CS3219 Group 38

Software Engineering Principles and Patterns

Project Report: PeerPrep

Members:
Keane Chan Jun Yu (A0205678W)
Lau Jun Hao Ashley (A0204888R)
Oh Jun Ming (A0197781L)
Lim Jin Hao (A0205878R)

Code Repository Link
https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g38

Tables of Content

1. Contributions

Member	Technical Contributions	Non-Technical contribution
Lau Jun Hao Ashley	- Set up backbone for frontend together with Jin Hao - Implement chat micro service	- Requirements documentation - Final Report - Frontend mockup design - Final Presentation
Lim Jin Hao	- Implement account microservice - Set up backbone for frontend together with Ashley - Set up AWS integration and deployment	- Requirements documentation - Final Report - Final Presentation - README
Keane Chan Jun Yu	- Implement match microservice - Implement match frontend (Polling, UI)	- Requirements documentation - Final Report - Final Presentation
Oh Jun Ming	-Implement interview microservice (coding platform service)	- Requirements documentation - Final Report - Final Presentation

2. Introduction

Increasingly, students face challenging whiteboard style interviews when applying for jobs which many have difficulty dealing with. Issues range from a lack of communication skills to articulate their thought process out loud to an outright inability to understand and solve the given problem. Moreover, grinding practice questions can be tedious and monotonous. To solve this issue, we have implemented a collaborative interview preparation platform called PeerPrep where students can find peers to practice whiteboard style interview questions together.

2.1 Purpose

The purpose is to create a web application that helps students better prepare themselves for technical whiteboard interviews. We aim to achieve this by using a peer learning system such as the one proposed above so students can learn from each other and break the monotony of revising alone.

2.2 Target users

PeerPrep aims to help students who are practicing for their whiteboard-style technical interviews.

2.3 Project scope

To start out the project, we establish certain goals that PeerPrep aims to achieve.

1. Students should be able to create an account and login to PeerPrep
2. Students should be able to choose the difficulty level of the question and the programming language
3. Students should be able to be matched with other students which selected the same difficulty and programming language
4. Our platform will randomly select a question of that difficulty level from Leetcode
5. Students should be able to chat with his match
6. PeerPrep should provide a collaborative whiteboard-style programming for the two students to code together in real-time
7. Students should be able to see their time taken for the current question in the interview session
8. Students should be able view a history of their past interview sessions in the home page

3. Functional Requirements

3.1 Account Management

ID	Functional Requirement	Priority
AS-FR-1	The account management microservice should allow users to register a new account using an email address and a password	High
AS-FR-2	The account management microservice should allow users to sign in into the application with their registered email and password	High
AS-FR-3	The account management microservice should allow the users to logout of their account	High
AS-FR-4	The account management microservice should store a list of interviews completed by the user. The interview should contain the following details: ● Leetcode problem name (e.g shortest n path) ● Time taken ● Interview partner ● Submission status (completed or forfeited)	Medium
AS-FR-5	The account management microservice should allow the user to view their list of completed interviews and its respective details (as discussed in AS-FR-4)	Medium
AS-FR-6	The account management microservice should allow the user to reset their password should they forget their account password	Medium
AS-FR-7	The account management microservice should verify the user's email on account registration.	Low
AS-FR-8	The account management microservice should support persistent user login across browser sessions	Low

3.2 Matchmaking Management

ID	Functional Requirement	Priority
MS-FR-1	The matchmaking management should allow users to select the difficulty of the question (easy, medium, hard)	High
MS-FR-2	The matchmaking management should allow users to select their desired programming language (Java, Python, Javascript)	High
MS-FR-3	The matchmaking management should match users with the same selected difficulty and programming language	High
MS-FR-4	The matchmaking management should timeout the user after **30 seconds** if there is no matching user with the same selected difficulty and language	High
MS-FR-5	The matchmaking management should allow users to stop matching	Medium
MS-FR-6	The matchmaking management should inform the user if there is no matching user	Low

3.3 Interview Session Management

ID	Functional Requirement	Priority
ISM-FR-1	The interview session management should generate a random Leetcode question based on chosen difficulty	High
ISM-FR-2	The interview session management should provide users with a shared text editor for the 2 users to construct their solution	High
ISM-FR-3	The interview session management should provide users with a chat platform for them to discuss their solutions and approach	High
ISM-FR-4	The interview session management should allow users to complete the interview session	High
ISM-FR-5	The interview session management should allow users to forfeit the interview session	High
ISM-FR-6	The interview session management should provide an interview summary at the end of the session, which includes the following information: ● Question Name ● Difficulty ● Language ● Time taken for the interview	Medium
ISM-FR-7	The interview session management should display a timer for the entire duration of the session	Low
ISM-FR-8	The interview session management should notify the user when the other party has forfeited the session	Low
ISM-FR-10	The interview session management should allow users to copy their code upon successful completion of the interview	Low

3.4 Application Client Management

ID	Functional Requirement	Priority
AC-FR-1	The application client management should provide a landing page for the users to login / create an account	High
AC-FR-2	The application client management should provide a user interface for the users to start and hold the interview session	High
AC-FR-3	The application client management should provide a user interface for users to view a list of his past interview sessions	Medium
AC-FR-4	The application client management should allow users to sign in as a guest user	Low

4. Non-functional Requirements

NFR in terms of priority:

1. Security
2. Performance
3. Availability
4. Scalability

We choose to prioritize security of the application as we believe a robust system that protects the information our end-users is required to form the foundation of our application. Next, we prioritized performance as we believe in providing existing users with the best end to end experience (from creating an account to starting a session to ending a session) to leave a long lasting impression on them. We prioritize availability next as we want our application to be available for our end users even when our usage is high. Lastly, as our application is in its infancy phase, we do not foresee a large amount of traffic, thus we placed scalability as the last of our priorities.

4.1 Security

Non-Functional Requirements

ID	Non-Functional Requirement	Priority
SEC-NFR-1	The system should prevent access from unauthorized users into an interview session	High
SEC-NFR-2	The database should securely store encrypted salted-hash of user passwords	High
SEC-NFR-3	The system should prevent users from accessing any unauthorized user data	High
SEC-NFR-4	All databases should only allow its corresponding microservice to access it	High

Mozilla Observatory Test Scores

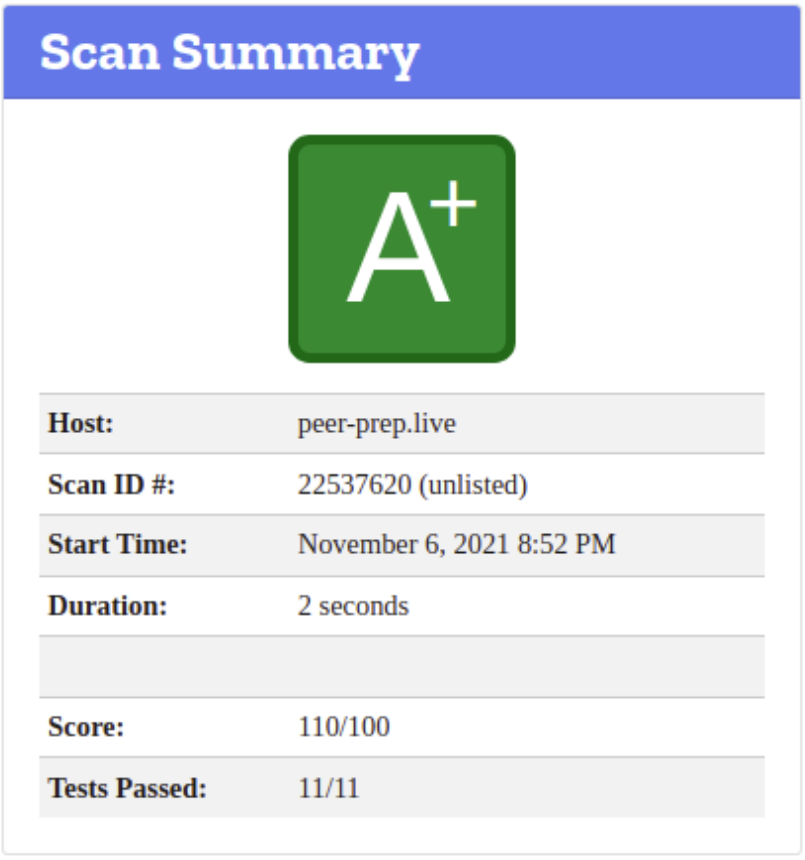
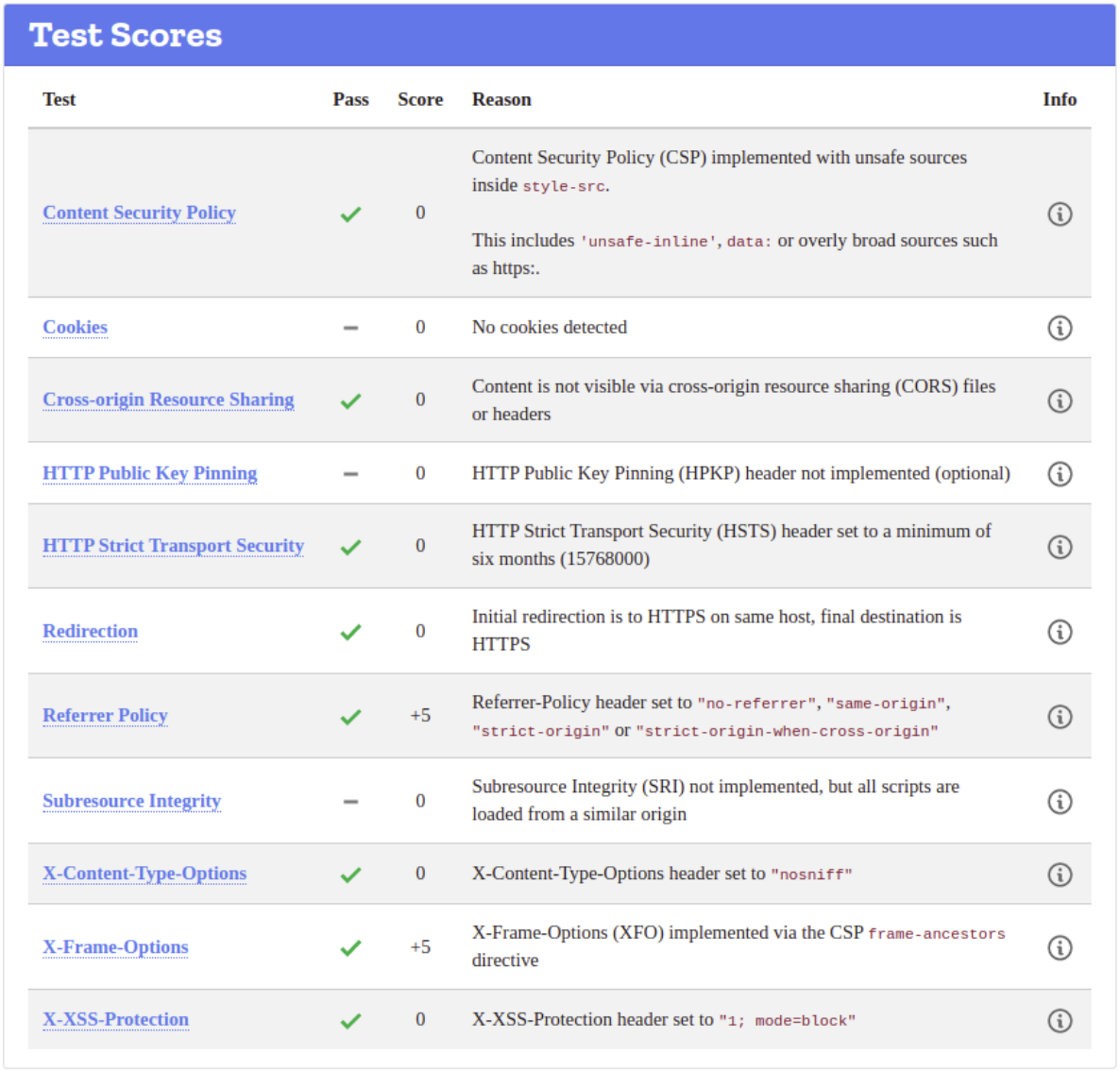


Fig 4.1.1. Mozilla Observatory Scan Summary



Test Scores

Test	Pass	Score	Reason	Info
Content Security Policy	✓	0	Content Security Policy (CSP) implemented with unsafe sources inside `style-src`. This includes `'unsafe-inline'`, `data:` or overly broad sources such as https:.	
Cookies	–	0	No cookies detected	
Cross-origin Resource Sharing	✓	0	Content is not visible via cross-origin resource sharing (CORS) files or headers	
HTTP Public Key Pinning	–	0	HTTP Public Key Pinning (HPKP) header not implemented (optional)	
HTTP Strict Transport Security	✓	0	HTTP Strict Transport Security (HSTS) header set to a minimum of six months (15768000)	
Redirection	✓	0	Initial redirection is to HTTPS on same host, final destination is HTTPS	
Referrer Policy	✓	+5	Referrer-Policy header set to `"no-referrer"`, `"same-origin"`, `"strict-origin"` or `"strict-origin-when-cross-origin"`	
Subresource Integrity	–	0	Subresource Integrity (SRI) not implemented, but all scripts are loaded from a similar origin	
X-Content-Type-Options	✓	0	X-Content-Type-Options header set to `"nosniff"`	
X-Frame-Options	✓	+5	X-Frame-Options (XFO) implemented via the CSP `frame-ancestors` directive	
X-XSS-Protection	✓	0	X-XSS-Protection header set to `"1; mode=block"`	

