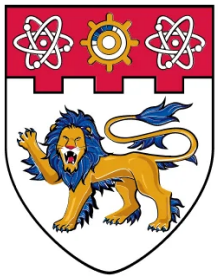


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

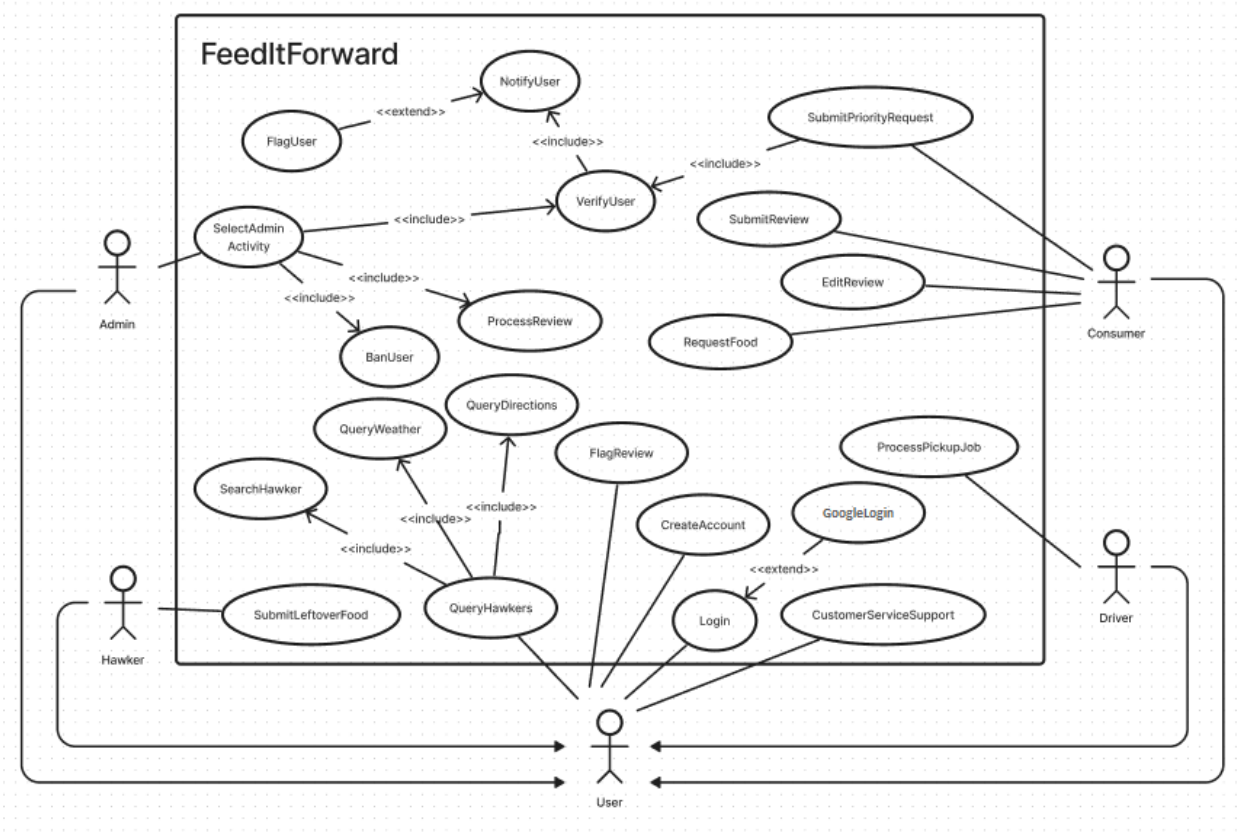
SC2006 – Software Engineering

Lab 3 Deliverables

Lab Group	SCSX
Team	FeedItForward
Members	Toh Jing Qiang (U2121442H)
	Toh Jing Hua (U21210232L)
	Tommy Wee Chung Kiat (U2120448F)
	Tay Jia Ying, Denise (U2122458K)
	Twu Pin Yang (U2121072C)
	Teh Min Ze (U2111370H)

1. Complete Use Case Model

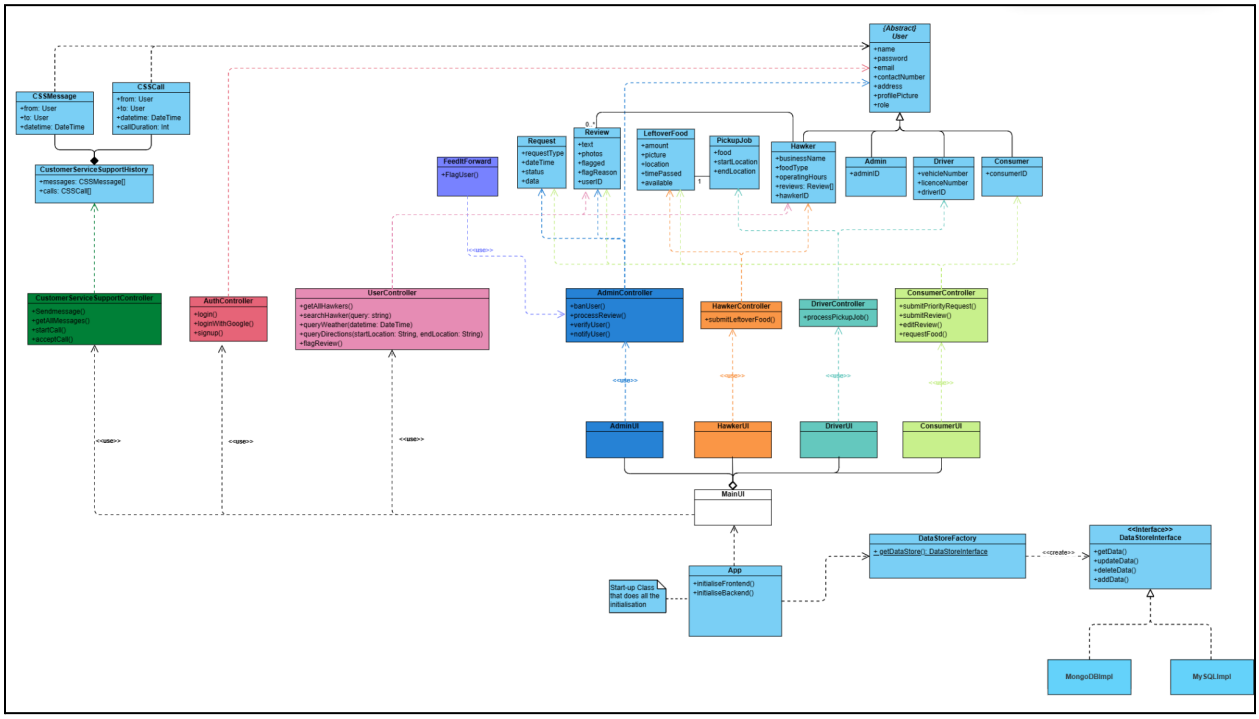
If the image is unclear, please refer to the raw png file that is uploaded together with this document.



2. Design Model

A. Class Diagram

If the image is unclear, please refer to the raw png file that is uploaded together with this document.



App

Start-up class for initialisation of the frontend and backend.

Operations:

- **initialiseFrontend():** renders the frontend User Interfaces (UI)
- **initialiseBackend():** starts the backend application and getDataStore() for data access and modification.

User Interfaces (UI)

User Interfaces are what the users see and interact with on the application.

- **MainUI:** This is the main screen for Users. Users can perform tasks which can be classified under 3 broad categories: Customer Service Support, Authentication, and FeedItForward's common app features that are accessible to Users of different roles.
- **AdminUI:** Screens specific and only accessible by Admins. E.g.: Admin Management Screen with the features to verify user, ban user, and process reviews.
- **HawkerUI:** Screens specific and only accessible by Hawkers. E.g.: SubmitLeftoverFoodScreen.
- **DriverUI:** Screens specific and only accessible by Drivers. E.g.: Accept Job Pickup Screen.

- **ConsumerUI:** Screens specific and only accessible by Consumers. E.g.: Submit Food Priority Request Screen.

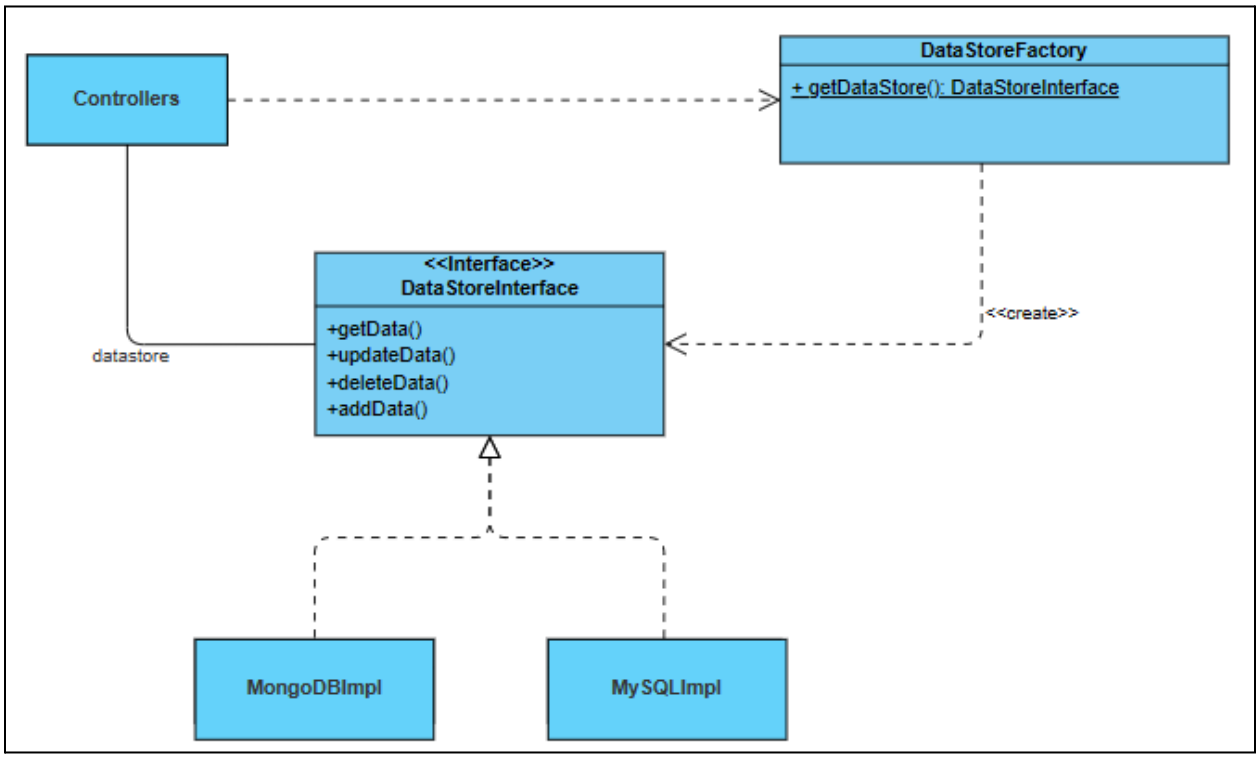
Controllers (Facade Pattern)

Application of the Facade Pattern where controllers serve as “interfaces” to specific implementation logic of business objects and services. There are multiple controllers, depending on the specific use case and/or role of the User.

