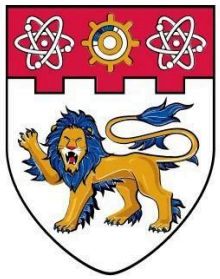


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2006 – Software Engineering

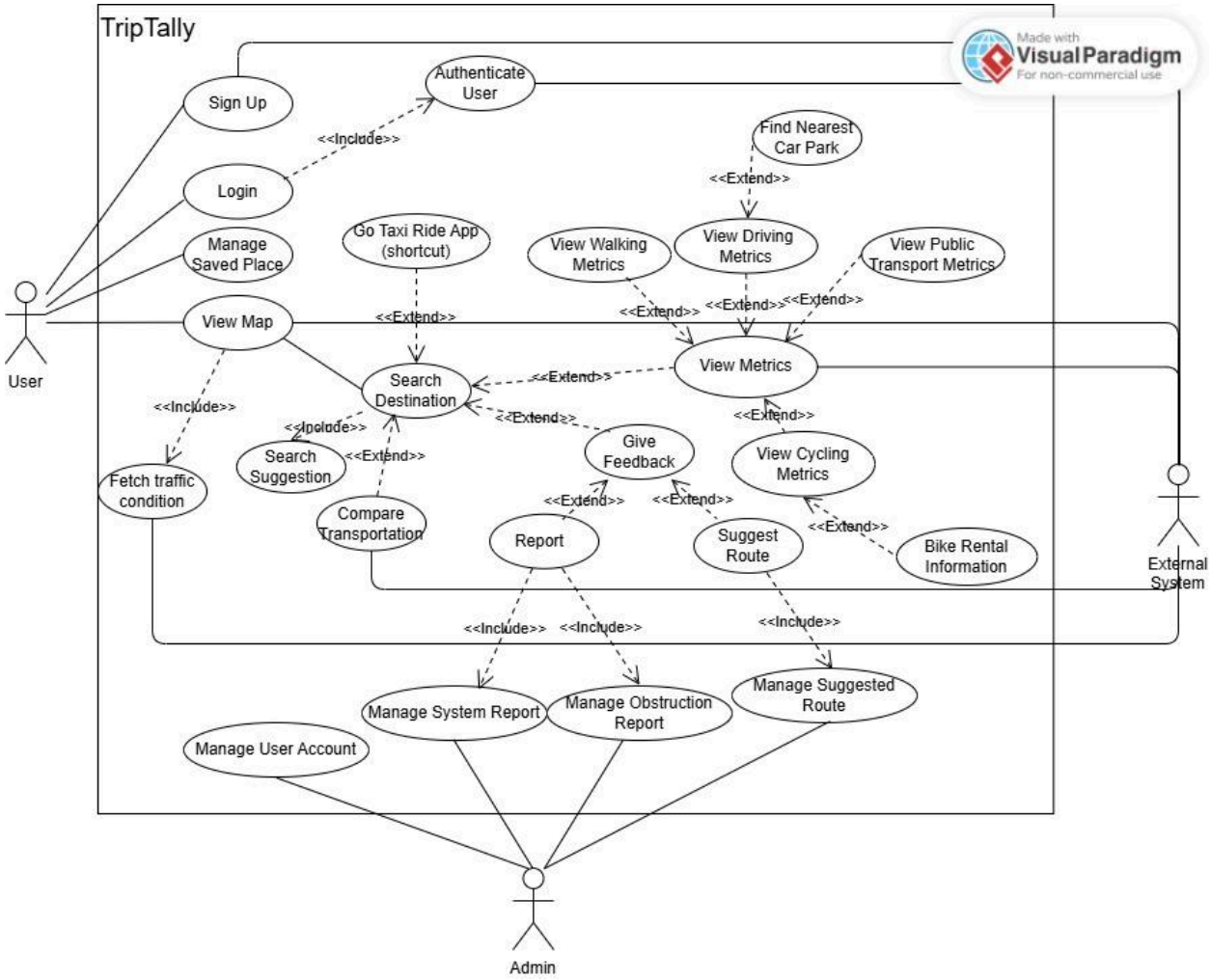
Lab 3 Deliverables

Lab Group	SCSI-30
Team	TEAM 5
Members	Tan Wei Song Russell Bryan (U2421844K)
	Justin Woon Thean Woon (U244641B)
	Ethan Jared Chong Rui Zhi (U2421895B)
	Evelyn Theresia Cuaca (U2320523L)
	Parvez Kurniawan Wijaya (U2423845G)

Table of Contents

1. Complete Use Case Model

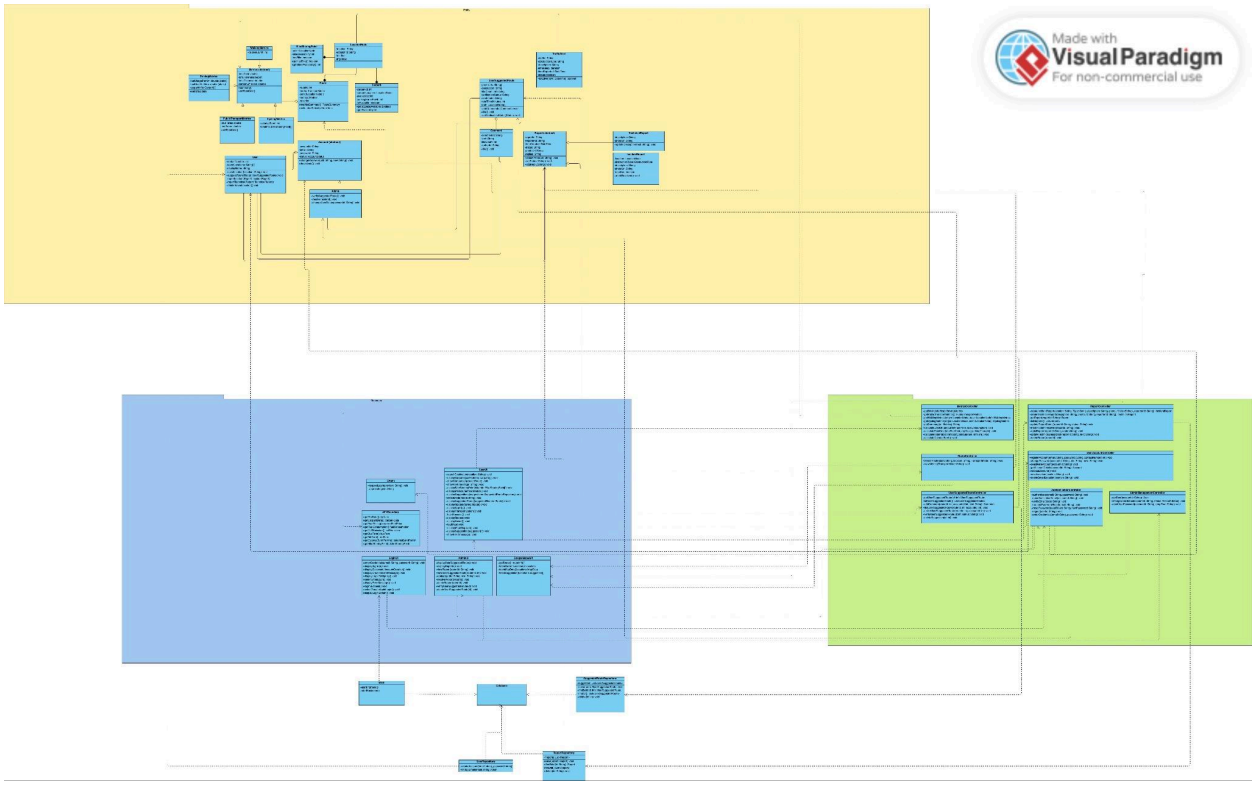
If the image is unclear, please refer to the raw png file that is uploaded together with this document.



