School of Computing

CS3219 Software Engineering Principles and Patterns

AY22/23 Semester 1

Project Report

Group 51

Team Members	Student No.	Email
Han Zong Yu	A0149931X	e0014835@u.nus.edu
Ryan Chang Jia Jie	A0218314M	e0544350@u.nus.edu
Sim Jia Ming	A0218083H	e0544119@u.nus.edu
Tan Yu Qi	A0217509A	e0543545@u.nus.edu

Table of Contents

Background and purpose of the project

When finding internships or full-time roles, students often face challenging technical interviews. Some problems they face include explaining their solution and their inability to solve the given problem. Furthermore, practicing questions continuously can be repetitive and tiresome.

A general description of our application:

A student who is keen to prepare for his technical interviews visits the site. He creates an account and then logs in. After logging in, the student selects the question difficulty level he wants to attempt today (easy, medium, or hard). The student then waits until he is matched with another online student who has selected the same difficulty level as him. If he is not successfully matched after 30 seconds, he times out. If he is successfully matched, the student is provided with the question and a free text field in which he is expected to type his solution. This free text field should be updated in near-real time, allowing both the student and his matched peer to collaborate on the provided question. After the students finish working on the question and are ready to end the session, any of them clicks on a "Finish" button which returns each student to the question difficulty selection page. From this page, the student logs out.

The primary motivation is to build a web application that could aid students in preparing for technical interviews. We aspired that students could engage in a peer learning system that could enable them to practice coding together and learn from one another. This not only reduces the monotony of practicing but also accelerates the learning of the students.

Individual Contributions

Name	Technical contributions	Non-technical contributions
Han Zong Yu	1) Implemented chat service 2) Implemented real-time collaboration editor 3) Debug and refactor both frontend and backend codes 4) Implemented backend history services 5) Help out in designing the User Interface	1) Documentation 2) Organize Meetings 3) Set soft deadlines
Ryan Chang Jia Jie	1) Implement authentication for user service. 2) Implement create, update and delete operations in user service. 3) Assist in debugging.	1) Documentation 2) Consolidate project information for easy reference and planning 3) Format JIRA project tickets
Sim Jia Ming	1) Implemented overall frontend for login, signup, history, profile, and editor page. 2) Set up MongoDB atlas and connect to codebase 3) Help out in debugging.	1) Documentation 2) Organize meetings 3) Set soft deadlines 4) Set up JIRA project management tool
Tan Yu Qi	1) Implemented the first iteration of user service (initial login system). 2) Implemented the real-time collaboration editor feature 3) Refactored the editing password feature to integrate with Firebase. 4) Helped out in debugging of features.	1) Documentation 2) Ensure timelines are kept in check 3) Update JIRA project management tool 4) Coordinate meetings with tutors for milestone check-in.

Requirements

Functional Requirements

Category	S/N	Functional Requirement	Priority
User Service	1	The system should allow users to create an account with a username and password.	High
	2	The system should ensure that every account created has a unique username.	High
	3	The system should allow users to log into their accounts by entering their username and password.	High
	4	The system should allow users to log out of their account.	High
	5	The system should allow users to delete their account.	High
	6	The users' passwords should be hashed and salted.	High
	7	The logging in of the user should make use of a JWT token for authentication.	Low
	8	The logging in of the user should make use of a JWT token for authentication.	Low
Matching Service	9	The system should allow users to select the difficulty level of the questions they wish to attempt	High
	10	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High
	11	If there is a valid match, the system should match the users within 30s.	High
	12	The system should inform the users that no match is available if a High No match cannot be found within 30 seconds.	High
	13	The system should provide a means for the user to leave a room Medium No once matched.	Medium
Question Service	14	The system should be able to store questions in a question bank by difficulty level.	High
	15	The system should be able to allow the user to change questions.	Medium
	16	The system should be able to retrieve questions from the difficulty level matched and show them to both users.	High

Collaboration Service	17	The system should provide users with a means to code collaboratively through an editor.	High
	18	The system should allow users to run the code to see the output of the code that both users have written.	Medium
	19	The system should allow the users to code in the programming language that they prefer.	Medium
	20	The application should be able to allow the user to upload / download codes.	Low
	21	The system should provide users with a means to communicate with each other through chat.	Medium
Frontend	22	The system should be able to have some basic UI for users to interact with.	High
	23	The system should have an appealing UI with the intention of making it easy for users to navigate the application.	Medium
History Service	24	The system should be able to display the past questions attempted for each user.	Medium
	25	The system should be able to save the details of a Peerprep session.	Medium

Non-Functional Requirements

1. Performance			
S/N	Non-Functional Requirements	Priority	Achieved
1.1	The system should react to user inputs in less than a second.	Medium	✓
1.2	In less than 1 second, the system should authenticate users.	Medium	✓
1.3	The system should update the conversation in real-time in less than one second.	Medium	✓
1.4	In less than one second, the system should update the text editor.	Medium	✓
1.5	The system should move between pages (for instance, from the login screen to the home screen) in under 2 seconds.	Medium	✓

1.6	The system should finish making API requests to retrieve data in less than 2 seconds.	Medium	✗

2. Security			
S/N	Non-Functional Requirements	Priority	Achieved
2.1	A user should be prevented from trying indefinitely after several unsuccessful login attempts.	Medium	✓
2.2	The system should make sure that only the two authenticated users involved in a session can join it.	High	✓

3. Usability			
S/N	Non-Functional Requirements	Priority	Achieved
3.1	Before telling the user that it cannot find a match, the system should keep the user in the queue for no more than 30 seconds.	Medium	✓
3.2	The system should utilize a font color that the user can read readily.	High	✓
3.3	System should render on screen width starting depending on the user's screen size and should be dynamic.	Medium	✓
3.4	System should use a readable font, namely the sans-serif and Arial font.	High	✓
3.5	By utilizing the same design system, the user interface of the system should appear cohesive and have a consistent usage of colors and fonts.	Low	✓
3.6	When the user does an irreversible action, the system should prompt them for confirmation (e.g deleting an account or changing a password)	High	✓

4. Portability			
S/N	Non-Functional Requirements	Priority	Achieved
4.1	System should work on Safari Version 16 and above.	High	✓
4.2	System should work on Google Chrome Version 106 and above.	High	✓

Prioritization of requirements

After looking at the requirements table, we prioritized requirements based on mainly 2 criterions. The first criterion is whether the requirements contributed to meeting must-have features or nice-to-have features. The second criterion is to think from the user's perspective and decide if the requirements are mission-critical to the user's experience on the app.

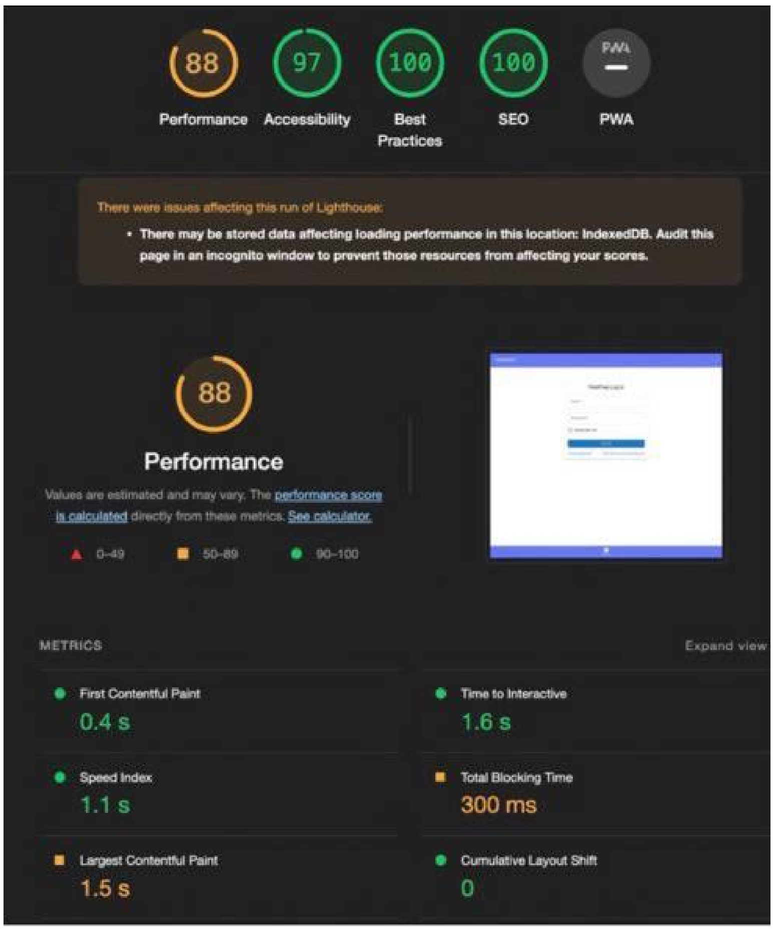
We mainly prioritized authentication, and Create-Read-Update-Delete(CRUD) functionalities because it was the most practical to the user in our perspective as well as mostly coincides with the must-have features.

In terms of prioritization of microservices, we prioritized the user and matching services first because it was core to making the minimum viable product (MVP). The front end was a lower priority because it was imperative to get everything working before beautifying and optimizing the application.

Throughout the implementation, we also kept in mind the Non-Functional Requirements (NFRs) so that we could ensure meeting them at the end of each milestone.

Meeting the NFRs

We made use of Google Chrome's inbuilt auditing tool, Lighthouse, to measure the performance of our application. We managed to achieve the first contentful paint (FCP) of 0.4s. FCP is a performance metric that measures how quickly visitors can view actual content on our page. It was made possible by our intuitive and clean frontend design, allowing it to load quickly. In addition, our Time to Interactive (time for the page to be interactive after FCP) is 1.6s. The speed index measures how quickly the content of our page is loaded and visible to the user and we managed to achieve a speed of 1.1s, hence meeting our NFR of loading a page in 2s.



88 Performance
97 Accessibility
100 Best Practices
100 SEO
PWA

There were issues affecting this run of Lighthouse:

- There may be stored data affecting loading performance in this location: IndexedDB. Audit this page in an incognito window to prevent those resources from affecting your scores.

88

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

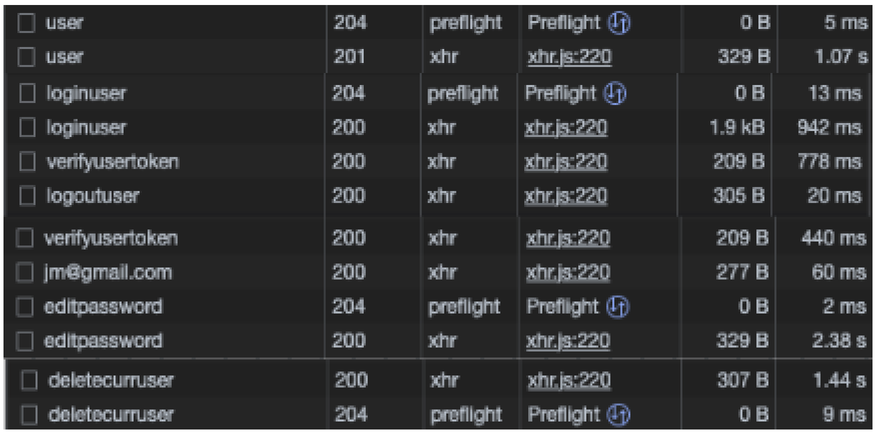
▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● First Contentful Paint 0.4 s	● Time to Interactive 1.6 s
● Speed Index 1.1 s	■ Total Blocking Time 300 ms
■ Largest Contentful Paint 1.5 s	● Cumulative Layout Shift 0

Figure M1: Lighthouse report for Peerprep Application.