- **AuthController:** This is the controller for Authentication. It lets Users login, login with Google Authentication, and sign up. It has access to Users.
- **UserController:** This is the controller for FeedItForward’s main usage. It lets Users get information on all hawkers, search for a particular hawker, query the current weather conditions, query the directions to a particular hawker, and flag a review for a particular hawker. It has access to the following entities: Hawker, and Review.
- **AdminController:** This is the controller for Admins. It lets Admins ban users, process reviews, verify users, and notify users. It has access to the following entities: Request, Review, and User.
- **HawkerController:** This is the controller for Hawkers. It lets Hawkers submit leftover food. It has access to the following entities: Request, and Hawker.
- **DriverController:** This is the controller for Drivers. It lets Drivers process pickup jobs. It has access to the following entities: PickupJob, and Driver.
- **ConsumerController:** This is the controller for Consumers. It lets Consumers submit priority requests for leftover food, submit reviews for a particular hawker, edit their reviews, and request for leftover food. It has access to the following entities: Request, Review, LeftoverFood, and Consumer.
- **CustomerServiceSupportController:** This is the controller for CustomerServiceSupport (CSS). It lets Users send messages to the CCS team, get all previous messages, start a call to CSS or accept a call from CSS. It has access to the following entities: CustomerServiceSupportHistory, CSSMessage, and CSSCall. These entities are assigned to respective Users.

Data Access Layer – DataStore (Factory Pattern + Strategy Pattern)

A DataStore is used to ensure that data access, modification and storage is always consistent at all times. Additionally, it serves as the application's global state with ease of accessibility. This allows the application to handle large amounts of data operations and users at scale.



The DataStore implements 2 design patterns – **Factory Pattern** and **Strategy Pattern**.

Factory Pattern

- Factory Pattern is applied to create static instances of data store classes without exposing the instantiation logic directly in the client code. The **DataStoreFactory** class serves as the factory responsible for creating instances of data store classes.

Strategy Pattern

- Strategy Pattern is applied to handle data access in a flexible and extensible manner. In our implementation, it allows developers to switch between different data storage systems (e.g., MySQL and MongoDB) without changing the core code. This is achieved through the following
    a. **DataStoreInterface**
    This is the abstraction that defines a common set of data access methods, such as getData(), updateData(), deleteData(), and addData()..

    b. **MySQLImpl and MongoDBImpl**

These are concrete implementations of the DataStoreInterface, tailored for MySQL and MongoDB, respectively. They encapsulate the specifics of data storage for each system. They can be used interchangeably by the application depending on which datastore implementation is required.

c. **DataStoreFactory**
This factory class is responsible for creating instances of the appropriate data store class based on a given type (e.g., 'mysql' or 'mongodb').

d. **Controllers**
This represents a high-level entity encompassing all controller classes in the application.
It has dependencies on the DataStoreFactory to create data stores and is associated with the DataStoreInterface to indicate that controllers interact with data stores through this common interface.

Other Design Patterns

1. **Observer Pattern**

Observer Pattern is used to facilitate notification of new pickup jobs (**Subject**) for “subscribed” drivers (**Observers**) to accept or reject.

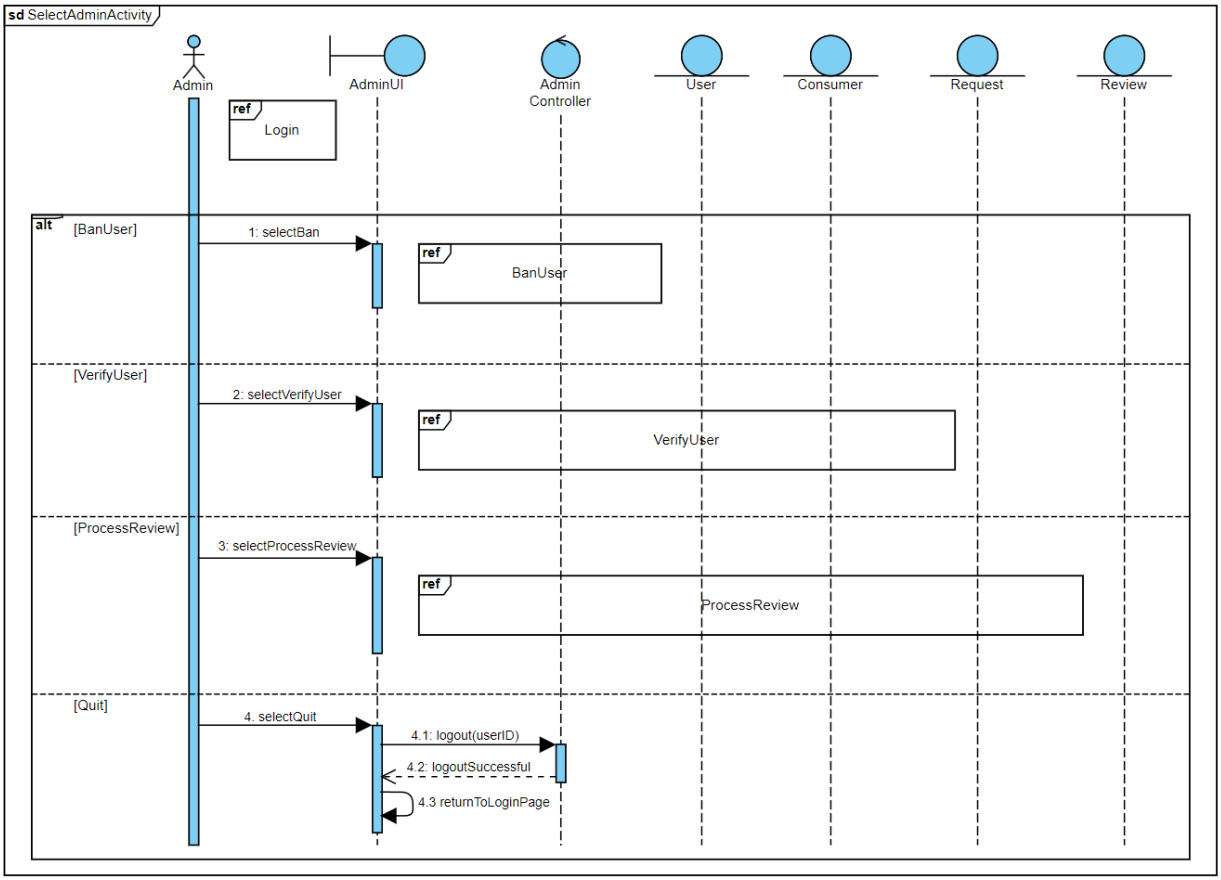
2. **Publisher-Subscriber Pattern**

For websocket connection between different Users for text communication in CustomerServiceSupport