Fig 4.1.2. Mozilla Observatory Test Scores

Security implementations

Potential Threat	**Implemented strategy**	**CIA triad (Tags)**
SQL Injection/ Unsanitized user input	Frontend: - Form validation using Yup (Fig 4.1.1) Backend: - Automatic validation of all incoming requests using class-validator (relevant NestJS doc) - Use of TypeORM's API and QueryBuilder which has in-built protection against SQL injection	Confidentiality, Integrity

Distributed Denial of service (**DDOS**)	Use of Cloudfront (a content delivery network service) for edge caching - Provides AWS Shield Standard to defend against DDoS attacks (reference) - Edge caching avoids making multiple calls to S3.	Availability
MITM attacks	Implement HTTPS best practices: - Obtain certificate from reliable CA (ACM) - Enforce HTTP to HTTPS redirection - Use of secure URL in the codebase - Support HSTS (refer to above test scores)	Confidentiality, Integrity
XSS & **Clickjacking**	Use of security headers to protect against well-known web vulnerabilities. (Fig 4.1.2) - Content-Security-Policy - X-Frame-Options - X-XSS-Protection	Confidentiality, Integrity
Unauthorized access to **backend services** (eg. database)	Access to RDS and Elasticache Redis is restricted by Security Groups. Hence it is only accessible within the cluster.	Confidentiality, Integrity
Hijacking of **interview session**	Use of uuid helps produce a unique and less-predictable id for each interview session. Checking for details provided during the matching phase serves as an additional layer of protection to ensure that the user has gone through the matching process and is allowed to access the session	Confidentiality
Unauthorized access to **user account**	Application-level: - Enforce strong password usage to prevent malicious actors from getting access - Use of JWT access token to authenticate and validate user for all protected endpoints (in Account MS) Database-level: - Salted hash of all user passwords in the database using bcrypt (Fig 4.1.3) - Separation of authentication data from profile data to prevent leaking of user's personally identifiable information (email, pasword).	Confidentiality, Integrity

Screenshots of security implementations:

```
const validationSchema = yup.object({
  // checks according to OWASP password policy
  password: yup
    .string('Enter your new password')
    .min(8, 'Password should be of minimum 8 characters length')
    .matches(
      /^(?=.*[~`!@#$%^&*+=\-\[\]\\';,/{}|":<>?()._])/,
      'Password should contain at least one special character'
    )
    .matches(/^(?=.*[0-9])/, 'Password should contain at least one number')
    .matches(
      /^(?=.*[a-z])/,
      'Password should contain at least one lowercase letter'
    )
    .matches(
      /^(?=.*[A-Z])/,
      'Password should contain at least one uppercase letter'
    )
    .required('Password is required'),
  confirm: yup
    .string('Confirm your new password')
    .oneOf([yup.ref('password'), null], 'Password does not match'),
});
```

Fig 4.1.1 Yup validation schema, following OWASP password policy

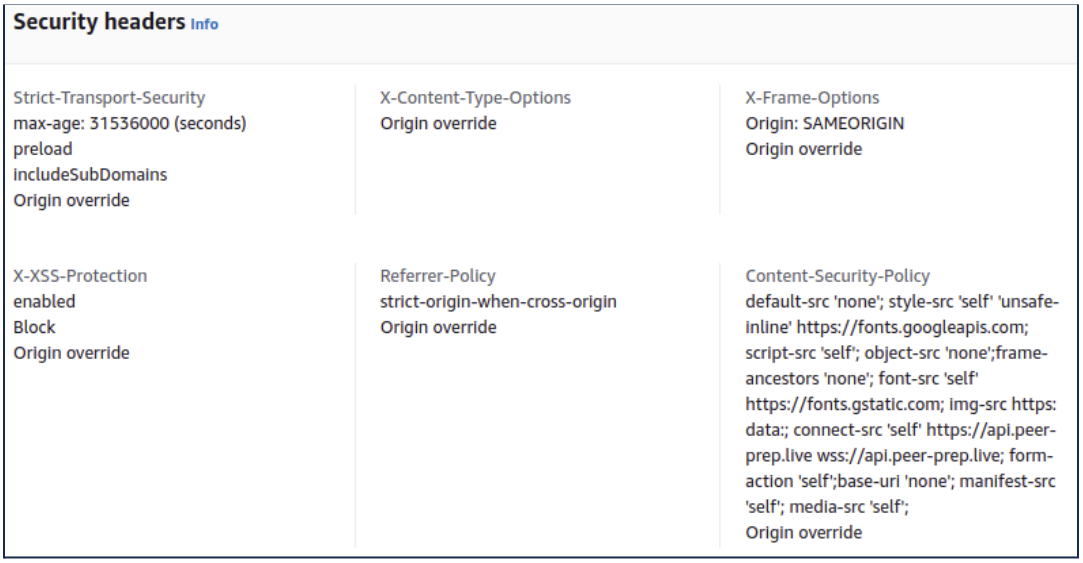


Fig 4.1.2 Implemented security headers in CloudFront

```
public async signup(createUserDto: CreateUserDto): Promise<User> {
  if (await this.usersService.doesEmailExist(createUserDto.email)) {
    throw new ConflictException('Email already exists');
  }

  const { password, email, name } = createUserDto;
  const passwordHash = await bcrypt.hash(password, 12);
  return this.usersService.createUser({
    email,
    name,
    passwordHash,
  });
}
```

Fig 4.1.3 Use of bcrypt to hash user's password

4.2 Performance

Non-Functional Requirements

Preconditions:

1. User has a stable internet connection

ID	Non-Functional Requirement	Priority
PER-NFR-1	The application should successfully load the interview questions within 5 seconds upon matchmaking	High
PER-NFR-2	The application should successfully match the users within 5 seconds if there is a valid matching pair and available room	High
PER-NFR-3	During live coding, the response delay should be at most 2 seconds	High

Strategies

The following are strategies to achieve the performance NFR listed above.

Strategy	Implementation	How it helps
Fallback Leetcode Qns	- Created a backup of 3 easy, 3 medium and 3 hard questions to serve as fallback. - Timeout if the LeetCode API fails to return a valid question within **3 seconds** and randomly selects a fallback question	- Dependency on external API for retrieving Leetcode interview questions means that there may be unexpected downtime or latency issues. - Fallback questions ensure that response time for retrieving leetcode questions falls within 5 seconds
Live coding	- Use of CodeMirror's extension to push small Operational Transformation (OT) changes through socket.io - Eg. {index, text} for an insert op Refer to section on interview microservice for detailed implementation details	- Pushing small OT changes instead of full code reduces the number of bytes sent through the socket - Reduces the latency and hence the live coding response delay
Cloudfront Edge Caching	- Use of Cloudfront service to serve static content worldwide	- Cloudfront caches file at edge locations - Reduces the need for users to retrieve data from S3 server as it is available in a closer location - Faster loading time
Redis caching	- Use of Redis to cache interview session content - Use of Redis to cache matching details (Support for efficient polling of match status)	- Caching is used for data that need to be frequently read/write (matchmaking queue & code for each session) - Reduces read/write time as compared to a traditional database - Persistent storage

4.3 Availability

Non-Functional Requirements

ID	Non-Functional Requirement	Priority
AVL-NFR-1	The application should maintain an uptime of 99% during peak hours	High

Availability Overview

Service/ Node	Redundancy	Failure detection	Failover	Replication
Frontend	Edge caching	-	-	-
Kubernetes worker nodes (Node group)	Cluster Autoscaling across multi-AZ	Kubelet	✅	Managed by cluster autoscaler
Account Microservice	Horizontal Pod Autoscaling	Ingress Gateway	✅	Managed by HPA
Match Microservice	Horizontal Pod Autoscaling	Ingress Gateway	✅	Managed by HPA
Interview Microservice	Horizontal Pod Autoscaling	Ingress Gateway	✅	Managed by HPA
Chat Microservice	Horizontal Pod Autoscaling	Ingress Gateway	✅	Managed by HPA
ElastiCache Redis Cache	Multi-AZ	Cluster-Master Node	✅	3 nodes in different AZ
RDS Postgres Database	Multi-AZ	RDS	✅	Managed by RDS

4.4 Scalability

ID	Non-Functional Requirement	Priority
SCA-NFR-1	The application should host 5 concurrent interviews at the same time	High
SCA-NFR-2	The application should support 50 users using the application simultaneously	Medium
SCA-NFR-3	The database should handle information storage of 100 users	Medium

Scalability Overview

Application Load Balancer (ALB), Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler (CA) work together to scale the application when workload increase:

- When incoming traffic to a particular service pod increases, HPA detects the increased load and increases the number of pods for that service
- If there is a lack of worker nodes to manage the increased pods, CA automatically adjusts the number of nodes in the cluster to handle the workload.
- ALB brings it together by distributing incoming traffic to the various pods and balances the load for each pod.

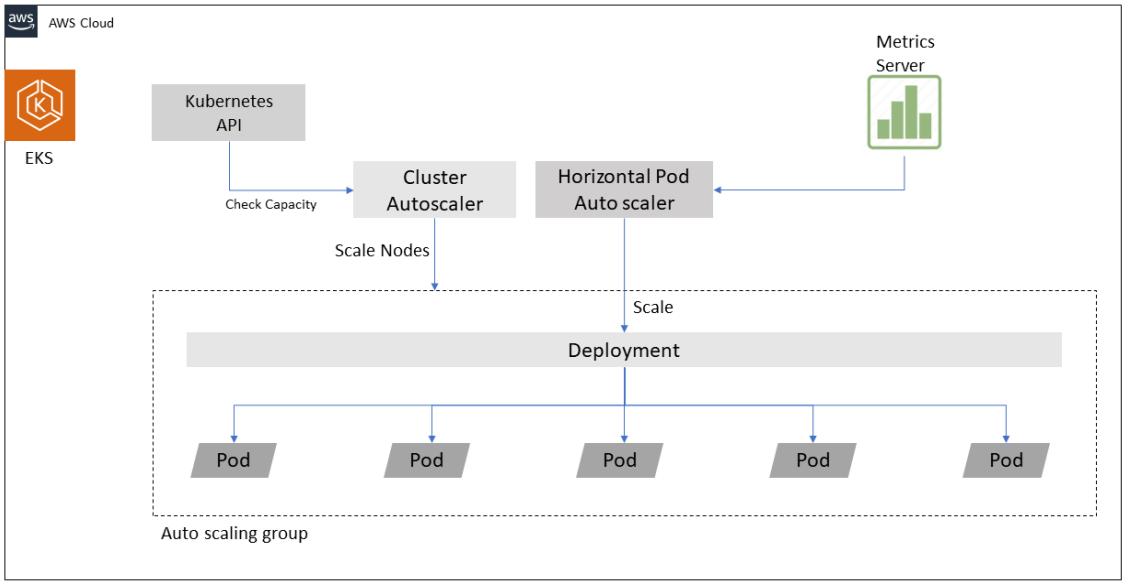


Fig 4.4.1. HPA and CA Architecture[1]

[1] Image credit to https://medium.com/tensult/cluster-autoscaler-ca-and-horizontal-pod-autoscaler-hpa-on-kubernetes-f25ba7fd00b9