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.



Main

Class that starts up the frontend and backend of the application
Operations:

- **startFrontend():** starts the frontend User Interfaces.
- **startBackend():** starts the backend application logic, databases and APIs.

User Interfaces (UI)

User Interfaces are the means by which the user and the computer system/ application interact.

- **LoginUI:** These are the screens that serve as the main authentication for both Users and Admins. Upon logging in, the system implements role-based access control, directing users to their appropriate interface based on their roles.
- **UserUI:** These are the screens accessible by only the Users. It allows the to view and interact with features such as the map, navigation, directions and real time data on traffic conditions and carpark availability.
- **AdminUI:** These are the screens accessible by only the Admins. Admin handles the verification of reports and is able to make changes to data provided on the user end of the application. Admin has to log in as a user in order to use User functionalities

Factory Pattern (via dependency injection factories)

In our design, we apply the factory pattern through FastAPI's dependency injection. db.py acts as the session factory, the function `get_db` creates a fresh SQLAlchemy session per request , so endpoints never construct or manage connections themselves. In the our API layer, each route assembles what it needs from factories: it *requests* a db session (factory #1), uses that to instantiate a repository (factory step #2), and then instantiates a service that depends on the repo (factory step #3).

Singleton Pattern (Configuration Settings)

We treat the app's configuration as a single, shared object so everyone reads the same settings. In core/config.py, a Pydantic Settings class loads values (DATABASE_URL, SECRET_KEY) from environment variables/.env. We then create one instance, settings, and import that everywhere. This is singleton-like because there's effectively a single source of truth for config across the whole process.

Strategy Pattern (Interchangeable Repositories)

In our design, we use the strategy pattern to keep the business logic independent of any specific data source or implementation. We define ports (interfaces) that specify the operations each repository should perform, such as creating, updating, or retrieving data. Then, we implement multiple adapters for example, the SQLAlchemy-based repositories for different entities. This approach allows the

application to easily switch between different data-access implementations (e.g PostgreSQL, or MongoDB) without changing the core logic.

Facade Pattern (Service Layer Simplifaction)

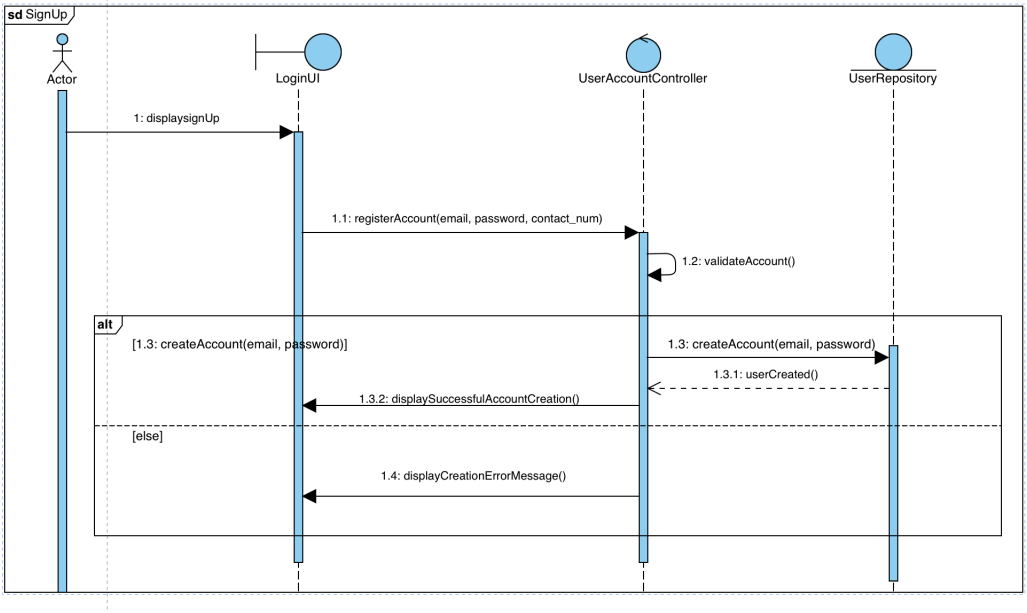
dIn our design, we apply the facade pattern through the service layer. Each service acts as a single interface that coordinates multiple internal operations. For example, instead of having the API endpoint handle database queries, check for existing users, and hash passwords separately, the endpoint simply calls UserService.register(). The service (facade) hides all that complexity inside its method.

By doing this, our endpoints stay clean and focused only on request handling, while the service layer manages the business logic.