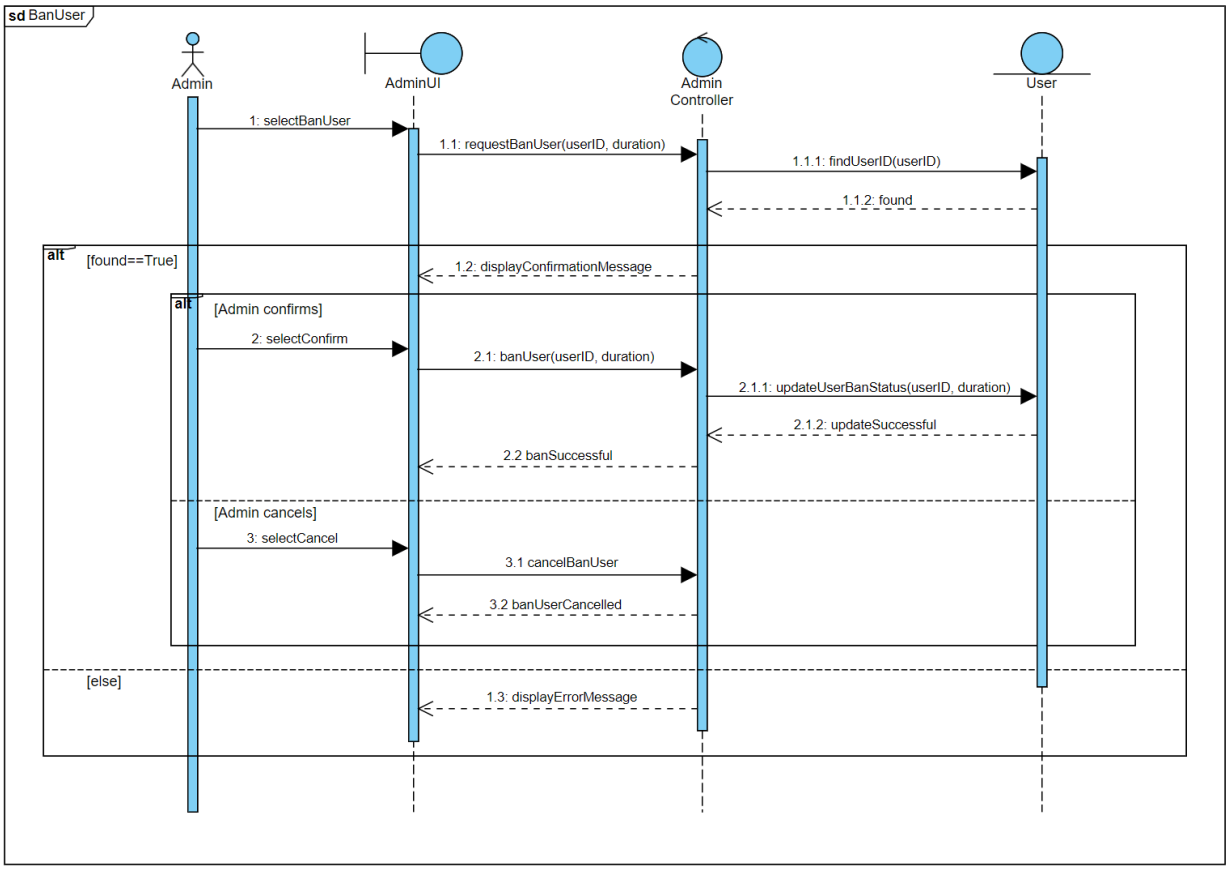
B. Sequence Diagrams

I. For Use Cases under I (Admin)

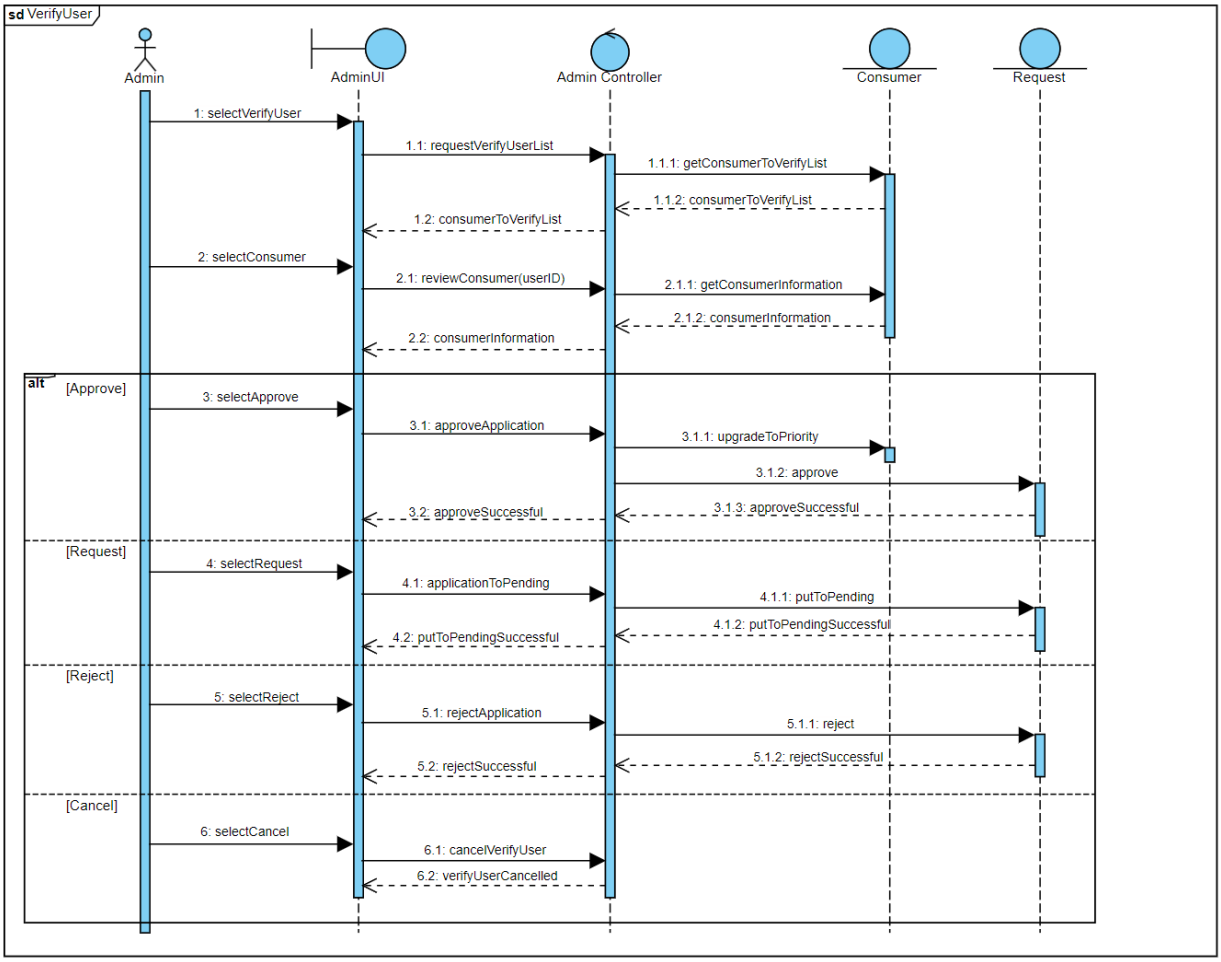
I.I SelectAdminActivity



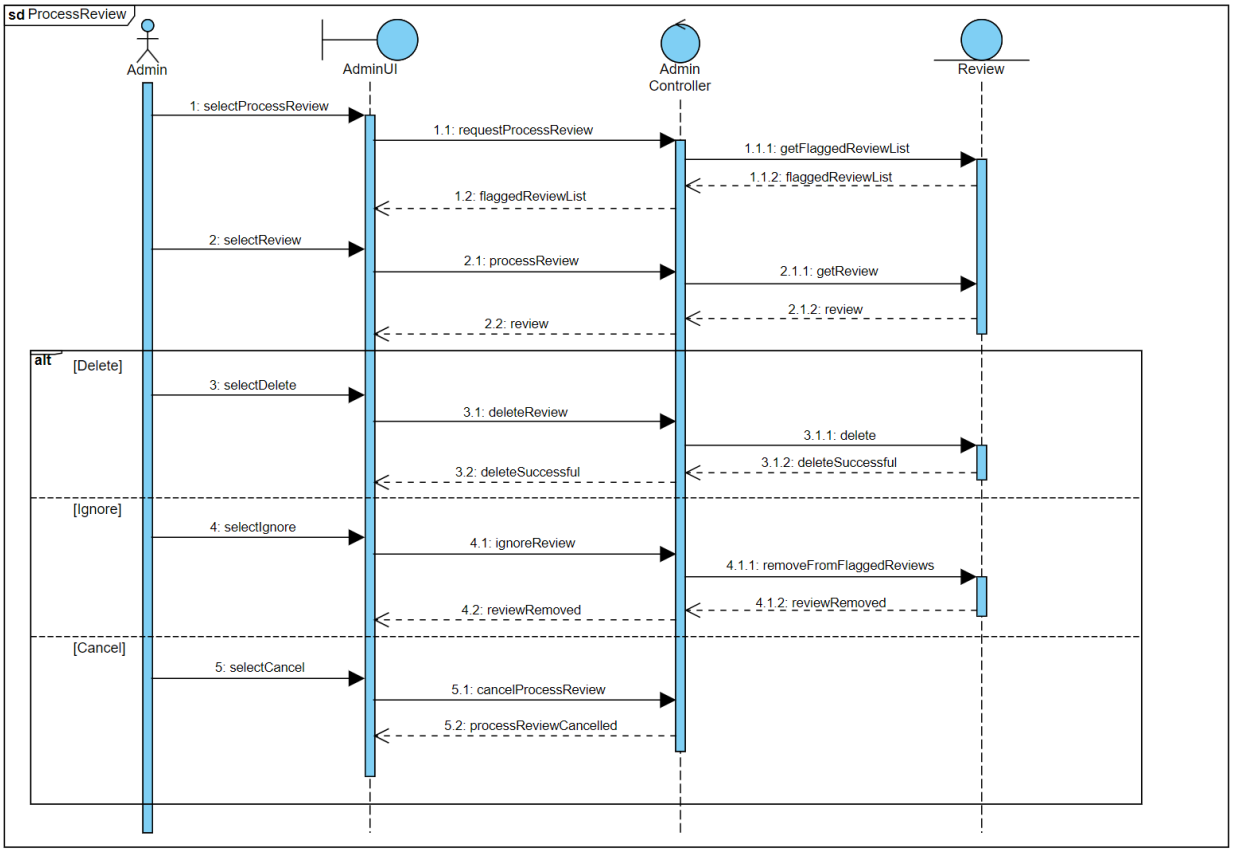
I.II BanUser


sd BanUser
Admin
AdminUI
Admin Controller
User
1: selectBanUser
1.1: requestBanUser(userID, duration)
1.1.1: findUserID(userID)
1.1.2: found
alt
[found==True]
1.2: displayConfirmationMessage
alt
[Admin confirms]
2: selectConfirm
2.1: banUser(userID, duration)
2.1.1: updateUserBanStatus(userID, duration)
2.1.2: updateSuccessful
2.2 banSuccessful
[Admin cancels]
3: selectCancel
3.1 cancelBanUser
3.2 banUserCancelled
[else]
1.3: displayErrorMessage

