NUS SCHOOL OF COMPUTING

AY 2022/2023 SEMESTER 1

CS3219 SOFTWARE ENGINEERING PRINCIPLES AND PATTERNS

FINAL REPORT

Prepared By:

G 23

Name	Student ID
GOH TIAN YONG	A0200679A
HO PIN XIAN	A0149796E
OLIVIA JULIANI JOHANSEN	A0205672H
WAYNE TAN KIN LUN	A0200801A

TABLE OF CONTENT

1. Introduction

1.1. Purpose

PeerPressure is a web application designed for computer science students to practise algorithm questions with other students. As the name implies, PeerPressure provides users with peer pressure from other users to encourage them to practise for interviews and perform during the actual one. We believe PeerPressure will be an engaging app that keeps users in tip top condition by closely simulating a standard interview environment.

1.2. Document Conventions

Every table and diagram has a corresponding table/ figure number located at the bottom. This number indicates the section of the report it is in and the sequence number. (e.g. Table 2.1 indicates that this is the first table inside section 2).

1.3. Intended Audience and Reading Suggestions

This document is intended to be read by developers working on subsequent versions of PeerPressure. The developer guide aims to align developers on the intended mission of the application and to onboard them on to the overall structure of the application. After reading this guide, developers are expected to be equipped with the knowledge to add features to existing services of PeerPressure, debug unforeseen issues and integrate new services.

It is suggested that readers read section 3. Overall Description to have an understanding of the structure of PeerPressure and the constraints under which we expect PeerPressure to function. Readers looking to start working on a specific requirement or service may identify the area in which they are working on, by searching in section 4. System Features to get a more specific understanding of the service they are working with. Readers may then read the specific subsections of 5. Design of PeerPressure to understand how their features may be integrated with the existing code.

1.4. Project Scope

The project scope of PeerPressure involves gathering functional requirements (FR) and non-functional requirements (NFR), writing a development guide and implementing PeerPressure in accordance with the requirements gathered. The codebase of PeerPressure is accessible at Github and a running deployment here https://peerpressure-366414.et.r.appspot.com.

1.5. References

https://refactoring.guru/design-patterns serves as a strong reference in guiding the design decisions seen in section 4. Design of PeerPressure and will be useful for readers to refer to when deciding on the specific implementation choices.

Users may reference the following links for specific documentation of the tech stack used in PeerPressure.

- Socket.io: https://socket.io/docs/v4/
- Yjs: https://docs.yjs.dev/
- MongoDB Atlas: https://www.mongodb.com/atlas/database
- ReactJS: https://reactjs.org/docs/getting-started.html
- ExpressJS: https://expressjs.com/
- NodeJS: https://nodejs.org/en/
- GCP app engine: https://cloud.google.com/appengine/docs/standard/nodejs/runtime.

2. Individual Contribution

Name	Technical Contribution	Non-Technical Contribution
Goh Tian Yong	Set up - Frontend - Mongodb Feature - User service API endpoints - History service (frontend) - Authentication for user API test - User service	Figma - Designed entire frontend UI see figma Documentation - Subsection 4.2: Non-functional Requirements - Subsection 3.5: Design and Implementation Constraints - Subsection 4.1.1: User service design
Ho Pin Xian	Set up - Deployment of app on Google Cloud Platforms Feature - Communication service - User service API endpoints (JWT) Tests - Collaboration service - Communication service	Documentation - Section 1: Introduction - Subsection 4.1.5: Communication service requirements - Subsection 5.4: Communication service design
Olivia Juliani Johansen	Set up - Frontend CI/CD Feature - Matching users - Collaboration service API endpoints - Real-time collaborative editor - Updating history	Documentation - Section 3: Overall description - Subsection 4.1.2: Matching service design - Subsection 4.1.4: Collaboration service design
Wayne Tan Kin Lun	Set up - Mongodb Feature - Matching service API endpoints - History service API endpoints - Question service API endpoints Test - Matching service API endpoints - Matching service socket testing - History service API endpoints - Question service API endpoints	Documentation - Subsection 4.1: Functional requirements - Subsection 4.1.6: History service design - Subsection 4.1.3: Question service design

Table 2.1 Individual Contributions

3. Overall Description

3.1. Product Perspective

The product stakeholders, such as students in the technology industry increasingly face challenging technical interviews and need a better way to practise for coding interviews. This is where our interview preparation platform and peer matching system, PeerPressure comes into play. PeerPressure aims to serve as a way for stakeholders to improve their communication and collaborative problem solving skills to better prepare them for interviews.

3.2. Product Features

Feature 1: Authentication	Service	FR	NFR	Test ID
Login	User Service	FR1.3	NFR1.2, NFR3.2	6.1.10
Register	User Service	FR1.1	NFR1.1, NFR3.1	6.1.1
Logout	User Service	FR1.4	N.A.	N.A.
Feature 2: User Account				
Delete account details	User Service	FR1.5	NFR1.3	6.1.11
Reset password	User Service	FR1.6	NFR1.4, NFR3.3	6.1.15, 6.1.18
Feature 3: History				
View the question. users. and code for past collaborations	History Service	FR6.4	NFR1.7, NFR3.7	6.6.2
Feature 4: Match user				
Select question difficulty level	Matching Service	FR2.1	N.A.	6.4.1, 6.4.2, 6.4.3
Match to another user with same difficulty level	Matching Service	FR2.2	NFR2.1, NFR3.4	6.4.4, 6.4.5
Feature 5: Shared code editor				
Users in the same room should see the same text in the editor	Collaboration Service	FR4.2	NFR2.4	N.A.
Users in the same room should be able to update the text in the editor	Collaboration Service	FR4.1	NFR2.5	N.A.
Feature 6: Coding questions				
Users in the same room should be able to see the question of their chosen difficulty	Question Service	FR3.2	NFR2.2	6.5.2.

