C
8:00PM 5/10/22	Valid Perfect Square	User D
8:00PM 6/10/22	Decode the Message	User B

Figure 2: Screenshot of dashboard page with resolution of 2560 pixels by 1440 pixels



Add Binary

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

Example 1:

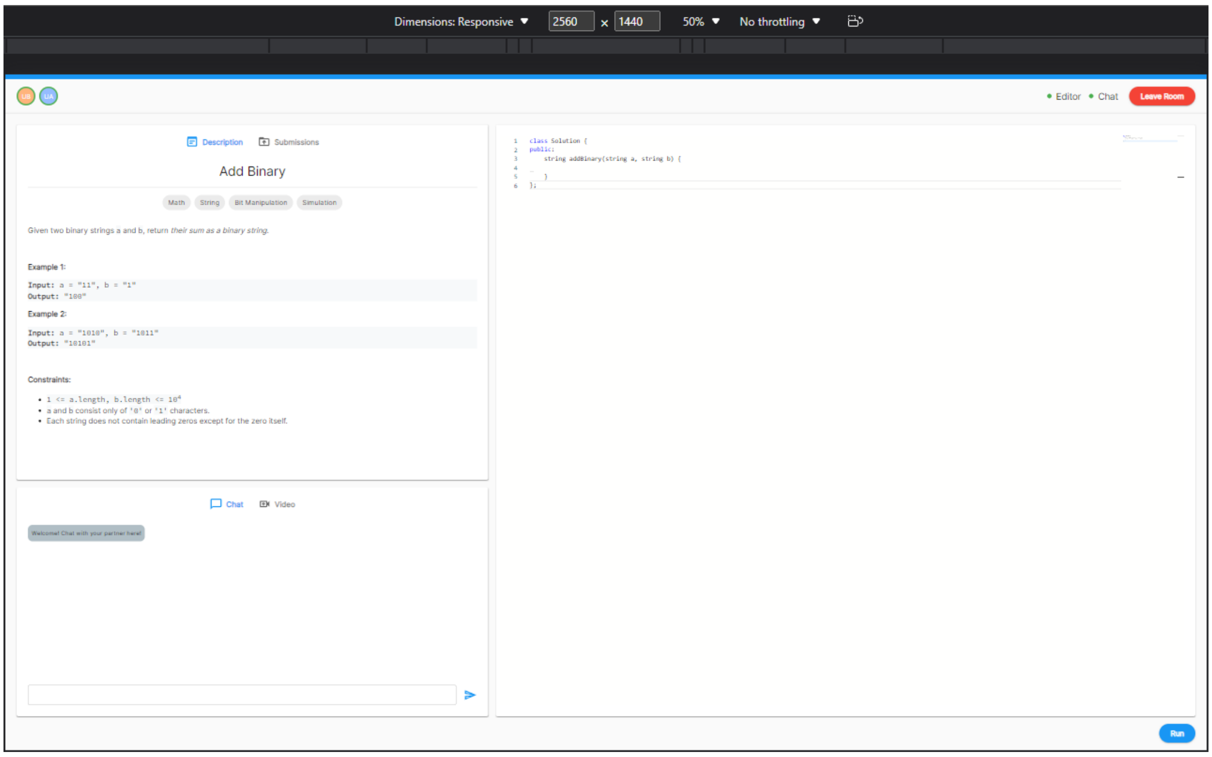
```
Input: a = "11", b = "1"
Output: "100"
```

Welcome! Chat with your partner here!

```
class Solution {
public:
    string addBinary(string a, string b) {

    }
};
```

Figure 3: Screenshot of room page with resolution of 1440 pixels by 900 pixels



Add Binary

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

Example 1:

```
Input: a = "11", b = "1"
Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"
Output: "10101"
```

Constraints:

- `1 <= a.length, b.length <= 10`^{4}
- a and b consist only of `'0'` or `'1'` characters.
- Each string does not contain leading zeros except for the zero itself.

Welcome! Chat with your partner here!

```
class Solution {
public:
    string addBinary(string a, string b) {

    }
};
```

Figure 4: Screenshot of room page with resolution of 2560 pixels by 1440 pixels

NFR-Usability-2: The flow chart shown in Figure 5 outlines the number of clicks needed to reach the Coding Room page, with each **red arrow** representing a click. Based on the diagram, the maximum number of clicks needed by the user to navigate to the Coding Room page is 5, therefore fulfilling the requirement outlined.

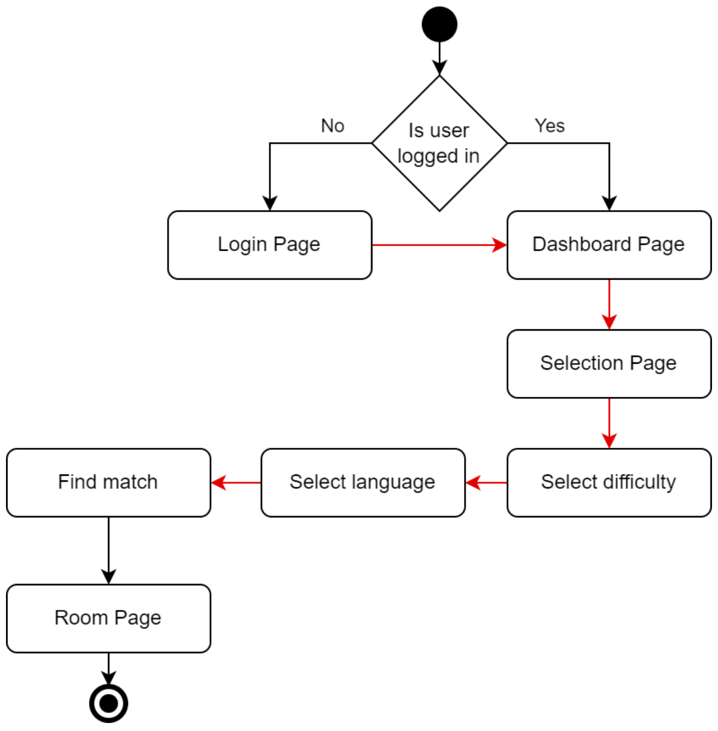


Figure 5: Flow chart showing navigation to the room page

3.2.3. Maintainability

ID	Requirement	Reason	Priority
NFR-Maintainability-1	The system should allow new questions to be added to the question bank easily.	The question bank should be updated frequently to ensure relevance of the questions provided.	Should-have
NFR-Maintainability-2	Majority (85%<) of the system should be written in a common type-safe programming language.	Reduces the risk of introducing new bugs from incompatible types during future maintenance.	Could-have

Table 13: Requirements traceability matrix for maintainability requirement

Maintainability ranks relatively high amongst the NFRs identified for the application as it enables new features and bug fixes to be rolled out more easily. As the application is still in its infancy stages, setting a strong foundation for good maintainability would significantly improve the developer experience down the road. As such, the team has taken the following steps to achieve the shortlisted requirements.

NFR-Maintainability-1: A serverless script has been developed to automatically scrap and process questions from LeetCode before ingesting them into the database.

NFR-Maintainability-2: The team has as far as possible used TypeScript as the main programming language of choice with close to 90% of the code is written in it (Figure 6).

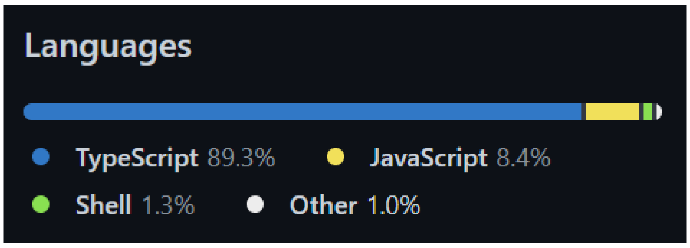


Figure 6: Breakdown of languages used in the application

3.2.4. Performance

ID	Requirement	Reason	Priority
NFR-Perf-1	The initial web page load time should be less than 3 seconds.	To reduce waiting time of users.	Could-have
NFR-Perf-2	The time taken to find a match when another user is already in the queue should be no more than 2 seconds.	To reduce waiting time of users.	Could-have
NFR-Perf-3	The time between when a match is found between two users and when the coding room page renders completely should be no more than 7 seconds.	To reduce waiting time of users.	Could-have
NFR-Perf-4	Changes by one user in the code editor should be reflected in the other user's code editor within 1 second.	Low latency ensures positive user experience.	Should-have
NFR-Perf-5	Chat messages sent by one user should be received by the other user within 2 seconds.	Low latency is necessary for the chat to be real-time.	Should-have

Table 14: Requirements traceability matrix for performance requirement

Performance is an important part of user experience and is particularly important when an application has a large user base that may generate loads impacting its performance. As the application is still in its early phases, performance has been considered but is not a top priority at this stage of the application's lifecycle. To verify that the NFRs listed in Table 14 have been achieved, the following user acceptance tests (UAT) have been performed as means of verification.

NFR-Perf-1: Test Case Recording | Estimated page load time: Under 2 seconds.

NFR-Perf-2: Test Case Recording | Estimated user matching time: Under 0.5 seconds.

NFR-Perf-3: Test Case Recording | Estimated room load time: Approximately 5 seconds.

NFR-Perf-4: Test Case Recording | Estimated code editor latency: Under 0.5 seconds.

NFR-Perf-5: Test Case Recording | Estimated chat message latency: Under 1 second.

3.2.5. Scalability

ID	Requirement	Reason	Priority
NFR-Scale-1	The system should support coding rooms with users connected to different server instances.	To ensure support for horizontal scalability of the application under load.	Could-have

Scalability is an important part of a web application to ensure that it can grow to support a large user base. However, as the application is still in its early stages, the user base is unlikely to be large and hence it is ranked the lowest in terms of importance for NFRs. To support scalability the system has implemented various features, and additional UAT has been performed to ensure that the system can scale horizontally.

NFR-Scale-1: Inter-server synchronization is achieved using the SynchronizedSocketAdapter (see section 4.3.3) and editors are synchronized using redundant protocols (see section 4.3.7). The system has also been validated against the following test case.

- Test Case Recording | Users connected to different servers can match with each other, code in the shared editor, and room events are propagated across servers.

4. Developer Documentation

This section introduces CodeCollab's project development process, system architecture, design of key components, continuous integration/continuous delivery workflow, and deployment architecture.

4.1. Project Development Process

This section describes the management tools and workflows that the team uses to manage the project. Table 15 shows an overview of the process.

Software Development Process	Adapted Hybrid Scrum-Kanban (Scrumban)
Workflow Management Platform	GitHub Issues and Project Board
Scheduling System	Priority-based

Table 15: Overview of project development process

4.1.1. Software Development Process

The team has adopted a hybrid Scrum-Kanban (Scrumban) software development process. This involves a weekly meeting to decide on issues that should be prioritized for the upcoming week but only assigning a subset of issues. As the week progresses, members are given the opportunity to assign themselves any other issues that have not been completed and are not strictly restricted to the prioritized list.

The team decided on a Scrumban workflow for the benefits it offers towards team morale while offering some structure. This process offers the benefit of the Scrum workflow from regular planning and synchronization between team members while offering flexibility of the Kanban workflow by not strictly assigning team member roles and allowing a variation of workload as they deem fit. Each team member is given the freedom to decide which tasks best suit their expertise and the amount of work that they can handle for the week.

4.1.2. Weekly Meetings

The team follows a weekly planning system which takes place during regular team meetings. Figure 7 shows the typical agenda for a weekly meeting.

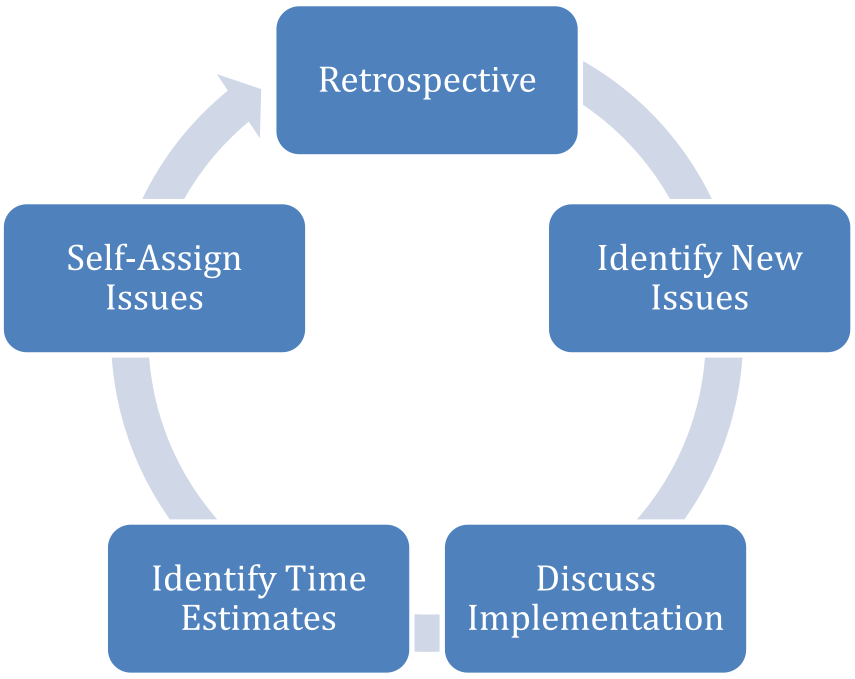
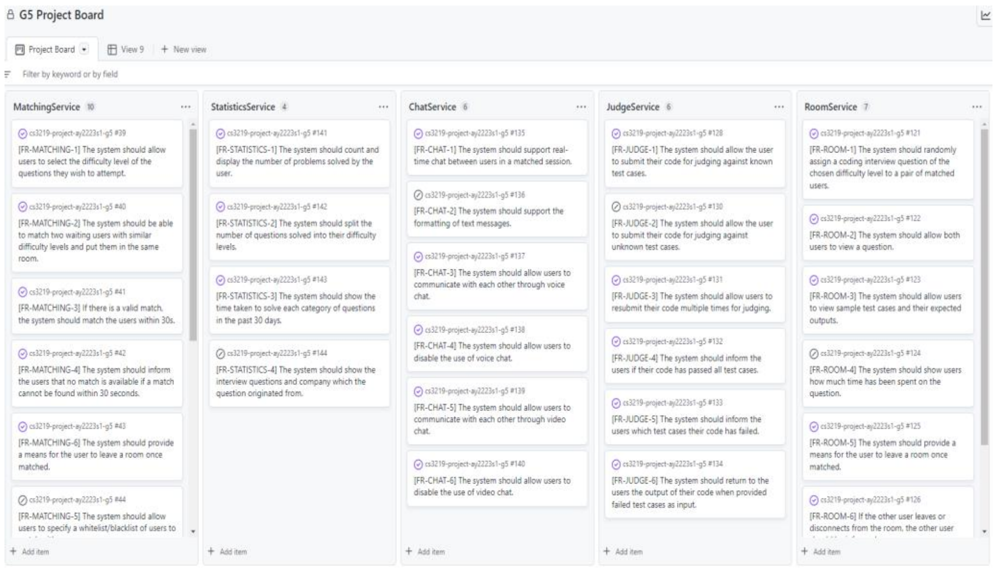


Figure 7: Items discussed during each weekly meeting

The meeting begins with a retrospective where members share their contributions and progress for the previous week. Any unresolved issues are brought forward to be included in the upcoming week's agenda. Following which, new issues may be identified and are given a priority. Time is also set aside to discuss implementation details for any new complex tasks. Finally, a list of issues that should be prioritized for that week is generated and the estimated workload for each of the issues are discussed. At this juncture, members can volunteer to assign themselves to one or more issues but not all issues may be assigned a member. Throughout the week, members can assign themselves to other issues should they have the bandwidth to do so.

4.1.3. Management Tools and Workflow

The team has adopted both the use of GitHub Project Boards and GitHub Issues to track outstanding tasks and workload. The Project Board is used to track the progress of features based on their functional requirements (FRs). Meanwhile, GitHub Issues are used to track the workload and priority of outstanding tasks, which are usually smaller steps required to fulfil an FR. During weekly meetings, a subset of the uncompleted FRs is identified on the Project Board based on their priority (Figure 8). The team then outlines a set of tasks needed to fulfil the FRs identified. Finally, GitHub Issues are created for these tasks (Figure 9) and team members can assign themselves issues when they are available to do so.



MatchingService 10

cs3219-project-ay2223s1-g5 #39
[FR-MATCHING-1] The system should allow users to select the difficulty level of the questions they wish to attempt.

cs3219-project-ay2223s1-g5 #40
[FR-MATCHING-2] The system should be able to match two waiting users with similar difficulty levels and put them in the same room.

cs3219-project-ay2223s1-g5 #41
[FR-MATCHING-3] If there is a valid match, the system should match the users within 30s.

cs3219-project-ay2223s1-g5 #42
[FR-MATCHING-4] The system should inform the users that no match is available if a match cannot be found within 30 seconds.

cs3219-project-ay2223s1-g5 #43
[FR-MATCHING-6] The system should provide a means for the user to leave a room once matched.

cs3219-project-ay2223s1-g5 #44
[FR-MATCHING-5] The system should allow users to specify a whitelist/blacklist of users to

StatisticsService 4

cs3219-project-ay2223s1-g5 #141
[FR-STATISTICS-1] The system should count and display the number of problems solved by the user.

cs3219-project-ay2223s1-g5 #142
[FR-STATISTICS-2] The system should split the number of questions solved into their difficulty levels.

cs3219-project-ay2223s1-g5 #143
[FR-STATISTICS-3] The system should show the time taken to solve each category of questions in the past 30 days.

cs3219-project-ay2223s1-g5 #144
[FR-STATISTICS-4] The system should show the interview questions and company which the question originated from.

ChatService 6

cs3219-project-ay2223s1-g5 #135
[FR-CHAT-1] The system should support real-time chat between users in a matched session.

cs3219-project-ay2223s1-g5 #136
[FR-CHAT-2] The system should support the formatting of text messages.

cs3219-project-ay2223s1-g5 #137
[FR-CHAT-3] The system should allow users to communicate with each other through voice chat.

cs3219-project-ay2223s1-g5 #138
[FR-CHAT-4] The system should allow users to disable the use of voice chat.

cs3219-project-ay2223s1-g5 #139
[FR-CHAT-5] The system should allow users to communicate with each other through video chat.

cs3219-project-ay2223s1-g5 #140
[FR-CHAT-6] The system should allow users to disable the use of video chat.

JudgeService 6

cs3219-project-ay2223s1-g5 #128
[FR-JUDGE-1] The system should allow the user to submit their code for judging against known test cases.

cs3219-project-ay2223s1-g5 #130
[FR-JUDGE-2] The system should allow the user to submit their code for judging against unknown test cases.

cs3219-project-ay2223s1-g5 #131
[FR-JUDGE-3] The system should allow users to resubmit their code multiple times for judging.

cs3219-project-ay2223s1-g5 #132
[FR-JUDGE-4] The system should inform the users if their code has passed all test cases.

cs3219-project-ay2223s1-g5 #133
[FR-JUDGE-5] The system should inform the users which test cases their code has failed.

cs3219-project-ay2223s1-g5 #134
[FR-JUDGE-6] The system should return to the users the output of their code when provided failed test cases as input.

RoomService 7

cs3219-project-ay2223s1-g5 #121
[FR-ROOM-1] The system should randomly assign a coding interview question of the chosen difficulty level to a pair of matched users.

cs3219-project-ay2223s1-g5 #122
[FR-ROOM-2] The system should allow both users to view a question.