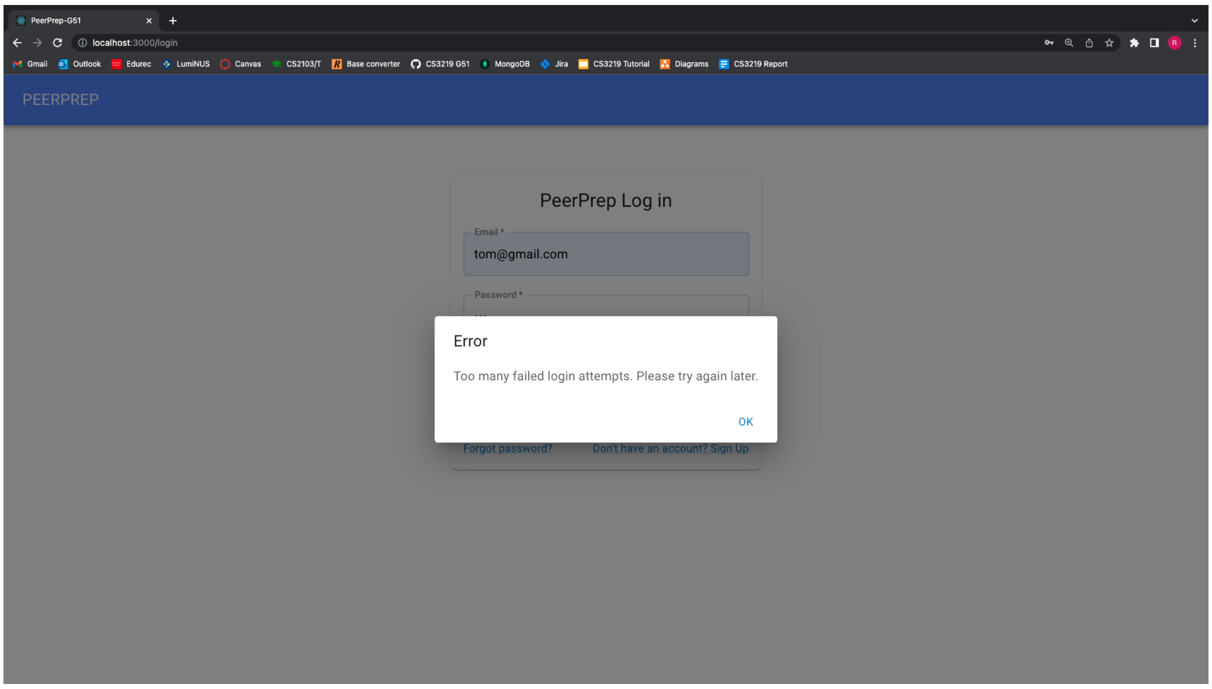
We have also tested all of our api calls and all of them are able to respond within 2s except for edit password. We were unable to achieve this NFR due to the checks that we have in place to ensure the multiple required password fields were entered correctly before allowing the user to edit the password. Hence, the inevitable delay in the response time. However, we feel that this should not affect the user experience of the whole application.



user	204	preflight	Preflight	0 B	5 ms
user	201	xhr	xhr.js:220	329 B	1.07 s
loginuser	204	preflight	Preflight	0 B	13 ms
loginuser	200	xhr	xhr.js:220	1.9 kB	942 ms
verifyusertoken	200	xhr	xhr.js:220	209 B	778 ms
logoutuser	200	xhr	xhr.js:220	305 B	20 ms
verifyusertoken	200	xhr	xhr.js:220	209 B	440 ms
jm@gmail.com	200	xhr	xhr.js:220	277 B	60 ms
editpassword	204	preflight	Preflight	0 B	2 ms
editpassword	200	xhr	xhr.js:220	329 B	2.38 s
deletecurruser	200	xhr	xhr.js:220	307 B	1.44 s
deletecurruser	204	preflight	Preflight	0 B	9 ms

Figure M2: Network speed report for Peerprep Application.

We also handled users that try to log in with un-authenticated credentials indefinitely by using Firebase as it does not allow users from logging in after more than 5 failed attempts.

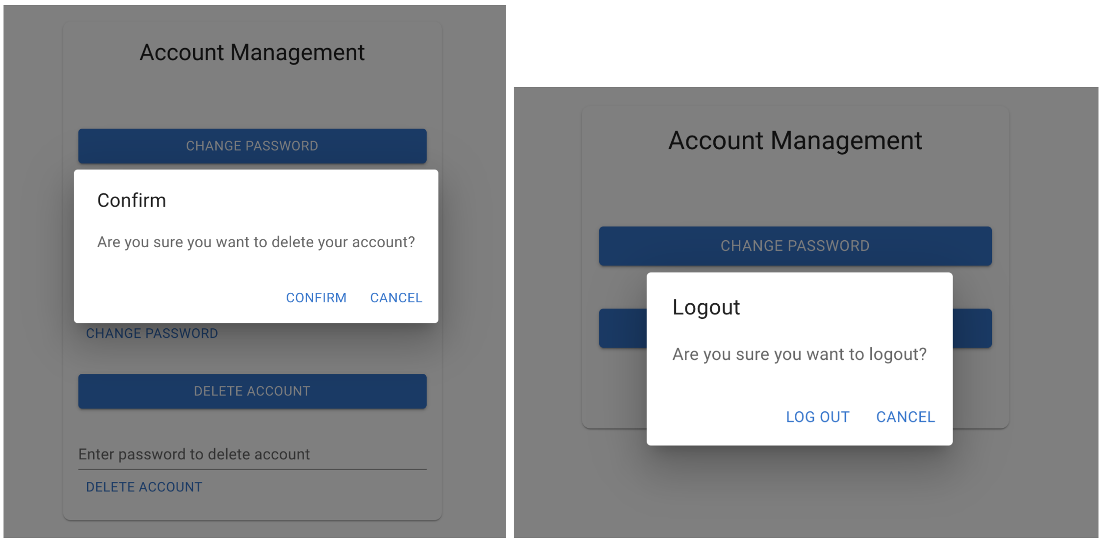


Error

Too many failed login attempts. Please try again later.

Figure M3: Login error message.

Based on user feedback from our peers, they mentioned that if we were to change a password without re-entering the new password again, it might be susceptible to typo errors and the user might mistakenly enter an unintended password. Hence we added a safety net of typing the new password twice in order to change the password. In addition, the delete account button could permanently delete an account without asking for confirmation initially. Hence, we implemented a field for the user to enter their password as a confirmation step to delete their account. On top of these measures, we implemented a confirmation prompt for critical actions like deleting an account and logging out as shown in Figure M4 and M5.



Confirm

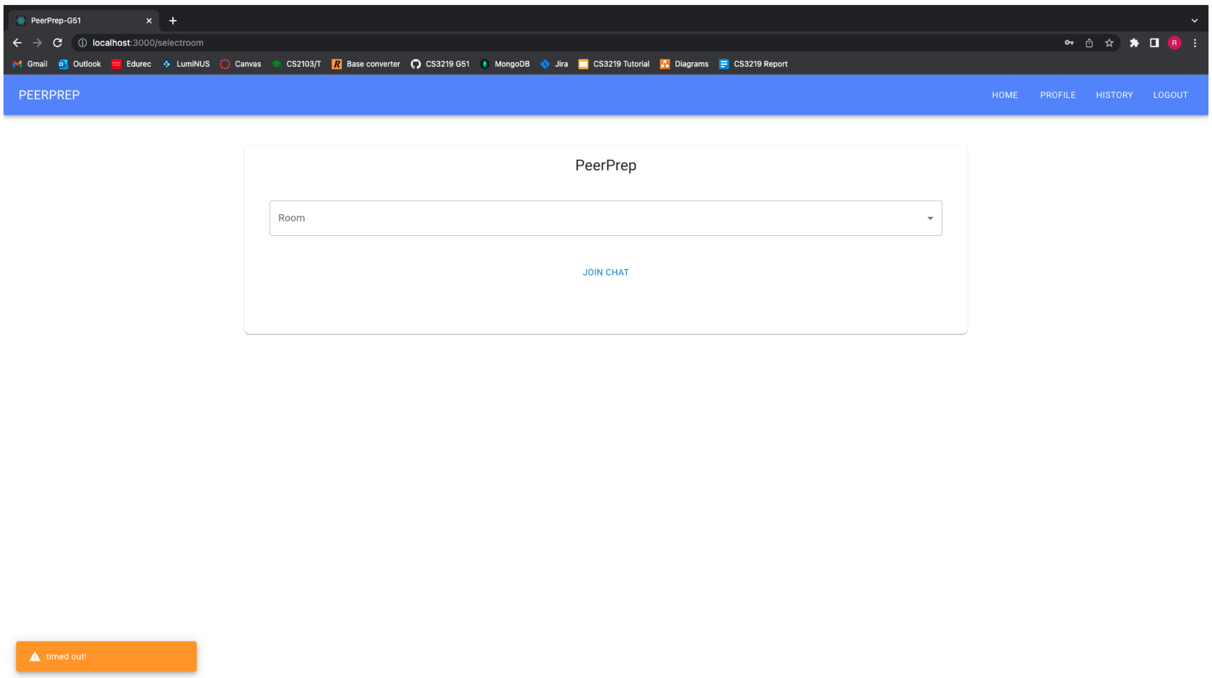
Are you sure you want to delete your account?

Logout

Are you sure you want to logout?

Figure M4. Delete account confirmation & Figure M5. Logout confirmation

The system also removes the user from the queue after 30 seconds if there are no other users to match with.



timed out!

Figure M6: Timeout notification after failing to match the user.

Development Process

The development process of the project broadly followed the waterfall model in terms of the software development life cycle. Everyone on the team gathered together to discuss and clearly define the functional and nonfunctional requirements that we chose to pursue. The rough architecture diagram was also drawn out and shown to our mentor for approval. After which, we started milestone 1 for development.

Timeline

The project was generally split into 3 milestones, with milestone 1 having 2 weeks, milestone 2 having 4 weeks, and milestone 3 having 4 weeks. Each sprint in a milestone is 1 week long. The development and documentation were aimed to be completed by the end of week 11. This gave us one week of buffer to tie up loose ends and make sure everything is ready before the presentation. The development plan can be found in the timeline under Appendix**.**

Project Management

As a team, we made sure to set aside time every Wednesday night from 9pm to 10pm for weekly standup meetings to update each other on our progress and voice out if any issues were faced so that we could debug together. We also made sure to rotate the role of project manager for the entire project throughout the milestones. Every member manages the Jira tickets for their own tasks.

For effective project management, we made use of Jira to keep track of tickets needed to be completed in each milestone. Furthermore, we employed the use of the GitHub Issue tracker during milestone 1 because there were more requirements that we needed to complete. However, for subsequent milestones, we realized that it was not necessary as Jira has already served the purpose of tracking the requirements we needed to complete.

