NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2006: Software Engineering

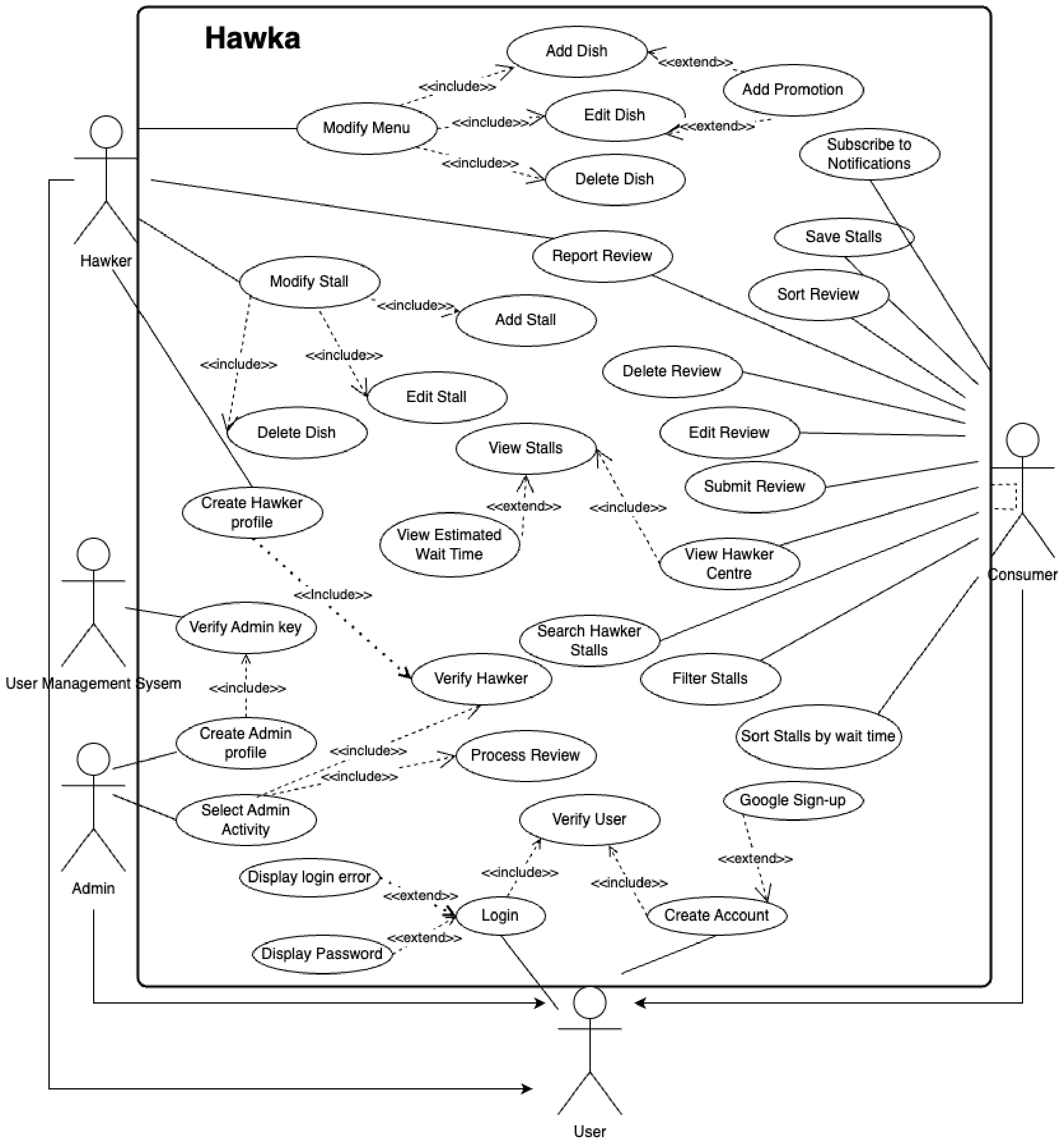
Lab 3 Deliverables

Lab Group	ACDA2
Team	Hawka
Members	Thant Htoo Aung (U2220809L)
	Cao Jun Ming (U2310254A)
	Kow Zi Ting (U2310485B)
	Muhammad Aliff Amirul Bin Mohammed Ariff (U2322581A)
	Saeng-nil Natthakan (U2220832B)

Complete Use Case Model

Use Case Diagram

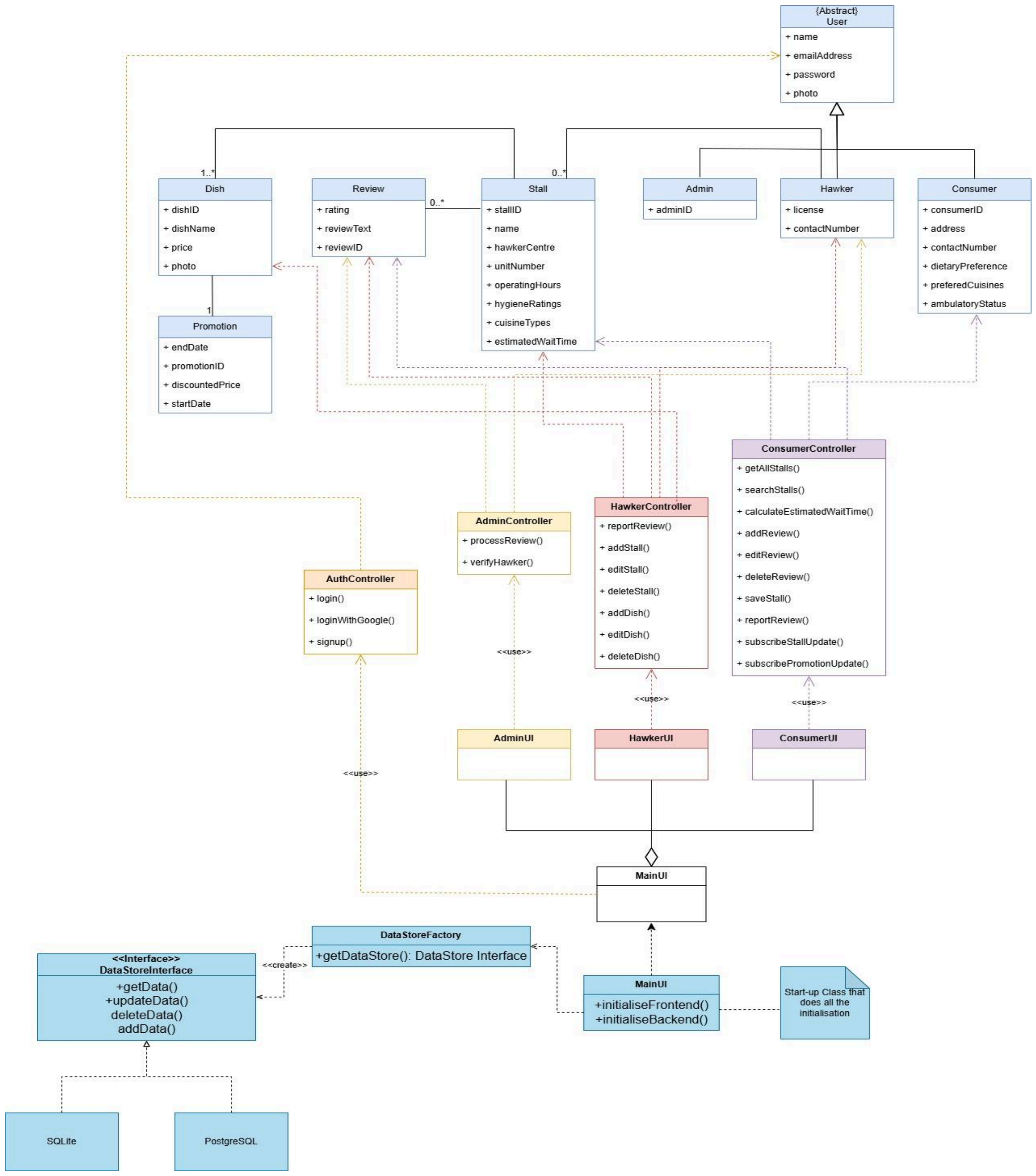
If the image is unclear, please refer to the accompanying raw PNG file uploaded with this document.



Design Model

Class Diagram

If the image is unclear, please refer to the accompanying raw PNG file uploaded with this document.



App

This is a launcher class whose main purpose is initialising the frontend and backend of our application.

Operations:

- **initialiseFrontend():** activates the frontend user interfaces.
- **initialiseBackend():** executes the backend application and getDatabase() for data storage, modification and retrieval.

User Interfaces (UI)

User interfaces refer to the means through which users will interact with our application.

- **MainUI:** This is the main screen for Users. Users refer to those who have not created profiles in the respective categories: Consumer, Hawker, Admin. Users are only able to perform tasks relating to Authentication.
- **AdminUI:** These are screens accessible only by Admins. Admins are able to perform tasks like verifying hawkers and processing reviews.
- **HawkerUI:** These are screens accessible only by Hawkers. Hawkers are able to perform tasks like adding, editing and deleting dishes and stalls and reporting reviews.
- **ConsumerUI:** These are screens accessible only by Consumers. Consumers are able to perform tasks like filtering hawker centres and submitting reviews.

Controllers (Facade Pattern + Mediator Pattern)

In the design of our application, we implemented the **Facade Pattern** and **Mediator Pattern**.

Facade Pattern

In implementing the **Facade Pattern**, we use our controller classes as a layer between our users and our entity classes, allowing users to view only a simplified interface of the application.

Mediator Pattern

In implementing the **Mediator Pattern**, we use our controller classes as a mediator between entity classes. This decreases coupling within our application as entity classes are only dependent on their respective mediator classes and not each other.