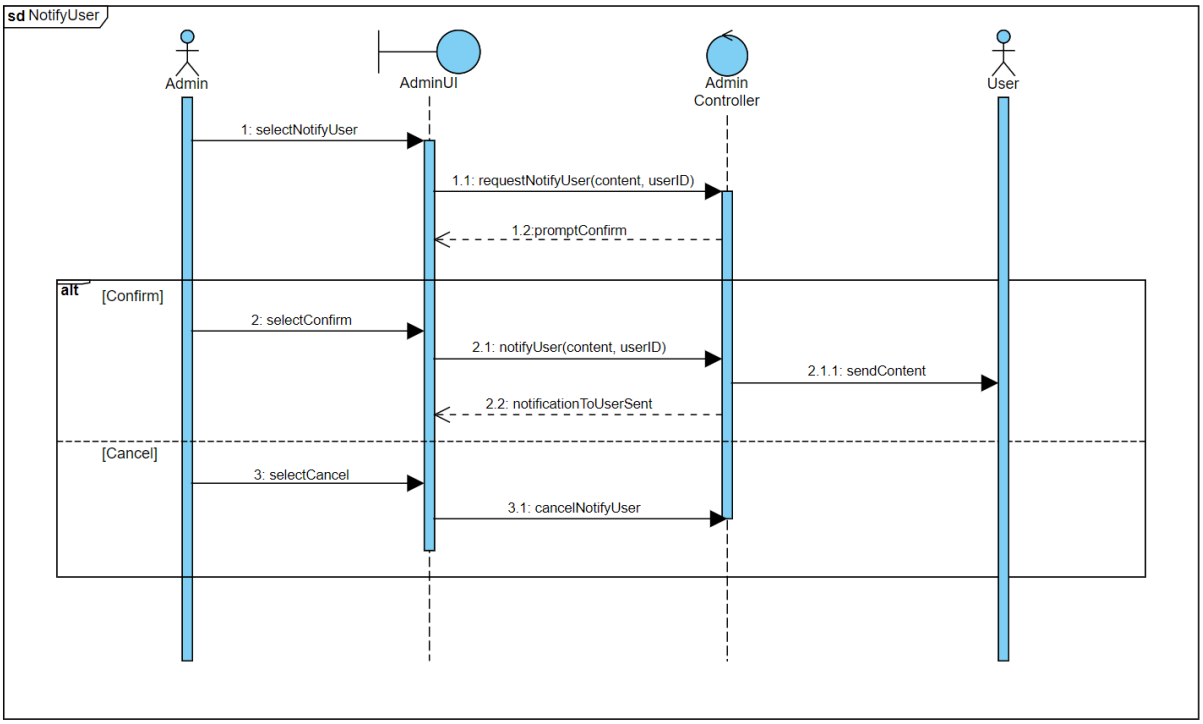
I.III VerifyUser



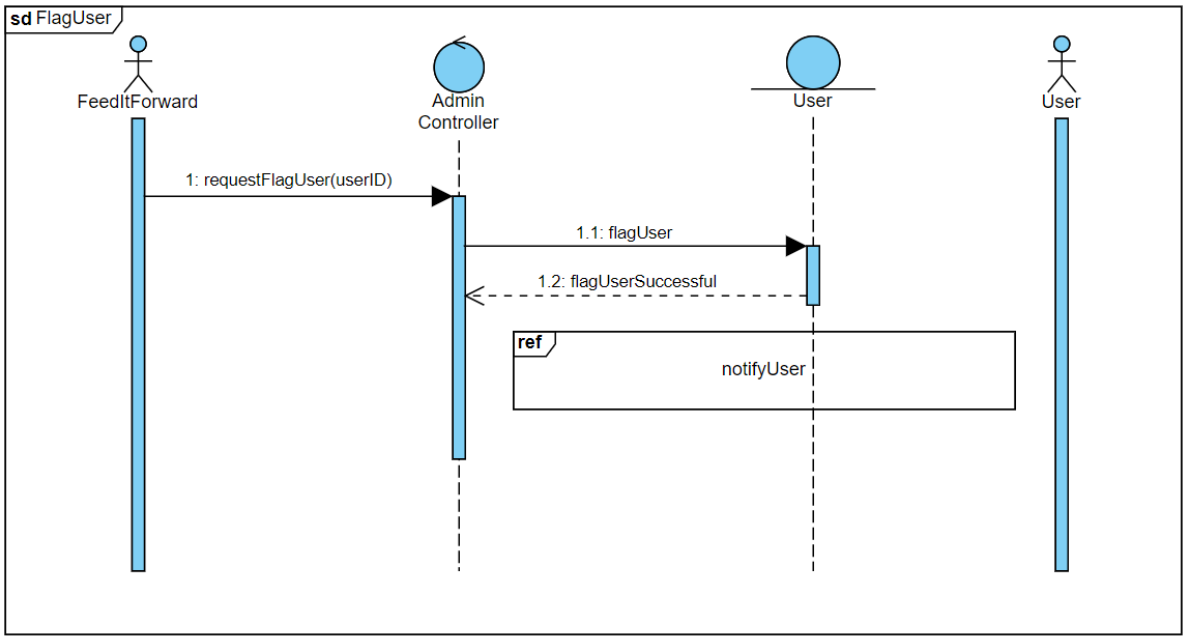
I.IV ProcessReview



I.V NotifyUser

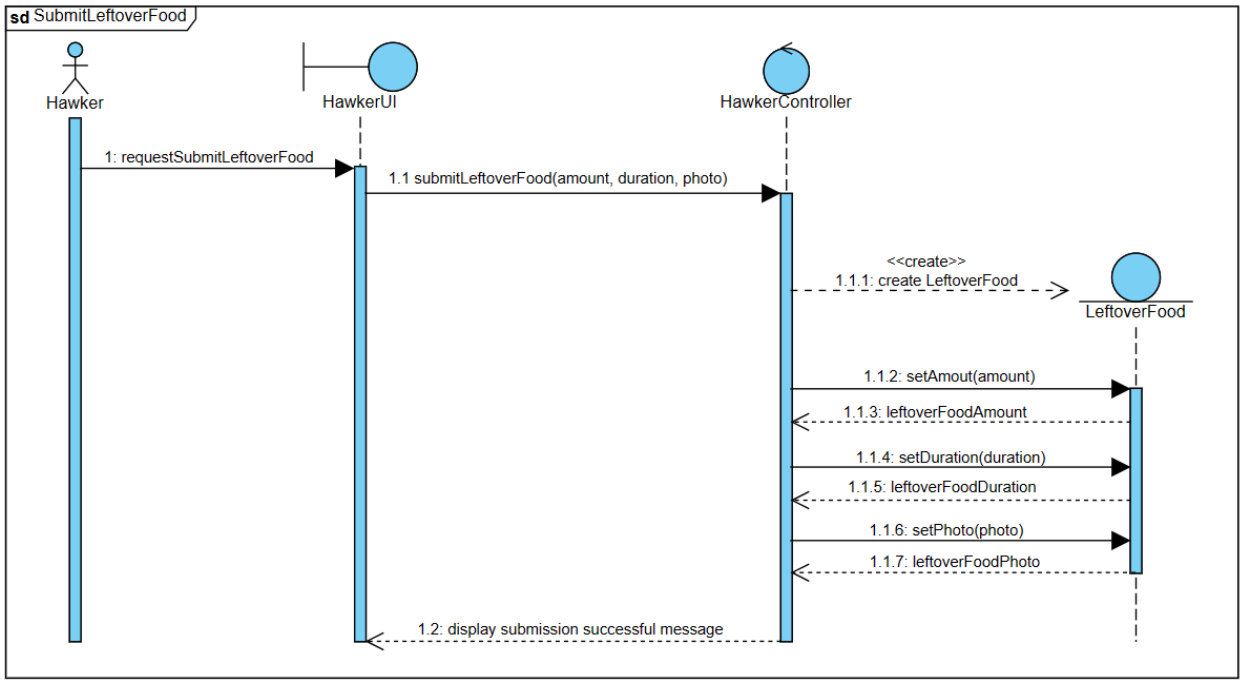


I.VI FlagUser



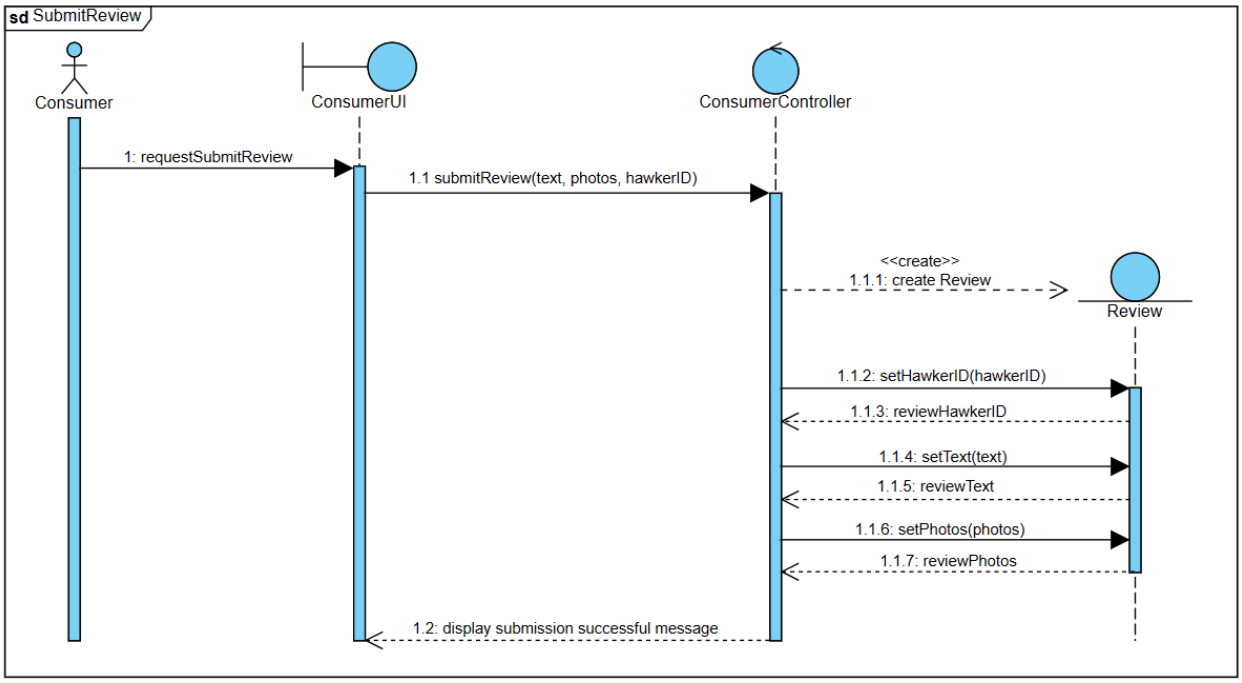
II. For Use Cases under II

II.I SubmitLeftoverFood

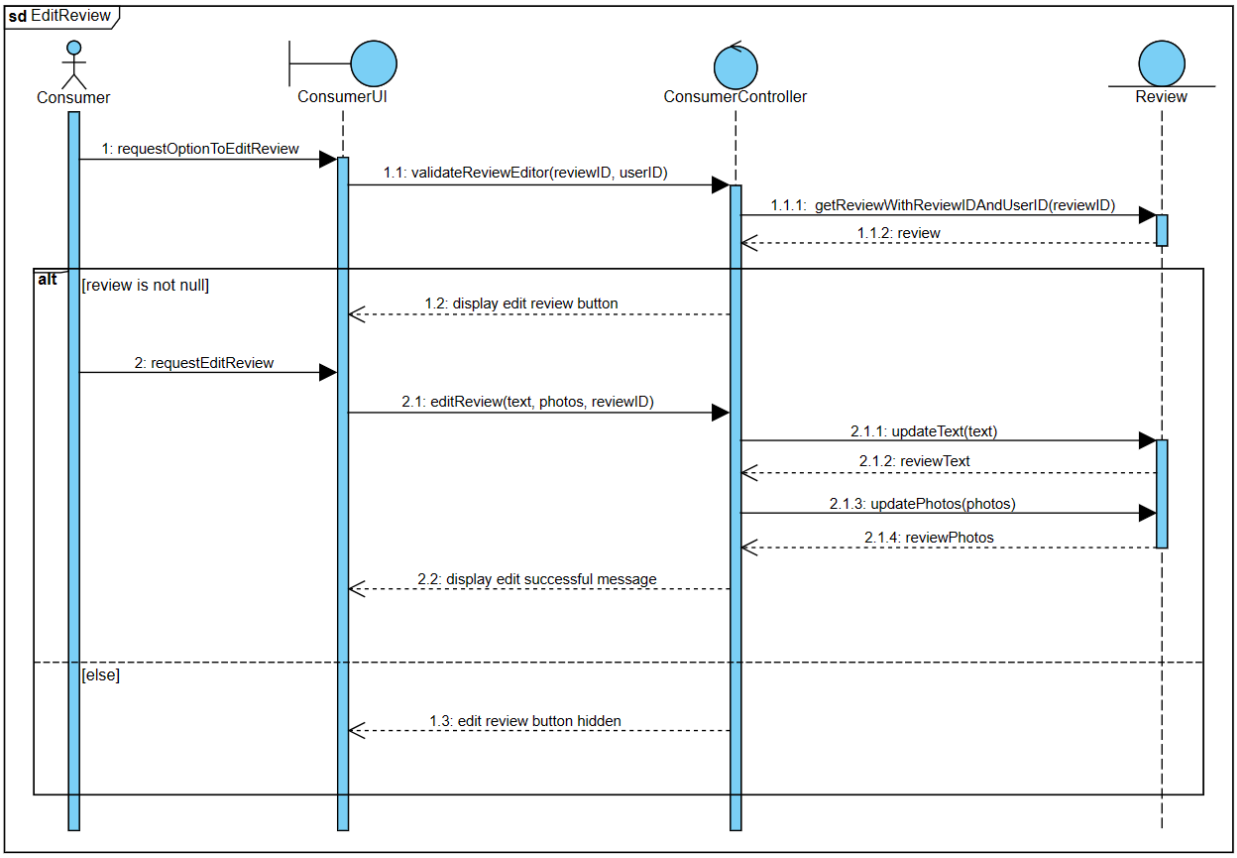


III. For Use Cases under III

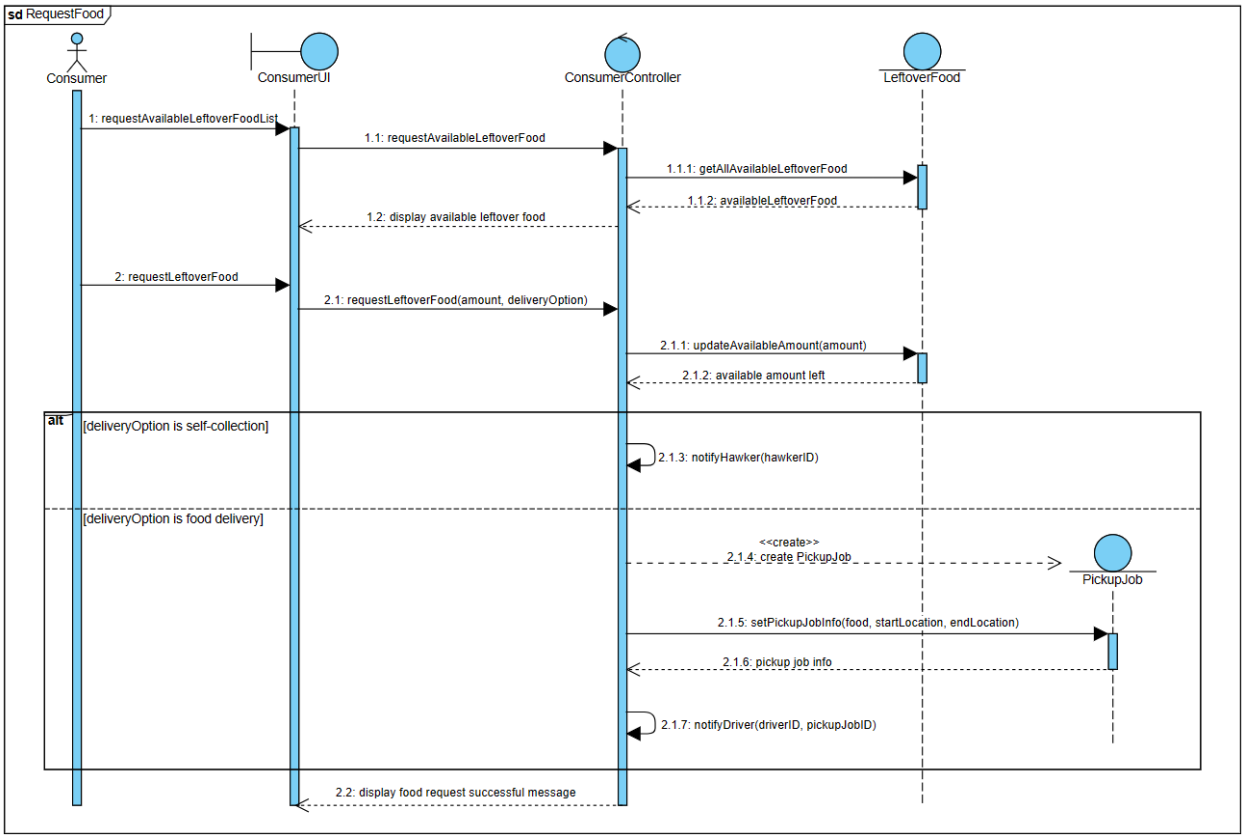
III.I SubmitReview



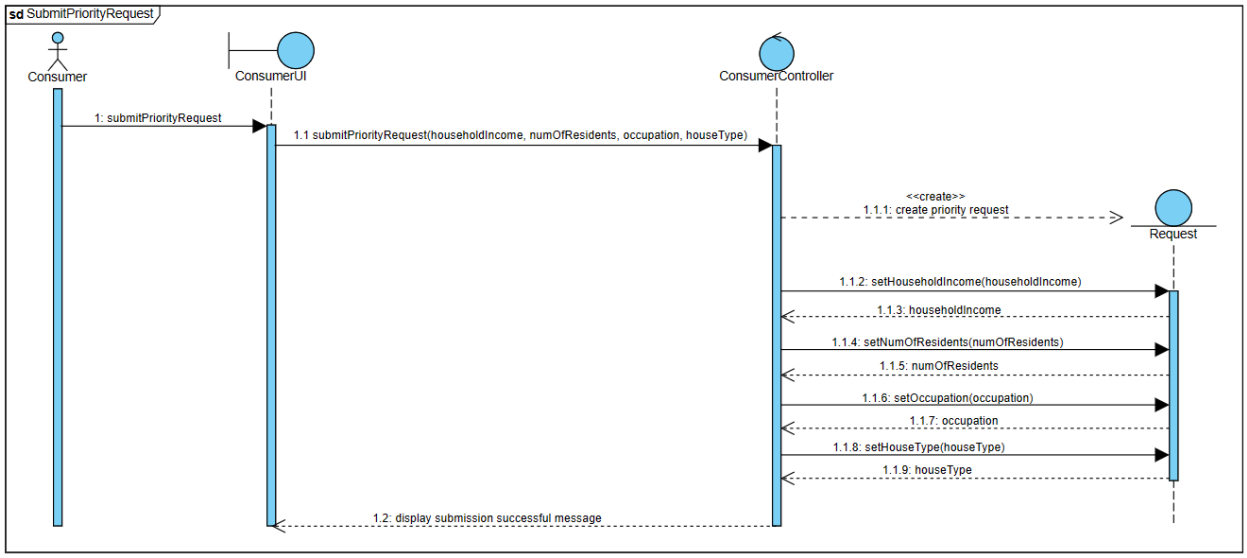
III.II EditReview



III.III RequestFood

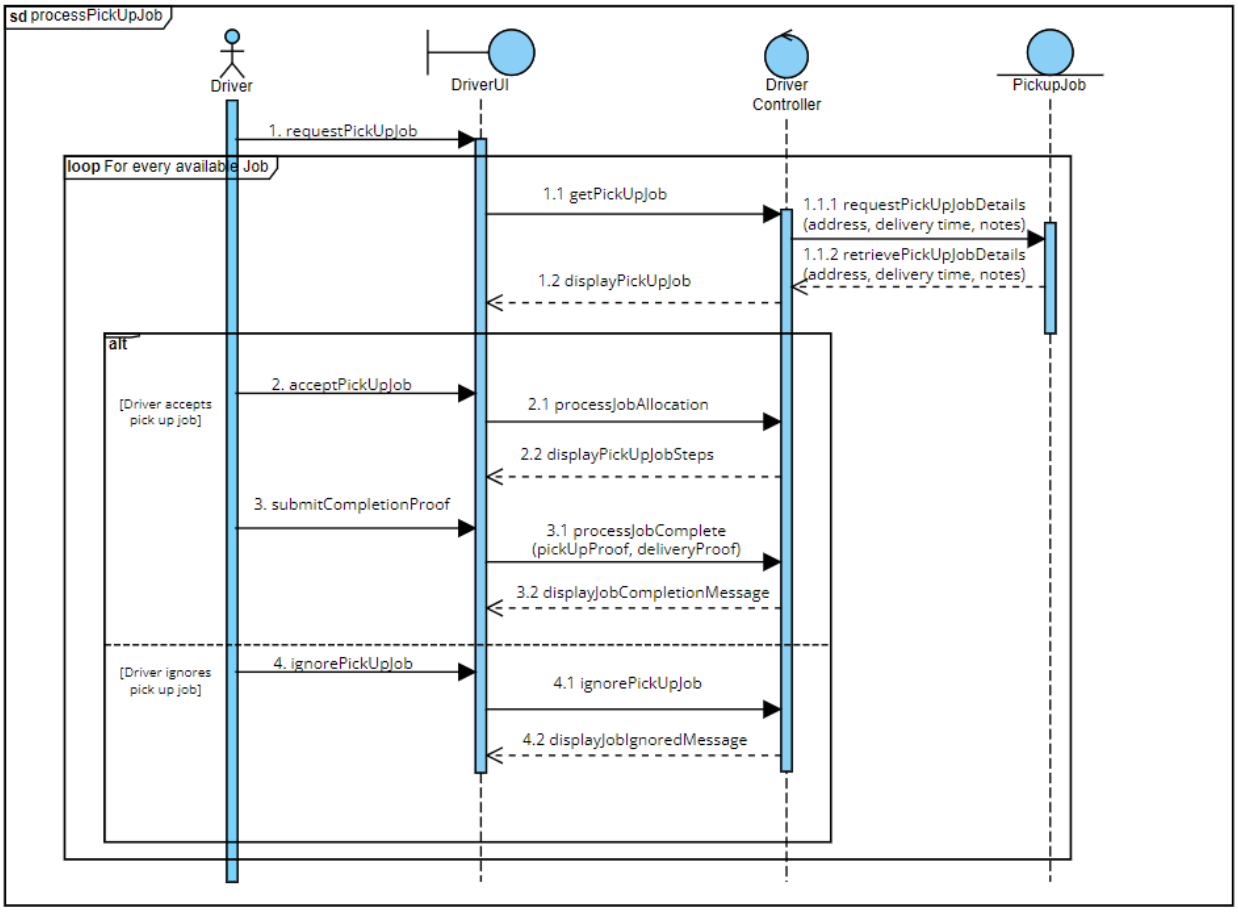


III.IV SubmitPriorityRequest



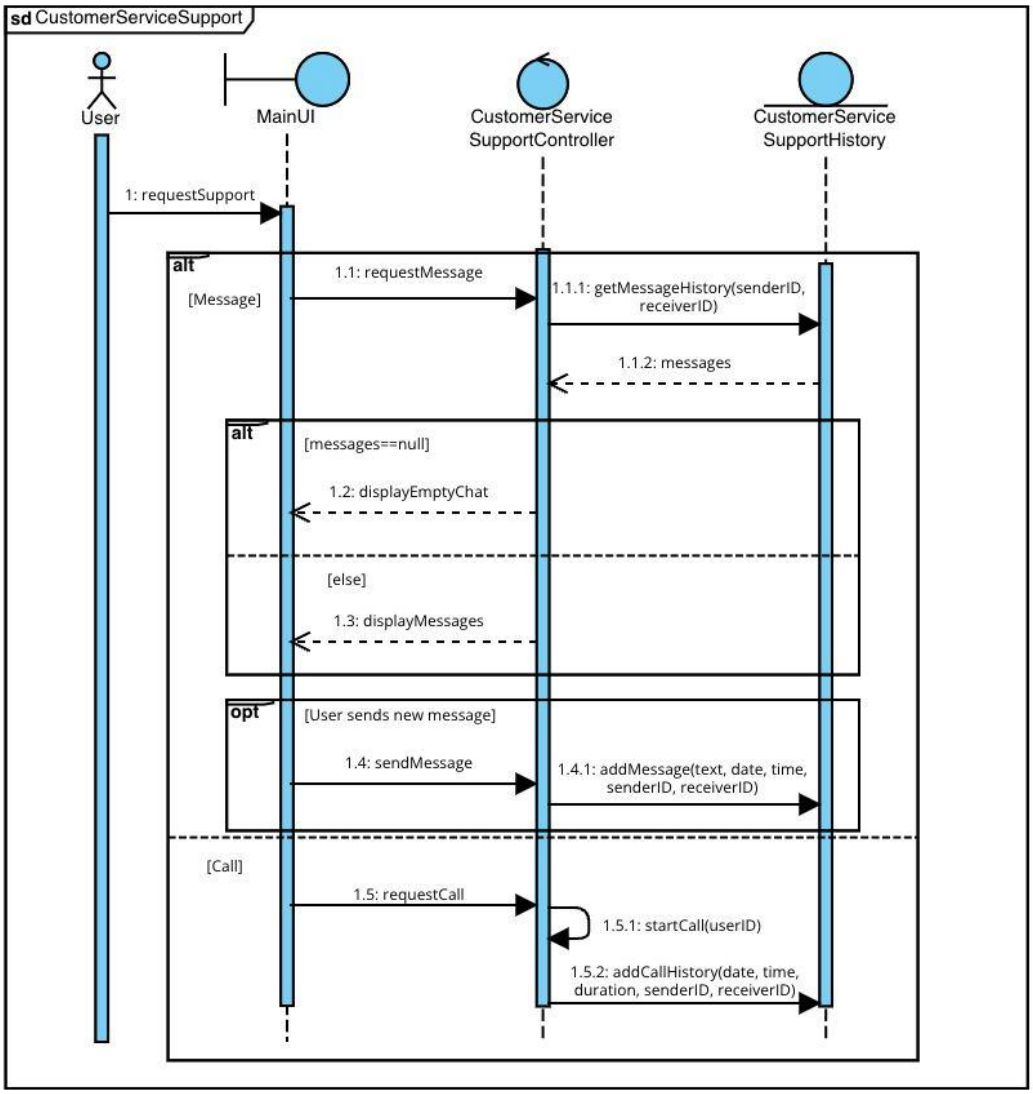
IV. For Use Cases under IV

IV.I ProcessPickupJob



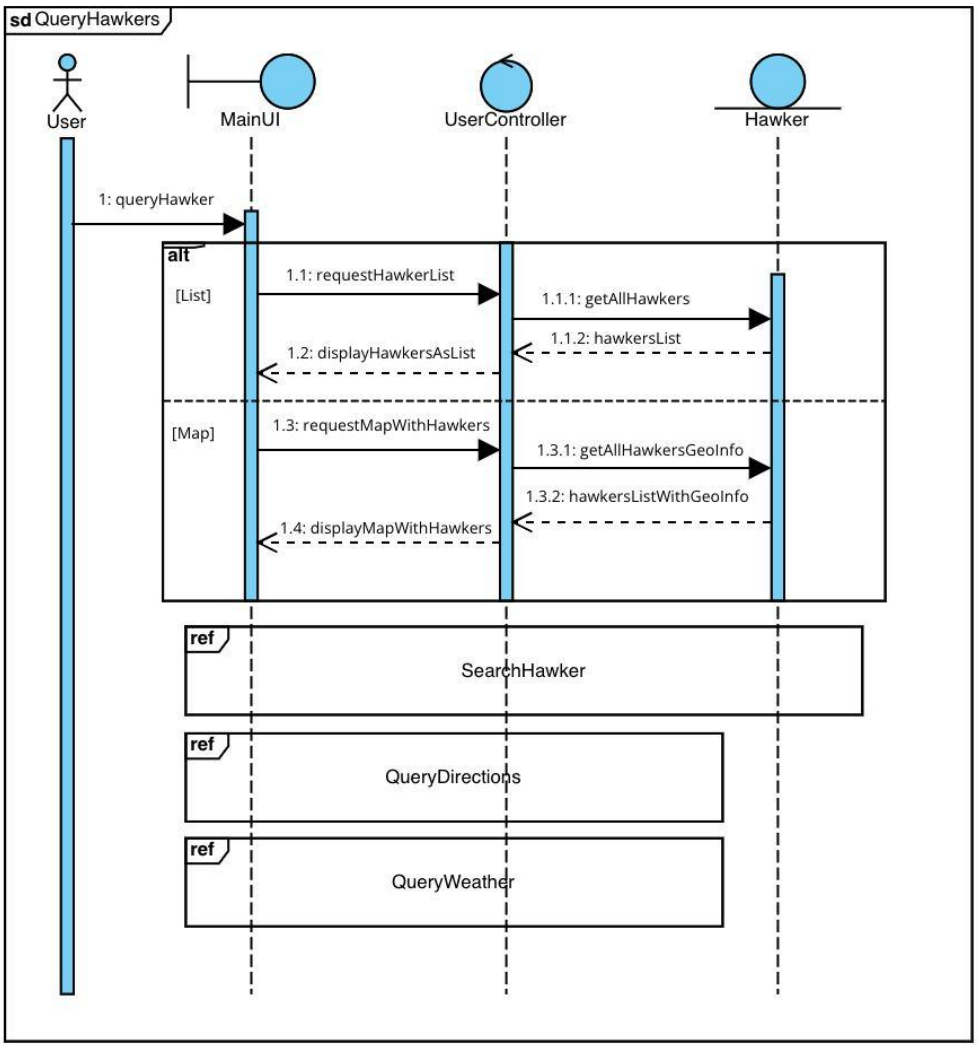
V. For Use Cases under V

V.I CustomerServiceSupport

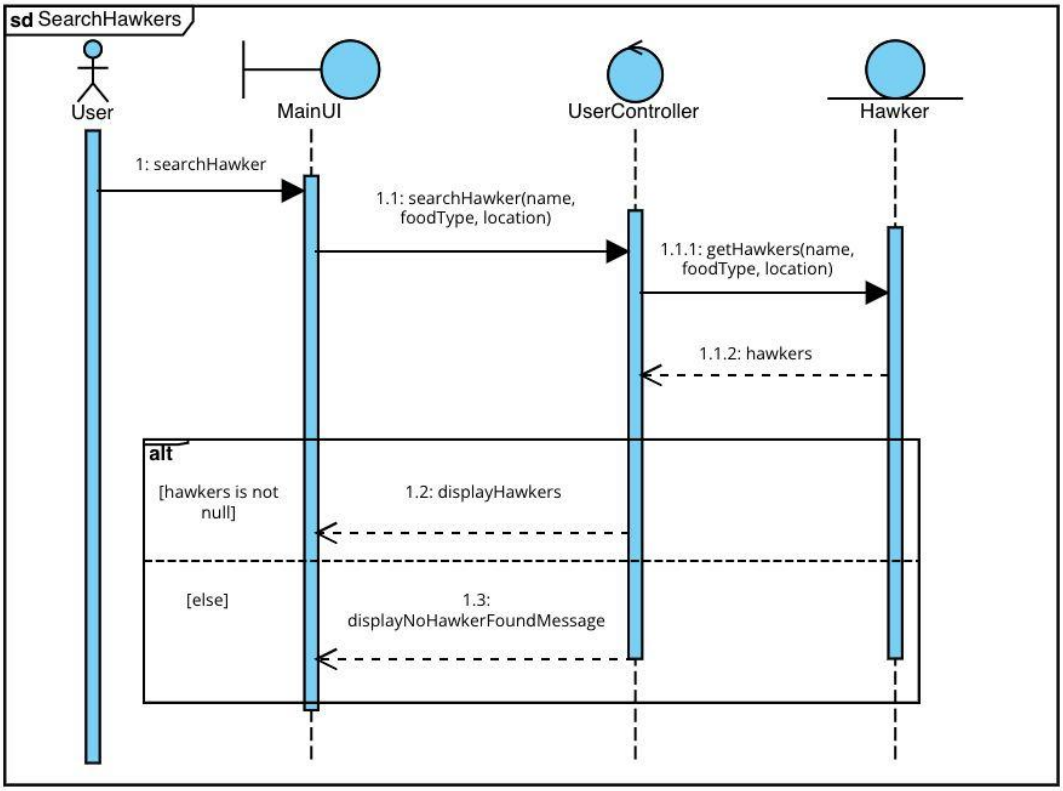


VI. For Use Cases under VI

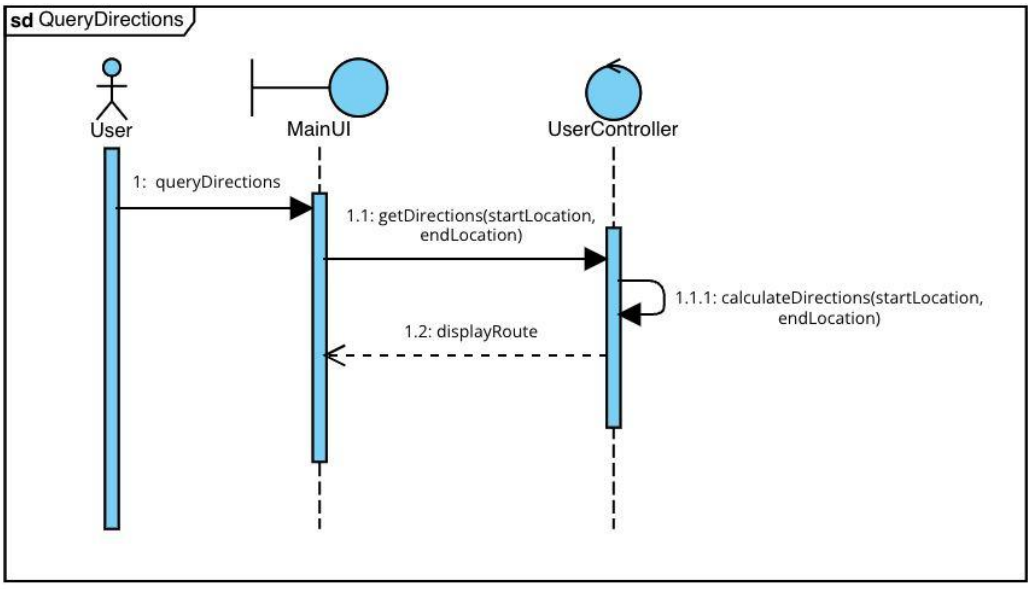
VI.I QueryHawkers



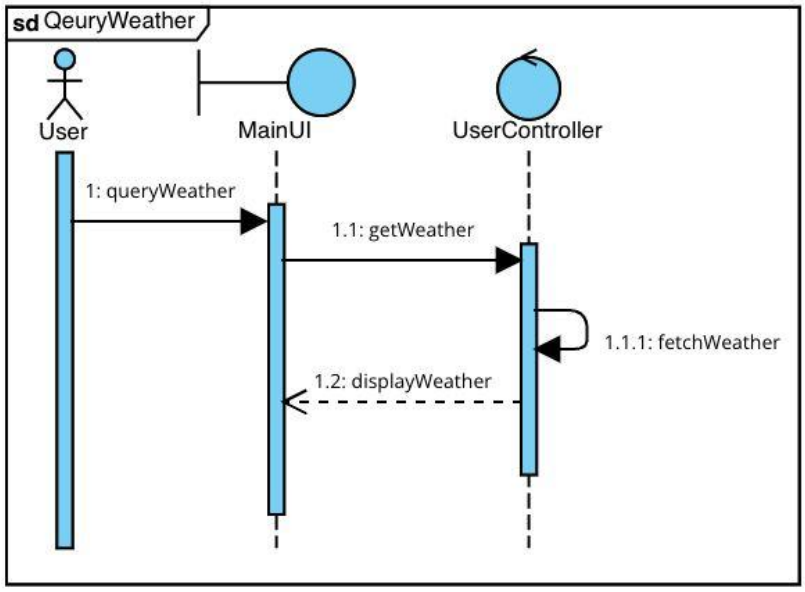
VI.II SearchHawkers



VI.III QueryDirections

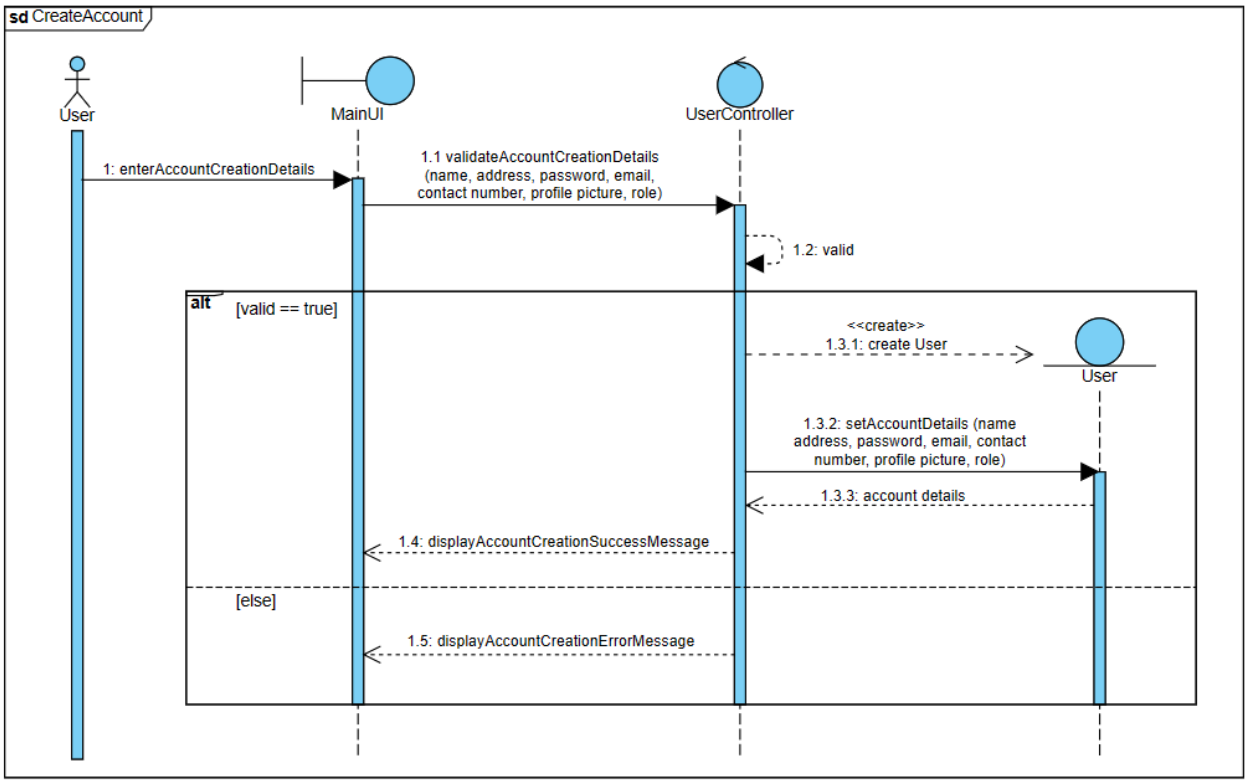


VI.IV QueryWeather

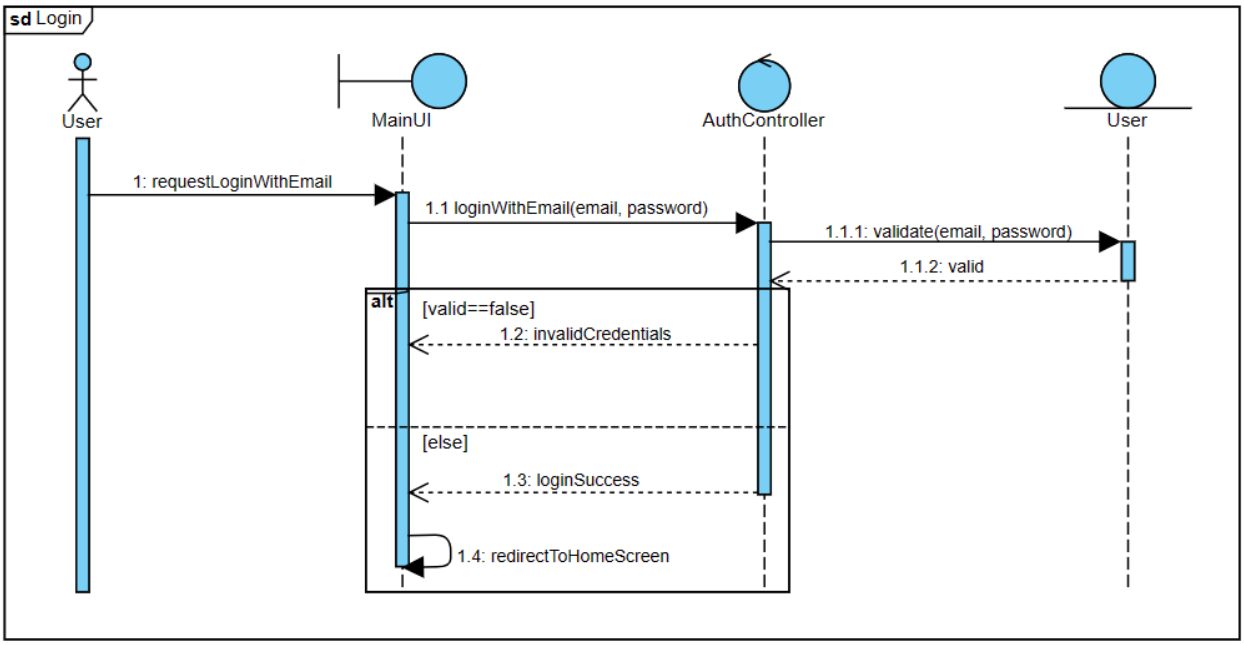


VII. For Use Cases under VII

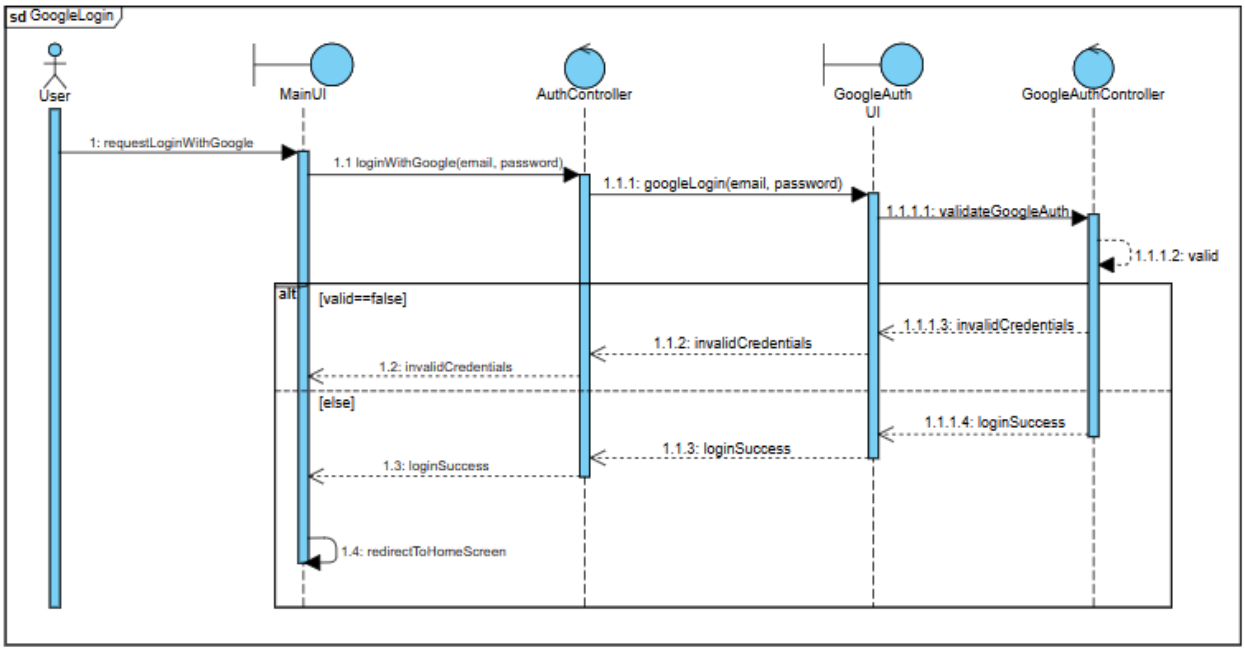
VII.I CreateAccount



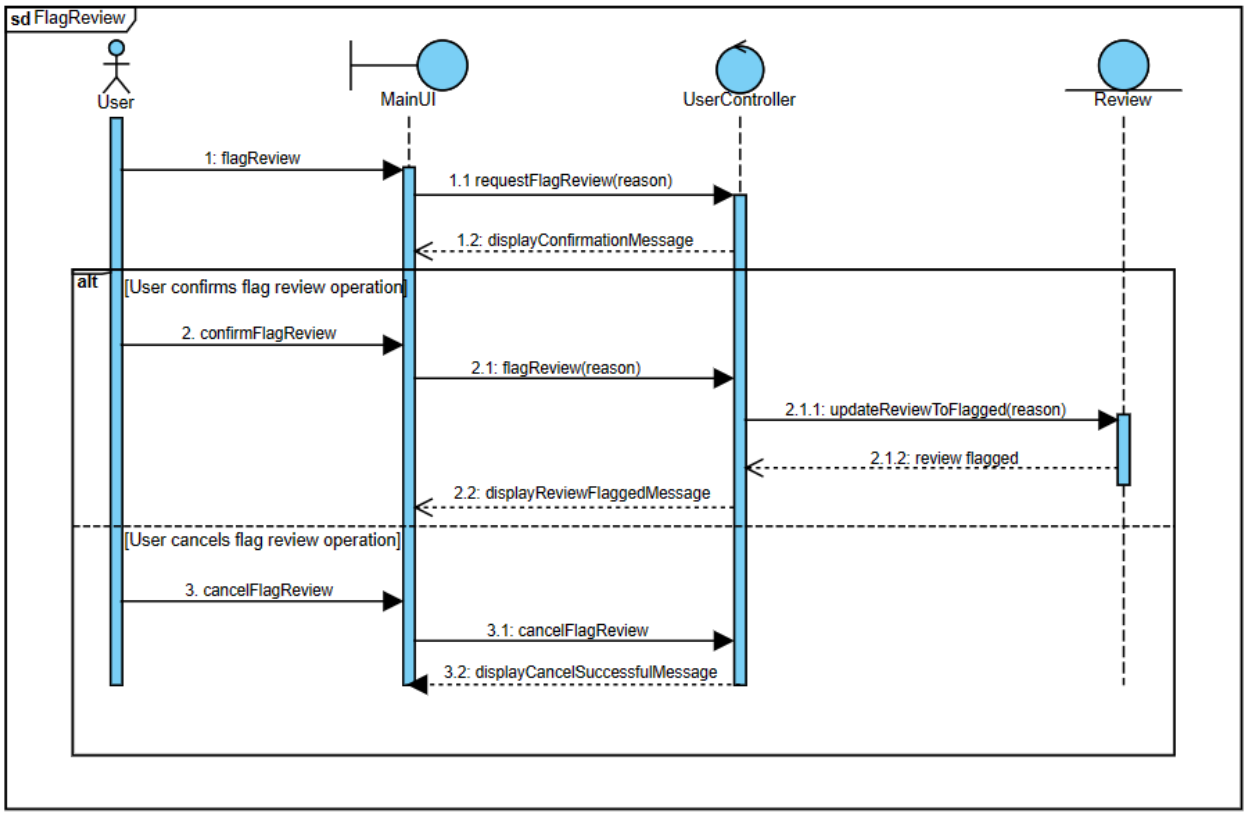
VII.II Login



VII.III GoogleLogin

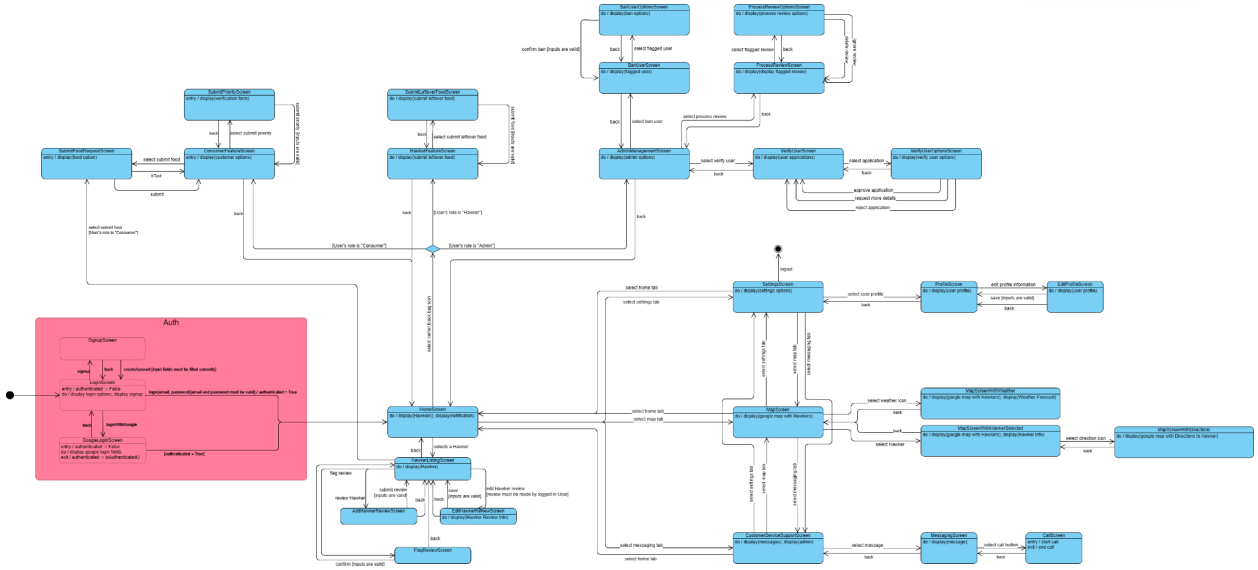


VII.IV FlagReview



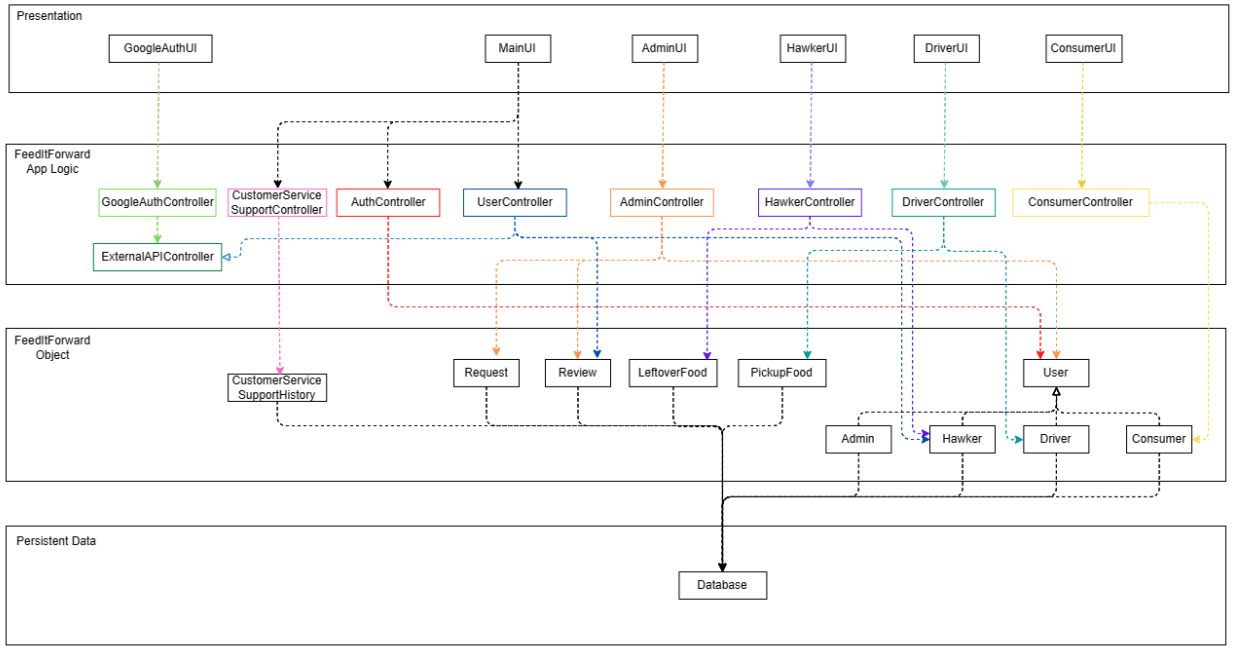
C. Dialog Map Diagram

If the image is unclear, please refer to the raw png file that is uploaded together with this document.



3. System Architecture

If the image is unclear, please refer to the raw png file that is uploaded together with this document.



Presentation Layer

This layer is mainly responsible for the interaction between Users and FeedItForward. The different UIs will then call for the respective controllers to run the App Logic.

This layer consists of:

1. **GoogleAuthUI**
   Allow Users to log in via Google Authentication by using the service of GoogleAuthController.
