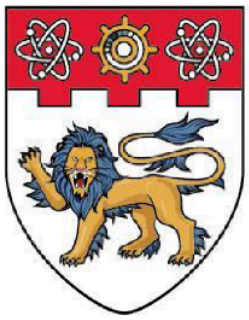


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2006 Software Engineering

Tutorial Group SCSD

Lab Deliverables

Group Members	Matric Number
Lim Li Ping Joey	U2321331C
Roger Kwek Zong Heng	U2320447G
Matz Chan	U2321258E
Ong Hong Xun	U2321248G
Wan Li Xin Yuan	U2321934B
Cheah Wei Jun	U2320561K

Table of Contents

1. Introduction

1.1. Purpose

Bussin Buses is a mobile application designed to provide a seamless and user-friendly platform for NTU students who reside far from campus. The primary purpose of Bussin Buses is to provide a faster and more convenient transportation option by providing a direct shuttle service that picks up students from NTU and drops them off at their destinations without making intermediate stops. The application provides a seamless and effective commuting experience by integrating real-time tracking, seat booking and optimised routes. The app will be a useful tool for all NTU students looking for a faster and more reliable transport option.

1.2. Product Scope

Bussin Buses helps connect NTU students with a faster and more efficient commuting option while offering a range of supportive features for hassle-free travel. The app utilises GPS technology to easily track bus locations in real-time and book seats in advance, ensuring a smooth and predictable journey. The app also uses API such as OpenStreet that allows personalised route recommendations and traffic updates, helping drivers to plan their route. In addition, the app offers functions to check bus schedules, manage bookings and receive notifications for traffic delays, making it a comprehensive resource for NTU students looking for a more convenient transportation option.

1.3. References

- GitHub: https://github.com/softwarelab3/2006-SCSD-C2.git

2. Overall Description

2.1. Product Perspective

Bussin Buses is a new, self-contained product designed to facilitate seat reservations for scheduled buses while providing real-time tracking, optimised route recommendations and traffic updates. This system is not a bus dispatching system but focuses on allowing Users to book specific seats on pre-scheduled buses.

The system interacts with:

A. Supabase - To handle User Registration, Login and Password Recovery.

2.2. Product Functions

Bussin Buses provides the following core functionalities:

A. User Account Management
    a. Register, Login, and Logout

B. Seat Booking and Cancellation
    a. Search for available buses
    b. Book and Cancel a seat on a scheduled bus
    c. View upcoming bookings

C. Real-time bus tracking
    a. Track live locations of buses
    b. View estimated arrival time

D. Route and Traffic Optimisation
    a. Provided optimised route for drivers based on traffic conditions
    b. Update estimated arrival time dynamically.

E. Driver Operations
    a. Manage start/end time of bus journey
    b. Manage bus schedule
    c. View commuters list for clarification
    d. Update any delay for traffic

2.3. User Classes and Characteristics

NTU Students

- Primarily use the app for transportation-related features such as bus tracking and seat booking.
- Authenticated with their login credentials to access the features.
- Need real-time information on bus locations and estimated arrival times to plan their commute.
- May prefer mobile-friendly, fast and easy-to-navigate interfaces as they are likely to use the app while on-the-go.

Bus Drivers

- Responsible for driving the bus on optimised routes.
- Require real-time communication tools to receive instructions about route changes or any potential incident.
- May need a simpler, clear interface to monitor student pickup and drop-off locations.

2.4. Operating Environment

The Bussin Buses app operates on smartphones, requiring an internet connection for accessing real-time bus data, location services and seat booking features. The app is compatible with <Android version 8.0 and above>. To ensure smooth functionality, users must have sufficient device storage, an active internet connection and location services enabled for accurate bus tracking and route optimization.

2.5. Design and Implementation Constraints

1. Bussin Buses requires a stable internet connection to access its functionalities as it depends on real-time data and location-based services for route planning and tracking.
2. The app utilises third-party APIs such as OpenStreet API for location services. Any changes or deprecation in these APIs may impact the app's performance or availability of certain features.
3. Bussin Buses must be installed on devices to function properly.

2.6. User Documentation

A demonstration of how the app works along with a guide to install and run the app is provided in the README.md file on the GitHub repository which contains the source

code for this app (see Section 1.3 References).

