CS3219

Software Engineering Principles and Patterns

Group 28 Final Report: NUSociaLife

Chee Erynne	A0205455J
Lim Jean Tong Rachel	A0205760L
Marcus Lee Eugene	A0202061A
Tay Kai Xiang	A0200236Y

Website:

https://www.nusocialife.net/

Code Repository URL:

https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g28

Table of Contents

1. Contributions

1.1 Individual Contributions

Chee Erynne	**Report** - [2] Introduction - [3] Functional Requirements - [4] Non-Functional Requirements - [5] Quality Attributes Prioritization Matrix - [7.2] Overall Application Architecture - [7.3] Frontend Architecture - [7.7] User Flow - [7.8] Application Flow - [8] Future Plans - [10] Appendix
	Code - [F1] Sign Up UI, Redux Logic - [F4] Forum (Posts) UI, Redux Logic, Integration of Backend API - [F5] Forum (Comments) UI, Redux Logic, Integration of Backend API - [F6] Forum (Functionality) UI, Redux Logic, Integration of Backend API - [F7] Profile UI
Lim Jean Tong Rachel	**Report** - [2] Introduction - [3] Functional Requirements - [4] Non-Functional Requirements - [5] Quality Attributes Prioritization Matrix - [7.2] Overall Application Architecture - [7.4] Backend Architecture - [7.5] Database Design - [8] Future Plans - [9] Reflection - [10] Appendix
	Code - [F4] Forum Microservice, Post Model, CRUD for Posts - [F5] Forum Microservice, Comment Model, CRUD for Comments - [F6] Forum Microservice, Sort Posts/Comments, Upvote/Downvote Posts/Comments - Setup MongoDB database - Mock data preparation for demo - Eslint Configuration for Backend
Marcus Lee Eugene	**Report** - [2] Introduction - [3] Functional Requirements

	- [4] Non-Functional Requirements - [5] Quality Attributes Prioritization Matrix - [6] Software Development Process - [7.1] Tech Stack - [7.2] Overall Application Architecture - [7.3] Frontend Architecture - [7.8.2, 7.8.3] UI Images for Frontend - [8] Future Plans - [10.1] Glossary
	Code - [F1] Login UI, Sign Up UI, Redux Logic, Integration of Backend API - [F2] UI for Navigation Bar, Redux Logic, Integration of Backend API - [F3] FindFriend UI, Chat UI, Chat Microservice, Redux Logic, Integration of Backend API - [F7] Profile UI, Redux Logic, Integration of Backend API - Set up Github Actions CI/CD for Frontend and Backend to AWS Amplify & AWS Cloudformation respectively - Eslint Configuration for Frontend - Github README files
Tay Kai Xiang	**Report** - [2] Introduction - [3] Functional Requirements - [4] Non-Functional Requirements - [5] Quality Attributes Prioritization Matrix - [6] Software Development Process - [7.1] Tech Stack - [7.2] Overall Application Architecture - [7.4] Backend Architecture - [7.5] Database Design - [7.6] Deployment Architecture - [7.8] Application Flow - FindFriend matching algorithm - [8] Future Plans - [10.1] Glossary
	Code - [F1] Users microservice, bcrypt for password hashing, jwt for authorisation, Nodemailer setup for email notifications, express validator for input validation, Integration with Frontend Login UI, Sign Up UI - [F3.1, F3.2, F3.3, F3.6] FindFriend microservice, Integration with Frontend FindFriend UI - [F7] Users microservice, AWS S3 Profile Image upload, Integration with Frontend Profile UI - Dockerise app for deployment - In-charge of DevOps and setup script (AWS Cloudformation, AWS CLI commands) that can automate app deployment on AWS cloud

	- Setup load testing using hey - Github README files

2. Introduction

2.1 Background

The purpose of NUSociaLife (a web application) is to allow students in NUS to meet new people, giving them the opportunity to find activity groups that they are interested in and to interact with those who share similar interests in a very fun manner.

The app consists of 2 main features - Find Friends and Forum. These 2 features allow students to interact with one another on a personal level (Find a Friend) and in a group based setting (Forum).

2.1.1 Find New Friends

Find Friends allows 1-1 real time communication between 2 students through chat. A student can find a new friend with matching interests and chat in real-time with their newly found friend. To match with a friend, the student can first specify his interests and the matching system will match him with another student of similar interests. Once matched, they will both be invited into a private chat room where they can find out more about each other.

2.1.2 Forum

Forum allows students to interact in a group based setting with other registered users. A student can participate in group interactions through forum discussions. He/she is able to start a new discussion topic by creating a new forum post. He is also able to add on to existing discussion topics by adding comments. In addition, the student can edit or delete posts/comments created by them. Other actions include filtering discussion posts by the list of topics available (shown below) and upvoting/downing a post.

Forum Topics: Academic, Admin, Accommodations, CCA, Tips, Misc

2.2 Purpose

Covid-19 has become a prevalent issue over the past 2 years and has impacted the society negatively. One of the more pressing issues would be the lack of social interaction between people, especially between students who are about to enter university. Due to the lack of physical orientation camps, it has made it even more challenging for them to make new friends.

To solve this problem, we have decided to create a social media platform for NUS students to meet new friends and communicate with one another, even before school starts.

3. Functional Requirements (FRs)

Functional Requirements			
FR	**Description**	**Use Cases**	**Priority**
User Authentication (F1)			
F1.1	The app should allow users to sign up for an account using their NUS emails with the domain 'u.nus.edu'.	UC3	High
F1.2	The app should allow registered users to login.	UC1	High
F1.3	The app should allow users who are logged in to logout.	UC5	High
F1.4	The app should allow users to verify their email address upon signing up.	UC3, UC4	Medium
F1.5	The app should allow users to resend activation email should the email verification link expires.	UC4	Medium
F1.6	The app should allow users to reset their password should they forget it.	UC2	Medium
Navigation (F2)			
F2.1	The app should allow users to decide which functionality of the app they would like to explore.	UC6	High
F2.2	The app should allow users to return to the home page (Find friend page) easily.	UC6	High
Find Friends (F3)			
F3.1	The app should allow users to indicate their interests (Music, Sports, Faculty, Gender, Art, Anyone).	UC7	High
F3.2	The app should allow users to match with one other user on the app according to their interests.	UC7	High
F3.3	The app should allow users to match randomly with one other user if the users do not have any matching preference.	UC7	High
F3.4	The app should allow users to chat in real-time with another user.	UC7, UC8	High
F3.5	The app should allow the matching service to reset, should there be a time out in matching (30sec).	UC7	High
F3.6	The app should allow the user to cancel the matching service while the app is still searching for a match.	UC7	High
F3.7	The app should allow users to end the chat when they are in a chatroom with another user.	UC7	High

Forum (F4) - Post			
F4.1	The app should allow users to create a new post in a chosen topic.	UC9	High
F4.2	The app should allow users to read all the posts from a chosen topic.	UC8	High
F4.3	The app should allow users to view all the posts posted by them for a topic.	UC10	High
F4.4	The app should allow users to edit a post posted by them.	UC11	High
F4.5	The app should allow users to delete a post posted by them.	UC12	High
Forum (F5) - Comment			
F5.1	The app should allow users to create a new comment within a post.	UC16	High
F5.2	The app should allow users to read all the comments within a post.	UC15	High
F5.3	The app should allow users to view all comments made by them for a topic.	UC17	High
F5.4	The app should allow users to edit a comment made by them.	UC18	Medium
F5.5	The app should allow users to delete a comment made by them.	UC19	Medium
Forum (F6) - Functionality			
F6.1	The app should allow users to sort posts by upvote popularity.	UC14	Medium
F6.2	The app should allow users to sort posts by date of post.	UC14	Medium
F6.3	The app should allow users to downvote/upvote a certain post.	UC13	Medium
F6.4	The app should allow users to sort comments by date of comment.	UC21	Low
F6.5	The app should allow users to sort comments by the number of votes	UC21	Low
F6.6	The app should allow users to downvote/upvote a certain comment.	UC20	Low
Profile (F7)			
F7.1	The app should allow users to view their profile.	UC22	High
F7.2	The app should allow users to edit their profile particulars (name, password, profile picture).	UC23	High
F7.3	The app should allow users to upload a profile picture.	UC24	High
F7.4	The app should allow users to view their email address under their profile but remains uneditable.	UC22, UC23	High

F7.5	The app should allow users to delete their account.	UC26	High
F7.6	The app should allow users to change their password in the profile page.	UC25	Medium

4. Non-Functional Requirements (NFRs)

Non-Functional Requirements		
NFR	**Description**	**Priority**
Performance (N1)		
N1.1	The app should not appear sluggish, experience noticeable lag during runtime.	High
N1.2	The app should be able to load within 5 seconds.	Medium
Usability (N2)		
N2.1	The app should be able to be viewed properly in different screen sizes.	High
N2.2	The app should be easy to navigate between pages.	High
N2.3	The app should be self explanatory (i.e. new users should be able to use the app without instructions).	Medium
Scalability (N3)		
N3.1	The app should be able to handle 30K students.	Medium
N3.2	The app should be able to withstand a few hundreds text messages within a chat.	Medium
N3.3	An auto scaler shall generate new instances when the workload of microservice instances exceeds 50% CPU Utilization.	High
Security (N4)		
N4.1	The app should protect users' data (e.g. storing hashed password in database).	High
N4.2	The app should verify the authenticity of the users.	High
N4.3	The app should enable strong password checks (8 minimum characters, 1 uppercase, 1 lowercase, 1 digit, 1 special character) when the user signs up or changes their password.	High
Portability (N5)		
N5.1	The app should be able to run in Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.	High
N5.2	The app should be compatible on Mac OS, Windows OS.	Low
Availability (N6)		

N6.1	Users should be able to use the app again within 30 minutes after the app's downtime to prevent any inconvenience.	High
N6.2	The app should have an uptime of 99% at all times.	High
Integrity (N7)		
N7.1	The app should be protected against unauthorized insertion, deletion and modification of data.	High
Constraints (N8)		
N8.1	The app requires internet connectivity in order to function.	High
N8.2	Profile image uploaded should not be more than 10MB.	High
N8.3	Profile image uploaded should be of image type (.jpg, .jpeg or .png format)	High
N8.4	A user can only upvote or downvote a specific post once	High
N8.5	A user can only upvote or downvote a specific comment once	High

4.1 Reasoning for Non-Functional Requirements (NFRs)

Performance (N1)

N1.1	This enables the users to have a smooth and pleasant experience while instant messaging a matched party.
N1.2	This will allow the user to have minimal waiting time to start using the app, improving our app satisfaction levels.

Usability (N2)

N2.1	This is to ensure that users can use the web app via different devices of different screen sizes.
N2.2	This is to ensure that the users would not need to spend time to figure out where to click to visit a feature they would like to use.
N2.3	The app should be straightforward and intuitive for the users to use.

Scalability (N3)

N3.1	From the statistics by NUS (https://www.nus.edu.sg/registrar/docs/info/student-statistics/enrolment-statistics/undergraduate-studies/ug-enrol-20202021.pdf), it has been reported that there are ~30k students enrolling in NUS. Therefore, the app must be able to handle the traffic of around 30k students.
N3.2	Users might get along very well with their newly matched friend. They may hold longer conversations, there is a need to at least handle a few hundreds of messages.
N3.3	This will enable the app to handle a sudden surge in the number of users using the application at the same time.

Security (N4)

N4.1	This will prevent leakages of user's data, which may contain sensitive information such as password.
N4.2	This will prevent unauthorised users from accessing the app, protecting our users' privacy and data.
N4.3	This allows for higher security to prevent users' accounts from being hacked, protecting their data.

Portability (N5)

N5.1	We assume that different users will use different browsers of their choice to enter NUSociaLife. Therefore, we would have to support different environments to ensure that our

	web app is usable for everyone. From our research, Chrome is the most popular browser at 69.28%, and we should be able to minimally support it.
N5.2	Mac OS and Windows OS are the popular operating systems that NUS students are using. We should be able to support the app on these operating systems.

Availability (N6)

N6.1	This will minimise the disruption that the user experienced due to app crash.
N6.2	This will ensure the availability of the app for the majority of time and users can have access to the functionality of the app.

Integrity (N7)

N7.1	This will ensure that the users' data will not be accessed or modified by unauthorised personnel.

Constraints (N8)

N8.1	This is to ensure that users will be able to use the functionalities of the app such as Find New Friends which require internet connection for real-time communication.
N8.2	This is so that there would not be unnecessary usage of database storage for the web app.
N8.3	This is to ensure that the S3 storage is consistent in the type of files being uploaded. Only image file types are to be uploaded into the S3 storage.
N8.4	To prevent users from abusing the voting button and spamming.
N8.5	To prevent users from abusing the voting button and spamming.

5. Quality Attributes Prioritization Matrix

Attribute	Score	Performance	Usability	Scalability	Security	Portability	Availability	Integrity
Performance	2		^	^	^	<	^	<
Usability	5			<	<	<	^	<
Scalability	4				^	<	<	<
Security	3					^	^	<
Portability	2						^	<
Availability	4							^
Integrity	1							

According to the quality attributes prioritization matrix, our top 4 prioritized attributes are Usability, Availability, Scalability and Security.

Usability

Our top prioritized attribute is usability. Our app should be self explanatory, easy to use and users should be able to easily navigate around the app. To ensure the ease of usage, we designed our UI in a manner where each page consists of related executable actions. Users do not need to navigate to different pages to obtain information, before returning to the previous page. Also, we have 'BACK' buttons on nested pages so that users can easily return to the previous page instead of starting from the first page again. Greater detail on this ease of usage can be found in Section 7.8.2 and Section 7.8.3 of our report.

We also focused on ensuring that checks are done on user inputs so that users will be notified of their invalid inputs. We used React-Toastify to add notifications (with appropriate messages) to our app so that users will be able to correct their inputs.

Apart from easy usage, we focused on ensuring that our UI is responsive so that our app adjusts smoothly to various screen sizes. We chose to use Material-UI (MUI), together with CSS, to build our responsive UI. We used various MUI components such as Grid and Container to ensure consistency across layouts.

Availability

Our second prioritized attribute is availability. For profile image upload, our group has decided to use AWS S3 as a storage service. According to Amazon S3 official documentation, S3 storage buckets guarantee 99.99% (four 9s) availability by redundantly storing objects on multiple devices across a minimum of three Availability Zones in an Amazon S3 Region. This enables the application to access the images quickly in the AWS S3 buckets.

For containerisation, we have decided to use AWS EC2 (virtual servers to host container images) because Amazon has several built-in capabilities such as availability zones where we can put instances in different availability zones, elastic load balancing where we can launch several EC2 instances and distribute traffic between them. This will ensure that in the event that one of the EC2 instances becomes unhealthy, the other EC2 instances can take over the job and ensure the app still runs smoothly for the users.

This also aligns to what is considered a good microservices architecture where if one of the microservices goes down, the rest of the microservices is still available.

Scalability

Our third prioritized attribute is scalability. When features are broken out into microservices, then the amount of infrastructure and number of instances used by each microservice class can be scaled up and down independently. This makes it easier to measure the cost of a particular feature, identify features that may need to be optimized first, as well as keep performance reliable for other features if one particular feature is going out of control on its resource needs.

For our application, we have used AWS AutoScaling step tracking where it allows our application to scale up or scale down based on a particular metric for each microservice so this ensures that only microservice that is under heavy load will have to scale up by adding new copy of the same microservice to cope with the load.

Security

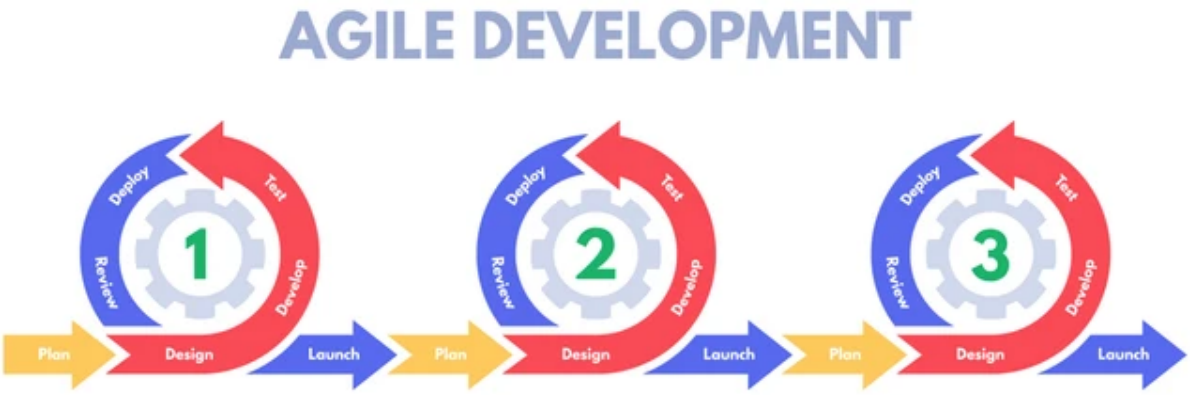
Our fourth prioritized attribute is security. This meant that our app should protect users' data. Users' data such as passwords should not be compromised. We hashed the user's password in a one-way hashing function with an appropriate number of salt rounds using bcrypt, before storing the hashed-password into the database. This ensures the passwords are stored securely since it is a one-way hash function, it is computationally infeasible/impossible to retrieve the password given only the hashes.

Under security, we also focused on verifying users' authenticity. We ensured authenticity right from the user's signup process by implementing an email verification feature.

In addition, we ensured that unauthorized API calls do not pass through. More details on this can be found in Section 7.4.1.

6. Software Development Process

Our team worked on the project using the agile development process.



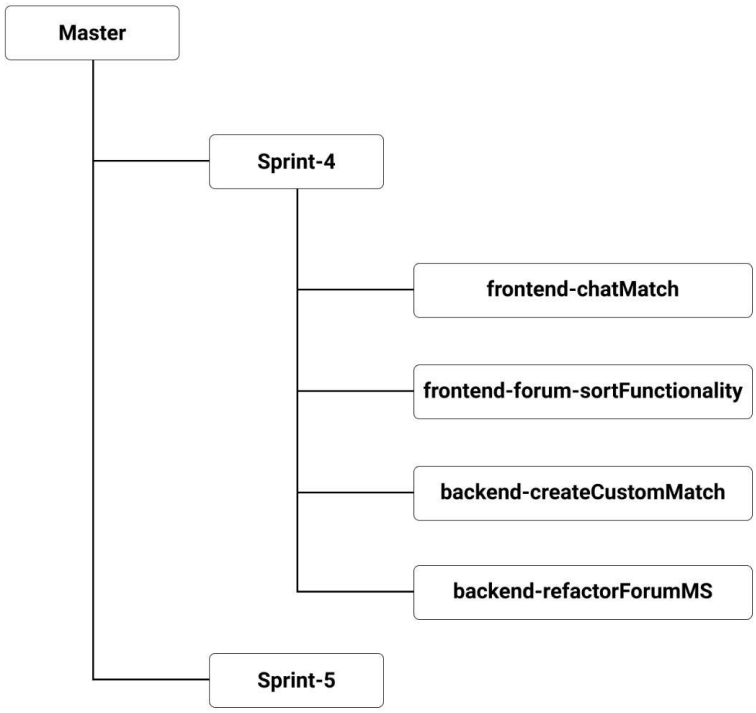
Agile Development Process

Each sprint goes through a whole cycle of planning, designing, developing, testing, deployment and review. This allows us to build up the application incrementally, and allows for frequent iterations and changes whenever we feel that a feature should be refined or extended.