5. Architecture

5.1 Architecture design

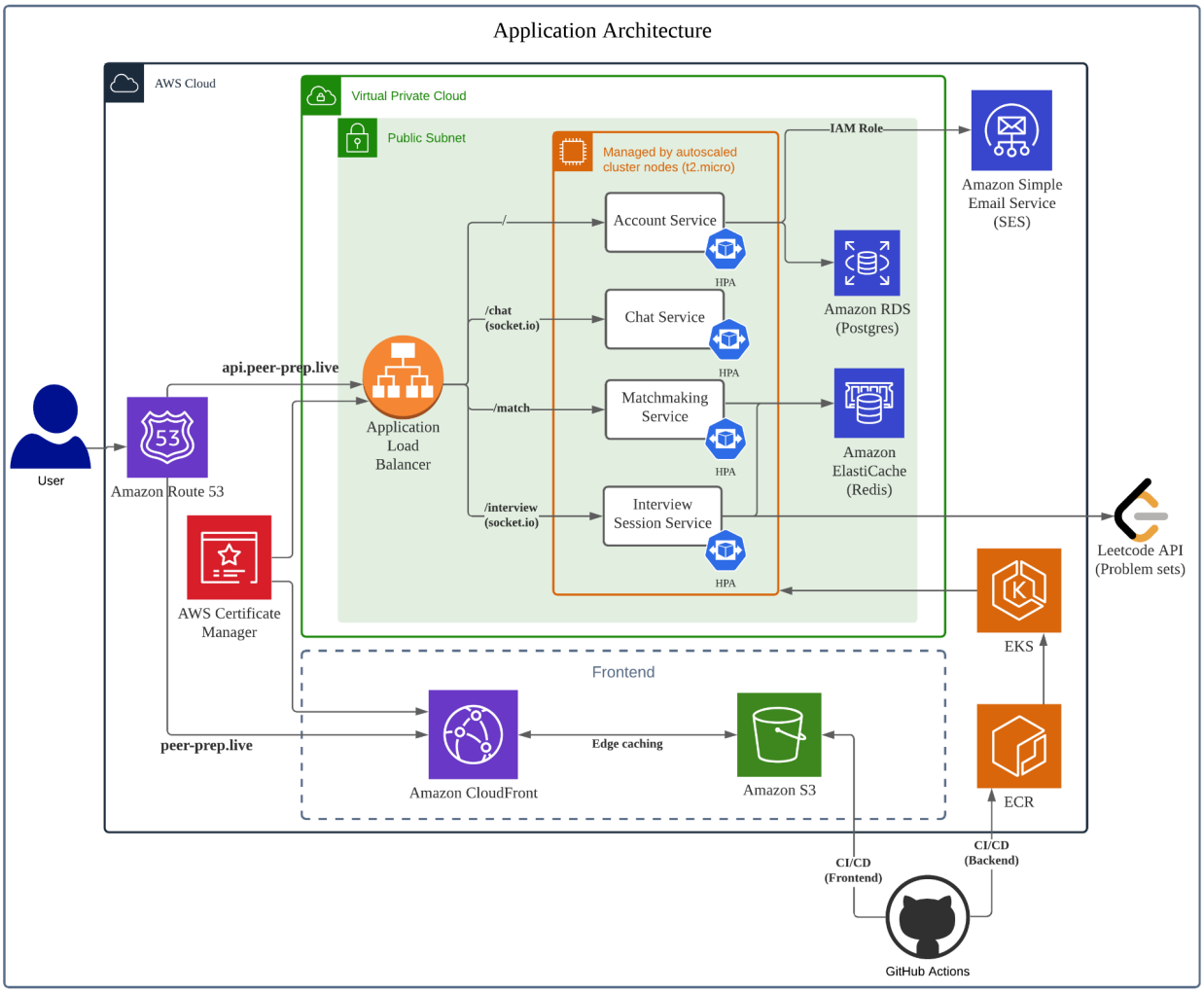


Fig 5.1.1. Architecture Diagram

1. For frontend, we host our static webpage on Amazon S3 and use Cloudfront as our Content Delivery Network (CDN) for its improved load performance and security feature (S3 is only accessible through Cloudfront).
2. Route 53 hosted zone gets inbound traffic from end user and routes api.peer-prep.live requests to ALB and peer-prep.live requests to Cloudfront.
3. Certificates provided by AWS Certificate Manager are used to enable secure HTTP.
4. When a request is made to the backend, it will be directed to the Application Load Balancer (ALB) in the Virtual Private Cloud (VPC).
5. The ALB uses Ingress to direct the API call to the relevant microservices.

6. Horizontal Pod Autoscaler scales the services horizontally by automatically managing the number of replicas for each service to prevent overloading of a particular service during peak usage.
7. Our data layer consists of PostgreSQL and Redis using Amazon Relational Database Service and Elasticache respectively.
    a. The data layer are only accessible within the cluster and by services in the same security group.
8. Amazon SES is used to provide mail functionality to allow for email validation and password reset features.
9. Interview Session microservice makes external requests to the Leetcode API to retrieve data on problem sets.
10. For chat and interview session microservice, socket.io is used for pub/sub functionalities.
11. Github Actions push changes to S3 for the frontend and updates to Elastic Cloud Registry (ECR) for backend changes.

5.2 Architectural decisions

Microservice vs Monolithic Architecture

Microservice Architecture	vs	Monolithic Architecture
- Reduces the coupling of different logical components. - Allows us to have a manageable code base for each service, which makes it easier to test and debug code - Allow for autoscaling of individual services based on load requirement. - Easy to divide responsibilities among team members	**Benefits**	- Traditional client-server architecture → Increased familiarity - Easier to deploy (only 1 service to deploy)
Decision: Microservice architecture **Justification:** A microservice architecture will greatly decouple our backend server and reduce it into multiple smaller services that are easier to manage. More importantly, it helps speed up our development as one service is not tightly coupled with another and each member can work independently on their microservice.		

Docker-compose vs Kubernetes Cluster

Docker-compose	vs	Kubernetes Cluster
- Easy to use and manage as it runs on a **single host** - Easy to set up for hot-reload by creating volumes for each service	**Benefits**	- A container orchestrator that is used for networking containers across multiple hosts - Allows for easy deployment with auto scaling and monitoring

Decision: Docker-compose for local development and Kubernetes Cluster for actual deployment

Disclaimer: This is not really a comparison but more for justifying our rationale for using separate tools for local development and deployment.

Justification:
For local development, we opted with using Docker-compose for 2 reasons:

1. Easy to set up the same coding environment for all members by adding Postgres and Redis into the docker-compose network. Hence, removing the need to separately install these applications to run the backend server.
2. Still allow for hot-reload by defining the respective code base as a docker volume. This allows the service to detect changes in the code and reflect it in the docker container. This was something that is much more complicated to set up in the Kubernetes cluster which uses multiple hosts.

For actual deployment, we used Kubernetes Cluster for 2 reasons:

1. Simplicity in setting it up for deployment with features such as auto-scaling
2. AWS provides Elastic Kubernetes Service (EKS) which simplifies the actual deployment onto the cloud. (This is not possible for docker-compose networks)

Ingress vs API Gateway

AWS ALB Ingress	vs	AWS API Gateway
- Single entry point, smaller attack surface - Able to set up with ALB to distribute incoming traffic	**Benefits**	- Higher flexibility, allow for specifying specific entry points, authentication, etc

Decision: Ingress-managed load balancer

Justification:
API Gateway allows for higher customizability but it is complicated to set up and unnecessary for our application's use case. A simple Ingress with ALB will satisfy our use case with less effort required. A single entry point also makes it easier to monitor traffic and protect against

DDOS attacks.

Public-only Subnet vs Private & Public Subnets

Public-only Subnet	vs	Private & Public Subnet
- Each services have a public IP - Network-Address Translation (NAT) not required for communicating to internet	**Benefits**	- Common AWS architecture - ALB-ingress lies in public subnet - Individual services lies within private subnet - Better security as services are not in the public network

Decision: Public-only Subnet

Justification:
We initially started deployment with a private & public subnet with each microservice located within the private subnet as it provides better security. However, as the Interview subnet requires a connection to the internet to communicate with the LeetCode server, we had to use NAT gateway to translate private IP to public IP.

The problem: However, NAT gateway is expensive and incur a charge of $0.059/hour (it is on 24/7). We weighed the benefits of the improved security it provides against the cons of a higher cost.

We decided to go with a public-only subnet as having a public-only subnet does not greatly compromise our security. All of our data services (Postgres & Redis) can only be communicated from within the subnet and by users with the same security group. Hence, users cannot bypass the microservices to reach the data services.

For future improvements, our team can consider creating a specialised API gateway for communication with LeetCode API. However, we did not go about implementing it due to the steep learning curve and lack of time.

Redis vs RDBMS vs Memcached

Redis	vs	RDBMS	vs	Memcached
- Fast lookup time - Persistency - High Availability	**Benefits**	- Persistency - High Availability	**Benefits**	- Fast lookup time

Decision: Redis cache

Justification:
Our matching and interview session microservice requires a database that is constantly accessed and written to. For the matching service, the polling system requires a database that supports frequent read requests. For the interview session service, the database should support constant updating and reading of the session's code.

Hence, our priority was to ensure these 3 factors were optimized:

1. Fast Lookup time
2. Persistency
3. Availability

Redis cache ticks all the boxes mentioned above and hence is chosen for caching.

6 Design Patterns

6.1 Observer Pattern

```
const addListener = (cb: TokenListenerType) => {
  listeners.push(cb);
  return () => {
    const index = listeners.indexOf(cb);
    if (index > -1) {
      listeners.splice(index, 1);
    }
  };
};

const RefreshTokenService = {
  store,
  remove,
  get,
  addListener,
};

export default RefreshTokenService;
```

Fig 6.1.1 Observer pattern used in Account Token Authentication (Subject)

```
const UserProvider: React.FC = ({ children }) => {
  const [user, setUser] = useState<UserProfile | null>(null);
  const [loading, setLoading] = useState(true);

  // init connection with background and maintain synced data
  useEffect(() => {
    const unsubscribe = RefreshTokenService.addListener(getUser);

    // auto login
    refresh().catch(() => {
      // preload guest account
      const guest = getGuestAccount();
      setUser(guest);
      setLoading(false);
    });
    return unsubscribe;
  }, []);
```

Fig 6.1.2. Observer pattern used in Account Token Authentication (Observer)

We employ the use of observer pattern within the frontend client to maintain user authentication status. The RefreshTokenService serves as the **subject** that provides the *addListener* method for observers to subscribe a callback and listen to changes in authentication status.

It pushes changes to its subscribers when there is a change in authentication status. This pattern is used to auto-update the user context when a change in authentication status is detected. (user login or logout)

6.2 Dependency Injection

```
export class ProfileController {
  constructor(private readonly profileService: ProfileService) {}

  /**
   * Get profile of authenticated user
   */
  @ApiOkResponse({ type: Profile })
  @UseAuth(JwtAuthGuard)
  @Get('me')
  findMe(@AuthUser() requester: User): Promise<Profile> {
    return this.profileService.findOne(requester.id);
  }
```

Fig 6.2.1. ProfileController injecting ProfileService as a dependency

The Controllers in the backend services inject their relevant services as a dependency to handle the internal logic in the REST API call. This dependency injection (DI) pattern is used consistently throughout our backend services as NestJS uses a framework similar to Angular's DI framework.

6.3 Data Transfer Object Pattern

```
export class JwtResponseDto {
  userId: string;
  userEmail: string;
  expiresIn: number;
  accessToken: string;
  refreshToken: string;
  tokenType: 'cookie';
}
```

Fig 6.3.1. DTO pattern used in Account Token Authentication

The login REST API endpoint returns a DTO to batch multiple data items that the frontend client needs when the user logs in to our application.

6.4 Pub-sub Pattern

The Interview and Chat Microservice uses Socket.io, which implements the pub-sub pattern, where Participants and the microservices can both publish and subscribe to messages sent through Socket.io.

Refer to Section 7.3 & 7.4 for further explanation.

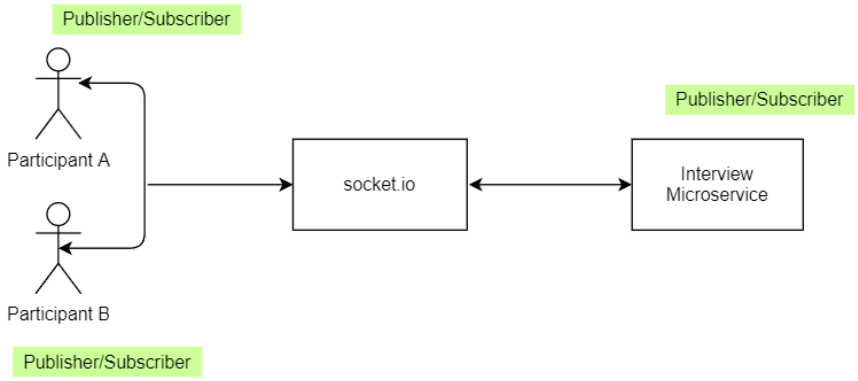


Fig 6.4.1. Pub-Sub pattern used in code

7. Backend (Microservices)

7.1 Account Microservice

Technology

- NestJS
- Swagger for API documentation (Development env only)
- Passport.JS
- NodeMailer
- Postgres database

ER Diagram

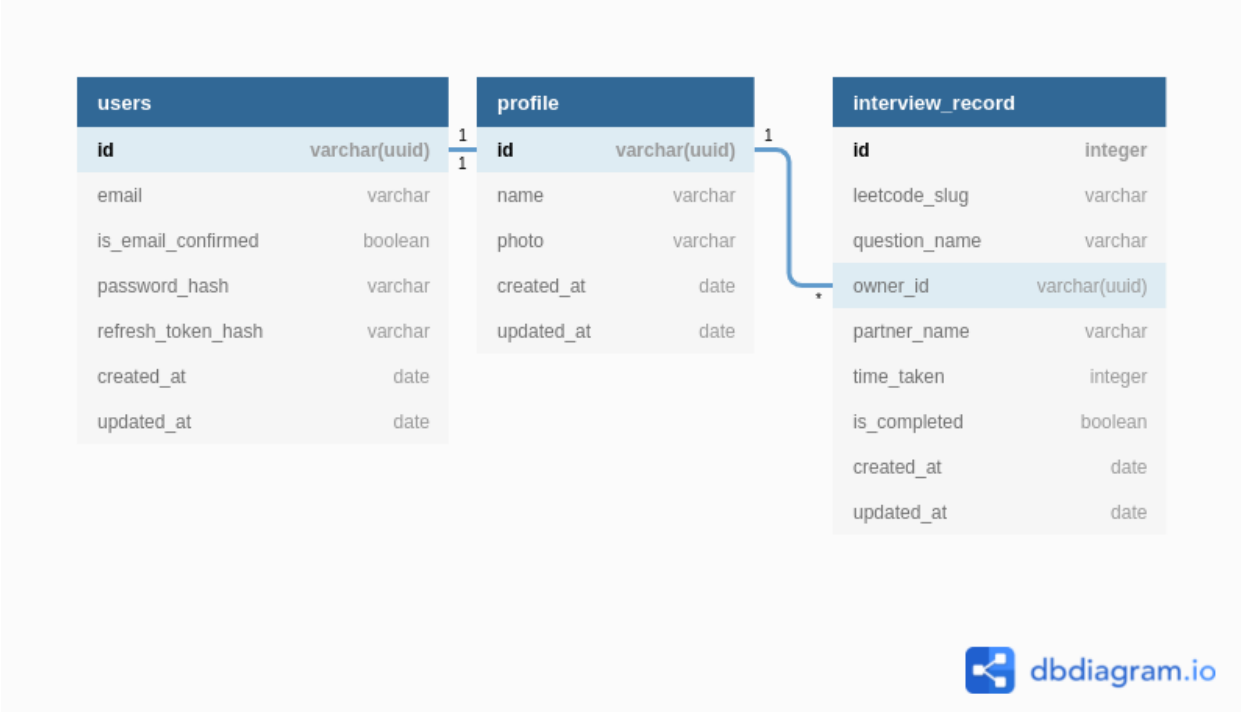


Fig 7.1.1. ER Diagram for user account management (Interactive Link)