Feature 7: Communication				
Users in the same room should be able to hear each other speaking in real time	Communication Service	FR5.1, FR5.2, FR5.3	NFR2.6, NFR3.6	6.3.1, 6.3.2, 6.3.3, 6.3.4

Table 3.2.1: Requirement Traceability Matrix

3.3. Modules and Characteristics

Module	Responsibility	Tech stack
Frontend	User interface that user will interact with	React, Material UI, socket
User Service	Authentication, user account	ExpressJS, NodeJS, mongoose
Matching Service	Selecting difficulty, matching users	ExpressJS, NodeJS, mongoose, Socket.io
Collaboration Service	Shared code editor	ExpressJS, NodeJS, mongoose, Socket.io, yjs framework
Question Service	Supply questions of easy, medium, difficulty levels	ExpressJS, NodeJS, mongoose
Communication Service	Support voice communication	ExpressJS, NodeJS, Socket.io
History Service	Saves past collaborations	ExpressJS, NodeJS, mongoose

Table 3.3.1: Module Characteristics

3.4. Operating Environment

Browser: Firefox, Chrome
OS: MacOS, Windows

3.5. Design and Implementation Constraints

PeerPressure uses the microservices architecture and is implemented using the API-REST based topology and database per service pattern. We split the server into 6 services, each performing specific business functions, independent of all the other services. They are accessed via a REST-based interface implemented in the API layer. On the client side is a Multi Page React application which calls the API exposed by each service.

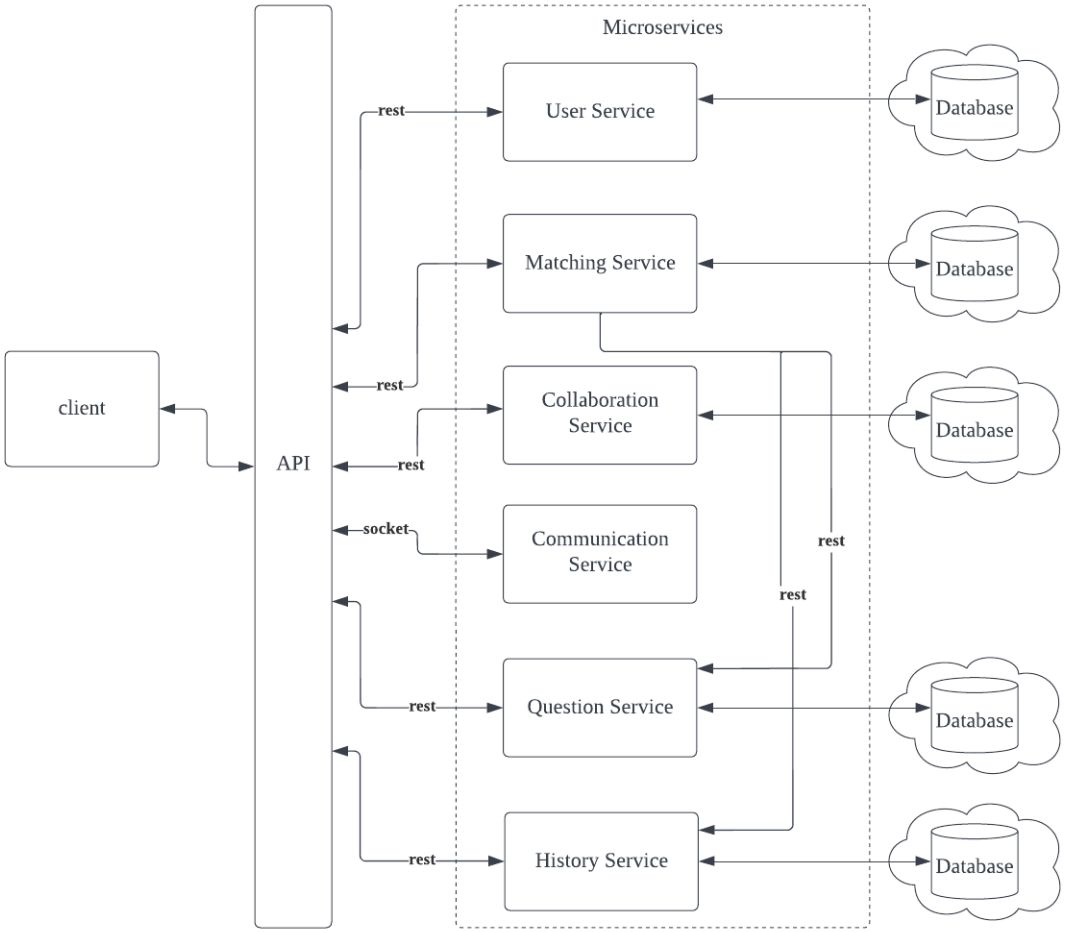

Figure 3.5.1: Architectural Diagram

Implementation Constraints

Constraint 1: TIme Constraint

The project had a hard deadline of 12 weeks. As such at times, we had to de-prioritise maintainability and extensibility in order to work on the must-have features for the project.

Constraint 2: Knowledge Constraint

As our team was familiar with only a few tech stacks, and together with the time constraint on the project, we usually chose our tech stack based on familiarity and experience rather than the absolute best tools for our specific use case in order to develop a well functioning product within the time given.

Design Decisions

Design	Pros	Cons
Micro-services	- Low coupling. Services communicate via inter-service communication (async or sync) rather than inter-process communication. A change in one service will not greatly propagate other services. - High cohesion as each service only contains strongly related functionalities related to the service. - High Scalability. Developers can customise the tech stack and frameworks according to the resource needs of each service. - High productivity. Services are independent and developing each service can be done simultaneously. - Maintainability as it is easier to read and debug smaller and focused components rather than 1 big component.	- Require more resources such as network bandwidth, and databases. - Increased complexity due to inter-service communication. - Complex deployment due to the coordination needed between services.
Monolith	- Simple deployment as we only have to monitor and deploy one overall component. - Low latency as everything is built and data transfer is within a single unit.	- Low scalability and only supports horizontal scaling making the resource use inefficient - High coupling and changes propagate to the whole project. - Corroboration difficulty. As there is only a single unit, developers need to distribute the work to avoid massive merge conflicts.
Layer (n-tier)	- Achieves separation of concerns and single responsibility principle as the system is separated into different layers based on functionality. This also achieves low coupling - a change in one component does not propagate to the whole product. - Maintainability as it is easier to read and debug smaller and focused components rather than 1 big component.	- Can lead to tight coupling if there are too many modules within each layer. - Small modifications require redeployment of the whole application resulting in significant downtime. - Slower performance when more layers are added as processed need to interact with every layer

We adopted the microservices architecture because it achieves low coupling, high cohesion, high scalability, productivity and maintainability. Additionally, the cons of more network and database resources needed and complex deployment is balanced by being able to use the computing resources more efficiently as each service can be vertically scaled depending on demand. The increased complexity of inter-service communication is generally more efficient than the high level of coordination and communication needed in monolith applications.