We set up weekly sprints and performed scrums every Monday in order to keep track of tasks, ensuring accountability and for everyone to sync up and update on the project progress. For this, we used Github's project board to keep track of tasks. We allocated 2 people (Marcus & Erynne) to the Frontend, and 2 people (Rachel & Kai Xiang) to the Backend.

The following are links to our project trackers:
Frontend Project Tracker
Backend Project Tracker

In Github, we will create a Sprint branch for each week, and branch off the created Sprint branch to work on. Throughout the week we will merge these branches into the Sprint branch so that everyone gets the latest updates and changes in the codebase. When the sprint ends on Mondays, we would discuss and demonstrate our progress to our team members and merge the sprint branch to the master branch. Then, we would create a new sprint branch for the new week to work on. Here is a diagram demonstrating our team's branching process.

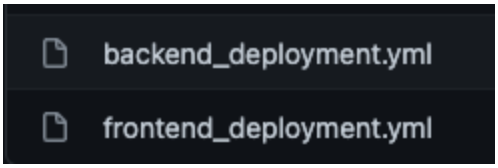


Branching Process Diagram

In between sprints, we also coordinated closely with our team members when working on the application. The frontend team worked closely with the backend team in order to sync up and ensure that both sides are able to get the features up as intended. There is a lot of planning and discussion involved in deciding what the format of the API requests and response will be like so that the frontend will be able to make these calls correctly and integrate the results to update the UI.

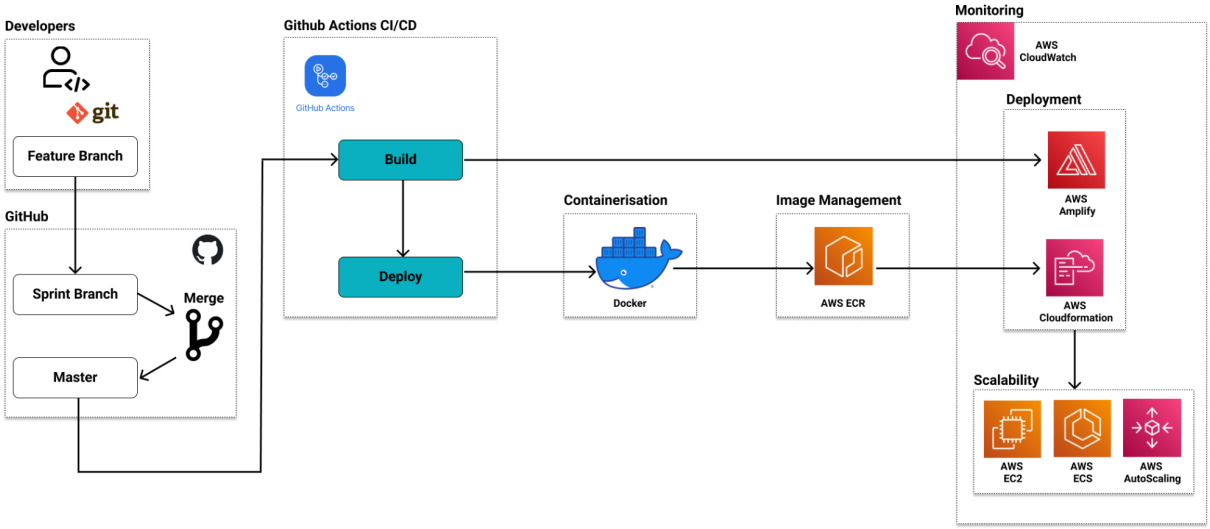
We allocated tasks at the start of each sprint depending on the priority level of our features according to the FR and NFRs. We prioritised the high level of required functionalities, and only worked on the less important features when we had completed the high priority tasks for the week.

To speed up our development process, we used continuous integration and continuous deployment on Github Actions that triggers whenever we merge to the master branch at the end of the sprint. This saves us the time in having to manually deploy the Frontend and each of the Backend microservices to AWS.



Github Workflow files

In our github workflows folder, we have 2 deployment files.



Developer Workflow Diagram

The frontend_deployment.yml configures the AWS credentials that are stored as secrets in our GitHub Project, then deploy the Frontend codebase to AWS Amplify.

On the other hand, the backend_deployment.yml file builds each Backend Microservice into Docker images, configures AWS credentials, then runs a bash script: deploy.sh to deploy all the built Docker images into AWS CloudFormation. We then use AWS CloudWatch to monitor if the deployed application works as intended.

7. Application Design

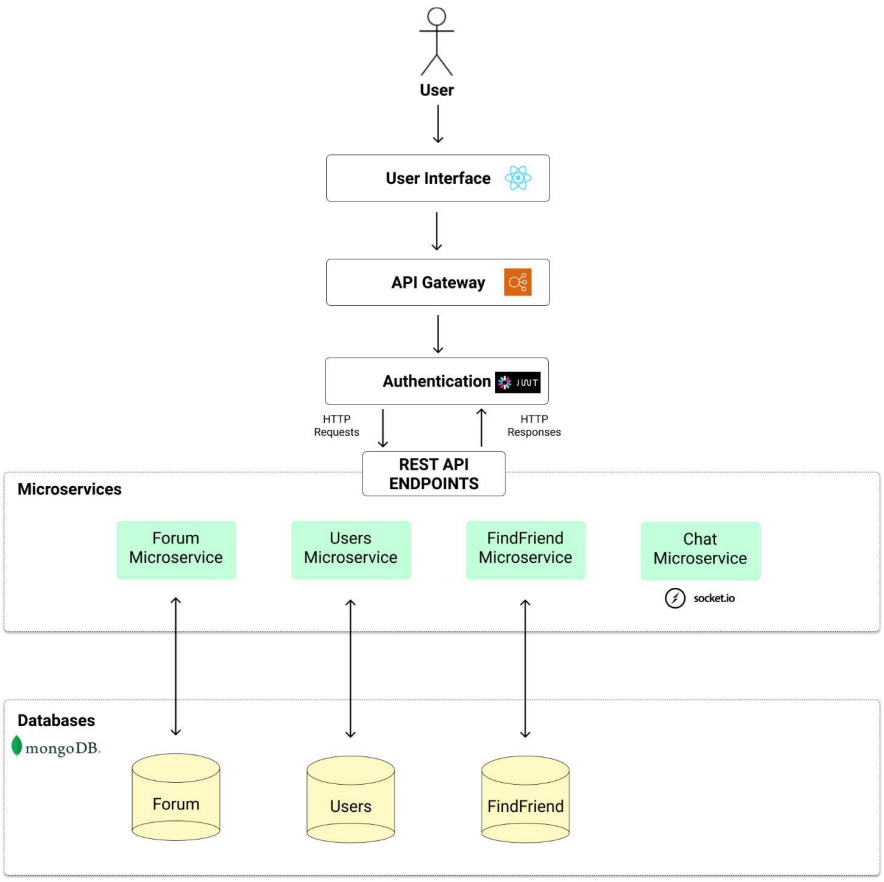
7.1 Tech Stack

Tech Stack			
Type	**Choice**	**Rationale**	**Alternatives**
FrontEnd	React Redux MaterialUI Prettier ESLint	- Virtual DOM feature that allows for quick rendering of UI - Popular developer community, allows us to find solutions to errors quickly - Easy to maintain with modular code structure - Entire team comfortable and had prior experience with React	- AngularJS - VueJS - ASP.NET
BackEnd	Node.js Express.js	- Only need to learn Javascript across Frontend & Backend, allowing more focus on feature development - High support from developer community with multiple NPM packages available - Ability to scale application quickly	- Laravel - Django - Ruby on Rails
Database	MongoDB	- Flexible data models that can be quickly changed according to business logic - Faster development time with fewer bugs given that data structure is in JSON format - Perform fast queries to database and retrieves data quickly - Allows for integrity of data using atomicity, consistency, isolation, durability (ACID) transactions - Monitoring of database performance using tools like MongoDB Atlas	- SQL - DynamoDB - Firebase
Pub-Sub Messaging	Socket.io	- Enables bidirectional and event based communication between browser and server. - Supports reverse proxies and load balancers when deployed to AWS - Rooms feature that complements our FindFriend matching implementation	- Apache Kafka - RabbitMQ

Cloud Providers	AWS	- Multiple options in storage and compute features for more flexibility - Costs are determined based on usage - Security in infrastructure - Easy integration with Docker	- Google Cloud - Microsoft Azure
CI/CD	GitHub Actions	- Our project repository is on GitHub, which makes it easier to have the entire workflow done in the same repository - There is a live log feature, which allow us to track workflow progress CI/CD is triggered - GitHub actions marketplace provides multiple boiler plate code for us to easily modify to fit our uses	- Travis - Jenkins - GitLab
Monitoring	AWS CloudWatch	- Collects monitoring and operational data in the form of logs and metrics and visualises them using automated dashboards - Able to set up automated notifications if an alarm is triggered - Ability to execute autoscaling automatically when alarm is triggered - Ability to track API requests and its information	- PM2 - Grafana - Prometheus
Orchestration Service	AWS ECS (Clustering) AWS EC2 (Instance)	- Automates the scheduling, deployment, networking, scaling, health monitoring, and management of containers - Horizontal scaling for web traffic - Load balancing of containers evenly among hosts - Optimal resource allocation	- Kubernetes - Docker Swarm
Service Discover	AWS Application Load Balancer	- Takes requests from client and distributes across EC2 instances - Server-side discovery router with load balancing capabilities combined in one solution	- Kubernetes - Marathon

Deployment	AWS Amplify AWS Cloudformation	- Quick and easy deployment of web application within AWS cloud - Manage all AWS resources in a single platform - Easy monitoring for CI/CD automations	- Github pages - Vercel - Google Cloud deployment manager
Project Management Tools	GitHub Issues GitHub Project Board	- Entire project resides in a single repository, therefore it is a convenient way to perform all tasks related to the project in a central place - Link Github issues to pull requests	- Trello - JIRA - Asana

7.2 Overall Application Architecture



Overall Architecture Diagram

We decided to implement a Monolith Frontend, with Backend Microservices for our application. This decision was made after weighing the pros and cons of each architectural design.

Micro-Frontends are great for large projects but can be slower and complicated for small and medium sized projects, working off the same standards is challenging for developers. For example, the design of the Frontend should stay consistent across the Micro-Frontends. A very high level of coordination is required to manage the consistency of the different micro frontends.

Since our application is considered a small/medium sized project, we felt that a single Frontend codebase is easier to set up and to work on. This allows us to put more focus into developing the features, ensuring code quality and keeping our designs consistent within the codebase.

We chose a Microservice Backend Architecture as this allows us to separate the various modules into independent services. The loosely coupled design provides more flexibility and scalability, in terms of adding new features, as compared to a Monolithic Architecture.

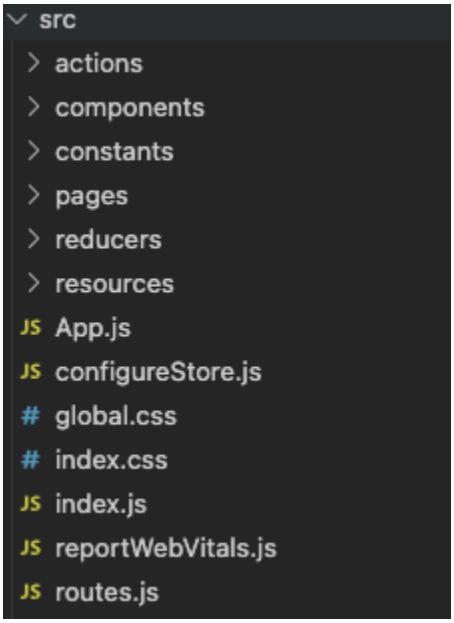
Also, this reduces downtime through fault isolation. Should a specific microservice fail, the failure can be isolated from other microservices, allowing the other services in the app to continue functioning. For instance, should there be a bug in the Forum service, users will still be able to use the FindFriend service.

Furthermore, Microservice Architecture allows our team to develop, maintain and deploy each microservice autonomously and quickly.

With a Microservice Architecture for the Backend, it reduces tight coupling between the backend services/components, prevents unnecessary complexity in testing and developing as the code base is relatively small as compared to a Monolith codebase, provides for more opportunities in scalability and reusability, and a faster deployment rate as compared to a Monolithic application.

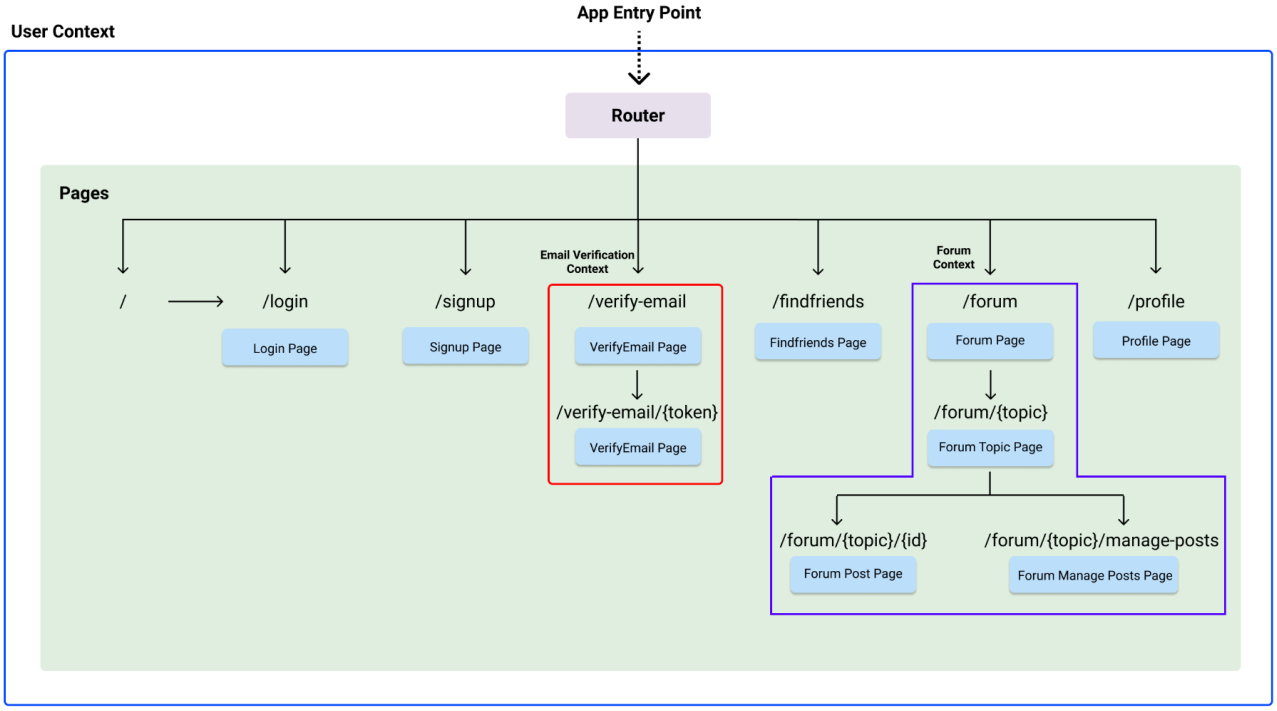
7.3 Frontend Architecture

As mentioned earlier, we decided to implement a Monolith Frontend for our application due to the scale of our project. A Monolith Frontend gives us more allowance in developing our features. Furthermore, as all the codes are located in a single codebase, our Frontend developers are able to keep the code and design style consistent within the project easily with much less coordination required as compared to a Micro-Frontend.



Src folder structure of Frontend

1) App.js will be the entry point of the Frontend application
2) The file: routes.js contains the react-router routes of our application
3) Actions folder contains all Redux actions
4) Reducers contains all the reducer files for the different features in our application
5) The file: configureStore.js sets up the Redux store and persisted reducer for the application
6) Constants folder contains constant values used by Redux, FindFriend components and Forum components
7) Components folder contains the components used to build a page within our application
8) Pages folder consists of the relevant pages to be rendered by our application
9) Resources folder contains all images used by our application



Overall Frontend Architecture Diagram

7.3.1 Router

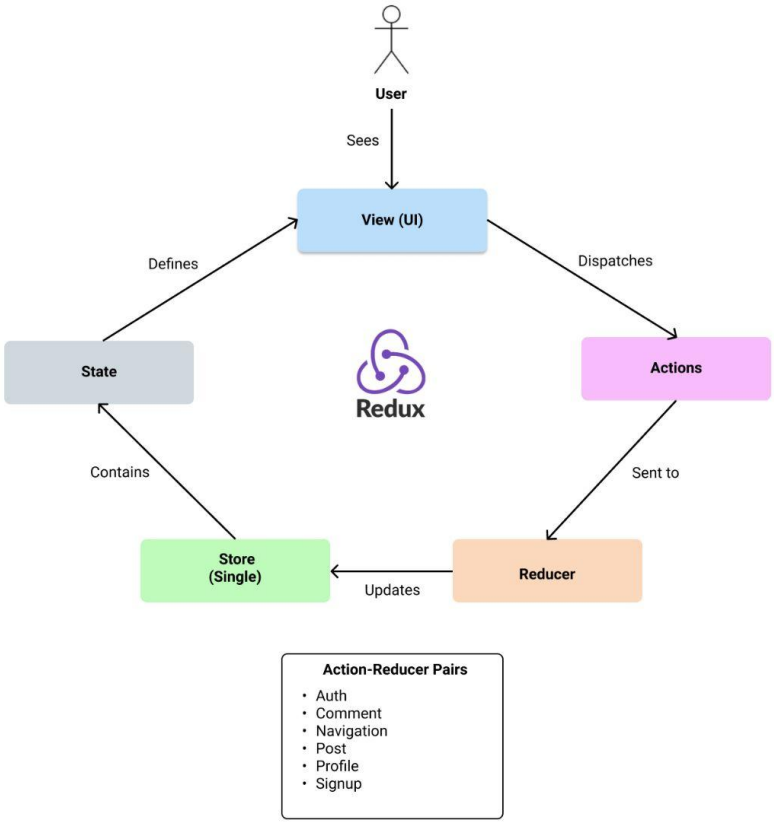
The router component is the initial entry point to our application. Users will be redirected to the login page (/login) upon submitting the app's url. Depending on the user's exact path and state, different pages will be rendered accordingly

7.3.2 User Context

The user context is responsible for holding and maintaining the entire state of the application (and user's information) for the current session. For our architecture, the user's context is the single Redux store which includes the various user's information such as:

1. Authentication - Contains the access token (JWT Web Tokens) which is used to authenticate all API requests made by the client. Every access token expires in 24 hours (upon generating it) and users will be logged out automatically when the token expires.
2. Profile - Contains user's details such as name, email and profile image.
3. Match - Contains the state of the match which provides the room id where Socket.io can use to join users into a common chatroom.
4. Navigation - Contains the current navigation link the user is in, which renders the correct page and allows users to keep track of system navigation status.
5. Signup - Tracks if users are verified and conditionally renders the signup page depending on verification status.
6. Post - Contains the current forum topic the user is in and all the posts of that forum topic.
7. Comment - Contains the comments of the current post the user is in.