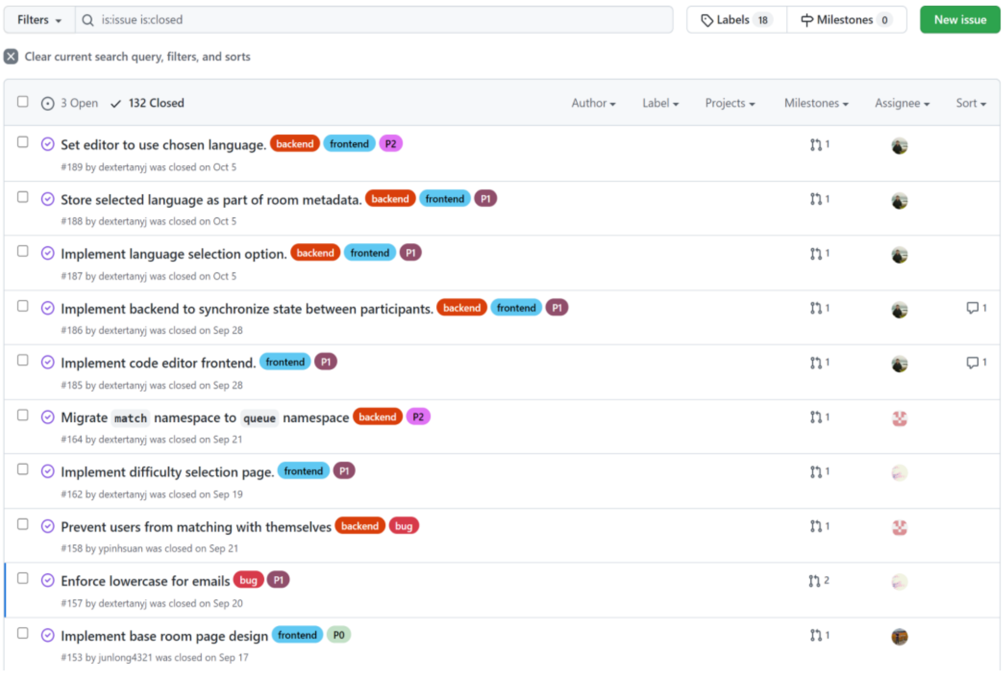
cs3219-project-ay2223s1-g5 #123
[FR-ROOM-3] The system should allow users to view sample test cases and their expected outputs.

cs3219-project-ay2223s1-g5 #124
[FR-ROOM-4] The system should show users how much time has been spent on the question.

cs3219-project-ay2223s1-g5 #125
[FR-ROOM-5] The system should provide a means for the user to leave a room once matched.

cs3219-project-ay2223s1-g5 #126
[FR-ROOM-6] If the other user leaves or disconnects from the room, the other user

Figure 8: Usage of GitHub Project Boards



is:issue is:closed

3 Open ✓ 132 Closed

Set editor to use chosen language. backend frontend P2 — 1
#189 by dextertanyj was closed on Oct 5

Store selected language as part of room metadata. backend frontend P1 — 1
#188 by dextertanyj was closed on Oct 5

Implement language selection option. backend frontend P1 — 1
#187 by dextertanyj was closed on Oct 5

Implement backend to synchronize state between participants. backend frontend P1 — 1 — 1
#186 by dextertanyj was closed on Sep 28

Implement code editor frontend. frontend P1 — 1 — 1
#185 by dextertanyj was closed on Sep 28

Migrate `match` namespace to `queue` namespace backend P2 — 1
#164 by dextertanyj was closed on Sep 21

Implement difficulty selection page. frontend P1 — 1
#162 by dextertanyj was closed on Sep 19

Prevent users from matching with themselves backend bug — 1
#158 by ypinhsuan was closed on Sep 21

Enforce lowercase for emails bug P1 — 2
#157 by dextertanyj was closed on Sep 20

Implement base room page design frontend P0 — 1
#153 by junlong4321 was closed on Sep 17

Figure 9: Usage of GitHub Issues

4.2. System Architecture

This section introduces the core technologies used in the development of CodeCollab and highlights some of the key considerations considered when choosing these technologies. It begins with an outline of the various high-level components of the application before going into deeper detail about the frontend and backend components.

4.2.1. System-Level Architecture

Figure 10 presents the high-level architecture of the application. Details of the frontend component can be found in Figure 11 of section 4.2.2 and details of the backend can be found in Figure 12 of section 4.2.3.

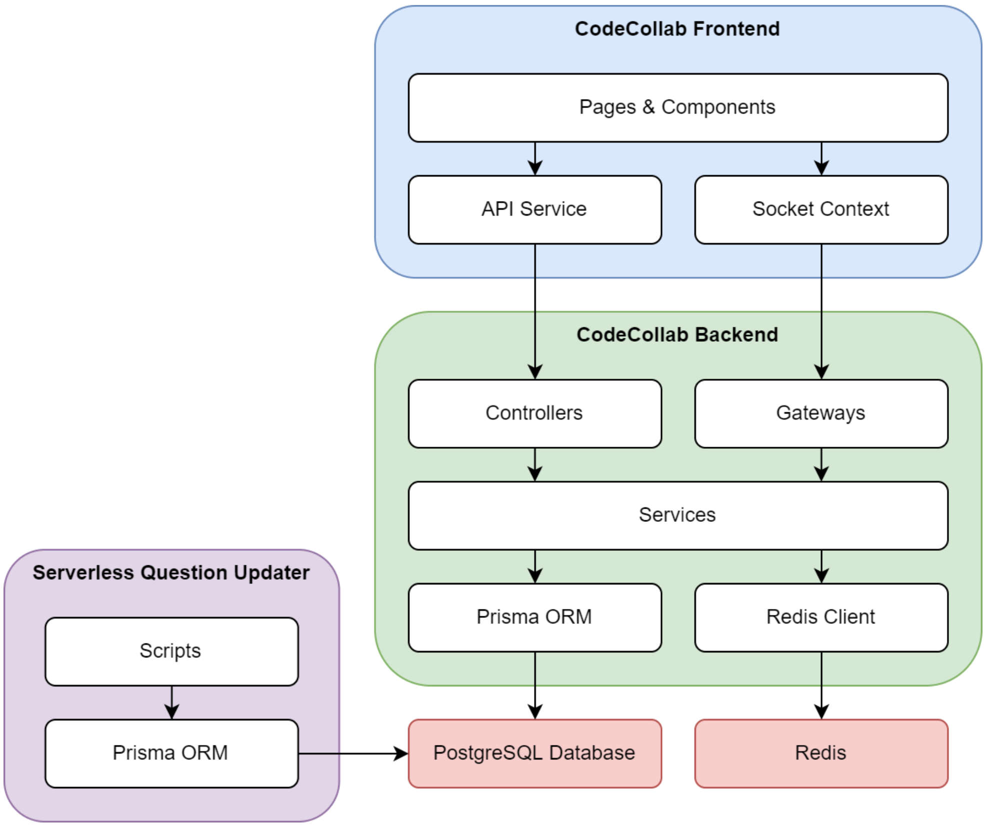


Figure 10: System architecture diagram

The application uses a web model-view-controller (MVC) architecture. It comprises three key distinct components which have all been written in TypeScript and are each contained in a folder within a monorepo.

1. A single page application (SPA) frontend which contains the view aspect of the application.
2. A monolithic backend serving both HTTP and WebSocket traffic which contains the controller and model aspect of the application.
3. A serverless component to handle the automatic periodic updating of the question database.

Apart from the subfolders containing the three different components, a fourth subfolder, shared, is used to coordinate types between the frontend and backend components.

Key System Architecture Decisions

Use of a Monolithic Backend

The backend was designed using a monolith architecture rather than a microservice architecture to reduce the amount of effort spent on writing boilerplate code and simplify communication between backend services. Using a monolith architecture also simplifies the local development environment and deployment process, allowing developers to onboard onto the project faster. As the project is still in its minimum viable product (MVP) phase, being able to dedicate more resources to developing user-facing features and functionality would increase the chances of a product launch being successful and attracting users.

Use of a Single SQL Database

The application uses a single SQL database rather than a NoSQL database or multiple databases. The main factor was that a single SQL database offers better consistency than a NoSQL database or multiple databases, reducing the risk of incorrect application layer logic resulting in corrupted data. To achieve high performance and scalability for frequent input/output (I/O) intensive operations, a Redis store is used as a supplementary database for ephemeral data such as session management.

Standardization of the Use of TypeScript

Another key system level decision was the use of a single programming language, TypeScript, throughout the application. A single programming language reduces the mental overhead for developers from having to context switch between different languages frequently. TypeScript was chosen for its type safety which reduces the chances of run time errors in exchange for a steeper learning curve for programmers new to the language.

As part of ensuring type safety throughout the entire application, the application uses a shared folder which contains type declarations for the request and response of API endpoints exposed by the backend. This allows both the frontend and backend projects to import a common set of type declarations without having one to import the other, thereby reducing import coupling.

4.2.2. Frontend

Figure 11 presents the architecture of the frontend and showcases the key technologies used.

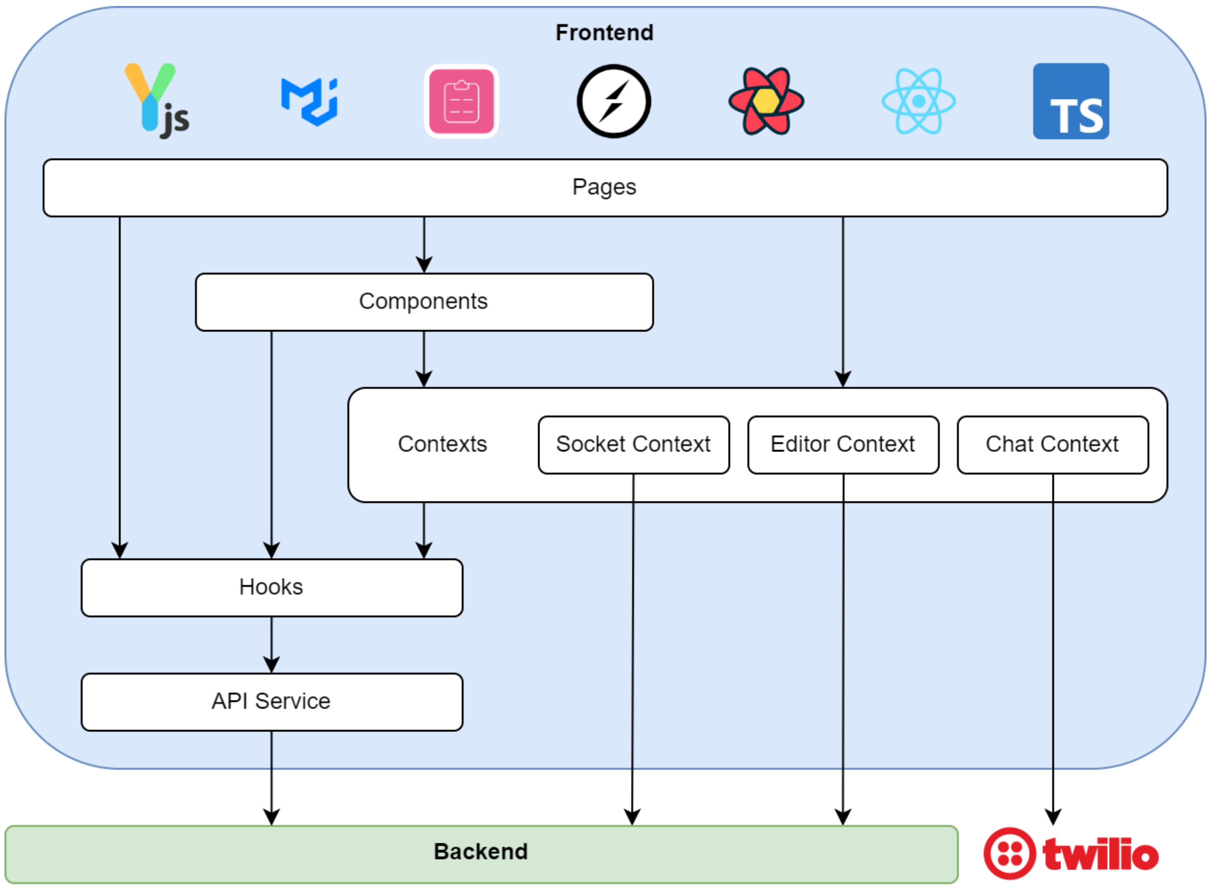


Figure 11: Architecture diagram of the frontend component

The frontend was developed using React[4] and follows a hybrid layered component-based architecture. Frontend elements are grouped into the following four layers:

1. Services & Hooks: These components provide low-level and core functionalities such as processing API requests and responses.
2. Contexts: These components maintain and manage states that are used across multiple components and pages.
3. Components: Reusable UI elements that can be integrated into different pages across the application.
4. Pages & Layouts: Organizes and coordinates the placement and integration of components.

[4] See https://www.npmjs.com/package/react

Key Frontend Architecture Design Decisions

Use of Hybrid Layered Component-Based Architecture

The frontend follows a component-based architecture which is further organized into different layers. The component-based architecture encourages the development of reusable and consistent UI elements, making it easier to maintain a consistent design language throughout the UI and significantly reducing code duplication. On the other hand, grouping of components into different layers helps to encourage separation of concerns where core functionality such as API call handlers and hooks are separated from UI elements such as components and pages.

Use of Functional Components

The React components are implemented using functional components rather than class-based components as it reduces the amount of boilerplate code required. Functional components also encourage the use of component composition[5] which is largely favored over inheritance as seen from the Gang-of-Four principles. Well-written functional components can also be easier to read than their class-based counterparts as the construction of a component evaluates the function from the start of the function to the end sequentially. However, the use of functional components may result in a steeper learning curve for developers who are already used to writing class-based components due to the introduction of hooks.

Use of React-Hook-Forms and React Query

The frontend uses two popular React libraries, React Hook Form[6] and React Query[7], to help manage form states and API calls respectively.

The use of React Hook Form greatly simplifies the management of form states for functional components by managing the context of a form, form field validation, and form submission hooks behind a developer friendly interface.

The use of React Query helps to manage both queries and mutations and provides a simple interface to cache query results, therefore reducing unnecessary API calls during component re-renders.

[5] Source: https://www.educative.io/blog/react-component-class-vs-functional#which-to-use
[6] See: https://www.npmjs.com/package/react-hook-form
[7] See: https://www.npmjs.com/package/react-query

4.2.3. Backend

Figure 12 presents the architecture of the backend and showcases the key technologies used.

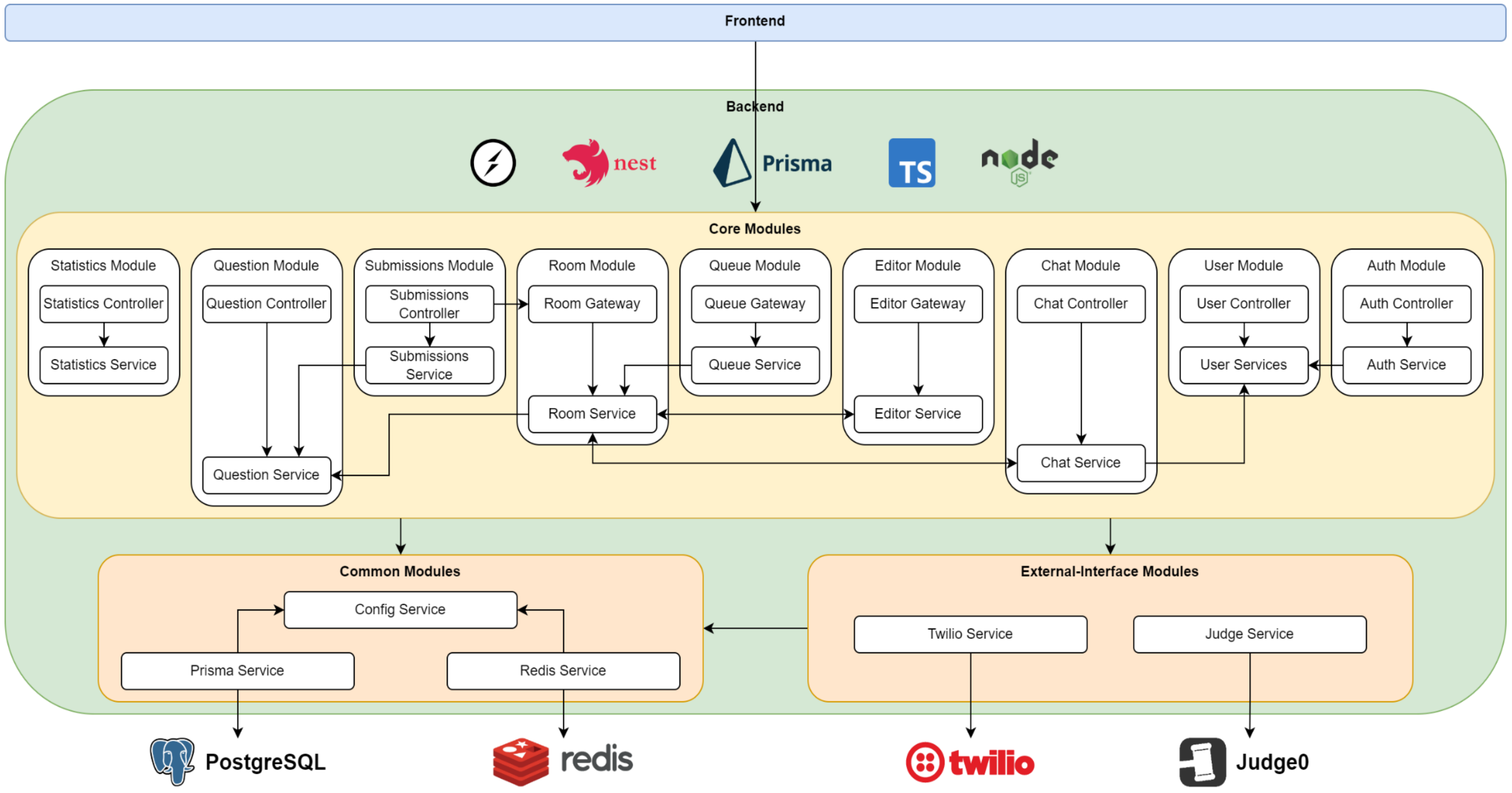


Figure 12: Architecture diagram of the backend component

The backend was developed using the NestJS[8] framework along with Prisma[9] as the database ORM. The backend is largely organized using domain-oriented modules with the exception of a few common modules.

Key Backend Architecture Design Decisions

Use of NestJS Framework

The NestJS framework was chosen over basic Express for a variety of reasons, one of which is the opinionated structure that it enforces with modules, controllers, and providers, allowing developers to concentrate on implementing features rather than debating about how to organize and structure code. Additionally, dependency injection is a fundamental concept in NestJS and is prevalent throughout the structure it enforces. The dependency injection system which NestJS exposes to developers greatly reduces the amount of boilerplate code required to gain the benefits of dependency injection such as greater testability, code reuse, and decoupling of implementations. Furthermore, NestJS also offers integrations with a wide range of tools such as PassportJS[10] and HTTP sessions for authentication, Socket.IO[11] for WebSocket support, and class-validator[12] for validation and sanitation of inputs. However, the use of NestJS does increase the learning curve required for developers new to the framework.

Use of Prisma

Prisma was chosen over Sequelize (sequelize-typescript) and TypeORM for managing database interactions. Compared to Sequelize and TypeORM, Prisma has significantly better type safety when it comes to the result of queries involving joins. In both Sequelize and TypeORM, queries which involve retrieving nested data return an object with more fields than the specified return type of the function.[13] In particular, the nested relations are not part of the return type of the query function and hence cannot be easily accessed when using TypeScript. Another benefit of using Prisma is the decoupling of model definitions from code. In Prisma, the data model and migrations are handled separately from the backend code and query engine. Therefore, this allows the generated Prisma client, the query engine, to be used across multiple code bases, such as in the serverless codebase as well, allowing strong type safety through the project.

[8] See https://www.npmjs.com/package/@nestjs/core
[9] See https://www.npmjs.com/package/prisma
[10] See https://www.npmjs.com/package/passport
[11] See https://www.npmjs.com/package/socket.io
[12] See https://www.npmjs.com/package/class-validator
[13]See https://www.prisma.io/dataguide/database-tools/evaluating-type-safety-in-the-top-8-typescript-orms#typeorm for more information.

Use of Domain Oriented Modules

In contrast to the frontend architecture, the backend is mostly organized using domain-oriented modules. Each of the domain modules contain an external interface, usually known as a controller for REST APIs or a gateway for WebSocket connections, which corresponds to the controller layer in the MVC architecture. A domain module also contains one or more services which contain the domain specific logic, and these services form the model layer of the MVC architecture. The controllers and gateways route requests to the appropriate methods in the services and perform a minimal amount of business logic. Interactions between modules have mostly been deferred to the service layer instead to reduce domain specific logic in the controller layer.

In addition to the modules that take care of the domain specific logic, there are a set of globally accessible singleton common and external modules which provide interfaces to the database, configuration settings, and external dependencies. These modules provide the business-related modules a simplified interface with some built-in error handling to reduce the amount of code duplication across business-modules.

By using domain-oriented modules instead of a layered architecture for the backend components, the application can be more easily converted to use a microservice architecture should the need arise. This is because each module encapsulates the components which are key to the functionality of a particular domain and can be used as the basis of a microservice.

4.3. Design of Key Components

4.3.1. Authentication: User Accounts

User accounts creation, email verification, and password reset requests are handled by the user module, which consists of the `UserController`, `UserService`, `VerificationService`, and `ResetPasswordService` classes. The user module also makes use of an external service, Twilio, to send verification and password reset emails. The Twilio Verify[14] service is used to manage verification emails and password reset emails as it handles the creation and verification of one-time-passwords (OTPs) sent in the email.

Figure 13 shows a class diagram of the classes in the user module and their key relations with other services. The `UserService` facilitates interactions with the database for creating, updating, and retrieving users, while the `ResetPasswordService` and `VerificationService` interact with `TwilioService` to send emails to users to reset their password and verify their email respectively.