Tech Stack

MERN Stack

We have decided to use the MERN stack: MongoDB, Express.js, React.js, and Node.js for our project. This collection of technologies enables faster application development as it allows the development of our application using JavaScript only. This is because the four technologies that make up the technology stack are all JS-based. Thus, if one knows JavaScript (and JSON), the backend, frontend, and database can be operated easily.

MERN Stack Components

There are four components of the MERN stack.

- The first component is MongoDB, which is a NoSQL database management system.

- The second MERN stack component is ExpressJS. It is a backend web application framework for NodeJS.
- The third component is ReactJS, a JavaScript library for developing UIs based on UI components.
- The final component of the MERN stack is NodeJS. It is a JS runtime environment, i.e., it enables running JavaScript code outside the browser.

Benefits of MERN

One major benefit is that it allows the creation of a 3-tier architecture that includes frontend, backend, and database using JavaScript and JSON.

MongoDB, which is the base of the MERN stack, is designed to store JSON data natively. Everything in it, including CLI and query language, is built using JSON and JS. The NoSQL database management system works well with NodeJS and thus, allows manipulating, representing, and storing JSON data at every tier of the application. Furthermore, it comes in a variant called MongoDB Atlas that further eases database management by offering an auto-scaling MongoDB cluster on any cloud provider with just a few clicks.

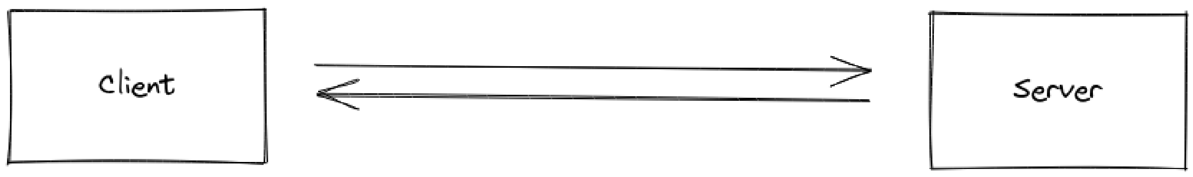
Express is a server-side framework that wraps HTTP requests and responses and makes mapping URLs to server-side functions easy. This perfectly complements the ReactJS framework, a front-end JS framework for developing interactive UIs in HTML while communicating with the server.

As the two technologies work with JSON, data flows seamlessly, making it possible to develop fast and debug easily. To make sense of the entire system, you need to understand only one language, i.e JavaScript and the JSON document structure.

Socket IO

We have decided to use Socket IO for our collaborative features such as chat service and real-time collaborative editor.

Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server.



It is built on top of the WebSocket protocol and provides additional guarantees like fallback to HTTP long-polling or automatic reconnection.

Benefits of Socket IO

The connection will fall back to HTTP long-polling in case the WebSocket connection cannot be established.

This feature was the #1 reason people used Socket.IO when the project was created more than ten years ago (!), as the browser support for WebSockets was still in its infancy.

Even if most browsers now support WebSockets, it is still a great feature as we still receive reports from users that cannot establish a WebSocket connection because they are behind some misconfigured proxy.

Firebase

Firebase was used for our user service authentication. It offers ready-made UI libraries, simple SDKs, and backend services. It also supports federated identity providers like Google, Facebook, and Twitter, as well as passwords, phone numbers, and other methods.

Benefits of Firebase

It helps us avoid having to design Web Service authentication mechanisms from scratch. After the user has authenticated with Firebase, we can simply have a function to save user information. Furthermore, if we decide to include social logins like Facebook and Google, we may avoid creating server-side methods for various forms of token verification. With Firebase, all of that can be handled effectively.

Overall Architecture

Architecture

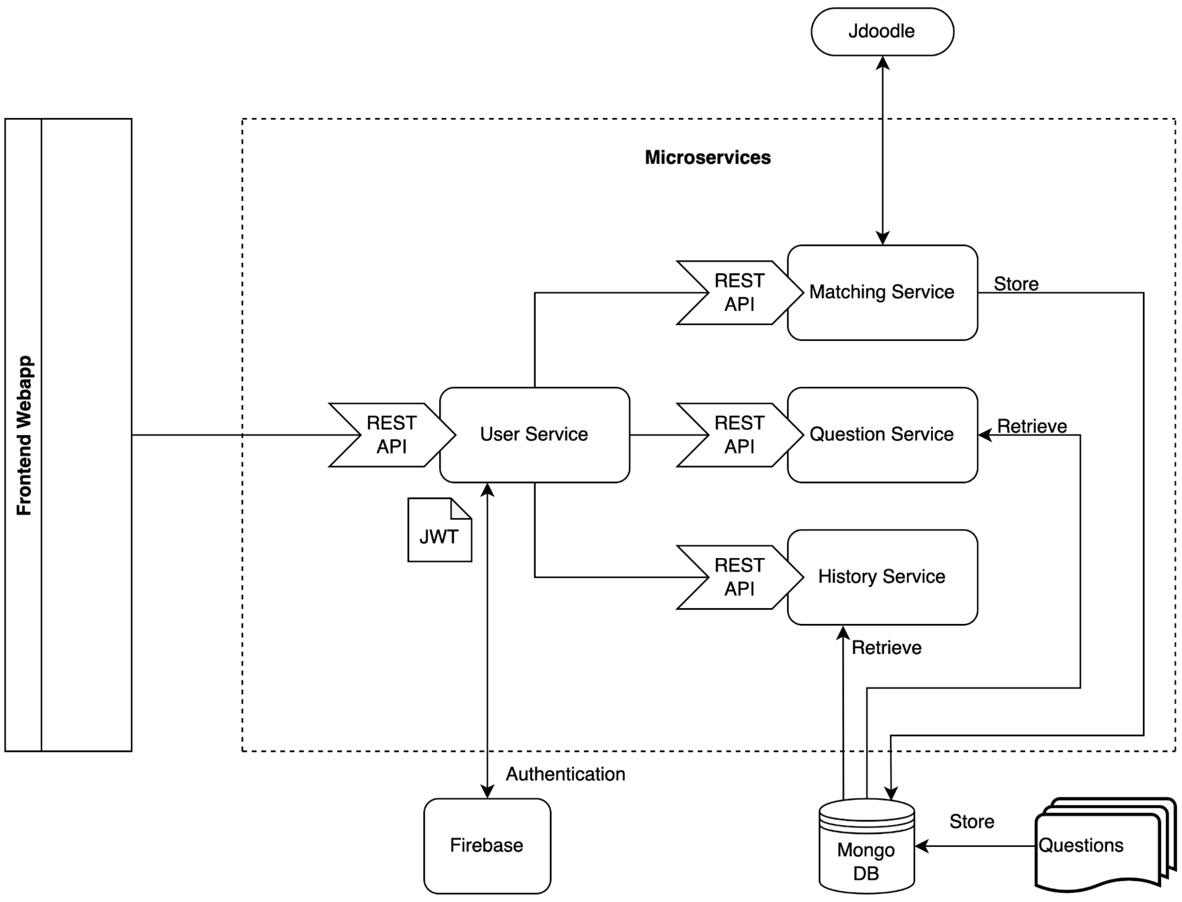


Fig O1: Overall Architecture diagram

For the architecture of the project, we decided to adopt a microservices architecture pattern because of several reasons. Firstly, it allowed us to segregate the different features into services that the users would use. This made code for each service easier to be handled and maintained. Furthermore, it meant that services were decoupled from each other. Secondly, the microservices made it easier to extend and create new services for users to use, which means that improvements can be made in the future much more easily. Lastly, the microservices gave us more options in terms of deployment, where we could deploy each service independently or perhaps deploy it through other means such as docker.

Design patterns and principles

Singleton Pattern (React.js)

For the front end, we employed the use of a singleton pattern. For instance, we kept a centralized state for several key states, such as user, room, etc. To illustrate specifically, we kept a central state for the user object so that we can pass the user object through the different components in react. This ensures that during implementation, we can break down the components and identify which state is needed. This also means that the state can be preserved throughout the app. However, one major con is that the state has to be referenced throughout the app, and this could result in prop drilling. One solution to this is that we could have used Redux to prevent prop drilling and ensure a cleaner and more structured codebase. However, given the scale of the project is small and that implementing features was of higher priority, we did not use Redux in our project.

Mediator Pattern

There were several use cases of the mediator pattern in our project. Firstly, we used the mediator pattern in assigning the room to the users. We used socket.io to manage the requests and socket.io acted as an interface in which the users could connect with each other. Another case would be using socket.io for the chat. This allowed us to manage all the interactions between two users, and let us focus on interacting with the socket while allowing the socket to ensure messages are sent through.

Single Responsibility Principle

Every single service should have a responsibility in the application, hence, they should only have one reason to change. By ensuring that every service achieves the single responsibility principle, it was easier to implement each feature and there was less coupling between services. Hence, this allowed different members of the team to work on different services at the same time as there are little to no dependencies. This reduces the number of bugs and development time.

Open-Closed Principle

Our microservice architecture allows our application to be opened for extension but closed to modification. This makes it easier for us to implement new services like the question service and history service as they are not dependent on other services. This allowed us to work in parallel during development as there was minimal coupling between services. Furthermore, this was very useful in our waterfall methodology, during the development phase where all our resources were put into pushing out the features. By ensuring that our application adheres to the open-closed principle, we enable our application to be scalable should we want to incorporate more services like “Lessons Service” to teach users more about data structures and algorithms.

Frontend Development

The frontend was mainly developed using React.js, where the User Interface is made up of different components. Material-UI is the main library used to create different components for the user interface in our React application.

Initially, we wanted to use the default sign-up page given but that did not match the theme of our application, hence we decided to redesign the sign-up page and added a login page. We also changed the sign-up logic. Previously, the user was able to reach the home page directly right after signing up. However, we wanted to make sure that the user remembered their password hence we directed the user back to the login page to log in.