REST API

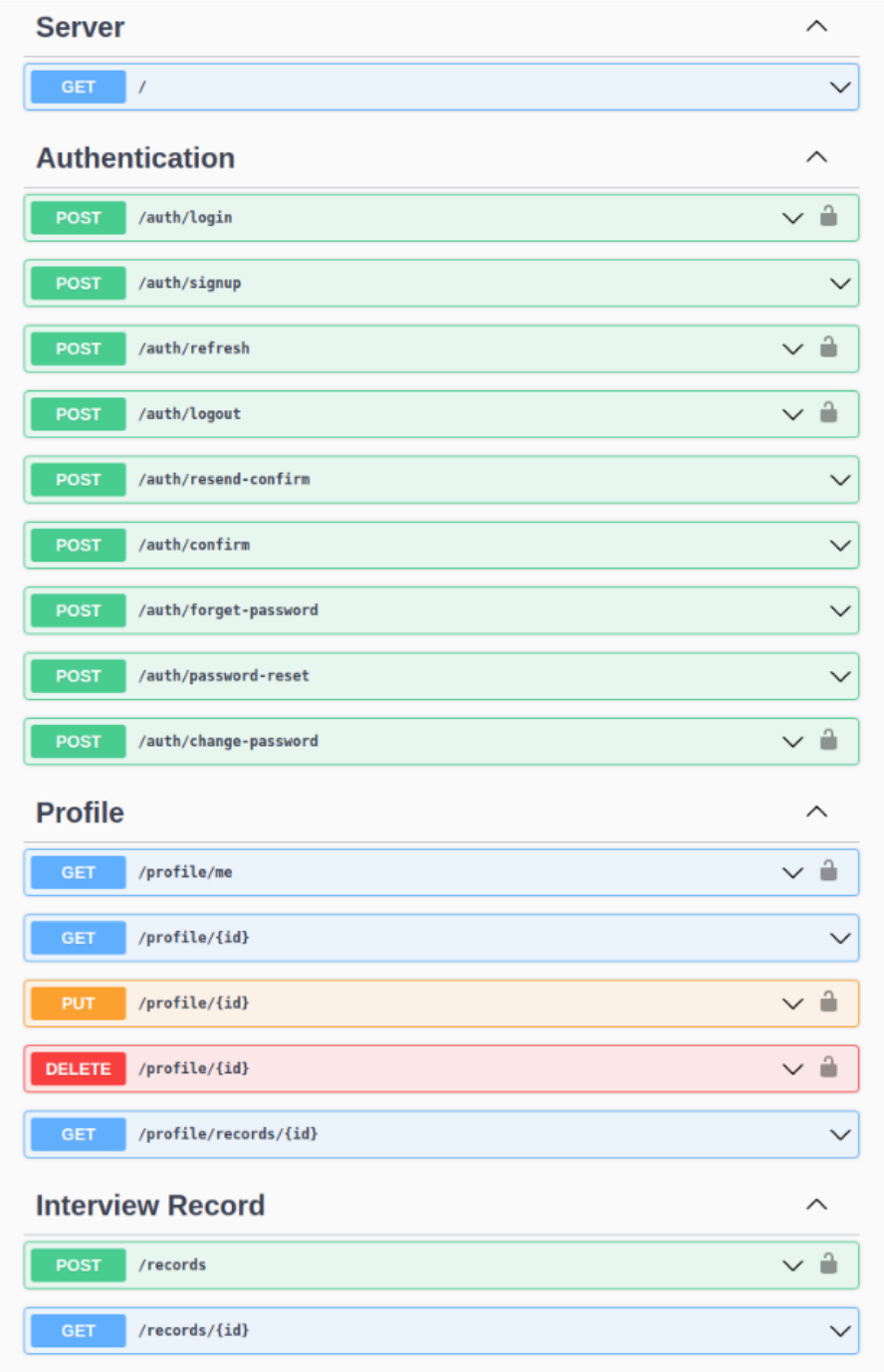


Fig 7.1.2. Swagger API Overview of Account Service

For detailed swagger documentation of the API, you can run the account microservice locally and access it through http://localhost:8081/api

Server

A simple GET endpoint is provided at `/` which returns a 'Hello World!' message. This endpoint serves as a health check for kubernetes ALB.

Authentication

Authentication is done using Passport.js and uses 2 type of strategies:

1. Local strategy (traditional email and password authentication)
2. JWT strategy (to decode signed JWT token and verify authenticity)

The account MS uses JWT access token to authenticate the user and JWT refresh token to persist the user's session until token expires or user logs out. The user's JWT refresh token is stored on client's local storage and the access token is saved in session storage.

Using JWT access token provides 3 main benefits:

- Stateless RESTful API design where server does not need to store user session state
- JWT token signed with server's secret key ensures integrity of token
- JWT access token with a short-lived expiry reduces the chance of unauthorized access

NodeMailer module is used for sending email to verify a user's email. Email validation is important as it allows us to send confidential password reset email to the right user.

Profile

Upon authentication, the client will make a GET request to `/profile/me` to retrieve the user's profile data and a GET request to `/profile/records/:id` to retrieve the user's interview history.

The endpoints are protected by adding a JWT guard to the controller method

```
/**
 * Get profile of authenticated user
 */
@ApiOkResponse({ type: Profile })
@UseAuth(JwtAuthGuard)
@Get('me')
findMe(@AuthUser() requester: User): Promise<Profile> {
  return this.profileService.findOne(requester.id);
}
```

Fig 7.1.3. Swagger API Overview of Account MS

Interview Record

The interview record endpoint at `/records/*` has 2 simple endpoints:

- POST `/records` to create new interview record *(Protected endpoint)*
- GET `/records/:id` to get an interview record by its id. (currently unused but can be used for displaying detailed information of a past interview)

7.2 Match Microservice

Technology

- NestJS
- Redis Cache
- Swagger for API documentation (Development env only)

Overview of REST API

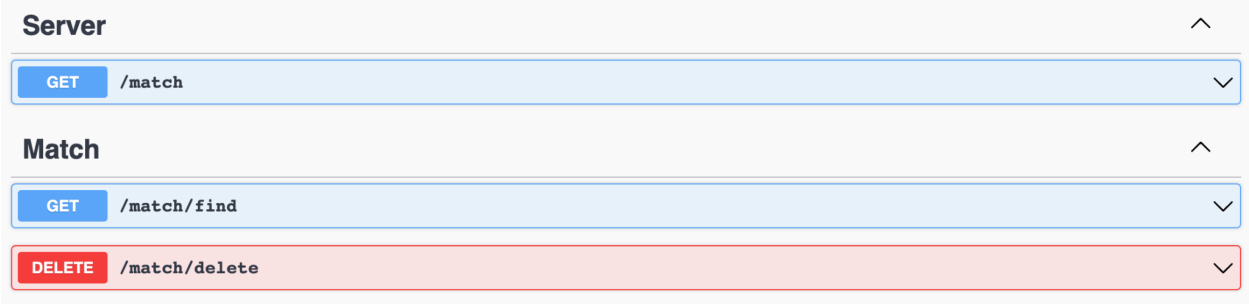


Fig 7.2.1. Swagger API Overview of Match Service

For detailed swagger documentation of the API, you can run the match microservice locally and access it through http://localhost:8084/match/api

GET /match

A simple GET endpoint is provided at `/match` which returns a 'Match Service is up!' message. This endpoint serves as a health check for kubernetes ALB.

GET /match/find

The find match method attempts to match this user with another user in the queue with the same selected difficulty and language. A redis cache is used as the matchmaking queue.

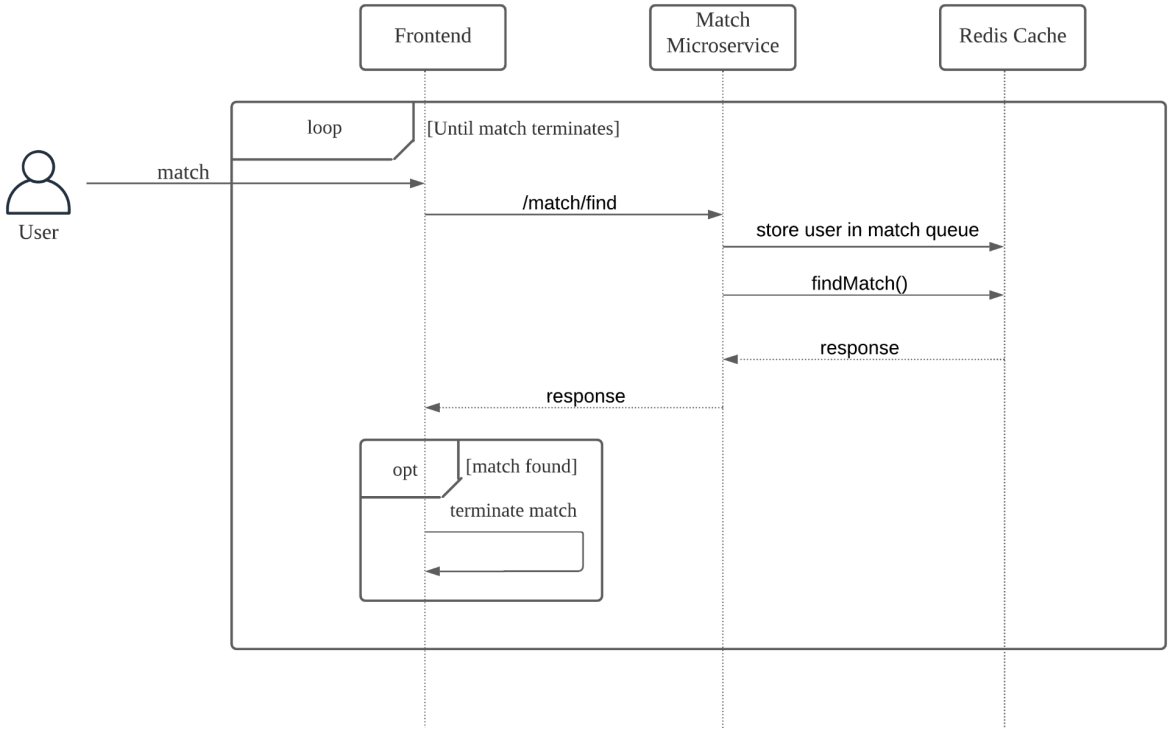


Fig 7.2.2. Match sequence diagram

Request Flow:

1. User requests to match on the frontend
2. Frontend calls the match/find API endpoint.
3. The match microservice first stores the user into the matchmaking queue
4. It attempts to find another user on the queue with the same selected difficulty and language
    a. On successful match, a unique session id is generated and the frontend creates and routes the user to the interview page.
    b. On an unsuccessful match, frontend will retry the match (step 2) after waiting for 5 seconds. After 5 attempts (30 seconds), the match is terminated and the user is informed of a failed match.

DELETE /match/delete

The delete match method removes the user from the matchmaking queue. This API endpoint is called when the matching terminates, and prevents the user from being matched again with other users.