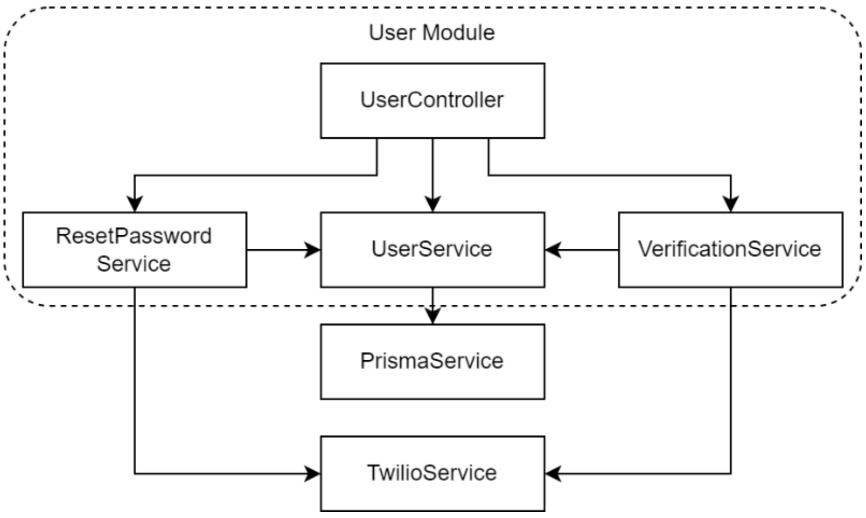


Figure 13: Class diagram of the user module and related classes

[14] See https://www.twilio.com/verify

New users can sign up for an account using their email address in the form shown in Figure 14 [FR-Auth-1]. Users are prompted to provide their email address, name, and a password.

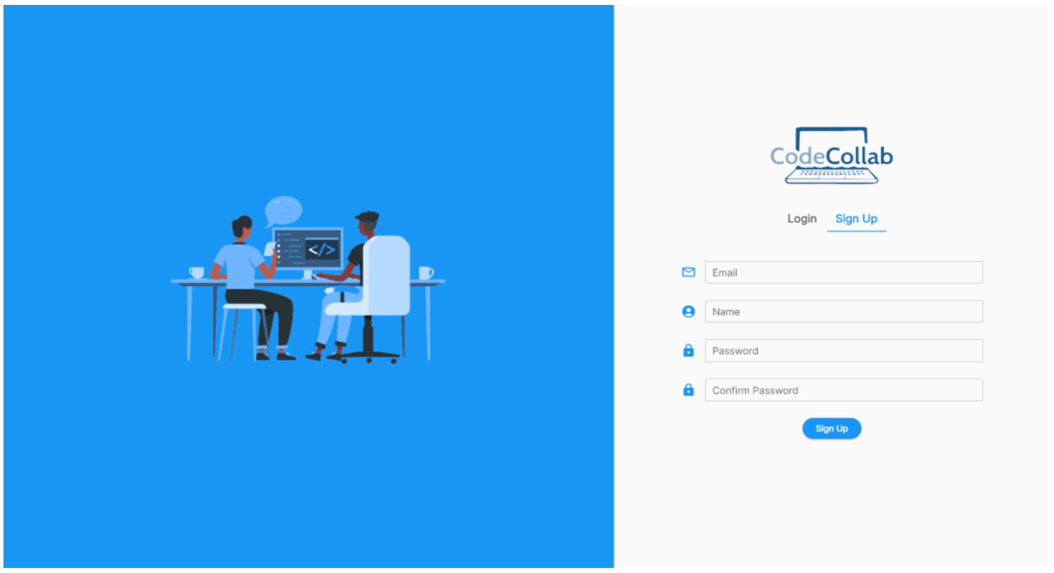


Figure 14: Screenshot of the sign up page

Figure 15 shows the key interactions between classes for the user sign up flow, excluding email verification. Some frontend interactions such as the use of hooks and API service have been omitted for brevity.

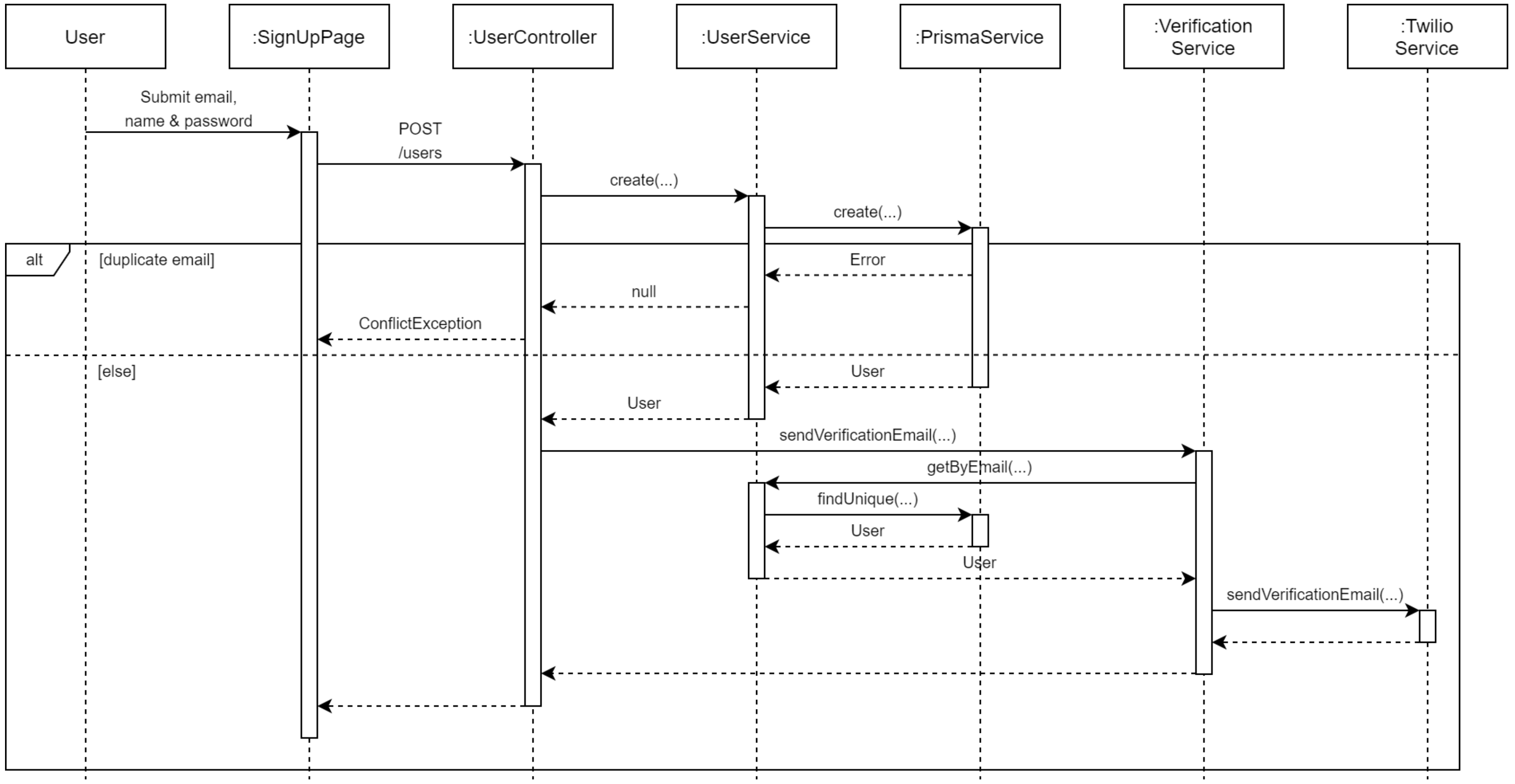
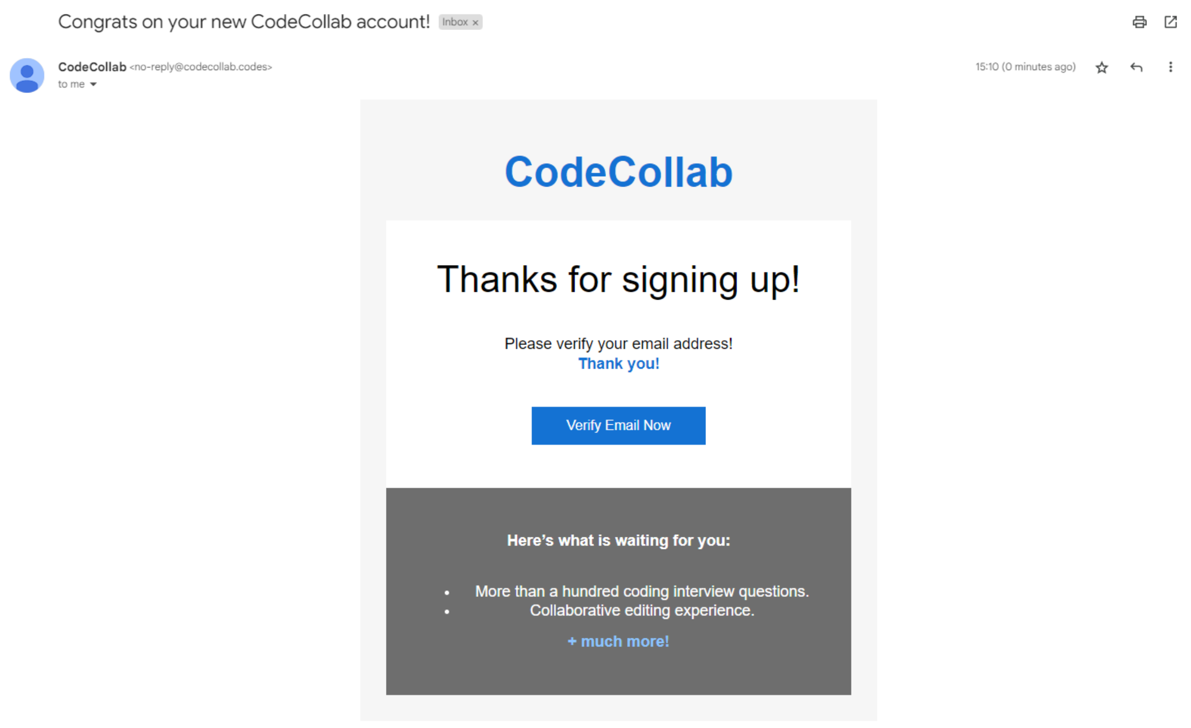


Figure 15: Sequence diagram of the user sign up feature

As users are uniquely identified by their email address [FR-Auth-2], the account creation process relies on the database to throw a unique constraint violation error on the email address to detect and prevent accounts with duplicate email addresses. By relying on the database to detect and throw such an error, the possibility of a race condition causing data integrity issues should a manual check have been performed prior to user creation instead is removed.

In the `VerificationService::sendVerificationEmail` method the user object is once again retrieved from the database as the method accepts an email as an argument rather than a User object. This design allows the resend verification email workflow [FR-Auth-8] to be simplified in the controller's view as it only needs to call the `VerificationService::sendVerificationEmail` with the email address received from the frontend, greatly reducing the amount of domain logic required in the controller layer. Figure 16 shows a sample verification email.


Congrats on your new CodeCollab account! Inbox ×

CodeCollab <no-reply@codecollab.codes>
to me

15:10 (0 minutes ago)

CodeCollab

Thanks for signing up!

Please verify your email address!
Thank you!

Verify Email Now

Here's what is waiting for you:

- More than a hundred coding interview questions.
- Collaborative editing experience.

+ much more!

Figure 16: Sample verification email

Figure 17 shows the sequence diagram for the email verification flow which new users must complete before being able to log in to their accounts [FR-Auth-7].

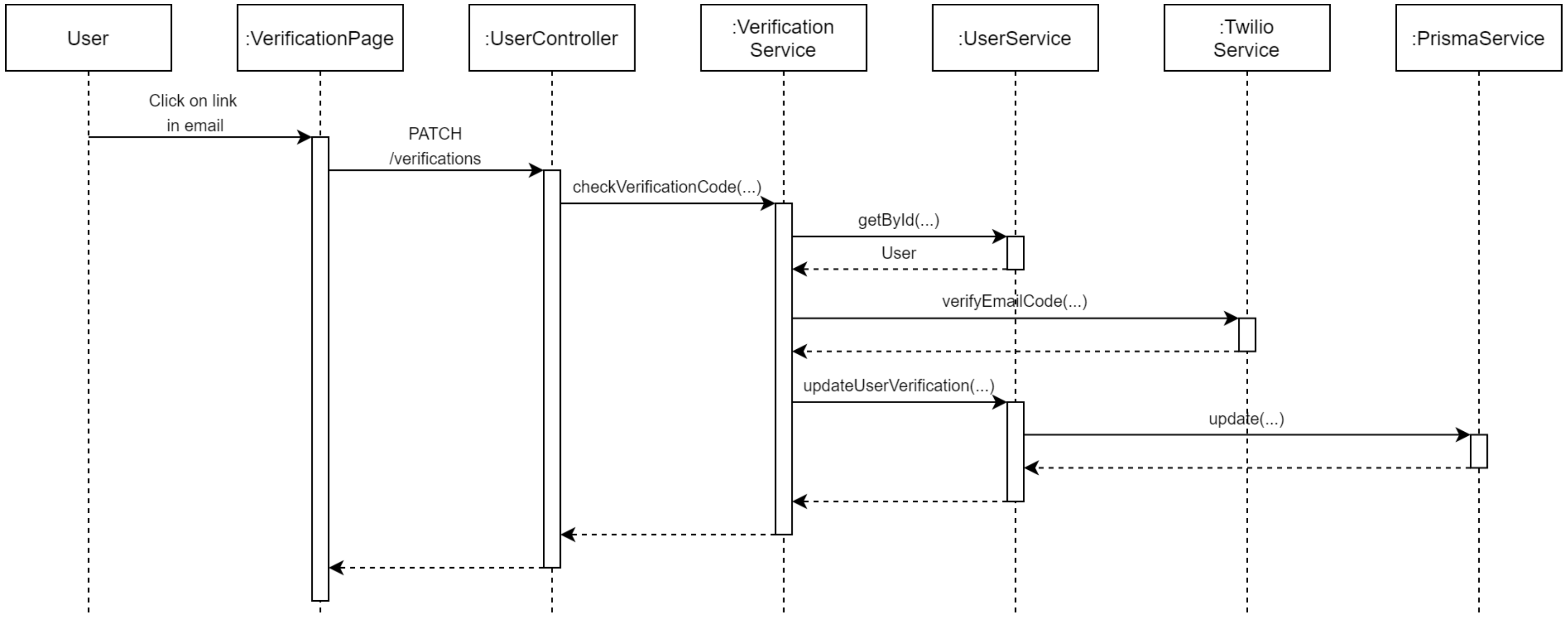
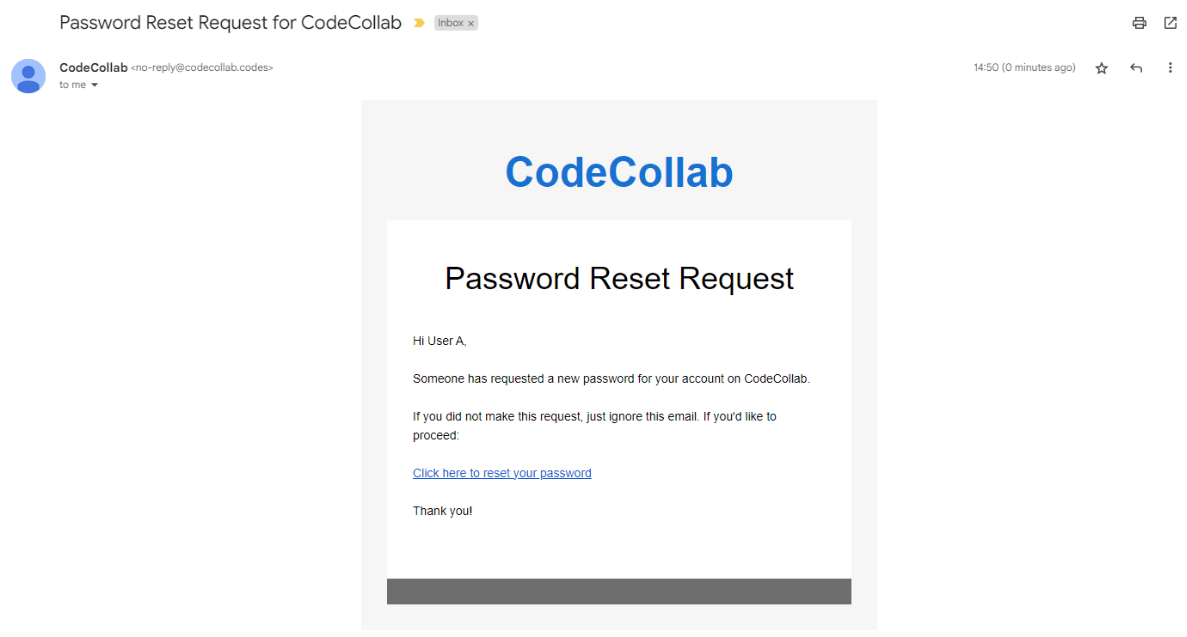


Figure 17: Sequence diagram of email verification process

When the user clicks on the link provided in the email, the user is redirected to the verification page with their user ID and the one-time-password as URL query parameters. On page load, a `PATCH` request is made automatically to the server with the URL parameters in the request body. The backend then verifies the returned code with Twilio and updates the user state in the database.

Users can also reset their password if their account has been locked (due to too many incorrect password attempts), or if they have forgotten their password [FR-Auth-6]. Figure 18 shows a sample password reset email. The password reset feature follows a similar workflow to the verification email flow, using a one-time-password contained within the email link to authenticate the user's identity before updating the user's password.


Password Reset Request for CodeCollab Inbox ×

CodeCollab <no-reply@codecollab.codes>
to me

14:50 (0 minutes ago)

CodeCollab

Password Reset Request

Hi User A,

Someone has requested a new password for your account on CodeCollab.

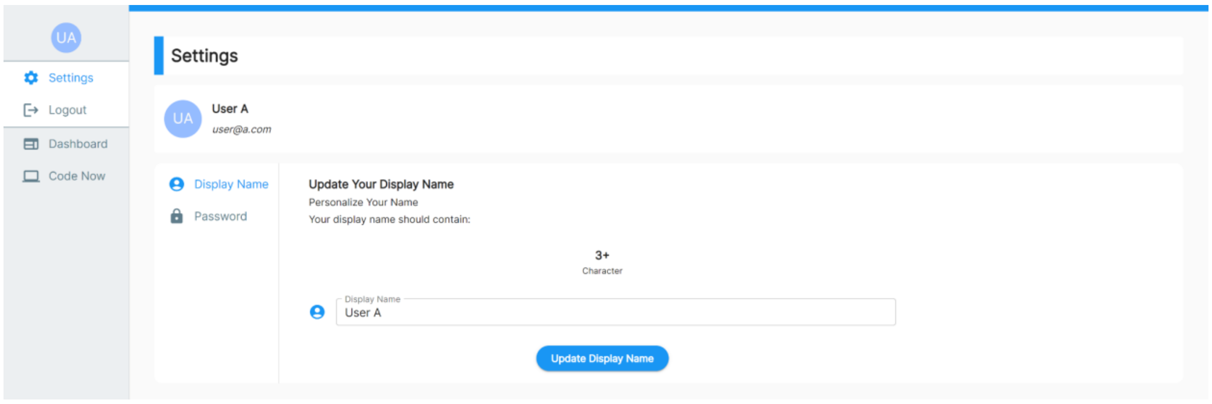
If you did not make this request, just ignore this email. If you'd like to proceed:

Click here to reset your password

Thank you!

Figure 18: Sample password reset email

Users can also update their display name or password in the account settings page shown in Figure 19 and Figure 20 respectively [FR-Auth-5 / FR-Profile-1]. As passwords are sensitive information, users have to provide their current password to update their password to prevent time-of-check to time-of-use (TOCTOU) attacks.

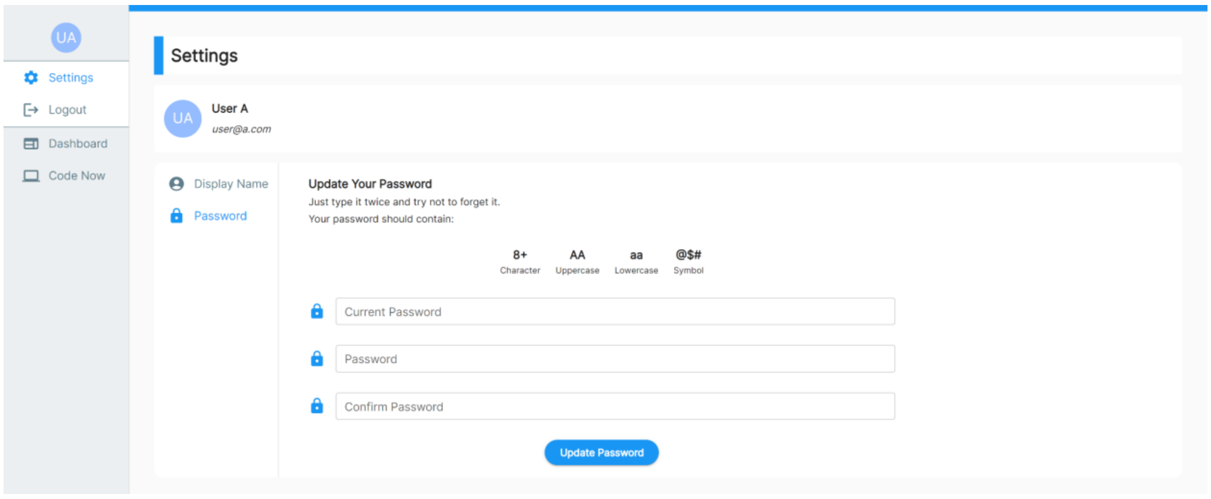


Personalize Your Name

Your display name should contain:

Display Name: User A

Figure 19: Screenshot of update display name page



Just type it twice and try not to forget it.