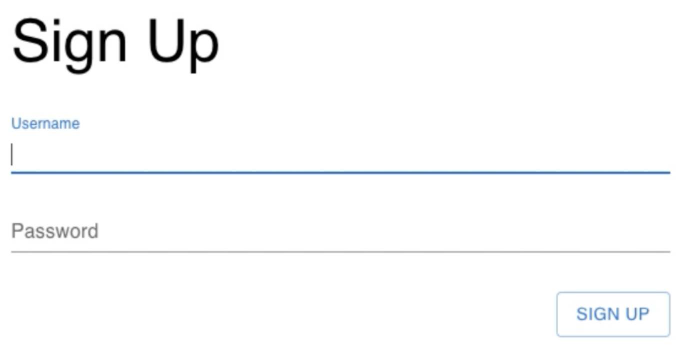


Fig F1. Initial sign up page

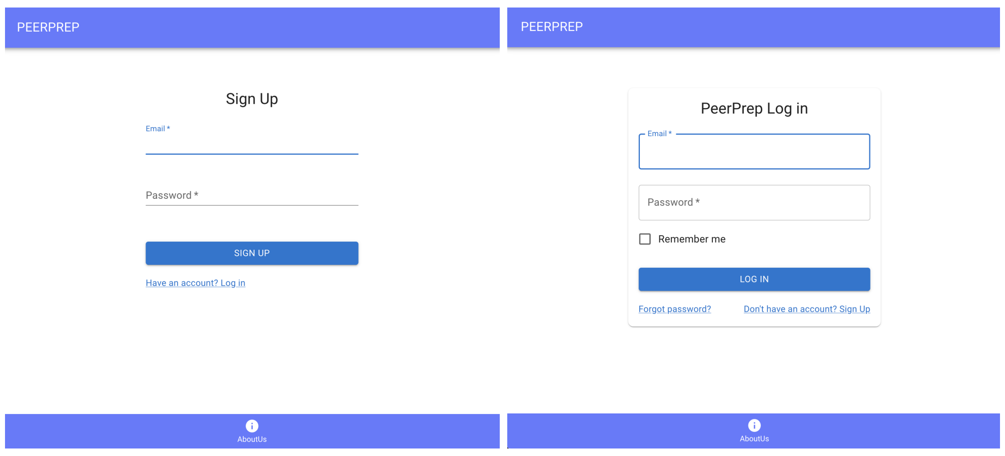
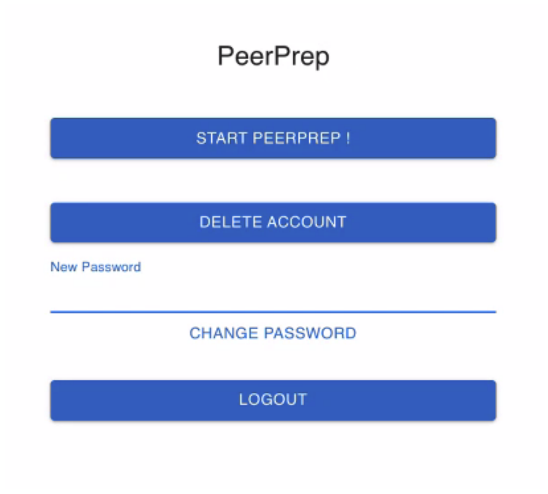


Fig F2. Final Sign-up and Fig F3 Login page

In our first milestone, we put all our functionality buttons such as changing passwords, deleting accounts, logging out, and entering into the room selection page into the home page. However we realized that it was not aesthetically pleasing, hence we decided to come up with the top navigation bar to facilitate the navigation between pages.

After gathering feedback from peers on the user experience on our application, we decided to move the room selection page to the first page after login so that there is one less click for the user to enter a room.



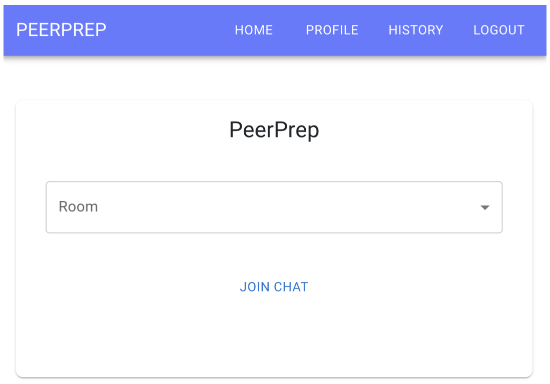


Fig F4. Initial home page and Fig F5. Final Home Page

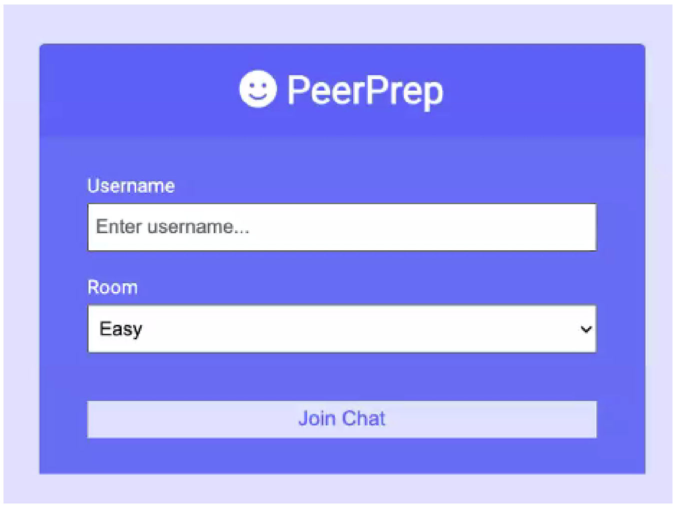
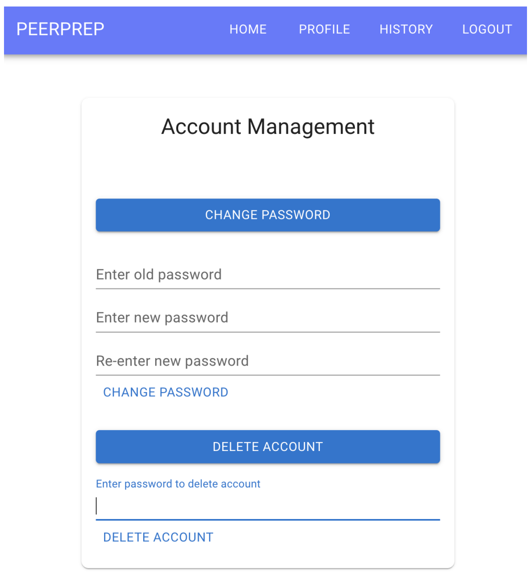


Fig F6. Initial room selection page after clicking on "START PEERPREP!" in Fig F4

We decided to simplify the design of the select room feature from Fig F6 to Fig F5 so as to achieve a cleaner user interface to match our minimalist theme of the application.



Account Management

CHANGE PASSWORD

Enter old password

Enter new password

Re-enter new password

CHANGE PASSWORD

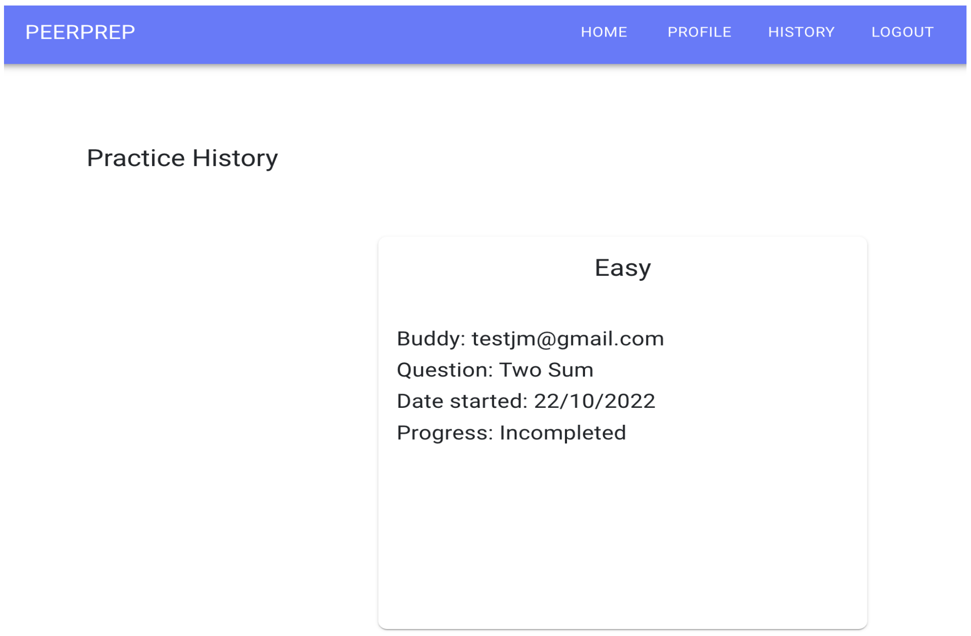
DELETE ACCOUNT

Enter password to delete account

DELETE ACCOUNT

Fig F7. Final Profile Page

Initially we wanted to have a history card for each history record as shown in Fig F10. However, we noticed a lot of space is wasted when the card is used and it might not be friendly for users when they have to scroll continuously to look for a history.



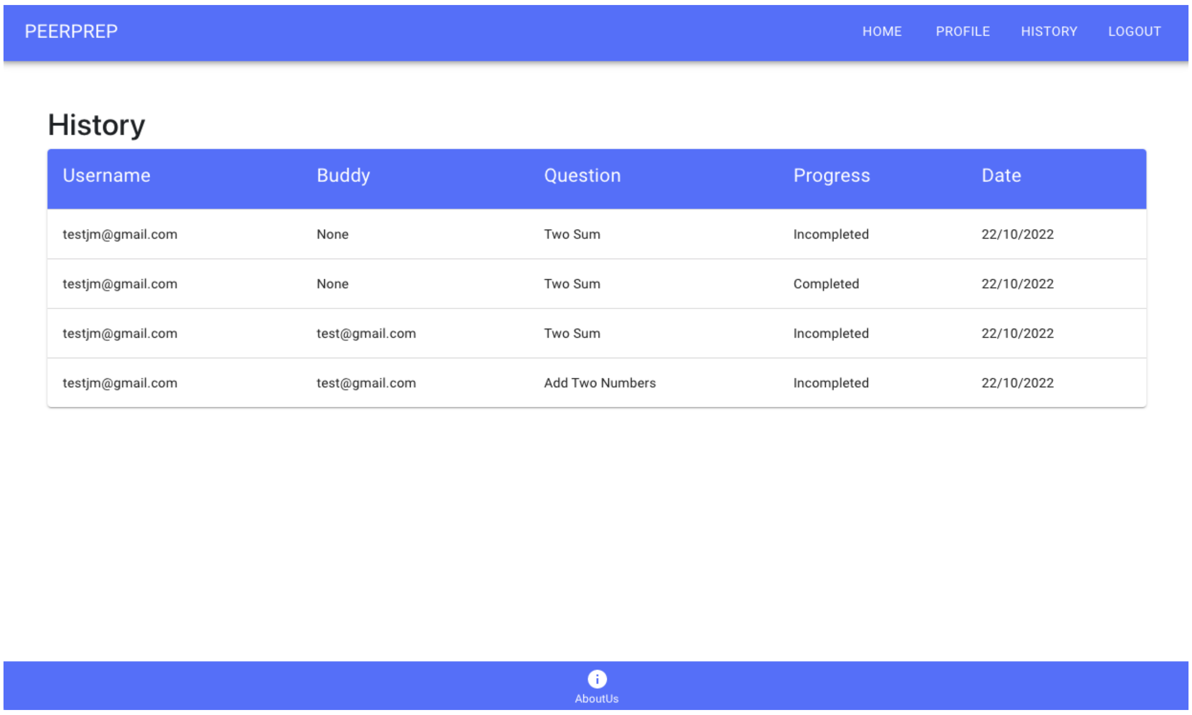
Practice History

Easy

Buddy: testjm@gmail.com
Question: Two Sum
Date started: 22/10/2022
Progress: Incompleted

Fig F8. Initial History Page

Hence, we decided to present the history in the form of a table so that records are compacted together and the user can find the record they want easily. The history records are sorted by date in ascending order. Some specific features that were not implemented were the search by keyword function, filter function, and sort by function due to time constraints.