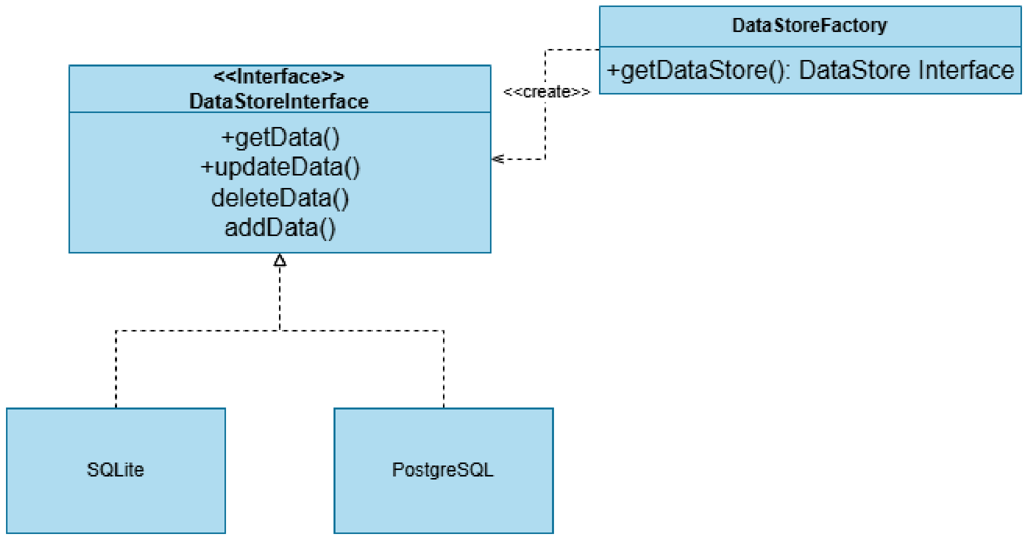
We have multiple controllers that serve different use cases and User roles:

- **AuthController:** As the controller for Authentication, it allows Users to sign up and login through Google Authentication or otherwise. It has access to the following entities: Hawker, Consumer, Admin.
- **AdminController:** As the controller for Admins, it allows Admins to verify Hawkers and process reviews. It has access to the following entities: Admin and Review.
- **ConsumerController:** As the controller for Consumers, it allows Consumers to subscribe to notifications, save stalls, sort, report, edit, submit and delete reviews, view hawker centres, search, filter and sort hawker stalls. It has access to the following entities: Consumer, Review.

- **HawkerController:** As the controller for Hawker, it allows Hawkers to modify menu, modify stall and report review. It has access to the following entities: Dish, Stall, Promotion.

Databases (Factory Pattern)

The use of databases ensures that our application can be stored, managed and retrieved effectively. They also provide flexibility for scalability along the later stages of the Software Development Life Cycle (SDLC) of our project.



Factory Pattern

Our database implements the **Factory Pattern.** This is reflected in DatabaseInterface, which defines an interface for database creation, concrete classes SQLiteDatabase and PostgresSQLDatabase that offer different database connections, and DatabaseFactory which allows the client to request an instance of either database type without exposing their instantiation logic.

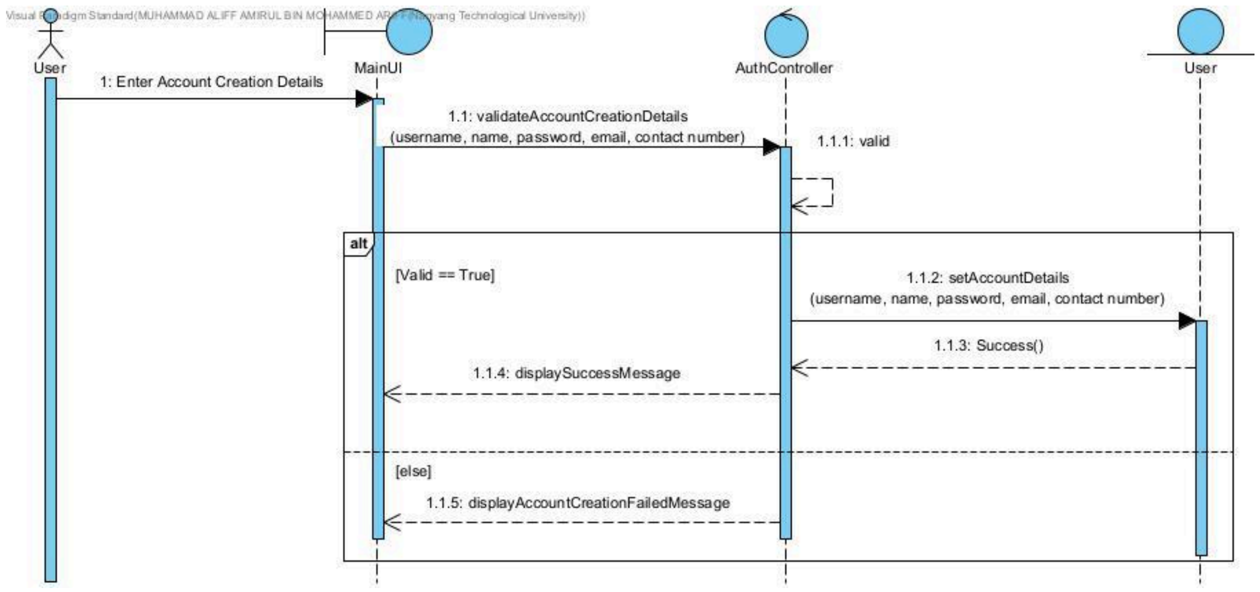
Observer Pattern

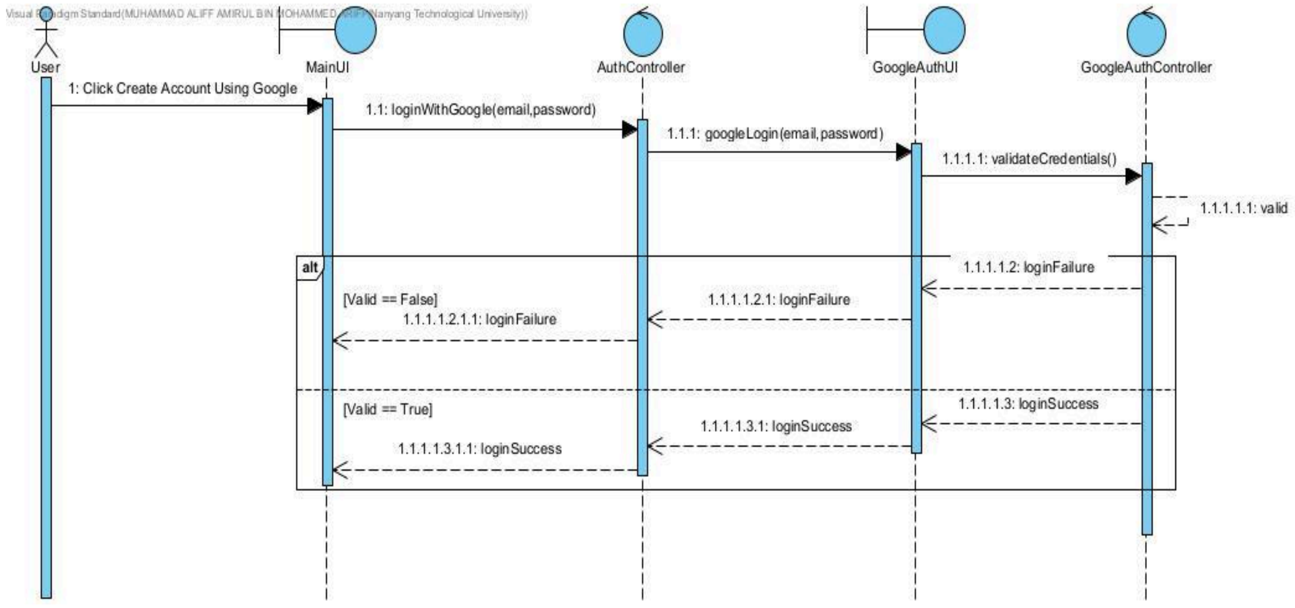
Our database seeding logic implements the **Observer Pattern**. This is implemented using SQLAlchemy's event system, where table definitions such as User._table_ serve as subjects and the seed_table() function acts as the observer. When an after_create event is generated upon table creation, the observer is automatically triggered to populate the table with predefined data. This setup enables the automatic and decoupled initialisation of database contents and reflects the core principles of the Observer Pattern by promoting loose coupling between the event source and its handler.