2. **MainUI**
   MainUI will use the services of UserController, AuthController and CustomerServiceSupportController.
3. **AdminUI**
   AdminUI is part of MainUI, allowing admins to do its job by calling AdminController.
4. **HawkerUI**
   HawkerUI is part of MainUI, allowing hawkers to do its job by calling HawkerController.
5. **DriverUI**
   DriverUI is part of MainUI, allowing drivers to do its job by calling DriverController.
6. **ConsumerUI**
   ConsumerUI is part of MainUI, allowing consumers to do its job by calling ConsumerController.

App Logic Layer

This layer contains all the controller classes that will provide the presentation layer with its services. The controller classes will request for entities from the Object Layer if necessary to run its logic.
This layer consists of:

1. **CustomerServiceController**
   CustomerServiceController is called by MainUI for Users to interact with Administrators. Logic within this controller includes call and message methods that enable the interaction between Users and Administrators.
2. **AuthController**
   AuthController is called by MainUI for Users to log in. Logic within this controller includes the log in and sign up methods.
3. **GoogleAuthController**
   GoogleAuthController is called to allow authentication via google authentication.
4. **ExternalAPIController**
   ExternalAPIController is called by GoogleAuthController and UserController. It contains the logic to handle external API calls for the app logic to run. Logic within this controller includes calling of Google Authentication API and OneMap Route API.
5. **UserController**
   UserController is called by MainUI to handle the search function to return hawker results with directions and weather.
6. **AdminController**
   AdminController contains the logic for administrators. The logic includes verifying users, banning users, notifying users and processing reviews.
7. **HawkerController**
   HawkerController contains the logic for hawkers. The logic includes submitting leftover food.