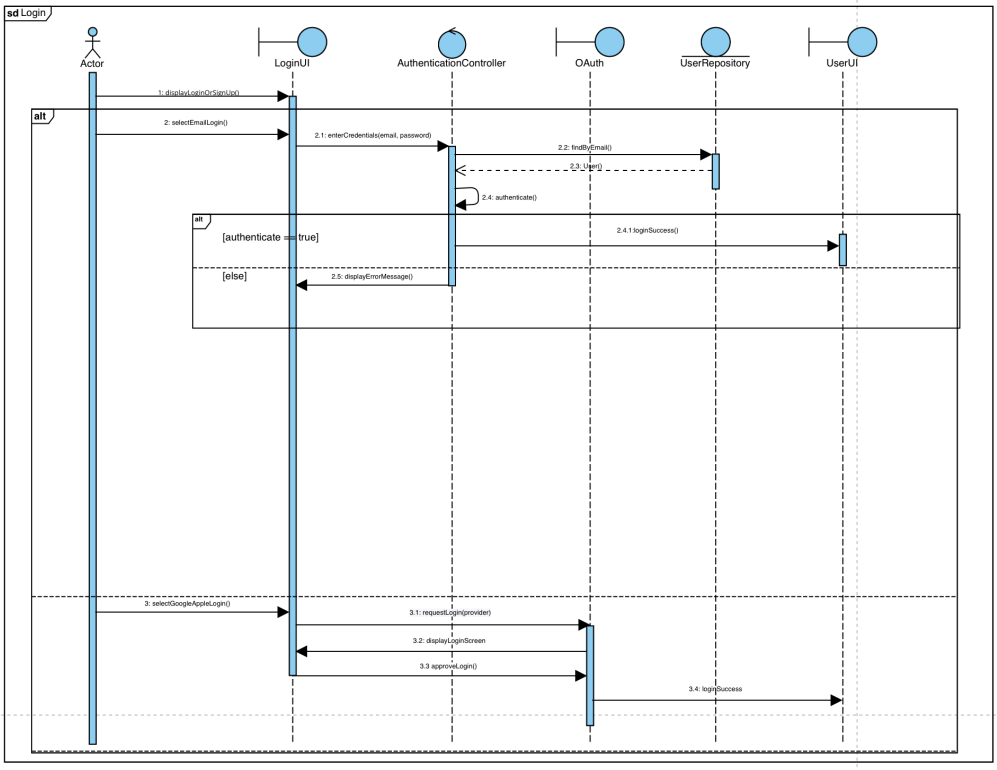
B. Sequence Diagram

I. For Use Cases under I

I.I SignUp

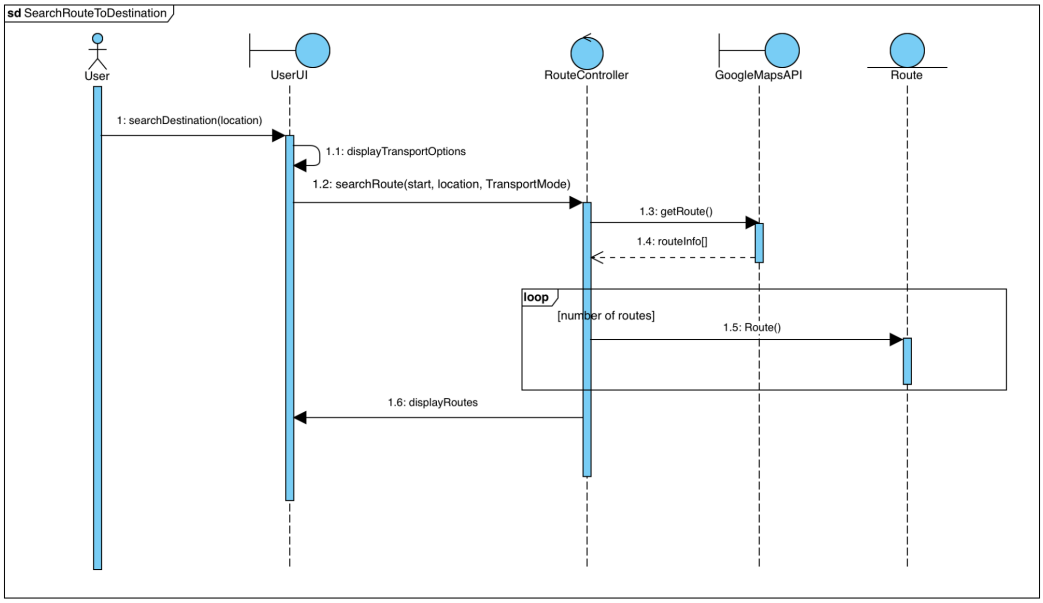


I.II Login

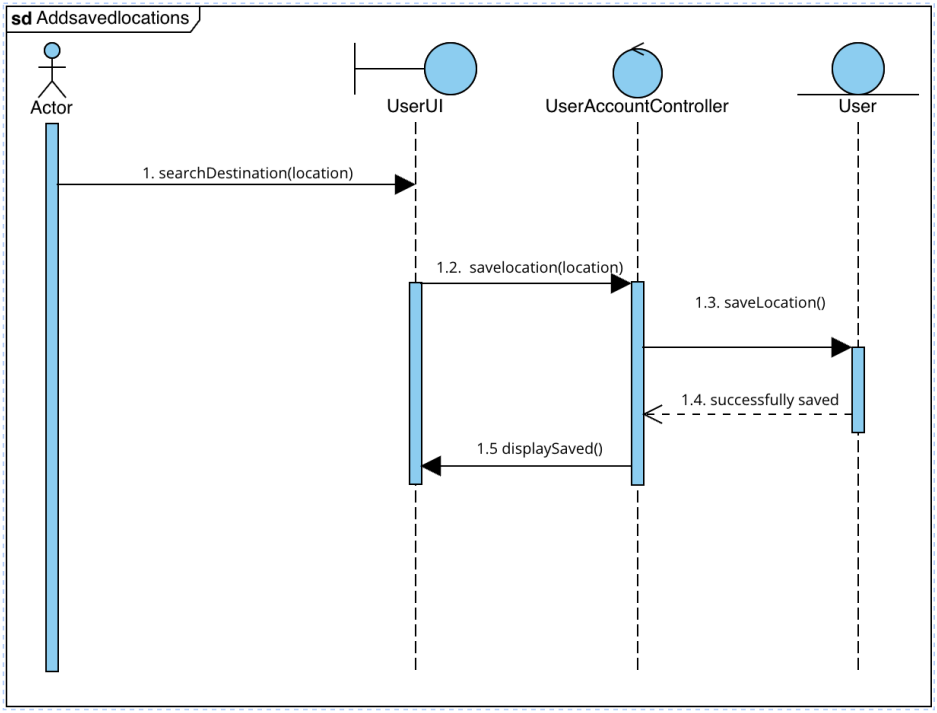


II. For Use Cases Under II

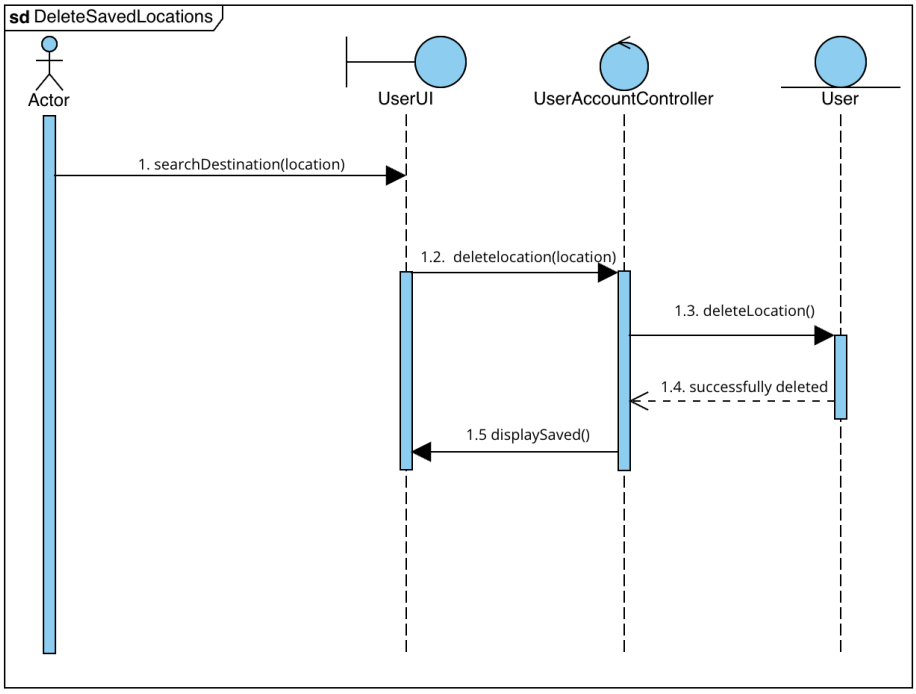