Sequence Diagrams

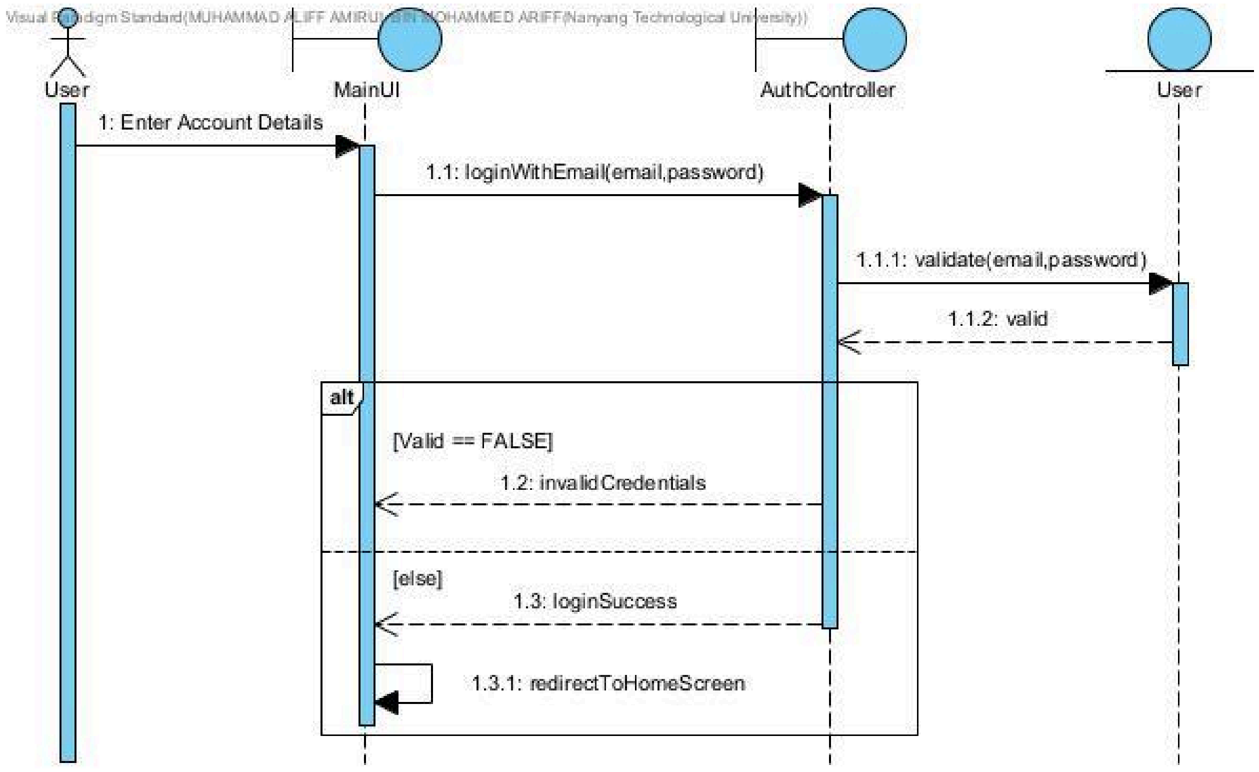
1. Use Cases for User

1.1 User Account Creation

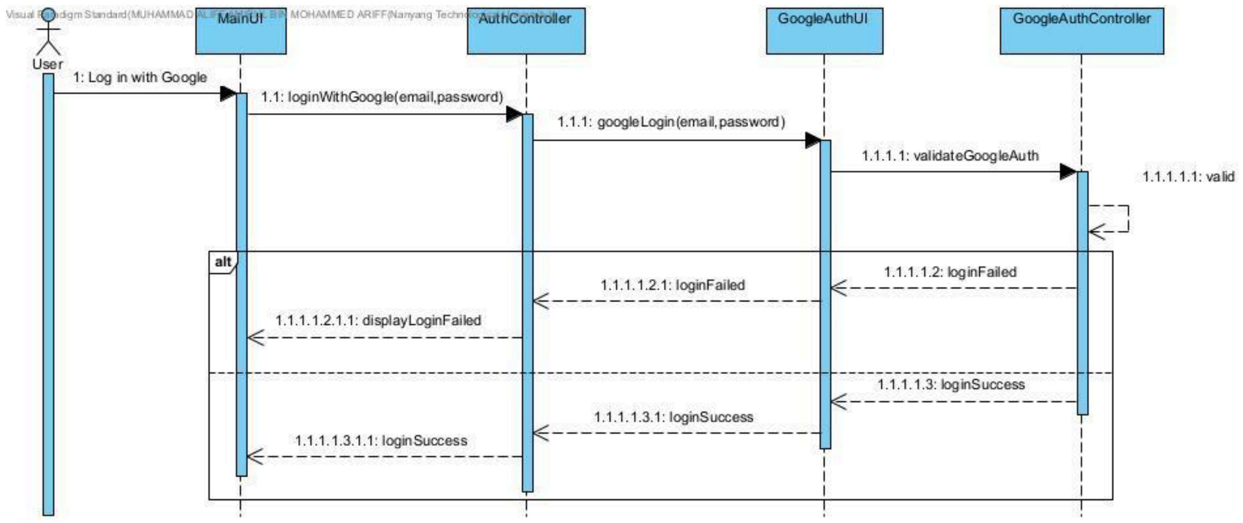

Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
User
MainUI
AuthController
User
1: Enter Account Creation Details
1.1: validateAccountCreationDetails (username, name, password, email, contact number)
1.1.1: valid
alt
[Valid == True]
1.1.2: setAccountDetails (username, name, password, email, contact number)
1.1.3: Success()
1.1.4: displaySuccessMessage
[else]
1.1.5: displayAccountCreationFailedMessage


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
User
MainUI
AuthController
GoogleAuthUI
GoogleAuthController
1: Click Create Account Using Google
1.1: loginWithGoogle(email,password)
1.1.1: googleLogin(email,password)
1.1.1.1: validateCredentials()
1.1.1.1.1: valid
alt
1.1.1.1.2: loginFailure
[Valid == False]
1.1.1.1.2.1: loginFailure
1.1.1.1.2.1.1: loginFailure
[Valid == True]
1.1.1.1.3: loginSuccess
1.1.1.1.3.1: loginSuccess
1.1.1.1.3.1.1: loginSuccess

1.2 User Login

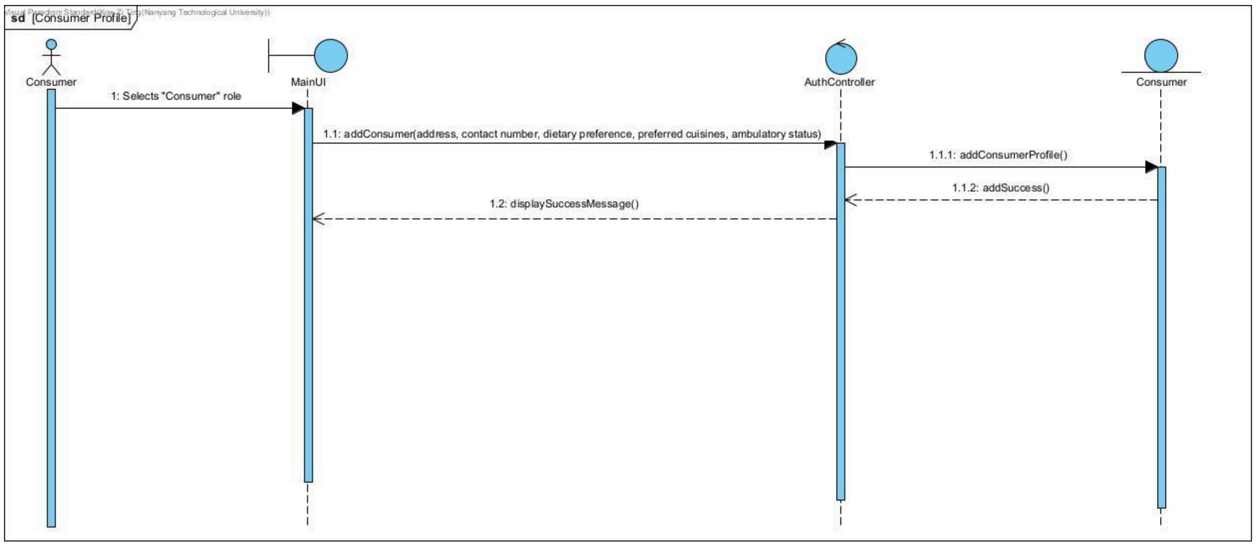

Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
User
MainUI
AuthController
User
1: Enter Account Details
1.1: loginWithEmail(email,password)
1.1.1: validate(email,password)
1.1.2: valid
alt
[Valid == FALSE]
1.2: invalidCredentials
[else]
1.3: loginSuccess
1.3.1: redirectToHomeScreen

1.3 Google Login


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
User
MainUI
AuthController
GoogleAuthUI
GoogleAuthController
1: Log in with Google
1.1: loginWithGoogle(email,password)
1.1.1: googleLogin(email,password)
1.1.1.1: validateGoogleAuth
1.1.1.1.1: valid
alt
1.1.1.1.2: loginFailed
1.1.1.1.2.1: loginFailed
1.1.1.1.2.1.1: displayLoginFailed
1.1.1.1.3: loginSuccess
1.1.1.1.3.1: loginSuccess
1.1.1.1.3.1.1: loginSuccess