8. **DriverController**
   DriverController contains the logic for drivers. The logic includes picking up jobs.
9. **ConsumerController**
   ConsumerController contains the logic for consumers. The logic includes requests for food, submitting priority requests, submitting reviews and editing reviews.

App Object Layer

This layer contains the entity classes that will be called by the App Logic layer to implement its logic. The entity classes are stored in the database of the persistent layer.
This layer consists of:

1. **CustomerServiceSupportHistory**
   CustomerServiceSupportHistory contains the messages and call log between the User and administrator.
2. **Request**
   Request contains the request type, date and time, status and data.
3. **Review**
   Review contains text, photos, flagged status, flagged reason and userID.
4. **LeftoverFood**

LeftoverFood contains the amount, picture, location, time passed and availability of food.

5. **PickupJob**
   Pickupjob contains the food, start and end location.
6. **User**
   User contains all information regarding the user. These include name, password, email, contact number, address, profile picture and role.
7. **Admin**
   Admin contains adminID.
8. **Hawker**
   Hawker contains business name, food type, operating hours, reviews and hawkerID.
9. **Driver**
   Driver contains driverID, vehicle number and license number.
10. **Consumer**
    Consumer contains consumerID.

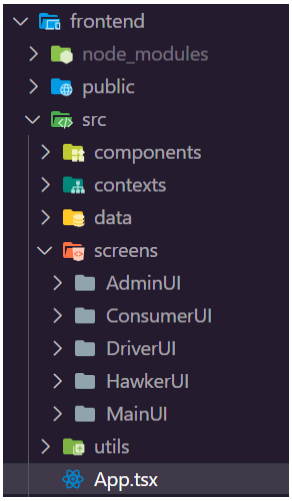
Persistent Data Layer

This layer contains the database that will store all of the entities.

4. Application Skeleton

Please refer to the source code uploaded in the github repository for the application skeleton

A. Frontend

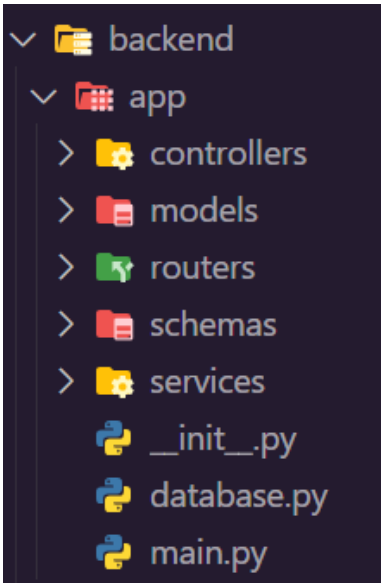


The frontend (React.js) mainly consists of the different **User Interfaces (Screens),** which are structured and categorized into AdminUI, ConsumerUI, DriverUI, HawkerUI, and MainUI as depicted in the class diagrams. More detailed sub-screens can be found in the respective UI screen folders..