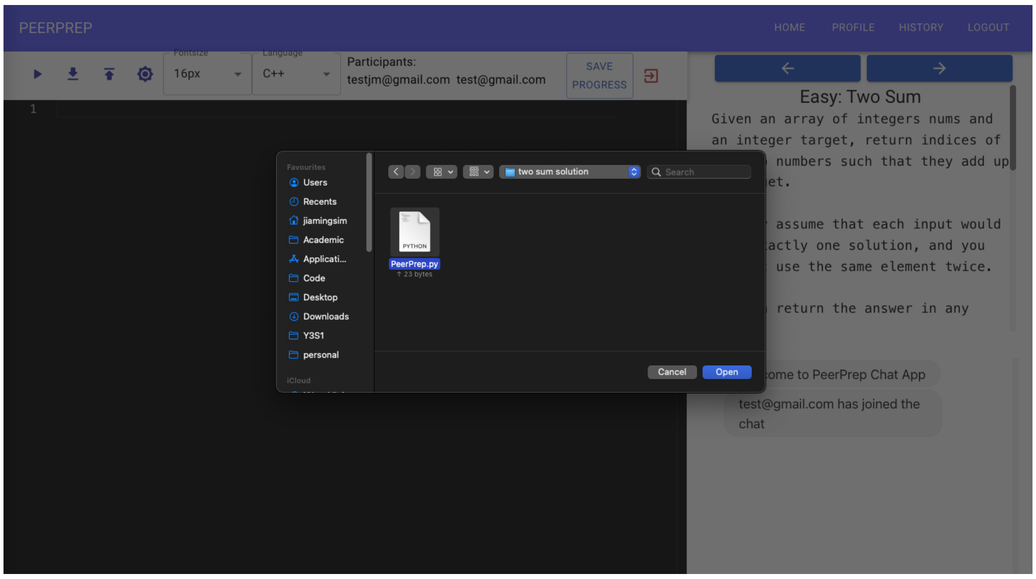


History

Username	Buddy	Question	Progress	Date
testjm@gmail.com	None	Two Sum	Incompleted	22/10/2022
testjm@gmail.com	None	Two Sum	Completed	22/10/2022
testjm@gmail.com	test@gmail.com	Two Sum	Incompleted	22/10/2022
testjm@gmail.com	test@gmail.com	Add Two Numbers	Incompleted	22/10/2022

Fig F9. Final History Page

We added the upload code feature and the download code feature so that users can save their code and reuse it in the future.



Fontsize 16px

Language C++

Participants:
testjm@gmail.com test@gmail.com

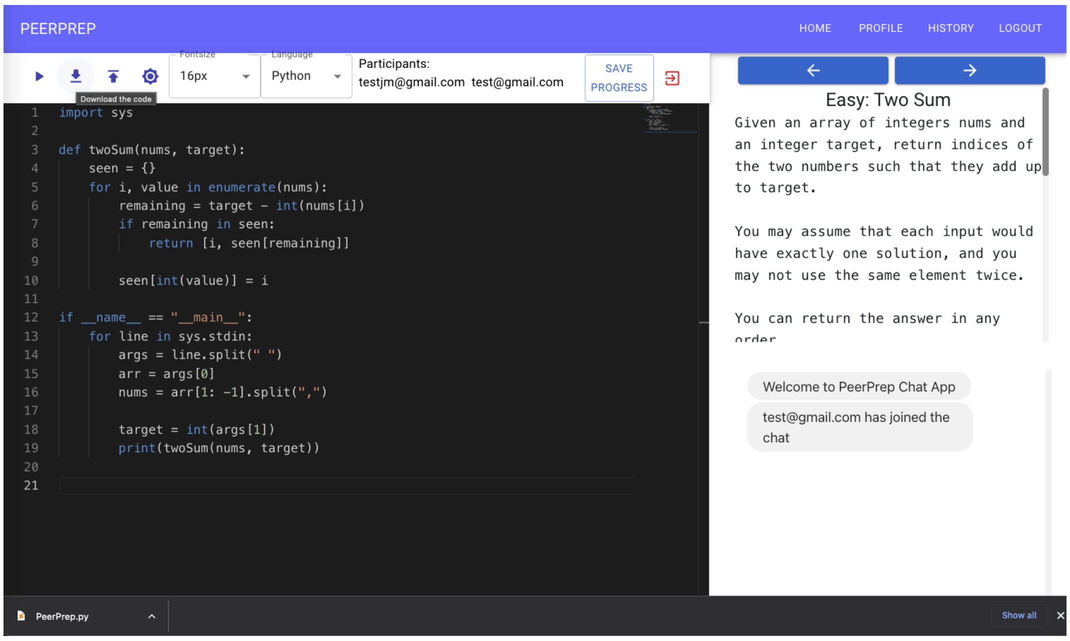
Easy: Two Sum

```
Given an array of integers nums and
an integer target, return indices of
  numbers such that they add up
et.

  assume that each input would
actly one solution, and you
  use the same element twice.

  return the answer in any
```

ome to PeerPrep Chat App

test@gmail.com has joined the chat

Fig F10: Uploaded Solution for the problem Two Sums



16px
Python
Participants:
testjm@gmail.com test@gmail.com

```
import sys

def twoSum(nums, target):
    seen = {}
    for i, value in enumerate(nums):
        remaining = target - int(nums[i])
        if remaining in seen:
            return [i, seen[remaining]]

        seen[int(value)] = i

if __name__ == "__main__":
    for line in sys.stdin:
        args = line.split(" ")
        arr = args[0]
        nums = arr[1: -1].split(",")

        target = int(args[1])
        print(twoSum(nums, target))
```

Easy: Two Sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

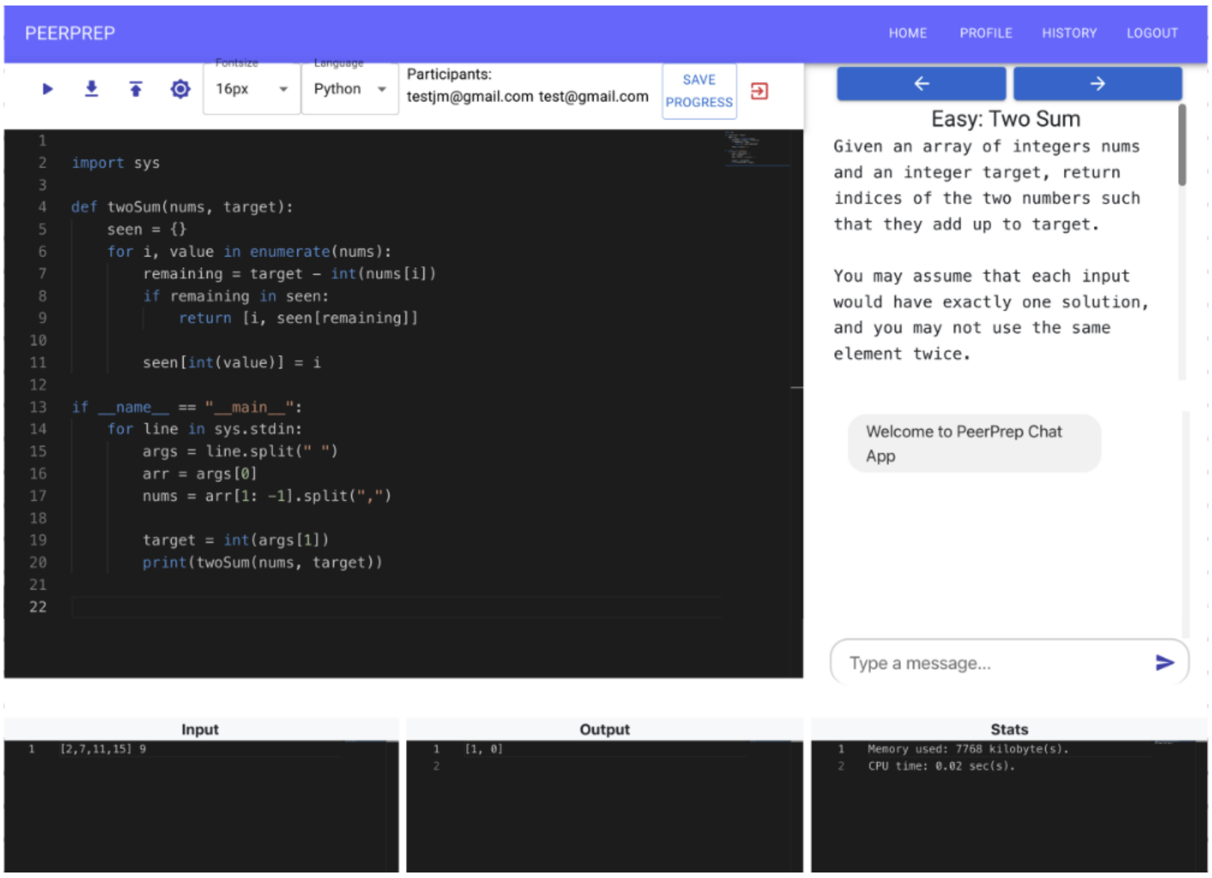
You can return the answer in any order

Welcome to PeerPrep Chat App

test@gmail.com has joined the chat

Fig F11: Downloaded the current code in the editor as a .py file

The collaboration room for two users is specially designed for the chat right under the questions so that the users are able to communicate effectively while having reference to the question and code at the same.



16px
Python
Participants:
testjm@gmail.com test@gmail.com

```

import sys

def twoSum(nums, target):
    seen = {}
    for i, value in enumerate(nums):
        remaining = target - int(nums[i])
        if remaining in seen:
            return [i, seen[remaining]]

        seen[int(value)] = i

if __name__ == "__main__":
    for line in sys.stdin:
        args = line.split(" ")
        arr = args[0]
        nums = arr[1: -1].split(",")

        target = int(args[1])
        print(twoSum(nums, target))
```

Easy: Two Sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Welcome to PeerPrep Chat App

Input

```
[2,7,11,15] 9
```

Output

```
[1, 0]
```

Stats

```
Memory used: 7768 kilobyte(s).
CPU time: 0.02 sec(s).
```

Fig F12. Collaboration Page with Chat function.

Question Service

Our team decided to have three different difficulty levels of questions that are stored in the database. In order to achieve a more efficient retrieval time of the questions from the database, we created three different question bank objects in the database based on the difficulty level. For example within the “Easy” question object, there is an array of questions in it. Hence, when the user enters the easy room and retrieves all the easy questions, the database does not need to loop through all the questions to filter out the easy question. This ensures that the performance of the question retrieval is fast regardless of the size of the question data.

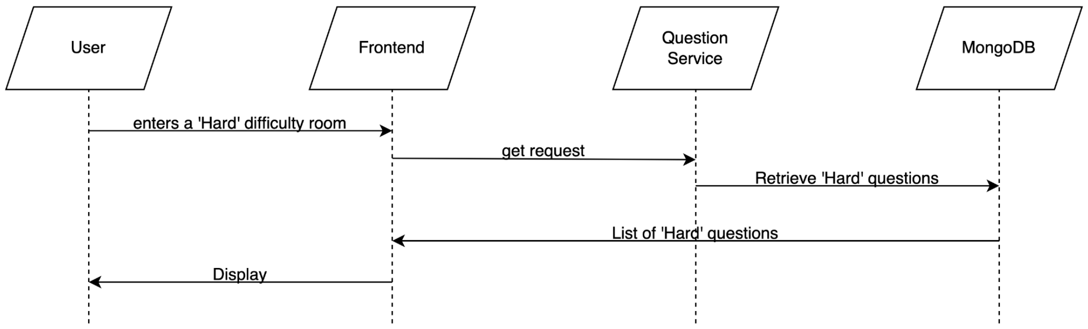


Fig Q1. Question Retrieval Sequence Diagram

History Service

At the end of every practice session, users can choose to save their progress, indicating whether they have completed the question or not. A record of their practice session, which includes their buddy name, code, progress, question, difficulty, and date of attempt will be stored in the database. The user can visit the history page to view their past Peerprep practice records.