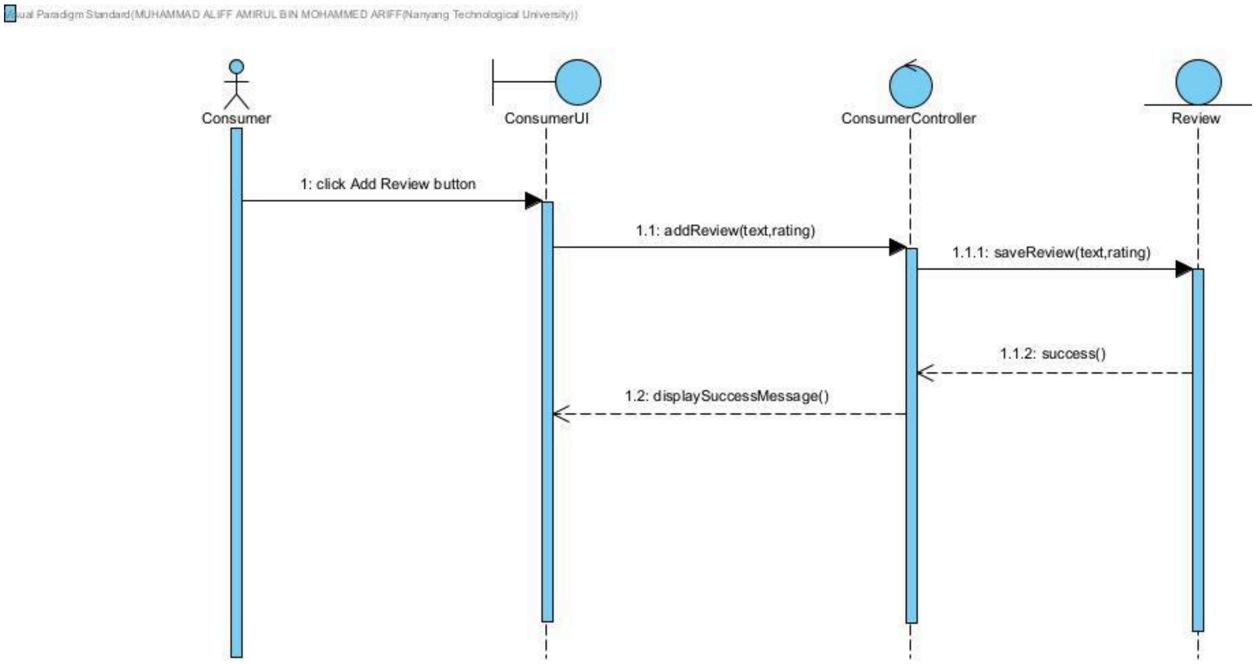
2. Use Cases for Consumer

2.1 Consumer Profile Creation



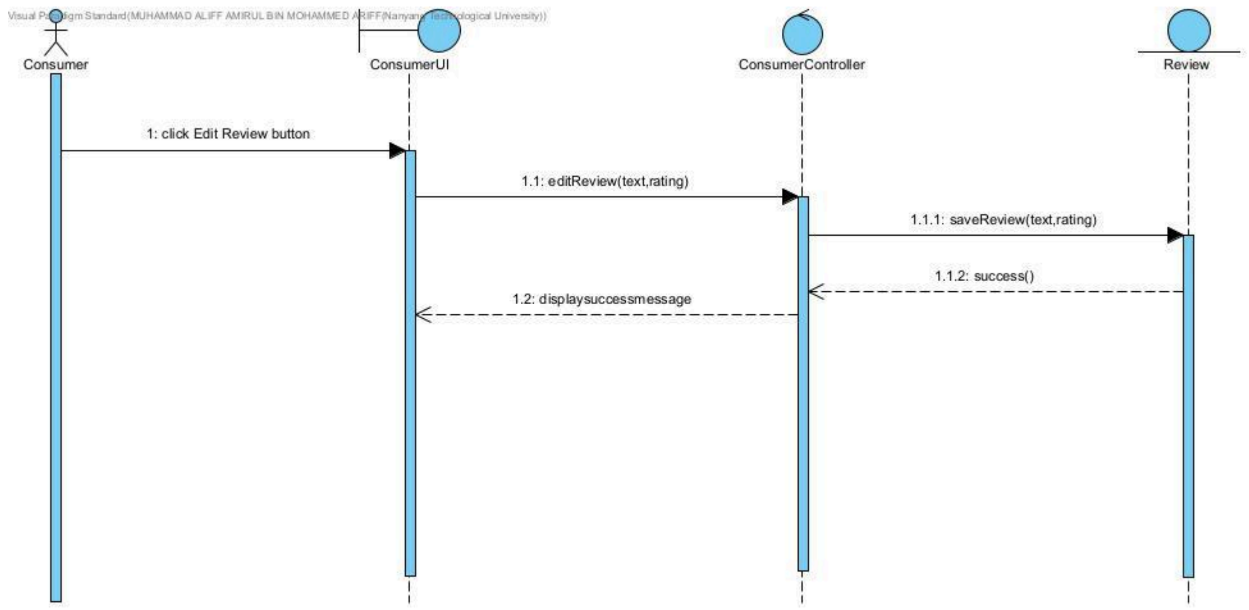
Visual Paradigm Standard(Kow Zi Ting(Nanyang Technological University))
sd [Consumer Profile]
Consumer
MainUI
AuthController
Consumer
1: Selects "Consumer" role
1.1: addConsumer(address, contact number, dietary preference, preferred cuisines, ambulatory status)
1.1.1: addConsumerProfile()
1.1.2: addSuccess()
1.2: displaySuccessMessage()

2.2 Add Review

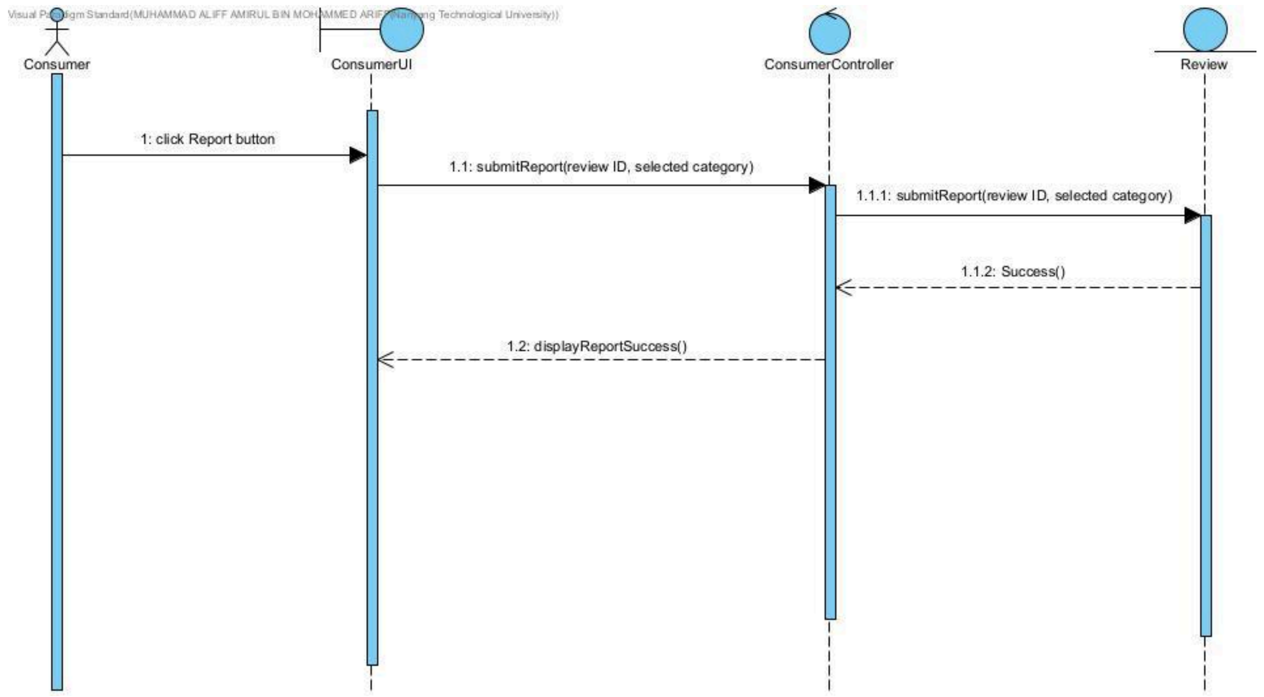


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
Consumer
ConsumerUI
ConsumerController
Review
1: click Add Review button
1.1: addReview(text,rating)
1.1.1: saveReview(text,rating)
1.1.2: success()
1.2: displaySuccessMessage()

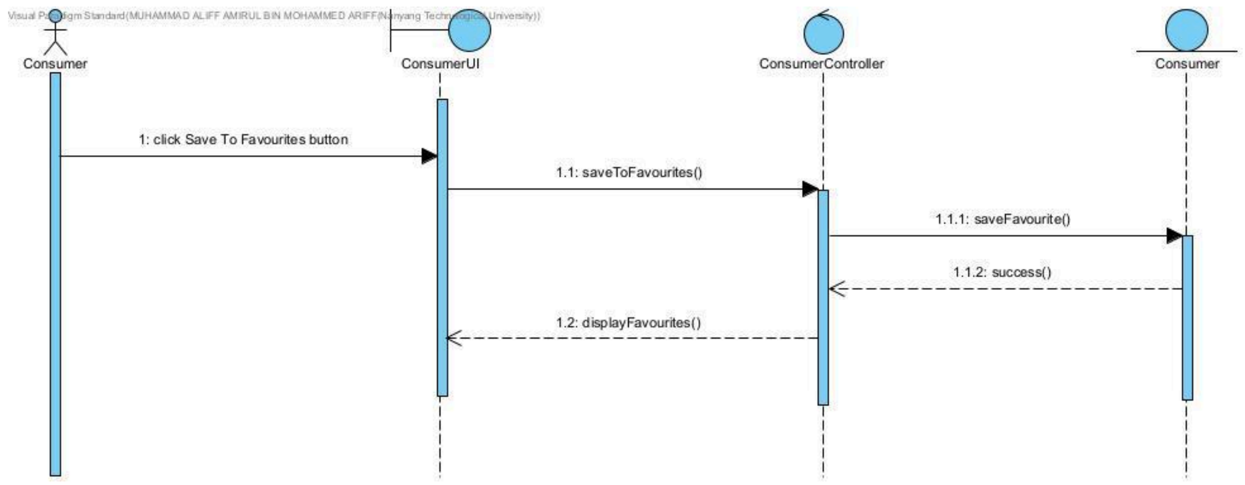
2.3 Edit Review


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
Consumer
ConsumerUI
ConsumerController
Review
1: click Edit Review button
1.1: editReview(text,rating)
1.1.1: saveReview(text,rating)
1.1.2: success()
1.2: displaysuccessmessage

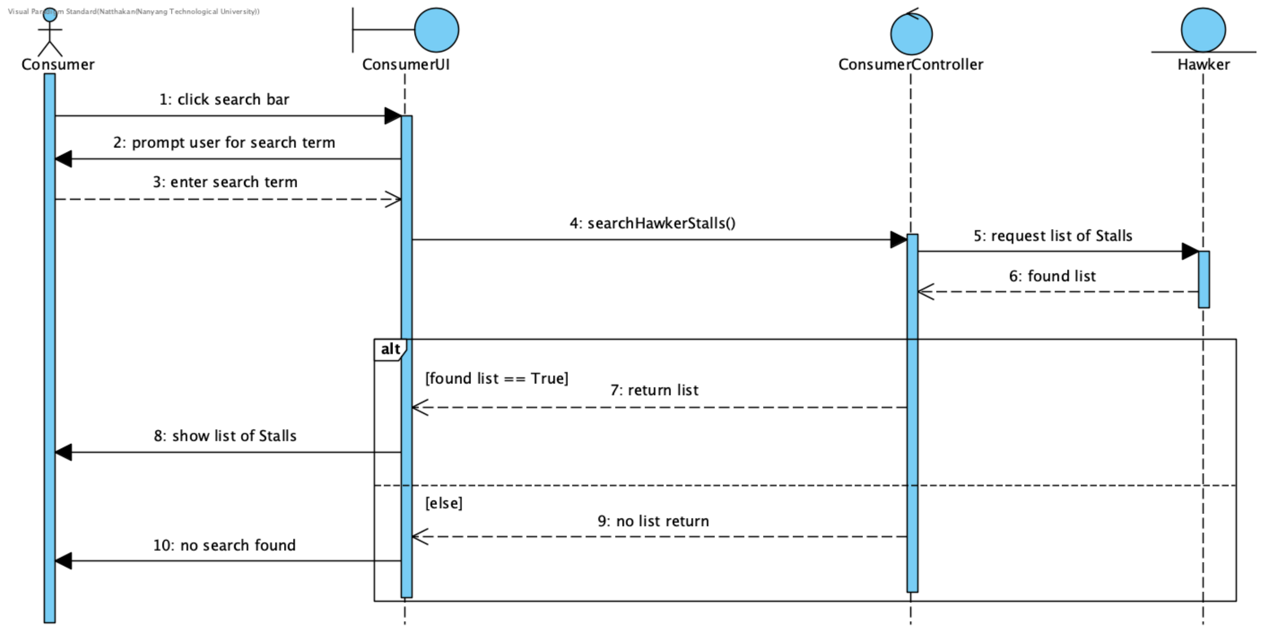
2.4 Report Review


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
Consumer
ConsumerUI
ConsumerController
Review
1: click Report button
1.1: submitReport(review ID, selected category)
1.1.1: submitReport(review ID, selected category)
1.1.2: Success()
1.2: displayReportSuccess()