II.I SearchRouteToDestination



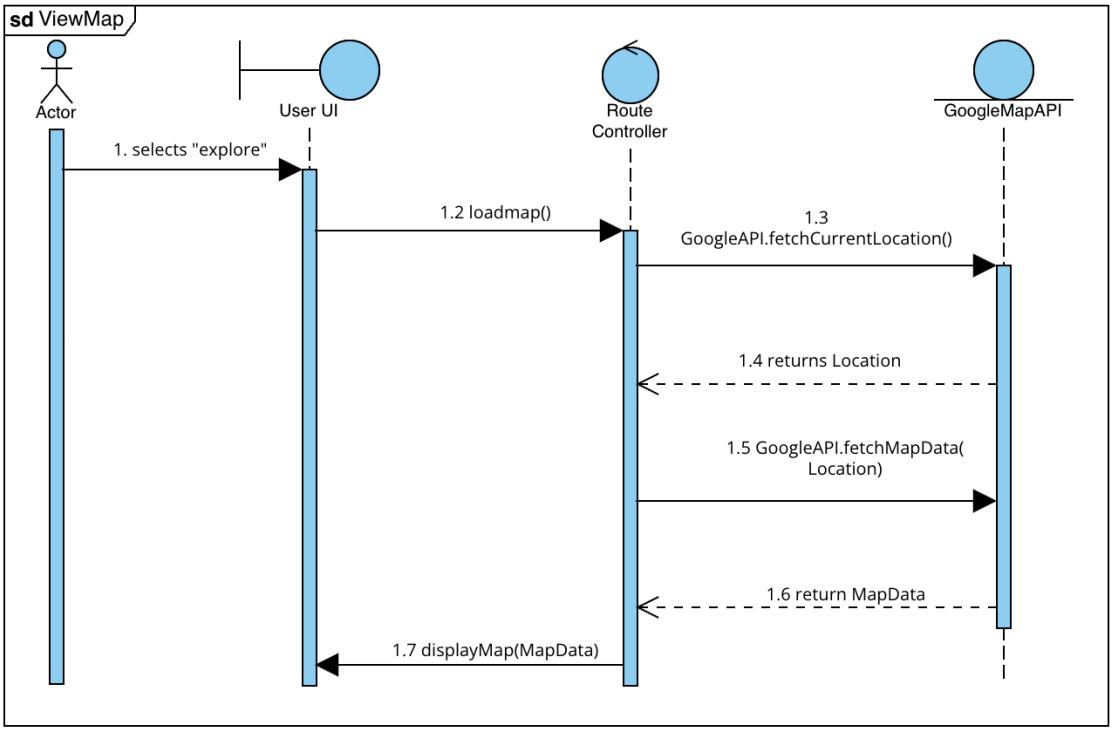
II.II ManageSavedPlaces



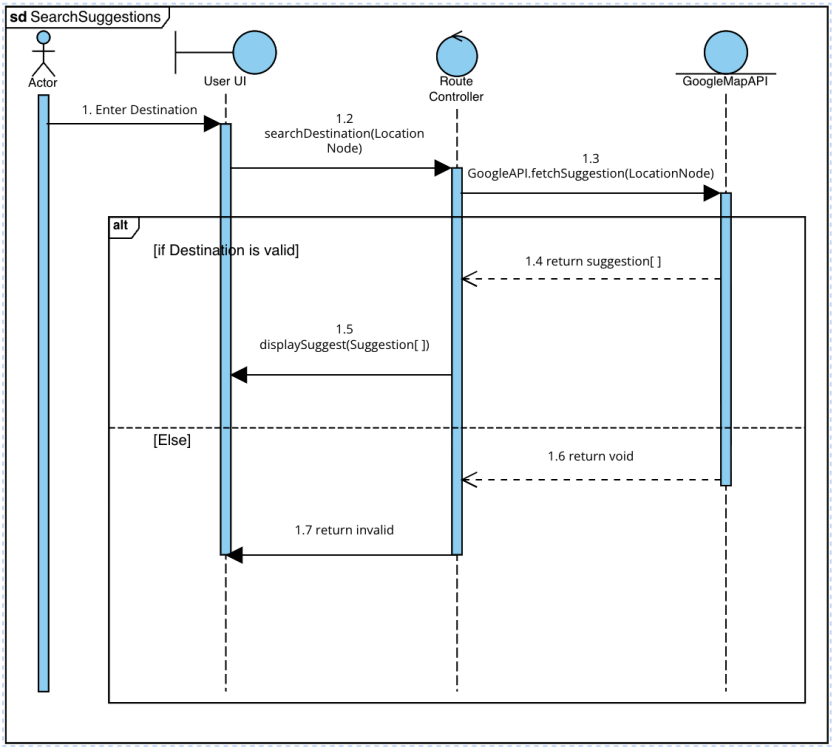
II.III DeleteSavedLocations



II.IV View map

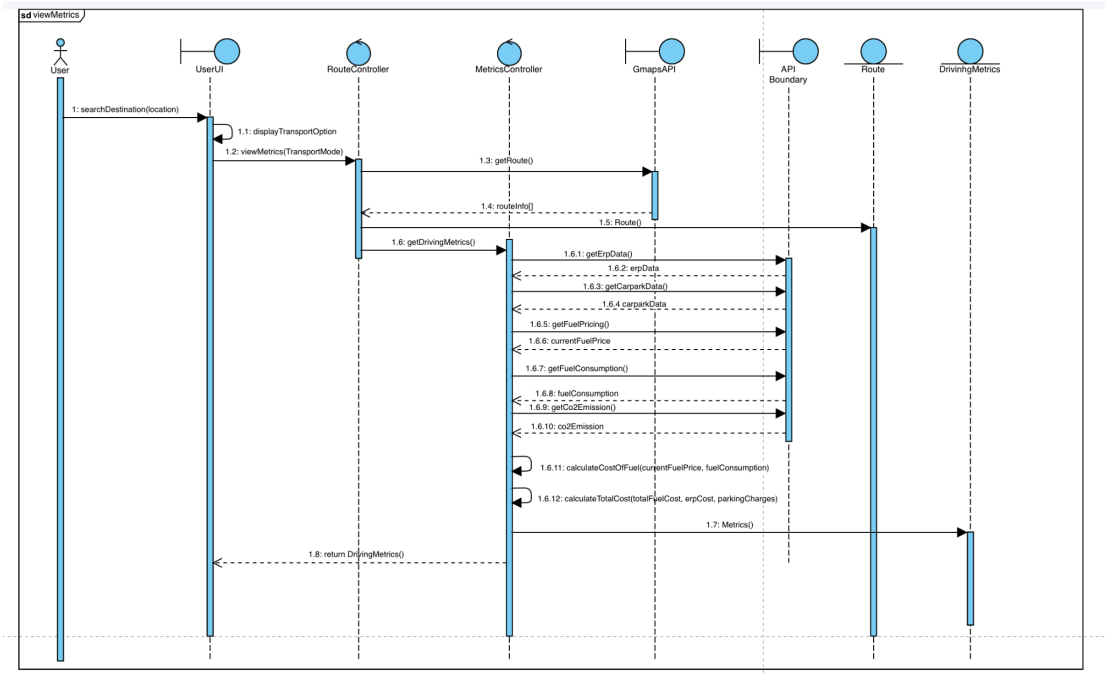


II.V SearchSuggestion