7.3.3 Redux Architecture



Redux Architecture Diagram

We decided on Redux for a few reasons. Firstly, it reduces the complexity of passing around props into nested components. Passing props into components may not be a problem for a small application, but it is challenging for the Frontend developers, especially when the Frontend will be using a Monolith Architecture. The complexity of using props increases as the number of components increases.

Secondly, Redux closely follows the Model, View, Controller (MVC) design pattern, where store is the Model, Reducers are the Controller, and our UI are the views. This gives us a cleaner codebase, and separates our business logic from our UI codes. It helps developers work quickly as files and logic are easy to locate and understood due to separation of concerns.

We set up the store using the Redux Persist library, which uses the localStorage for web applications. This allows us to store our Authentication token hashed by the Backend, into the Auth redux state. Subsequent API calls from our Frontend will retrieve this Auth Token to make Authorized API calls to the Backend.

Redux will work in the following manner. The user will first communicate with the UI, which will dispatch actions based on the user's interaction (Clicking Login button, Change Photo button etc.). Within the actions, API calls to the Backend will be made and will return a state to the reducers. The reducers

will determine the returned state and update the state and data in the store. The UI will then re-render whenever a state changes, to show an updated view to the user.
Redux allows the logged in user's information to be accessible by all components which are nested within the user context. This is a form of Shared Data Pattern where each component of the app can have direct access to the state of the application (and user's information) stored in a single common data store. As compared to having a single component listening to the store and data being passed between components via props or callback functions (Flux Pattern), this pattern has multiple advantages and disadvantages.

Advantages

1. **Centralized state management system**
   The single store acts as a single source of truth where all data are centralized. This makes it easier for any component to get the information/state it requires as all components nested within the user context can access the data directly. Furthermore, as this is a complex application with many components and contains different user information, there would be a difficulty in tracking data passed between components. Having a centralized state management system also helps to avoid issues regarding data inconsistency bugs and data synchronization issues between components. Whenever an action is dispatched, the data store is updated and all components which access the data will have the updated data. There is no need to pass down the updated data from one component to another or worry about data inconsistency since data is being shared among all subscribed components concurrently.

2. **Performance optimization**
   As we have chosen to use React for the frontend of our application, by default, whenever a component is updated, all nested components are re-rendered. This results in unnecessary re-rendering if the data of a given nested component did not change. By having a single data store, performance is improved as components are re-rendered only when its data changes since it obtains data from the store directly.

3. **Easy debugging**
   We used Redux DevTools to track our state changes when we were developing the application. This allowed us to open the DevTools window in our browser and allow us to trace whenever an action is called and how the state of our store changes each time. This reduces the usage of console.logs as we normally would use to debug the application.

Disadvantages

1. **Code complexity**
   The Shared Data Pattern introduces code complexity as the store, along with the necessary actions and reducers are required to be implemented. This is often not required for applications with simple UI changes where state sharing between different components is preferred as it is easier to maintain. Furthermore, boilerplate code may be introduced through the actions and reducers code.

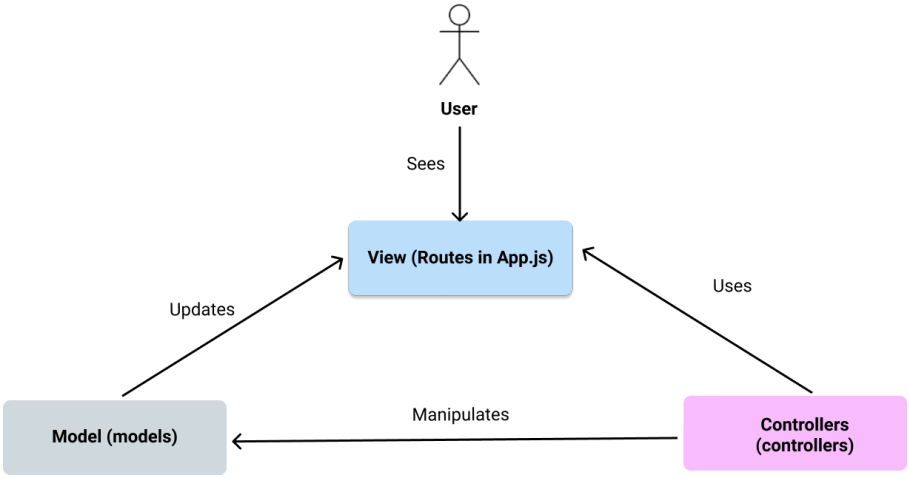
2. **Overhead memory costs**

States in redux are immutable objects. Whenever an action is dispatched, the reducer would update the state by returning a new state each time. This causes extra usage of memory which is not ideal for the application.

The disadvantage of increased code complexity is less applicable to our application as our application has different components that require data from multiple sources.

Considering the multiple benefits as compared to the disadvantages, and that our application is much more complex than an application with simple UI changes, we decided to adopt the Shared Data Pattern with Redux as a state management tool.

7.4 Backend Architecture



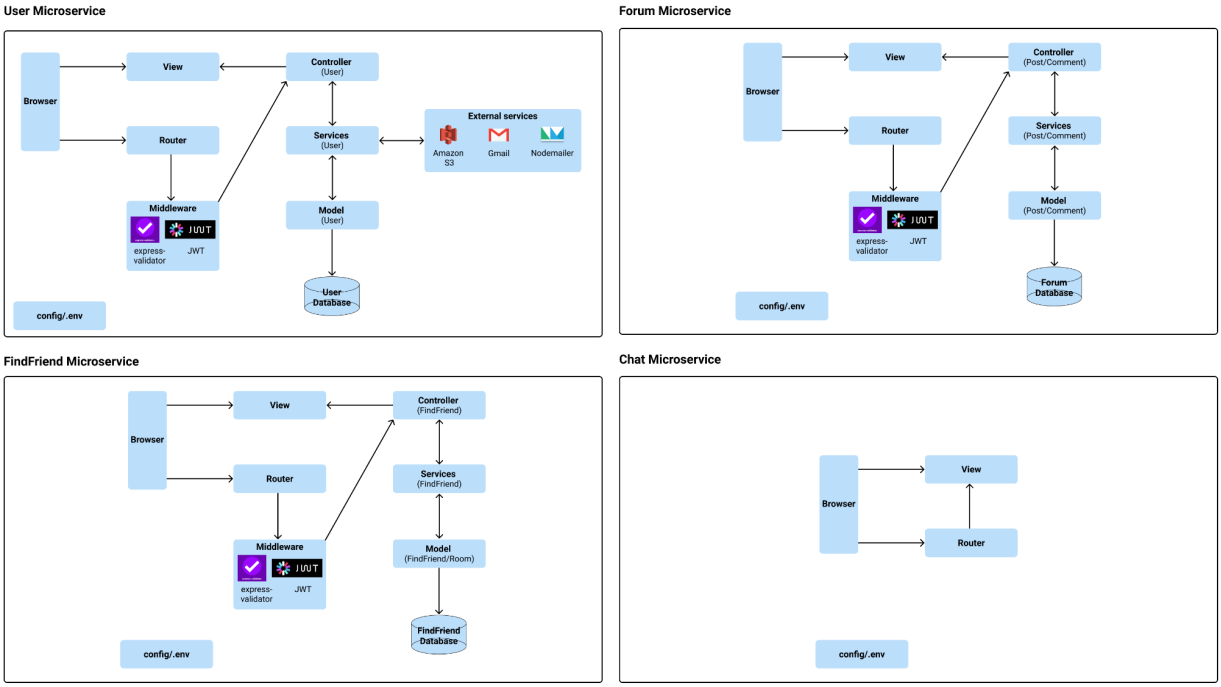
Overview of Backend Architecture Diagram

For the backend architecture, we have decided to adopt microservices architecture where the NUSociaLife application is a collection of microservices (Users, FindFriend, Forum and Chat). This ensures that the microservices are highly maintainable and testable, loosely coupled with high cohesion, independently deployable and developable.

The main motivation for our team to use the microservices architecture is how microservices are decomposed in a way that abides to the **Single Responsibility Principle** (SRP) in the object-oriented design (OOD) where each microservice only has a single responsibility, implementing a small set of related functions (e.g User microservices is only responsible for user management) and has only a single reason to change. The microservice architecture also encourages **Separation of Concerns** where changes in one microservice will only affect that particular microservice (e.g. Change in Forum microservice will not affect User microservice) which can be very critical in agile software development processes where there are new implementations due to change in business requirements throughout the software development cycles.

The backend architecture is also inspired by the Domain-Driven design (DDD) where the structure and language of software code in each microservice matches the business domain closely.

Within each microservice, we have implemented an architecture style that is inspired by the MVC model. The internal structure of each microservice is split into different parts (model, view, controller, middlewares, router, services) which follows closely to the MVC model. The user will first invoke an action which will be directed to the controllers, the controller then updates the model as per user actions and then the model will trigger the view to update.



Backend Architecture Diagram of each Microservice

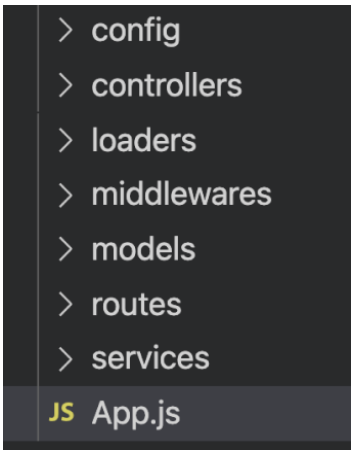
For the Users Microservice, when the user interacts with the Browser, the API routes will be called in the router. The information will then be passed to the Middlewares which handles the Authentication of Users and Validation of Inputs. After authenticating and validating, the controllers now take charge regardless of the success of the authentication or validation. If there is a failure in Authentication and Validation, the Controllers will then return an error response to View where it will be displayed on the Browser. If Authentication and Validation are successful, the functions called by the API routes will then proceed. The Controllers will draw on the logic implemented by the Services where the necessary queries from the database are performed. After all the operations are executed, the controller class will then send a successful or unsuccessful response to the view where the responses will be loaded in the Browser to the User.

In the Users Microservice, there are also other three external services namely -- Amazon S3, Gmail and Nodemailer. For Amazon S3, it is mainly used to upload the Profile Image of the users which will then be stored in S3 Bucket. Gmail and Nodemailer work together to send email to users to verify their email accounts and to reset their password. All the emails will be transported via Gmail's service.

For the Forum and FindFriend Microservice, the flow is similar to the User Microservices except that no external services would be involved.

For the Chat Microservice, it has the simplest architecture among the four microservices. The API routes will be called in the Router when the Chat connection is being utilised by the users. The API route then calls the functions which will all be encapsulated in the App.js file which is situated in the View. All the connections to the Socket IO are done within the entry point of the microservice architecture and will be

returned to the Browser subsequently.



Main folder structure of each Microservice

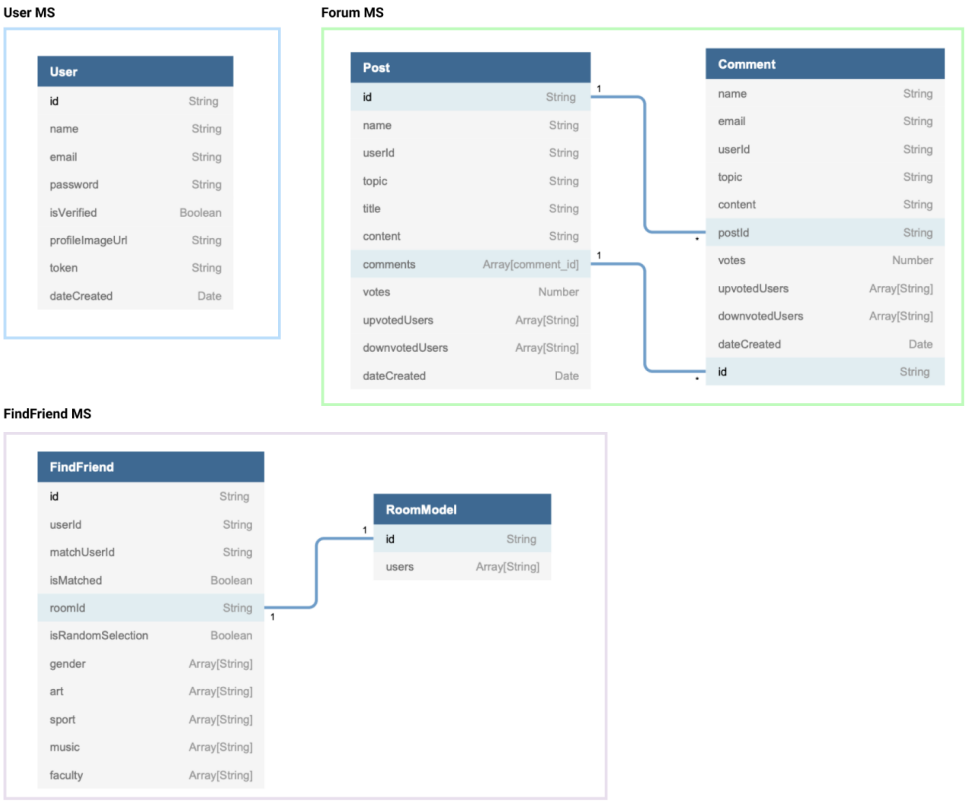
1) App.js will be the entry point of the microservice application
2) Config folder consists of the declaration of application environment variables and secrets
3) Controllers folder consists of express controllers for routes, respond to client requests, call services
4) Loaders folder consists of startup processes like MongoDB database startup
5) Middlewares consists of operations that check or manipulates API request before the controller utilise the API request
6) Routes folder consists of express routes that defines the API structure
7) Services folder encapsulates all the business logic

7.4.1 Security

For our application, we have decided to use the access token pattern where for each new user registration, a temporary JWT token signed using the user email address with a lifetime of 15 minutes will be issued for the user to activate the account. Upon successful user account registration through account activation link being sent to the email, the user will then be able to proceed to login where a JWT token signed using the user id with a lifetime of 24 hours will be generated upon successful login and returned to the client (user) in json data format.

We have decided not to use the API routes with /userId or /token being append to the back of the route (e.g. http://localhost:5000/api/users/userId or http://localhost:5000/api/users/token) as it is common for hackers using web crawlers to search for userId or token that is attached to the back of the API routes. Instead, in subsequent API routes where the client (user) tries to perform HTTP requests, the token will be placed in the request authorization headers as bearer token. For each API request, there will be a token verification check in the middleware section of the backend code where only valid tokens being passed in are allowed to proceed on. In this way, the identity of the client (user) will be securely passed around the system from frontend to backend.

7.5 Database Design



Database Schema Design

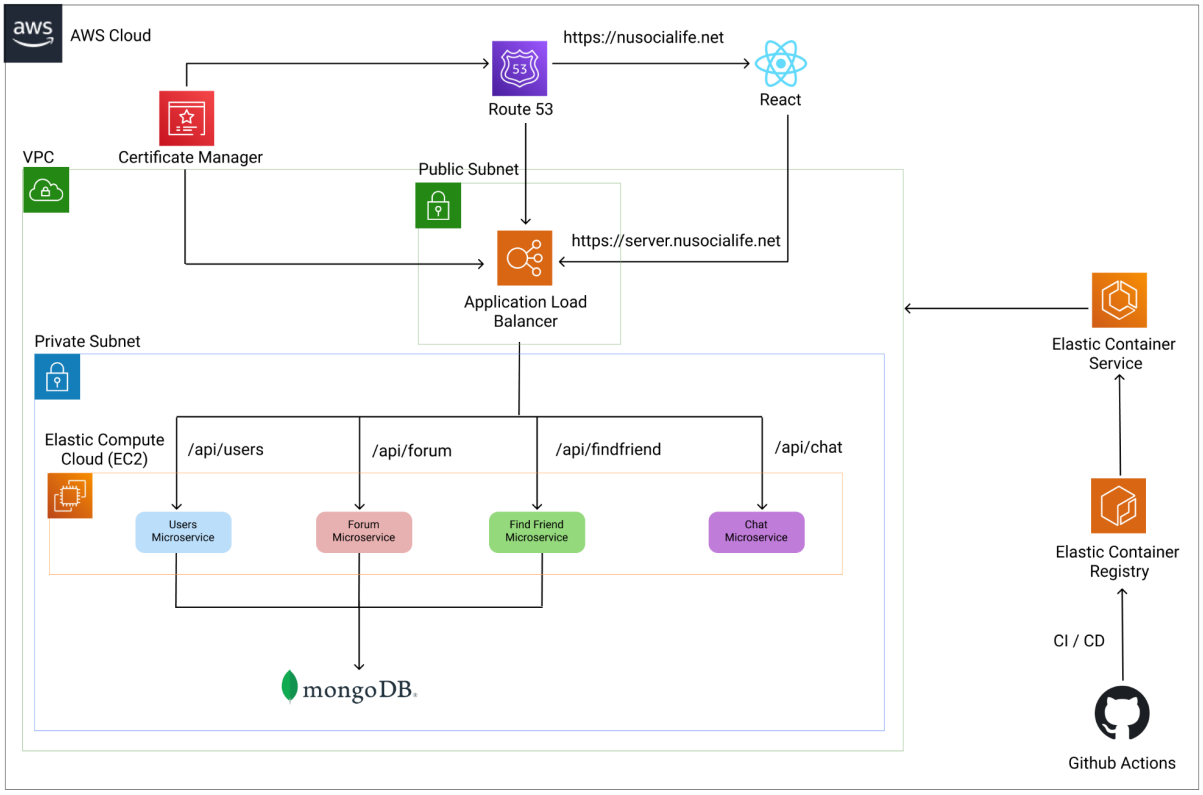
We have chosen a Non-Relational Database so that we are able to retrieve queries quickly and increase its performance as compared to Relational Databases like SQL. Our team has adopted the database per service pattern where each microservice has its own database table(s). This ensures that there is no coupling between the tables of the different microservices as shown in the diagram above. As there are only a few relationships between the Tables in each Microservice, it would be better to use a Non-Relational Database to speed up its performance especially when we need to store a large number of entries in the tables, instead of joining multiple tables together to query which can be quite taxing on the performance of our web application.

Furthermore, each Microservice has its own Database so that we are able to keep the persistent data private to that service and can only be accessible via its API, and that a service's transaction only involves its database. Next, this ensures that each Microservice's database cannot be accessed directly by other services. With all these in place, changes to one Microservice's database will not affect any other microservices. This also ensures greater clarity in ownership in the different microservices.