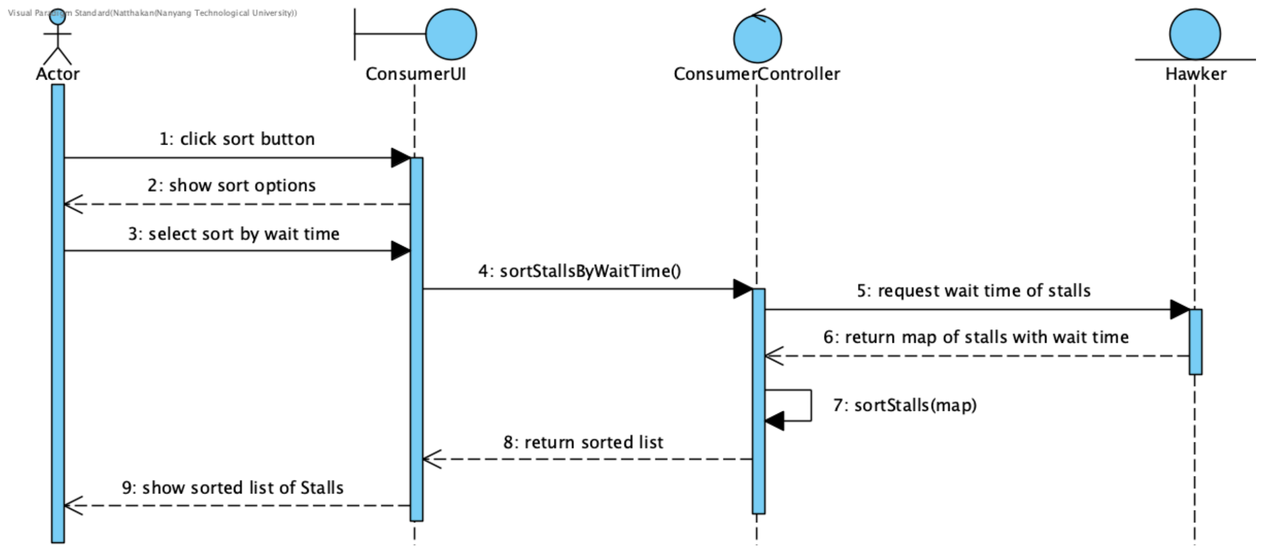
2.5 Save Favourite Stalls


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
Consumer
ConsumerUI
ConsumerController
Consumer
1: click Save To Favourites button
1.1: saveToFavourites()
1.1.1: saveFavourite()
1.1.2: success()
1.2: displayFavourites()

2.6 Search Stalls

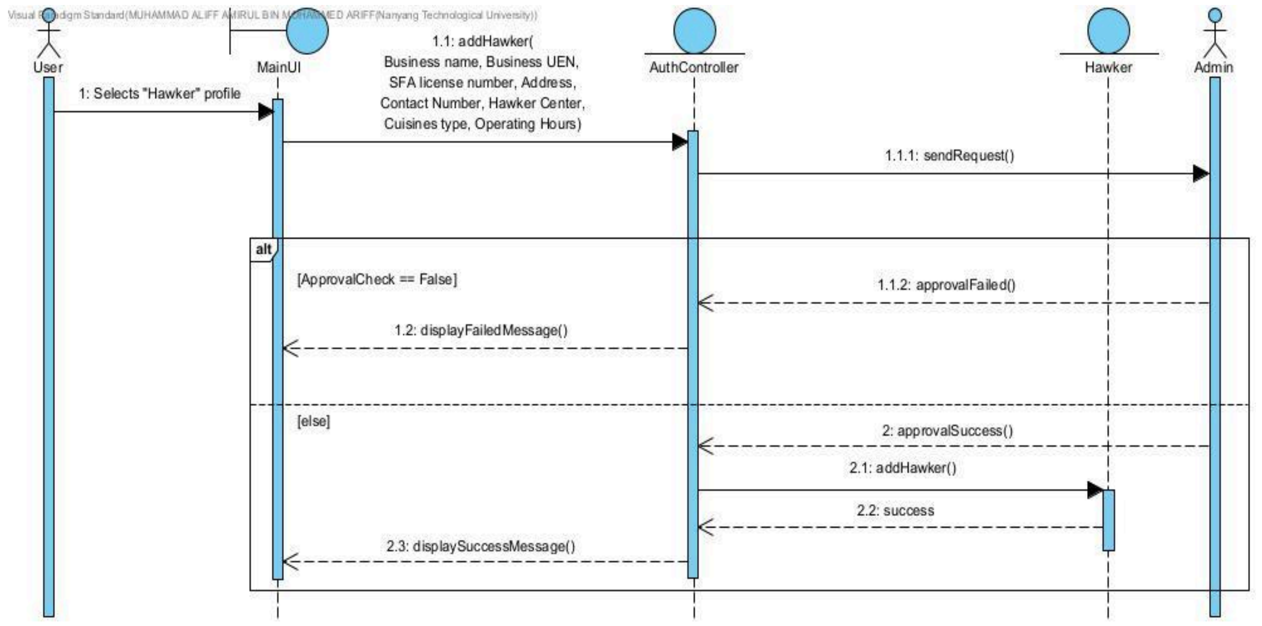


2.7 Sort by Wait Time

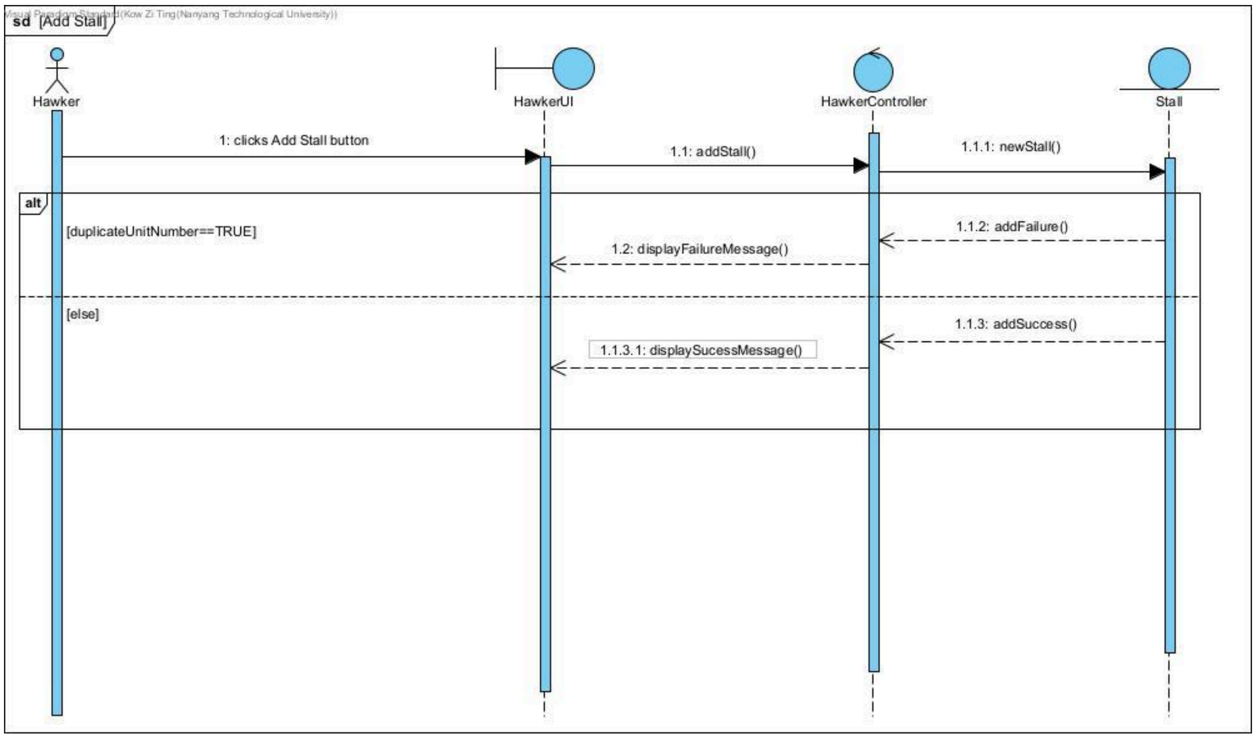

Visual Paradigm Standard(Natthakan(Nanyang Technological University))
Actor
ConsumerUI
ConsumerController
Hawker
1: click sort button
2: show sort options
3: select sort by wait time
4: sortStallsByWaitTime()
5: request wait time of stalls
6: return map of stalls with wait time
7: sortStalls(map)
8: return sorted list
9: show sorted list of Stalls

