



CS3219

Software Engineering Principles and Patterns

Project Report: PeerPrep

Student Name	Chen Caijie	Keith Teo	Nicolas Wee	Permas Teo
Matriculation Number	A0166916U	A0188985Y	A0227243L	A0180316M

Code Repository URL:

<https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-ay2021-s1-project-2020-s1-g23>

Contributions	4
1. Introduction	5
2. Functional Requirements	5
2.1. Account Management	5
2.2. Matchmaking System	5
2.3. Mock Interview Process	6
2.4. Application Client	6
3. Non-functional Requirements (NFRs)	6
3.1. Security Requirements	7
Key points	7
Security View	8
3.2. Performance Requirements	9
Key points	9
Performance View	9
3.3 Availability Requirements	11
Key points	11
Availability View	11
3.4 Scalability Requirements	12
Key points	12
Scalability view	12
4. Architectural Design	13
4.1 Architecture Diagram	13
4.2 Architectural Decisions	14
5. Design Patterns	17
5.1. Builder pattern	17
5.2. MVC pattern	17
5.3 Pub-Sub Pattern	18
6. DevOps	18
6.1. Sprint Process	18
6.2. CI/CD	21
6.3. Manual Deployment	22
7. Microservices	23
7.1. User	23
Currently supported endpoints:	23
New User	23
Login	24
Start Interview	24
End Interview	24
Get Profile	25
7.2. Match	26
Currently supported endpoints:	26

Get Match	26
Update Elo	26
Create User	27
Get User	27
7.3. Real-time Communication Microservices	27
7.3.1. Chat	28
7.3.2. Editor	30
8. Frontend	30
8.1. Tech Stack	30
8.2. Redux	30
9. Application Screenshots	32
9.1. User Authentication	32
9.2. User Home Page	33
9.3. Matching System	33
9.4. PeerPrep Interview	34
9.5. PeerPrep Interview Rating	35
10. Remarks	36
10.1. Challenges Faced	36
10.2. Potential Extension Features	36
10.2.1. Video Conferencing	36
10.2.2. Store Code History	37
Appendix A: Sample Meeting Minutes	38

Contributions

Name	Technical Contributions	Non-technical Contributions
Chen Caijie	Implement user and chat microservices. Implement chat user interface Implement interview history page Set up CI/CD for frontend	Requirements documentation Project report
Keith Teo	Implement editor microservice, frontend integration	Requirements documentation Project report
Nicolas Wee	Implement match microservice (backend) DevOps, Backend deployment	Requirements documentation Project report
Permas Teo	Set up frontend (routing/styling/architecture) Implement static frontend pages Integrate match microservice	Webapp design Requirements documentation Project report

1. Introduction

Students often have trouble with technical interviews. Many find it tough to articulate their thoughts properly while coding at the same time. Our purpose is to create an interview Preparation Platform where students can find peers to practice whiteboard style interview questions together.

We created Peerprep, a peer support system where users are engaged with collaborative whiteboard-style programming to practice for challenging technical interviews. The app is expected to handle collaborative whiteboard-style programming. With IPP, everyone can code together on the same file at the same time. You can share with anyone, view edited code in real time, chat and comment for discussions. The app also supports some basic features as well e.g., user authentication and save and retrieve relevant data. Questions will be taken from leetcode (free questions), we will keep the sample size of our questions small as we are in the minimal viable product stage. The questions will be categorized based on difficulty (e.g easy, medium, hard).

2. Functional Requirements

2.1. Account Management

ID	Functional Requirement	Priority
AM-1	The user management microservice shall allow the creation of accounts using a ".edu" email address and a password.	High
AM-2	The user management microservice shall allow users to login to the website with their registered email and password.	High
AM-3	The user management microservice shall allow users to view their profile, which consists of information about previous mock interviews (time started, time completed, interview partner details) and their user ELO (based on ratings and number of interviews completed successfully).	Medium
AM-4	The user management system will allow users to view the past 10 mock interviews on their profile.	Medium
AM-5	The user management microservice shall allow users to request interviews with past mock interview partners via their profile.	Low

2.2. Matchmaking System

ID	Functional Requirement	Priority
MS-1	The matchmaking microservice will assign every user with a base ELO of 1000	High

MS-2	Each user's ELO is based on 1-5 stars, determined by your peer review after the mock interview	High
MS-3	The matchmaking microservice will modify every user's ELO based on the following metrics: 1 star = -10, 2 stars = -5, 3 stars = 0, 4 stars = 5, 5 stars = 10.	High
MS-4	The matchmaking microservice will match the user with peers around their ELO (± 100).	High

2.3. Mock Interview Process

ID	Functional Requirement	Priority
MIP-1	The chat microservice will allow users to communicate with each other in near real-time during the mock interview.	High
MIP-2	The editor microservice will allow both users to view and edit the collaborative notepad in near real-time during the mock interview.	High
MIP-3	When the mock interview is over, the application UI will automatically show a pop up screen for both users to rate each other based on 5 stars.	High

2.4. Application Client

ID	Functional Requirement	Priority
AC-1	The application UI will serve a landing page to introduce potential users to the purpose of our application.	Medium
AC-2	The application will limit the time for users to respond to interview requests to 2 minutes.	Low
AC-3	The application will be able to display who is online, from a list of past mock interview partners.	Low

3. Non-functional Requirements (NFRs)

Our NFRs are listed in order of priority:

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

We prioritize security first because we believe there is a need to set a strong foundation for security from the beginning. Performance comes next because it is important for the user to have a good user experience when using our platform. We need to ensure there is minimal delay during the collaboration process. We placed availability in third place because we don't foresee a need to support other browsers/operating systems in the foreseeable future. Finally, we prioritize scalability last because we do not expect to have high traffic during our early stages of deployment.

3.1. Security Requirements

Key points

1. All microservices will only allow relevant IPs addresses (e.g frontend) to access the AWS EC2 instance (backend)
2. All microservices will be configured such that only relevant backend services should be able to access and make modifications on the databases
3. The database will store user passwords encrypted with SHA-256 hashing algorithm

Observatory
moz://a

[Home](#) [FAQ](#) [Statistics](#) [About](#) ▾

HTTP Observatory

TLS Observatory

SSH Observatory

Third-party Tests

Scan Summary



Host:	peerprep.live
Scan ID #:	16331384
Start Time:	November 10, 2020 6:48 PM
Duration:	9 seconds
Score:	105/100
Tests Passed:	11/11

Recommendation

Initiate Rescan

You're on the home stretch!

The use of Referrer Policy can help protect the privacy of your users by restricting the information that browsers provide when accessing resources kept on other sites.

- [Mozilla Web Security Guidelines \(Referrer Policy\)](#)

Once you've successfully completed your change, click Initiate Rescan for the next piece of advice.

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

Figure 3.1.1. Mozilla Observatory Security Test Score for peerprep.live.

Security View

No.	Asset/Asset Group	Potential Threat/ Vulnerability Pair	Possible Mitigation Controls
1	Data	SQL Injection / Unsanitised user input (integrity, confidentiality)	Implemented: User input sanitisation.
2	Data	Man-in-the-middle Attacks / Packers (integrity, confidentiality)	Implemented: HTTPS enforced with HSTS header
3	Services	Distributed Denial of service (DDOS) / Application (availability)	Implemented: Lambda caching script protects the service from DDOs (See Figure 3.1.2)
5	Data	Click-jacking/User interface (integrity, confidentiality)	Implemented: X-Frame-Options was set to deny

6	Data	Unauthorized access to backend services/ False service access (integrity, confidentiality)	Implemented: Security groups to only allow access from the ip address of known services.
---	------	--	---

Lambda Edge Functions: Edge functions are used to apply content security policy(CSP) headers to viewer response before they receive the web files at the client side.

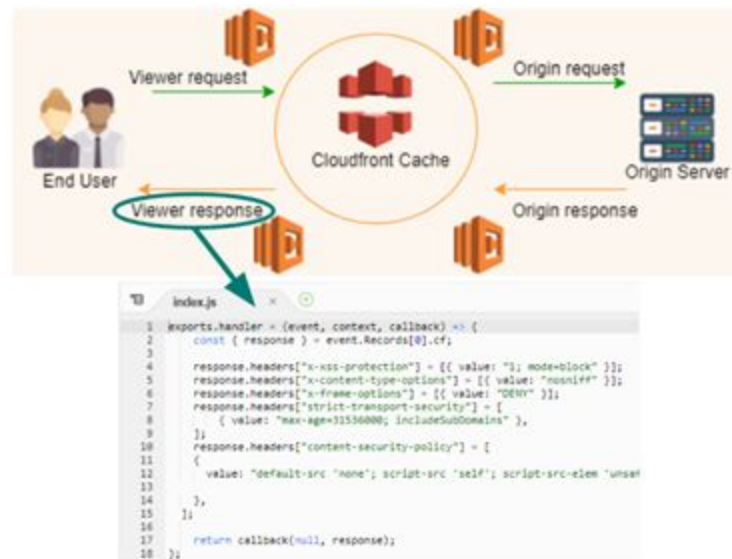


Figure 3.1.2: Implantation of Content Security Policy via Lambda Edge Functions

3.2. Performance Requirements

Key points

1. The application will ensure at most 3 seconds response delay during live coding when there is a change in the content in the collaborative notepad.
2. The application will not take more than 5 seconds to change from one screen to another.
3. API calls from frontend to backend should have a response time of <3s during peak periods

Performance View

No.	Description of the Strategy	Justification	Performance Testing (Optional)
1	API Load Testing	To test auto-scaling and overall performance of the system depending on users.	Refer to figure 3.2.2 below for k6 testing results.
2	Use of cloudfront to serve static sites worldwide.	Cloudfront caches the content to its edges allowing super fast downloads of the static site. This allows users from all over the world	

		to access the cache files. Refer to figure 11 below for its implementation.	
3	Use of redis cache to reduce load times of application	Redis can serve as a cache to cache the latest images of someone you follow. Caching that new information serves to reduce the load times.	
4	Internal calls between microservices are traversed through the internal application load balancer	Reduces round trip time, and therefore increases performance (call speed between microservices)	