Coupling between tables is only present in Forum MS and FindFriend MS. For Forum MS, the Post Model has an Array of Comments where it stores all the Id of the comments, where the comments are populated to retrieve all the comments associated with the post. For the FindFriend MS, there exists a One-to-One relationship between the roomId of the FindFriend Model and the id of the RoomModel such that every Matched Pair will be assigned a unique Chat room.

Since there are only a few relationships between the tables, we chose MongoDB as it is faster due to its ability to handle large amounts of unstructured data when it comes to speed.

7.6 Deployment Architecture



Deployment Architecture Diagram

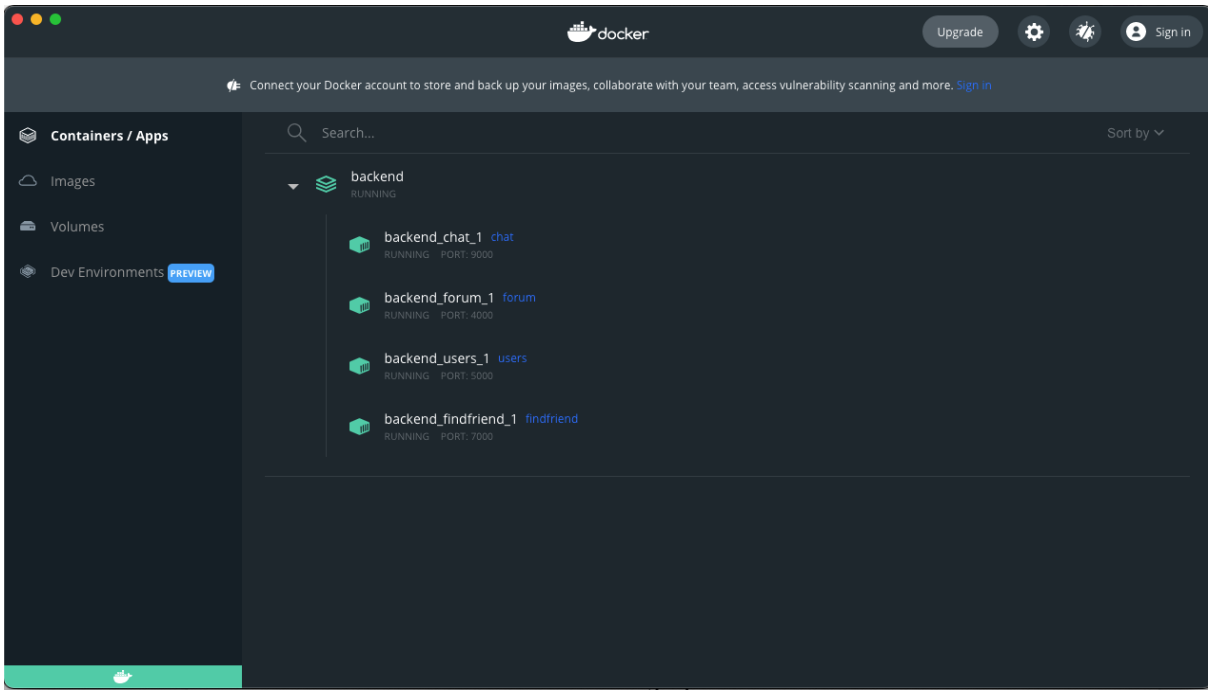
For deployment, service instances per container pattern will be used where each microservice is packaged as a Docker image and each microservice instance is a Docker container. One of the benefits of this pattern is the ease to scale up and down a microservice by changing the number of container instances. The other benefit is the fast build process and start process of containers where it is much faster to package an application as a Docker container compared to Virtual Machine (VM) since only the application process starts and not the entire Operating System (OS).

General Overview of the components in the Deployment Architecture

1) AWS Elastic Container Registry (ECR) will hold the Docker images of the microservice applications. There are a total of 4 repositories being hosted on AWS ECR for users, forum, findfriend and chat microservices.
2) AWS Elastic Container Service (ECS) is a fully managed orchestration service that will manage, deploy and scale the Docker containers which are also the microservices instances in the cloud environment. **Every microservice is also an ECS service where the ECS service can specify the number of tasks (copies of the microservice) that can be run.**
3) AWS Application Load Balancer (ALB) is a router/ingress controller which accepts incoming traffic from clients (users accessing the website from the frontend) and routes requests to its registered targets which are the microservices. It also monitors the health of the targets and redirects traffic only to healthy targets.
4) AWS Elastic Computing Cloud (EC2) is a Linux-based/Windows-based/Mac-based virtual server. This is the place where running containers of ECS services are being placed on the AWS cloud.
5) AWS Route 53 is a highly available and scalable cloud Domain Name System (DNS) web service which enables effective connectivity to the AWS ALB.
6) AWS Certificate Manager issues SSL certificates which allows our website links to be more secure using HTTPS connection.
7) A VPC network to host the AWS ECS cluster and associated security groups.
8) A **public subnet** is used for resources that must be connected to the internet, and a **private subnet** is used for resources that will not be connected to the internet.
9) Github Actions will trigger the whole continuous deployment process when there is a new commit.

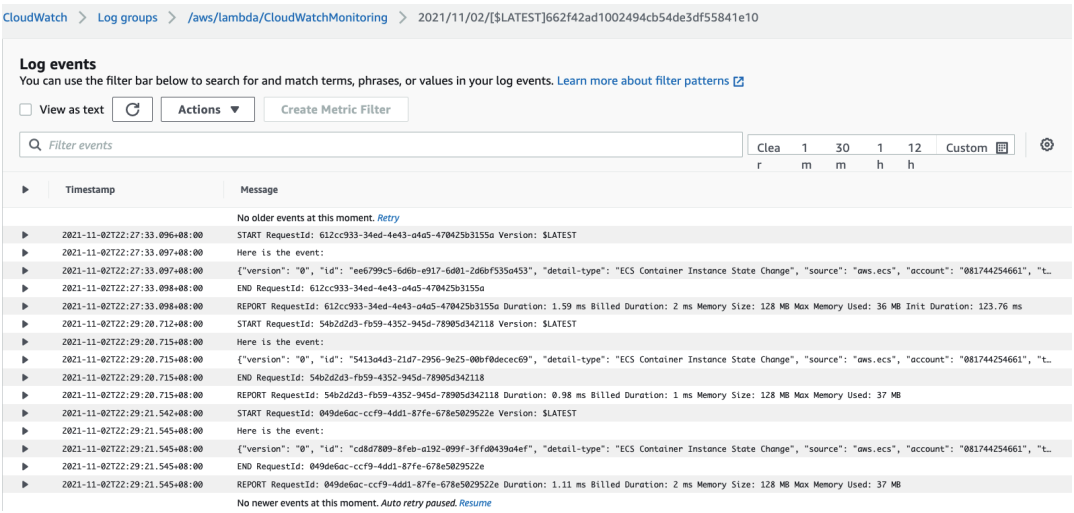
7.6.1 Deployment process

First of all, docker-compose build command will compile the docker images of the respective microservices of the application.



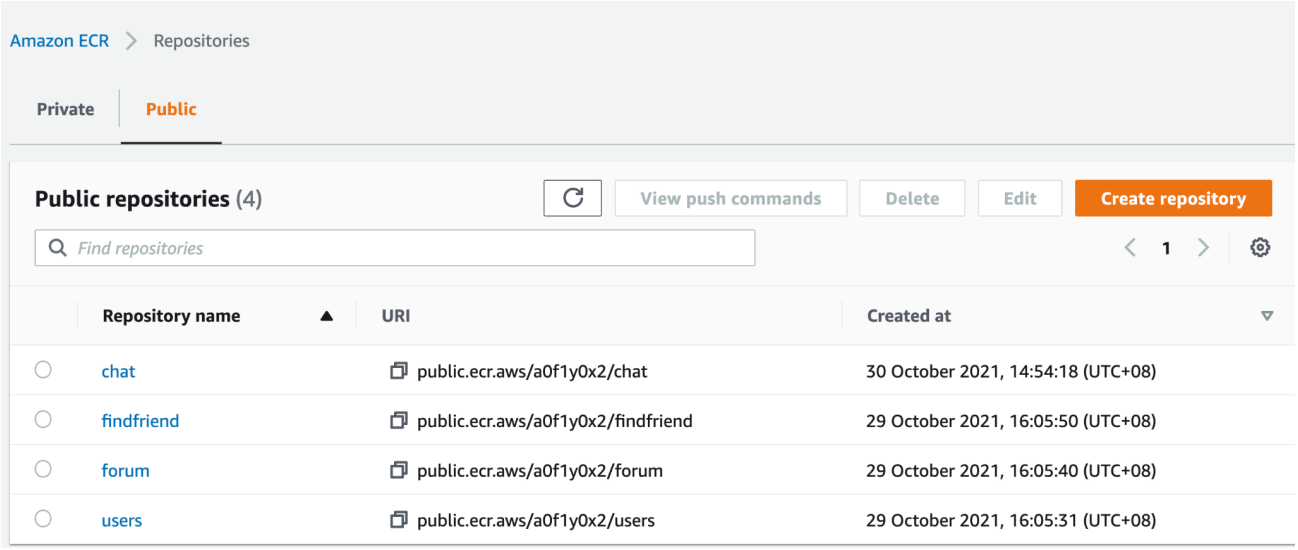
Sample screenshot of Docker images build

Prior to deployment on AWS cloud, AWS Lambda will be set up to send automated logs to AWS CloudWatch where new events of AWS ECS can be monitored from time to time. Here is the tutorial guide https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs_cwet.html to set up the AWS Lambda function to listen for AWS ECS events and write to CloudWatch.



Sample screenshot of AWS CloudWatch monitoring log

Next, the script deploy.sh will be run and the docker images will be pushed on to the Amazon ECS repositories where each repository belongs to each microservice.



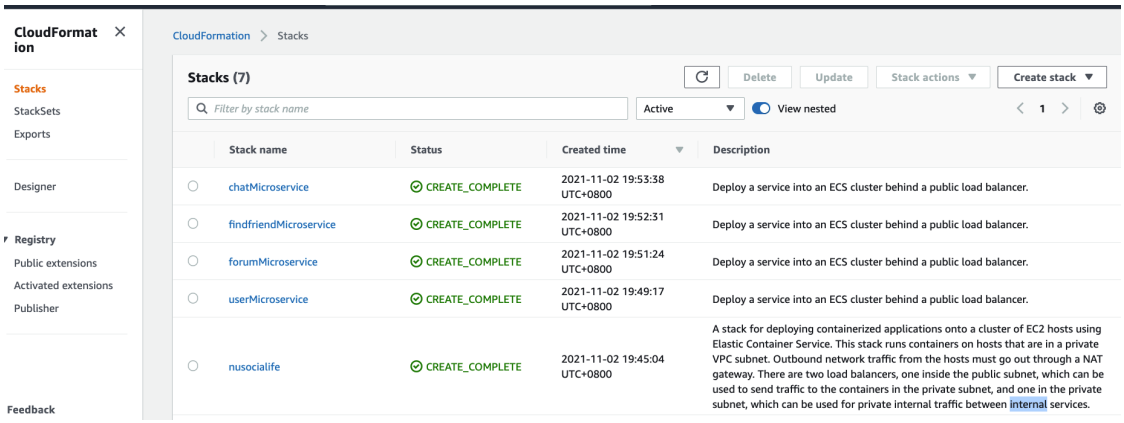
Sample screenshot of repositories in ECR

Lastly, the script will execute CloudFormation templates to automate deployment in AWS. AWS CloudFormation helps to model cloud environment and automate deployment of AWS resources by treating the infrastructure as a code. Upon deletion of the stack, AWS resources will be removed from the cloud too.

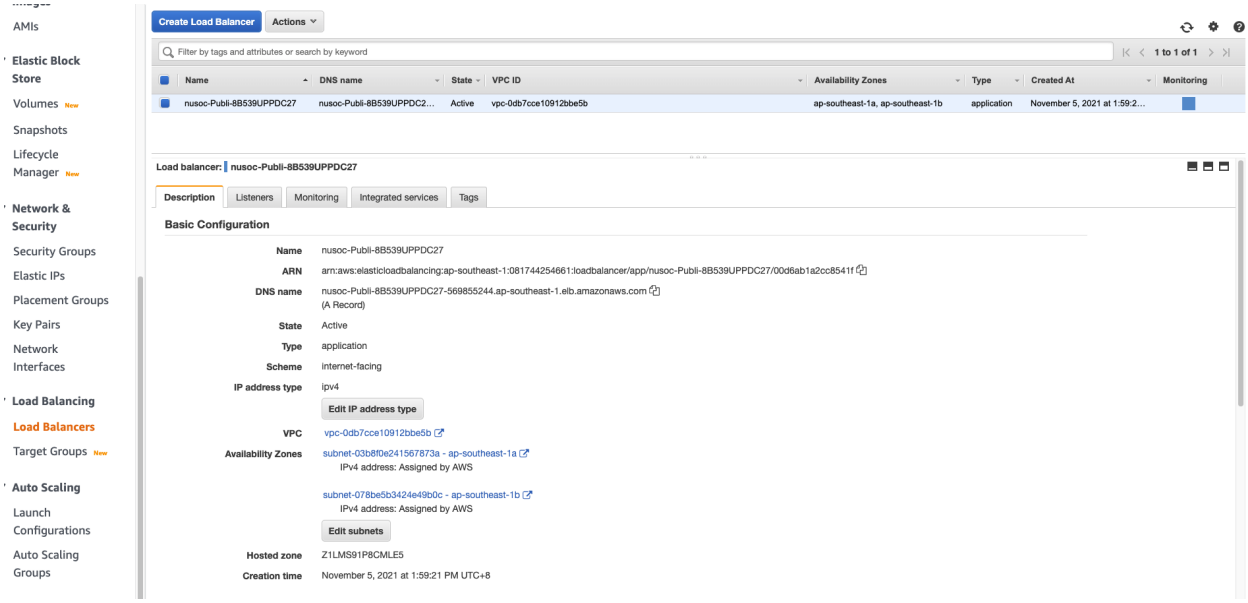
```
# Load balancers for getting traffic to containers.
# This sample template creates two load balancers:
#
# - One public load balancer, hosted in public subnets that is accessible
#   to the public, and is intended to route traffic to one or more public
#   facing services.
# - One private load balancer, hosted in private subnets, that only
#   accepts traffic from other containers in the cluster, and is
#   intended for private services that should not be accessed directly
#   by the public.

# A public facing load balancer, this is used for accepting traffic from the public
# internet and directing it to public facing microservices
PublicLoadBalancerSG:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: Access to the public facing load balancer
    VpcId: !Ref "VPC"
    SecurityGroupIngress:
      # Allow access to ALB from anywhere on the internet
      - CidrIp: 0.0.0.0/0
        IpProtocol: -1
PublicLoadBalancer:
  Type: AWS::ElasticLoadBalancingV2::LoadBalancer
  Properties:
    Scheme: internet-facing
    LoadBalancerAttributes:
      - Key: idle_timeout.timeout_seconds
        Value: "30"
    Subnets:
      # The load balancer is placed into the public subnets, so that traffic
      # from the internet can reach the load balancer directly via the internet gateway
      - !Ref PublicSubnetOne
      - !Ref PublicSubnetTwo
    SecurityGroups: [!Ref "PublicLoadBalancerSG"]
```

Sample screenshot of a AWS resource ALB being defined in AWS CloudFormation YAML file

In this step, AWS resources like AWS ECS, AWS EC2, VPC, subnets, security groups, AWS ALB and AWS Autoscaling will be created too.



Sample screenshot of successful deployment of the AWS resources



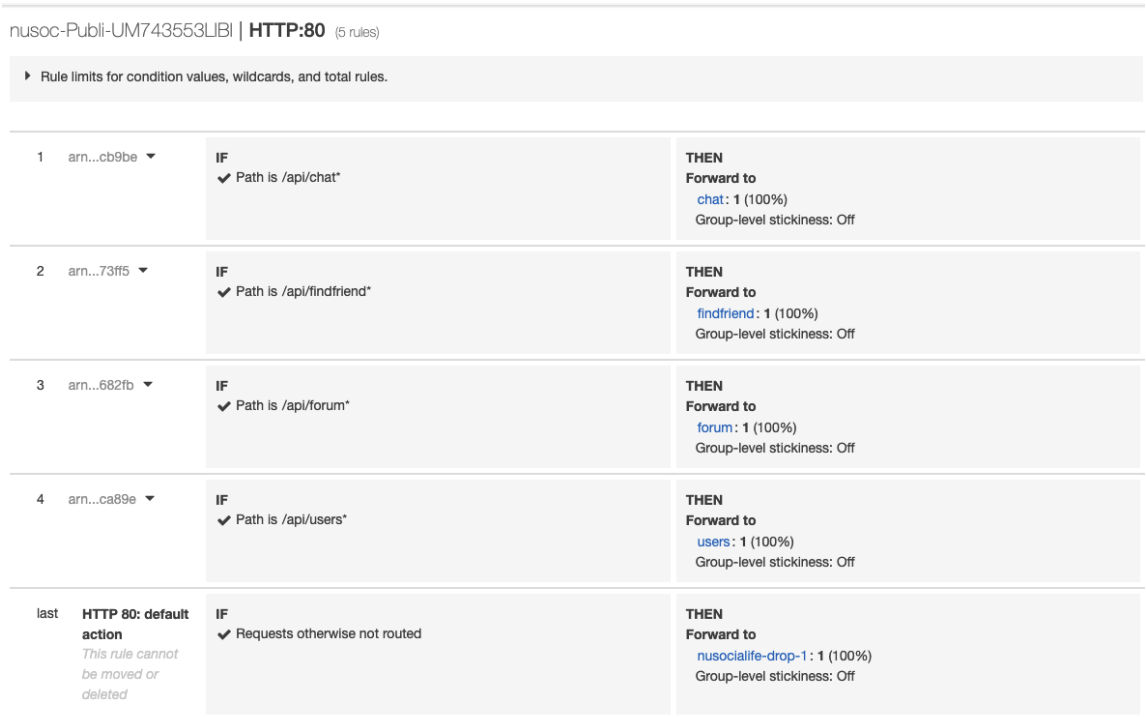
Sample screenshot of successful AWS ALB creation

The load balancer also serves as the single point of contact for clients. The load balancer distributes incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones. This increases the availability of the application.

Server-side service discovery pattern has been adopted where any HTTP(s) requests from the client side will be passed to the AWS ELB which not only balances the traffic among AWS EC2 instances but also functions as a service registry so there is no extra need to query an external service registry.

The architecture also depicts the running containers being deployed into a private subnet. The containers do not have direct internet access, or a public IP address. Any outbound traffic must go out via a NAT gateway, and recipients of requests from the containers will only see the request coming from the IP address of the NAT gateway. However, inbound traffic from the public can still reach the containers because there is a public facing load balancer that can proxy traffic from the public to the containers in the private subnet.