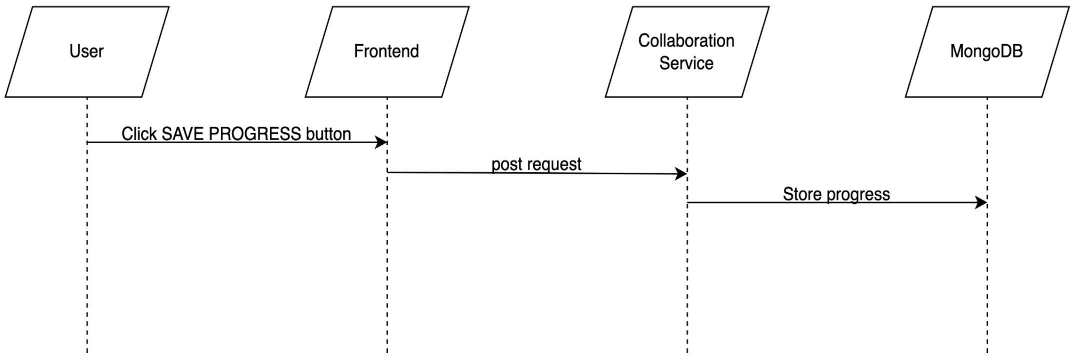


Fig H2. History Storing Sequence Diagram

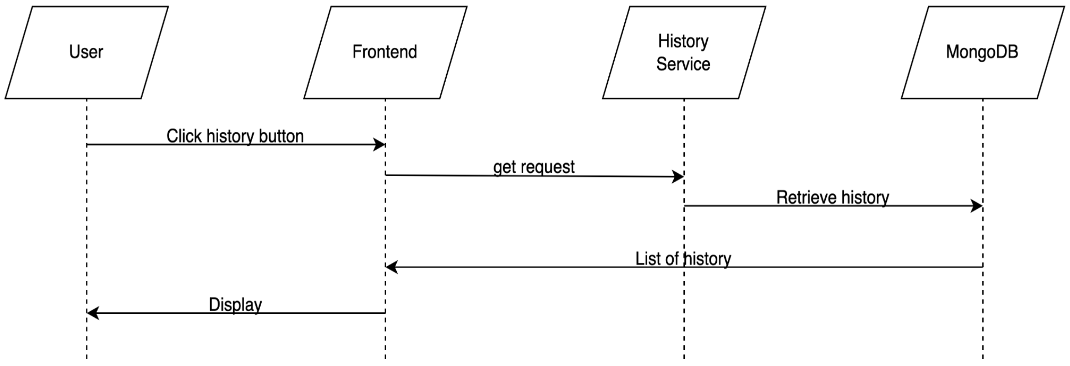


Fig H2. History Retrieval Sequence Diagram

Databases

Question Service and History Service both use MongoDB.

Question Service uses MongoDB to store the questions of Easy, Medium, and Hard difficulties. When a user enters a room of a specified difficulty, an API will be executed to retrieve questions based on the difficulty of the room from MongoDB

The History Service uses MongoDB to store the history of attempts made by the user. When the user accesses the history page, an API will be executed to retrieve the user's history from MongoDB

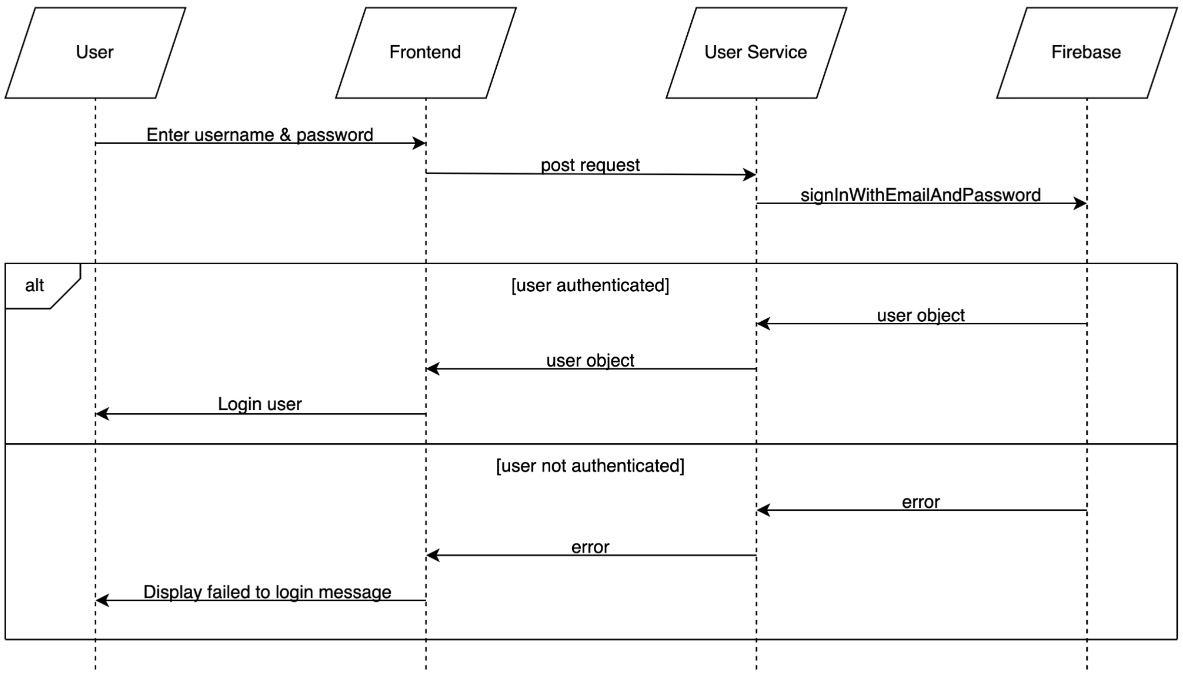
MongoDB Model Schemas

Model	**Field**	**Data Type**
Question	difficulty	string
	questions	object array
Room	roomtype	string
	roomname	string
	count	int
	users	string array
History	username	string
	buddy	
	code	
	progress	
	question	
	difficulty	
	date	

User Service

User Service uses Firebase to handle the creation and storage of users. When the user attempts to create or log in either on the sign-up or login page, an API will be executed to create or authenticate the user depending on the respective task. The sequence diagram below shows a user login event along with a description.

User logging into their account:



In this case, the user enters their username and password into the frontend, upon clicking the submit button, a post request is made to the user service. This then calls a Firebase method to authenticate the user. If the user is authenticated, the corresponding user object is returned, however, if authentication fails, an error is thrown and the corresponding error message will be displayed back in the frontend.

Collaboration (Matching)

User getting matched with another user:

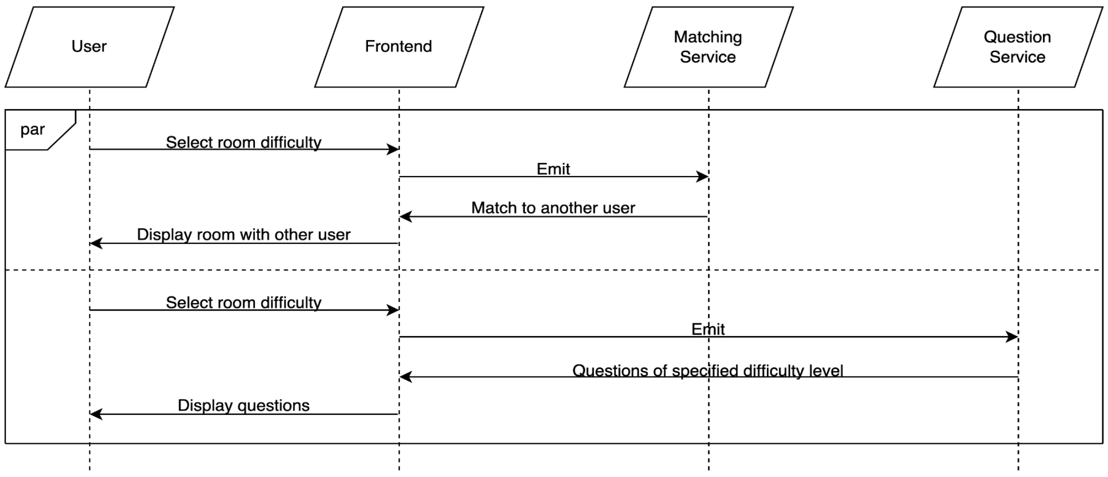


Fig C1. Assigning a user to room Sequence Diagram

In this case, a user would select a room difficulty and this is emitted to the matching service from the front end. Similarly, another user could have done the same, and they would be matched. Moreover, the questions can be fetched once two users choose the same difficulty, and can be displayed on the front end for the user.

Collaboration (Editor)

The editor that we have decided to use is Monaco Editor that is similar to the one in VS Code. The main mechanism to ensure the editing of code in real time is using Socket.IO.

Firstly, both users that entered the room will have a socket.io connection with the server in the collaboration service. Any changes made by the users will be detected by the editor and a socket.io emission of the new code changes will be made to the server.

The server will then emit these changes back to the frontend which will then be displayed to all the users in the room, and the changes will be reflected in their editor. The diagram below shows how the editor ensures the collaboration is in real-time.

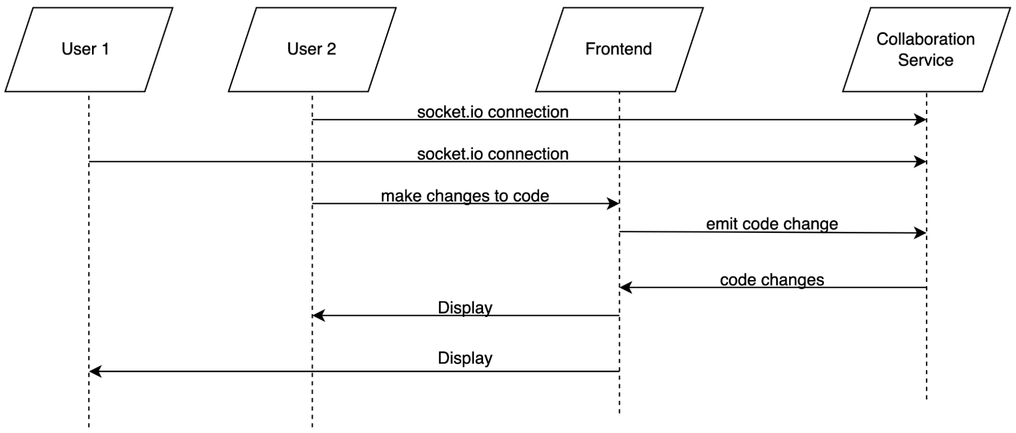


Fig C2. Concurrent code editing Sequence Diagram

Communication (Chat)

The communication feature is based on Socket.IO. When both users join the room, they will establish a socket.io connection to the server in the collaboration service. Any messages typed and submitted by the user will emit these messages to the server. The server will then emit these messages back to the frontend and be displayed to all the users that are present in the room. The messages sent will then be displayed in the message box which is viewable by all the users in the room. The diagram below shows how a message is transmitted to all the users in the room.