The App.tsx is the entry point of the frontend application.

Other folders such as components, contexts, data, utils contains helper files that makes the frontend code more organized and easier to read for ease of collaboration (as recommended by the framework used).

B. Backend



Models: Contains the Business Objects

Services: Contains the methods to access/modify the Business Objects

Controllers: Contains all the controllers that provide an “abstracted interface” of the detailed implementation of the various services.

Routers: Contains REST API endpoints for data communication between frontend and backend

Database.py: Entry point to database that will store all of the entities.

5. Appendix

Key Design Issues

A. Identifying and Storing Persistent Data (Section 7.4.2)

- Relational database
    - User (Hawker, Admin, Driver, Customer)
        - **user_id**, username (str), password (str, encrypted), name (str), address (str), singpass (bool)
        - Hawker
            - **hawker_id** (=user_id), overall_rating (float)
        - Admin
            - **admin_id** (=user_id)
        - Driver
            - **driver_id** (=user_id), vehicle (str)
        - Customer
            - **customer_id** (=user_id), priority (bool)
    - Food
        - **food_id**, description, image
    - Hawker Food (maps hawker to food, 1 to many)
        - ***hawker_food_id***, *hawker_id*, *food_id*
    - Available food
        - **available_food_id**, *hawker_food_id*, quantity
    - Reviews
        - **review_id**, *customer_id*, text (str), rating (int, 1-5), flagged (bool)
    - Food Requests
        - **food_request_id**, *customer_id*, *available_food_id,* *driver_id*, fulfilled (bool)
    - Priority Requests
        - **priority_request_id**, customer_id, status (enum: pending, review, approved), document (str: url to file)
    - Customer service support history