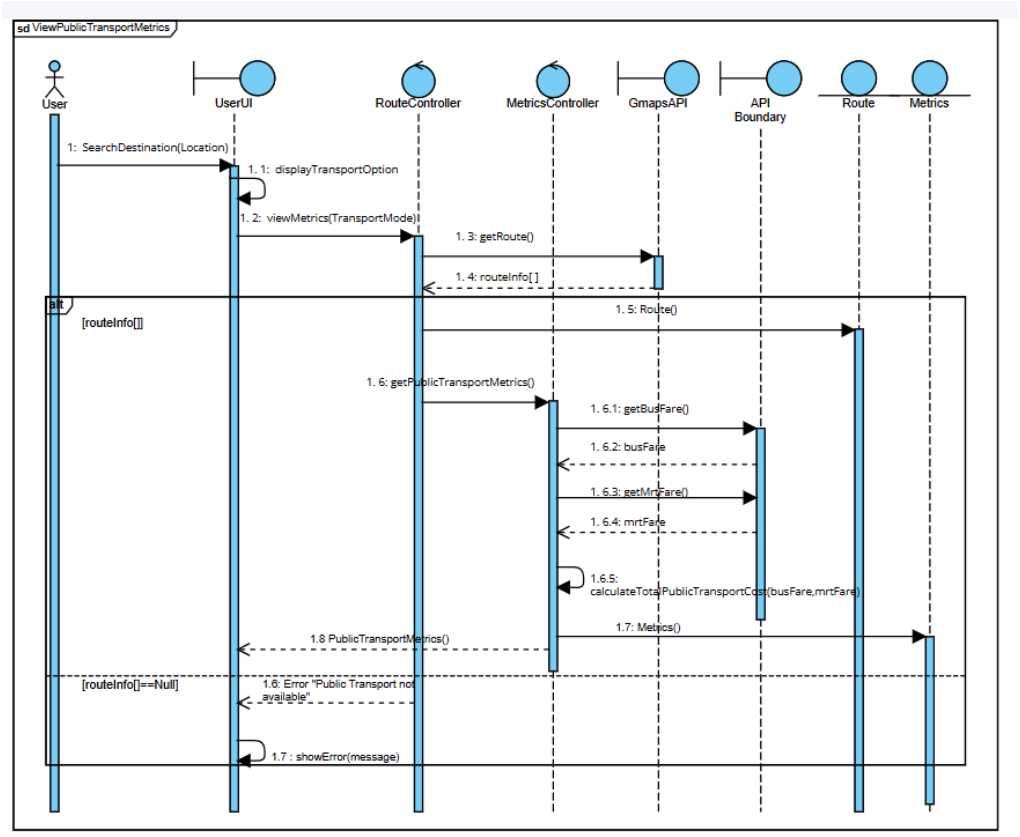


III. For Use Cases Under III

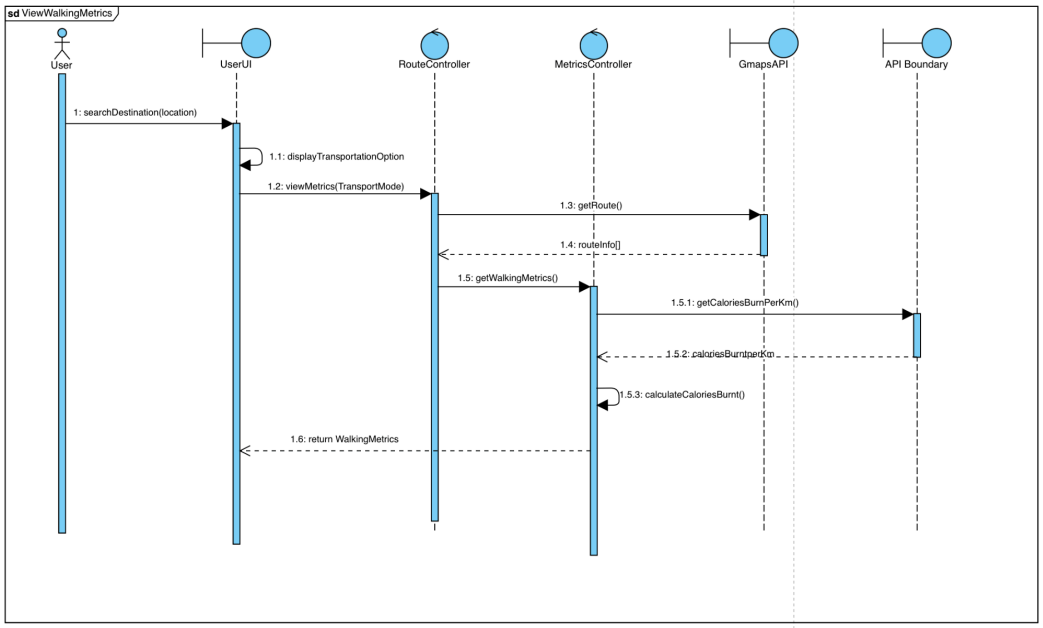
III.I ViewDrivingMetrics



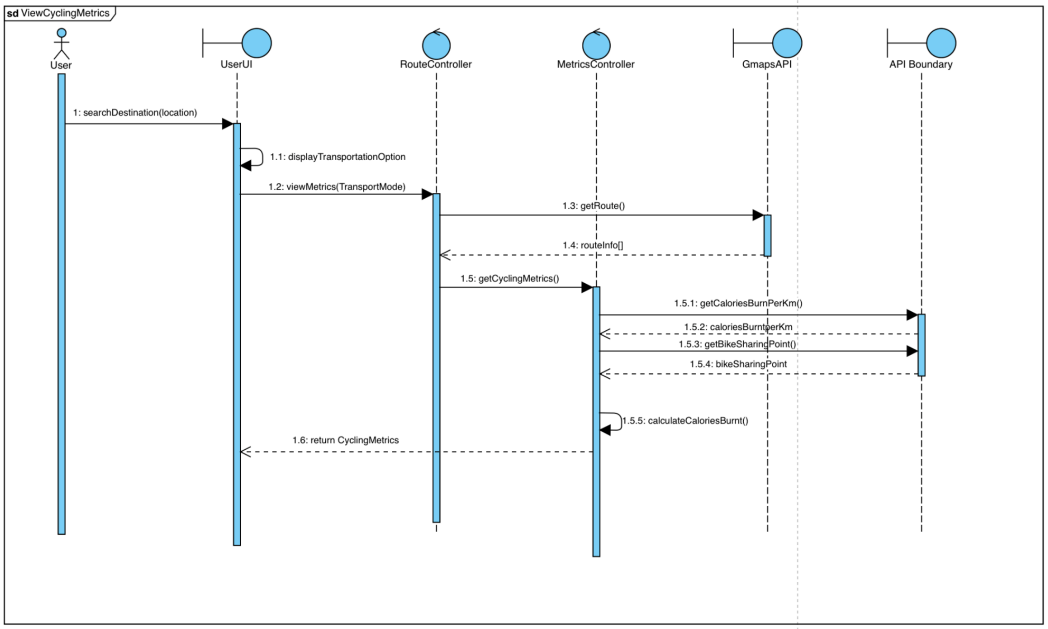
III.II ViewPublicTransportMetrics



III.III ViewWalkingMetrics