7.3 Interview Microservice

Technology

- NestJS
- Redis Cache
- Socket.io
- CodeMirror

Overview of REST API

To access the full Swagger API Documentation, you can run the interview service microservice locally and access it through http://localhost:8084/interview/api

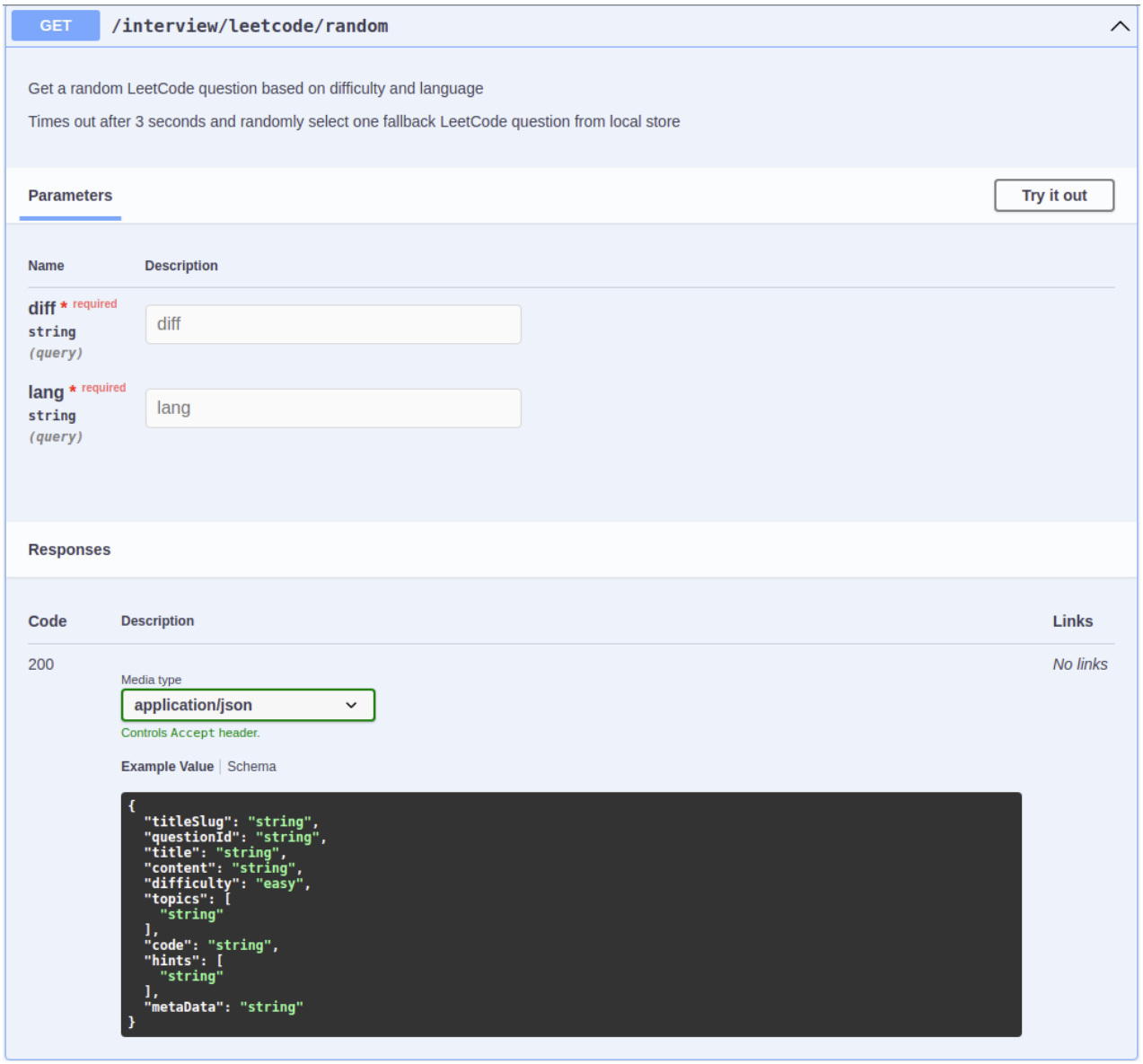


Fig 7.3.1. Endpoint to get a random LeetCode question

Implementation

The interview microservice provides a collaborative coding editor implemented using a third party text editor software (code mirror), for participants to edit their solution in real time.

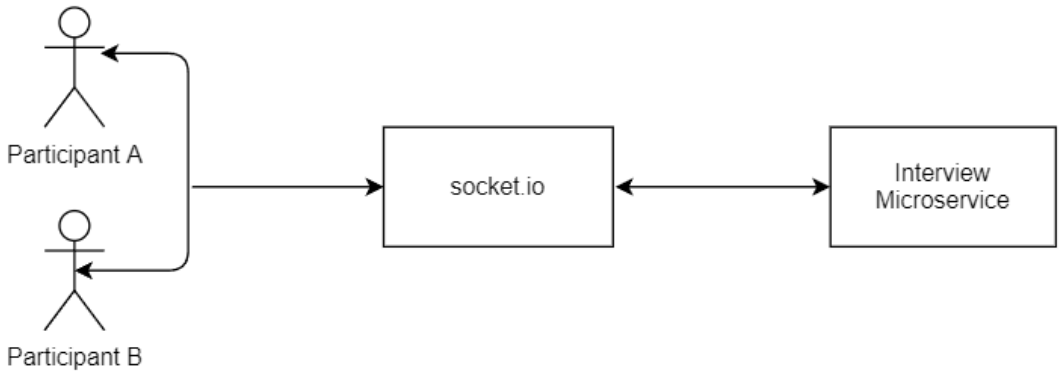


Fig 7.3.1. Pub-sub interaction between participants and interview microservice

As shown in Fig 7.3.1, Socket.io helps to enable real time communication between 2 participants in an interview session. Interview microservice will use the unique `sessionId` to send the messages only to relevant participants.

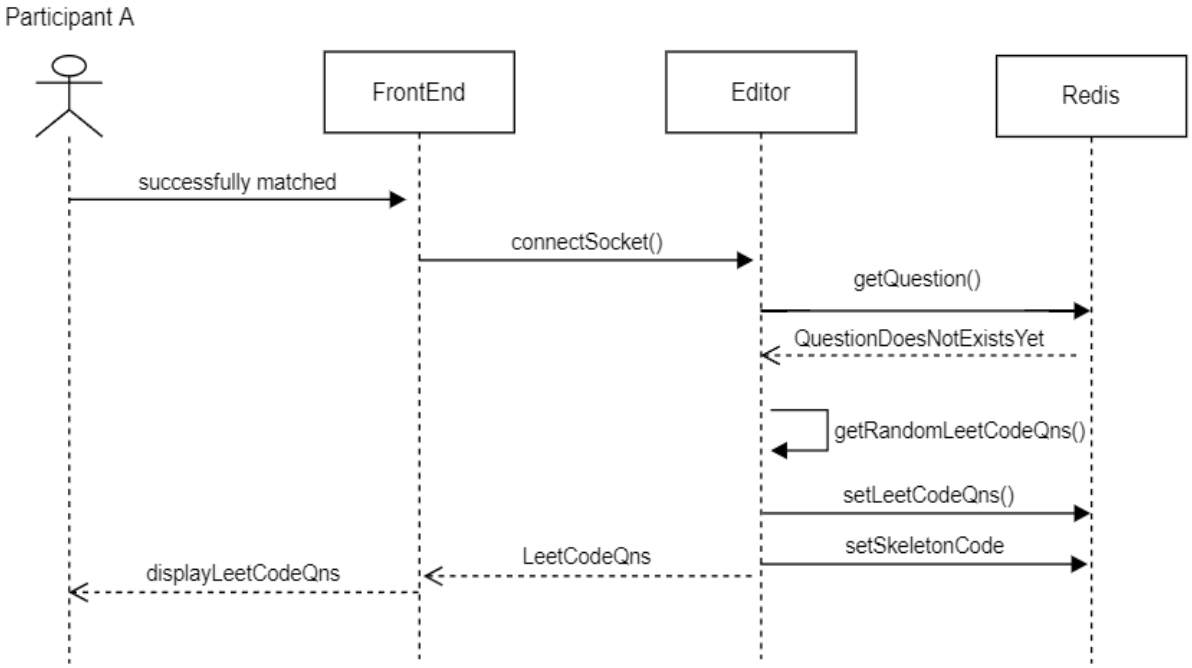


Fig 7.3.2. Sequence diagram between participants A connects to a session

After a successful match, the match microservice will provide the sessionId and the relevant session details (partner, difficulty, language) to the interview page. In the interview session page, the React client will connect to the editor websocket using Socket.io.

As seen in Figure 7.3.2, the Editor microservice then query the Redis service to check if there is any existing interview data. Otherwise, it calls the getRandomLeetCodeQns method which calls the LeetCode API to randomly retrieve a question based on the selected difficulty for the interview session. The service will then cache the code and question in Redis before sending the LeetCode question back to the users through Socket.io event

To overcome the issue of potential data conflict in the coding editor due the functionality of real-time concurrent edits by participants, we used an npm package extension, CodeMirror (https://github.com/convergencelabs/codemirror-collab-ext), that builds on a third party collaborative application development platform (convergence.io). More information on how it solves the concurrency issue is explained in section 8.3.

When participants are finished with their session, they can submit their solution as illustrated in Fig 7.3.3. Subsequently, the participants will receive an interview record and their unique sessionId will be removed from the Redis cache.

```
const handleSubmitAccept = () => {
  clearInterval(timer);
  editorSocket.emit('COMPLETE_SESSION_CONFIRM', {
    sessionId: sessionId,
    userId: user.id,
    time: time,
  });
  createRecord(time, true);
  setSubmissionStatus(SUBMISSION_STATUS.Summary);
};
```

Fig 7.3.3. Participants subscribing to the sessionId event

7.4 Chat Microservice

Technology

- NestJS
- Socket.io

Implementation

The chat microservice provides real-time communication for participants by using the publish and subscribe pattern (Fig 7.4.1) and is implemented using socket.io. Unlike the interview microservice, the chat does not use redis to cache the chat messages as we felt that the cost of caching the messages outweighs the benefits of doing so.

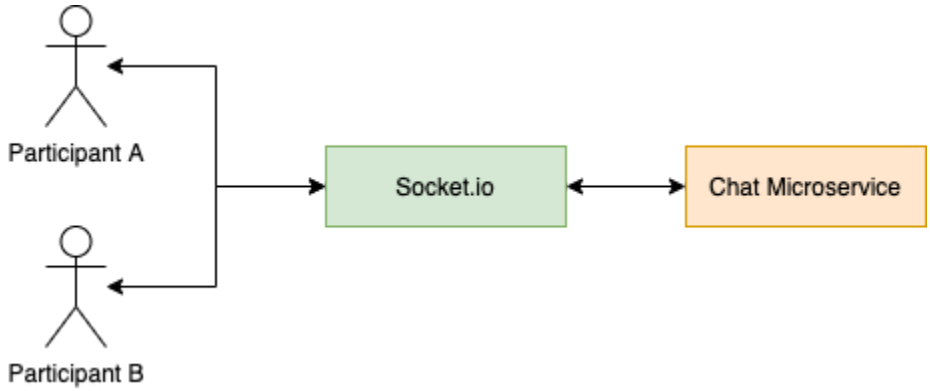


Fig 7.4.1. Pub-sub interaction between participants and chat microservice

Upon a successful match, both participants of the interview session will be connected to the chat microservice and subscribe to the sessionId event (Fig 7.4.2), which uniquely identifies each session. When one participant sends a message, handleSend function as shown in Figure 7.4.3 publishes a sessionId event with information about the message, which in turn notifies the other participant and him/herself about the new message.

```
chatSocket.on(sessionId, (message) => {
  setMessages((oldMessages) => [...oldMessages, message]);
  const ref = document.getElementById('chat-box');
  ref.scrollTo(0, ref.scrollHeight);
});
```

Fig 7.4.2. Participants subscribing to the sessionId event

```
const handleSend = () => {
  if (currMessage !== '') {
    chatSocket.emit('newMessage', {
      sessionId,
      payload: {
        id: user.id,
        sender: user.name,
        msg: currMessage,
      },
    });
    setCurrMessage('');
  }
};
```

Fig 7.4.3. Participants emitting a sessionId event

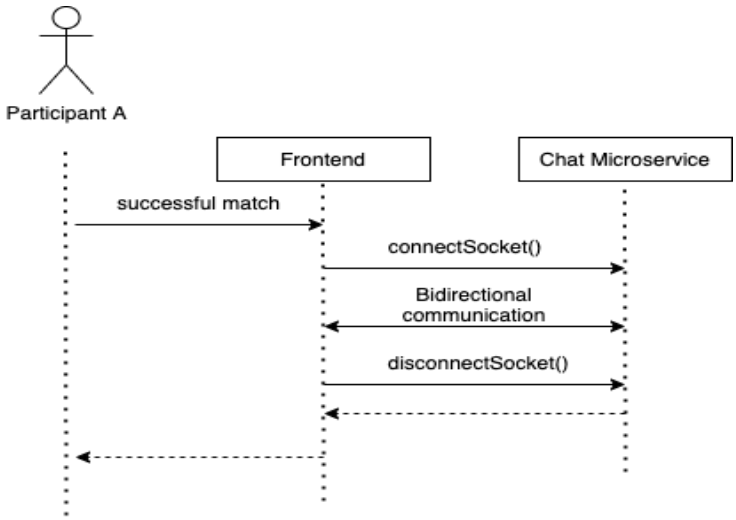


Fig 7.4.4. Sequence diagram to detail interaction between frontend and backend

The UI will be updated each time a participant is notified of an event. The general flow of the interaction between frontend and chat microservice is detailed in Fig 7.4.4.

8. Frontend

8.1 React Framework

React is a frontend framework that places its emphasis on the concept of components, in particular reusing of components. Our team opted to use react due to its high maintainability and flexibility compared to other frontend frameworks.