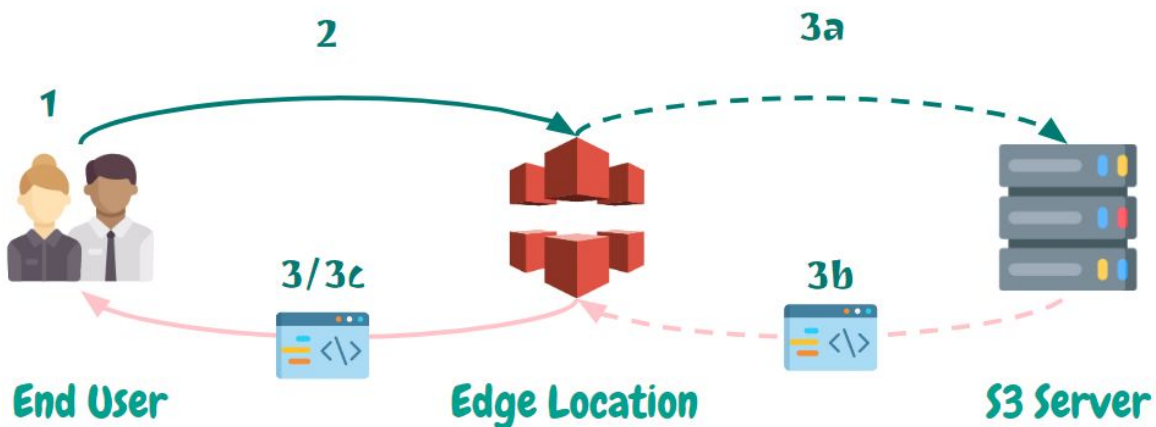


Figure 3.2.1: Achieving Faster Load Times with Cloudfront Caching

Edge Caching: By caching files within the edge locations, users would be able to retrieve the files from a closer location without needing to access the actual S3 Bucket server. If files are found within cache, steps 3a - 3c are not required.

```
running (2m30.6s), 00/10 VUs, 1084 complete and 0 interrupted iterations
default ✓ [=====] 00/10 VUs  2m30s9

data_received.....: 570 kB 3.8 kB/s
data_sent.....: 100 kB 662 B/s
http_req_blocked.....: avg=170.66µs min=0s med=0s max=45.99ms p(90)=0s p(95)=0s
http_req_connecting.....: avg=37.82µs min=0s med=0s max=4.99ms p(90)=0s p(95)=0s
http_req_duration.....: avg=254.86ms min=152.02ms med=213.86ms max=1.01s p(90)=360.99ms p(95)=409.89ms
http_req_receiving.....: avg=155.34µs min=0s med=0s max=2.99ms p(90)=999.5µs p(95)=1ms
http_req_sending.....: avg=3.69µs min=0s med=0s max=1ms p(90)=0s p(95)=0s
http_req_tls_handshaking...: avg=121.76µs min=0s med=0s max=30.99ms p(90)=0s p(95)=0s
http_req_waiting.....: avg=254.7ms min=152.02ms med=213.86ms max=1.01s p(90)=360.99ms p(95)=409.25ms
http_reqs.....: 1084 7.197678/s
iteration_duration.....: avg=1.25s min=1.15s med=1.21s max=2.01s p(90)=1.36s p(95)=1.41s
iterations.....: 1084 7.197678/s
vus.....: 1 min=1 max=10
vus_max.....: 10 min=10 max=10
```

Figure 3.2.2: Sample k6 Load Test

k6 Load Test: We used k6 for load testing, the configuration used was 50 users for 2 minutes. The result showed that the average http duration was 254.86ms, with a load of 100 users mainly without redis cache. Due to an issue where the redis cache could not keep up with populating new data for the interview sessions.

3.3 Availability Requirements

Key points

1. Ingress conducts load balancing to ensure that all backend container services remain at 50% capacity
2. Ensure an uptime of 99.5% during peak hours (Monday to Friday 8am-5pm)
3. The system will promise full support for Chrome browsers, which holds 65.99% market share.
4. The system will be available on both MacOS and Windows operating systems.

Availability View

Node	Redundancy	Clustering			Replication (if applicable)			
		Node Config.	Failure Detection	Failover	Repl. Type	Session State Storage	DB Repl. Config	Repl. Mode
Peerprep Frontend	Edge caching	Active-Active	Ping	Route53	-	-	-	-
Peerprep Kubernetes Cluster	Multi-Region	Active-Passive	Ping	Route53	-	-	-	-
Peerprep Backend server	Cluster Scaling	Active-Active	Ping	Kubelet	-	Client	-	-
Users Microservices	Horizontal Pod Scaling	Active-Active	Ping	Ingres Gateway	-	Client	-	-
Match Microservice	Horizontal Pod Scaling	Active-Active	Ping	Ingres Gateway	-	Client	-	-
Editor Microservice	Horizontal Pod Scaling	Active-Active	Ping	Ingres Gateway	-	Client	-	-
Chat Microservice	Horizontal Pod Scaling	Active-Active	Ping	Ingres Gateway	-	Client	-	-
Redis Cluster	Horizontal Scaling	Active-Active	Ping	Cluster Master Node	-	-	Master-Master	Synchronous
Peerprep DB (MongoDB)	Horizontal Scaling	Active-passive	Ping	DB	DB	Client	Master-Master	Synchronous

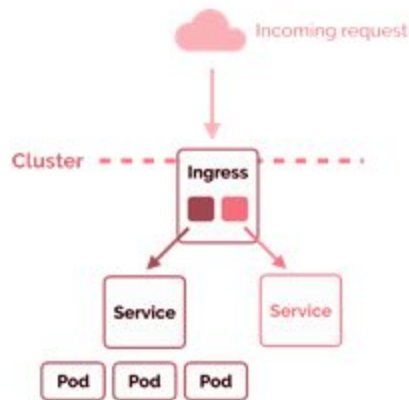


Figure 3.3.1: Amazon Route 53 Configuration

Ingress has an active-active configuration monitoring the health checks of services with no shared state

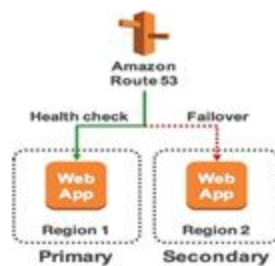


Figure 3.3.2: Amazon Route 53 Configuration

Amazon route 53 was configured to have a failover plan to route traffic to a different region when the health check of the cluster fails.

3.4 Scalability Requirements

Key points

1. The system can host 5 concurrent interviews at any point in time
2. The system should support >100 concurrent users browsing PP

Scalability view

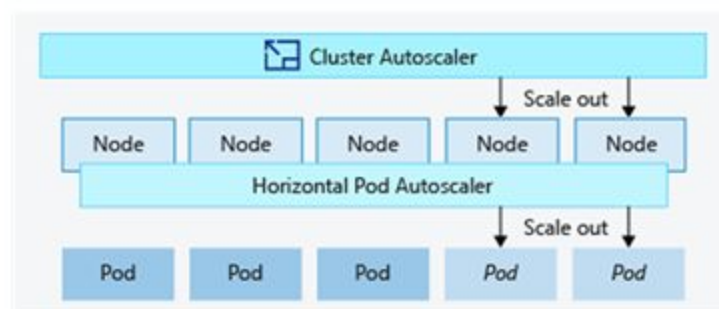


Figure 3.4.1: Autoscaling Configuration

The kubernetes cluster is set to have a minimum of 2 nodes and max of 5 nodes. It scales based on the cpu and capacity of the pods. The pods have a custom upscaling and downscaling, the downscaling policy has a 15 second grace period before downscaling.

4. Architectural Design

4.1 Architecture Diagram

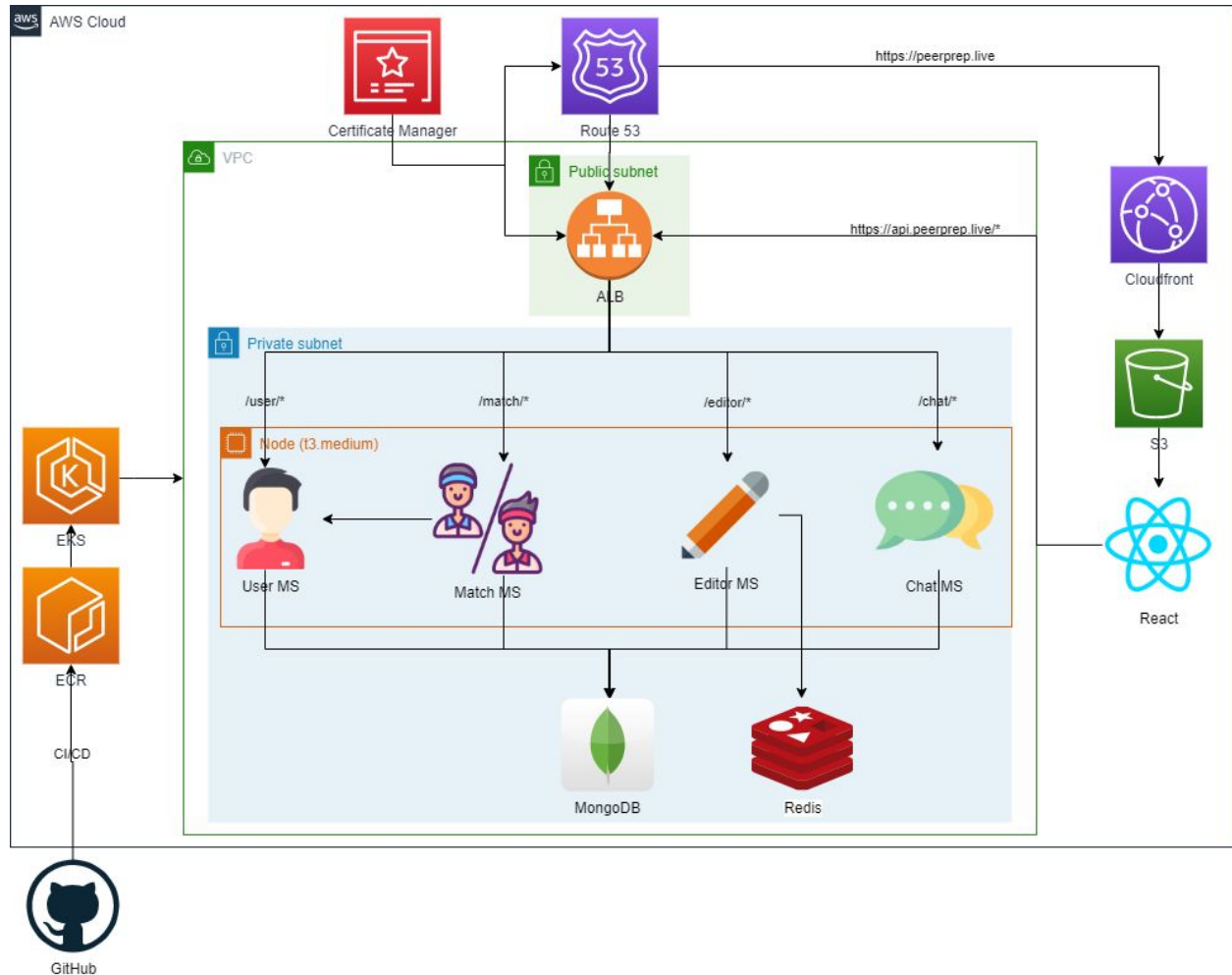


Figure 4.1: Architecture Diagram

1. Route 53 gets inbound traffic from end users. SSL certificate is used to authorize http requests.
2. When requests hit backend services, they will go through [https://api.peerprep/live](https://api.peerprep.live), which is directed to the application load balancer in the public subnet in the Virtual Private Cloud (VPC).