Table 3.5.1: Overall Architecture

Design	Pros	Cons
Shared Database	- Less databases needed as all services share a database. - Less data replication across databases. - Direct access to the database to read or write any necessary data.	- Single point of failure. If one data server goes down, the entire system is disrupted. - Data model may become bulky since it is used across multiple services. - Changes to the data model will propagate to all services. - Increased complexity of the database since all of the data are stored in the same database.
Database per Service	- Services are loosely coupled and changes to the data model is confined to the database's service - Supports customised database and data models that are best suited to the functionalities of the service.	- Challenging to implement queries that join data across multiple databases - Increased complexity of managing multiple and different (SQL, noSQL) types of databases
We chose the database per service pattern as it achieves loose coupling and data customization. Although it is more challenging to read data across multiple databases, we can still do this by replicating the relevant data across databases. This also comes with faster read queries since the data models now are more lean. As for the increased complexity in managing multiple and different types of databases, this can be mitigated with database planning and additionally, the benefit of database and data model customization outweighs this cost.		

Table 3.5.2: Microservices Database Pattern

Design	Pros	Cons
Repository pattern	- Achieves separation of concerns and low coupling by separating domain logic from the data source CRUD operations. Changes to data stores do not propagate to the domain logic. - Achieves don't repeat yourself (DRY) principle via the base repository as the code to query and fetch data for any data model is not repeated	- Adds a layer of abstraction (of the data layer) which adds complexity
Command /Query Separation Pattern	- Better scalability as it supports database and data model customization especially because read queries are usually more frequent than writes	- Increased complexity and cost of having to manage multiple, complex databases
We went with the repository pattern because it achieves separation of concerns, low coupling, and don't repeat yourself. This benefit exceeds the added complexity of the additional layer of abstraction.		

Table 3.5.3: Service architecture

3.6. Assumptions and Dependencies

Assumptions

Technology Assumptions

1. Users have either Windows or Mac OS and have access to chrome or firefox to access our app.
2. Users have access to internet connection that does not block connections with MongoDB Atlas.
3. Users have a device with a speaker and microphone to communicate with the users in the room.
4. Users device has sufficient RAM space to support redis cache

Standard Assumptions

1. Users have a valid email to register an account with PeerPressure
2. As the app does not support localization, users must be able to write, read and speak english.

Dependencies

Logical Dependencies

1. A interview practice session can only begin if another user has chosen the same difficulty level as this user
2. As users in the session are randomly found, the effectiveness of the interview practice can vary greatly and largely depend on the quality of interaction of the two users.

Resource Dependencies

1. Many client sockets may connect to only one socket server. A method to scale is by having each socket server own a socket client that connects to an overall socket server. However, this comes at the cost of setting up an additional server instance.
2. There is only one deployed server instance so if too many users begin the matching, it might overload the server and some users will not be able to use the application.

4. System Features

We have used high, medium and low to rank our functional and non-functional requirements. We have defined the ranks as such:

Rank	Description
High	Must-have requirements. They are crucial for the project. If the requirement is not present in the project. Application would not be a complete application hence could be considered as a failed project.
Medium	Nice-to-have requirements. They are extra features that add value to the application or to give a better experience to the user.
Low	Low priority requirements. They are to be included at a later date as they either do not contribute much of the value to the application or they not yet affect the functionality of the usual application operation.

Table 4.1: Requirement Priority

4.1. Functional Requirements

We have categorised all the functional requirements by each service:

- User service
- Matching service
- Question service
- Collaboration service
- Communication service
- History service

Additionally, we have referenced the code implementation and test cases to each requirement.

4.1.1. User Service

S/N	Requirement	Priority	Code Reference	Test case reference
FR1.1	The system should allow users to create an account with username, email and password.	High	`User-controller .createUser`	user-service/test/createUserTest.js 6.1.1
FR1.2	The system should ensure that every account created has a unique username.	High	`User-model .UserModelSchema`	user-service/test/createUserTest.js 6.1.4
FR1.3	The system should allow users to log into their accounts by entering their username and password.	High	`User-controller .loginUser`	user-service/test/loginUserTest.js 6.1.10
FR1.4	The system should allow users to log out of their account.	High	`Navbar.handleLogout`	N.A.
FR1.5	The system should allow users to delete their account.	High	`User-controller .deleteUser`	user-service/test/loginUserTest.js 6.1.11
FR1.6	The system should allow users to change their password.	Medium	`User-controller .resetPassword`	user-service/test/forgetPasswordTest.js 6.1.15 user-service/test/resetPasswordTest.js 6.1.18

Table 4.1.1.1: User service functional requirements

4.1.2. Matching Service

S/N	Requirement	Priority	Code Reference	Test Reference
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High	`Match-controller .createMatch`	matching-service/test/test.js 6.4.1, 6.4.2, 6.4.3
FR2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High	`sockerController .updateMatchedUser`	matching-service/test/socketTest.js 6.4.4, 6.4.5
FR2.3	The system should inform the users that no match is available if a match cannot be found.	High	`socketController .initSocketEventHandler .socket.on("stop_find_match")`	N.A.
FR2.4	The system should provide a means for the user to leave a room once matched.	Medium	`RoomPage.handleLeaveRoom`	N.A.
FR2.5	The system should allow users to match with the other user they want to work with.	Medium	Not Implemented	Not Implemented

Table 4.1.2.1: Matching service functional requirements

4.1.3 Question Service

S/N	Requirement	Priority	Code Reference	Test Reference
FR3.1	The system should be able to store question banks indexed by difficulty level.	High	`question-controller .createQuestion`	question-service/tests/test.js 6.5.1
FR3.2	The system should be able to randomly select one of the questions from given difficulty	High	`question-controller .getQuestionByDiff`	question-service/tests/test.js 6.5.2
FR3.3	The system should be able to filter away questions that a user has seen before.	Medium	Not Implemented	Not Implemented