In this stage, one key thing that will be created is a listener for the AWS ALB. A listener is a process that checks for connection requests based on the ports and protocol being defined in the CloudFormation configuration file. The rules being defined for a listener determine how the load balancer routes requests to its registered targets which are the ECS services.



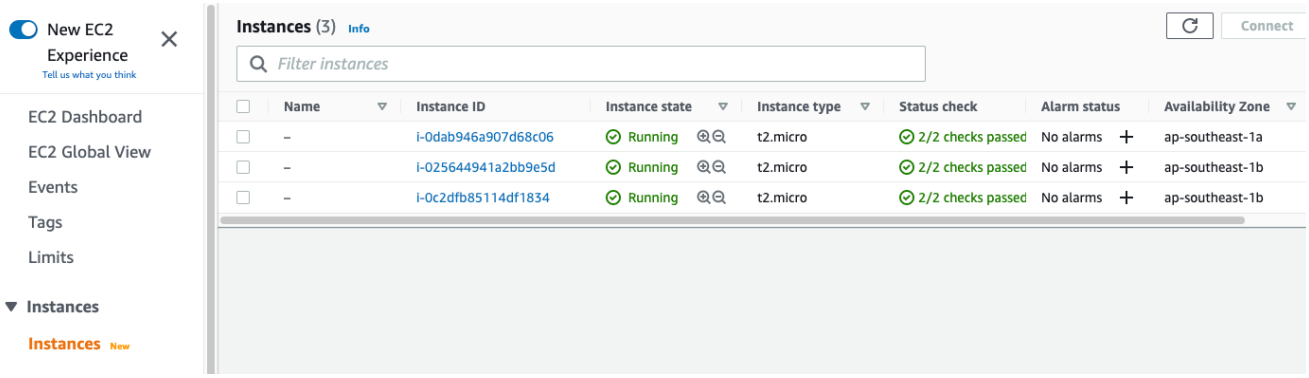
Sample screenshot of listener rules in AWS ALB

When the client request hits backend services, it will go through https://server.nusocialife.net which is directed to the application load balancer in the public subnet in the Virtual Private Cloud (VPC).

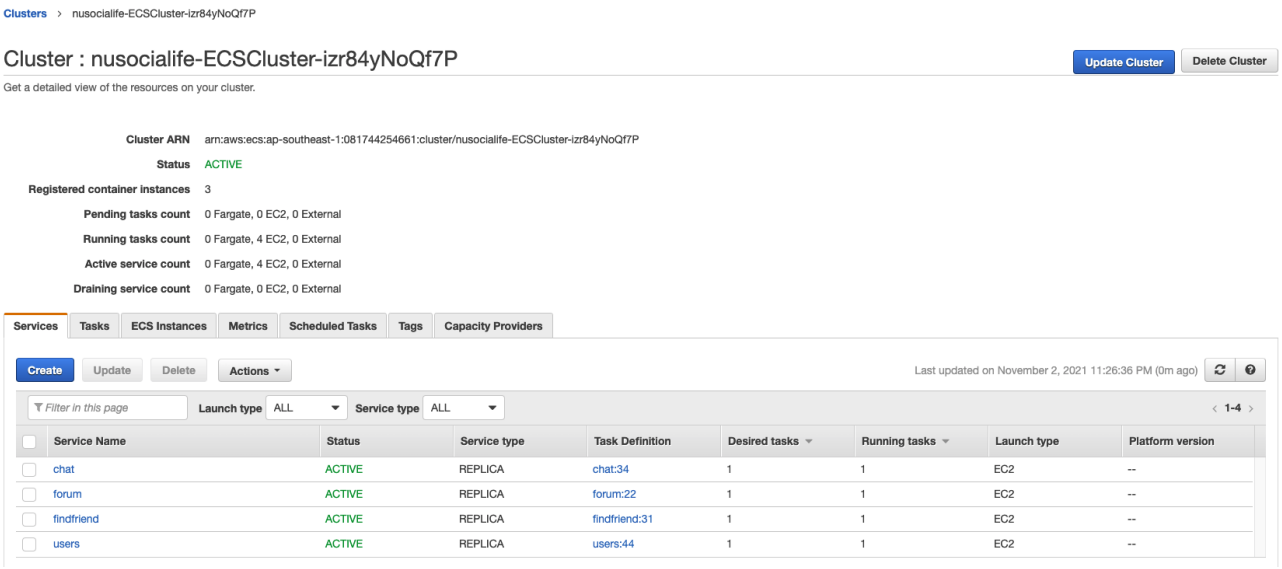
For example, if the client wants to see user microservice, it will redirect user to /api/users and traffic will be pushed to the microservice ports. The full endpoint link will be https://server.nusocialife.net/api/users.

Now, the AWS ECS cluster will also be created with 3 AWS EC2 instances. The AWS EC2 instances will host 4 ECS services which are microservices defined as tasks (running containers in ECS services) where the docker images can be retrieved from the AWS ECR repositories.

It is recommended to have more than one AWS EC2 instance for cloud deployment to ensure **high availability** of the application so in the event that one AWS EC2 instance goes down, there will be a backup instance available. This also enables the load balancer to have more options when it is checking which AWS EC2 healthy instance it can route request traffic to.



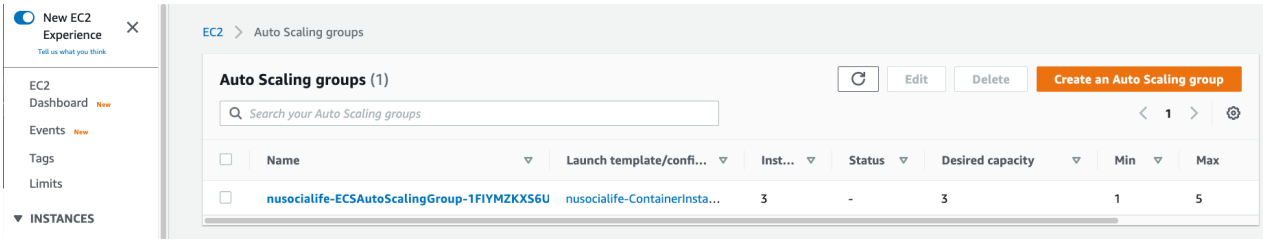
Sample screenshot of successful AWS EC2 instance creation



Sample screenshot of successful AWS ECS cluster creation with 4 ECS services

For our web application, horizontal scaling is being adopted instead of vertical scaling where we feel that it is more practical to add more machines (instances) on top of existing machines to handle surge in workload / client requests instead of upgrading resources for the existing machines running on the AWS cloud.

There will be an AWS AutoScaling group with a policy set to ensure that there is a desired count of 3 AWS EC2 instances being deployed in a AWS ECS cluster with a minimum of 1 AWS EC2 instance and maximum of 5 AWS EC2 instances. This is also known as **cluster scaling** where we scale by the number of servers which are EC2 instances.

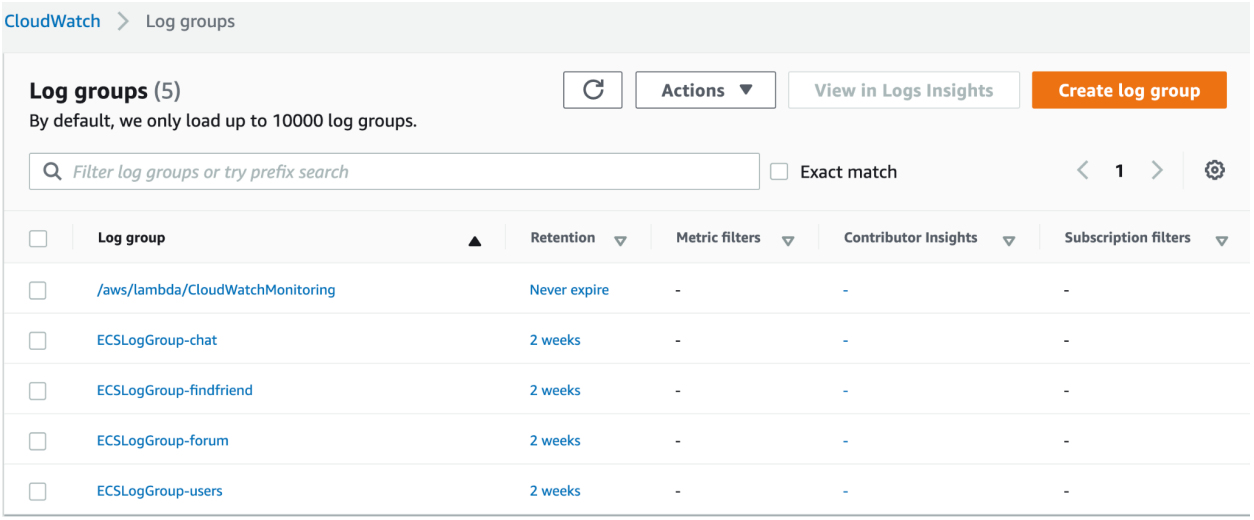


Sample screenshot of AWS Autoscaling group

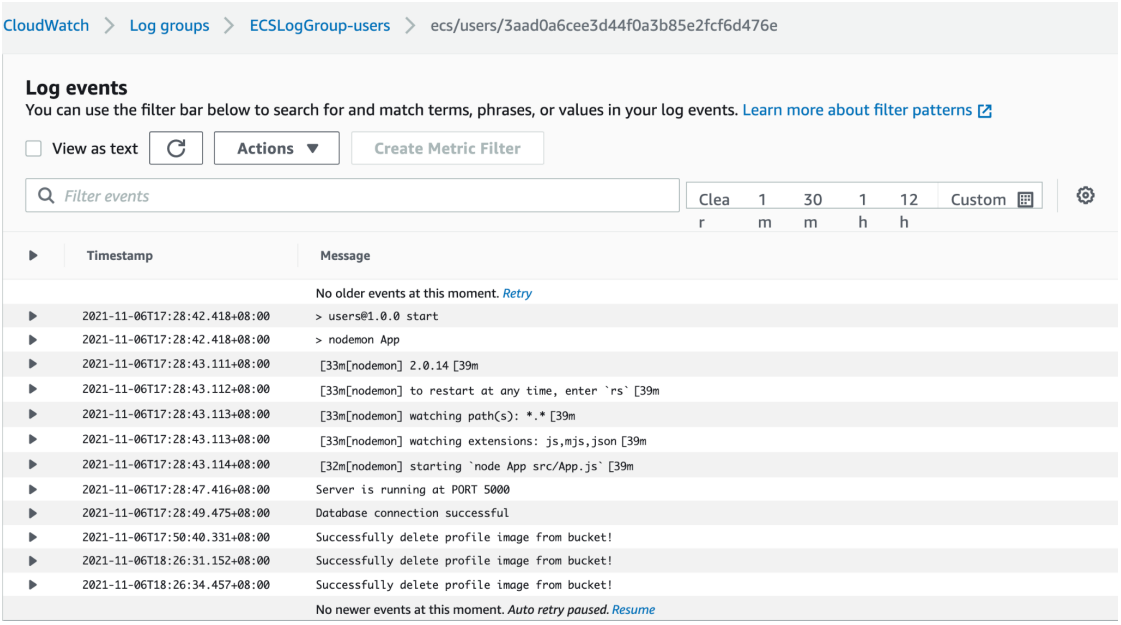
Apart from cluster scaling, there is another type of scaling called ECS service autoscaling where we scale by adding new copies of running tasks (containers) for the ECS service. In ECS service autoscaling, AWS AutoScaling (target tracking type) will be used to perform horizontal scaling for the running tasks inside AWS ECS services. AWS CloudWatch will be used to monitor the state of the ECS services to trigger alarm (workload alarm) if average CPU utilisation of the running containers in the ECS service is above 40%. New running containers (tasks) will be generated to handle the workload.

Note: Each ECS service has its own AWS AutoScaling policy and CloudWatch monitoring alarms

In addition, log groups are created for each API call that is logged to the ECS services and can be monitored from AWS CloudWatch.

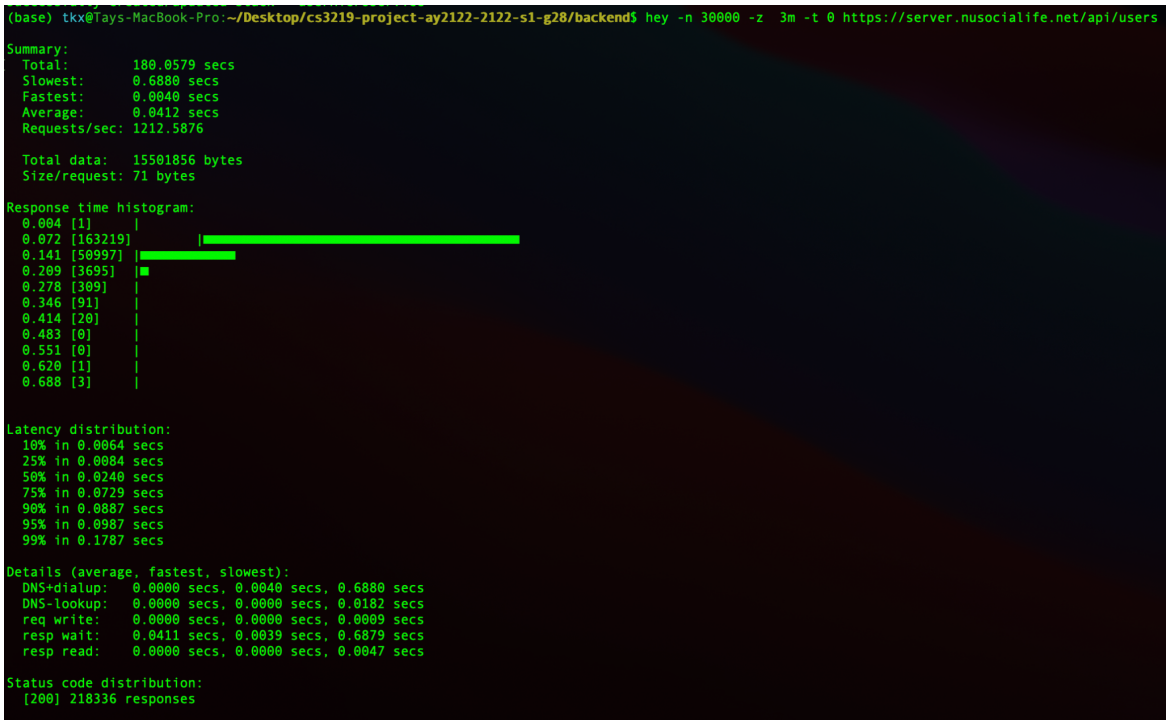


Sample screenshot of AWS Log Groups in AWS CloudWatch



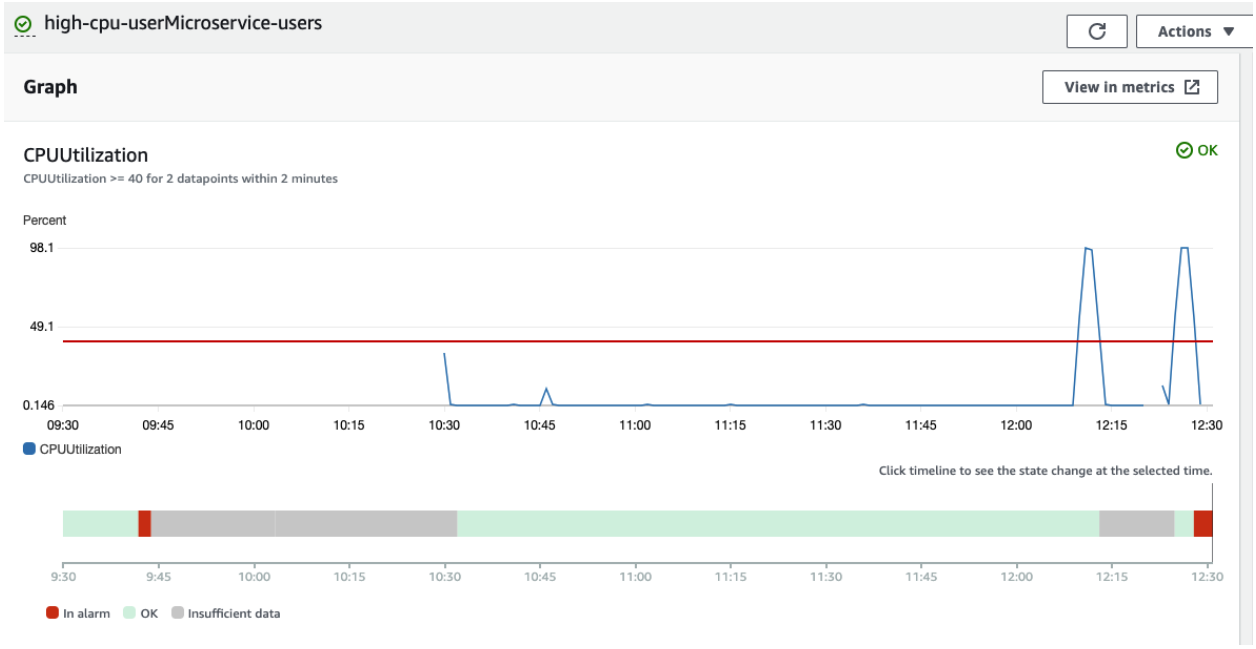
Sample screenshot of AWS logs for AWS ECS Service

For our application, hey is being used to conduct load testing and send heavy loads to microservice.