3. The application load balancer acts as the Ingress for the microservices and direct traffic to the relevant services. *Eg. If end user wants to see user microservice, it will redirect user to /user services, then traffic will be pushed to user microservice ports.*
4. Internal calls between microservices are traversed through the internal application load balancer and does not exit the private subnet. *Eg. user calls match*
<http://match-service.ipp.svc.cluster.local:5000/match/create>
5. Our Data layer consists of mongoDb and S3, which are also kept within the private subnet.

4.2 Architectural Decisions

Architectural Decision: Microservice Architecture	
Issue	<ol style="list-style-type: none">1. Avoid complexity in testing and development in a Monolithic Application2. Tightly coupled modules3. Difficulty in scaling individual functions4. Difficulty in scaling monolithic database
Architectural Decision	<p>A Microservice Architecture allows us to break a large application into loosely coupled modules that communicate through APIs. Kubernetes enables scaling of individual microservices based on at least 75% usage. Moreover, Cluster Autoscaling allows the system to scale the number of nodes when there is an influx of the number of pods.</p> <p>This allows separate team members to develop and test each microservice without any dependencies on another, and hence ensures continuous integration and deployment of individual services.</p>
Assumptions	None
Alternatives	Monolithic Application
Justification	It allows ease of autoscaling of microservices, which ensures that deployment of a single function does not conflict with others. The Microservice can be split into different responsibilities for different developers, which will speed up the development.

Architectural Decision: Kubernetes Cluster	
Issue	<ol style="list-style-type: none">1. Hard to manage services running in multiple EC2 instances2. Expensive to maintain a lot of EC2 instances.

Architectural Decision	<p>It is a cloud native techstack which utilises the infrastructure as a code process to allow easy maintainability of the architecture.</p> <p>It allows for easy deployment, auto scaling of backend microservices and monitoring of these services.</p> <p>Kubernetes is open source and is not susceptible to vendor lock in allowing us to change cloud providers or even convert to on-premise easily.</p>
Assumptions	None
Alternatives	Docker swarm
Justification	<p>AWS has EKS which is a kubernetes service managed by AWS whereas there isn't a product for docker swarm in any of the cloud providers which means having to provision EC2 instances manually. Giving the management of the container orientation service to the cloud provider means we do not need to manually configure the provisioning of the service.</p>

Architectural Decision	
Issue	<ol style="list-style-type: none"> 1. Complexity of the Underlying System Code 2. Security (Multiple Entry Points) 3. Manually redirect traffic to pods 4. No HTTPS without ingres
Architectural Decision	<p>The Ingress gateway sits in front of the Pods (API) and is the single point of entry for all the microservices.</p> <p>Having a single entry point provides benefits such as:</p> <ol style="list-style-type: none"> 1. Smaller attack surface, effectively making the application more secure 2. The ingress gateway allows sharing of common functionalities such as monitoring across all APIS. 3. Any changes made to the microservices will require only changing configuration of ingress gateway reducing the need to change multiple lines of codes in other microservices or frontend.
Assumptions	None
Alternatives	Direct client-to-backend/microservice communication, AWS API Gateway

Justification	<p>The ingress gateway handles the redirection of traffic to the correct pods through labels, eliminating the need to manually redirect traffic to the right pods.</p> <p>It also handles internal routing between microservices without the need for internet and requires HTTPS certificate from internet traffic before routing to relevant pods enforcing security.</p> <p>Easily to design for different client applications such as mobile or web applications.</p>
---------------	---

Architectural Decision: Redis Caching	
Issue	<ol style="list-style-type: none"> 1. For the API endpoint GET questions, the Editor Microservice requires the <code>session_id</code> and <code>question_id</code> to be stored to prevent generating different questions for the users in the same session. 2. This endpoint is frequently used, which slows down the process and results in poor performance.
Architectural Decision	<p>Storing the <code>session_id</code> and <code>question_id</code> pair in a Redis Caching layer instead of a database allows us to reduce the latency of pulling data from the database.</p> <p>In addition, caching this information is suitable due to the short-term nature of interview sessions.</p>
Assumptions	Data stored within the redis cache is small enough to improve speeds when called.
Alternatives	Memcached
Justification	<p>Without a caching layer, the average latency for getting posts is roughly 100-300ms (for a small dataset). With Redis Caching, the time is reduced to 10-100ms. This ensures minimum latency for this endpoint, which is frequently used.</p> <p>Redis supports persistence unlike memcached, with a point-in-time snapshot of all the datasets allowing the data to be restored on startup, in case of system failure.</p>

5. Design Patterns

5.1. Builder pattern

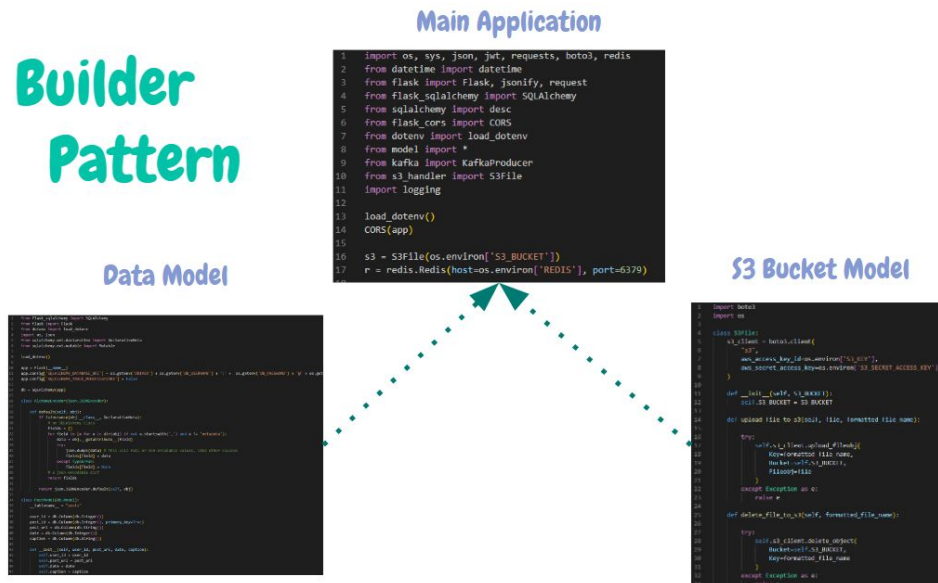


Figure 5.1.1: Builder pattern used in code

With the builder pattern in mind, the team abstract the intricacy of the data models and S3 API to their own builder classes to handle the creation of the object.

5.2. MVC pattern

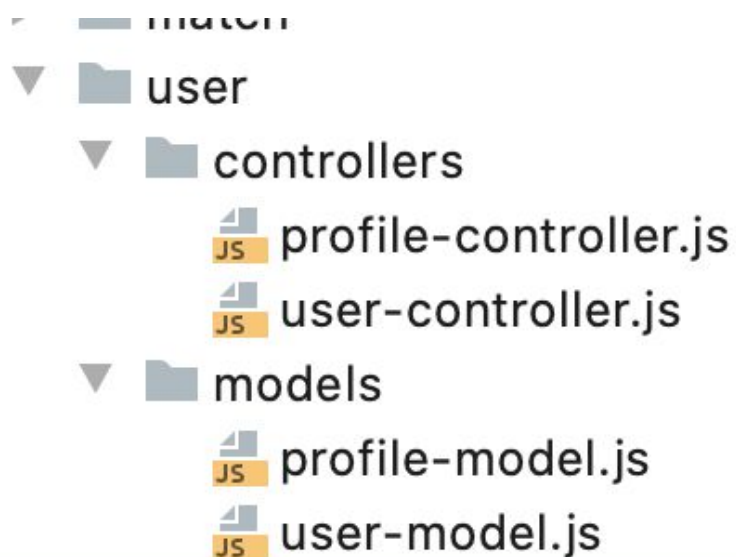


Figure 5.2.1: Relevant user microservice file directory

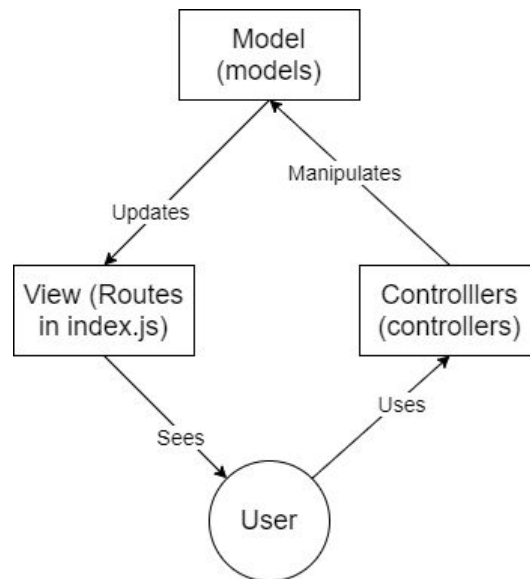


Figure 5.2.2: MVC pattern used in code

We employed the MVC pattern for the user microservice for better separation of concerns between handling user authentication flow and handling user profile information.

5.3 Pub-Sub Pattern

Our Chat and Editor Microservice uses Socket.io, which is an implementation of the pub-sub pattern in order to enable real-time communication between the two users in the interview session. Refer to Section 7.3 for more details.