Table 4.1.3.1: Question service functional requirements

4.1.4 Collaboration Service

S/N	Requirement	Priority	Code Reference	Test Reference
FR4.1	The system should allow both users to code concurrently	High	`new QuillBinding( ytext, q, awareness)`	N.A.
FR4.2	The system should allow users to view if the other user has highlighted a piece of code.	Low	Not Implemented	Not Implemented
FR4.3	The system should allow users to auto format the code	Low	Not Implemented	Not Implemented

Table 4.1.4.1: Collaboration service functional requirements

4.1.5 Communication Service

S/N	Requirement	Priority	Code Reference	Test Reference
FR5.1	The system should allow user to join a call	High	`socketController .initSocketEventHandlers .socket.on("userInfo")`	communication-service/test/socketTest.js 6.3.1
FR5.2	The system should allow user to leave the call whenever they want	High	`socketController .initSocketEventHandlers .socket.on("userInfo")`	communication-service/test/socketTest.js 6.3.2
FR5.3	The system should allow users to hear the other user speaking if both are in the call.	High	`socketController .initSocketEventHandlers .socket.on("voice")`	communication-service/test/socketTest.js 6.3.3
FR5.4	The system should alert the user if the other user is in the call	Medium	Not Implemented	Not Implemented

Table 4.1.5.1: Communication service functional requirements

4.1.6 History Service

S/N	Requirement	Priority	Code Reference	Test Reference
FR6.1	The system should keep track of the questions that users have attempted.	Medium	`history-controller .getHistoryByUserId`	history-service/tests/test.js 6.6.1
FR6.2	The system should add questions that users have attempted to a user history.	Medium	`history-controller .createHistory`	history-service/tests/test.js 6.6.3
FR6.3	The system should allow user to keep track of who they collaborated with	Medium	`history-controller .getHistoryById`	history-service/tests/test.js 6.6.2
FR6.4	The system should allow users to see the questions and answers they attempted.	Low	`history-controller .getHistoryById`	history-service/tests/test.js 6.6.2
FR6.5	The system should allow users to get aggregate data on the number of questions of each difficulty they have attempted.	Low	Not Implemented	Not Implemented
FR6.6	The system should allow users to reset their question history.	Low	Not Implemented	Not Implemented

Table 4.1.6.1: History service functional requirements

4.2. Non-functional Requirement

We mainly focus on three categories, out of the 16 categories in non-functional requirements

1. Security
2. Performance
3. Usability

Among the three, we emphasise most on the security of the system. In an application, we believe that, most of the time, security should be included as one of the top non-functional requirements. It is also an indicator toward a reliable system. Users would only use our product with confidence and even recommend our product when the system handles their account particulars appropriately and securely.

Aside from security, we also focus on performance. Having a strong security or risk free system with a slow response, will significantly impact the user experience. Specifically, users will feel frustrated if there are a lot of delays on every click of a button or load of a page and be turned away from using the product. Therefore, we also put a lot of effort into optimising the system to achieve a smooth, real-time like environment.

Last but not the least, the usability of the system. A system with a good performance does not necessarily have a good user experience. We considered from the perspective of our users and built the website to be intuitive and easily usable to people from all backgrounds.

4.2.1 Security

S/N	Requirement	Priority	Code Reference
User Service			
NFR1.1	Users' passwords should be hashed and salted before storing in the DB.	Medium	`user-orm .ormCreateUser`
NFR1.2	The system should not expose any sensitive personal particulars. (e.g. passwords should not be viewable inside the app)	Medium	`User-orm .ormLoginUser`
NFR1.3	User should only be able to delete their own account	Low	`router.delete(DELETE_PATH, verifyJWT, deleteUser)`
NFR1.4	User should only allow to change their password through email verification link	Low	`user-controller .resetPassword`
Communication service			

NFR1.5	The systems should seek the consent of the user when making voice calls.	High	`// The initial state of all clients is not // being in the call.`
			
			`socketController .initSocketController`

NFR1.6	The system should cut off the call once a user leave the room	High	`RoomPage .voiceSocket.disconnect`
History service			

NFR1.7	The system should only allow users to view their own history.	High	`history-controller .getHistoryByUserId`

Table 4.2.1.1: Security requirements

4.2.2 Performance

S/N	Requirement	Priority	Code Reference
Matching service			
NFR2.1	If there is a valid match, the system should match the users within 30s.	High	`// frontend/src/components/HomePage <MatchingDialog initSeconds={30}... />`
Question service			
NFR2.2	System should be able to select one question from questions bank in 3 seconds	Medium	`// Using redis cache decrease time to fetch a // question from approx 191ms to 138ms < 3s checkAndGetFromRedis(`ques?quesId=${quesId}`)`
NFR2.3	If the system has retrieved a question before user should able to retrieve the question within 1 sec in the history page/home page	Low	`question-orm .getOneQuestionById question-orm .getOneQuestionByDifficulty`
Collaboration service			
NFR2.4	The system should ensure that changes by a user on the code is seen by the other user in less than a second.	Medium	`// Use socket.io to communicate changes in the // editor which is 5-7 times faster than http // requests new SocketIOProvider(URL_COLLAB_SVC, roomId, ydoc ...)`
NFR2.5	System should be having a delay of < 2 seconds for concurrent code editing	Medium	

Communication service			

NFR2.6	System should have only a delay of < 2 seconds for voice calls	Medium	`// Each voice packet is recorded at 200`
			`// millisecond intervals`
			`RoomPage.setUpVoiceChat`

Table 4.2.2.1: Performance requirements

4.2.3 Usability

S/N	Requirement	Priority	Code Reference
User service			
NFR3.1	System should allow user to sign up, login and access the website without any prior coding experience	High	Friendly user interface: frontend/src/components/user/LoginPage.js frontend/src/components/user/SingupPage.js
NFR3.2	System should not require the user to login again if their token is still valid.	Medium	frontend/src/util/auth/AuthProvider.js
NFR3.3	System should allow user to reset their password to recover the account	Medium	User-controller.forgetPassword User-controller.resetPassword
Matching service			
NFR3.4	System should notify and start the 30s count down while matching	Medium	`// frontend/src/components/HomePage` `<MatchingDialog initSeconds={30}... />`
Question service			