3. Use Cases for Hawker

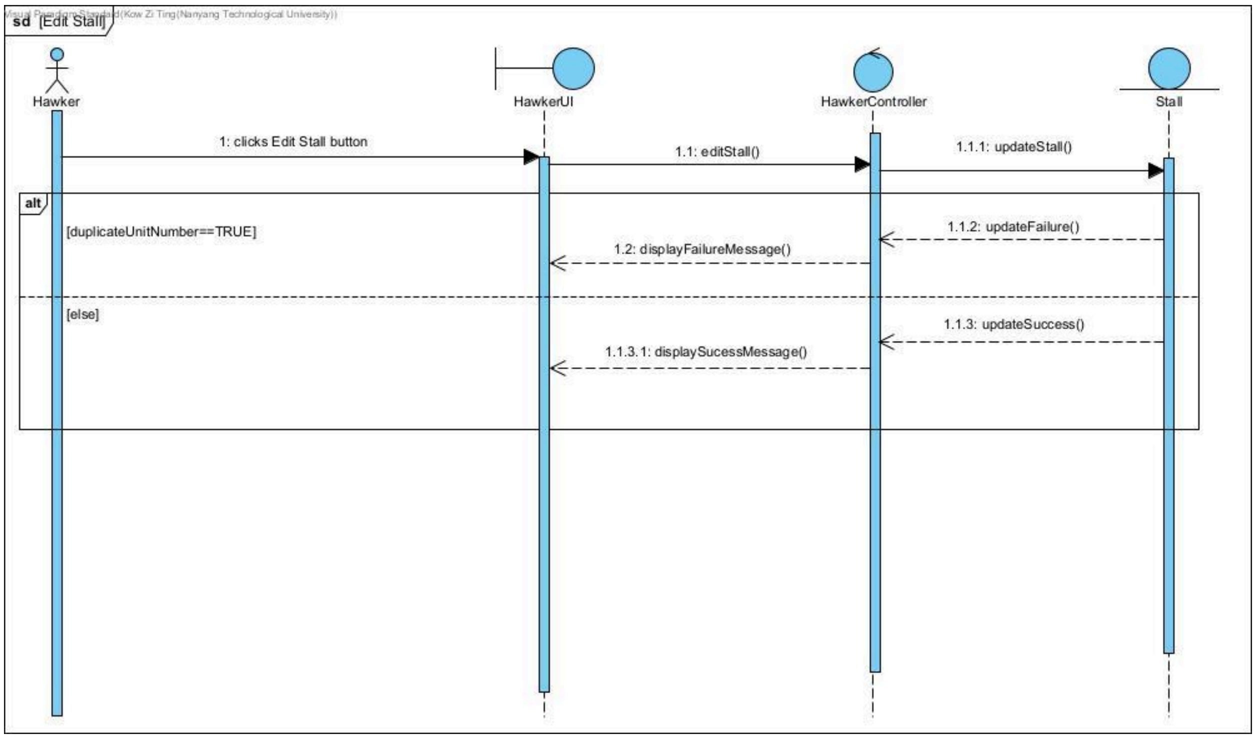
3.1 Hawker Profile Creation


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
User
MainUI
AuthController
Hawker
Admin
1: Selects "Hawker" profile
1.1: addHawker(Business name, Business UEN, SFA license number, Address, Contact Number, Hawker Center, Cuisines type, Operating Hours)
1.1.1: sendRequest()
alt
[ApprovalCheck == False]
1.1.2: approvalFailed()
1.2: displayFailedMessage()
[else]
2: approvalSuccess()
2.1: addHawker()
2.2: success
2.3: displaySuccessMessage()

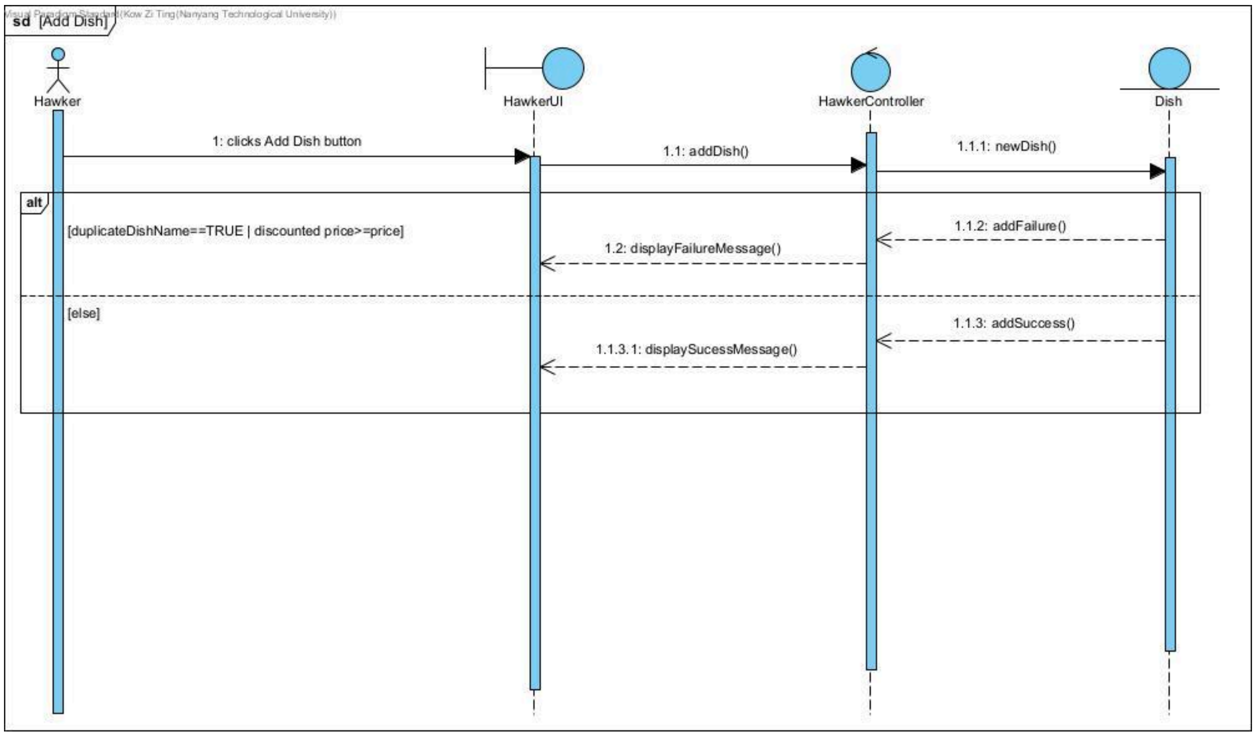
3.2 Add Stall


Visual Paradigm Standard(Kow Zi Ting(Nanyang Technological University))
sd [Add Stall]
Hawker
HawkerUI
HawkerController
Stall
1: clicks Add Stall button
1.1: addStall()
1.1.1: newStall()
alt
[duplicateUnitNumber==TRUE]
1.1.2: addFailure()
1.2: displayFailureMessage()
[else]
1.1.3: addSuccess()
1.1.3.1: displaySucessMessage()

3.3 Edit Stall

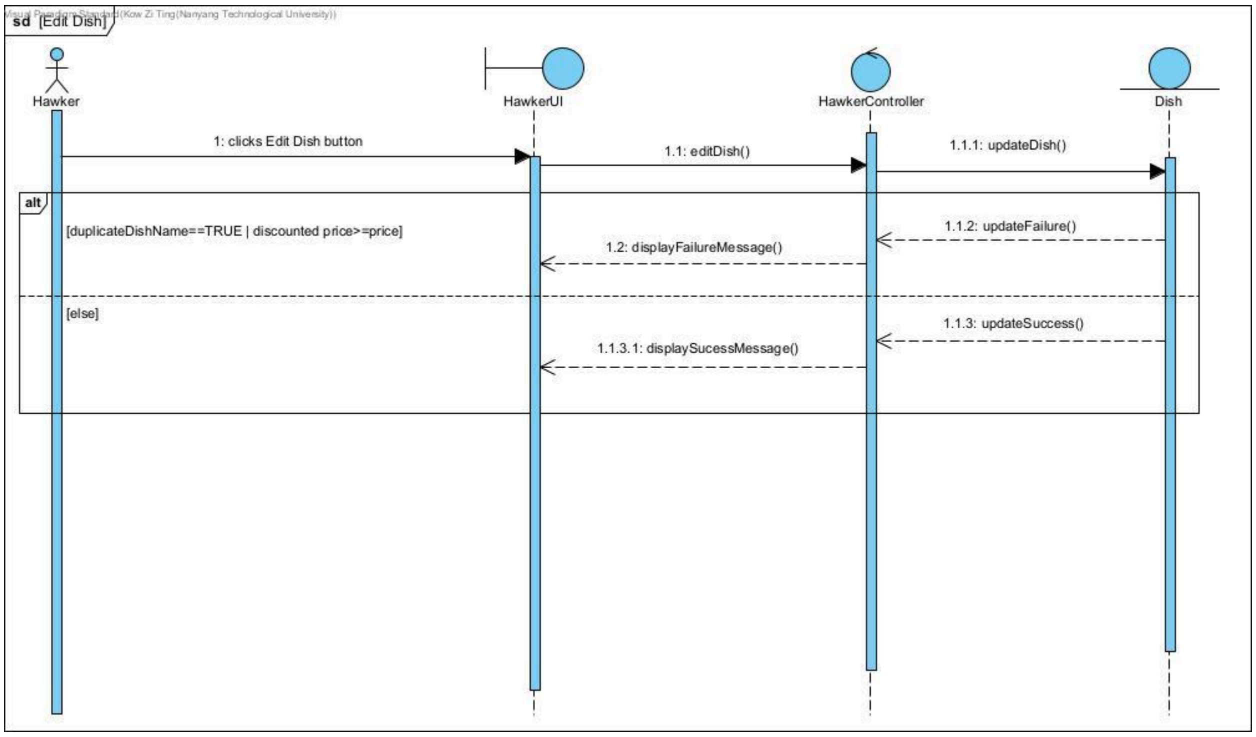

Visual Paradigm Standard(Kow Zi Ting(Nanyang Technological University))
sd [Edit Stall]
Hawker
HawkerUI
HawkerController
Stall
1: clicks Edit Stall button
1.1: editStall()
1.1.1: updateStall()
alt
[duplicateUnitNumber==TRUE]
1.1.2: updateFailure()
1.2: displayFailureMessage()
[else]
1.1.3: updateSuccess()
1.1.3.1: displaySucessMessage()