6. DevOps

6.1. Sprint Process

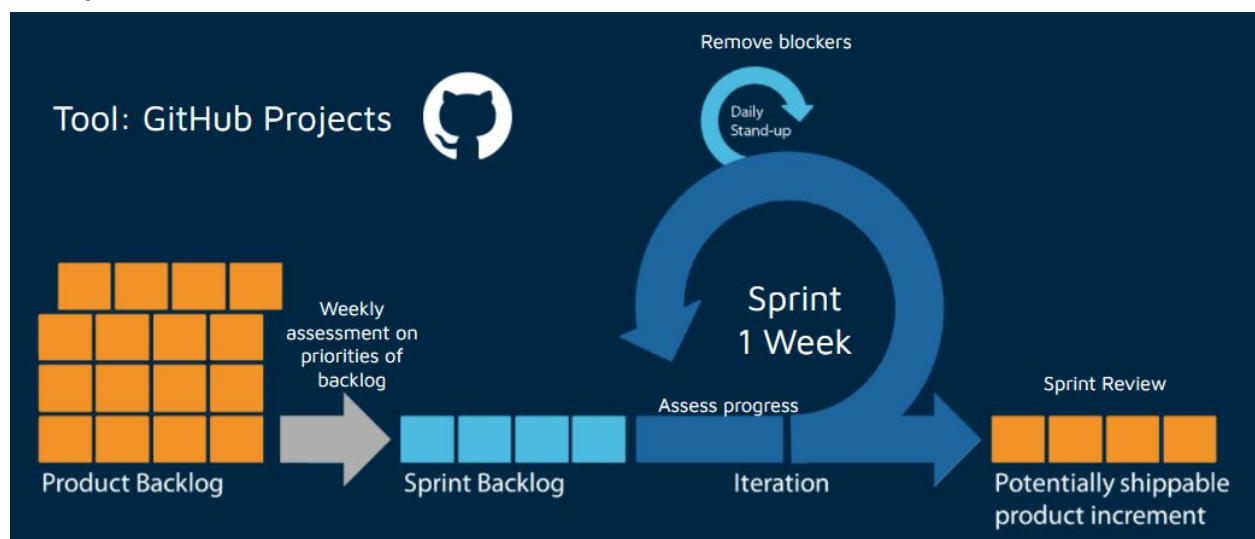


Fig 6.1.1 Sprint Process

We conduct Daily Standups on Zoom to achieve the following goals:

1. Resolve blockers
2. Update on all team members' progress



Fig 6.1.2 Group picture after a daily standup

We will conduct weekly sprint planning on the product and sprint backlog:

1. Reassess any sprint backlogs from previous sprint to be put into product backlog or next sprint
2. Weekly deployments into kubernetes cluster
3. Conduct regression testing on new features to see if any old features are broken

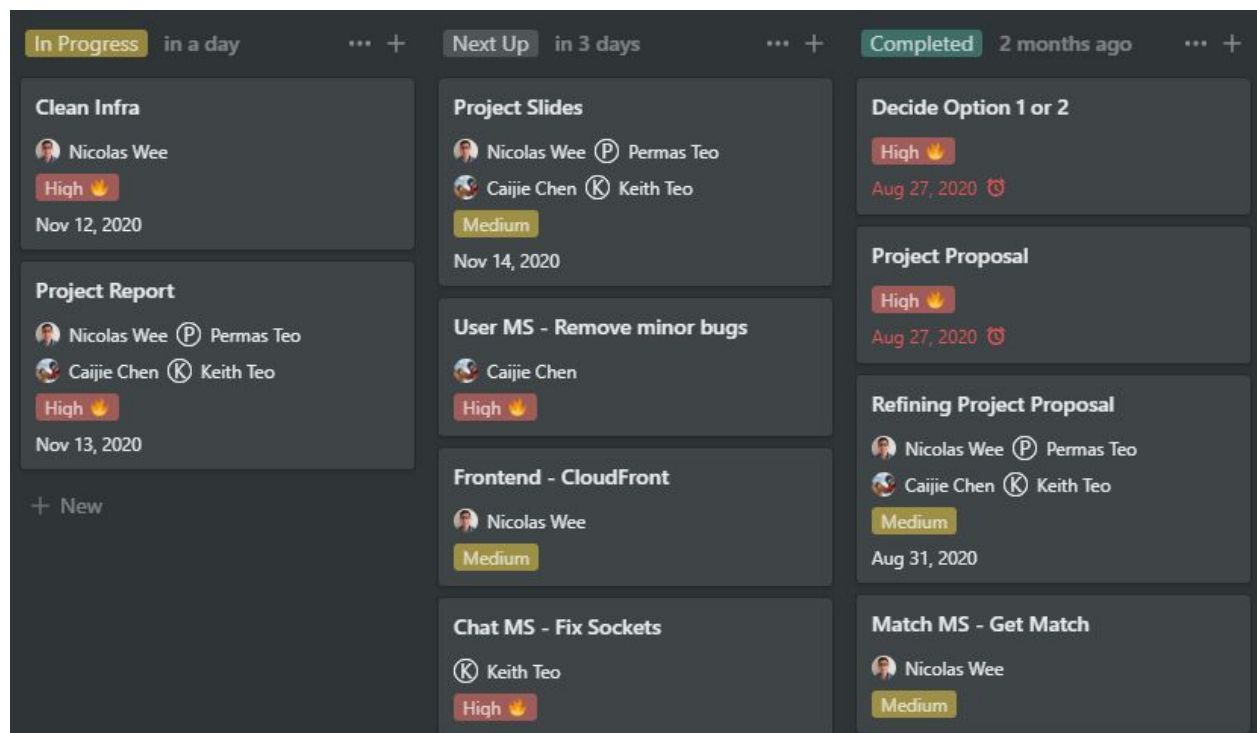


Fig 6.1.3 Sample Weekly Sprint Planning

We conduct sprint reviews at the end of each week (sample in Appendix A):

1. Identify areas of improvements for team members
2. Work on each team member's strengths by assigning task that they are most comfortable in

6.2. CI/CD

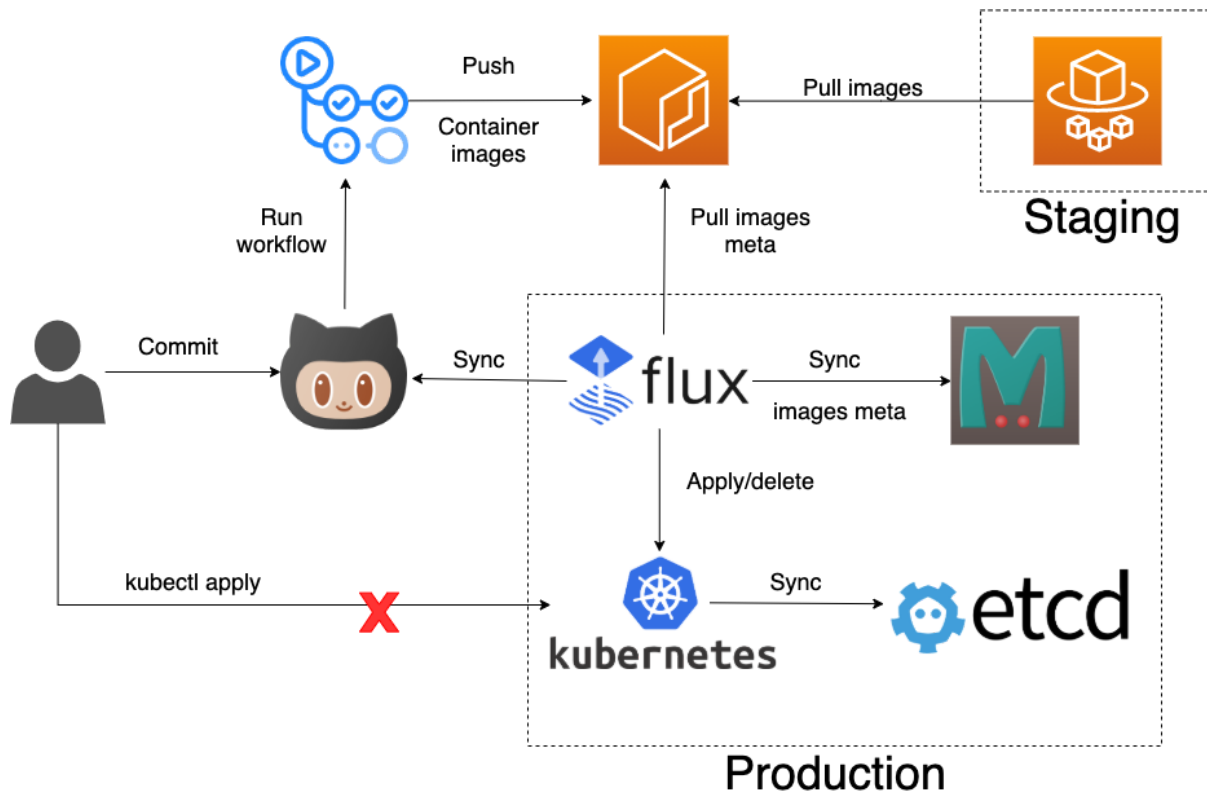
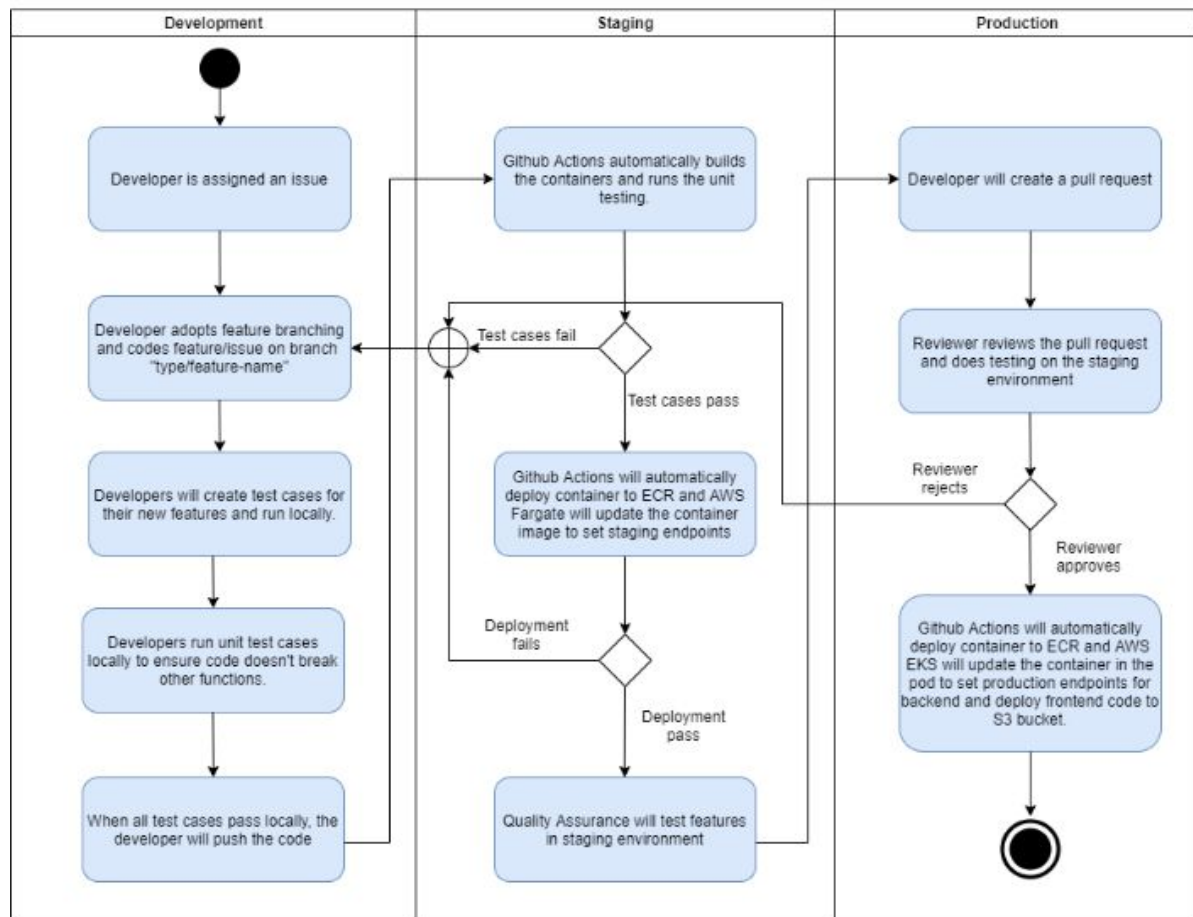