NFR3.5	System should be able to store at least 100 questions for user to answer	Low	`// Each question is about 521B, 100 // questions will be 5.21KB which is // much lower than the 5GB of max // storage of the MongoDB Atlas // Shared/Free tier mongoose.connect(mongoDB, ...)`
Communication service			
NFR3.6	System should ensure voice calls are heard clearly by both parties.	Medium	`// Each voice packet is recorded at 200 // millisecond intervals RoomPage.setUpVoiceChat`
History service			
NFR3.7	The system should order the question history from most recent to oldest.	Low	`history-service/model/repository.getHistoryModelByUserId()`

Table 4.2.3.1: Usability requirements

5. Design of PeerPressure

5.1. User Service

5.1.1. Summary

User service is responsible for managing user related tasks such as signup, login, delete user account and update user details. Username, email and encrypted password are saved in the user model. User Service will be the first service that users of PeerPressure access. Users are to verify and retrieve their information which will be used in other services.

5.1.2. Design

User service exposes APIs for the client to create and update user information. Each API endpoint has a controller which drives the functionality and interacts with the database via repository pattern.

APIs exposed

REST APIs	Functionality
POST(“api/user/”)	Create user account with the information provided (username, email, password)
POST(“/api/user/login”)	Check user credentials and return Jwt Token as a proof of success login. JwT token is used by the client to verify that it is making a request on behalf of a user.
DELETE(“/api/user/:username”)	Delete user account
POST(“/api/user/forget-password”)	Request to reset the password
PUT(“/api/user/reset-password/:token”)	Reset(update) password

Table 5.1.2.1: User service APIs

5.1.3. Justification (Design Decision)

Method of storing user details

Design	Pros	Cons
Cookies	- Data stored is persisted within the indicated duration.	- As cookies are stored in a hard drive as a text file there is a security risk where hackers hack to read the user's hard drive.

Local-storage	- Do not require any backend logic to store the data. - Fast to access to the data stored	- Data stored is not persistent would disappear once refresh - Do not provide data protection
Redux	- High maintainability where the structure of redux is easy to be understand	- In-memory state store where data/state would disappear once application crashes or refreshes. - Increase complexity and introduce overhead if the application does not need to keep track of state frequently.
After comparison, we decide to prioritise the usability of the application. Using cookies allows the user to access the website easily as long as the cookie is not expired. Besides, the information stored in the cookies will be encrypted using jwt token with only unique username and email. Hence, the security risk is minimal.		

Table 5.1.3.1: User service design decision

Method of Authenticating Users

Design	Pros	Cons
JWT	- Easy to scale since requests with JWT can be processed by any server. - No clean up required on the server side.	- Takes up some bandwidth due to the size of information in the token. - Difficult to revoke authentication access.
Sessions	- Take up little bandwidth - Easy to revoke access.	- Harder to scale since the server must always verify the session ID with a session store. - Sessions for users need to be maintained, often in memory which creates overhead on the server.
JWT is chosen because it is important for us to ensure the application is scalable. Information kept in the token is minimal, so we do not expect bandwidth to be a major constraint. Authentication access, while difficult to revoke is not impossible.		

Table 5.1.3.2: User service authentication decision

5.2. Matching Service

5.2.1. Summary

The matching service is responsible for enabling users to select a question difficulty level and match with another user who is keen on the same difficulty level and is also looking for a match. A user has 3 states:

1. Matching: user has selected difficulty level and is waiting to find a match
2. Match success: two users have matched and will move on to the coding room
3. Match failed: user has selected difficulty level and did not find a match within 30s

The match model has user id, difficulty level and the user id of the successfully matched user.

5.2.2. Design

Matching service exposes APIs for the client to create match information. Each API endpoint has a controller which drives the functionality and interacts with the database via repository pattern.

APIs exposed

REST APIs	Functionality
POST("/difficulties")	Create a match model with userId, matchedUser, difficulty level

Table 5.2.2.2: Matching service APIs

We use socket events for communication between the client and server. During matching, the client's socket instances do not know about each other and only directly communicate with the server's socket instance and rely on it to correctly route data to clients. This is a utilisation of the mediator pattern where the matching service server acts as the mediator between all the users in the matching phase.

Zooming into a match success scenario, when a user selects a difficulty level and starts the matching process, a find_match socket event is emitted to the server. If a match is found on the server, match_success socket events are emitted to the matched users. (Figure 5.2.2.1)