Your password should contain:

Figure 20: Screenshot of update password page

4.3.2. Authentication: Session Management

User authentication is managed using HTTP sessions, the PassportJS middleware, and is backed by Redis. Figure 21 shows the workflow for the binding of sessions during the login process.

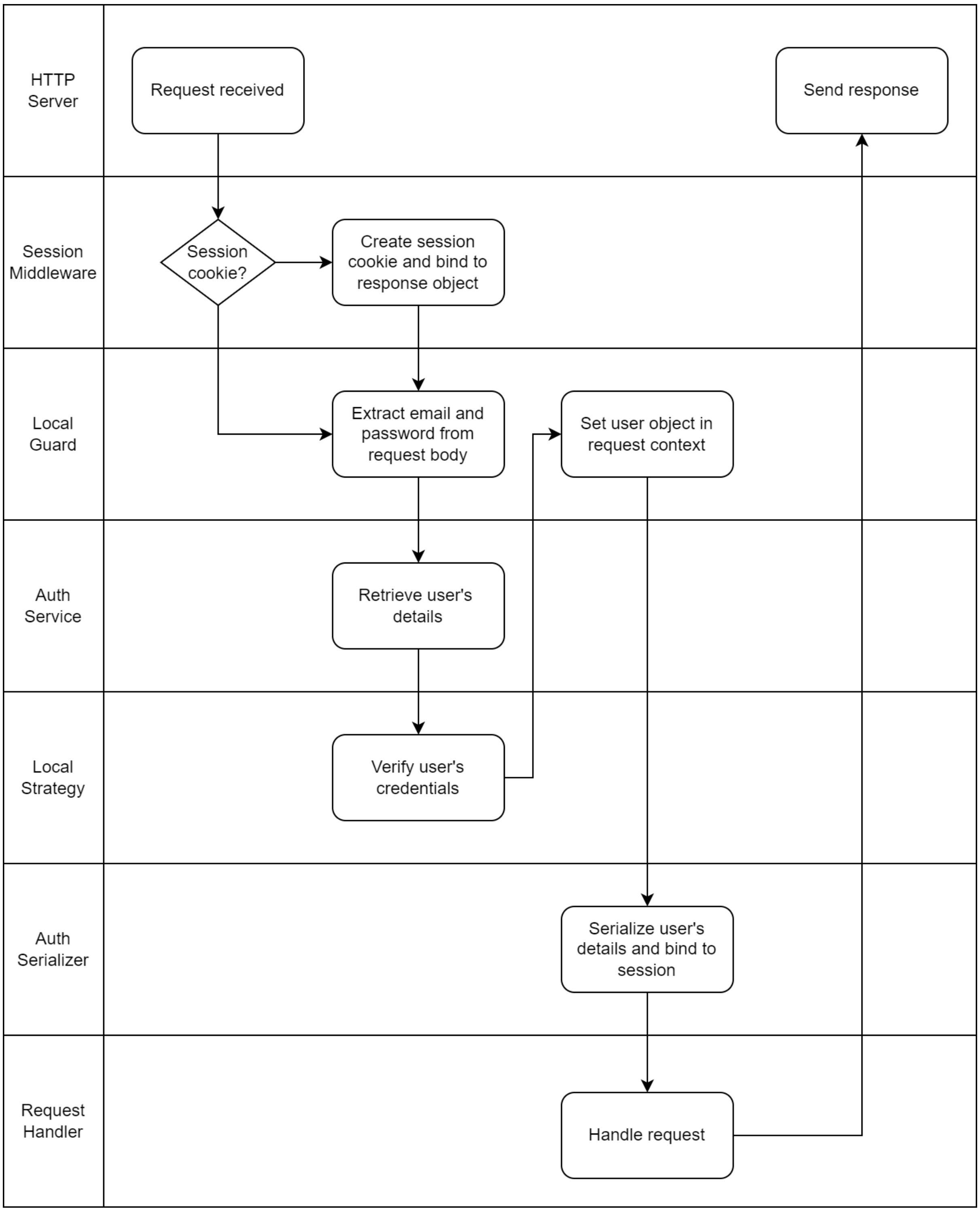

HTTP Server
Session Middleware
Local Guard
Auth Service
Local Strategy
Auth Serializer
Request Handler
Request received
Send response
Session cookie?
Create session cookie and bind to response object
Extract email and password from request body
Set user object in request context
Retrieve user's details
Verify user's credentials
Serialize user's details and bind to session
Handle request

Figure 21: Swimlane diagram of login workflow

When a user makes a `POST` request to the login endpoint, it is intercepted by the `SessionMiddleware` which creates a new session if no session cookie is present in the request. The request then carries on to the `LocalGuard` which extracts the email and password from the request body and invokes the validate function of `LocalStrategy` [FR-Auth-3]. This function makes a call to the `AuthService` to retrieve and verify the user's credentials. Once verified, the user's details are then bound to the session and serialized for storage in the session store which is backed by Redis. The ID for the newly created session will be stored as a cookie on the frontend via the cookie header in the response. Subsequent requests containing the session cookie will automatically load and deserialize the user object from the session store, indicating that the user is logged in. The user information bound to each subsequent request can then be used for authorization when necessary.

As session information is completely managed on the backend, logging out from the system requires a call to the backend. The logout endpoint deletes the user information, which is bound to the session, effectively logging the user out [FR-Auth-4].

Apart from authenticating HTTP requests, Socket.IO connections are also fully authenticated. This is achieved using a custom Socket.IO adapter, `SessionSocketAdapter`, which inserts the PassportJS and session middleware into all namespaces created through NestJS. The middleware utilizes the session cookie sent during the handshake of the WebSocket connection to authenticate the user and binds the user object to the Socket.IO client socket object, preventing users from impersonating others.

Use of Socket.IO Middleware for Authentication

By performing the authentication of users in the middleware layer, unauthenticated users are prevented from initiating a WebSocket connection with the server which provides better security over authenticating them using the connection event. In the latter design, malicious actors may have a short window, between the connection event and the authentication check, for which they remain connected to the server and can send malicious payloads which will be received by the server. By using a middleware, they receive a connection error and do not establish a successful connection to the server. Using a middleware also ensures that the user's identity is available for access by all event handlers subsequently without worry of any race conditions. However, using a middleware comes at the cost of added complexity despite reducing code duplication since authentication is centralized.

Use of Sessions for Authentication

Another key design decision is the use of sessions over JSON web tokens (JWTs) to manage user authentication. The use of backend managed sessions allows for logout events to take effect immediately and reduces the chances of a malicious actor misusing a stolen session cookie. For JWTs to achieve the same effect, a blacklist table can be maintained to record JWTs that should be invalidated due to logout events. However, this also incurs the same overhead as it involves checking a database or in-memory store. Alternatively, JWTs can be implemented using a short-lived token along with a refresh token, but this still introduces additional complexity and refresh tokens must still be blacklisted upon logout events. Another benefit of using sessions is the ability to store a greater amount of user state and potentially sensitive information without having to re-retrieve the information from the database or store it in the JWT where the contents can be read by the user.

4.3.3. WebSocket Connections

WebSocket connections are handled using the Socket.IO library due to the first-class support offered by NestJS.

On the backend, each module that requires WebSocket connections, except for the editor module, is managed using its own Socket.IO namespace. The application uses a series of custom Socket.IO adapters to support authentication (see section 4.3.2) and coordination between multiple server instances. Figure 22 shows the relationships between the custom adapters written for the application.

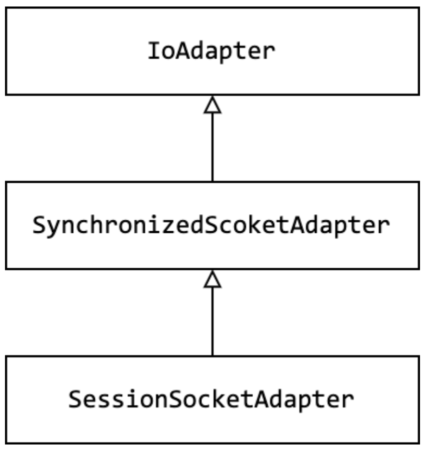


Figure 22: Class diagram of Socket.IO adapters

The `SynchronizedSocketAdapter` class extends the default `IoAdapter` class exposed by the NestJS library to provide support for broadcasting Socket.IO events to rooms and namespaces distributed across multiple server instances. It utilizes the @socket-io/redis-adapter[15] library which creates a publish-subscribe topic between all server instances. Events emitted from a server are published to the channel and received by all other server instances, who can then emit the events to the appropriate clients connected to them.

Figure 23 shows how two clients, A and D, can receive the same event broadcasted the Socket.IO room event though they are connected to different server instances. When an event is emitted by Server Y to Room 1, the event is also published to a Redis publish-subscribe channel by the Redis Adapter library. This event is received by Server Z whose Redis Adapter

[15] See https://www.npmjs.com/package/@socket.io/redis-adapter

automatically subscribes to the channel upon initialization. The event received is emitted to all clients connected to Room 1 on Server Z, if the room exists on Server Z.

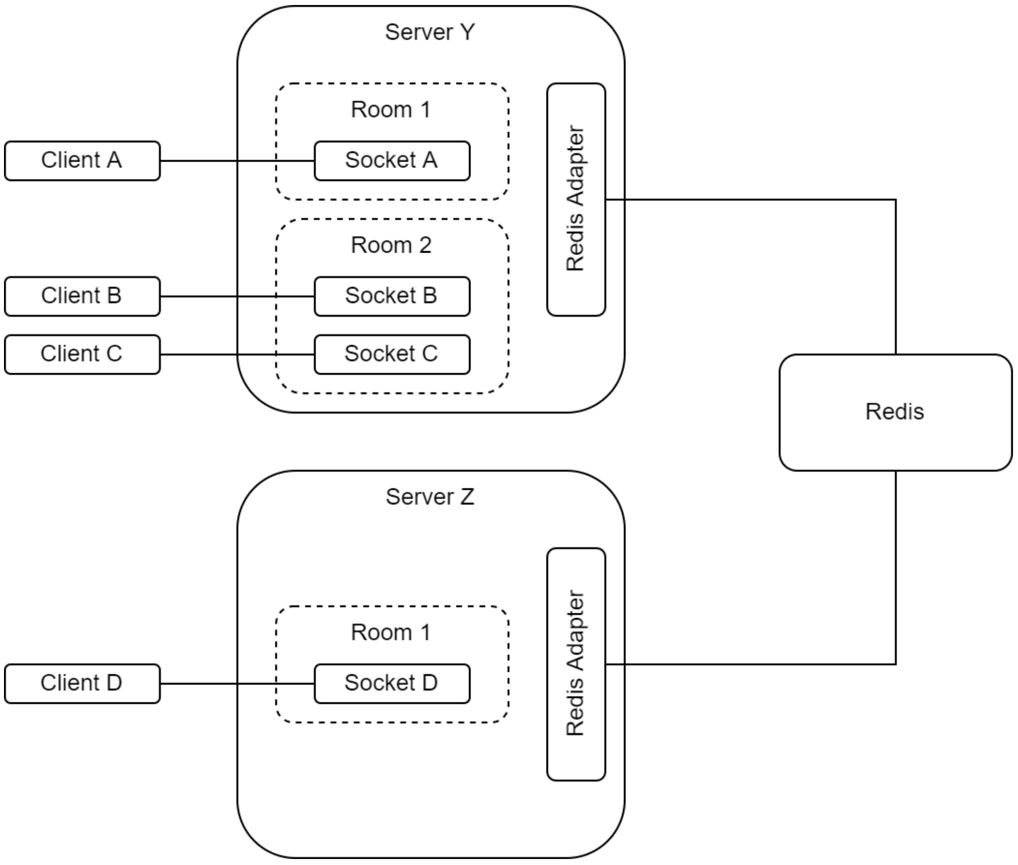


Figure 23: System diagram of inter-server communication

The `SessionSocketAdapter` class extends the `SynchronizedSocketAdapter` class to install the authentication middleware (see section 4.3.2).

On the frontend, Socket.IO connections are managed using the `SocketContext` which creates and maintains the Socket.IO clients required by the different frontend components and pages. The `SocketContext` supports multiple concurrent Socket.IO connections to different namespaces by maintaining each initialized client in a map of Socket.IO clients. Each component can request for a Socket.IO client from the `SocketContext` by specifying the target namespace. This centralized management of Socket.IO connections allows for basic error handling and proper teardown of the clients through the `SocketContext`, reducing code duplication.

4.3.4. User Matching

Users can find a match by navigating to the selection page (Figure 24), where they can select a difficulty level and a programming language [FR-Match-1 / FR-Match-6]. These are the two criteria that will be used to pair users.

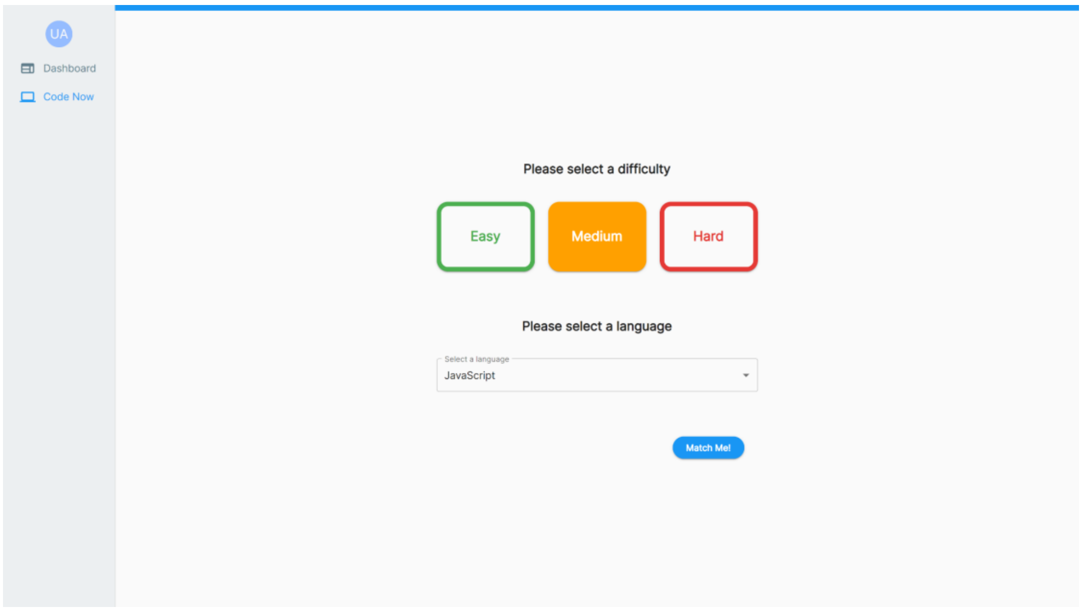
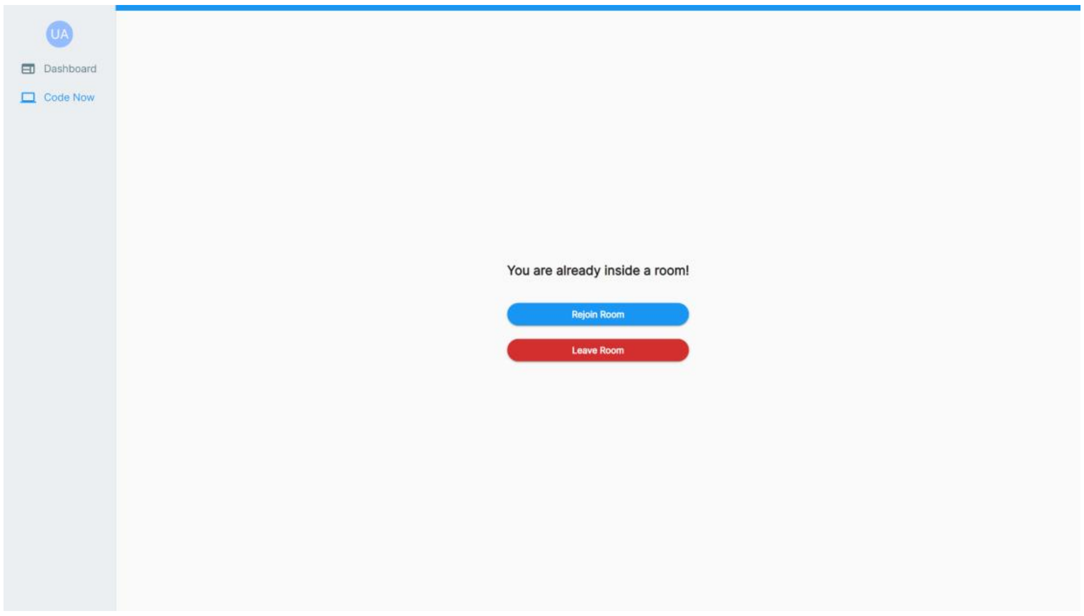


Figure 24: Screenshot of selection page

The currently supported programming languages are JavaScript, C++, Python, and Java. These languages were chosen from the languages supported by both the editor (see section 4.3.7) and code execution service (see section 0) used. The four languages were chosen to cover a wide range of potential users, such as web developers (JavaScript), low-level embedded system programmers (C++), enterprise system programmers (Java), and machine learning programmers (Python).

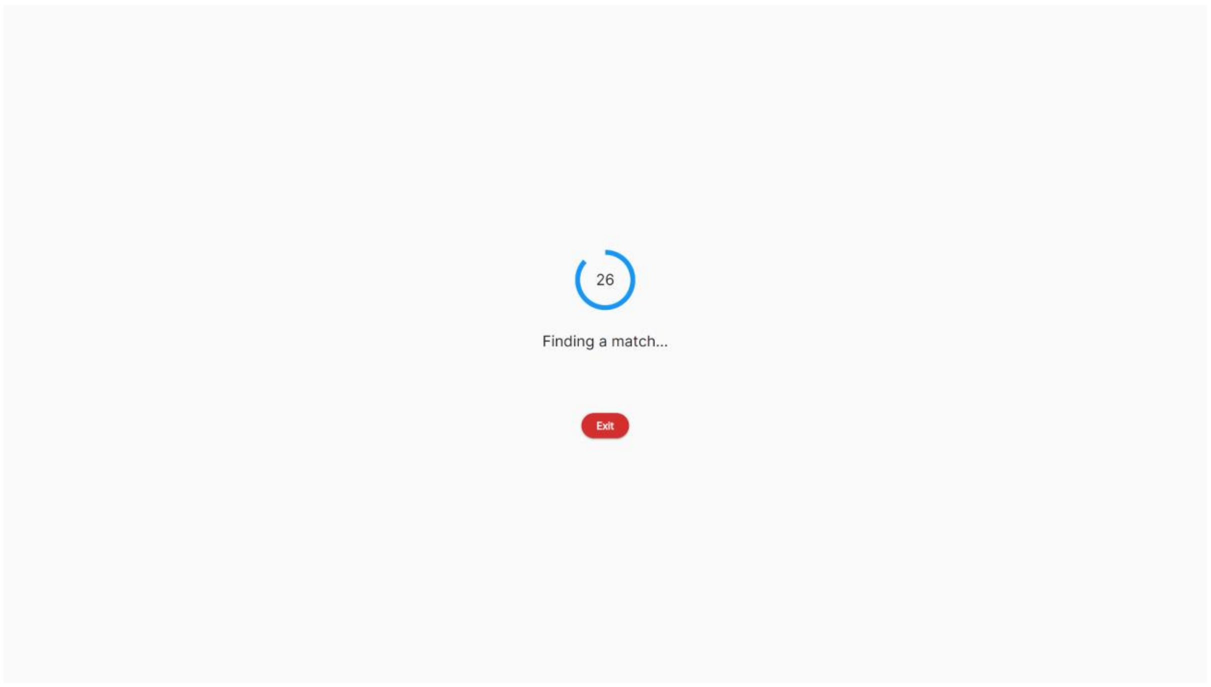
Additionally, when a user navigates to the selection page, an API request will be sent to the backend to check whether the user has an existing match (see section 4.3.5). If the user has an existing match, they will be given a choice to rejoin the room of the existing match or to leave the room before being able to find a new match [FR-Room-6] (Figure 25).



You are already inside a room!

Figure 25: Screenshot of selection page if the user has an existing match

After the user selects the difficulty level and the programming language, they will enter the queue page (Figure 26). The queue page opens a Socket.IO connection to the backend and emits an `ENTER_QUEUE` event with the selected difficulty and language as the payload. If no match is found in 30 seconds, the user will be automatically directed back to the selection page and the Socket.IO connection will be terminated [FR-Match-4].



Finding a match...

Figure 26: Screenshot of queue page

On the backend, user matching is handled by the queue module which uses Redis to achieve low latency when matching the users. Figure 27 shows the class diagram of the queue module and its related classes.