Figure 6.2.1: Continuous Deployment Workflow for Backend

Backend CI/CD: Code gets pushed and a Github Actions is activated. The workflow will run unit test cases before pushing the image into ECR. When a pull request is closed, Flux will keep track of new images in ECR and pull latest images to apply to kubernetes, whilst syncing the manifest in github.

Frontend CI/CD: CI/CD pipeline consists of Travis. When change is made to the frontend, Travis will automatically run all the tests. Once the tests pass, Travis will build the React application and upload the build folder containing the project artifacts to the designated Amazon S3 Bucket.

6.3. Manual Deployment

Deploying Frontend:

1. `cd frontend` then `yarn build` to create a production build folder
2. Login to AWS console and go to S3 service
3. Click on “peerprep.live” bucket
4. Click the orange “Upload” button
5. Go to the build folder than was created in Step 1 and drag all the files to the page.
6. Scroll down and click upload.
7. You’re all set!

Deploying Backend microservices:

Run `kubectl apply -f kubernetes/manifest/{name}/{name}_deployment.yaml`

Replace `{name}` with the microservice name (chat, editor, user, match)

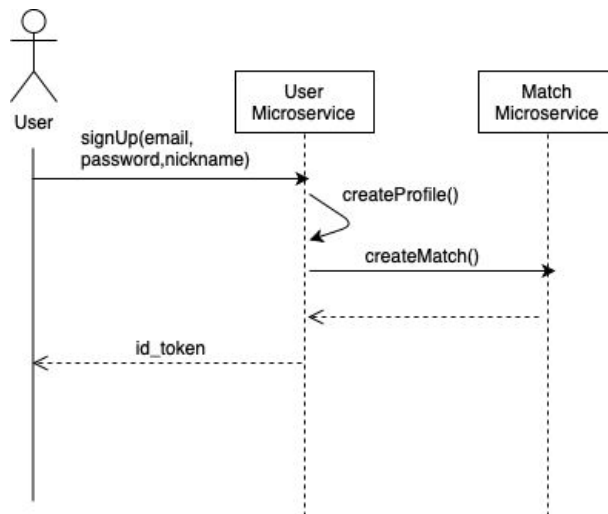
7. Microservices

7.1. User

The User Microservice handles logging in, signing up and updating the user profile. It is built with Express and Node.js. It interacts with the Match Microservice by creating a match object unique to each user to handle matchmaking when the user first signs up.

Host URL: <https://api.peerprep.live/user>

Sequence Diagram demonstrating sign up:



Currently supported endpoints:

New User

<https://api.peerprep.live/user/user> - HTTP POST Request

POST body format: JSON

Example body:

```
{
  "email": "test@gmail.com",
  "password": "test",
  "nickname": "test"
}
```

Sample success response: JSON

```
{
  "status": "success",
  "message": "New user created!",
  "data": {
    "_id": "5faaa985014de0718594954e", [user id token]
    "email": "test@gmail.com",
    "password": "test", [we only return the password for signup]
    "nickname": "test",
    "create_date": "2020-11-10T14:45:03.954Z",
    "__v": 0
  }
}
```

Login

<https://api.peerprep.live/user/login/{email}> - HTTP POST Request

To login, include the password as part of the POST body.

POST body format: JSON

Example body for uri <https://api.peerprep.live/user/login/test@gmail.com>:

```
{
  "password": "test"
}
```

Sample success response: JSON

```
{
  "status": "success",
  "message": "Successfully logged in",
  "data": {
    "_id": "5faaa985014de0718594954e", [user id token]
    "email": "test@gmail.com",
    "nickname": "test"
  }
}
```

Start Interview

https://api.peerprep.live/user/profile/interview/{id_token} - HTTP POST Request

This endpoint starts an interview session

POST body format: JSON

Example body for uri <https://api.peerprep.live/user/profile/interview/5faaa985014de0718594954e>:

```
{
  "partner_nickname": "bob"
}
```

Sample success response: JSON

```
{
  "status": "success",
  "message": "New interview started!",
  "data": {
    "_id": "5faaaa7f014de017e7949550", [interview id]
    "partner_nickname": "bob",
    "start": "2020-11-10T14:58:07.219Z"
  }
}
```

End Interview

https://api.peerprep.live/user/profile/interview/{id_token} - HTTP PUT Request

This endpoint ends the interview session

PUT body format: JSON

Example body for uri <https://api.peerprep.live/user/profile/interview/5faaa985014de0718594954e>:

```
{
  "interview_id": "5faaaa7f014de017e7949550"
}
```


Sample success response: JSON

```
{
  "status": "success",
  "message": "Interview completed!",
  "data": {
    "_id": "5faaa985014de02a9e94954f", [Identifies the profile and not the user!]
    "user_id": "5faaa985014de0718594954e", [This is equivalent to the id token]
    "interviews": [
      {
        "_id": "5faaaa7f014de017e7949550", [interview id]
        "partner_nickname": "bob",
        "start": "2020-11-10T14:58:07.219Z"
      }
    ],
    "__v": 1
  }
}
```

Get Profile

https://api.peerprep.live/user/profile/{id_token} - HTTP GET Request

This endpoint retrieves a user's profile information, including their past interview sessions

Example uri to retrieve profile information for user with id token "5faaa985014de0718594954e":

<https://api.peerprep.live/user/profile/5faaa985014de0718594954e>

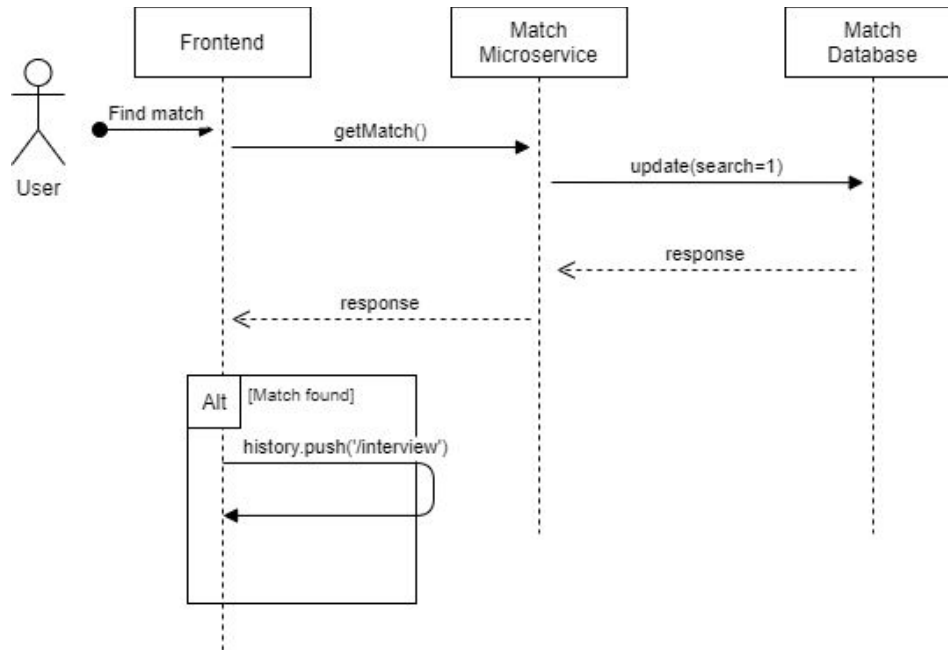
Sample success response: JSON

```
{
  "status": "success",
  "message": "Profile details retrieved successfully!",
  "data": {
    "_id": "5faaa985014de02a9e94954f", [Identifies the profile, not the user!]
    "user_id": "5faaa985014de0718594954e", [same as user id token]
    "interviews": [
      {
        "end": "2020-11-10T15:07:52.812Z",
        "_id": "5faaaa7f014de017e7949550", [interview id]
        "partner_nickname": "bob",
        "start": "2020-11-10T14:58:07.219Z"
      }
    ],
    "__v": 1
  }
}
```

7.2. Match

The Match Microservice stores every user's elo and handles matching of users based on their elo value.

Sequence Diagram demonstrating getMatch call:



Currently supported endpoints:

Get Match

<https://api.peerprep.live/match/get?email=test@gmail.com> - HTTP GET Request

Find a match for the user within the user's elo ($-100 < \text{User's elo} < +100$) and returns the partner's details.

Sample success response: JSON

```
{
  "elo": 1000,
  "email": "test2@gmail.com",
  "nickname": "bobbytest",
  "status": true
}
```

Update Elo

<https://api.peerprep.live/match/update?email=test@gmail.com&elo=1010> - HTTP POST Request

Update the elo after the interview is over and to end the interview session.