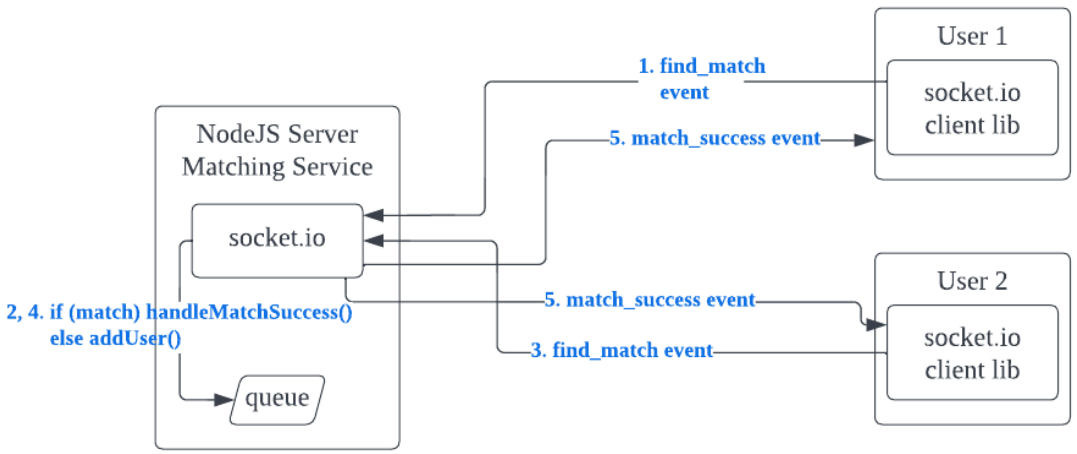

Figure 5.2.2.1 Match success scenario

5.2.3. Justification (Design Decisions)

Communication between client and server

Design	Pros	Cons
Socket.io	- Supports real time and fast communication - important since matching has a time constraint - Supports bi-directional communication which allows the server to notify clients on an event rather than having the client repeatedly poll the server for that event	- Requires cleaning up the sockets after use
HTTP Requests	- Stateless requests do not require clean up after use	- Slower because more resource and process intensive which can negatively impact the time sensitive matching of users - Unidirectional, clients have to repeatedly poll the server to check status which wastes resources (time, CPU)
Socket.io was selected due to its speed and bidirectionality allowing matching to be most efficient. The cons of having to clean up sockets after use is an additional but small implementation cost.		

Table 5.2.3.1: Matching service design decision

Module responsible for fetching the question object from question service and creating the history object using history service for use in the coding room.

Design	Pros	Cons
Matching-service	- To synchronise question id and history id for all users in a room	- Higher coupling across matching, question and history services
Frontend	- Less coupling between services	- More back and forth communication needed to synchronise the question number across users in the room. This results in higher latency in displaying the question for all users in the room

Matching service was chosen because it is more efficient and we prioritise user experience over slightly higher coupling.

Table 5.2.3.2: Matching service coupling

Choice of database

Design	Pros	Cons
MongoDB Atlas	- DB can be in the cloud	
SQLite		- Not recommended to push DB to the cloud because it is serverless and cannot do simultaneous concurrent write
MongoDB Atlas was selected because the cloud support aids the development process and reduces the complexity of deployment.		

Table 5.2.3.3: Matching service database decision

5.3. Collaboration Service

5.3.1. Summary

The collaboration service is responsible for maintaining the shared, real-time code editor. It enables users in the room to read and write a shared text concurrently. The collaboration model has the user ids of the two users in the room, room id, difficulty level and the shared text.

5.3.2. Design

Collaboration service exposes APIs for the client to create the collaboration information. Each API endpoint has a controller which drives the functionality and interacts with the database via repository pattern.

APIs exposed

REST APIs	Functionality
POST(“/collab”)	Creates a collab object with the userId of the two users matched, room id, difficulty level and shared text
GET(“/collab/:roomId”)	Fetches the collab object with the specified roomId
DELETE(“/collab/:roomId”)	Deletes the collab object with the specified roomId
PUT(“/collab”)	Updates the collab object by specifying the roomId and the updated values of fields in the request body

Table 5.3.2.2: Collaboration service APIs

We use socket events for communication between the client and server. During concurrent editing , the users do not know about each other and only directly communicate with the server's socket instance and rely on it to correctly route data to other user in the room. This is a utilisation of the mediator pattern where the collaboration service server acts as the mediator between all the users in the coding room.

Clarification of terms

Term	Meaning
ydoc	Data structure that holds the shared data and automatically sync and persist their state using providers
WebSocketProvider	Communication protocol that connects to the server's socket instance, clients (room id) and ydoc to enable clients to sync to each other
Quill editor	Code editor in the coding room implemented via Quill

Table 5.3.2.1: Collaboration service terminology

When one user in the room changes the shared text in the code editor, their ydoc is updated which automatically triggers a synchronisation with the ydoc of other clients with the same room id. ydoc, from the Yjs framework, is a high performance conflict resolution data type (CRDT). It features globally unique characters and globally ordered characters in order to support consistent document state across clients for all sequences of operations. As a result, users in the room will always be looking at the same shared text in the code editor.

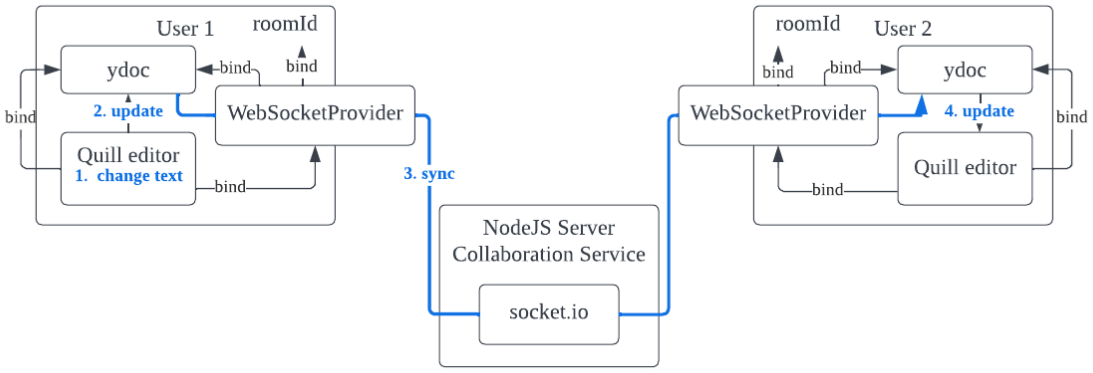

Figure 5.3.2.1 Scenario of a text change event propagated to the other user in the room

5.3.3. Justification (Design Decisions)

Implementing the code editor

Design	Pros	Cons
Quill + Socket.io	- Supports non-concurrent editing	- No guarantee that order of socket events are retained which could lead to jumbled up characters for users - There are complex edge cases which cause the document to not converge to the same state
Quill + Yjs framework + Socket.io	- Supports concurrent editing	
Yjs a CRDT framework supports concurrent editing which is crucial in a real-time collaborative editor.		

Table 5.3.3.1: Collaboration service code editor decision

Share socket instance across matching and collaboration service

Design	Pros	Cons
Share	- Easier to implement	- Difficult to handle and recover from disconnection events
Not share	- Reduce coupling across microservices - Better handling of and recovering from disconnection events	- More complex to implement
We set up independent socket instances for each service in order to reduce coupling across services. Although it is more complex to implement in the short run, it improves maintainability for the long run.		