3.4 Add Dish



Visual Paradigm Standard(Kow Zi Ting(Nanyang Technological University))
sd [Add Dish]
Hawker
HawkerUI
HawkerController
Dish
1: clicks Add Dish button
1.1: addDish()
1.1.1: newDish()
alt
[duplicateDishName==TRUE | discounted price>=price]
1.1.2: addFailure()
1.2: displayFailureMessage()
[else]
1.1.3: addSuccess()
1.1.3.1: displaySucessMessage()

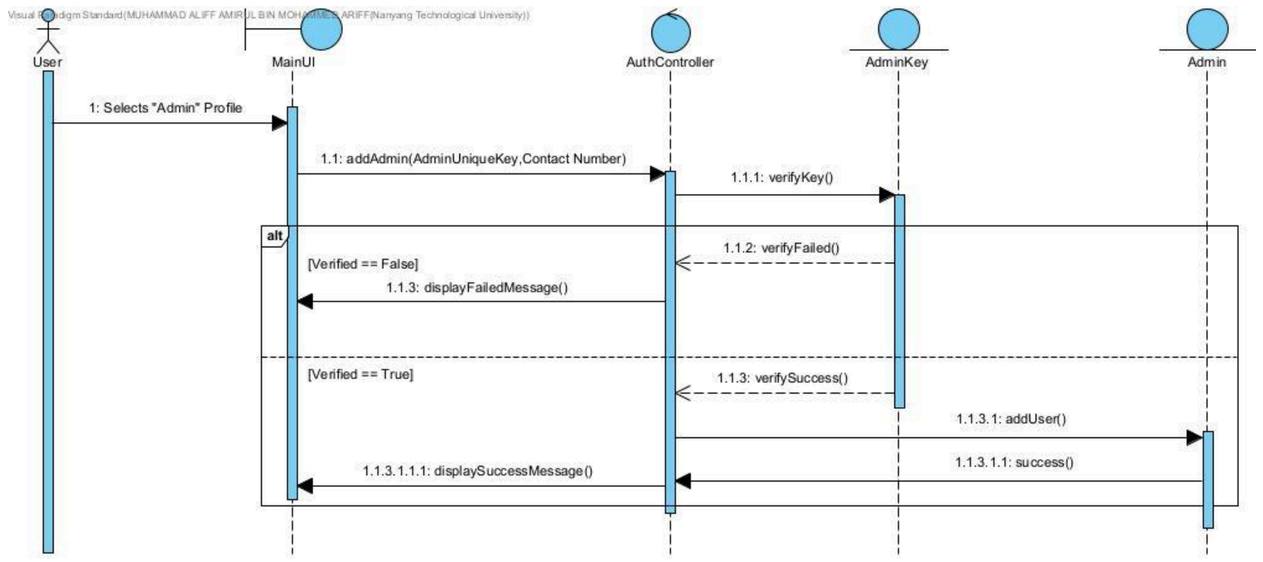
3.5 Edit Dish



Visual Paradigm Standard(Kow Zi Ting(Nanyang Technological University))
sd [Edit Dish]
Hawker
HawkerUI
HawkerController
Dish
1: clicks Edit Dish button
1.1: editDish()
1.1.1: updateDish()
alt
[duplicateDishName==TRUE | discounted price>=price]
1.1.2: updateFailure()
1.2: displayFailureMessage()
[else]
1.1.3: updateSuccess()
1.1.3.1: displaySucessMessage()

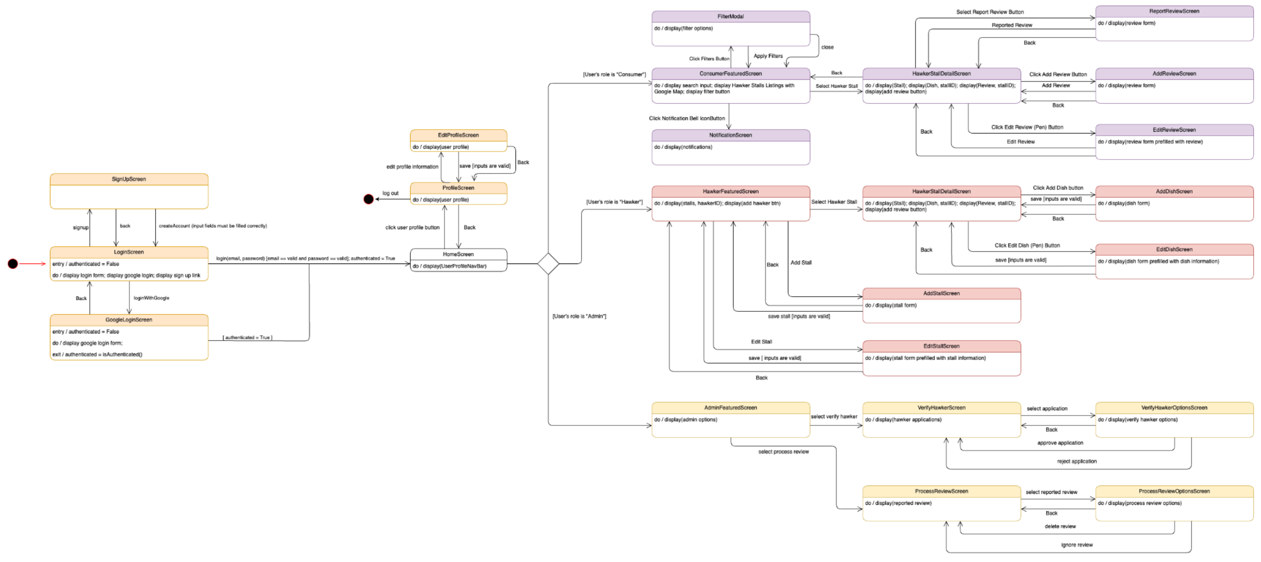
4. Use Cases for Admin

4.1 Admin Profile Creation


Visual Paradigm Standard(MUHAMMAD ALIFF AMIRUL BIN MOHAMMED ARIFF(Nanyang Technological University))
User
MainUI
AuthController
AdminKey
Admin
1: Selects "Admin" Profile
1.1: addAdmin(AdminUniqueKey,Contact Number)
1.1.1: verifyKey()
alt
[Verified == False]
1.1.2: verifyFailed()
1.1.3: displayFailedMessage()
[Verified == True]
1.1.3: verifySuccess()
1.1.3.1: addUser()
1.1.3.1.1: success()
1.1.3.1.1.1: displaySuccessMessage()

Dialog Map Diagram

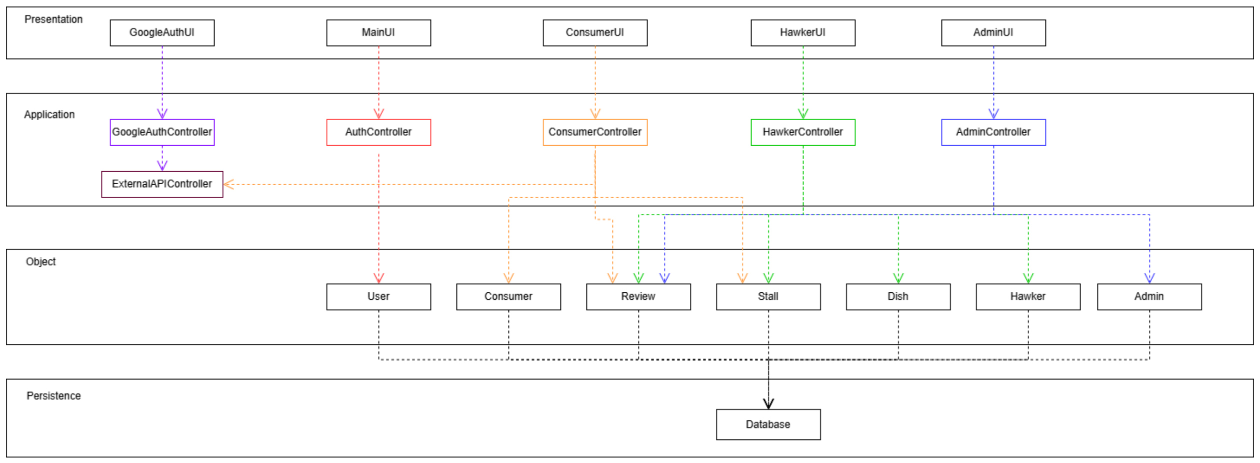
If the image is unclear, please refer to the accompanying raw PNG file uploaded with this document.