        - *user_id*, *admin_id*, text_history (str)
- Flat file database
    - Documents

B. Providing Access Control (Section 7.4.3)

- Encryption
    - Encrypt password to be stored inside database
- Dynamic access control
    - No need for dynamic access control as permissions are consistent and don't need to be changed frequently.
- Static access control

Actors	**User**	**Food**	**Food Request**
Hawker	createUser() updateUser()	createFood() updateFood() updateAvailableFood()	cancelRequest()
Driver	createUser() updateUser()		denyRequest()
Admin	createUser() updateUser() getAllUser() banUser()	createFood() updateFood() getAvailableFood() updateAvailableFood() getAllFood()	cancelRequest() denyRequest() setFulfilled() setDriver()
Customer	createUser() updateUser()	getAvailableFood()	cancelRequest()

Actors	**Reviews**	**Customer service**	**Priority Request**
Hawker	getReviews() flagReview()	getHistory() clearHistory()	
Driver	getReviews() flagReview()	getHistory() clearHistory()	
Admin	getReviews() getFlaggedReviews() deleteReview() clearFlaggedReview()	getHistory() clearHistory()	getAllDocuments() getDocument() deleteDocument() updateDocument() approve() reject() request_info()
Customer	getReviews() createReview() flagReview()	getHistory() clearHistory()	getOwnDocument() deleteOwnDocument() updateOwnDocument()

Design Patterns Used

Please refer to the class diagram section for the detailed description of the design patterns used.

3. **Observer Pattern (Driver - PickupJob)**
    - For Driver - Pickup Job
        - Subject: Pickup Job
        - Observers: Drivers
4. **Strategy Pattern + Factory Pattern (DataStore)**
5. **Facade Pattern (Controllers)**
6. **Publisher-Subscriber Pattern**
    - For websocket connection between different Users for text communication in CustomerServiceSupport

Tech Stack

- **Frontend**
    - React.js
    - TypeScript
    - TailwindCSS
- **Backend**
    - Fastapi (Python)
- **Database**
    - MySQL