Sample screenshot of Load Testing sent to ECS service

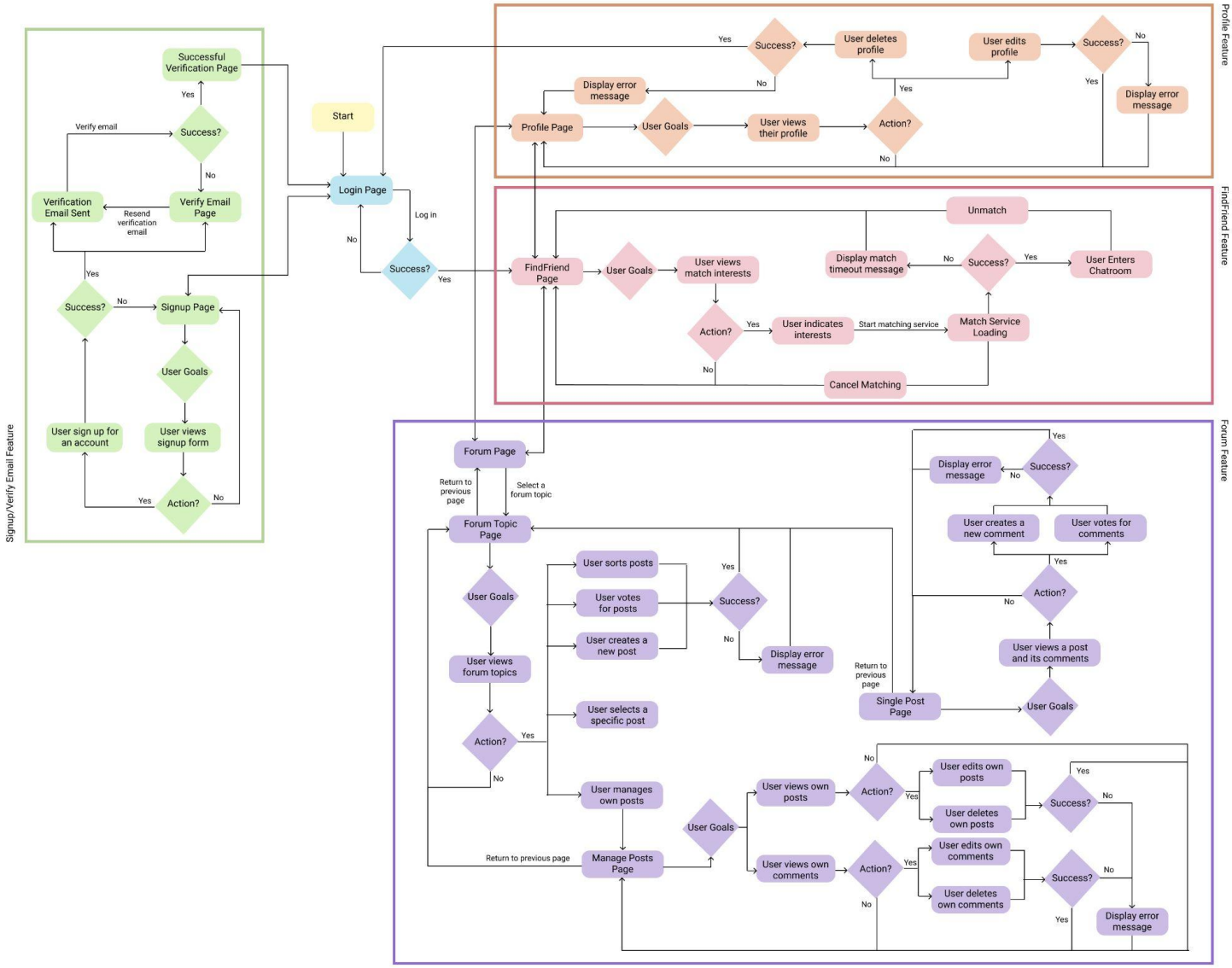
During our load testing experiments, our application is able to sustain the heavy loads when 30,000 requests are being sent in within a 3 minute timeframe. The ECS service undergoes heavy workload and CPU utilisation has reached above 40% as shown in the diagram below.



Sample screenshot of CloudWatch monitoring alarm for ECS service

From this diagram, when there are 2 data points that have passed 40% CPU Utilization for Users ECS service, a new instance will be deployed to support the workload. With this horizontal scaling setup, this ensures that the application is able to handle high demands supposingly there are 30,000 NUS students using the application at the same time.

7.7 User Flow



User Flow Diagram

7.8 Application Flow

7.8.1 FindFriend Matching Algorithm

```
let FindFriendSchema = mongoose.Schema({
  userId: {
    type: String,
    unique: true,
    required: true,
  },
  matchUserId: {
    type: String,
    default: '',
  },
  isMatched: {
    type: Boolean,
    default: false,
  },
  roomId:{
    type: String,
  },
  isRandomSelection: {
    type: Boolean,
    default: false,
  },
  gender: [String],
  art: [String],
  sport: [String],
  music: [String],
  faculty: [String],
});
```

FindFriend Schema

Key Idea:

1) userId is the user identifier that has been passed in as authorization request headers from the HTTP request where he/she wants to find new friends in the application,
2) matchUserId is the user identifier of the matchedPerson where the user has matched,
3) isMatched is used to determine whether the user has been matched,
4) roomId is the room identifier where the user will be inside upon a successful match,
5) isRandomSelection is used to determine whether the user has match preferences
6) gender, art, sport, music, faculty are 5 separate individual arrays that can have matching fields inside (For example: gender can be of [“male”, “female”] if the user wants to match with male and female friend, [“male”] which means only wants to match with male friend or [] which is empty array with no preference for the gender category)

Note: Only 2 possible matching types available where user will be randomly matched if both parties have no match preferences or user with match preferences will only be matched with other user based on the most number of matching match preferences.

<u>Scenario 1 (When there is no other users online)</u>

The user will be first in line and his/her match preferences for the following categories (gender, music, sport, faculty, art) will be saved along with the userId in the FindFriend database. As no other users are

online, a message will be returned, indicating there is no suitable match found at the moment. If the user does not have any match preference, the 5 separate individual arrays will be empty and the default boolean value of isRandomSelection which is false will be overwritten to true and updated in the database.

Scenario 2 (When there are other users online)

Suppose there are other users who are already online and finding a match, this will mean that their match preferences data has been saved in the database.

The current user who just clicked the match button will have their match preferences tabulated to determine whether the user has any match preference.

Outcome A

If the user does not have any match preference, the default boolean value of isRandomSelection which is false will be overwritten to true. The user will only be matched with other users that do not have any match preference and not matched with anyone else randomly.

Outcome B

If the user has match preferences, the user will be compared against the other users' match preferences based on each category (gender, music, sport, faculty, art).

Note: Only users who have not been matched => checked using isMatched boolean will be considered for comparisons.

For each non-empty category array that the current user has matching fields inside, the database will retrieve only users who have at least 1 matching field in the same category array. This is done using Mongoose's aggregate function on each FindFriend database entry where the unwind operation will flatten the category array, match operation to filter for matching fields in the category array as well as checking whether the other online user has been matched before applying group operation to group the datasets by _id which is the other online user id. There will also be a counter to count the number of matching fields for other online user against the current user.

Hence, for each category, an array that consists of the matching users with the counters will be tabulated (matchingGender, matchingArt, matchingSport, matchingFaculty, matchingMusic). For example, a single data entry inside the gender category array, matchingGender will be in this format: { _id, genderCount } where _id is the other online user's id and the genderCount will be the total number of matching fields in gender category that the other online user matched with the current user.

Next, custom merge operations will be used to merge matchingGender, matchingArt, matchingSport, matchingFaculty, matchingMusic to combine these 5 arrays into a matchResults array. During the merge operation, entries with the same _id will be joined together. For example, if matchingGender has { _id: "12345", genderCount: 1} and matchingArt has { _id: "12345", artCount: 2 } and other 3 arrays do not have entries with _id: "12345", the resulting entry in matchResults array will be {_id:"12345", genderCount: 1, artCount: 2}

For each entry in the matchResults array, depending on the number of counters that each user has,there will be a totalCount variable to count the total number of matches across all the counters. Using the same above example, the entry with _id: “12345” has 2 matching categories, genderCount, artCount of {_id:”12345”, genderCount: 1, artCount: 2}, this results in the totalCount will be 3 after adding genderCount and artCount where the final output will be {_id:”12345”, genderCount: 1, artCount: 2, totalCount: 3}

Next, we will sort the matchResults array based on the totalCount and only the entry with the highest count will be returned. Suppose there is a tie breaker, the online user that has been registered in FindFriend database first will be returned as the matched user.

Applicable to both Outcome A and B

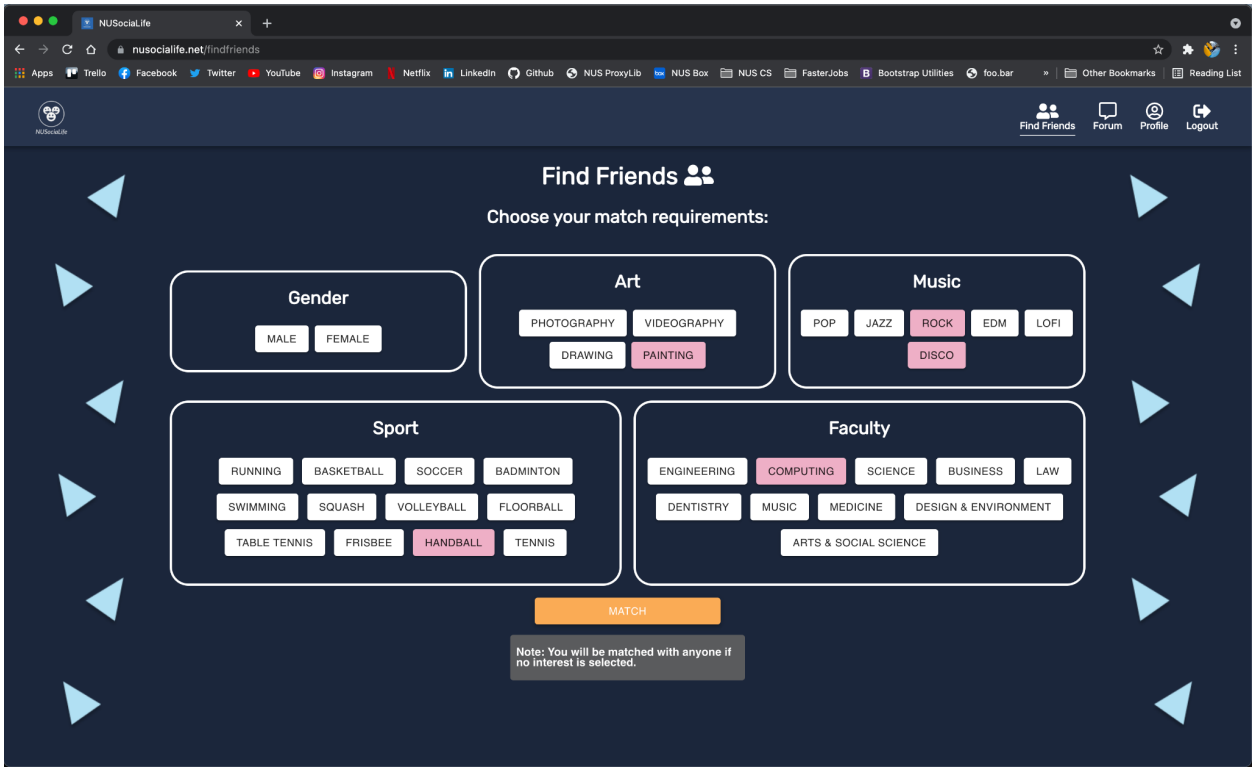
If there is a match, a new room will be created and both the users isMatched field will be set to true, the matchUserId and roomId will be updated correspondingly for both users too. The room will also hold the ids of the 2 users which will be saved into the Room database. Suppose the current user just clicked the match button for the first time, he/she will also be saved into the FindFriend database.

Otherwise, a message will be returned, indicating there is no suitable match found at the moment and if the user has not been saved in the FindFriend database, the user matched preferences along with userId will be saved in the FindFriend database.

During the chat session, suppose one user clicked the end session button, the session will end where the room will be deleted from the Room database and the 2 users will be deleted from FindFriend database too.

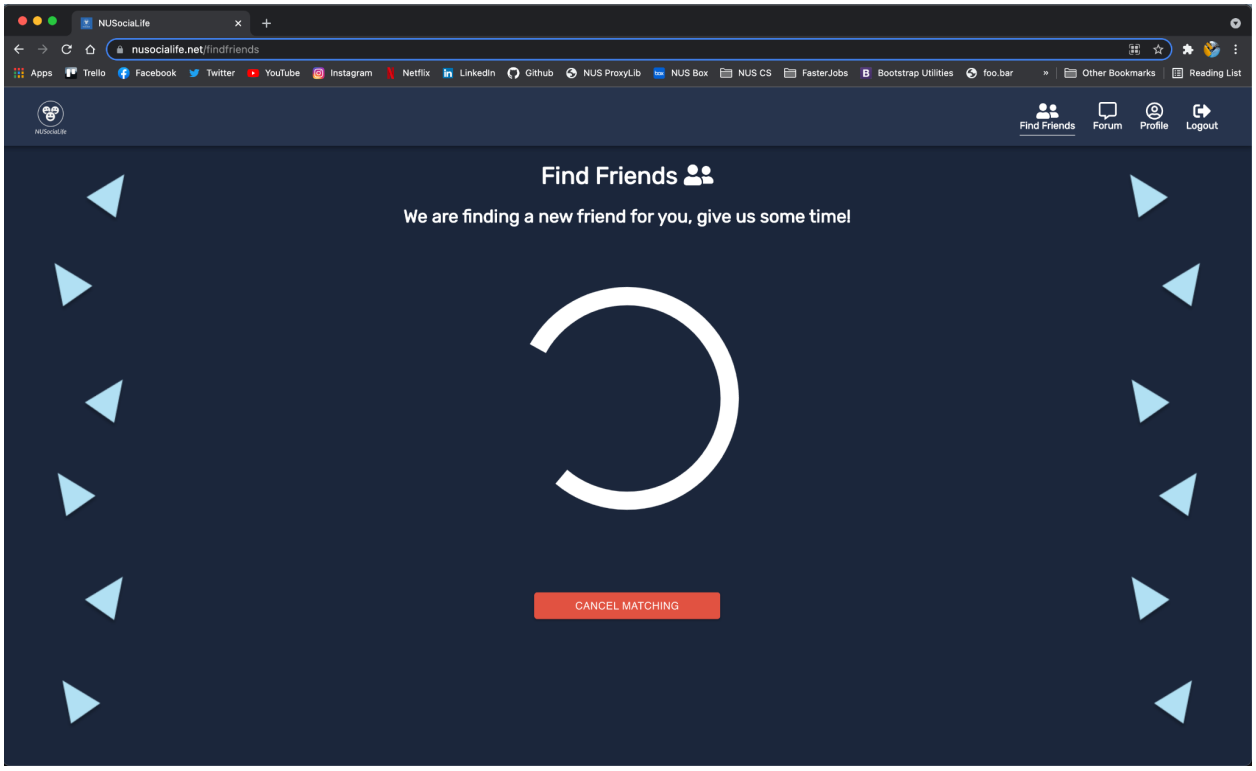
7.8.2 FindFriend Match/Chat UI

This subsection describes the flow to match and chat with another user.



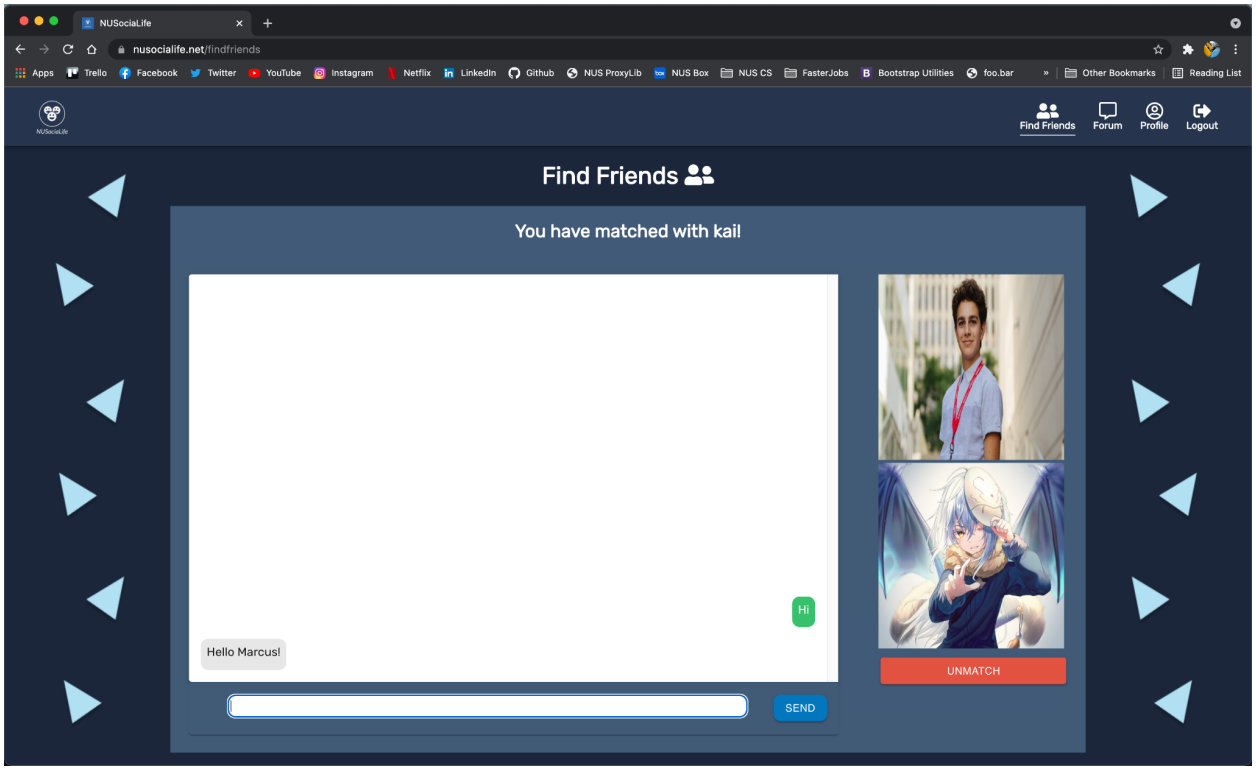
Find Friends Page

First, the user will select the range of interests for their match requirements. The user can choose to start the matching service without any interests indicated as well (the matching service will match the user with anyone). Next, the user will click on the ‘MATCH’ button located at the bottom of the page to start the matching service.



Matching In Progress

Upon clicking ‘MATCH’, the user will see the matching service in progress. The user will be able to cancel the matching service by clicking on the ‘CANCEL MATCHING’ button located at the bottom of the page.



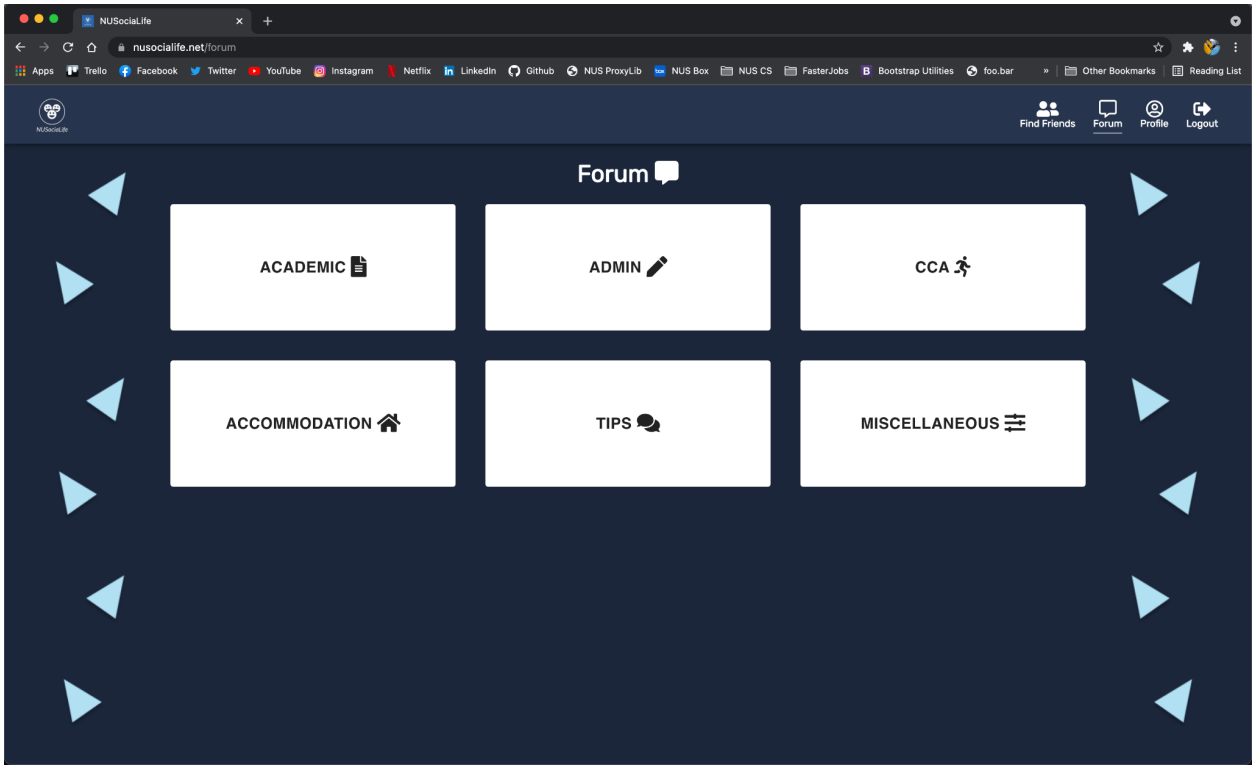
Chat

If the matching service successfully matches one user with another, both users will enter into a chatroom. They will be able to chat with one another in real-time.