8.2 Frontend Code Structure

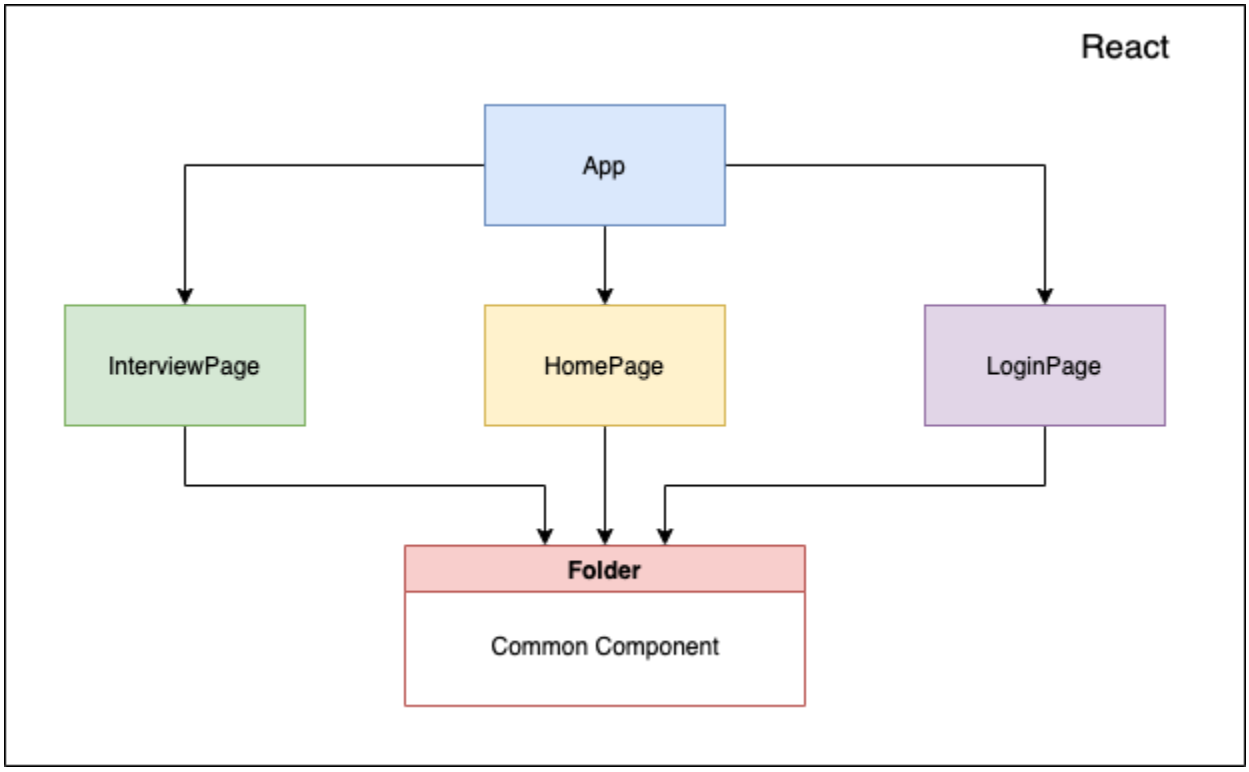


Fig 8.2.1. Architecture diagram of Frontend

The app component acts as the controller class which contains routes to 3 main pages/components of our application: InterviewPage, HomePage and LoginPage. These 3 main pages/components then utilize components in the common components folder (contains self-made components) to power up itself.

8.3 Frontend Decisions

Technology	Rationale/Purpose
Material UI	Chosen as the main frontend library as it is an enterprise-grade library with many powerful components with well written API that can power most of the components in our frontend application.
React Toastify	Used as the notification library as it is lightweight and easy to use. It also helps standardise the form of user feedback and notifications that our app uses.
React Context (State management)	Our team opted against using redux for state management as we only need to persist user profile data throughout the application. Hence, a simple React Context implementation was sufficient for our small application.
Browser Storage (Data management)	To persist interview session data across tabs, our team uses the in-built sessionStorage API which is compatible across various modern browser[2] To persist long-term persistent data (JWT refresh token), we store them in the browser's local storage.
CodeMirror	Code mirror was used as our code editor library as it is specialized for editing code, and comes with a number of language modes and addons that implement more advanced editing functionality. It also provides Operational Transformation (OT) support which is crucial for implementing a real-time collaborative[3]. Socket.io pub/sub functionality is used to help share document changes between two parties. One party publish the document changes (eg. insert operation) and the other party listens to the change and implement it by calling CodeMirror's insert operation In the event that a merge operation fails due to conflicting edits, the client will regard the current code document as 'dirty' and attempt to get the full code document from the server (the single source of truth).

[2] https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage#browser_compatibility
[3] https://codemirror.net/6/examples/collab/

9. DevOps

9.1 Tools

Tools	Purpose	Type
Docker Compose	- For standardising the development environment (provide postgres and redis image definition in yaml file)	Local Development
Nginx	- To replicate kubernetes ingress routing. - Route /match to match service, /interview to interview session service, /chat to chat service and remaining / to account service	Local Development

eksctl	To create and manage the EKS cluster
	- Used to create required AWS services:
	  - VPC, NodeGroup, CloudFormation and EKS cluster
	  - Definition for EKS creation is provided in cluster.yaml
	- Used to connect to EKS cluster
		Deployment

kubectl	- To deploy applications, inspect and manage cluster resources, and view logs - Definition for each cluster resources is defined in k8s folder	Deployment
aws (CLI tool & AWS console)	- CLI tool: to login to aws - Console: to manage other AWS service (eg. Cloudfront, Route53, ACM)	Deployment

9.2 CI/CD

CI/CD is done using Github Actions. The Github Actions is set to run on PR merge to the main branch. Currently, there are 5 defined Github actions (4 for backend and 1 for frontend). The Github action runs only when code changes are detected in their respective folder.

For each backend microservices, there are corresponding Github actions that update the image in Amazon ECR. The workflow consists of 3 main part:

1. Logging in with provided AWS IAM credentials
2. Building the image and tagging it with its corresponding Amazon ECR image tag
3. Push the image into ECR

For frontend, the Github Actions is responsible for building the React app and pushing the build folder into S3.

10. Workflow

10.1 Weekly Sprint

Our weekly sprints are done using Jira, a project and issue tracking tool for Agile project management.

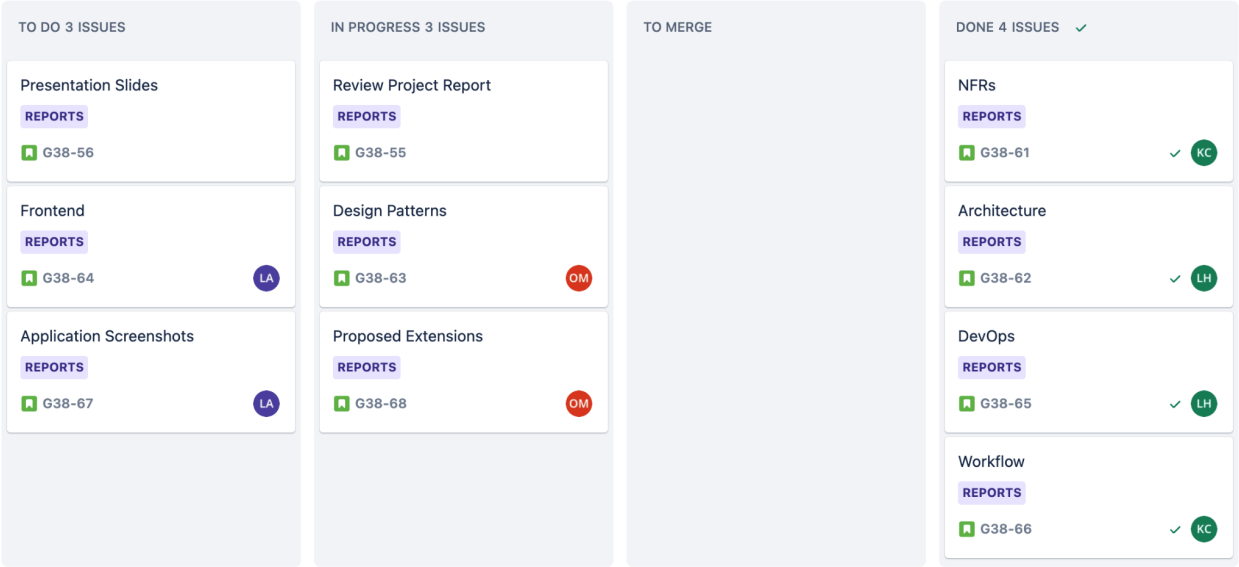


Fig 10.1.1. Sample issue tracking in Jira

Weekly tasks:

- Assessment on backlog priorities
- Progress sharing
- Assignments of tasks and planning of sprint for the week
- Deployment of large features
- Conduct regression testing on new features

10.2 Milestones

Milestones	Details	Weeks
1 (Setup)	● Design Frontend + Setup landing page ● Setup Microservices (Server, database) ● Setup API gateway and service registry ● CI/CD pipelines (Frontend, Backend)	Week 7, 8 **(2 weeks)**
2 (Achieve MVP)	● Account service ● Matchmaking service ● Coding platform service ● Chat service ● Integrating of services	Week 9, 10, 11 **(3 weeks)**
3 (Touch up & documentation)	● Deployment ● Functional testing ● Testing of NFRs (Scalability, Performance, Security, Availability) ● Documentation and report	Week 12, 13.5 **(1.5 weeks)**

10.3 Development Process

- Branching workflow
    - New branch for each new feature / bug fix
- Pull request workflow
    - Minimum of 1 approval before merging into the branch

11. Application Screenshots

11.1 User Authentication

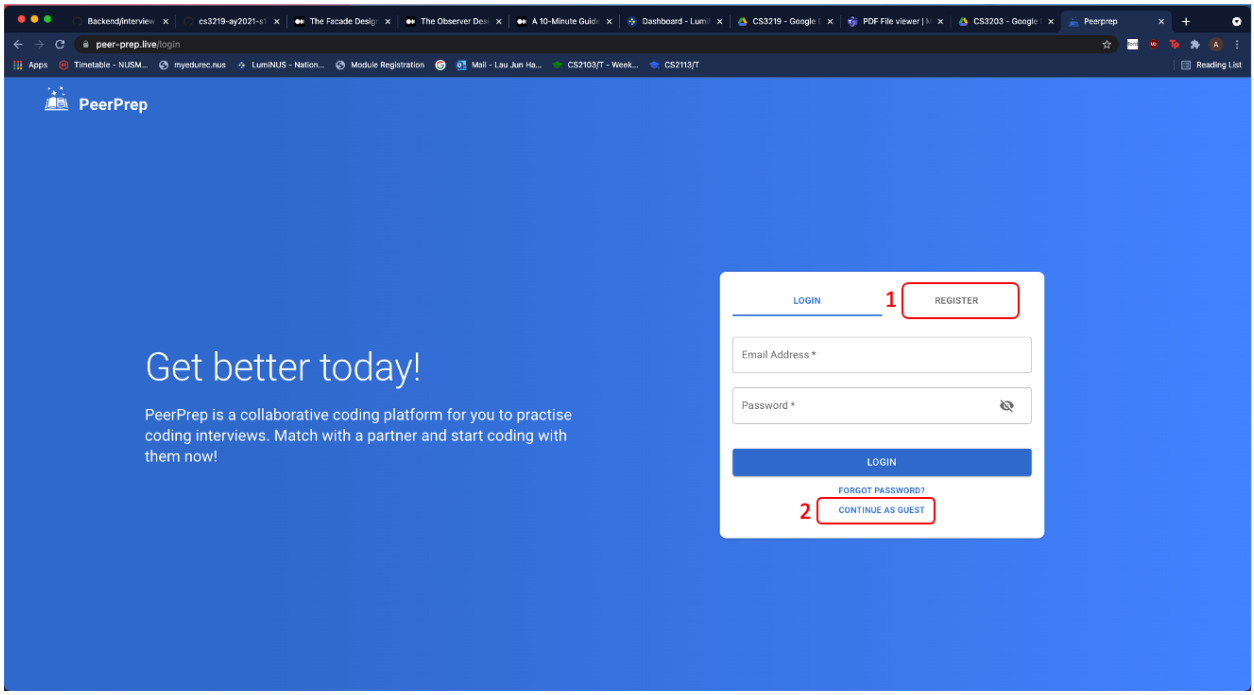


Fig 11.1.1. Authentication page

Users can login to their accounts at this page shown in Figure 11.1.1. If users do not have an account, they can register by clicking on the register button highlighted as 1, or they can choose to continue as a guest as shown in the 2nd red box.