Sample success response: JSON

```
{
  "message": "Successfully updated elo",
  "status": true
}
```

Create User

<https://api.peerprep.live/match/create?email=test3@gmail.com&nickname=rbth7e5> - HTTP POST

Request

Create a user in match DB when there is a new user created in the user DB.

Sample success response: JSON

```
{
  "message": "Successfully created",
  "status": true
}
```

Get User

<https://api.peerprep.live/match/user?email=test@gmail.com> - HTTP GET Request

Get match details of a specific user.

Sample success response: JSON

```
{
  "elo": 1000,
  "email": "test2@gmail.com",
  "nickname": "bobbytest",
  "status": true
}
```

7.3. Real-time Communication Microservices

The Chat and Editor Microservices use Socket.io to enable real time communication between the two users in an interview session.

Each communication channel for each interview session is identified by a `session_id`. This `session_id` is generated by the frontend using the unique identifiers for both users participating in the interview session.

On the client, it listens to the socket using this generated `session_id`. Since this `session_id` is unique to each interview session and generated on the fly from unique identifiers of the participating users, no third party can listen in to this channel.

The communication uses the publisher/subscriber message design pattern to send the correct data to the relevant parties as shown in the diagram below.

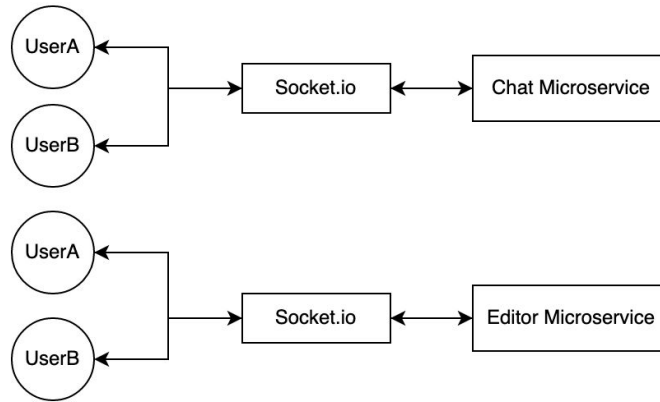


Figure 7.3.1. Publisher/subscriber Pattern

In this case, UserA, UserB, Chat and Editor microservices act as both publishers and subscribers to the `session_id` generated by UserA and UserB. When UserA sends a message, it will include the `session_id` and the message will be passed to the respective microservice, which then will publish it to the respective microservices via Socket.io. Each of the microservices (Chat or Editor) uses the `session_id` to route the received message to the participants (UserA and UserB).

Suppose there is another UserC, since UserC is not aware of the unique identifiers of UserA and UserB, it will not be able to generate the `session_id` required to receive messages belonging to the interview session between UserA and User B.

7.3.1. Chat

In addition to the Socket.io protocol, we also added a nifty feature that enables users to send `/end_session` as a message to end the interview session for both users.

```

chatSocket.on(sessionId, (message) => {
  if (message.msg === '/end_session') {
    if (message.sender !== user.nickname) {
      setBuddyEndedMsg('Your buddy has ended the session!');
    }
    setShow(true);
  } else {
    setMessages((oldMessages) => [...oldMessages, message]);
    const msgContainer = document.getElementById('chat-message-container');
    msgContainer.scrollTo(0, msgContainer.scrollHeight);
  }
});

```

The client can also emit new messages as follows:

```

chatSocket.emit('newMessage', {
  sessionId,
  payload: {
    sender: user.nickname,
    msg: '/end_session',
  },
});

```

Internally, in the Chat Microservice, it routes every message received using the `session_id` and broadcasts it to everyone listening to the socket using the same `session_id`.

```
io.on('connection', socket => {  
  socket.on('newMessage', msg => {  
    io.emit(msg.sessionId, msg.payload);  
  });  
});
```

7.3.2. Editor

While the Editor Microservice uses the same protocol to publish and subscribe to messages, the biggest difference from Chat Microservice is that the frequency of messaging is much higher than that of Chat's, as the Collaborating Notebook will constantly be updated on changes.

In addition, on first connect, the Editor Microservice will send the question based on the following protocol:

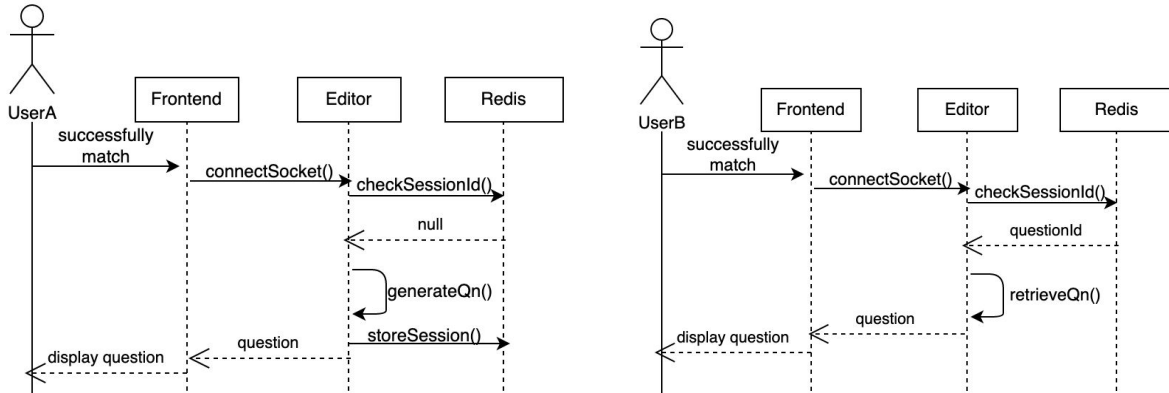


Figure 7.3.2: Sequence diagram after UserA and UserB matches and connects

From the figure above, we assume that UserA is the first person to connect to the Editor Microservice, i.e. UserA will initialise the session using the shared `session_id` which will then generate a random question based on difficulty. After UserB is connected to the Editor Microservice, since the `session_id` is already initialised in the Redis cache, the Editor Microservice will retrieve the unique question identifier and the respective question for UserB.

When the session is ended, UserA and UserB will notify Editor Microservice to remove the `session_id` from the Redis cache.

8. Frontend

8.1. Tech Stack

1. React
2. Redux
3. Material-UI
4. React-Bootstrap

React is a front-end development framework with a heavy focus on the concept of components. It maintains a virtual Domain Object Model (DOM) and uses a diffing algorithm to selectively update the parts of the DOM that actually change.

8.2. Redux

We use Redux, a state management tool, to store global states that need to be accessed all over the application. These global states include the match and user information.

Redux offers benefits like:

1. Keeps commonly accessed states to a centralised location so it's easier to keep track
2. Reducer functions that update the states are pure without side effects so it's easier to test
3. State updates are predictable and data flows in a single direction so it's easier to understand
4. Improves the debugging experience since you can easily log a single reducer to track how a single state changes

The following architecture diagram shows how Redux works with React at a high level.

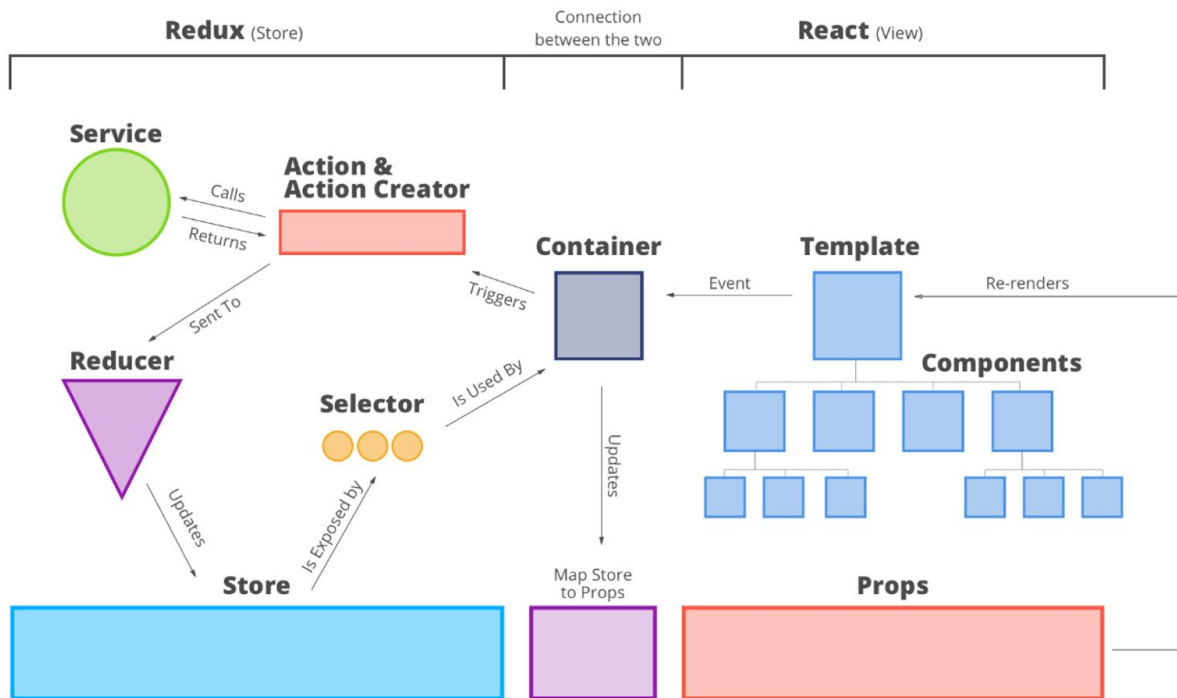
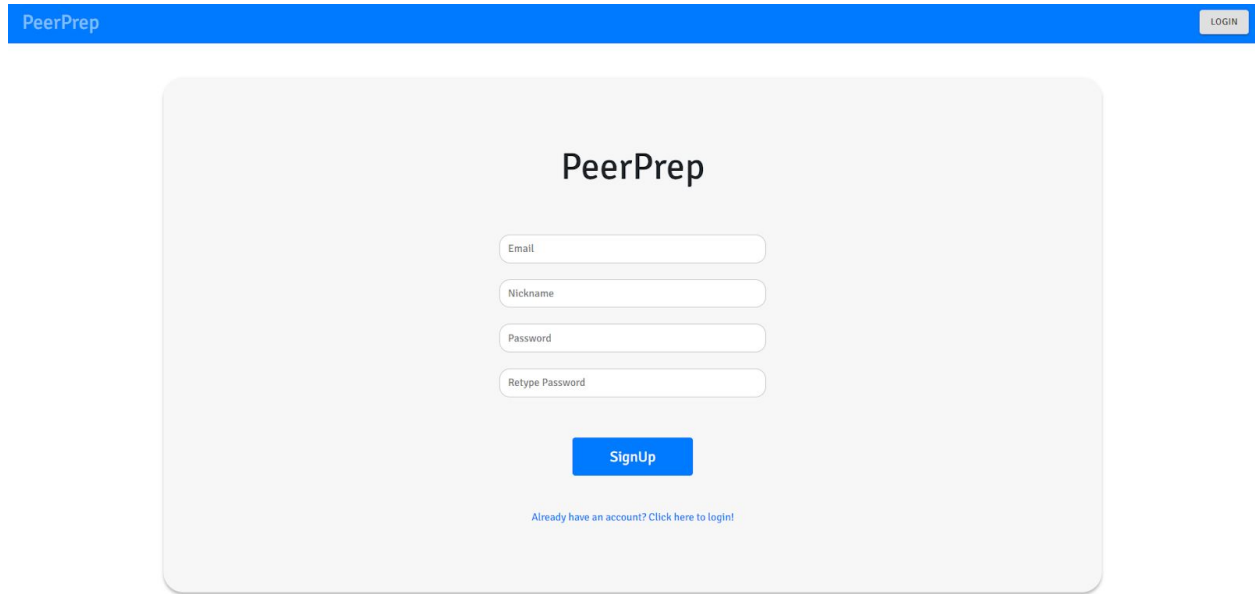