SignUpScreen
signup
back
createAccount (input fields must be filled correctly)
LoginScreen
entry / authenticated = False
do / display login form; display google login; display sign up link
Back
loginWithGoogle
GoogleLoginScreen
entry / authenticated = False
do / display google login form;
exit / authenticated = isAuthenticated()
[authenticated = True]
login(email, password) [email == valid and password == valid]; authenticated = True
HomeScreen
do / display(UserProfileNavBar)
click user profile button
Back
ProfileScreen
do / display(user profile)
log out
edit profile information
save [inputs are valid]
Back
EditProfileScreen
do / display(user profile)
[User's role is "Consumer"]
FilterModal
do / display(filter options)
Click Filters Button
Apply Filters
close
ConsumerFeaturedScreen
do / display search input; display Hawker Stalls Listings with Google Map; display filter button
Click Notification Bell IconButton
NotificationScreen
do / display(notifications)
Back
Select Hawker Stall
HawkerStallDetailScreen
do / display(Stall); display(Dish, stallID); display(Review, stallID); display(add review button)
Select Report Review Button
Reported Review
Back
ReportReviewScreen
do / display(review form)
Click Add Review Button
Add Review
Back
AddReviewScreen
do / display(review form)
Back
Click Edit Review (Pen) Button
Edit Review
EditReviewScreen
do / display(review form prefilled with review)
[User's role is "Hawker"]
HawkerFeaturedScreen
do / display(stalls, hawkerID); display(add hawker btn)
Select Hawker Stall
HawkerStallDetailScreen
do / display(Stall); display(Dish, stallID); display(Review, stallID); display(add review button)
Click Add Dish button
save [inputs are valid]
Back
AddDishScreen
do / display(dish form)
Back
Click Edit Dish (Pen) Button
save [inputs are valid]
EditDishScreen
do / display(dish form prefilled with dish information)
Back
Add Stall
AddStallScreen
do / display(stall form)
save stall [inputs are valid]
Edit Stall
save [inputs are valid]
EditStallScreen
do / display(stall form prefilled with stall information)
Back
[User's role is "Admin"]
AdminFeaturedScreen
do / display(admin options)
select verify hawker
VerifyHawkerScreen
do / display(hawker applications)
select application
Back
VerifyHawkerOptionsScreen
do / display(verify hawker options)
approve application
reject application
select process review
ProcessReviewScreen
do / display(reported review)
select reported review
Back
ProcessReviewOptionsScreen
do / display(process review options)
delete review
ignore review

System Architecture

Our application implements a layered system architecture. Such a design allows us to segment our application into more manageable layers.

If the image is unclear, please refer to the accompanying raw PNG file uploaded with this document.



Presentation Layer

The Presentation Layer governs the interactions between the User and our application. It is also responsible for displaying data. It consists of our UI components which call relevant Controllers in the Logic Layer. The components of this layer include:

1. **MainUI**
   MainUI facilitates account creation, profile creation and login processes by calling AuthController.
2. **AdminUI**
   AdminUI facilitates Admins in performing their job tasks by calling AdminController.
3. **HawkerUI**
   HawkerUI facilitates Hawkers in the maintenance of their stall pages, menus and stall reviews by calling HawkerController.
4. **ConsumerUI**
   ConsumerUI facilitates Consumers in searching for stalls and leaving reviews by calling ConsumerController.

Application Layer

Involving the Controller classes, the Application Layer bridges the Presentation and Object Layers. It implements the use cases by calling entity classes in the Object Layer and retrieves processed results. The components of this layer include:

1. **AuthController**
   AuthController is called by MainUI for User account creation and login operations. Its logic includes the login and sign up methods.
2. **ConsumerController**
   ConsumerController is called by ConsumerUI for Consumer-specific actions, including: searching and favouriting stalls, modifying and reporting reviews. Its logic includes the methods to perform the aforementioned operations.
3. **HawkerController**
   HawkerController is called by HawkerUI for Hawker-specific actions, including: modifying stalls and dishes. Its logic encompasses the methods to perform these operations.
4. **AdminController**
   AdminController is called by AdminUI for Admin-specific actions, including: processing reported reviews and verifying Hawker profiles. Its logic includes the methods to perform the aforementioned operations.

Object Layer

The Object Layer houses the entity classes and their business logic, which are called by the Application layer. The components of this layer include:

1. **User**
   User contains the name, email address, account password and photo of the registered User.
2. **Consumer**
   Consumer contains the address, contact number, dietary preferences, preferred cuisines, ambulatory status and unique consumerID of each Consumer.
3. **Review**
   Review contains the rating, content (as text) and unique reviewID of each submitted Review.
4. **Stall**
   Stall contains the name, hawker centre, unit number, operating hours, hygiene rating, cuisine types, estimated wait time and unique stallID of each Stall.
5. **Dish**
   Dish contains the dish name, price, photo and unique dishID of each dish.
6. **Hawker**
   Hawker contains the SFA food license number and contact number of each Hawker.
7. **Admin**
   Admin contains the unique adminID of each Admin.

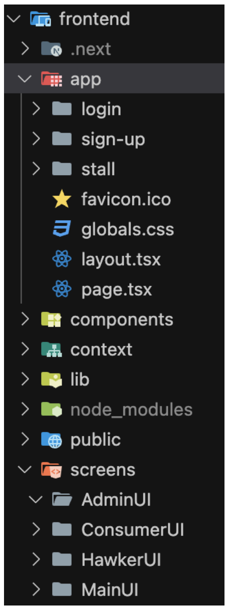
Persistence Layer

The Persistence Layer manages interactions with the databases that store entity data.

Application Skeleton

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

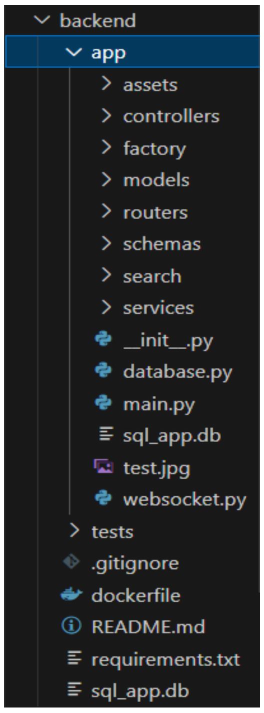
Frontend



```
frontend
  .next
  app
    login
    sign-up
    stall
    favicon.ico
    globals.css
    layout.tsx
    page.tsx
  components
  context
  lib
  node_modules
  public
  screens
    AdminUI
    ConsumerUI
    HawkerUI
    MainUI
```

The frontend is built with Next.js, TypeScript and Next.js, using TailwindCSS for styling. It is organized into role-based interfaces like MainUI, ConsumerUI, HawkerUI and AdminUI, each handling specific user tasks. The entry point is page.tsx with shared layout and styles defined globally.

Reusable components, utilities and static assets are structured into separate folders for clarity. This modular design improves maintainability, supports collaboration and also ensures consistent UI across the app.



```
backend
  app
    assets
    controllers
    factory
    models
    routers
    schemas
    search
    services
    __init__.py
    database.py
    main.py
    sql_app.db
    test.jpg
    websocket.py
  tests
  .gitignore
  dockerfile
  README.md
  requirements.txt
  sql_app.db
```

Backend

The backend uses FastAPI with Python, structured into routers, controllers, services, models and schemas. It connects to a MySQL database using SQLAlchemy with a factory pattern for session handling.

The logic is modular and separated by feature domains. Validation is handled with Pydantic, and trie-based search improves performance. WebSocket support and test coverage are also included for real-time features and reliability

Appendix

Key Design Issues

In the design of our application, we sought to overcome the following key design issues to ensure that Hawka remains robust and future-proof.

Maintainability

Paramount to any application, maximising maintainability ensures that updates, including the addition of future features, can be made to the application smoothly.

In our application, we have consciously adopted a Separation of Concerns (SOC) framework in view of enhancing maintainability. This is demonstrated through our use of Object-Oriented Programming (OOP) and a Layered Architecture.

Object-Oriented Programming

- Using an OOP framework and with consideration for the Single Responsibility Principle (SRP), we segregated our application's needs into separate classes, each responsible for a specific aspect of functionality.
- The use of an object-oriented programming paradigm also allows us to exercise polymorphism for the easy extension of each class' functionalities in the long run.

Layered Architecture

- As mentioned in our earlier segment on System Architecture, implementing a layered architecture allows us to split Hawka's system into distinct layers, each focussed on a specific concern.
    - **Presentation Layer**
      Governs the interactions between the User and our application
    - **Application Layer**
      Handles the application logic, implements the use cases and retrieves processed results from the Object Layer.
    - **Object Layer**
      Manages the business logic and houses the entity classes.
    - **Persistence Layer**
      Controls interactions with our databases.

Data Validation

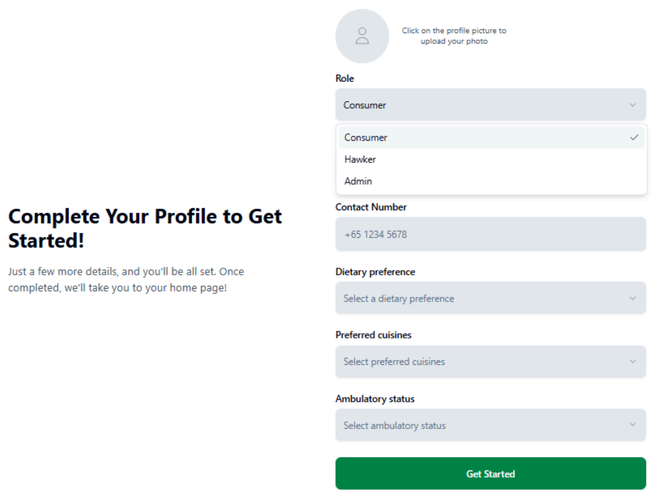
As a dynamic application supporting interactions between the user and the UI, ensuring that user inputs are valid becomes imperative to ensure the integrity of our data. We do so through the use of **case checks** and **predefined options**.

Case Checks
The use of case checks ensures that data stored is in a standardised format. This comes particularly useful in supporting our search functions.

Trie Data Structure
In using a trie data structure to facilitate Consumers' search for their favourite dishes, we perform multiple checks and changes to ensure all information populated in the trie is valid and uniform.

- **Missing Value Checks**
  Before data is populated into the trie, we first check if all necessary fields, for example hawker_name, hawker_id, dish_name and dish_id are valid. This ensures that information inserted into the trie will not corrupt its structure or cause unintended behaviour.
- **Converting Name Cases**
  In addition to checking for missing values, we also ensure that all dish names are standardised to the lowercase form before populating them to the database.

Predefined Options
To prevent the cases of unexpected user inputs and undefined user behaviour, we provide Consumers and Hawkers with predefined options where possible. This is particularly exhibited in the Profile Creation form, which allows users to select one out of three predefined roles, after which MainUI displays a form corresponding to the role.


Complete Your Profile to Get Started!

Just a few more details, and you'll be all set. Once completed, we'll take you to your home page!

Role: Consumer

Dropdown list in profile creation form