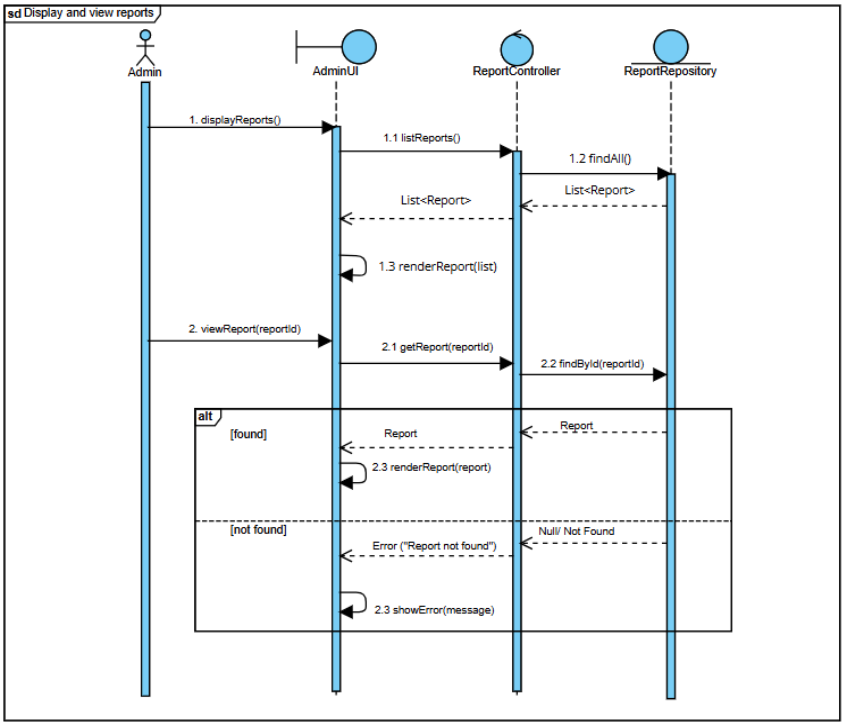


III.IV ViewCyclingMetrics

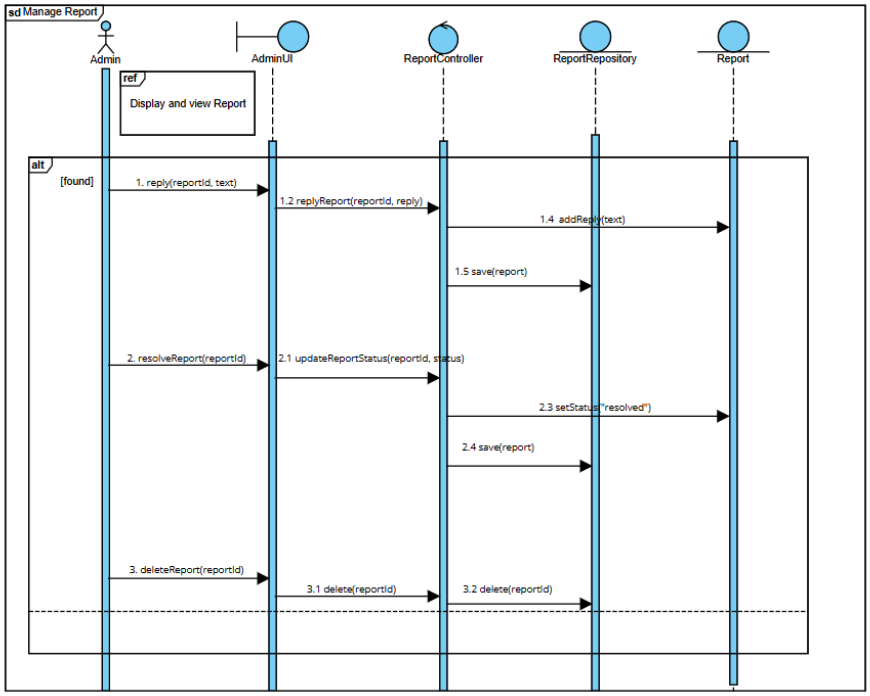


IV. For Use Cases Under IV

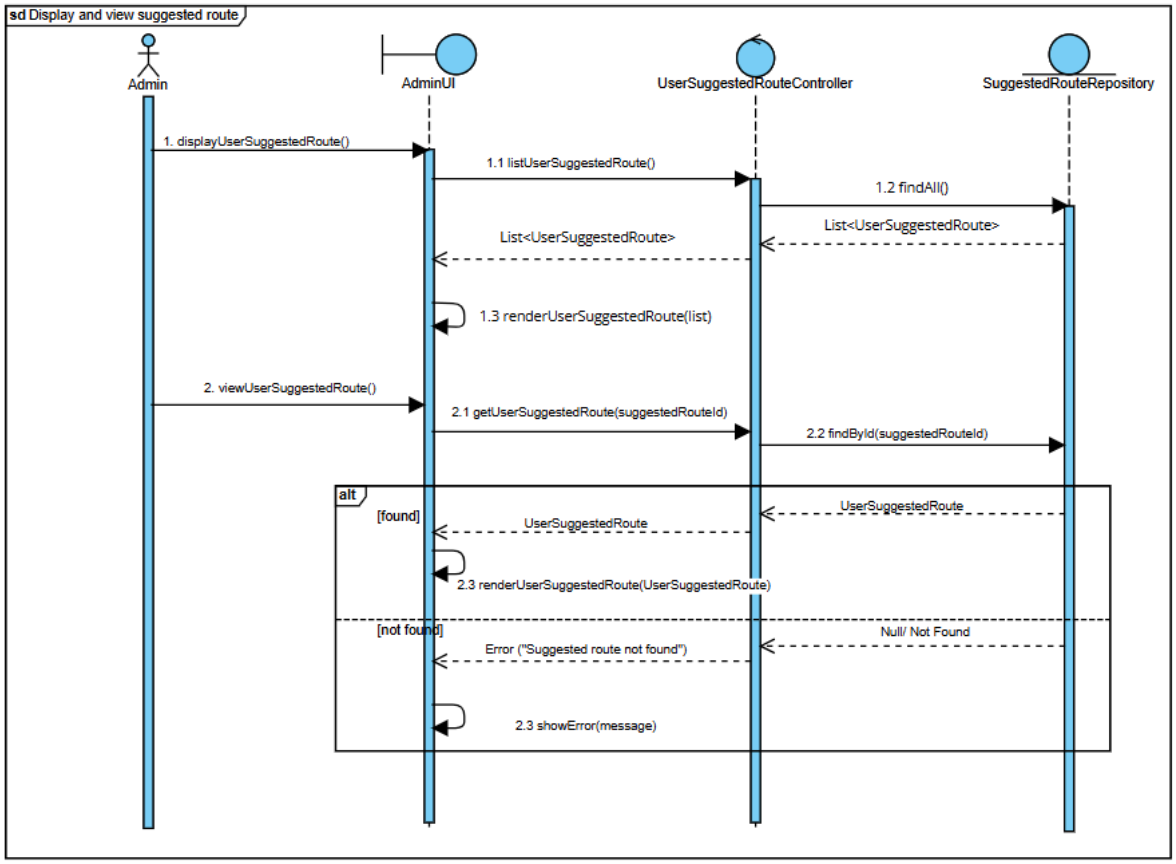
IV.I ViewReports (Admin)



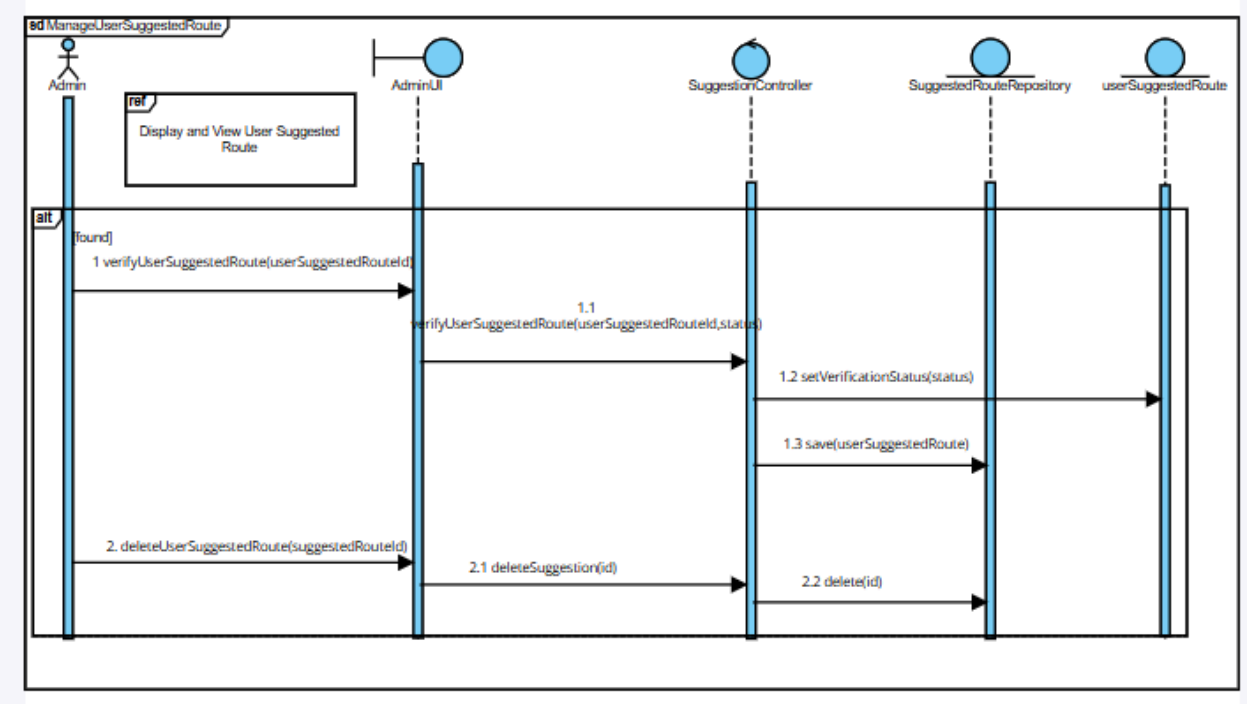
IV.II ManageReport (Admin)