Upon clicking on the ‘UNMATCH’ button located at the bottom right of the page, both users will be disconnected and brought back to the Find Friends page.

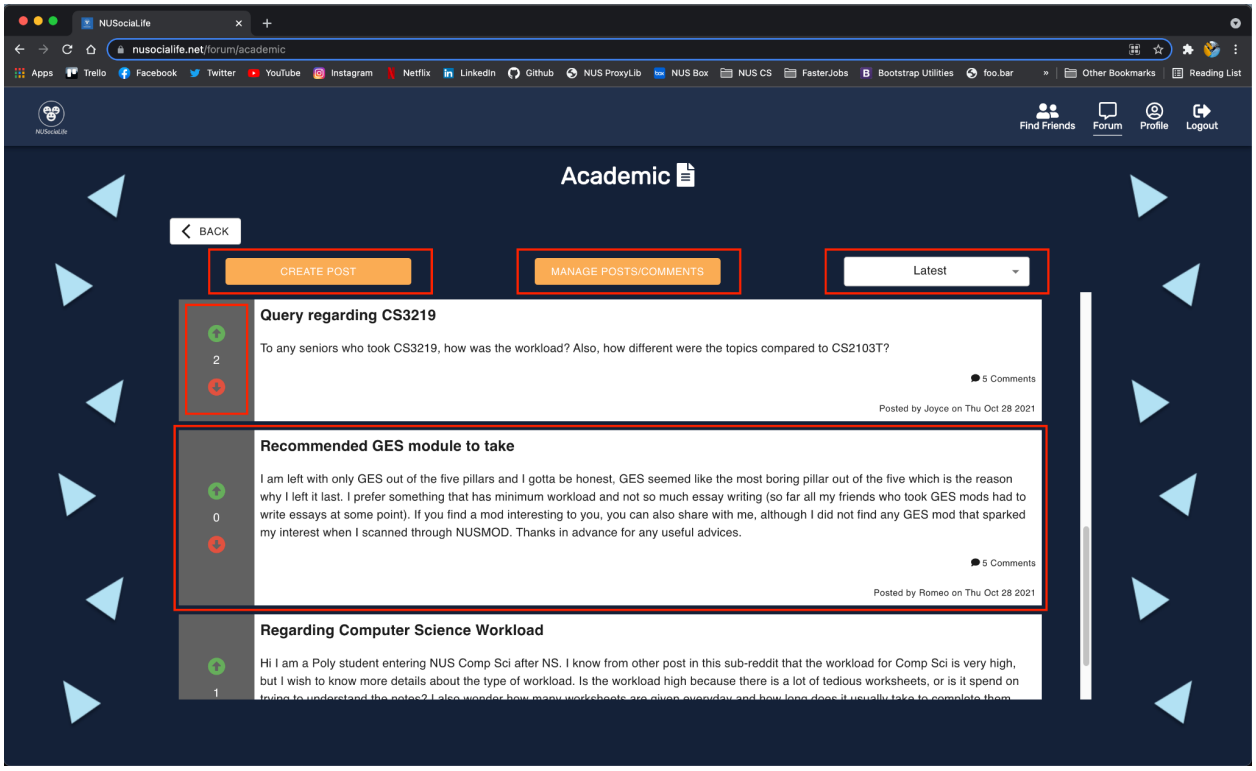
7.8.3 Forum UI

This subsection describes the flow of the forum page.



Forum Page

First, the user will click on any of the forum topics button to view the topic of interest. The user will be redirected to the individual forum topic page.

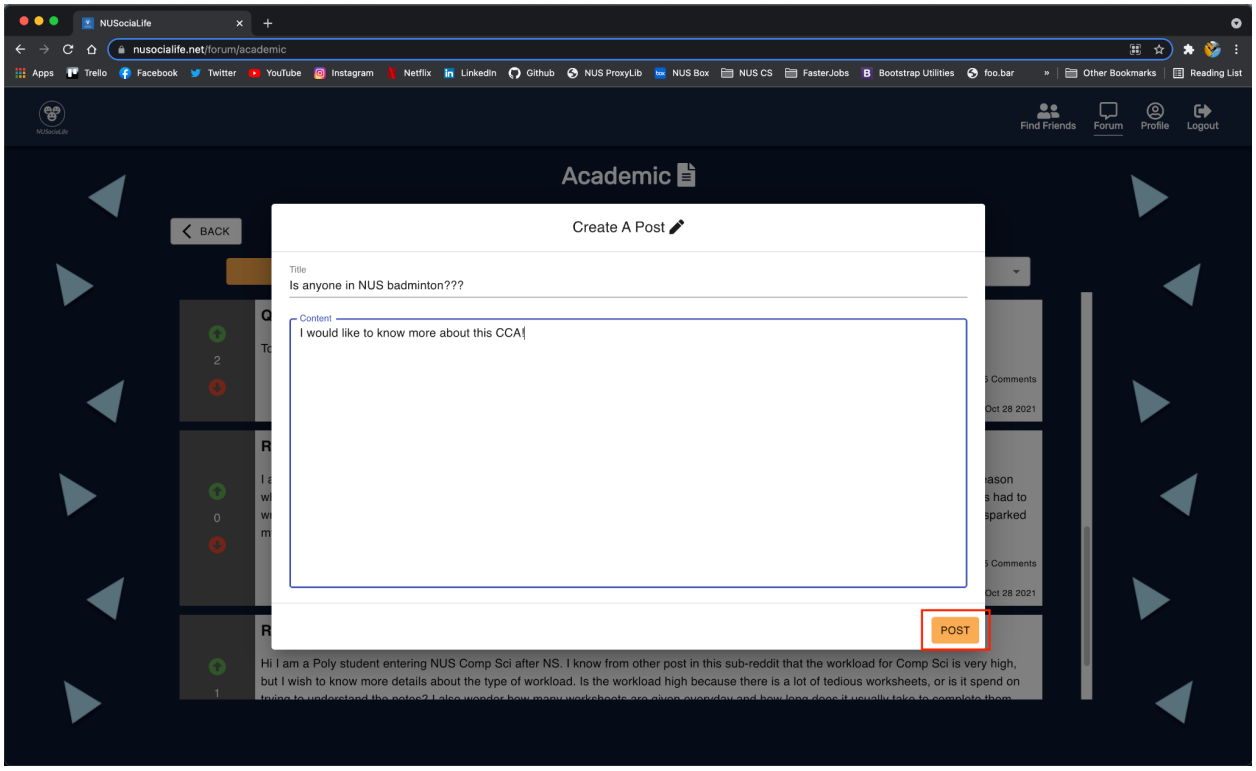


Individual Forum Topic Page

In the individual forum topic page, the user is able to execute various actions, specifically:

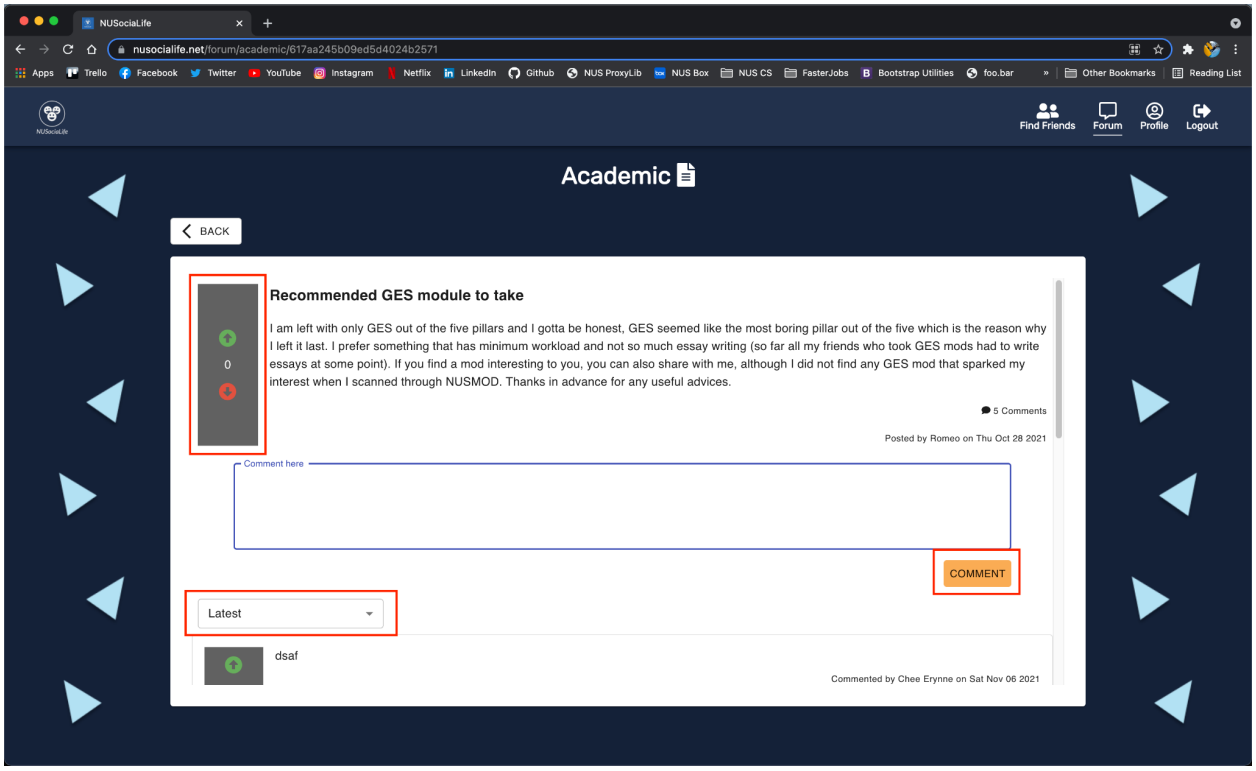
1. Create a new post
    - Click on the ‘CREATE POST’ button located at the top left of the page
2. Upvote an existing post
    - Click on the up arrow on the left of each post
3. Downvote an existing post
    - Click on the down arrow on the left of each post
4. Sort posts based on votes/date
    - Click on the ‘SORT BY’ button located at the top right of the page
5. View an individual post and its comments
    - Click on a specific post
6. Manage his/her posts and comments
    - Click on the ‘MANAGE POSTS/COMMENTS’ button located at the top left of the page

The highlight of the individual forum topic page is the ability for users to execute multiple actions all within a single page. This reduces the trouble of navigating to different pages just to execute the various actions. Furthermore, there are ‘BACK’ buttons located at the top left of every page under Forum for easy navigation between the various Forum pages.



Create Post Dialog

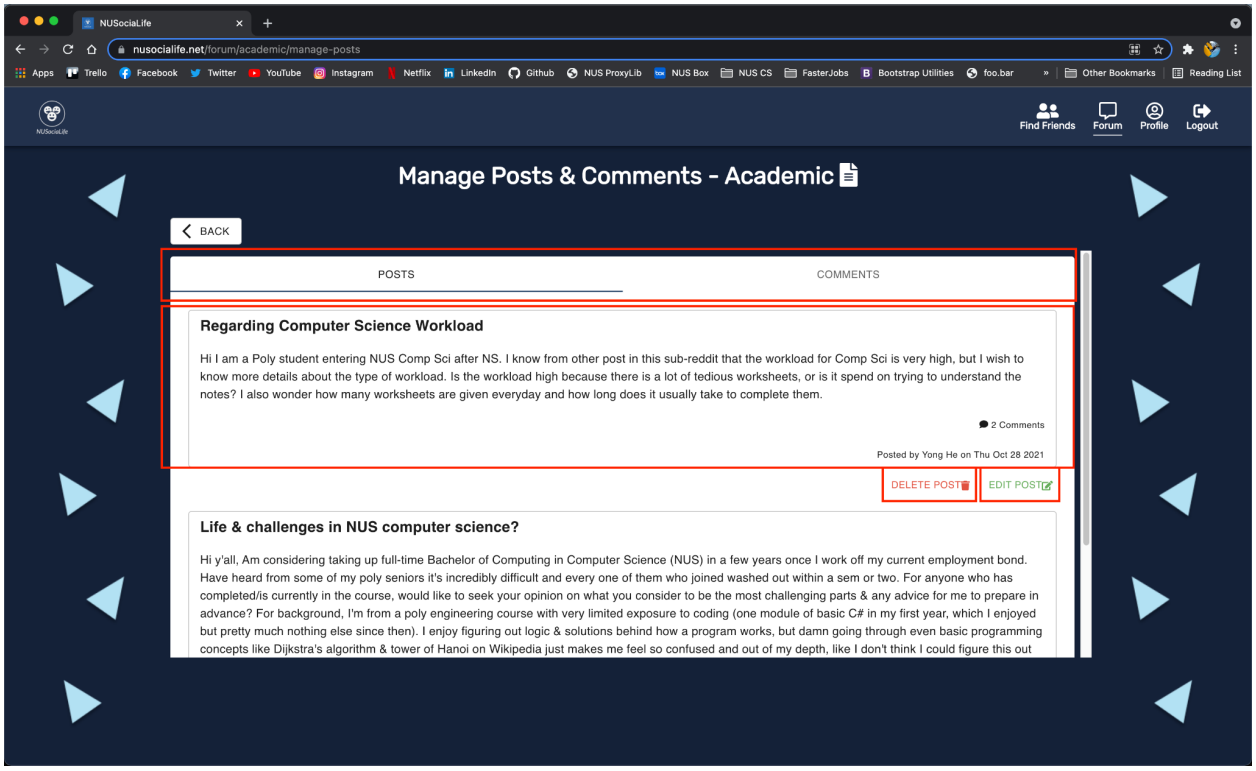
The user is able to create a new post with a title and content. Upon clicking on the ‘POST’ button located at the bottom right of the dialog, a new post will be created.



Individual Post Page

In the individual post page, the user is able to execute various actions:

1. Upvote the post
    - Click on the up arrow on the top left of the post
2. Downvote the post
    - Click on the down arrow on the top left of the post
3. Add a new comment to the post
    - Enter his/her comment into the ‘Comment here’ box before clicking on the ‘COMMENT’ button at the bottom right of the box
4. Upvote an existing comment (of the post)
    - Click on the up arrow on the left of each comment
5. Downvote an existing comment (of the post)
    - Click on the down arrow on the left of each comment
6. Sort comments based on votes/date
    - Click on the ‘SORT BY’ button

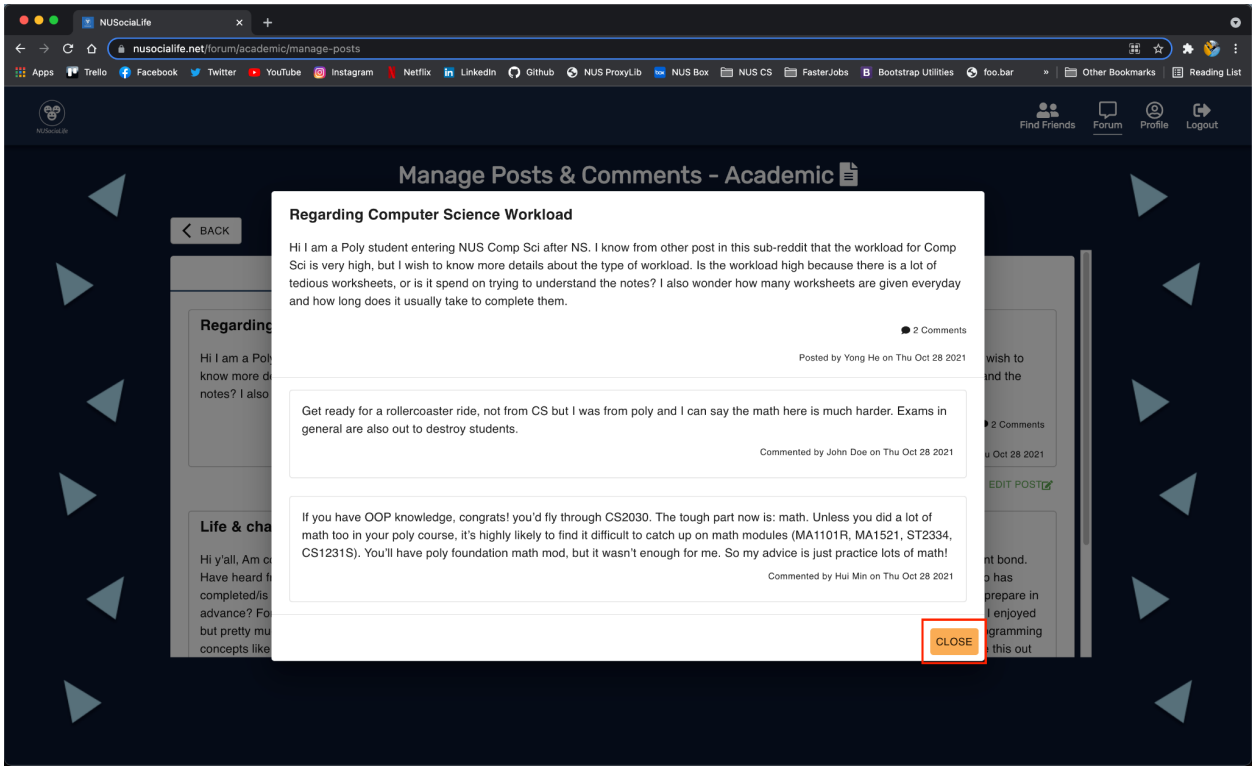


Manage Posts/Comments Page

In the manage posts/comments page, the user is able to see all the posts and comments created by them. Various actions can be executed on this page, specifically:

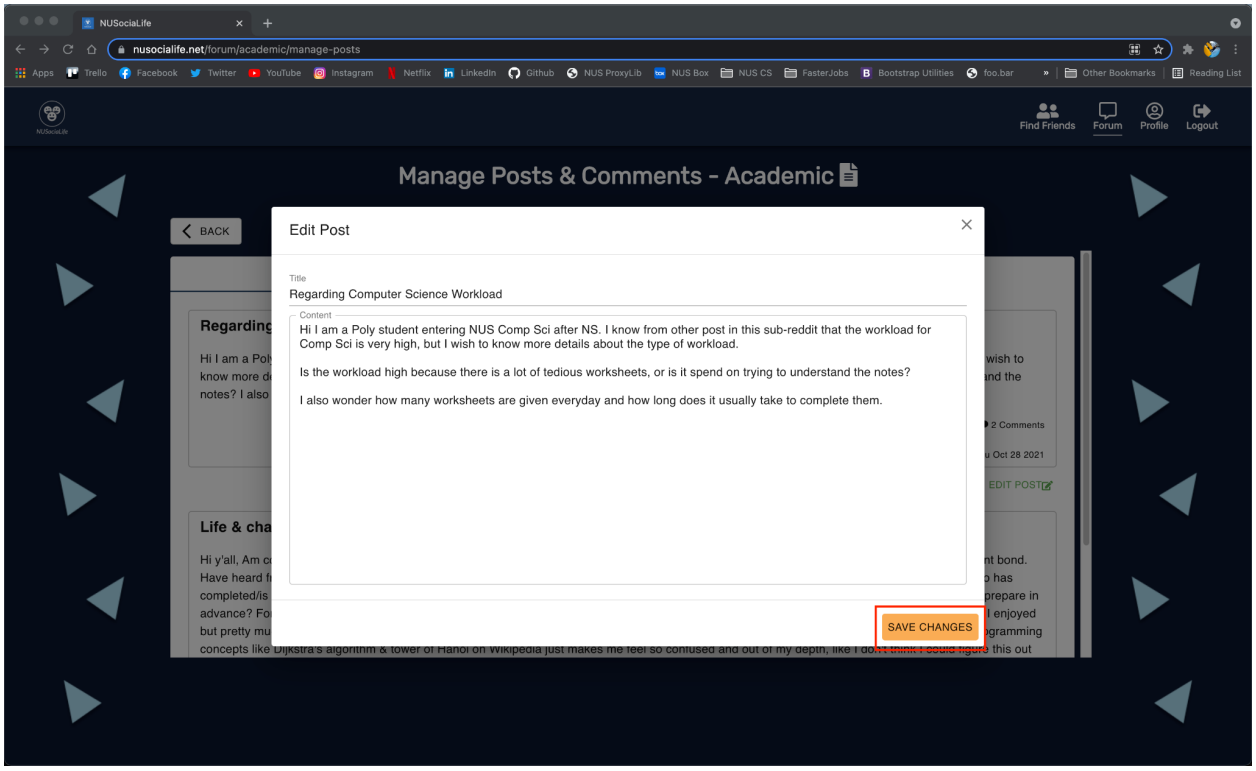
1. Toggle between the user's posts and comments
    - Click on the 'POSTS' and 'COMMENTS' tab
2. View the specific post and its comments
    - Click on a specific post
3. Delete the post/comment
    - Click on the 'DELETE' button located at the bottom right of each post/comment
4. Edit the post/comment
    - Click on the 'EDIT' button located at the bottom right of each post/comment

The manage posts/comments page allows users to easily toggle between his/her posts and comments. We decided to implement a tab UI for the user's posts and comments instead of having 2 separate pages as we felt that this is more convenient for the users. This reduces the hassle of having to navigate to different pages to view his/her posts and comments.



Specific Post Dialog

Upon clicking on a specific post, a dialog of the post and its comments will open. The motivation behind this dialog is for users to view their post within the same page. This saves them the trouble of having to navigate to the 'Individual Post' page just to view the post and its comments.



Edit Post Dialog

Upon clicking on the ‘EDIT POST’ button, the user will be able to edit the post’s title and/or content. Clicking on the ‘SAVE CHANGES’ button, located at the bottom right of the dialog, will allow the new post’s title/content to be saved.

8. Future Plans

8.1 Features

This section describes possible extensions we are considering to incorporate into our app in the future.

8.1.1 Video Call

One extension to our current app is to incorporate real-time video calls into the FindFriend feature. Currently, when 2 users successfully match and enter the chatroom, they can communicate with each other through real-time chat (text messages). With the video call feature, users can have the option to turn on their camera and communicate directly through the video call. This allows users to have a more interactive experience as they will be able to see the other user while communicating with them.

Furthermore, with the optionality of turning on their cameras, users who are less comfortable can choose not to turn on their cameras. This provides greater flexibility and a better user experience.

8.1.2 Upload/Download Files

Another extension to our app is to incorporate an uploading and downloading of files into the Forum feature. This enables the users to attach any relevant documents to their posts and provides greater convenience for them in sharing their queries. On some occasions, the users can upload images along with their posts. Research has shown the images can capture attention and convey large amounts of information in a relatively short amount of time. With this functionality in place, this allows the user to communicate their ideas more effectively.

8.1.3 Admin Role

The next extension to our app is to incorporate different access roles. This would mean giving different permissions for different users to access certain parts of the app. As our app grows to cater to a larger audience in the long run, it might be wiser to have a user who has privileged access to the more secure and important features of our app. This includes the deletion of users, deletion of inappropriate posts and comments etc. This ensures that there is a user that can manually manage the app instead of deleting inappropriate entries from the database which can be quite a hassle for the development team to do so. Furthermore, it is not a good practice to directly tamper the data from the database hence, implementing an admin role for users allows the user to manage the data more consistently.

9. Reflection

9.1 Challenges

We faced many challenges while completing this project. Firstly, we were all unfamiliar with the various architecture styles which were required so that we can adhere to a certain structure when implementing the code. Hence, we did a thorough research on the different architecture styles to see which suits our project even better. Next, we were also overwhelmed with the number of features we had to implement. As we chose option 1, we had to design the entire app ourselves, from the features to the UI. In the beginning, we churned out many functional requirements for our app. However, in the midst of coding out the functionalities, we were taken aback by the amount of functions our app encompasses, especially the Forum page where there were numerous features. We also faced many miscommunications where the Frontend developers and Backend developers had to sync up on what the api call parameters should take and what response should be retrieved. Despite all the challenges, we managed to complete all the functional requirements before the given frequent meetings and sync up sessions. Lastly, the deployment was the most tedious part as none of us had experiences with AWS, Kubernetes etc. When we deployed to the cloud, many of our features broke due to issues with SSL Certificates. There was lots of waiting time involved as well due to our DNS server cache taking days to update, which delayed our testing of our deployed web application.

9.2 Key Takeaways

All these challenges have made us more resistant to setbacks and failure. As we are working in a team, teamwork and communication are pivotal in making this project a success. With a pandemic happening, it is even more difficult to meet up physically which clearly hinders effective communication. All the meetings have to be brought online but luckily, we were all very adaptive to changes where the meetings proceeded seamlessly. As we are all probably entering the Software Engineering industry in the future, this project has provided us a platform to experience weekly meetings where we update one another on our progress, which is a common practice in a Software Engineering team as part of a Agile workflow. While working as a team in creating our app, we have instinctively learnt many Software Engineering practices such as Agile Development process, Refactoring of Code, Continuous Integration/Deployment, scaling etc. which are highly relevant to what we will be doing in the future. In spite of all the challenges we have faced during the course of the project, we have gained and learned much more valuable and practical skills where we could effectively apply in the future.

10. Appendix

10.1 Glossary

Term	Definition
Web App / App	NUSociaLife
UI	User Interface (Frontend)
AWS	Amazon Web Services
VPC	Virtual Private Cloud
EC2	Elastic Computing Cloud
ECS	Elastic Container Service
ECR	Elastic Container Registry
ALB	Application Load Balancer
SSL	Secure Sockets Layer
FRs	Functional Requirements
NFRs	Non-Functional Requirements
CI	Continuous Integration
CD	Continuous Deployment
Users	NUS Students

10.2 Use Cases

UC1	User login using NUS Email
Actor	User
MSS	1. User navigates to the login page. 2. User enters NUS Email and password. 3. NUSociaLife log user in. 4. User is directed to the NUSociaLife home page. Use Case ends.
Extensions	2a. User enters invalid email or password.

	2a1. NUSociaLife informs the user about the invalid credentials. 2a2.User enters new credentials. Steps 2a1-2a2 are repeated until the credentials entered are valid. Use Case resumes from Step 3. 2b. User enters an unverified email. 2b1. NUSociaLife informs the user about the invalid email. 2b2.User enters a new email. Steps 2b1-2b2 are repeated until the email entered is valid and verified. Use Case resumes from Step 3.

UC2	User reset password
Actor	User
MSS	1. User navigates to the login page. 2. User chooses to reset his password. 3. NUSociaLife sends an email to the user containing a new temporary password. Use Case ends.

UC3	User sign up using NUS Email
Actor	User
MSS	1. User navigates to the sign up page. 2. User enters username, email and password. 3. User is directed to the email verification page. 4. NUSociaLife sends a verification email to the user's email. Use Case ends.
Extensions	2a. User enters invalid username, email or password. 2a1. NUSociaLife informs the user about the invalid credentials. 2a2. User enters a new username, email or password. Steps 2a1-2a2 are repeated until the data entered are valid. Use Case resumes from Step 3.

UC4	User verifies email
Actor	User
MSS	1. User navigates to his email. 2. User clicks on the verification link in his email inbox. 3. User is directed to the NUSociaLife successful email verification page. Use Case ends.
Extensions	2a. User clicks on an expired or invalid verification link. 2a1. User is directed to the NUSociaLife email verification page. 2a2. User clicks on the resend verification email link.

	2a3. NUSociaLife sends a verification email to the user's email. Use Case resumes from Step 2.

UC5	User log out of NUSociaLife
Actor	User
Pre Condition	User is logged in
MSS	1. User chooses to logout. 2. User is directed to the NUSociaLife login page. Use Case ends.

UC6	Navigating around NUSociaLife
Actor	User
Pre Condition	User is logged in
MSS	1. User chooses the page he wants to go to from the navigation bar. 2. NUSociaLife directs the user to the chosen page. Use Case ends.

UC7	Match users (Find Friends feature)
Actor	User
Pre Condition	User is logged in
MSS	1. User navigates to the Find Friends pages. 2. User chooses his match preferences. 3. User requests NUSociaLife to match him. 4. NUSociaLife finds a match for the user. 5. NUSociaLife invites the user to a real-time chat room with the matched user. Use Case ends.
Extensions	2a. User does not indicate any match preferences. 2a1. User requests NUSociaLife to match him randomly. Use Case resumes from Step 4. 4a. User requests to cancel the match. 4a1. NUSociaLife cancels the match. 4a2. NUSociaLife redirects the user to the Find Friends page. Use Case ends. 4b. NUSociaLife detects a timeout after 30 seconds. 4b1. NUSociaLife cancels the match. 4b2. NUSociaLife redirects the user to the Find Friends page. Use Case resumes from Step 2. 5a. User chooses to end the chat.

	5a1. NUSociaLife closes the chat room. 5a2. NUSociaLife redirects the user to the Find Friends page. Use Case ends.

UC8	User views all posts of a topic
Actor	User
Pre Condition	User is logged in
MSS	1. User navigates to the Forum Page. 2. NUSociaLife displays a list of topics. 3. User chooses the topic of the posts he wishes to see. 4. NUSociaLife displays all the posts under the topic. Use Case ends.

UC9	User creates a new post
Actor	User
Pre Condition	User is logged in
MSS	1. User views the list of posts of a topic (UC8). 2. User requests to create a new post. 3. NUSociaLife prompts user to enter the content and title of the post. 4. User enters the content and title of the post. 5. User submits the post. 6. NUSociaLife displays the new post after the user submits. Use Case ends.
Extensions	5a. User leaves the content or title empty. 5a1. NUSociaLife informs the user that Title and Content are compulsory fields. Use Case resumes from step 4.

UC10	User views his own posts
Actor	User
Pre Condition	User is logged in
MSS	1. User views the list of posts of a topic (UC8). 2. User requests to view his own posts. 3. NUSociaLife displays all the posts written by him. Use Case ends.

UC11	User edits a post
Actor	User
Pre Condition	User is logged in
MSS	1. User views his own posts (UC10). 2. User requests to edit his post. 3. NUSociaLife displays the title and content of his post. 4. User edits the fields. 5. User submits the edited post. 6. NUSociaLife updates the post and updated fields are displayed in the post. Use Case ends.
Extensions	5a. User submits a post with an empty title or content. 5a1. NUSociaLife informs the user that Title and Content are compulsory fields. Use Case resumes from step 4

UC12	User deletes a post
Actor	User
Pre Condition	User is logged in
MSS	1. User views his own posts (UC10). 3. User requests to delete his own post. 4. NUSociaLife deletes the post and the post is no longer displayed in the Forum. Use Case ends.

UC13	User upvotes/downvotes a post
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of posts of a topic (UC8). 2. User requests to downvote/upvote a particular post. 3. NUSociaLife displays the updated vote count. Use Case ends.
Extensions	2a. User upvotes/downvotes his own post. 2a1. NUSociaLife alerts the user that he cannot upvote/downvote his own post. Use Case ends. 2b. User upvotes/downvotes more than once. 2b1. NUSociaLife alerts the user that he/she cannot upvote/downvote more than once. Use Case ends.

UC14	User sorts posts
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of posts of a topic (UC8). 2. User requests to sort posts by date or by upvote popularity. 3. NUSociaLife displays the sorted list of posts. Use Case ends.

UC15	User views comments in a post
Actor	User
Pre Condition	User is logged in
MSS	1. User navigates to the Forum Page. 2. NUSociaLife displays a list of topics. 3. User chooses the topic of the posts he/she wishes to see. 4. User selects a particular post. 5. NUSociaLife displays the post details and all the comments under the post. Use Case ends.

UC16	User creates a new comment
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of comments under a post (UC15). 2. User requests to create a new comment. 3. NUSociaLife prompts the user to enter the content of the comment. 4. User enters the content. 5. User submits the comment. 6. NUSociaLife displays the comment. Use Case ends.
Extensions	5a. User leaves the comment empty. 5a1. NUSociaLife informs the user that a comment cannot be empty. Use Case resumes from step 4.

UC17	User views his own comments
Actor	User
Pre Condition	User is logged in

MSS	1. User views the list of posts of a topic (UC8). 2. User requests to view his own comments. 3. NUSociaLife displays all the comments written by him. Use Case ends.

UC18	User edits a comment
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of his own comments (UC17). 2. User requests to edit his comment. 3. User edits the comment. 4. User submits the edited comment. 5. NUSociaLife updates the comment. 6. NUSociaLife displays the updated comment. Use Case ends.
Extensions	4a. User submits an empty comment. 4a1. NUSociaLife informs the user that a comment cannot be empty. Use Case resumes from step 3.

UC19	User deletes a comment
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of his own comments (UC17). 2. User requests to delete his comment. 3. NUSociaLife deletes the comment and the comment is no longer displayed. Use Case ends.

UC20	User upvotes/downvotes a comment
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of comments of a particular post (UC15). 2. User requests to downvote/upvote a particular comment. 3. NUSociaLife displays the updated vote count. Use Case ends.
Extensions	2a. User upvotes/downvotes his own comment. 2a1. NUSociaLife alerts the user that he/she cannot upvote/downvote his

	own comment. Use Case ends. 2b. User upvotes/downvotes a comment more than once. 2b1. NUSociaLife alerts the user that he/she cannot upvote/downvote a comment more than once. Use Case ends.

UC21	User sorts comments of a post
Actor	User
Pre Condition	User is logged in
MSS	1. User views a list of comments of a particular post (UC15). 2. User requests to sort comments by date or by upvote popularity. 3. NUSociaLife displays the sorted list of comments. Use Case ends.

UC22	User views his profile
Actor	User
Pre Condition	User is logged in
MSS	1. User navigates to the Profile page. 2. NUSociaLife displays the user's email but is uneditable. Use Case ends.

UC23	User edits profile
Actor	User
Pre Condition	User is logged in
MSS	1. User views his profile (UC22). 2. User requests to edit his profile particulars. 3. User submits the edited particulars. 4. NUSociaLife displays the updated profile particulars. Use Case ends.
Extensions	3a. User submits an empty particular. 3a1. NUSociaLife informs the user that particulars cannot be empty. Use Case resumes from step 1.

UC24	User uploads a profile picture
Actor	User

Pre Condition	User is logged in
MSS	1. User views his profile (UC22). 2. User requests to upload a profile picture. 3. User submits a profile picture. 4. NUSociaLife displays the updated profile and profile picture. Use Case ends.
Extensions	3a. User submits an invalid file type. 3a1. NUSociaLife informs the user about the invalid file type. 3a2. User submits a new profile picture. Steps 3a1-3a2 are repeated until the submitted profile picture is of a valid file type. Use Case resumes from step 4.

UC25	User changes password
Actor	User
Pre Condition	User is logged in
MSS	1. User views his profile (UC22). 2. User requests to change his password. 3. User submits a new password. 4. NUSociaLife displays a successful notification. Use Case ends.
Extensions	3a. User submits an invalid password. 3a1. NUSociaLife informs the user about the invalid password. 3a2. User submits a new password. Steps 3a1-3a2 are repeated until the user submits a valid password. Use Case resumes from step 4.

UC26	User deletes his account
Actor	User
Pre Condition	User is logged in
MSS	1. User views his profile (UC22). 2. User requests to delete his account. 3. NUSociaLife deletes his account. 4. NUSociaLife redirects the user to the NUSociaLife login page. Use Case ends.

11. References

1. What is agile? What is scrum?. (n.d.). cprime. Retrieved from https://www.cprime.com/resources/what-is-agile-what-is-scrum/

2. Tartila. (n.d.). Agile Development Methodology. Shutterstock. Retrieved from https://www.shutterstock.com/image-vector/agile-development-methodology-software-developments-sprint-1424774744

3. Quinn, S. (2019, April 17). Bulletproof node.js project architecture. Software on the road. Retrieved from https://softwareontheroad.com/ideal-nodejs-project-structure/
    - Backend folder structure

4. Richardson, C. (n.d.). What are microservices?. Microservice Architecture. Retrieved from https://microservices.io/patterns/decomposition/decompose-by-business-capability.html
    - Backend Architecture

5. Richardson, C. (n.d.). Pattern: Database per service. Microservice Architecture. Retrieved from https://microservices.io/patterns/data/database-per-service.html
    - Database architecture

6. Perry, Y. (2021, June 7). AWS High Availability: Compute, SQL and Storage. NetApp. Retrieved from https://cloud.netapp.com/blog/understanding-aws-high-availability-compute-sql-and-storage
    - High Availability

7. Richardson, C. (n.d.). Pattern: Service per container. Microservice Architecture. Retrieved from https://microservices.io/patterns/deployment/service-per-container.html