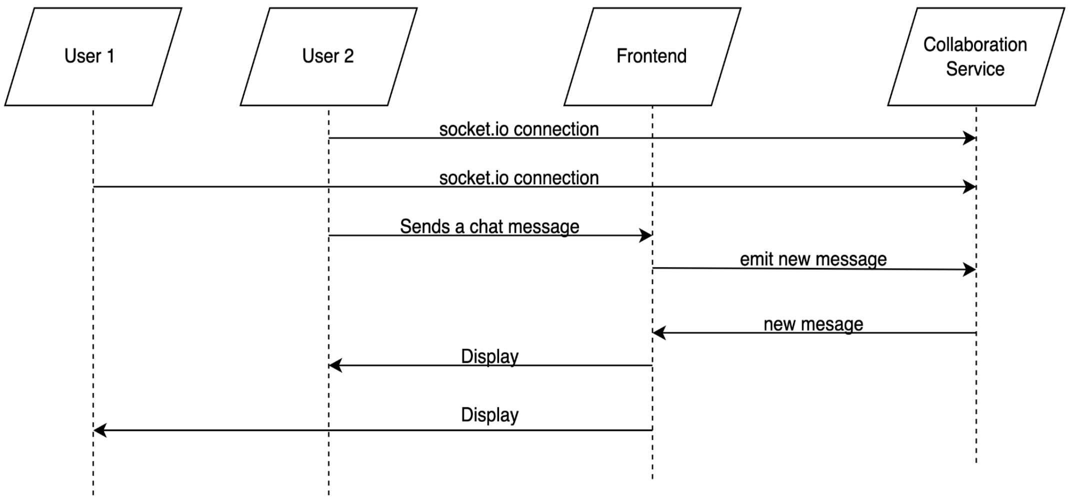


Fig C3. Chat Sequence Diagram

Project Design Decisions

Frontend

Real-time Collaborative Editor

The real-time collaborative editor was one of the application's primary features. This feature actually combines support for real-time collaboration and a code editor into a single feature.

Therefore, we made the decision to work with some of the existing libraries to address these features. We used the Microsoft-maintained Monaco Editor, which is also the code editor used in VS Code, as our code editor. The same keyboard shortcuts as VS Code are supported, and syntax highlighting is provided. This decision suited our criteria for usability because our editor now functions and feels just like VS Code, which many people are familiar with. There should be little to no learning curve as a result, and the editor is usable right away.

Design choice: Socket.IO for matching-services

For the real-time collaboration component, we used socket-io where any code changes to the editor will be detected and emitted to the matching-service server which in turn emits back the changes to all the users in the room. Hence, code changes will be reflected almost instantaneously for all users. We have incorporated socket.io for our chat feature as well.

We have tried implementing the Hackerrank team's rewrite of the Firepad Library, which is a collaborative code editor that uses the Firebase Real-time Database to store the editor's contents. However, this approach was buggy and the user who is not the first to join will have an editor that will always be read-only with no way to be adjusted to be writable. This is not ideal for us as we wanted an editor that is writable for both users.

Design choice: Download and Upload of Code

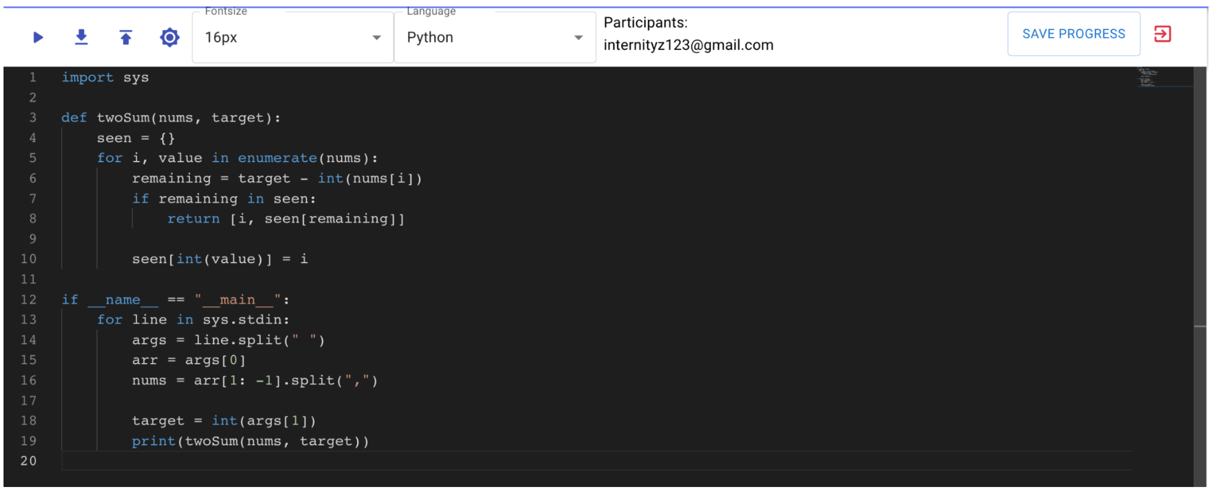
We have considered the scenario where the users have already pre-typed their own code, and also the possibility that users would want to save the code that they have discussed with their peers on PeerPrep. We decided to come up with a function that enables users to download and upload their own code. This feature will definitely improve the user-friendliness of PeerPrep as retyping codes is time inefficient whereas the downloading and uploading of code is almost instantaneous.

Design Choice: Ability to Run Code

Our collaborative editor enables users to run and debug their code. We have decided to implement this feature as we believe that the ability of running and executing the code is important to determine the correctness of the code. Without this feature, the process of copying, compiling and running the code on other platforms will be tedious.

This feature is possible with the help of a third party application, JDoodle, that has a REST-based compiler API that allows us to integrate compilers to our applications. When a user decides to run the code, an API call that contains the information of the execution such as the code and coding language will be sent to JDoodle. The result of the execution will be returned and displayed on the frontend for the users.

Furthermore, the execution results contain statistical information that could be useful for users who want to optimize their code. Information such as runtime and memory used is vital for questions of higher difficulty which often requires users to generate solutions of the highest optimality.

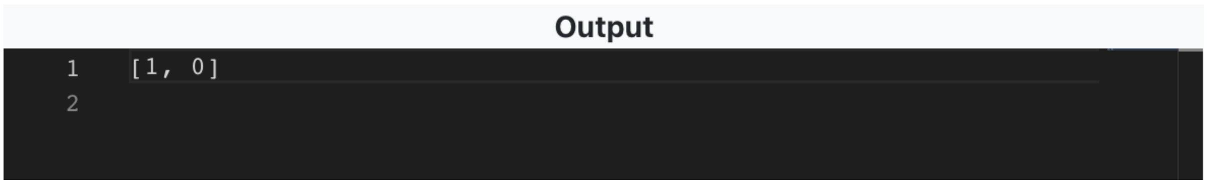


Participants:
internityz123@gmail.com

```python
import sys

def twoSum(nums, target):
    seen = {}
    for i, value in enumerate(nums):
        remaining = target - int(nums[i])
        if remaining in seen:
            return [i, seen[remaining]]

        seen[int(value)] = i

if __name__ == "__main__":
    for line in sys.stdin:
        args = line.split(" ")
        arr = args[0]
        nums = arr[1: -1].split(",")

        target = int(args[1])
        print(twoSum(nums, target))
```

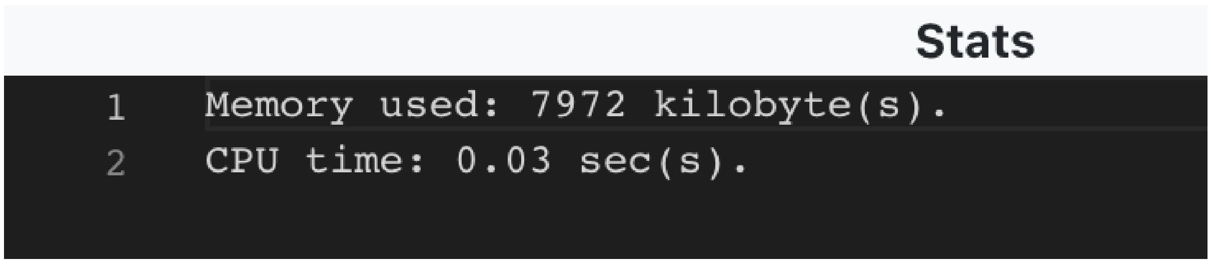
Fig P1: Solution for the problem Two Sums

Executing the above solution with the following input: `[2,7,11,15] 9` will generate the following results:



```
[1, 0]
```

Fig P2: Output for the solution Two Sums

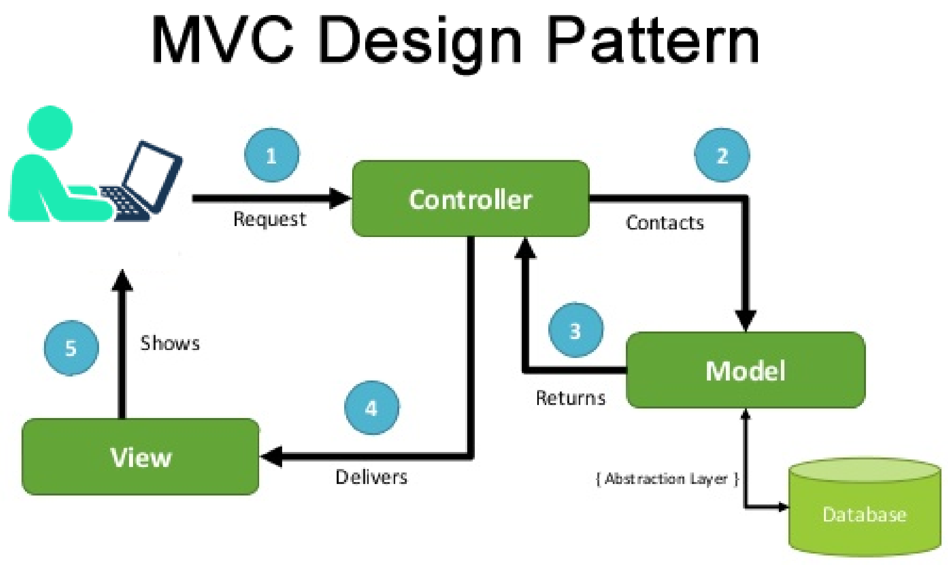


```
Memory used: 7972 kilobyte(s).
CPU time: 0.03 sec(s).
```

Fig P2: Statistics for the solution Two Sums

Backend

Software Architecture



Code Structure

The main bulk of our backend consists of code for database management functions. We decided to divide the code using MVC Design Pattern. The Controller will get the requests from the frontend and send it to the Model which will handle these requests and execute the appropriate database functions to communicate with the database. The result of the request will be returned to the frontend via View.

Architecture Styles

The codebase relies on the Shared Repository architecture style heavily. For our case, the repositories would be the Firebase Realtime Database and MongoDB. This means that all modules in the Model will have access to the databases since there are Firebase APIs and MongoDB APIs in place in the abstraction layer.

Furthermore, these APIs do not change much, reducing the requirement of a facade layer over these databases.

Design choice: Firebase for User Service

Initially we implemented the user service using our own hard-coded checks in the backend, however, we realized it was not robust in terms of security and scalability.