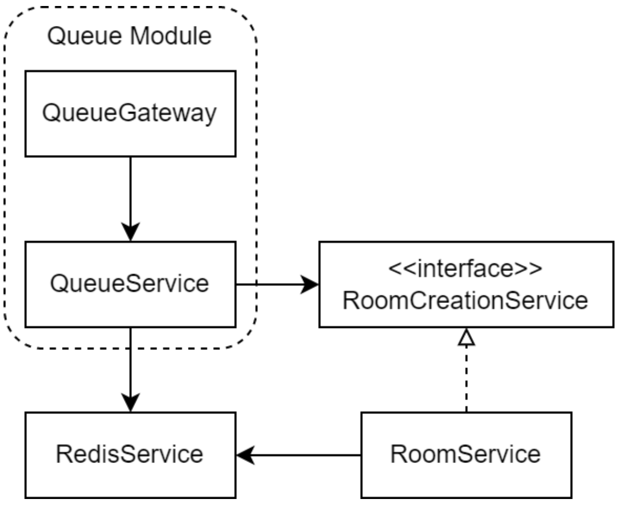


Figure 27: Class diagram of queue module and related classes

Figure 28 shows the steps taken by the `QueueService` to handle an `ENTER_QUEUE` event.

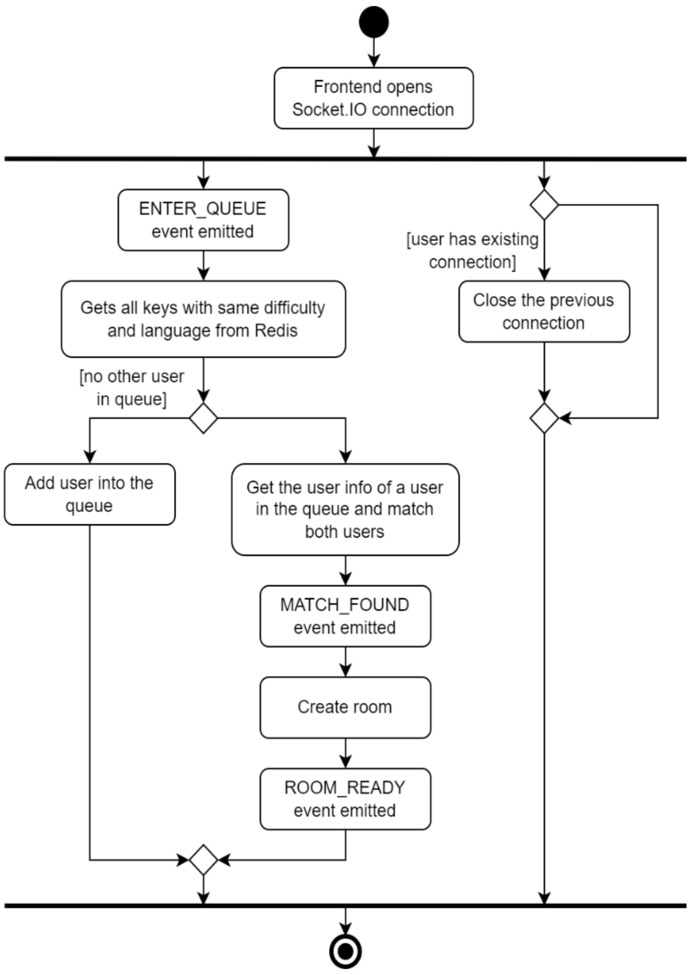

Frontend opens Socket.IO connection
ENTER_QUEUE event emitted
Gets all keys with same difficulty and language from Redis
[no other user in queue]
Add user into the queue
Get the user info of a user in the queue and match both users
MATCH_FOUND event emitted
Create room
ROOM_READY event emitted
[user has existing connection]
Close the previous connection

Figure 28: Activity diagram of the find match workflow

The `QueueService` uses Redis to store lists of users who are searching for the same difficulty and language. As the Redis store is used in other parts of the application as well, each module that uses the Redis store is assigned a namespace to prevent key conflicts.

Each user in the queue is represented using three key-value pairs in the Redis store (Table 16).

Key	Value	Usage
Difficulty | Language: Socket ID	User ID	For finding the list of matching users.
User ID	Socket ID	For knowing which socket to emit messages once a match is found.
Socket ID	Difficulty | Language	For removing a user from the queue during socket disconnection.

Table 16: Redis keys used to manage the queue system

To handle an `ENTER_QUEUE` event, the handler retrieves all keys which match on the selected difficulty and language to find the list of users currently in the same queue [FR-Match-5]. If there are no users found, the user will be added to the queue.

If a user is found in the queue, the user is removed from the queue and a `MATCH_FOUND` event will be emitted immediately to both users. The `QueueService` will proceed to create a room by calling the `RoomCreationService` (see section 4.3.5). The `MATCH_FOUND` event is used to provide users with real-time updates on the current matching status since the room creation process may take a while, therefore reducing the perceived waiting time [FR-Match-3].

Once the room is created, a `ROOM_READY` event will be sent to the frontend and the users will be redirected automatically to the coding room [FR-Match-2].

Duplicate Connection Handling Process

The system also has in place measures to prevent users from finding a match using two different browsers. If a user opens a duplicate Socket.IO connection to the queue namespace, an error message will be sent to any existing Socket.IO connections, causing them to close. This prevents users from making multiple match requests at once.

In order to detect duplicate connections from a user, all incoming connections are added to a Socket.IO room identified by the user's ID, which is taken from the session data provided by the session middleware. When a new connection is created, all existing sockets connected to the room keyed on the user's ID are disconnected before the new connection is added to the room. The keys associated with the user's previous sockets are also removed from the store. As such, the keys in the store are mainly identified using socket IDs rather than user IDs to facilitate this cleanup process.

4.3.5. Room Management

The state of a room is managed by the room module which orchestrates the creation and destruction of other room-related resources such as the editor document and chat rooms. Figure 29 shows the class diagram of the room module and its related components.

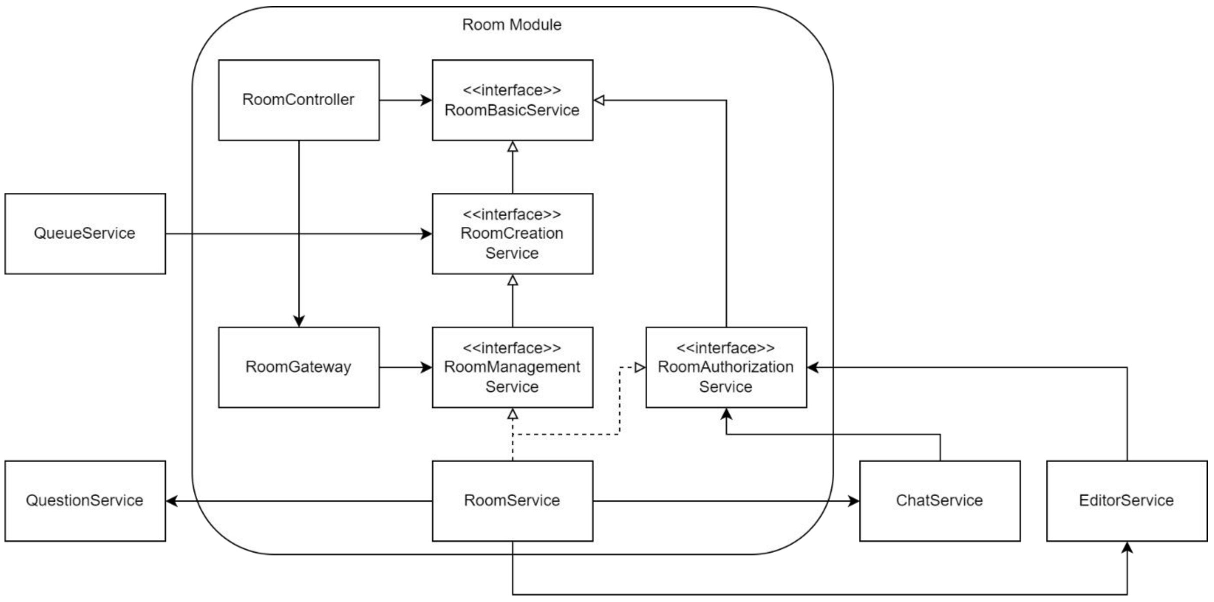

Room Module
RoomController
<<interface>> RoomBasicService
QueueService
<<interface>> RoomCreation Service
RoomGateway
<<interface>> RoomManagement Service
<<interface>> RoomAuthorization Service
QuestionService
RoomService
ChatService
EditorService

Figure 29: Class diagram of room module

The room module is one of the few places in the application where explicit interfaces are used for NestJS components rather than relying on the built-in dependency injection mechanism. This is required to prevent cyclical file imports which TypeScript does not allow, and the use of interfaces was chosen over splitting the `RoomService` class into separate classes to maintain cohesiveness and reduce code duplication. The use of explicit interfaces however leads to significantly more verbose boilerplate code, as the injection of dependencies must be manually managed, and hence is used sparingly.

The room module has two main external entry points, the `RoomController` which exposes two REST API endpoints for rejoining and leaving an existing room which is used during the selection page, and the `RoomGateway` which handles Socket.IO connections for users in a room.

Room Creation Process

A room is created when two users have been matched and the process is triggered by the `QueueService`. Figure 30 shows the key steps taken during the creation of a room.

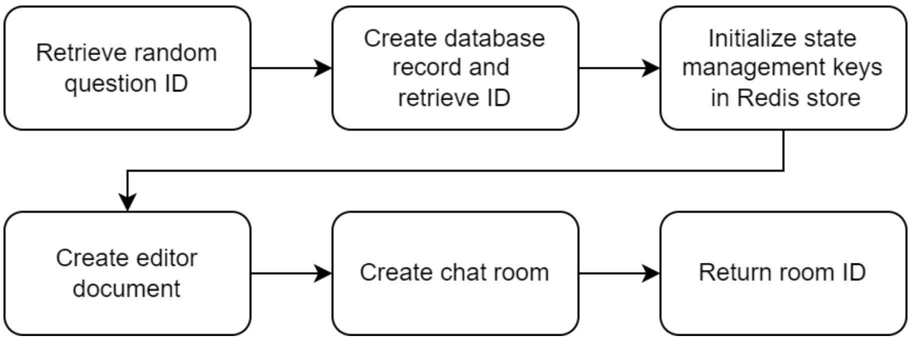

Retrieve random question ID
Create database record and retrieve ID
Initialize state management keys in Redis store
Create editor document
Create chat room
Return room ID

Figure 30: Flowchart of room creation process

Notably, the information regarding a room is stored in both the database and the Redis store. Long-term information such as session start and end times, members of the room and the question ID are stored in the database for persistence and tracking of a user's match history. Meanwhile, the Redis store tracks the dynamic state of a room during a session such as if users are connected or disconnected, or if any user has left the room.

The first step retrieves a random question via the `QuestionService` class. A room session database record is created in the database, storing information such as the question selected, the members of the room and the start time of the session. A room ID generated by the database is returned and used to initialize the following keys-values in the Redis store as shown in Table 17. Due to the large amount of state stored for each room, the namespace allocated to the room module is further subdivided into five different sub-namespaces.

Sub-Namespace	Key	Value	Usage
`PASSWORD`	Room ID	Randomly Generated Room Password	For editor WebRTC connection. (see section 4.3.8)
`LANGUAGE`	Room ID	Language	For setting frontend editor language and submission adapter.
`QUESTION`	Room ID	Question ID	For loading the question information on the frontend.
`MEMBERS`	Room ID	Set of {User ID : Socket ID}	For tracking user states in the room.
`REVERSE_MAPPING`	User ID	Room ID	For reconnecting users and room authorization.

Table 17: Redis keys used to manage room states

The keys for each room can be broken down into two main categories. The first stores static information about a room, such as the password, programming language chosen, and question ID. The second category, which refers to the `MEMBERS` and `REVERSE_MAPPING` sub-namespaces, stores dynamic information about a room and its users.

The `MEMBERS` sub-namespace tracks which users are connected to the room and their corresponding socket IDs. If a user is disconnected, the socket ID is set to a special constant. However, if a user leaves the room, the user is deleted from the set.

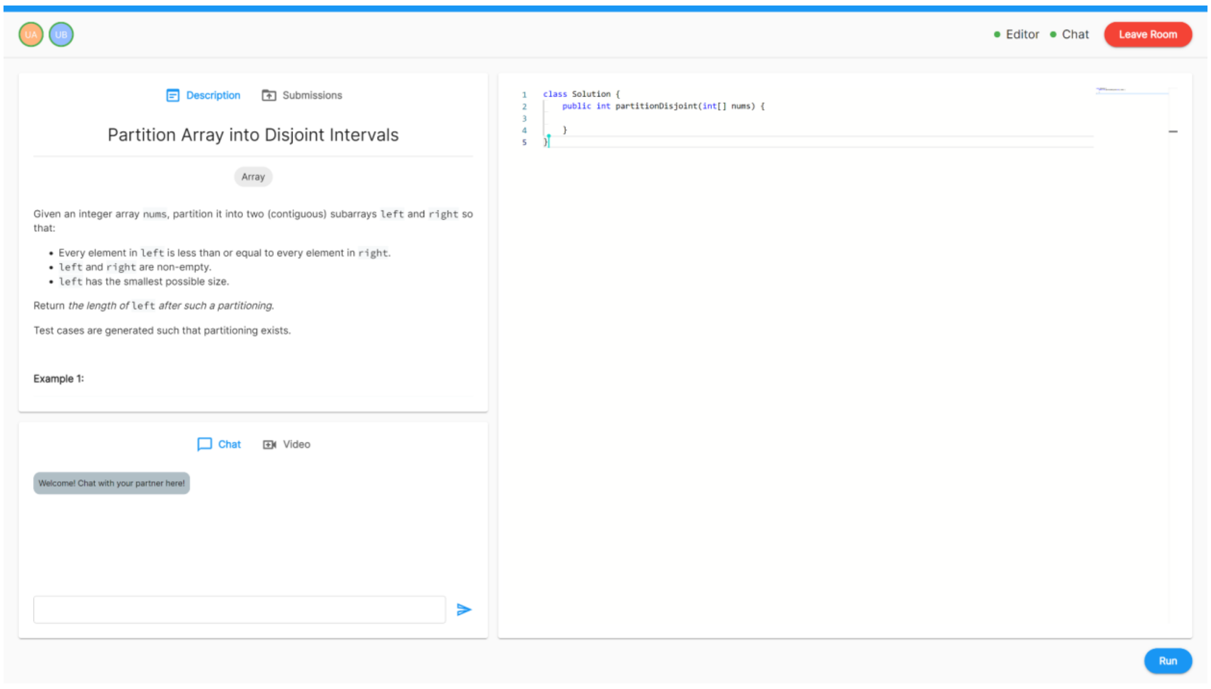
The `REVERSE_MAPPING` namespace is used to efficiently reconnect a user to their room without having to search the entire `MEMBERS` sub-namespace, therefore improving performance. This key-value pair is also used to check if users are authorized to join a room.

After the keys have been initialized the editor document and chat rooms are initialized by calling the `EditorService` (see section 4.3.7) and `ChatService` (see section 4.3.8) respectively.

After the room setup process is complete, users are automatically redirected to the coding room page (see section 4.3.4).

User Interface

Figure 31 is a screenshot of the coding room page. The user interface of the coding room page can be split into 4 main sections. At the top lies a status bar which indicates the status of members in the room and connection status of the chat and editor components (Figure 32). The top left segment of the main portion of the interface contains the question description (see section 4.3.7) and submissions information panel (see section 4.3.10). The bottom left segment contains the chat component (see section 4.3.9) and the right side of the page contains the shared editor (see section 4.3.8).



Partition Array into Disjoint Intervals

Array

Given an integer array nums, partition it into two (contiguous) subarrays left and right so that:

- Every element in left is less than or equal to every element in right.
- left and right are non-empty.
- left has the smallest possible size.

Return *the length of* left *after such a partitioning.*

Test cases are generated such that partitioning exists.

Example 1:

Welcome! Chat with your partner here!

```
class Solution {
    public int partitionDisjoint(int[] nums) {

    }
}
```

Figure 31: Screenshot of room page

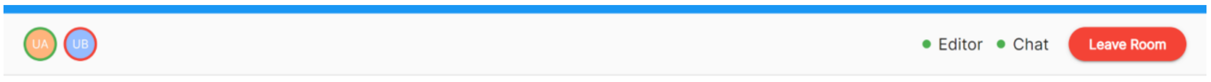


Figure 32: Screenshot of room page status bar

The room status bar contains information such as the users in the room, each shown as an avatar with their initials on the left side of the status bar. The colored ring around each avatar indicates if the user is connected to the room, with a green ring indicating the user is connected and a red ring indicating that the user is not connected. The right side of the bar contains the "Leave Room" button and two status indicators to provide the user with visual feedback on their connection to the chat and editor servers. Additionally, when a user disconnects, connects, or leaves the room permanently, a notification appears at the top of the user's screen (Figure 33).

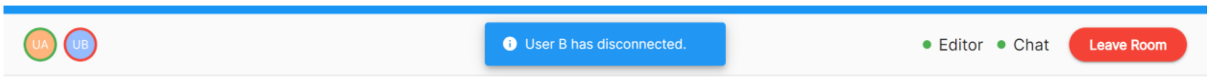


Figure 33: Screenshot of notifications in room page

Disconnections & Reconnections

When a client loads the room page, a Socket.IO connection is created to the `RoomGateway` and a `JOIN_ROOM` event is emitted from the client. The event handler updates the set stored in the `MEMBERS` sub-namespace with the socket ID to indicate that the user is connected to the room. The entire state of the room, which includes the connection status of all members, the question ID, the language selected, and the password, are broadcasted to all members currently connected to the room. This allows the newly connected user to receive the entire state of the room and updates the user connection status for existing users in the room.

When a client disconnects from the room page, the disconnection event triggers the backend to update the corresponding set in the `MEMBERS` sub-namespace to indicate that the user has disconnected. A `PARTNER_DISCONNECT` event is emitted to the remaining members of the room with the ID of the user that has disconnected, which triggers a notification to appear [FR-Room-5]. In the event that a user disconnects without explicitly leaving the room (due to poor network conditions etc.), the user is able to rejoin the room by navigating to the selection page.

The system also prevents a user from connecting to a room with two separate clients. If a user who is currently connected tries to establish another Socket.IO connection to the room gateway, the older connection is forced to disconnect. This is achieved in a similar fashion to the duplicate connection prevention mechanism in the queue module (see section 4.3.4). However, this leaves a possible race condition in the management of user connection states of the room. In particular, if the older socket disconnects after the newer socket has emitted the `JOIN_ROOM` event, the final state of the user may be recorded as disconnected rather than connected. This has been avoided by recording the socket ID of each connection of a user in the `MEMBERS` sub-namespace rather than a Boolean value so that such a race condition cannot occur.

Leaving Room

Users can leave the room by clicking on the “Leave Room” button on the top right-hand corner [FR-Room-3]. The activity diagram below describes the flow in the backend when a user leaves the room (Figure 34).

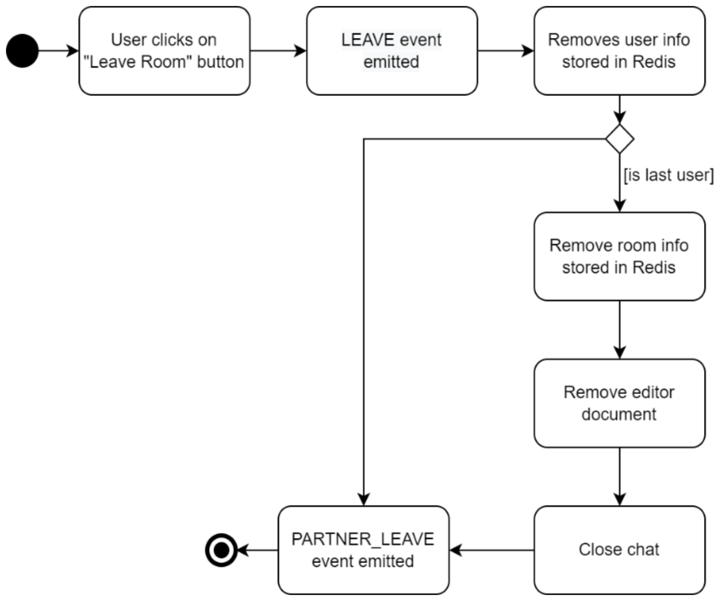

User clicks on "Leave Room" button
LEAVE event emitted
Removes user info stored in Redis
[is last user]
Remove room info stored in Redis
Remove editor document
Close chat
PARTNER_LEAVE event emitted

Figure 34: Activity diagram of leave room process

When a user clicks on the “Leave Room” button, a `LEAVE` event will be emitted. The user’s information related to the room will be removed from the `MEMBERS` and `REVERSE_MAPPING` sub-namespaces in the Redis store. A `PARTNER_LEAVE` event is then emitted by the gateway to all remaining connected clients which triggers a notification to be shown on the frontend [FR-Room-5].