Table 5.3.3.2: Collaboration service socket sharing decision

Communication between client and server

Design	Pros	Cons
Socket.io	- Supports real time and fast communication - a must since the shared code editor must be updated at the same time to function as a collaboration editor - Supports bi-directional communication which allows the server to send events to clients rather than having the client repeatedly poll the server for that event	- Requires cleaning up the sockets after use
HTTP Requests	- Stateless requests do not require clean up after use	- Slower because more resource and process intensive which can negatively impact the time sensitive matching of users - Unidirectional, clients have to repeatedly poll the server to check status which wastes resources (time, CPU)
Socket.io was selected due to its speed and bidirectionality allowing collaboration to be most efficient. The cons of having to clean up sockets after use is an additional but small implementation cost.		

Table 5.3.3.2: Collaboration service client communication decision

5.4. Communication Service

5.4.1. Summary

The communication service is responsible for allowing users in the same room to voice chat with each other. It keeps track of the online status of both users to decide on whether to transfer voices to the other user.

5.4.2. Design

Voice calls are recorded as packets of sound information which are periodically sent to the server. In this service, we utilise the pub-sub pattern to deliver voice packets from a speaker to the other party in the room. The client may subscribe to voice packets in a room by sending a userInfo message across socket io. The service will subscribe the client to a topic with the given room ID provided in the message. Whenever a user is speaking, voice packets are published to the topic with the room ID and communication service forwards it to any other user subscribed to voice packets coming from that topic. While the use case for this app implies that voice packets are sent to at most one other user, the service supports the use case where three or more users are in the same room.

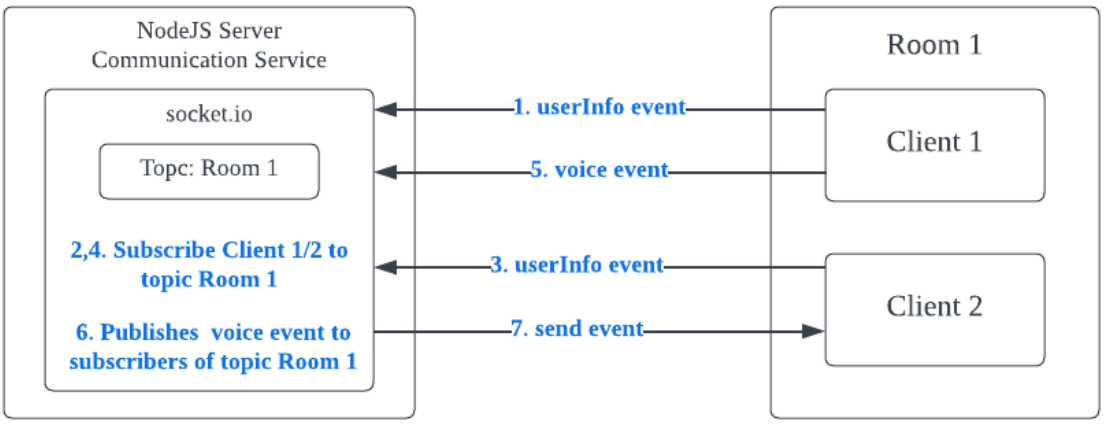

Figure 5.4.2.1: Communication service voice call scenario

5.4.3. Justification (Design Decisions)

Communication between client and server

Design	Pros	Cons
WebSockets	- Supports real time and fast communication - Supports a pub-sub pattern which allows users to subscribe to events	- Requires cleaning up the sockets after use
HTTP Requests	- Stateless requests do not require clean up after use	- Slower because more resource and process intensive which can negatively impact the time sensitive communication between users - Requires users to constantly make requests to fetch voice packets from the server
In voice calls, it is important that users are able to hear each other speak in real-time. WebSockets presents a lesser load on clients since clients do not need to proactively request for voice packets.		

Table 5.4.3.1: Communication service design decision

5.5. Question Service

5.5.1. Summary

The question service is responsible for maintaining a question bank which consists of a list of coding questions of easy, medium or hard difficulty. The question model has the title, body, and difficulty of the coding question and url where the question was fetched from.

5.5.2. Design

Question service exposes APIs for the client to create the question information. Each API endpoint has a controller which drives the functionality and interacts with the database via repository pattern.

APIs exposed

REST APIs	Functionality
POST(“/ques”)	Create a question by specifying title, body, difficulty and url in the request body
GET(“/ques/diff?diff={difficulty}”)	Retrieve a question randomly based on input difficulty.
GET(“/ques/id?id={questionId}”)	Retrieve a question based on input question id.

Table 5.5.2.1: Question service APIs

The questions in the question bank are obtained from LeetCode and inserted into the database using the create question API.

When there is a match found between two users, the matching service will fetch a question based on the difficulty both users have selected. This API pipes questions from the data source in the question collection, and the first filter applied is based on the difficulty level and the second is randomly sampling 1 question to output as illustrated in Figure 5.5.2.1.

```
async function randSelectQuestionId(difficulty) {
    const quesId = await QuestionModel.aggregate()
        .match({ difficulty: difficulty })
        .sample(1) // randomly select
        .then((res) => {
            return res[0]._id
        })
        .catch((err) => {
            console.log(err);
        })
    return quesId;
}
```

Figure 5.5.2.1: Relevant code block that filter the questions from the database

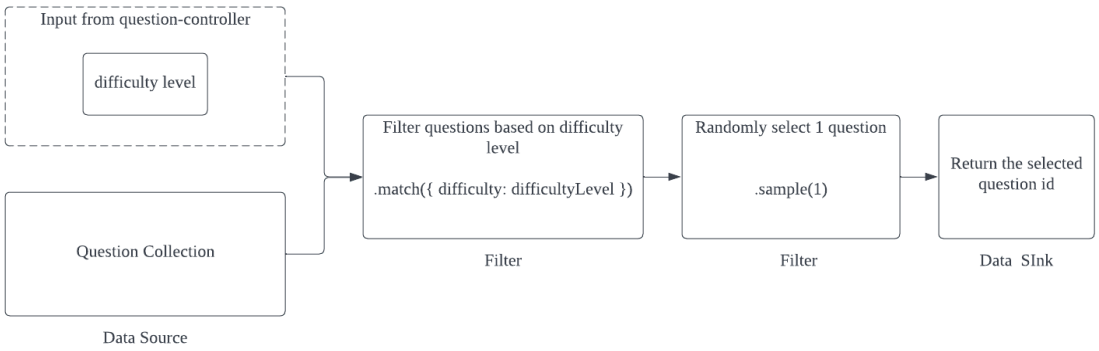

Figure 5.5.2.1: Illustrating how question is retrieved based on difficulty level

5.5.3. Justification (Design Decisions)

Design	Pros	Cons
Question service as a microservice	- Low coupling - Easier for work collaboration	- More time required to develop
Matching service to handle question functionality	- Less time required to develop	- Huge coupling - Difficult for work collaboration
We decided to go with question service as a microservice because it decreases coupling between microservices like matching service and history service. Even though more time is required for development, we believe the productivity will be higher when the system becomes complex.		