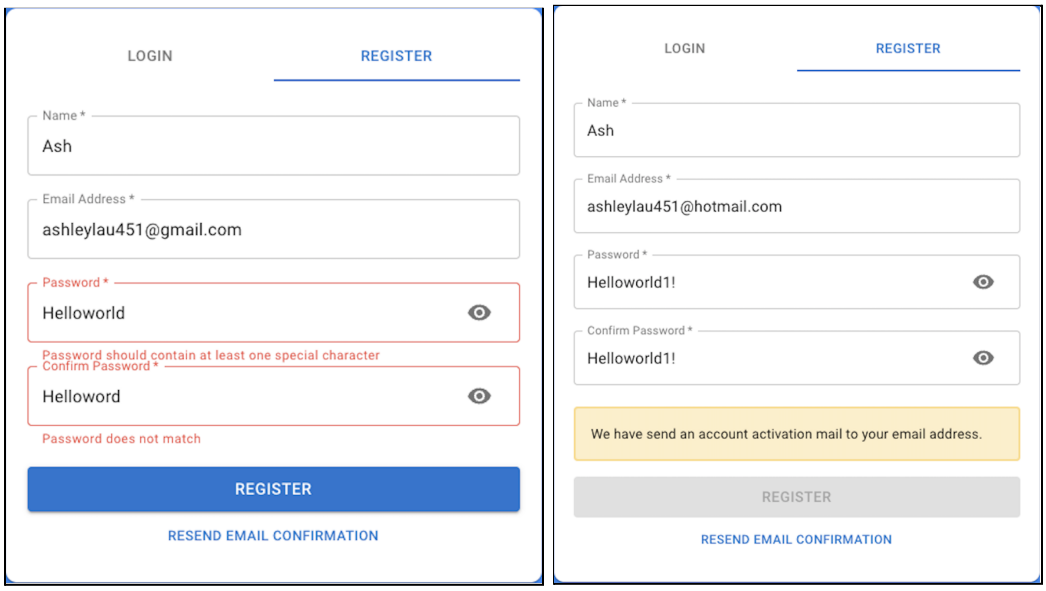


Fig 11.1.2. Registration page

When creating an account, there will be checks to ensure the strength of the password (e.g. one capital letter, one number and one special character) and both input passwords must match (Fig 11.1.2).

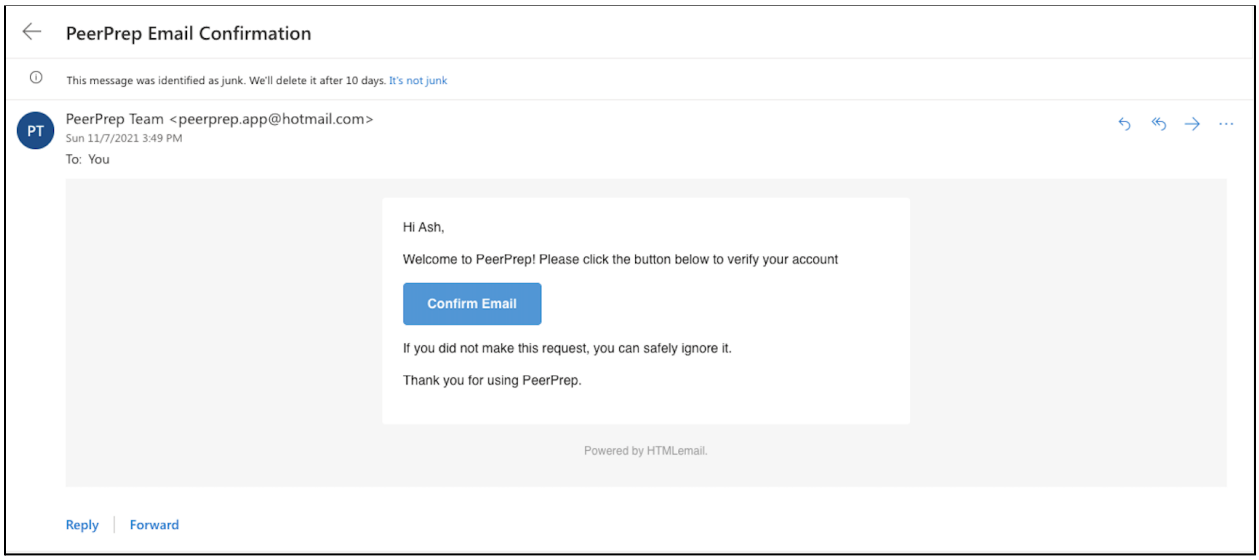


Fig 11.1.3. Email confirmation

Upon the creation of an account, users will need to verify their accounts by using the unique email verification link sent to their email.

11.2 Home page and Matching making page

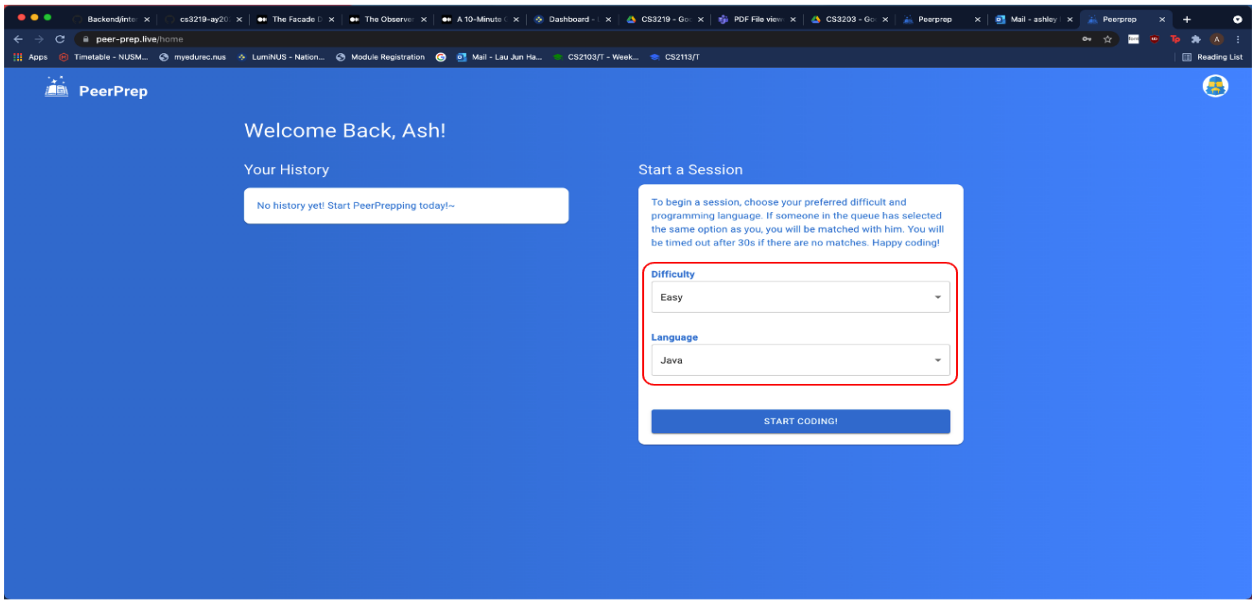


Fig 11.2.1. Email confirmation

Upon logging in, users can start their coding session by selecting the difficulty and programming language as shown in the red box in Figure 11.2.1. Users can then press the "Start coding" button to start matching as shown in Figure 11.2.2. Users will be timed out if there are no matches within 30 seconds.

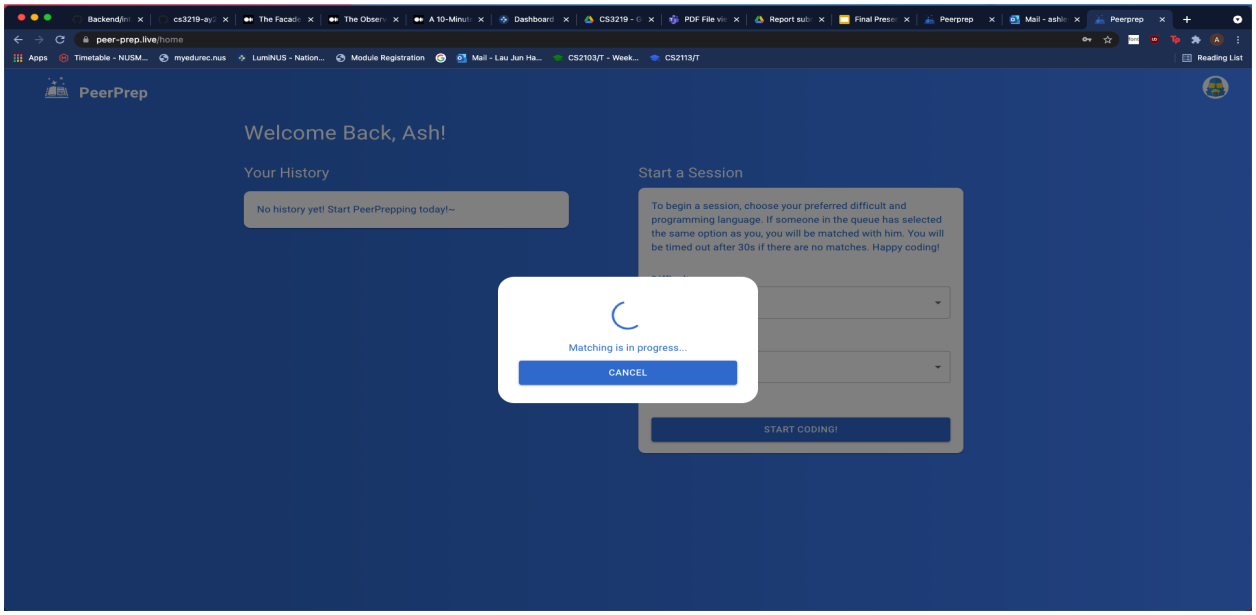


Fig 11.2.2. Email confirmation

11.3 Interview Page

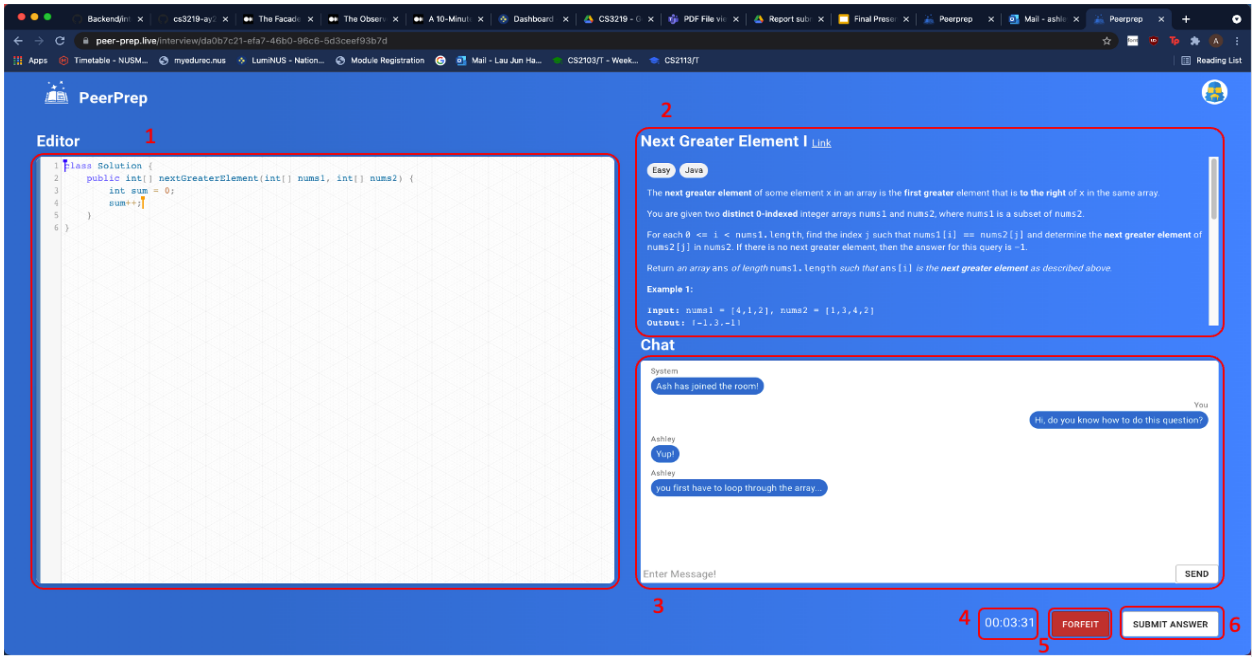


Fig 11.3.1. Interview page

Upon a successful match, participants will be directed to the interview page, which contains the following features highlighted in the red boxes (Figure 11.3.1):

1. Editor - Live collaborative platform.
2. Question - Question from LeetCode.
3. Chat - Chat box for participants to chat and communicate their answers.
4. Timer - timer that keeps track of the interview duration.
5. Forfeit button - Forfeit button for participants to forfeit. Participants can forfeit but their partners will still be able to work on the question.
6. Submit answer button - Participants can complete the session when they are done by pressing this button. If participant A is the one who initiates the submit answer request (Figure 11.3.2), participant B has to accept it before the interview session is closed (Figure 11.3.3).