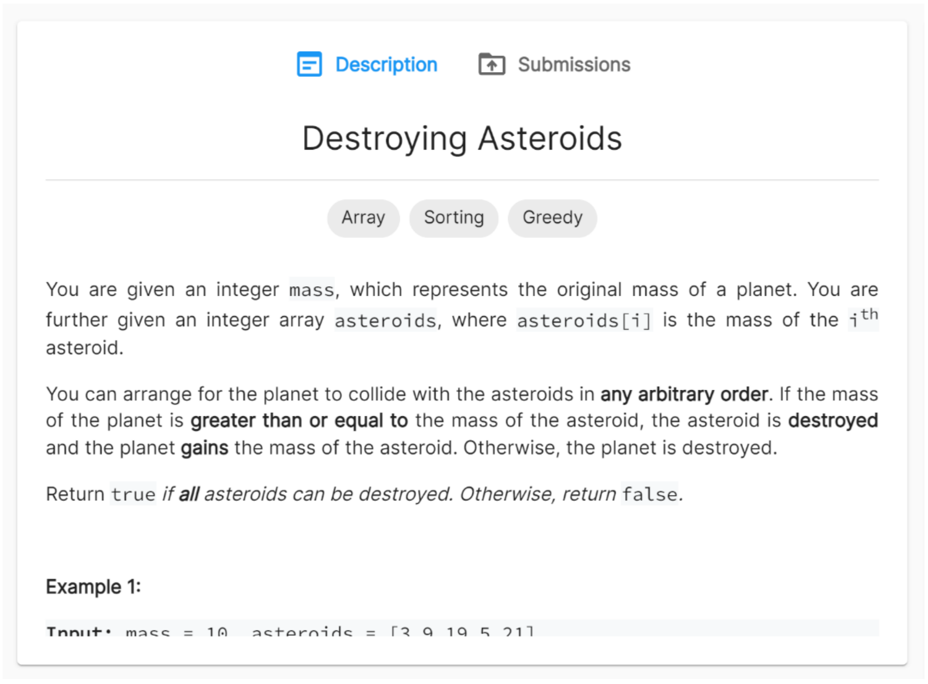
Additionally, if there are no more users in the room, identified by an empty set in the `MEMBERS` sub-namespace, the `RoomService` terminates the room and removes all associated keys in the Redis store. The termination of a room also signals the editor and chat modules to remove any resources associated with the room.

4.3.6. Question Selection and Display

The question selection feature is handled by the question module. This process is orchestrated by the room service which calls `QuestionService::getIdByDifficulty` to get a random question ID of the desired difficulty [FR-Room-1]. The random selection uses a random number generator to select a question based on an offset in from all question IDs of the selected difficulty.

When a client joins the room, the question ID assigned to the room is sent to the frontend as part of the room metadata. The room page is then able to retrieve the question to be displayed by making a `GET` request to the `/questions/:questionId` endpoint, which returns the question as a DTO containing all the required information to display the question, including the question title, category, topics, description, examples, and hints [FR-Room-4]. The use of the DTO pattern here reduces the network and disk I/O needed to retrieve the different components of a question.

As the question information is gathered from external sources (see section 4.3.11), the question description and hints are sanitized using a package called DOMPurify[16], before they are directly rendered to the frontend with all HTML tags preserved [FR-Room-2] (Figure 35). This sanitation step is required to remove any malicious scripts that may be embedded within the description or hints which may cause cross site scripting (XSS) attacks.



Destroying Asteroids

You are given an integer `mass`, which represents the original mass of a planet. You are further given an integer array `asteroids`, where `asteroids[i]` is the mass of the `i`th asteroid.

You can arrange for the planet to collide with the asteroids in **any arbitrary order**. If the mass of the planet is **greater than or equal to** the mass of the asteroid, the asteroid is **destroyed** and the planet **gains** the mass of the asteroid. Otherwise, the planet is destroyed.

Return `true` *if **all** asteroids can be destroyed. Otherwise, return* `false`.

Example 1:

```
Input: mass = 10, asteroids = [3,9,19,5,21]
```

Figure 35: Screenshot of question display panel

[16] See https://www.npmjs.com/package/dompurify

4.3.7. Collaborative Editor

The collaborative editor feature is mainly backed by Yjs[17], a conflict-free replicated data type (CRDT) implementation, Socket.IO using y-socket.io[18], and WebRTC using a patched version of y-webrtc[19]. The code editor used in the frontend is the Monaco editor[20] by Microsoft, which is the same editor used to power Visual Studio Code, one of the most popular code editors today.

During the creation of a room, a shared document is initialized on the backend and loaded with the template of the randomly selected question in the programming language chosen [FR-Editor-3]. The state of this document is then serialized and saved into the Redis store. When the `EditorContext` is loaded on the frontend, a shared document is created and is bound to both a Socket.IO connection and WebRTC connection. Figure 36 shows a state transition diagram for the shared document with respect to the backend service.

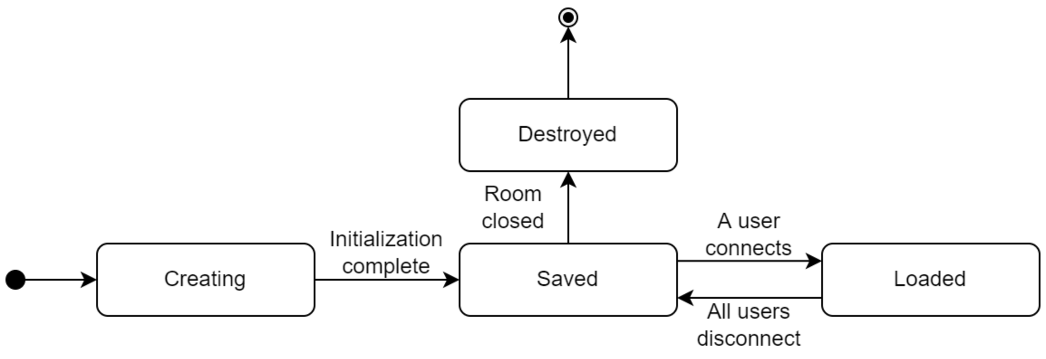


Figure 36:State diagram of shared document

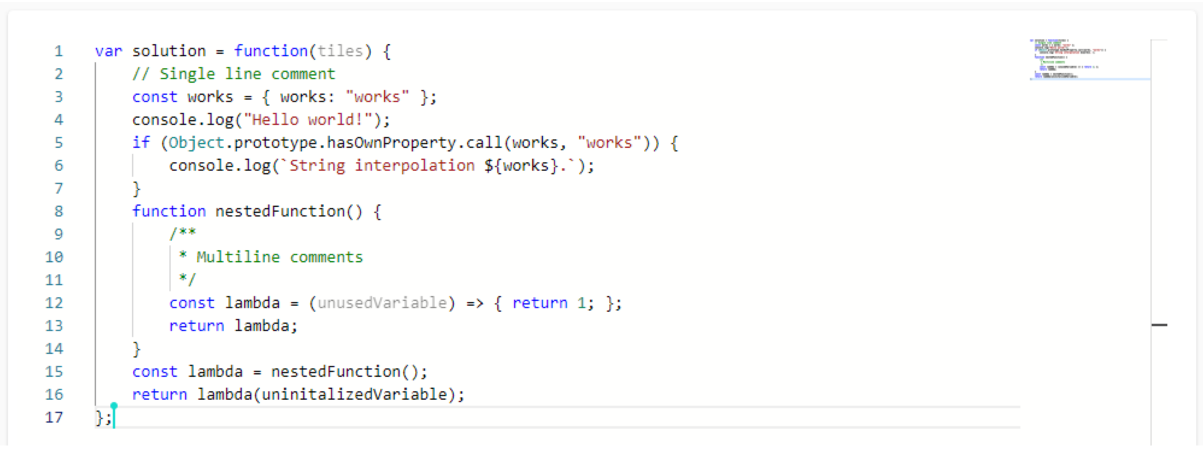
The Socket.IO connection connects to a dynamic Socket.IO namespace, uniquely identified by the room ID, in the editor module of the backend. When the first user connects to the dynamic namespace, it triggers a 'document-loaded' event on the server. This causes a restoration of the saved document state from the Redis store, loading the template into editor in the frontend. When changes are made to the document on the frontend, the changes are propagated to the server and broadcasted to users connected to the same namespace [FR-Editor-1]. Changes received by a client are automatically deconflicted and merged with their document by the Yjs library. When all users disconnect from a namespace, the "all-document-connections-closed" is emitted and this causes the document state to be saved into the Redis store again. As documents can be loaded and saved into the Redis store, the state of the shared editor persists even when both users disconnect and will be restored when any of the users reconnect [FR-Editor-4].

---

[17] See https://www.npmjs.com/package/yjs
[18] See https://www.npmjs.com/package/y-socket.io
[19] See https://www.npmjs.com/package/y-webrtc
[20] See https://www.npmjs.com/package/monaco-editor

The WebRTC connection is used as a backup synchronization mechanism between users to cater to the case where the users may have established Socket.IO connections to different instances of the backend. When such a scenario occurs, the propagation of changes via the Socket.IO connection is less reliable as the changes must be propagated via the Redis message channel mentioned in section 4.3.3. As such, the WebRTC forms a peer-to-peer connection directly between two users and propagates changes between them. However, due to a bug in the implementation of y-webrtc which causes a phenomenon known as WebRTC glare, a patched version of the library is included in the lib folder of the frontend.[21] As the WebRTC connection acts as a backup mechanism, the connection uses public signaling servers to coordinate connections between users. To ensure the privacy of the users, during the creation of a room, a random password is generated and this random password is used to encrypt signaling messages sent to the server (see section 4.3.5).

The use of Monaco editor on the frontend allows for first-party integration with Yjs documents and supports syntax highlighting for multiple languages [FR-Editor-2], therefore streamlining the development process of collaborative editor on the frontend significantly. Figure 37 shows the user interface of the Monaco editor with syntax highlighting for one of the support languages.



```
var solution = function(tiles) {
    // Single line comment
    const works = { works: "works" };
    console.log("Hello world!");
    if (Object.prototype.hasOwnProperty.call(works, "works")) {
        console.log(`String interpolation ${works}.`);
    }
    function nestedFunction() {
        /**
         * Multiline comments
         */
        const lambda = (unusedVariable) => { return 1; };
        return lambda;
    }
    const lambda = nestedFunction();
    return lambda(uninitalizedVariable);
};
```

Figure 37: Screenshot of editor user interface with syntax highlighting

<u>Use of Multiple Editor Synchronization Protocols</u>

The use of two different synchronization mechanisms allows the application to achieve both persistence of the content in the shared editor using the Socket.IO connection, as well as scalability using the WebRTC connection since users could be matched across server instances. This comes at the cost of added complexity due to the need to manage two different types of connections. However, given the importance of the functionality of the shared editor to the core purpose of the application, it is deemed that this added complexity was acceptable to deliver a well-polished feature.

[21] See https://github.com/yjs/y-webrtc/pull/39 for more information about the patch.

4.3.8. Communication Services

The textual and video chat features are largely handled by Twilio Conversations[22] and Twilio Video[23]. Figure 38 shows the key interactions between the services required to set up the Twilio Conversations chat room.

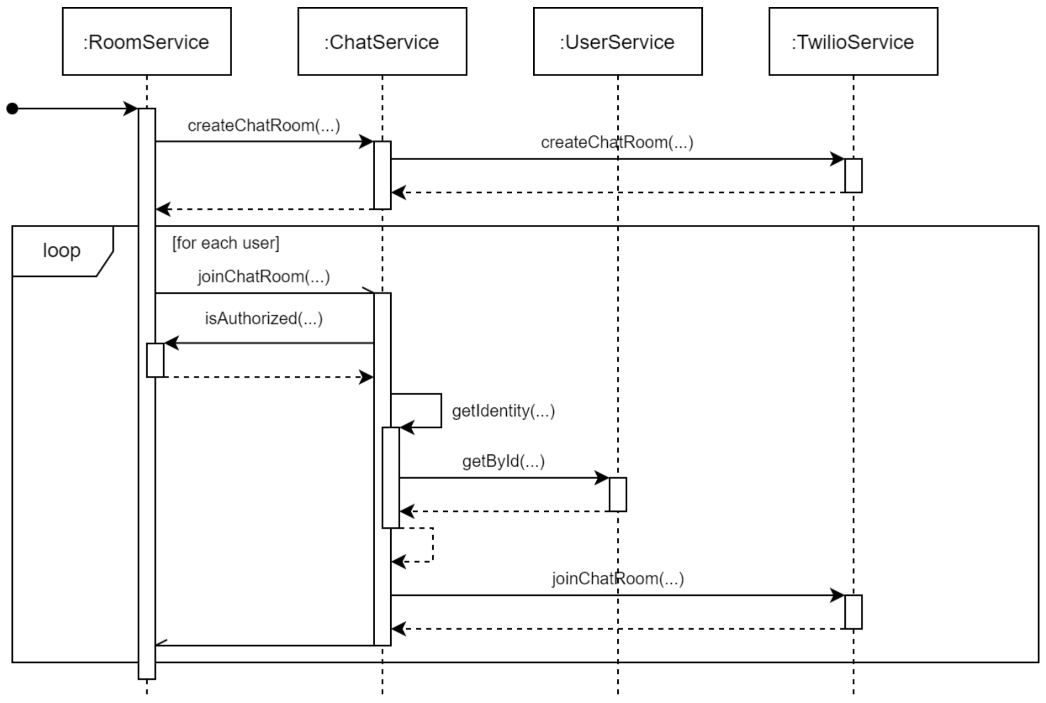


Figure 38: Sequence diagram of chat room creation process

During the creation of a room, a Twilio Conversations chat room is created using the room ID as a unique identifier. At the same time, a system message is sent to the chat room and will be displayed when users connect to the chat room. After creation of the chat room, the matched users are added to the chat room based on their identity, which in this case is the user's email address. During the entire process, the `ChatService` stores the string identifiers returned by Twilio in the Redis store so that they can be used to delete the resources once a room session ends.

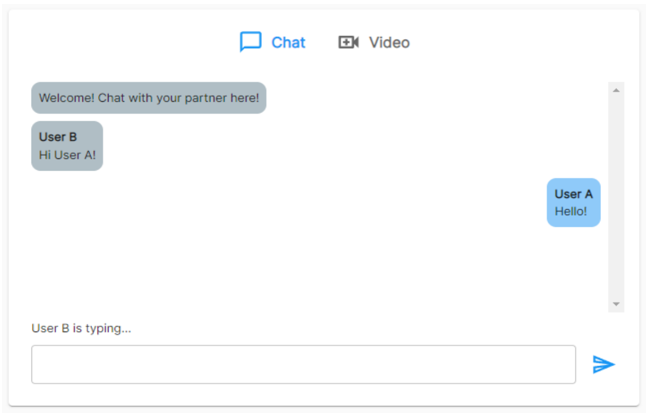
The creation and teardown of Twilio Video rooms are handled automatically by Twilio and thus do not need any setup to be performed when a room is created.

When the `ChatContext` is loaded on the frontend, it makes a `POST` request to the `ChatController::createToken` endpoint. This endpoint creates a Twilio JWT access token that contains both a chat grant for the Twilio Conversations chat room previously set up,

[22] See https://www.twilio.com/messaging/chat
[23] See https://www.twilio.com/video

and a video grant for a Twilio Video room with the room ID as its identifier. The access token is then returned to the frontend for use by the Twilio JavaScript clients.

On the frontend, the textual chat is handled by the @twilio/conversations[24] library which uses WebSockets to transmit messages between clients and Twilio servers [FR-Chat-1]. By using Twilio Conversations for textual chat, messages can be persisted even when users disconnect and useful features such as typing indicators can be integrated into the application with relative ease. Figure 39 shows the textual chat interface.



Welcome! Chat with your partner here!

User B
Hi User A!

User A
Hello!

User B is typing...

Figure 39: Screenshot of textual chat interface

Video chat functionality is handled by the twilio-video[25] library which has been configured to use peer-to-peer rooms over WebRTC for lower costs [FR-Chat-3 / FR-Chat-5]. Using Twilio Video also abstracts away the complexity of managing WebRTC connections manually for video and audio streams and has integrations which allow the user to disable video and audio streams temporarily [FR-Chat-4 / FR-Chat-6]. Figure 40 shows the user interface of the video chat feature with buttons to toggle the user's video and audio streams at the bottom left corner, and a button to mute the other user's audio stream in the bottom right corner.

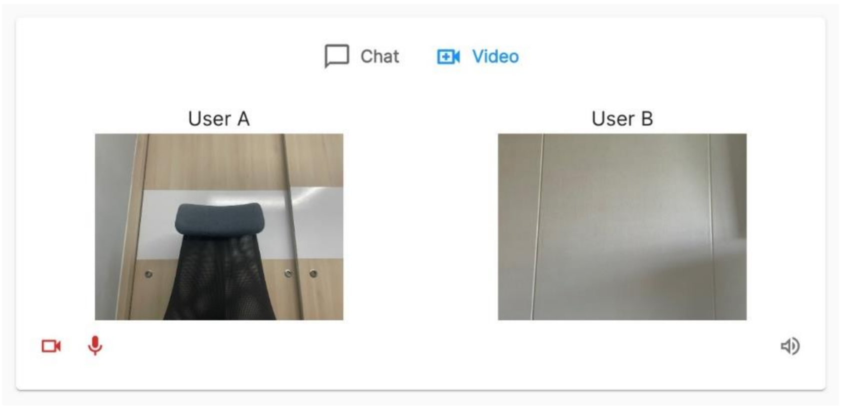


Figure 40: Screenshot of video chat user interface

[24] See https://www.npmjs.com/package/@twilio/conversations
[25] See https://www.npmjs.com/package/twilio-video

4.3.9. Code Submission

The code submission feature uses Judge0[26] for code execution. By relying on an external platform, the application does not need to execute untrusted code within its own servers. This prevents code injection vulnerabilities and potential remote code execution attacks.

Figure 41 gives a high-level overview of how the different components interact when a user makes a submission.

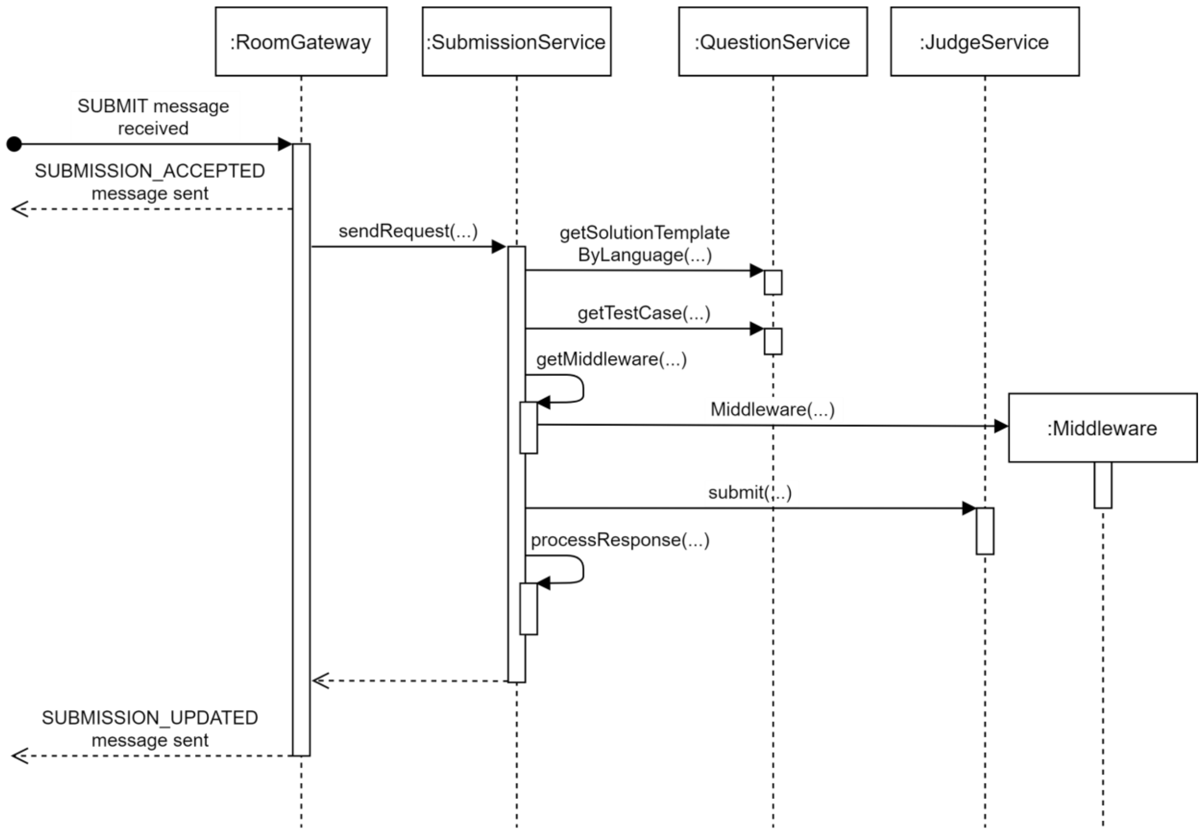


Figure 41: Sequence diagram of code submission process

When the `RoomGateway` receives a `SUBMIT` message, it acknowledges the submission by emitting a `SUBMISSION_ACCEPTED` event to all users in the room. The room ID will be stored in the Redis store and will be removed once the code execution is completed [FR-Judge-2]. The `SubmissionService` uses this key to check if there is an ongoing submission for the room to prevent users from making multiple submissions at once.