IV.III ViewSuggestedRoute (Admin)

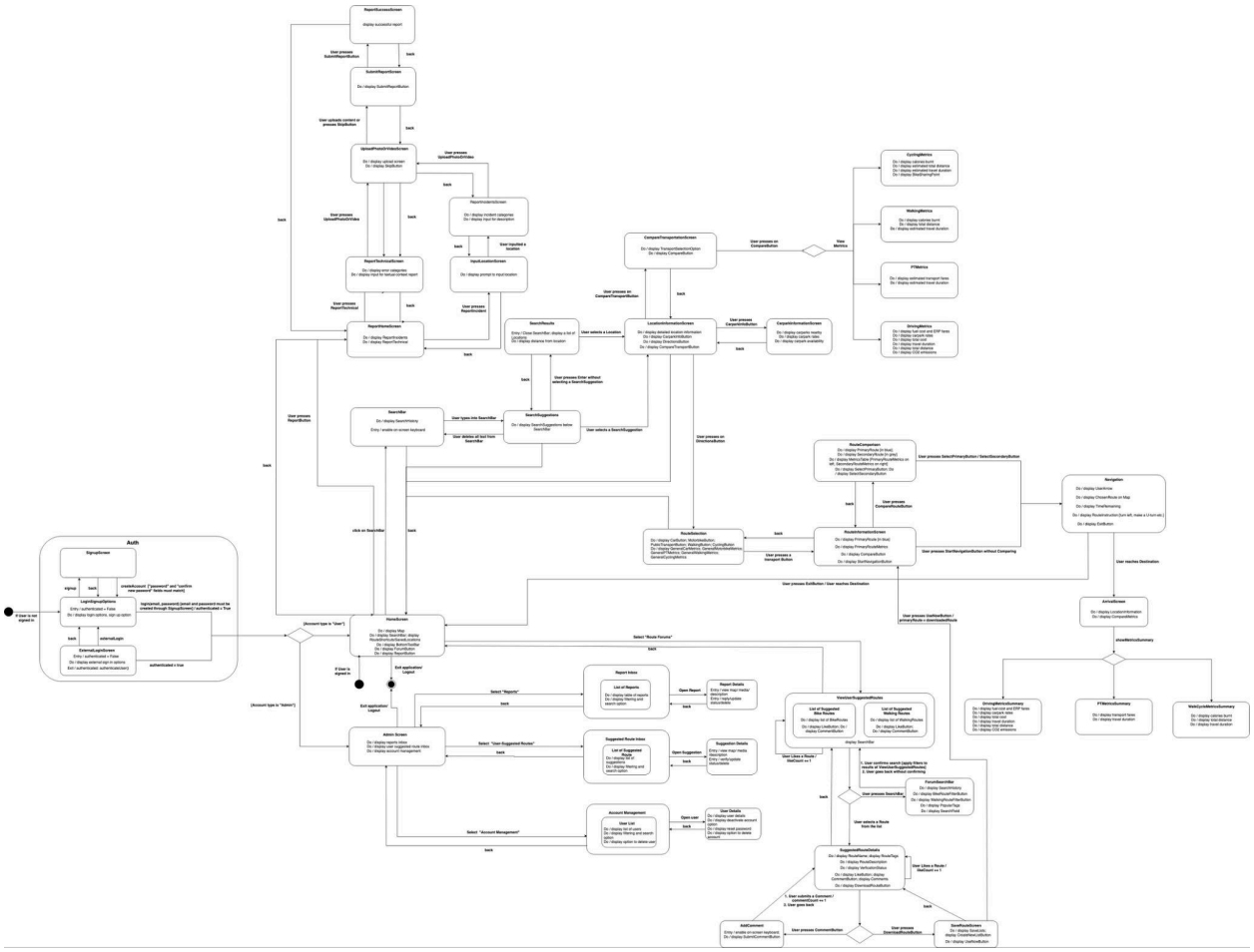


IV.IV ManageUserSuggestedRoute (Admin)



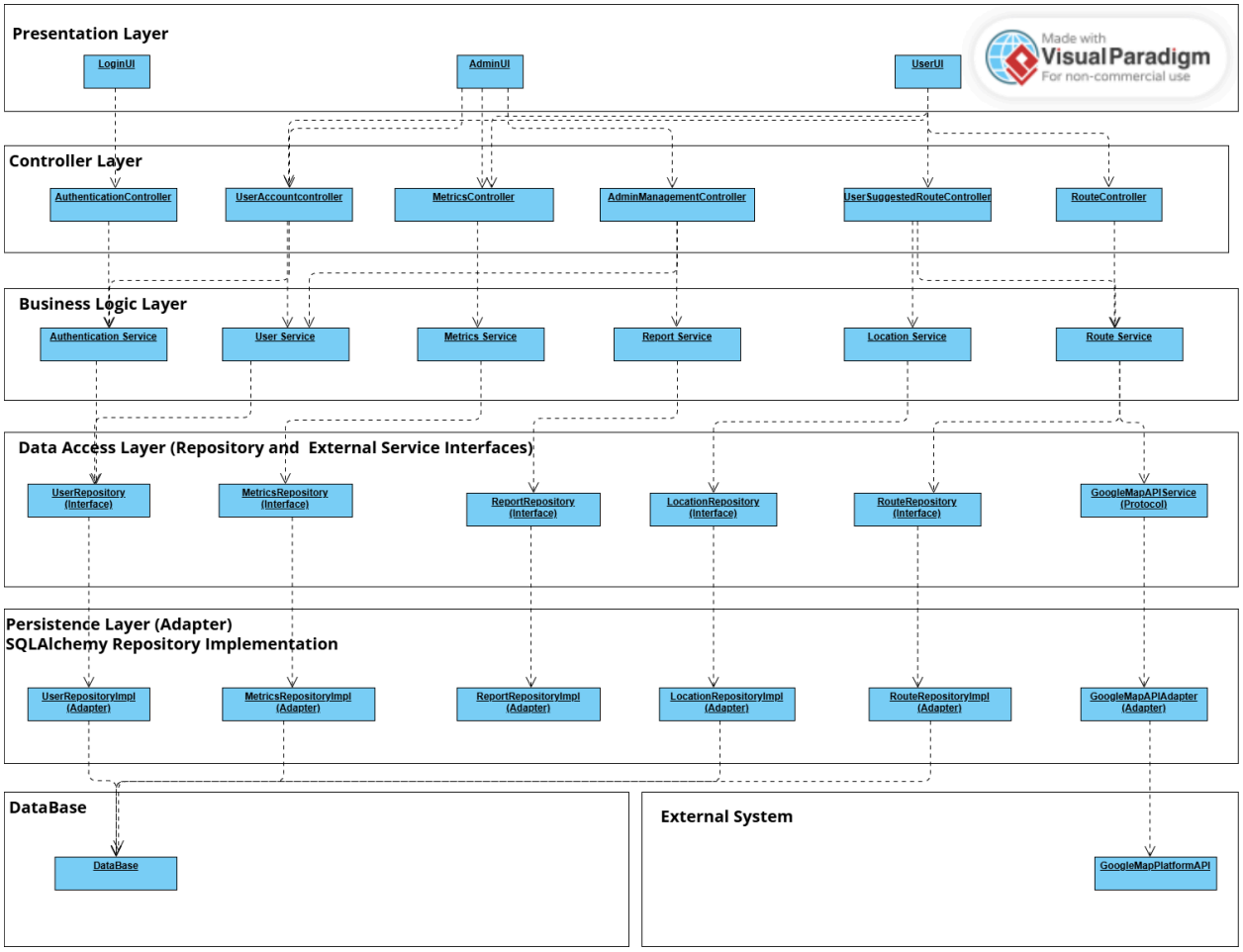
C. Dialog Map

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 responsible for the interaction between Users/Admins and the application. The various UI would call for the different controllers to run the application logic from the Application Logic Layer.