Figure 8.2.1: React + Redux Architecture Diagram

source: <https://medium.com/mofed/react-redux-architecture-overview-7b3e52004b6e>

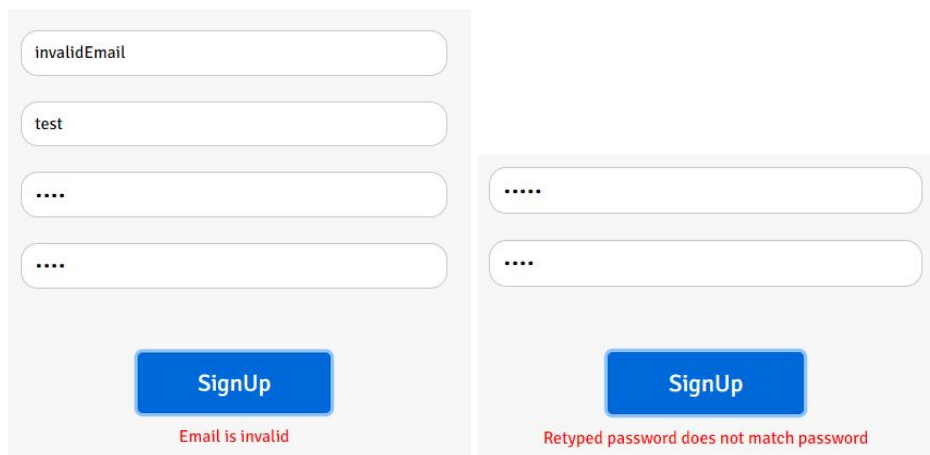
9. Application Screenshots

9.1. User Authentication



The screenshot shows the PeerPrep application's signup page. At the top, there is a blue header bar with the text "PeerPrep" on the left and a "LOGIN" button on the right. The main content area is a light gray rounded rectangle. Inside, the word "PeerPrep" is centered at the top. Below it are four input fields: "Email", "Nickname", "Password", and "Retype Password". A blue "SignUp" button is centered below these fields. At the bottom of the form, there is a link that says "Already have an account? Click here to login!".

Figure 9.1.1: Signup page



The screenshot shows two examples of validation errors on the signup page. The left example shows the "Email" field with the text "invalidEmail" and the "SignUp" button. Below the button, a red error message says "Email is invalid". The right example shows the "Password" and "Retype Password" fields, both with masked text (dots). Below the "Retype Password" field, a red error message says "Retyped password does not match password".

Figure 9.1.2: Validation sample

Client-side handles basic data validation before passing the login/sign up request to the user microservice.

9.2. User Home Page

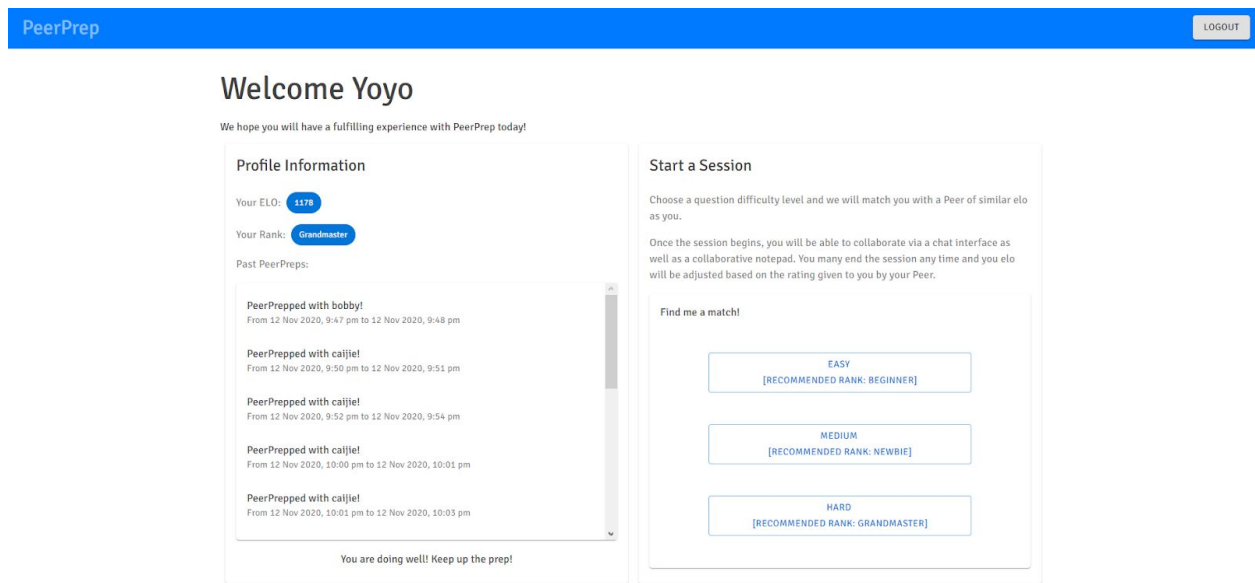
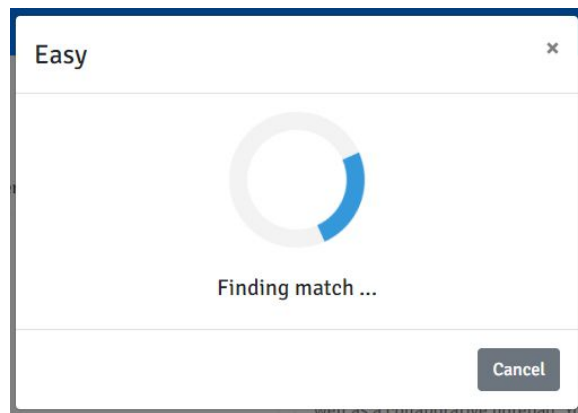
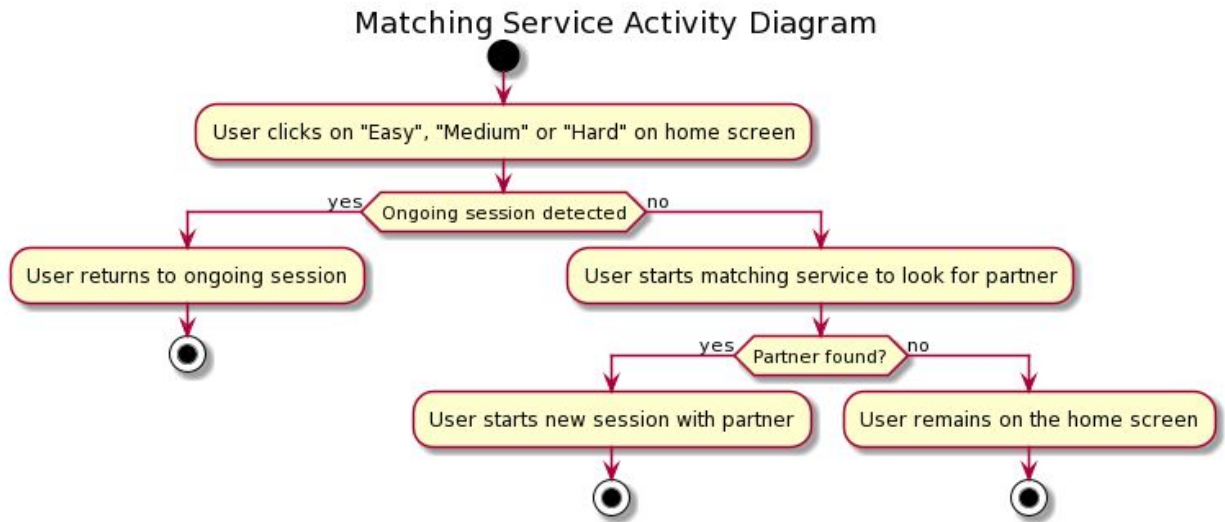


Figure 9.2.1: User Home Page

9.3. Matching System



The user will be prompted with the Modal above when they attempt to start a match. The Header of the Modal indicates the difficulty setting chosen by the user.



This is the activity diagram for the Matching System.

9.4. PeerPrep Interview

The user will proceed to work on the given question in the interface below. Sample usage of the editor and chat features are shown below.