The submitted code is passed to the `SubmissionService` together with the question ID and language chosen. The question ID is used to retrieve the template code and the test case from the `QuestionService` [FR-Judge-1].

[26] See https://ce.judge0.com/

In order to execute the code on the Judge0 platform, an adapter is used to wrap the user's code with an entry point function and adds imports for common libraries. The adapter parses the template code to get the information it needs to construct the entry point. The class diagram of the adapters is shown in Figure 42 below.

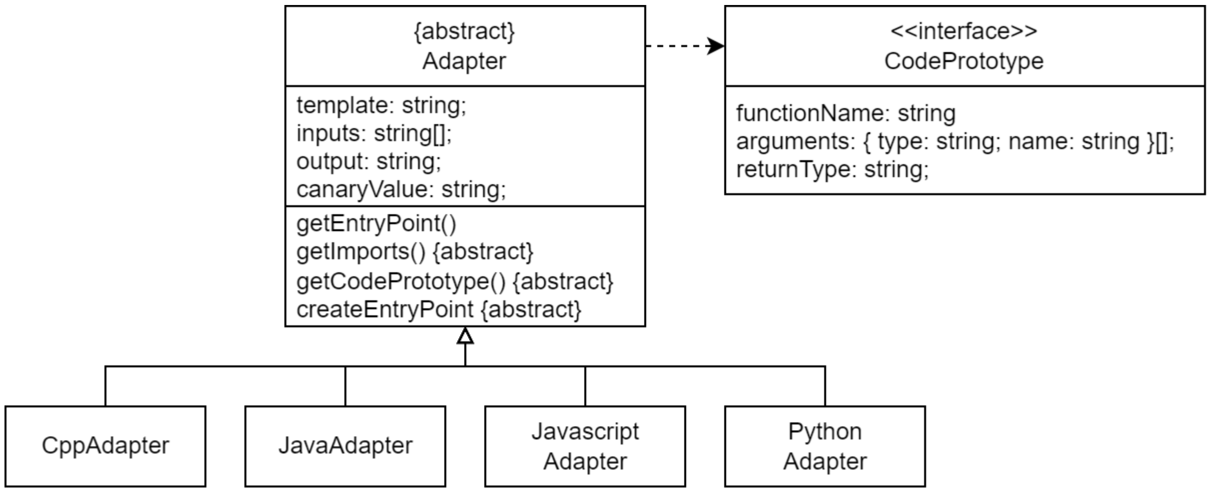


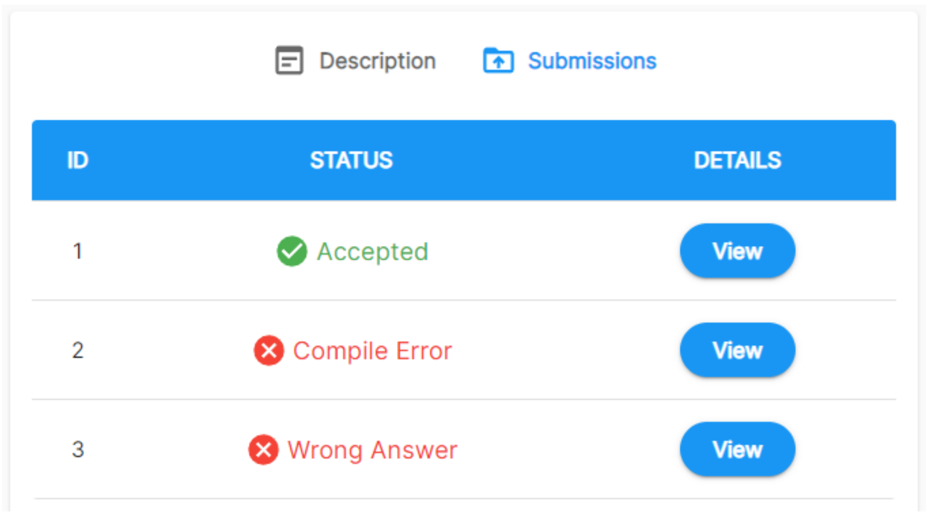
Figure 42: Class diagram of adapters used for code transformation

The adapter for each language extends from the abstract class `Adapter`. The concrete adapter is created using a factory method in the `SubmissionService`. This follows the Open-Close Principle and makes our code open to supporting new languages.

The result of the execution, success or failure, will be printed to the standard error output. This allows users to use debug messages to the standard output without interfering with result checking. To prevent users from abusing the system by printing the expected value to standard error, a secret value, often known as a canary, is added in front of the execution result and is verified before comparing the result.

The `SubmissionService` passes the modified code to the `JudgeService` which makes an API call to Judge0. The result of this call is a token that uniquely identifies the submission in Judge0 and this token is saved into the database with the unmodified code. A callback URL is provided in the request and Judge0 will make a `POST` request to this URL once the execution has completed. A callback endpoint was implemented rather than using polling to reduce computation and I/O load on the servers.

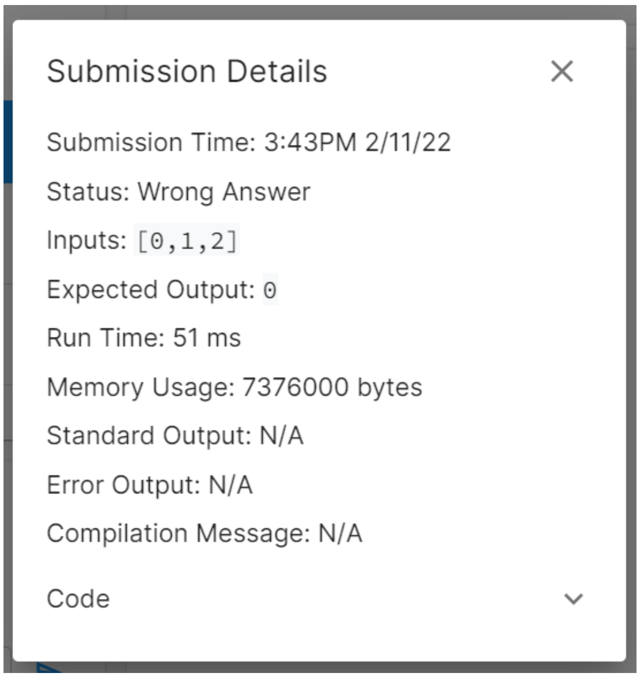
The callback response from Judge0 will be processed and the result, execution duration, memory usage, and any printed messages will be stored in the database by updating the record of the submission previously stored (omitted in the sequence diagram for brevity). The `RoomGateway` informs both users that the submission is completed by sending a `SUBMISSION_UPDATED` message. Upon receiving the `SUBMISSION_UPDATED` message, the React Query cache is invalidated, and an API call is automatically made to retrieve all submissions of the room. Figure 43 shows a screenshot of how the submissions of a room are displayed to users.



ID	STATUS	DETAILS
1	✓ Accepted	View
2	✗ Compile Error	View
3	✗ Wrong Answer	View

Figure 43: Screenshot of submissions table

Users are allowed to make multiple submission attempts and all the past results will be displayed in the submissions table [FR-Judge-5]. The details of the submission such as the test case input, expected output, and error messages, will be displayed in a dialog when users click on the "View" button [FR-Judge-3 / FR-Judge-4] (Figure 44).


Submission Details

Submission Time: 3:43PM 2/11/22

Status: Wrong Answer

Inputs: `[0,1,2]`

Expected Output: `0`

Run Time: 51 ms

Memory Usage: 7376000 bytes

Standard Output: N/A

Error Output: N/A

Compilation Message: N/A

Figure 44: Screenshot of dialog showing submission details

Handling of Race Conditions During Callback

As the callback from Judge0 occurs asynchronously, it is possible for the callback to arrive before the initial database entry was created with the token returned during the submission request. Figure 45 shows such an example, with event dependencies shown using boxes that share the same color, where the box in dashed outline requires the corresponding solid outlined box to complete before being able to take place. This results in an error as during the processing of the callback, an update request is made to the database to store the result of the submission.

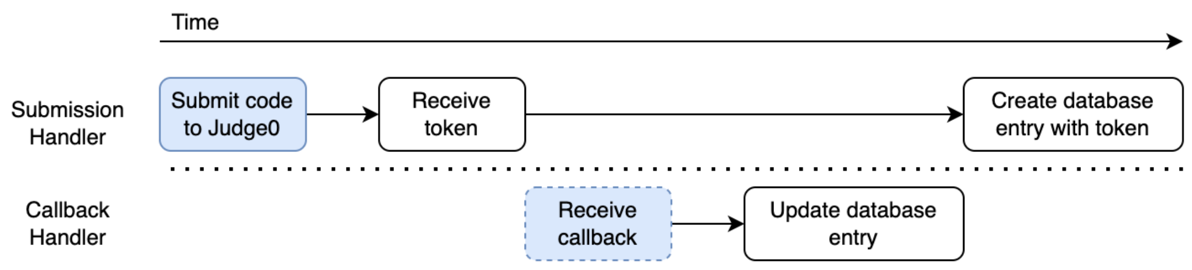


Figure 45: Timeline of events during race condition

To prevent the race condition, the following algorithm shown in Figure 46 was developed.

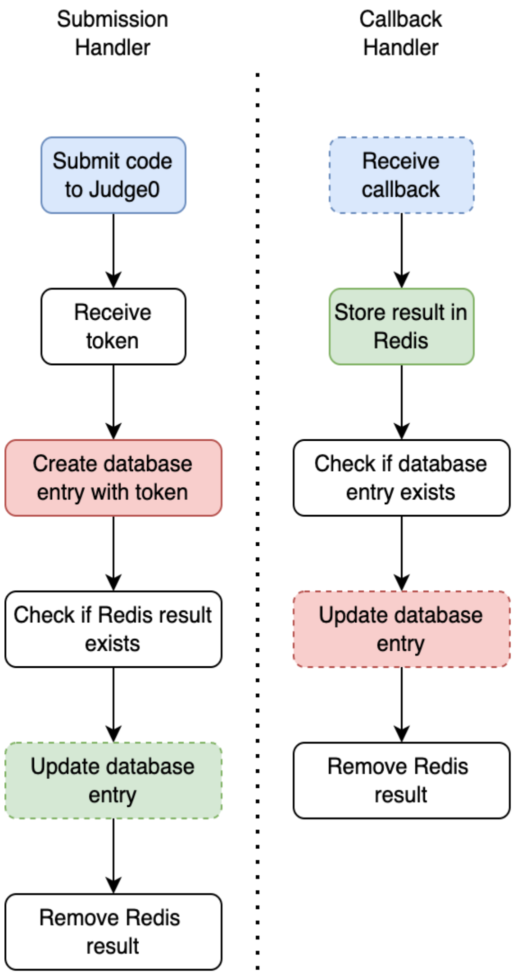


Figure 46: Sequence of events in submission callback algorithm

In the algorithm, if the check for the stored result in Redis fails for the submission handler or if the check for the database entry fails in the callback handler, the function returns immediately and does not continue.

This algorithm prevents the race condition as the database entry is checked by the callback handler before being updated. It also prevents lost updates by always storing the result in Redis before checking if the database entry exists. This guarantees that if the database entry check fails, there will still be a subsequent check for the stored result in Redis by the submission handler. In addition, since the database updates are idempotent, repeated updates can be tolerated while still retaining correctness of the algorithm.

4.3.10. Statistics

The statistics shown on the dashboard uses an open-source JavaScript visualization library called Apache ECharts[27]. Apache ECharts was chosen over another popular data visualization library, D3.js, as ECharts is a declarative framework unlike D3.js which uses low-level, data-driven documents. ECharts allows its users to rapidly construct web-based visualization by describing the chart in code while D3.js requires users to implement charts in code from scratch.

Although D3.js is a powerful visualization tool, there is significant technical overhead due to the ground-up chart implementation required. This may slow down the pace of CodeCollab's development and therefore ECharts was chosen over D3.js.

The metrics shortlisted for the dashboard, its associated chart representation, and rationalization are shown in the tables below (Tables 18, 19, 20, 21, 22). The metrics chosen encourages users to set goals for themselves and track their progress towards these goals.

[27] See https://echarts.apache.org/en/index.html

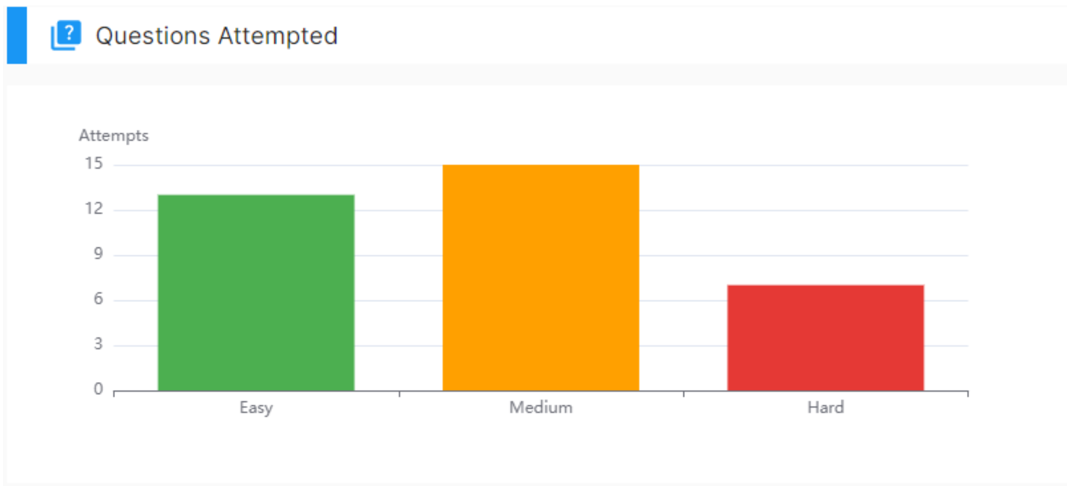
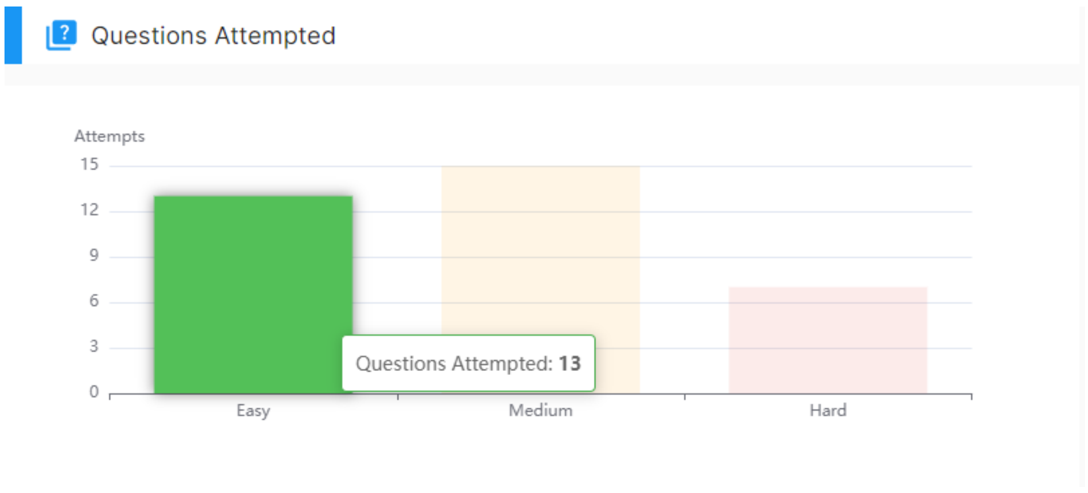
Metric Title: Questions Attempted [FR-Statistics-2]
Format: Vertical Bar
Example:  

Table 18: Information about "Questions Attempted" chart

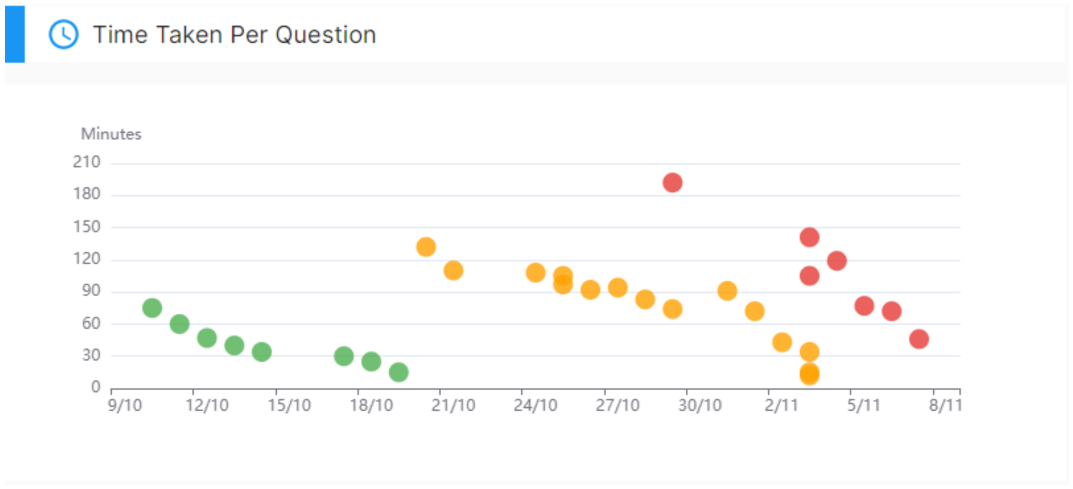
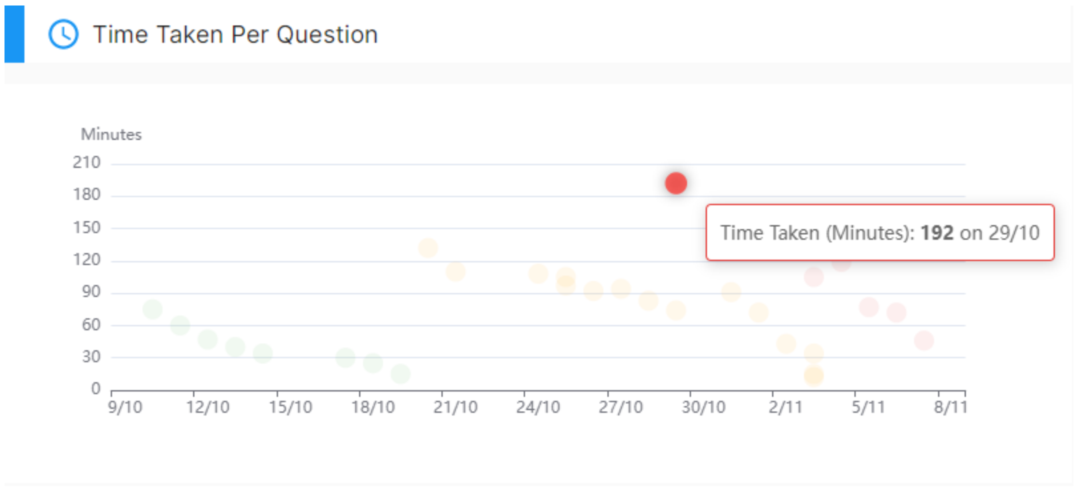
Metric Title: Time Taken Per Question [FR-Statistics-3]
Format: Scatter Plot
Example   Time Taken (Minutes): **192** on 29/10