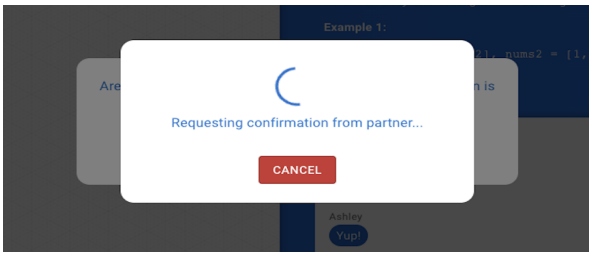


Fig 11.3.2. Participant A initiating the submit answer request

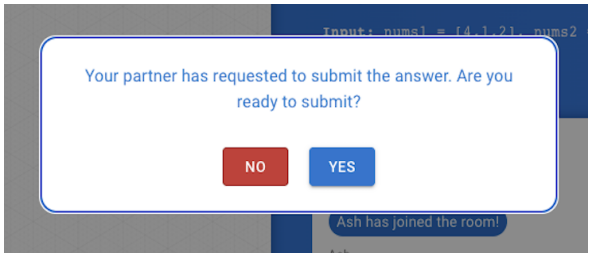


Fig 11.3.3. Participant B has to accept the submit answer request

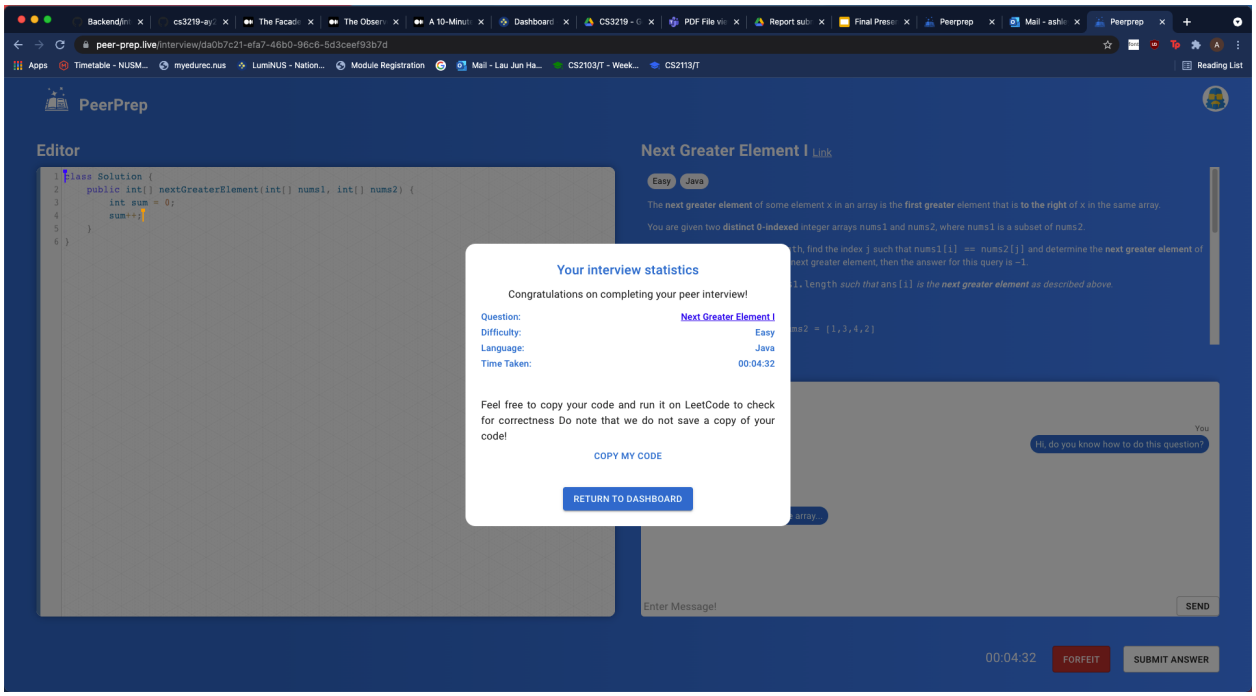


Fig 11.3.4. Interview statistics

Upon successful submission, the interview statistic will be shown (Figure 11.3.4), and participants will get the option to copy their code from the interview session.

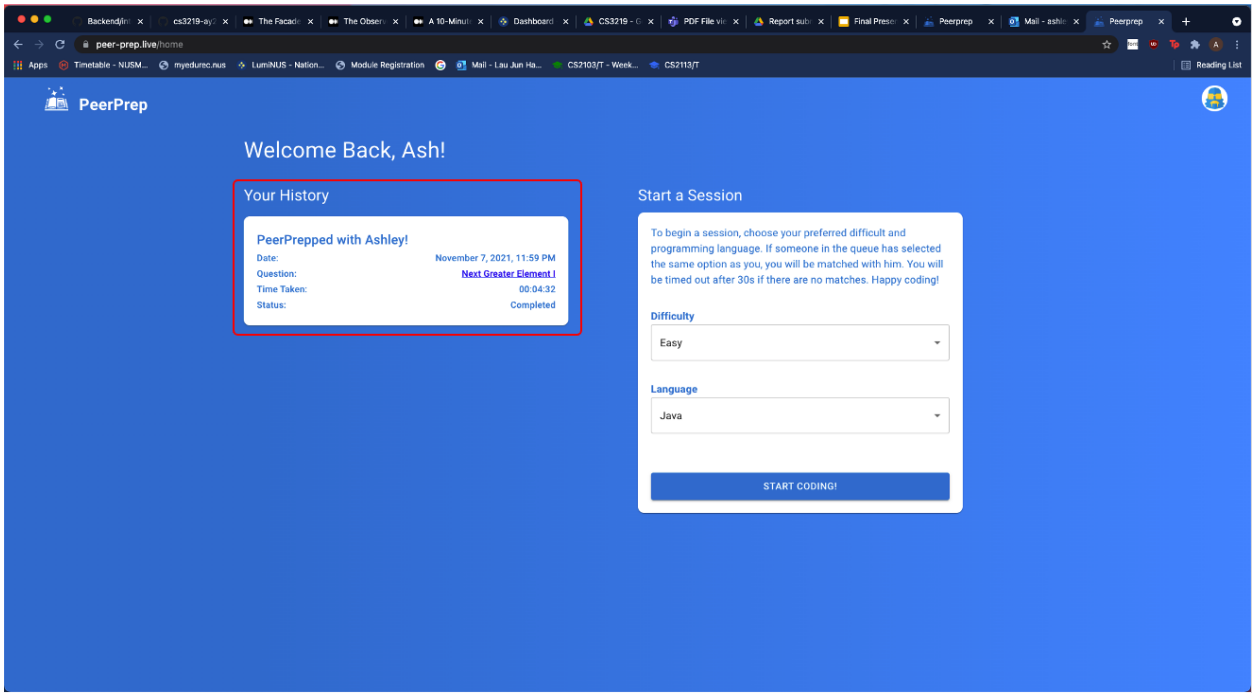


Fig 11.3.5. History section in Home page

After every interview session, the history section will display information such as partner name, date and time of interview, question name, time taken and interview status to the users as shown in the red box in Figure 11.3.5.

12. Proposed extensions

12.1 WhiteBoard Collaboration

To better assist participants in communicating their ideas by allowing them to draw diagrams and models on the whiteboard tool, which would be especially helpful in tackling algorithm questions that involve graphs (DFS, BFS, Dijkstra).

With our PeerPrep web application based on the microservices architecture, we will be able to build this extension with much ease by adding a WhiteBoard microservice.

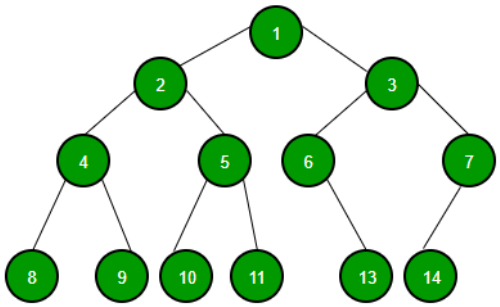


Fig 12.1.1. Example of binary tree diagram drawn on the whiteboard tool

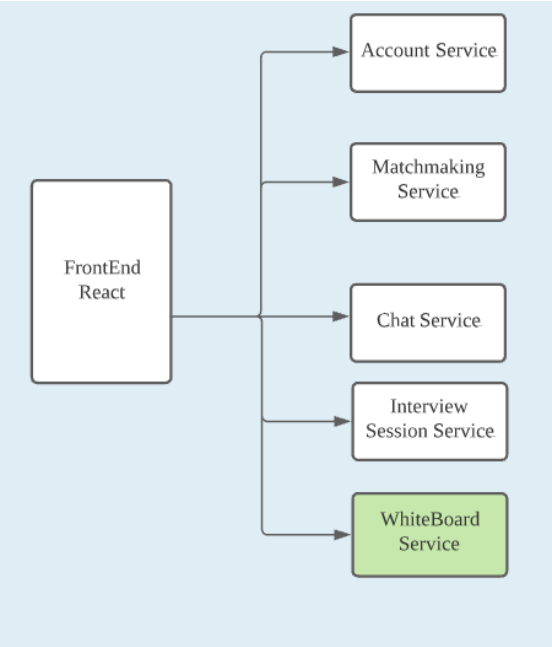


Fig 12.1.2. Proposed architecture with the new WhiteBoard service

12.2 Voice Chat

For a more interactive and fruitful discussion, having a VoiceChat function would enable participants to readily bounce ideas off each other.

With most of our PeerPrep web application based on the microservices architecture, we will be able to build this extension with much ease by add a VoiceChat microservice

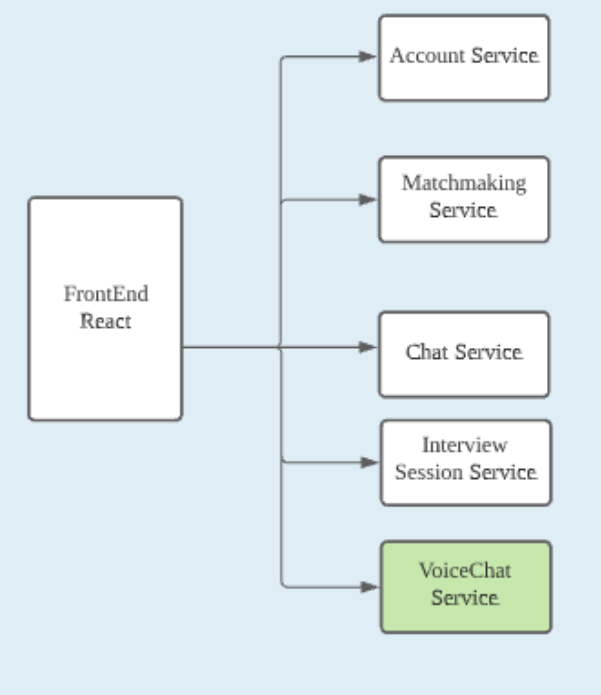


Fig 12.2.1. Proposed Voice Chat Extension

12.3 Comprehensive Interview History

Currently, our interview session history does not store the user's code and only provides a barebone of the user's past interview session.

To provide more benefits for registered users, we can allow them to better track their own interview progress. This can be done by providing them a specialised page for viewing a specific interview history. The page shows the submitted code so that the user can review his submission.

To achieve this, we can split the current account microservice into 2, where 1 only focuses on the user authentication and profile, while the other handles the user interview history (CRUD functionalities)

As the user interview history is expected to scale with increased usage, the new Interview History microservice can also provide the logic to support pagination, sorting and filtering on the interview history API endpoint.

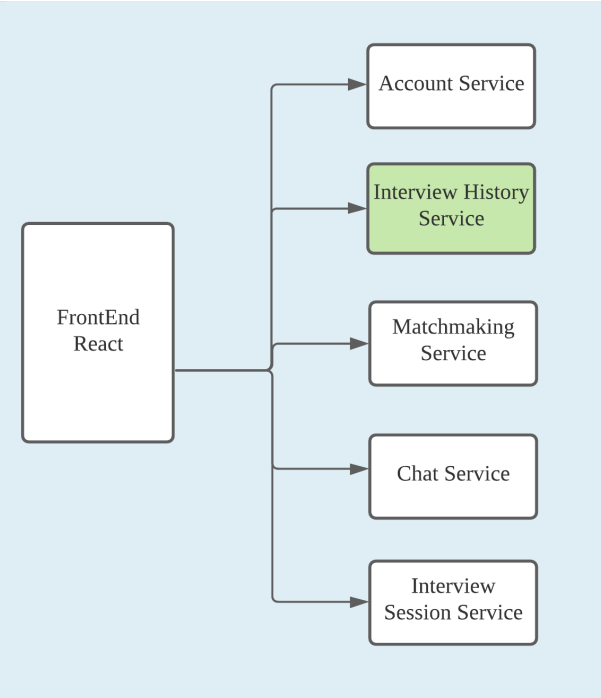


Fig 12.3.1. Proposed Interview History Improvement

13. Reflection

Overall, this project was a fulfilling one as it exposed us to the entire software engineering cycle, from planning to coding for the frontend and backend frameworks and finally deploying our code. The following are the main learning points for the entire project:

1) Learning curve for the project was steep as we had to pick up React, Nestjs, TypeScript and deployment of application using aws
2) Microservice architecture is harder to deploy initially, but greatly improves scalability and manageability of our code base
3) Live collaborative text editor is difficult to implement as it requires knowledge on distributed systems, which is something that none of us are familiar with
4) Deployment is expensive in general. For new AWS users, it is easy to make deployment mistakes which may result in unintended increase in deployment costs.

Despite our best efforts, we were unable to get the necessary clearance to register an account using our NUS mail. If a user was to register for a PeerPrep account using their NUS emails, the verification email will be successfully sent out. However, it is not located in NUS mail or in the spam folder. It seems that the PeerPrep email got filtered out by NUS.

After some research, we found a post which implies that EC2 IPs are banned and blacklisted almost in all spam filtering service[4]. Hence we shifted our mailing service from within the EC2 cluster to AWS SES, in the hope that it fixes the email filtering issue. However, NUS users are still not able to receive mail from PeerPrep.

After several trial and errors, we came to the conclusion that NUS mail is auto-filtering out all emails that originate from AWS. This remains as a limitation of our application which results in NUS users not being able to create an account using their NUS email.

[4] https://serverfault.com/questions/165854/my-ec2-instances-email-is-being-spam-blocked-by-gmail