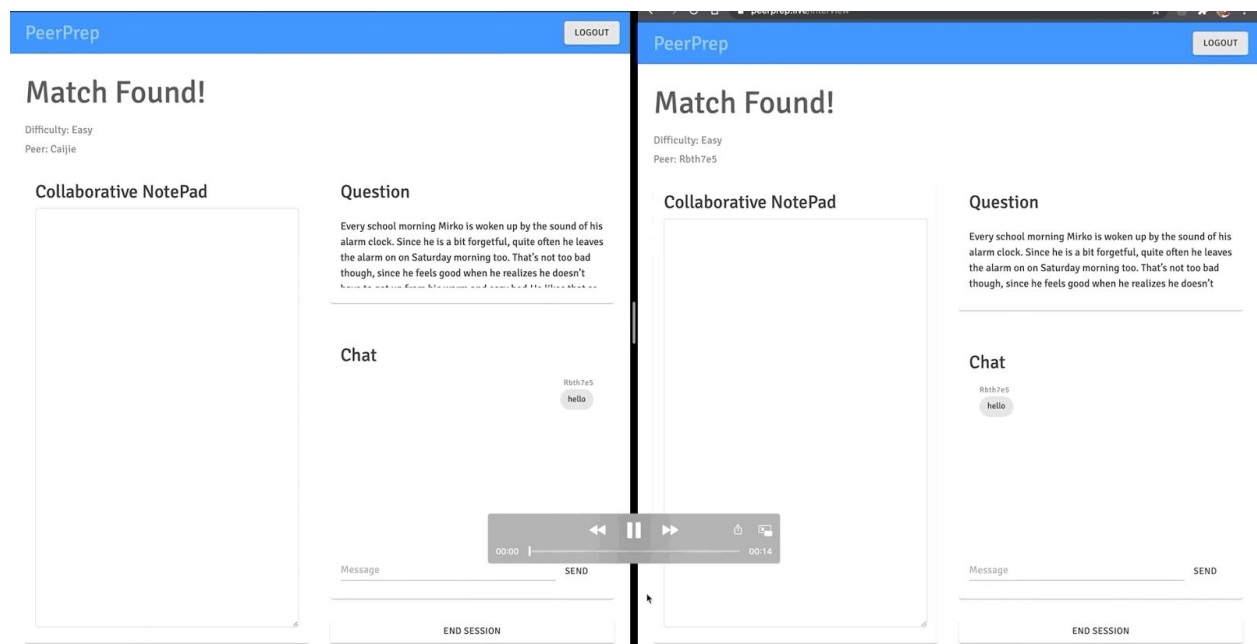


Figure 9.4.1. Demonstration of chat feature

<https://photos.app.goo.gl/vGu7c2ubzEhLwRNA6>

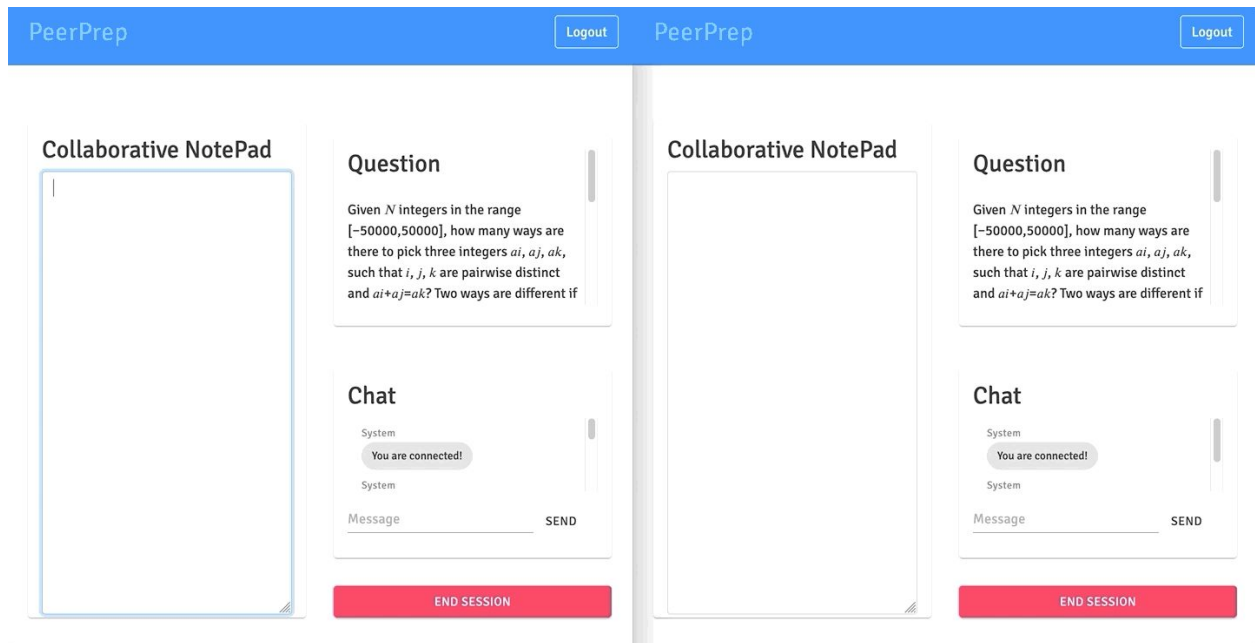


Figure 9.4.2. Demonstration of Collaborative Notepad in Real-time

<https://photos.app.goo.gl/TgaQspmvCQwZtsCt9>

9.5. PeerPrep Interview Rating

When either of the users ends the session for the interview, both peers are invited to rate each other. The modal popup is given below.

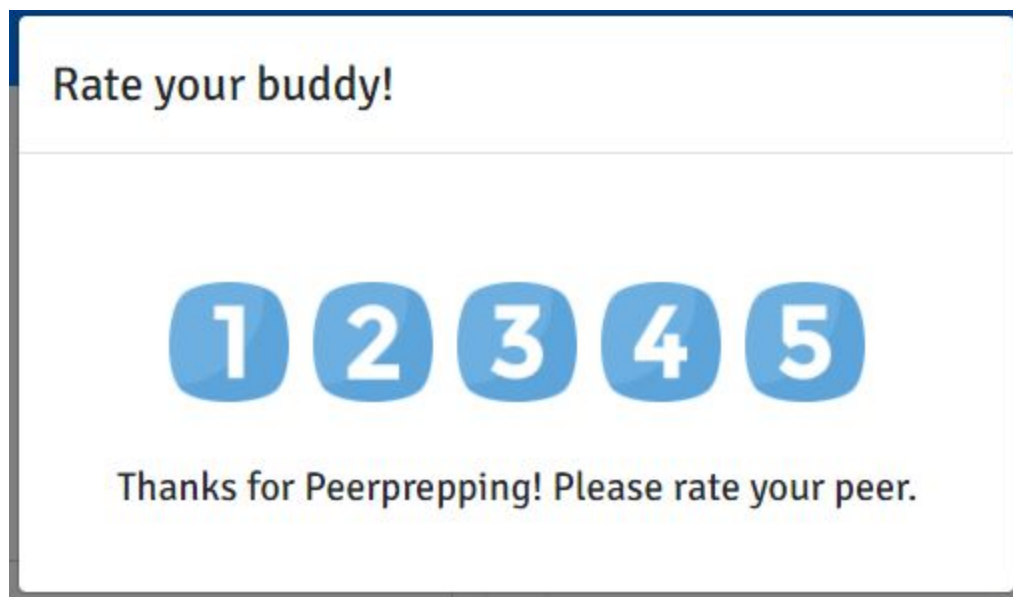


Figure 9.5.1. Demonstration of Collaborative Notepad in Real-time

A higher rating contributes to a higher elo for the other user.

ELO Computation:

Completion of a session: +10

Rating received: $+r$, where r is the rating number given by the other user

10. Remarks

10.1. Challenges Faced

- We had a hard time figuring out how to ensure real time updates for both editor and chat microservices
- Concise communication between team members working on the frontend and backend
- Configuring Continuous Deployment using Travis was frustrating because developer documentation for deploying to Amazon S3 Bucket was outdated and we had to comb through forums to figure out a solution.
- Steep learning curve for picking up Kubernetes and AWS services

10.2. Potential Extension Features

Microservices allow us to break down PeerPrep into small loosely coupled modules such as Editor and Chat microservices. This allows for extensibility in case a new feature is developed or a current microservice is replaced. Some potential extension features are:

10.2.1. Video Conferencing

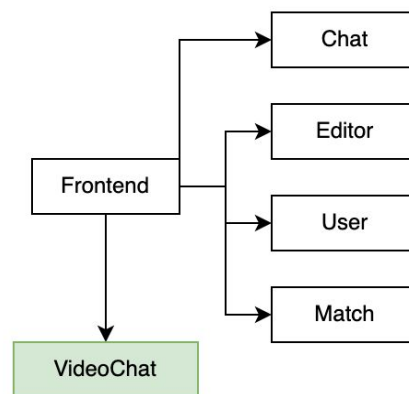
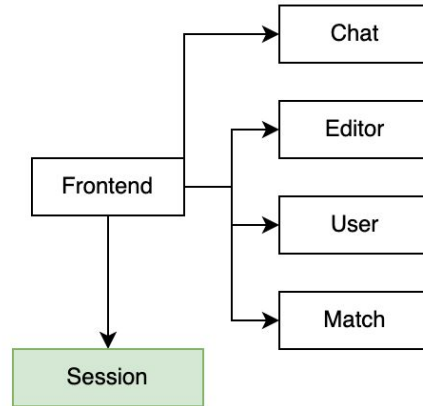


Figure 10.2.1.1. New Microservices Architecture with VideoChat Microservice

To better replicate a technical interview session, a good feature to add would be video chats.

Even though most of the PeerPrep software is built in Javascript, the use of microservices architecture allows for language agnostic development. This means that we can build a VideoChat microservice in a more performant language such as C++.

10.2.2. Store Code History



A microservice to handle session history will potentially be very useful for users to review and revise past interviews and questions.

We can do this by adding a new “session” microservice that handles storing the code and chat history. You can store each session history with the generated `session_id` as the identifier. Without modifications to existing code and services, it will then be possible to retrieve all code and chat history by generating the `session_id` with the unique identifiers of both users.

Appendix A: Sample Meeting Minutes

Date: 19/09/2020
Time: 2200
Venue: Zoom

Attendees: Caijia, Keith, Nicolas, Permas

Agenda: **Weekly standup**
1. Align schedule for team meeting
 - every Monday
2. How Agile and sprints work
 - Sprint: every week
3. How to set up MongoDB
4. Task delegation

No.	Description	Done by	Due Date
1	Set up React	Keith	26 Sept
2	Design frontend	Permas	26 Sept
3	Set up user microservice	Caijie	26 Sept
4	Set up infrastructure	Nicolas	26 Sept

The meeting was adjourned at 12.10 am. These minutes will be circulated and adopted if there are no amendments reported in the next three days.

Prepared by,
Nicolas

Vetted and edited by,
Keith