Table 19: Information about "Time Taken Per Question" chart

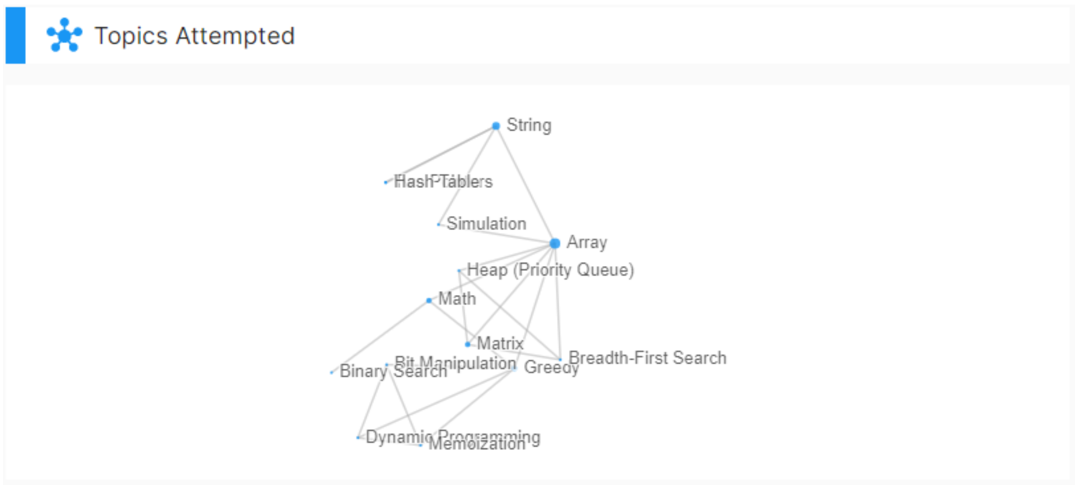
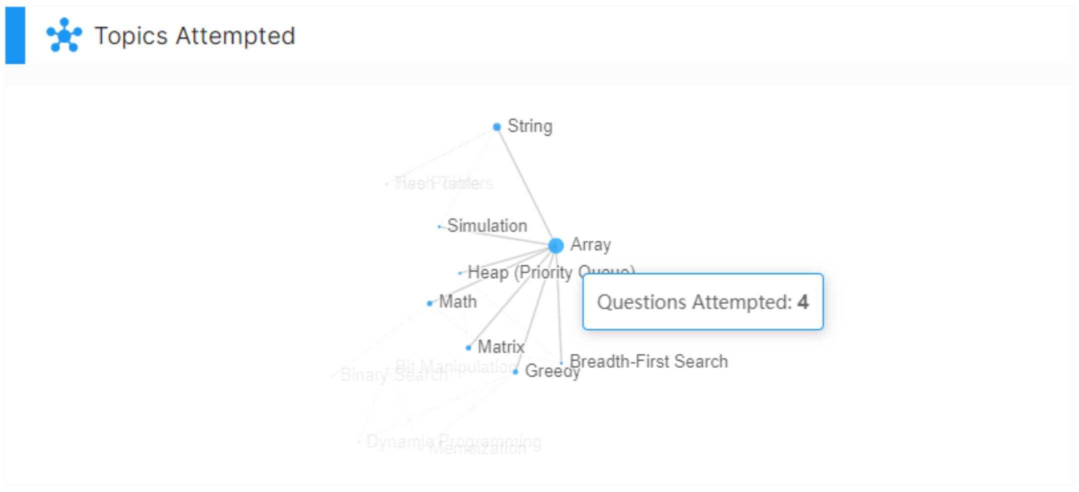
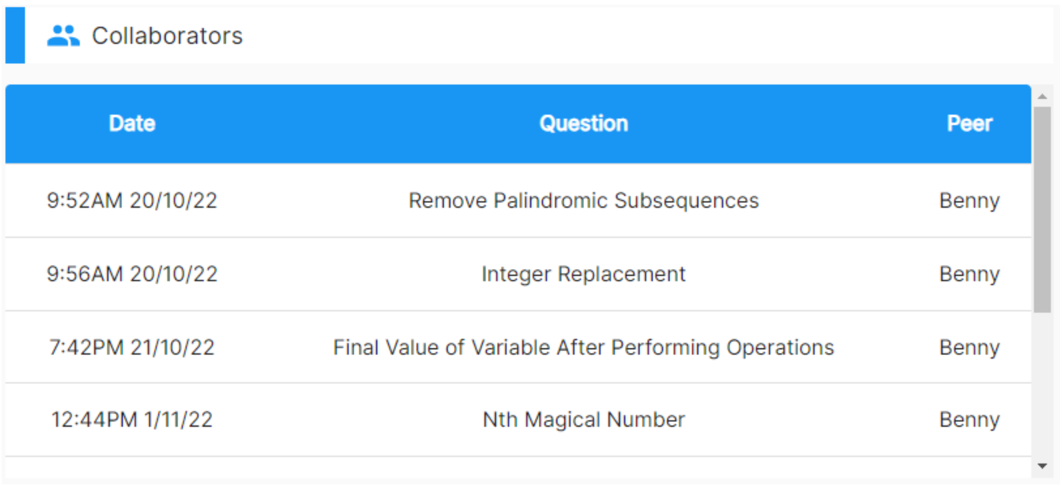
Metric: Topics Attempted [FR-Statistics-4]
Chart: Graph
Example  

Table 20: Information about "Topics Attempted" chart

Title: Collaborators [FR-Statistics-5]

Format: Table

Example



Collaborators

Date	Question	Peer
9:52AM 20/10/22	Remove Palindromic Subsequences	Benny
9:56AM 20/10/22	Integer Replacement	Benny
7:42PM 21/10/22	Final Value of Variable After Performing Operations	Benny
12:44PM 1/11/22	Nth Magical Number	Benny

Table 21: Information about "Collaborators" chart

Metric: Submissions In The Last Year [FR-Statistics-1]

Chart: HeatMap

Examples:

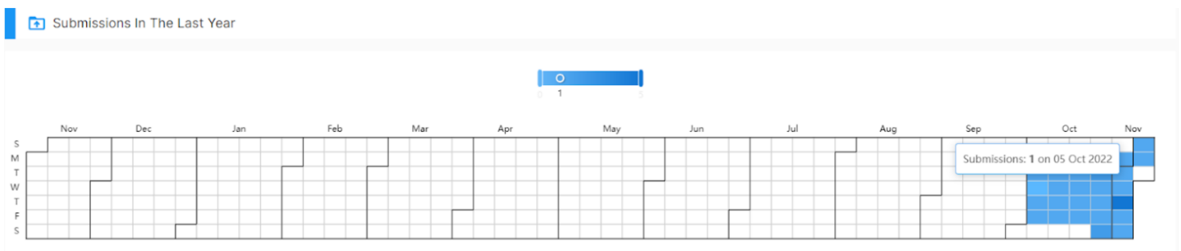


Table 22: Information about "Submissions In The Last Year" chart

To retrieve the data required for the dashboard, a single `GET` request is made to the `StatisticsController::getStatistics` endpoint (Table 23). This endpoint returns the data required by all charts in a single response, utilizing the DTO pattern to minimize disk and network I/O.

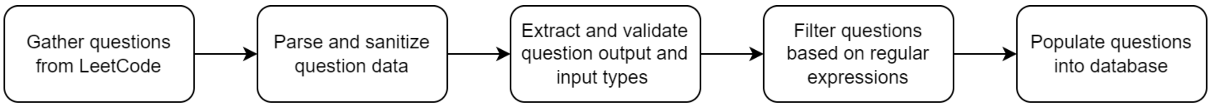
Request Endpoint: `GET api/user/statistics`

```
{
  attemptSummary: Record<Difficulty, number>;
  durationSummary: {
    difficulty: Difficulty;
    timetaken: number;
    date: Date | string;
  }[];
  peerSummary: {
    userName: string;
    questionTitle: string;
    date: Date | string;
  }[];
  heatmapData: {
    date: Date | string;
  }[];
  networkData: {
    topics: {
      id: number;
      name: string;
      count: number;
    }[];
    links: {
      smallTopicId: number;
      largeTopicId: number;
    }[];
  };
}
```

Table 23: Statistics endpoint and return type

4.3.11. Question Ingest (Serverless Component)

The application uses a scheduled serverless job to periodically update its question database from LeetCode[28]. Figure 47 shows the pipeline for gathering and filtering questions before they are ingested into the database.



Gather questions from LeetCode
Parse and sanitize question data
Extract and validate question output and input types
Filter questions based on regular expressions
Populate questions into database

Figure 47: Flowchart of question ingest pipeline

Due to constraints from the code submission feature, a subset of questions must be filtered out due to their semantics such as questions that require in-place modification of arrays, questions that have multiple possible answers, and questions that have custom classes defined. This filtering is performed through a mix of regular expression searches and parsing of expected return types from code templates.

To maintain a single source of truth for the database schema, the serverless component reuses the Prisma schema defined for the backend component. To achieve this, the Prisma package and folder was hoisted from its usual location within the backend folder to the root folder. This allowed a single Prisma schema to be defined and a single Prisma client to be generated and used in both the backend and serverless components, therefore helping to maintain type safety and database consistency.

[28] See https://leetcode.com/

4.4. Deployment Architecture

The frontend assets are served by the backend server using the serve-static package from NestJS. This allows the frontend and backend components to be built and packaged into a single docker image while the serverless component is contained in a separate docker image. Both containers are deployed using Google Cloud Platform (GCP). Figure 48 provides an overview of the GCP services used to host the application.

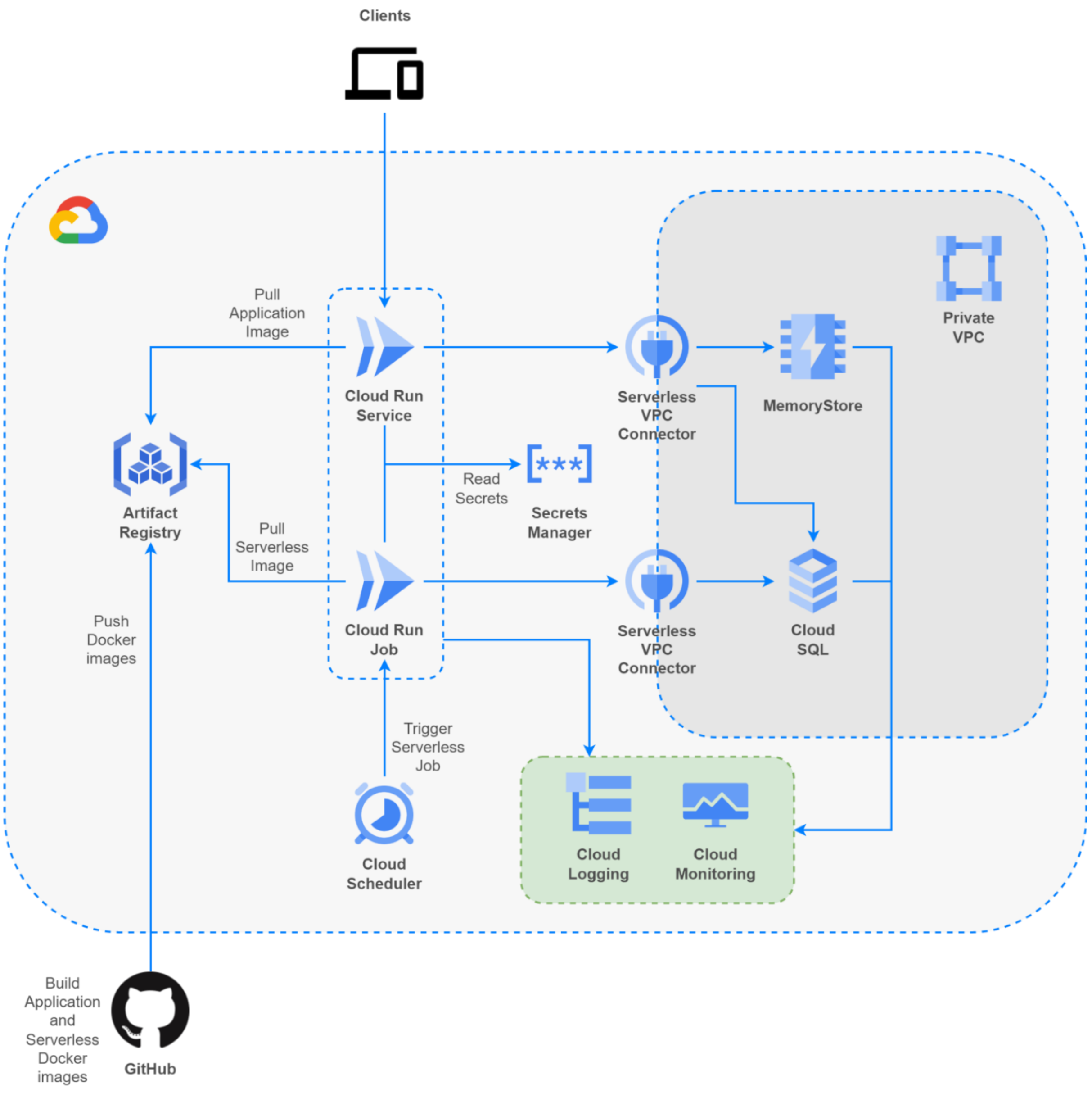


Figure 48: Deployment architecture

The application database is hosted on Cloud SQL within a private VPC and has external connections disabled to improve security by reducing the attack surface. A Redis instance is also hosted using MemoryStore within the same private VPC. The database and Redis instance are accessible from the Cloud Run instances using serverless VPC connectors.

The deployment process is performed using GitHub Actions which builds the application and serverless Docker images and publishes them to the Artifact Registry (see section 4.5). The application Docker image is loaded onto a Cloud Run Service while the serverless Docker image is loaded on to a Cloud Run Job.

The application is hosted on Cloud Run Service instead of Google Kubernetes Engine due to the lower amount of user configuration required and the automatic handling of load balancing, scaling, and certificate management. Cloud Run also offers a free tier and being serverless in nature, only bills for times when the application is serving users, therefore reducing overall cost.

The serverless component is executed on Cloud Run Jobs, currently still in beta, instead of Cloud Functions as it supports the use of Docker images which is necessary due to the use of the Prisma client in the serverless component as well. In addition, Cloud Run Jobs also support the execution of jobs which run for a longer duration, while Cloud Functions are limited to 15 minutes for event-driven jobs.

4.5. Continuous Integration/Continuous Delivery Workflow

GitHub Actions is used for the application's continuous integration/continuous delivery (CI/CD) workflows. There are three main workflows defined, one for CI, and another two for CD.

4.5.1. Continuous Integration

Figure 49 shows the overview of the CI workflow.

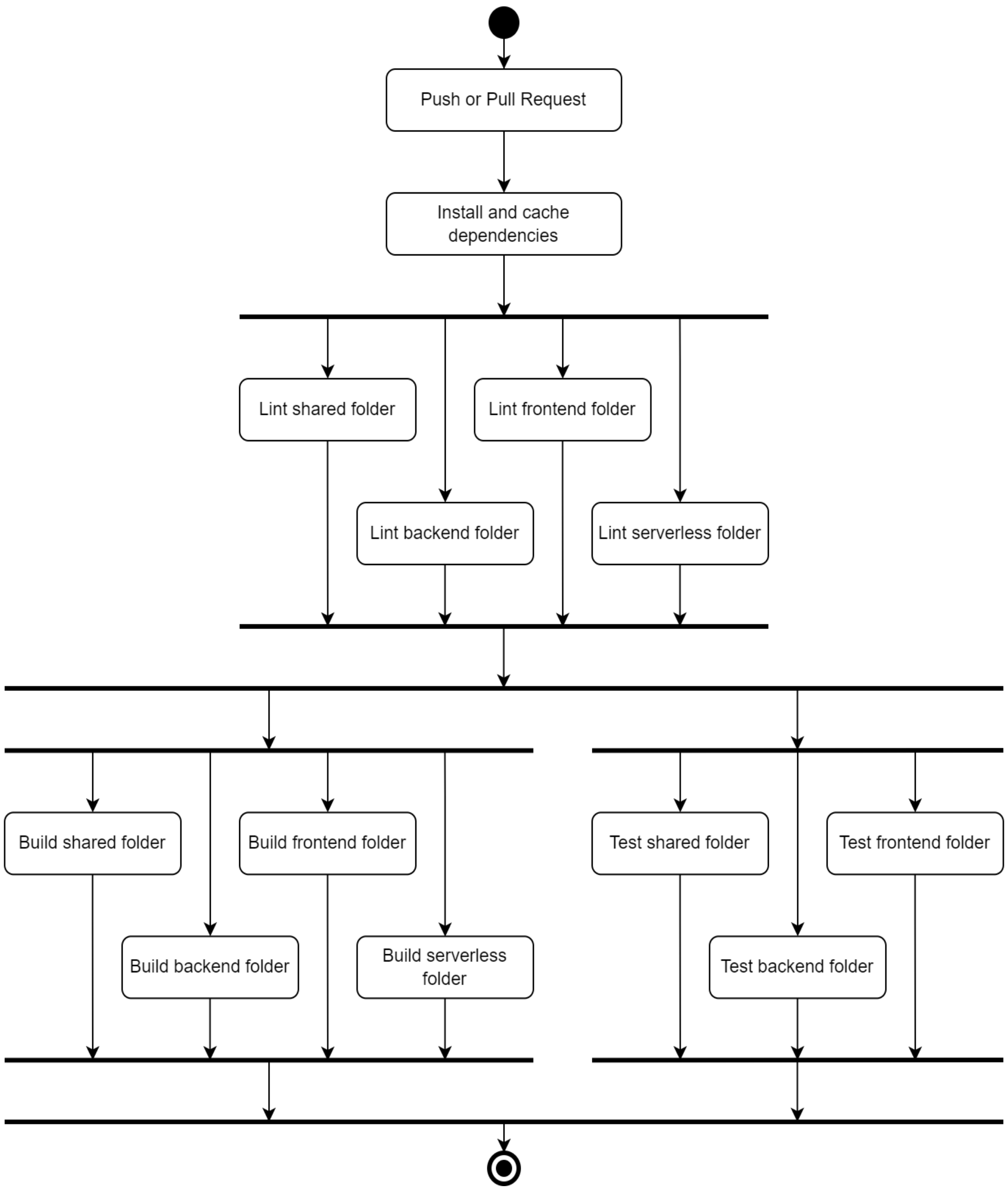


Figure 49: Overview of CI pipeline

The CI workflow is executed on every pull request and push to a branch and consists of four key steps, with the last three being executed using a job matrix for each of the four main subfolders of the repository:

1. Install: Package dependencies are installed and cached for future steps.
2. Lint: Code in the respective folders are checked for lint and stylistic errors.
3. Build: The respective components are built.
4. Test: Unit and integration tests are performed for each of the components, where applicable.

Apart from the CI workflow performed on GitHub Actions, pre-commit hooks are also implemented in the repository using husky[29]. The pre-commit phase lints and formats code locally, using ESLint[30] and Prettier[31], before committing them to ensure that the repository uses a consistent coding style.

4.5.2. Continuous Delivery

The CD workflow is triggered only on pushes to the staging or production branches and comprises two main workflow manifests, one for deploying the application and another for deploying the serverless component.

The staging branch, which is not protected, also serves as the testing branch for developers when deployment is required to test new features. This lack of separation of environments is mainly a cost-saving measure to reduce the amount of cloud resources needed. When a branch is pushed to override the staging branch, the Deploy to Staging workflow is triggered, which calls the Build and Deploy to Cloud Run workflow with the staging credentials. The Build and Deploy to Cloud Run workflow builds the application's Docker image, publishes it to Artifact Registry, and triggers the Cloud Run service to use the new application image. As staging may be frequently updated, the serverless component is deployed separately by a manual workflow trigger that invokes the Deploy Serverless to Staging workflow. This workflow calls the Build and Deploy to Cloud Run Jobs workflow with the staging credentials and works in a similar fashion to the Build and Deploy to Cloud Run workflow.

The production branch is protected and requires a pull request to be made before changes can be made to the branch. When a new release is pushed to the production branch, the Deploy to Production workflow is triggered. This workflow calls both the Build and Deploy to Cloud Run and Build and Deploy to Cloud Run Jobs workflows with production credentials.

[29] See https://www.npmjs.com/package/husky
[30] See https://www.npmjs.com/package/eslint
[31] See https://www.npmjs.com/package/prettier

5. Improvements and Enhancement Suggestions

Due to time constraints in development, the team was unable to implement certain features that can add value to CodeCollab's functionality. These enhancements will be elaborated on in this section.

5.1. Topic Selection in Selection Page

Apart from Difficulty and Programming Language Selection on the Selection Page (Figure 24), some users may wish to practice questions on topics that they are weaker in. A Topic selection functionality will be beneficial for this group of users.

5.2. Friends List

A friends list feature will be useful to users that prefer to work with users that they have previously worked with before. This feature will empower end-users to choose who they wish to collaborate with.

5.3. Profile Personalization

Currently, users are only able to update their display name and password (Figures 19 & 20). Some users may wish for more profile personalization to express their individuality. The feature also serves the functional purpose of allowing the users to be more identifiable to other users. Thus, the settings page can be improved by allowing for profile picture uploads and updating of personal biography.

6. Reflections and Learning Points

Throughout the development process, the team gained a better understanding of the event-driven architecture pattern and asynchronous communication. WebSockets are used heavily in our application to build event-driven APIs and enable bidirectional communication between server and client. The asynchronous nature of WebSockets helps to improve the responsiveness of our application. However, it also meant that more care was needed when handling them to prevent race conditions.

Another technical learning point for the team was the difficulty in achieving scalability. While the team was conscious to not store state locally on any server, the very nature of how WebSockets work mean that they are stateful. Additionally, libraries and packages written by others may not be implemented in a stateless manner as they may not have been built with horizontal scalability in mind. Furthermore, emulating a distributed environment on a local development environment requires additional work while the standard development environment would prevent any bugs relating to horizontal scalability from arising.

Although the team gained a better understanding of pattern applications and the attainment of scalability, it was ineffective to communicate in terms of abstractions as the project was engineered on a small scale with four members per team. Moreover, as the team was composed of undergraduates, there was a lack of advanced subject domain knowledge to communicate effectively in abstractions.

On the side of project management, Scrumban was suitable and effective due to the demographics of the team. Having the liberty to decide which tasks best suit one’s expertise and the amount of work that one can handle for the week made it possible to work around the team members’ busy and varying schedules as undergraduates.