- **LoginUI**
  Handles the authentication of Users and Admins by calling AuthenticationController, directing them to their specified UIs based on their assigned roles.
- **UserUI**

User UI will call MetricsController, UserSuggestedRouteController, ReportController and RouteController.

- **AdminUI**
  Admin UI will call UserAccountController, AdminManagementController and MetricsController.

Controller Layer

Controllers handle incoming requests from the UIs, validate data, and delegate processing tasks to their respective services.

- **AuthenticationController**
  Validates user credentials and communicates with AuthenticationService for login verification.
- **UserAccountController**
  Manages user information by calling UserService for operations such as updating profiles or managing accounts.
- **MetricsController**
  Coordinates with MetricsService to retrieve and send data to the UI.
- **AdminManagementController**
  Handles administrative operations and communicates with UserService or ReportService as needed.
- **UserSuggestedRouteController**
  Handles user-submitted route suggestions and passes them to LocationService and RouteService for processing.
- **RouteController**
  Facilitates route display, retrieval by calling RouteService.

Business Logic Layer

This layer contains the core application logic.
Each service defines business rules, processes data, and coordinates between repositories and external APIs through interfaces.

- **AuthenticationService** - Authenticates users, manages login sessions, and interacts with UserRepository.
- **UserService** - Handles user-related logic such as profile updates, registration, etc.
- **MetricsService** - Processes data and retrieves performance metrics from MetricsRepository.
- **ReportService** - Manages user-submitted reports from ReportRepository.
- **LocationService** - Manages location-based information and works with LocationRepository.
- **RouteService** - Responsible for route planning and optimization. It integrates data from RouteRepository and GoogleMapAPIService to generate accurate routes.

Data Access Layer

This layer defines interfaces (contracts) that describe how services interact with data sources and external APIs.
It separates logic from the underlying implementation, allowing flexibility and easier maintenance.

- UserRepository, MetricsRepository, ReportRepository, LocationRepository, and RouteRepository define methods for CRUD operations and data retrieval.
- GoogleMapAPIService (Protocol) defines how the system interacts with external mapping services like Google Maps.

Persistence Layer

This layer contains the actual implementation of repository interfaces.
Each adapter uses SQLAlchemy to perform database operations defined by the interfaces.

- UserRepositoryImpl (Adapter) - Implements UserRepository for CRUD operations on user data.
- MetricsRepositoryImpl (Adapter) - Implements MetricsRepository to fetch and store performance metrics.

- ReportRepositoryImpl (Adapter) - Implements ReportRepository to retrieve and generate stored reports.
- LocationRepositoryImpl (Adapter) - Implements LocationRepository for location data handling.
- RouteRepositoryImpl (Adapter) - Implements RouteRepository to store and fetch route information.
- GoogleMapAPIAdapter - Implements GoogleMapAPIService to connect with GoogleMapPlatformAPI, retrieving map and routing data from the external system.

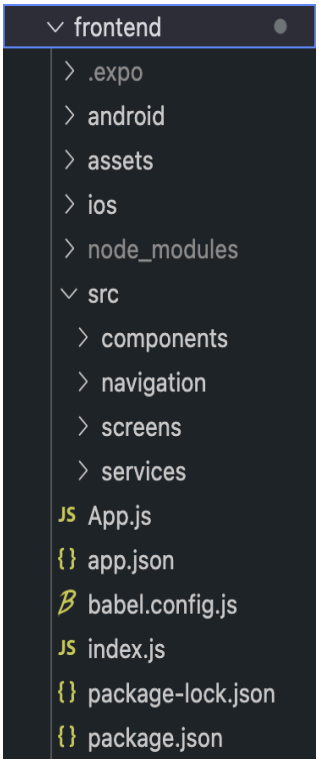
Database and External System

This layer handles data storage and external integrations.

- Database - Stores all application data, including users, metrics, reports, locations, and routes.
- External System (GoogleMapPlatformAPI) - Provides route and map information used by RouteService through the GoogleMapAPIAdapter.

4. Application Skeleton

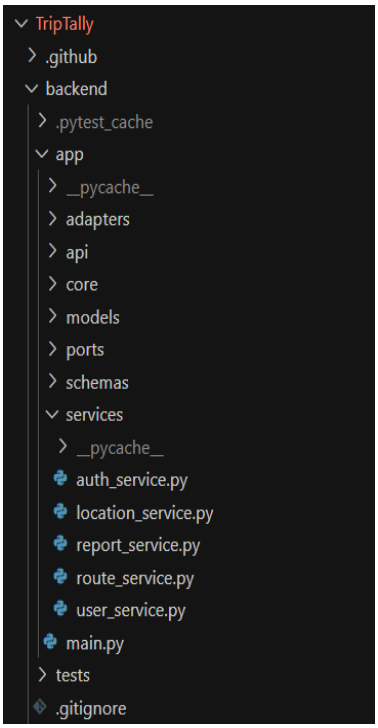
A. Frontend



The frontend of our application is built on React Native and organised into a modular structure where each folder serves a specific purpose in the development and configuration of our application.

- **.expo/** - generated by the Expo framework and simplifies development by handling native builds and deployment.
- **android/ and ios/** - store platform specific native project files which includes build scripts, configuration files and are handled by Expo and React Native.
- **node_modules/** - contains all installed project dependencies and libraries from npm.
- **src/** is the main source directory containing the application code and logic.
    - **components/** - reusable UI elements.
    - **navigation/** - routing, screen transitions and manages flow of application.
    - **screens/** - contains main user interface pages.
    - **services/** - handles logic that interacts with backend services.
- **App.js** - main entry point for React Native application.
- **index.js** - registers root component and links codebase to runtime.
- **app.json** - defines project configurations such as app name, icons etc.
- **package.json** - list all dependencies.

B. Backend



The backend of our application is built using FastAPI and PostgreSQL, organized into a clean, modular structure that separates logic, data handling, and infrastructure setup.

Each folder serves a specific role to make the backend easy to maintain, scale, and test.

- **main.py** – The entry point of the backend application that creates the FastAPI app, connects routes, and starts the server.
- **api/** - Contains all the FastAPI route definitions (endpoints) that handle incoming HTTP requests from the frontend and return responses.
- **services/** - Implements the main business logic of the application. It processes requests, interacts with repositories, and applies core rules or computations.
- **models/** - Defines pure domain models that represent real-world entities.
- **schemas/** - Stores Pydantic models that validate, structure, and format data for requests and responses between the API and frontend,
- **core/** - Handles essential backend configurations, including the PostgreSQL database setup, session management.
- **ports/** - Defines abstract interfaces (contracts) that describe what operations external systems must provide, such as saving or retrieving data.
- **adapters/** - Provides implementations of the interfaces defined in ports.
- **tests/** - Contains unit and integration tests for backend functions and database operations.
- **.gitignore** - Tells Git which files and folders to ignore, such as virtual environments, temporary files, etc.

This structure ensures that the FastAPI application remains organized, with a clear separation between business logic, API handling, and database interaction.