Table 5.5.3.1: Question service design decision

Design	Pros	Cons
Use Redis	- Decrease data access latency resulting in less wait times for the user - Simple to set up	- Overhead of having to connect to Redis cluster - In-memory database hence requires more RAM
Without Redis (pure Mongodb)	- Simpler to implement	- Longer wait times to fetch the question since it has to be fetched from the database
We decided to use Redis as it reduces the wait times for the user. To show the history, it needs to interact with the question service API frequently. A good user experience is important to retain users. The cons of additional overhead to connect to the Redis cluster is a one time off overhead, but every subsequent get question request will have the reduced latency. Additionally, the RAM usage will only be used to store a constant number of questions and will not scale with number of users or number of collaborations, hence is not greatly memory intensive.		

Table 5.5.3.2: Question service caching decision

5.6. History service

5.6.1. Summary

The history service is responsible for maintaining the data of all past collaborations of a user. The history model stores the question id, user ids of the users in the room and the shared text of the past collaboration.

5.6.2. Design

Question service exposes APIs for the client to create the question information. Each API endpoint has a controller which drives the functionality and interacts with the database via repository pattern.

APIs exposed

REST APIs	Functionality
POST(“/hist”)	Create a new history by specifying two userIds and a questionId in the request body. Answer field is not required.
GET(“/hist/id?id={historyId}”)	Fetch a history based on the historyId.
GET(“/hist/userId?userId={userId}”)	Fetch all the histories belonging to the userId. The returned histories will have either userId1 or userId2 equal to the specified userId. They are ordered descendingly by the updatedAt field.
PUT(“/hist”)	Update the answer of a history by specifying the id as the historyId and answer in the request body.

Table 5.6.2.1: History service APIs

In the database, each history records the userId1, userId2, questionId, answer, createdAt and updatedAt. userId1, userId2, questionId and answer are strings, while createdAt and updatedAt are dates which are auto computed by MongoDB. Only the userIds and questionId are required when creating the row.

5.6.3. Justification (Design Decisions)

Design	Pros	Cons
History service as a microservice	- Low coupling - Easier for work collaboration	- More time required to develop
Collaboration service to handle the functionality	- Less time required to develop	- Huge coupling - Difficult for work collaboration
We decided to create the history service as a microservice since this decision will reduce the responsibilities for collaboration service and thus reduces coupling. Even though more time is required for development, we think that productivity will increase as the system becomes complex.		

Table 5.6.3.1: History service design decision

6. Other Requirements

Appendix A: Test cases

1. User service
    - 1.1. POST: create user with valid input
    - 1.2. POST: create user with repeated email
    - 1.3. POST: create user with repeated password
    - 1.4. POST: should not create user if username already exist
    - 1.5. POST: should not create user if missing fields
    - 1.6. POST: should not create user if invalid email format
    - 1.7. POST: should not login user that does not exist
    - 1.8. POST: should not login user with wrong password
    - 1.9. POST: should not login user with missing field
    - 1.10. POST: login user with valid input
    - 1.11. DELETE: delete user with valid authentication
    - 1.12. DELETE: should not delete user that does not exist
    - 1.13. DELETE: should not delete with no authentication
    - 1.14. DELETE: should not delete with invalid authentication
    - 1.15. POST: send reset link with valid input
    - 1.16. POST: should not send reset link to user that does not exist
    - 1.17. POST: should not send link with missing field
    - 1.18. PUT: reset password with valid link and input
    - 1.19. PUT: should not reset password with invalid link
    - 1.20. PUT should not reset password with invalid input
    - 1.21. PUT: should not reset password with missing field
2. Collaboration service
    - 2.1 POST: collab with valid input
    - 2.2 POST: collab with repeated room ID
    - 2.3 POST: collab with missing parameters
    - 2.4 GET: existing collab
    - 2.5 GET: collab not in database
    - 2.6 DELETE: collab by room ID
    - 2.7 DELETE: collab not in database
    - 2.8 PUT: update collab with new user, difficulty, text values
    - 2.9 PUT: update collab not in database
    - 2.10 PUT: update collab without room ID
3. Communication service
    - 3.1 SOCKET: client joins call
    - 3.2 SOCKET: client leaves call
    - 3.3 SOCKET: client joins a room without room ID
    - 3.4 SOCKET: client 1 sends voice heard only by client in same room
4. Matching service
    - 4.1. POST: create matching with easy
    - 4.2. POST: create matching with medium
    - 4.3. POST: create matching with hard
    - 4.4. SOCKET: match
    - 4.5. SOCKET: no match

5. Question service
    5.1. GET: randomly select question
    5.2. POST: create new question
6. History service
    6.1. GET: history by userId
    6.2. GET: history by Id
    6.3. POST: create new history

Appendix B: Glossary

Term	**Description**
System	This includes both the front-end for the user interface and the back-end for the logic, which is hidden from the user of PeerPressure.
Database	An organised collection of structured information, or data, consisting of the user's data and a list of questions.
Difficulty	The difficulties of questions that user could select (Easy, Medium, Hard)
Matching	The process of matching two users who have selected the same difficulty level to work on.
User	Computer science students and anyone who uses PeerPressure
Collaborative editor	The text field that has the real-times updates for both users when one user makes a change on the editor while approaching the question.