Hence, we decided to utilize Firebase to implement the user service. The main reason is that generating user credentials through Firebase authentication is convenient and secure. Firebase has many APIs for setting up user authentication, thus making it quite plug-and-play. Specifically, implementing the CRUD operations and user authentication simply means knowing where to call these APIs. Firebase also handles the nitty gritty of username and password requirements. For example, users are required to input emails for their username and duplicate emails are not allowed to be used during account creation. Passwords also have to be at least 6 characters long. Therefore, using Firebase helps us reduce the need to implement many things on our own.

Design choice: MongoDB for Question and History Service

We have decided to store the data of question and history services using MongoDB due to its scalability and fast data access.

We envisioned that there will be a huge volume of data that is required to be stored as the user base and the creativity of the question creators grow. Since MongoDB stores most of the data in the RAM. It allows quicker performance while executing queries. Furthermore, it is also easy to access documents using indexing. MongoDB performs 100 times faster than other relational databases and provides high performance. Also, scalability is one of the most important advantages of MongoDB. MongoDB uses “sharding”, which expands the storage capacity.

We have also taken into consideration the cons of using it. For example, MongoDB allows a limited size of only 16 MB for a document. Performance nesting for documents is also limited to only 100 levels. This might cause an issue if we decide to store more data per document. However, given that the information stored per question and history, we believe that it is unlikely that we will exceed the size limit.

We have also considered using firebase but we have decided MongoDB is a much more reliable database. The comparison of these two databases can be summarised in the table below.

Criteria	Firebase	MongoDB
Performance	Slower	Faster
Use of Service	Anybody, anywhere, and anonymously may utilize the Firebase	Only authorized users can access the database’s stored data in MongoDB.
Usage	real-time synchronization	high-speed logging in an application.

Comparison of Firebase vs MongoDB

Design Choice: Reduce waiting time for occupied room

To ensure fairness for users already waiting in the room, we choose to implement a FIFO queue system which comprises queues of different room difficulties. Whenever a user joins a room and the room is not fully occupied, it will be added into the queue. When a new user wants to join the room of the same difficulty, the queue of the specified difficulty will perform a deque operation that removes the first room from the queue, and the new user will join that room.

This implementation takes into consideration the instances of multiple half-occupied rooms. It prevents a situation where a user who is already waiting in the room for a long time, to be matched later compared to a user who has only just started waiting.

We implemented this queue system using MongoDB realtime database and the query for the next available room can be processed instantaneously from the frontend.

Another way to implement this would be to store the queue in a local cache such as Redis. However, this is a risky option as a crash in the local server might cause the data to be lost. Even with a 30 second time-out in-place, in a situation where many users are waiting, it will affect user experience as many of them will hit time-out if a server is down.

Suggestions for improvements and enhancements to the delivered application

Deployment

We could have done better in terms of deployment. Currently, our app can only be run locally. Furthermore, we would have to manually start the different services to get the entire app running. This could have been automated by dockerizing the different microservices into containers and running them all at once locally. Furthermore, we could have deployed the application on Heroku or Netlify, or Amazon Web Services instead of running it locally. This could have been done in various ways, including the usage of the docker image to be deployed. However, as the app is to be used by a small number of users currently, we felt that deployment would not have been of high priority and hence we did not manage to pursue it by the end of the project.

Scalability

We could have included some way to manage the load balancing and allow the application to scale when there is a high load.

Improving the scalability of the application would be doable with Kubernetes. However, due to the practical nature of developing the application for a small user base, we realized that implementing this would not be meaningful currently because the application is only used by a small number of users currently.

Authentication

The app could have some way to enable authentication via other means, i.e, using federated identity providers like Google or Facebook sign-in etc.

The foundation for such a feature has been made because Firebase authentication has been implemented. This can easily be extended by enabling access in the Firebase console for the sign-in methods. However, this would require some effort on the front end side, where we would need to include new options on the front end for the choice of login options. However, we did not manage to implement it due to time constraints.

Tracking session history

The app could have a feature for tracking the session history, that allows the maintenance of the information and the history of the session, be it the chat messages, the code typed and the questions accessed.

There is some basis for this feature. We already have an existing history service that keeps the questions that are done. We could extend this to keep track of the chat messages and code typed. However, there would be some re-designing of the database and choosing how to best store the information. That being said, this careful design and implementation would have taken too much time and due to reprioritization, we decided not to implement this feature.

Question matching algorithm

The app could have a matching algorithm that prioritizes the matching based on the difficulties of the questions done.

Currently, the matching is done via the same difficulty. This forms the basis to include the algorithm that is based on information from the history service. The algorithm could have used this information to match the most frequent difficulty level chosen or the most recent difficulty level chosen. Yet, this would require some changes in the database as well as the implementation of the algorithm for prioritization. This was not a mission-critical feature to have and due to prioritization of the features, we decided to not implement this feature.

Reflections and learning points from the project process

Reflections

On a broad view, we thought that the project was a great learning experience and exposed us to new frameworks as well as technologies that are used in the industry. Initially, we did face some difficulties initially as we needed to get the minimum viable product to start working and we had to get used to each other's working styles. However, after the first milestone, we managed to get a better footing and were able to make progress incrementally.

The project also instilled into each and every one of us that it is possible to learn new technologies on the fly, as long as we have the basic theories and principles. Furthermore, we all found that the project has helped us to appreciate the benefits of working in a team, as debugging was made easier when done as a team or in pairs. Lastly, we saw the real-life application of the theories and patterns that we have learned from the lectures throughout the project, which helped us to enforce the learnings of the entire module.

Overall, we had many learnings from the projects which we will highlight below as well.

Learning point 1: There is a need for experimentation to find out what works best for your team.

Initially, we experimented with project management tools such as GitHub actions and Jira, but we soon realized that Jira would have been a better choice because it was a much easier way to centralize all the tickets and have an overview of what needed to be prioritized. This was one of the examples where we experimented with new technologies and tools to find out what works best for us.

Learning point 2: There is a need to coordinate properly when implementing certain features.

We realized that there were coordination issues when we were implementing certain features. In particular, the features of concern were the ones that required heavy implementation of the back end but rather easier implementation on the front end. These features meant that members working on the front end had to wait for the back end to be implemented in order to test the feature or mark the feature as complete.

This meant that development time was longer than necessary. However, we fixed this by ensuring that members coordinated carefully such that the member working on the back end would start first and the member on the front end would start working a few days later to match the timeline such that no members would have to wait for one another.

Learning point 3: Investing time to read documentation could reduce overall development time when working with new packages and libraries.

With the nature of the project using many new technologies which we may not have used before, we tried googling and watching tutorial videos online to learn about these technologies. However, past a certain point, the benefit of googling and watching videos would be minimal, and reading the documentation while implementing would reduce the chances of errors and reduce the total development time.

We only realized this after a few iterations of sprints, which meant that we could have pushed out features much faster. After realizing this, we streamlined the process of how we learn new technologies and relied more heavily on documentation, which helped us to cut down development time by a significant portion.

Appendix

Timeline:

Team 002 - DEVELOPMENT PLAN			MS1	Recess			MS2				MS3 9Nov Wed 5pm	
	Feature	Feature type	W6	R	W7	W8	W9	W10	W11	W12	W13	
	F1. User Management	Development - MUST HAVE										
F1.1	The system should allow users to create an account with username and password.	High	🟩									🟩
F1.2	The system should ensure that every account created has a unique username.	High	🟩									🟩
F1.3	The system should allow users to log into their accounts by entering their username and password.	High	🟩									🟩
F1.4	The system should allow users to log out of their account.	High	🟩									🟩
F1.5	The system should allow users to delete their account.	High	🟩									🟩
F1.6	The system should allow users to change their password.	High	🟩									🟩
F1.7	The application should ask the user to double confirm a delete account request	High			🟩							🟩
F1.8	The application must use identity management software (e.g Auth0) to prevent unauthorized access	High			🟩							🟩
F1.9	The users' passwords should be hashed and salted	High			🟩							🟩

Team 002 - DEVELOPMENT PLAN			MS1	Recess			MS2				MS3 9Nov Wed 5pm	
	Feature	Feature type	W6	R	W7	W8	W9	W10	W11	W12	W13	
	F2. Matching	Development - MUST HAVE										
F2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High	🟩									🟩
F2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High	🟩									🟩
F2.3	If there is a valid match, the system should match the users within 30s.	High	🟩									🟩
F2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High	🟩									🟩

Team 002 - DEVELOPMENT PLAN			MS1 Recess				MS2				MS3 9Nov Wed 5pm	
	Feature	Feature type	W6	R	W7	W8	W9	W10	W11	W12	W13	
	F3.Collaboration	Development - MUST HAVE										
F3.1	The application should provide a front end IDE for concurrent code editing.	High						🟩				🟩
F3.2	The application should provide a back end to conenct the IDE for concurrent code editing.	High						🟩				🟩
F3.3	The application should be able to compile the code, take in any inputs and return any ouputs.	High							🟩			🟩
F3.4	The application should be able to show the memory used and runtime of the code.	Low							🟩			🟩
F3.5	The application should be able to allow the users to choose the programming language they want to use to solve the question.	High							🟩			🟩
F3.6	The application should be able to allow the users to change the font size of their codes.	Low							🟩			🟩

Team 002 - DEVELOPMENT PLAN			MS1 Recess				MS2				MS3 9Nov Wed 5pm	
	Feature	Feature type	W6	R	W7	W8	W9	W10	W11	W12	W13	
	F4. Question	Development - MUST HAVE										
F4.1	The application must allow users to choose the difficulty level of the questions.	High						🟩				🟩
F4.2	The difficulty of the questions allocated must be of the user specified difficulty.	High						🟩				🟩
F4.3	The questions should have a difficulty level tag.	Medium						🟩				🟩
F4.4	The user should be allowed to request another question of same difficulty after completion	Medium						🟩				🟩

Team 002 - DEVELOPMENT PLAN			MS1	Recess			MS2				MS3 9Nov Wed 5pm	
	Feature	Feature type	W6	R	W7	W8	W9	W10	W11	W12	W13	
	F5. Frontend - BASIC + AI	Development - MUST HAVE										
F5.1	The application should have a login page.	Medium	🟩									🟩
F5.2	The application should have a signup page.	Medium	🟩									🟩
F5.3	The application should have a home page.	Medium	🟩									🟩
F5.4	The application should have a profile page.	Medium		🟩								🟩
F5.5	The application should ask for confirmation for delete account / edit password.	Medium		🟩								🟩
F5.6	The application should ask for old password when edit password.	Medium		🟩								🟩
F5.7	The application should require user to re-enter new password.	Medium		🟩								🟩
F5.8	The application's collaboration page should have the editor, chat, question, input, output and statistics displayed.	High							🟩			🟩
F5.9	The application should have an overall minimalist style.	Medium								🟩		🟩

Team 002 - DEVELOPMENT PLAN			MS1	Recess			MS2				MS3 9Nov Wed 5pm	
	Feature	Feature type	W6	R	W7	W8	W9	W10	W11	W12	W13	
	F6. Communication	Developement - Nice to have										
F6.1	Chat communication between users	High	🟩									🟩
F6.2	Connect Communication service to DB	High		🟩								🟩
	F7. History Service	Development - Nice to have										
F7.1	The application should be able to save history of a Peerprep session	Medium							🟩			🟩
F7.2	The applciation should be able to display a history of all the Peerprep sessions for each user	Medium							🟩			🟩