2.7. Assumptions

1. Bussin Buses relies on external APIs to provide relevant real-time data and services.
2. The data retrieved from external APIs is assumed to be accurate, up-to-date and reliable.
3. In cases where the API services become unavailable, the app may not meet some requirements specified in this document.
4. Bussin Buses assumes that users have access to a smartphone with an active internet connection which is essential for the app's functionality.
5. It is assumed that users have a basic understanding of how to navigate mobile applications and can use app's features without requiring extensive technical assistance.

3. Functional Requirements

1. The booking system shall allow Users to manage their individual accounts through Supabase Authentication.
    1.1. Users must consist of Commuters and Drivers.
    1.2. The system shall allow Users to register their account in the server.
    1.3. The system shall allow Users to login into their account.
    1.4. The system shall allow Users to logout their account.

2. The booking system shall allow Commuters to manage seat bookings on buses.
    2.1. The booking system shall be able to search for bus schedules
        2.1.1. The system shall display available buses based on date, time, pre-assigned pickup, and pre-assigned drop-off locations.
    2.2. The booking system shall allow Commuters to book a seat for an available bus.
        2.2.1. The system shall display a "Book Seat" button for available buses.
        2.2.2. The system will reserve a seat for the Commuter.
            2.2.2.1. If there are clashes of requests for the bus seat, the first request shall be processed and the rest shall be rejected.
    2.3. The booking system shall allow Commuters to cancel their booking
        2.3.1. The system shall allow commuters to cancel their booking up to 30 minutes before departure.
        2.3.2. Upon cancellation, the system shall update seat availability in real-time.
    2.4. Commuters shall be able to view their upcoming bookings.
        2.4.1. The system shall display all confirmed bookings with bus details, seat number, and timings.

3. The system shall allow Commuters to see and track their bus bookings.
    3.1. The system shall display an individual bus schedule which is managed by Bus Driver.
        3.1.1. Commuters shall be able to view their desired bus timings and locations shown in the bus schedule.
        3.1.2. Commuters shall be able to search for their desired bus schedule.
    3.2. The system shall display the live information of the booked bus.
        3.2.1. If the Commuters checked into the bus, they would be able to get live updates about their current bus status
    3.3. The system shall estimate and display estimated arrival time.
    3.4. The system shall allow students to check into the bus upon arrival.

4. The system shall allow Bus Drivers to manage bus operations.
    4.1 Bus Drivers shall be able to view their assigned schedules.
        4.11 The system shall display a list of upcoming shifts.
        4.12 Each shift shall include preassigned pickup and drop-off locations.

4.2 Bus Drivers shall be able to manage bus schedules.
- 4.21 Drivers shall be able to start and end their journey.
- 4.22 Drivers shall update their real-time location throughout the journey.

4.3 Drivers shall be able to view their Commuter list.
- 4.31 The system shall display the list of Commuters who booked a seat.
- 4.32 Drivers shall confirm attendance when commuters board the bus.

4.4 The system shall display the Bus Driver's schedule and shift information.
- 4.41 The system shall display a list of timings of all the Bus Driver's shifts
- 4.42 Each shift shall include information about the assigned pickup and drop-off locations

5. The system shall provide optimized routes and traffic updates to Bus Drivers.

5.1 The system shall recommend the most efficient route for a given trip.
- 5.11 The system shall optimize the route based on real-time traffic data using APIs (e.g., HERE API).

5.2 The system shall provide real-time traffic updates to drivers.
- 5.21 Drivers shall receive estimated arrival times for drop-off points.
- 5.22 The system shall notify drivers of traffic incidents or delays.

5.3 The system shall allow drivers to report incidents or delays.
- 5.31 Drivers shall be able to send delay notifications to the system.
- 5.32 The system shall update arrival estimates accordingly.

6. The system shall allow Supabase Authentication to manage User access.

6.1 The system shall validate User login credentials.

6.2 The system shall handle logout requests.

7. The system shall assist with bus routing.

7.1 The system shall analyze traffic conditions and recommend optimal routes.

7.2 The system shall update estimated arrival times dynamically.

8. The system shall provide real-time traffic updates.

8.1 The system shall track live traffic conditions and notify drivers.

8.2 The system shall receive real-time traffic updates to recommend a route.

4. Other Non-functional Requirements

1. Usability
    1.1. The booking system should be clear and organised for users to easily navigate around (User shall be able to book a bus seating within 5 clicks .
    1.2. The UI elements (buttons, icons, colours, fonts) should be consistent and easy to read and follow
    1.3. The booking system should personalize the user experience by remembering their preferences and choices

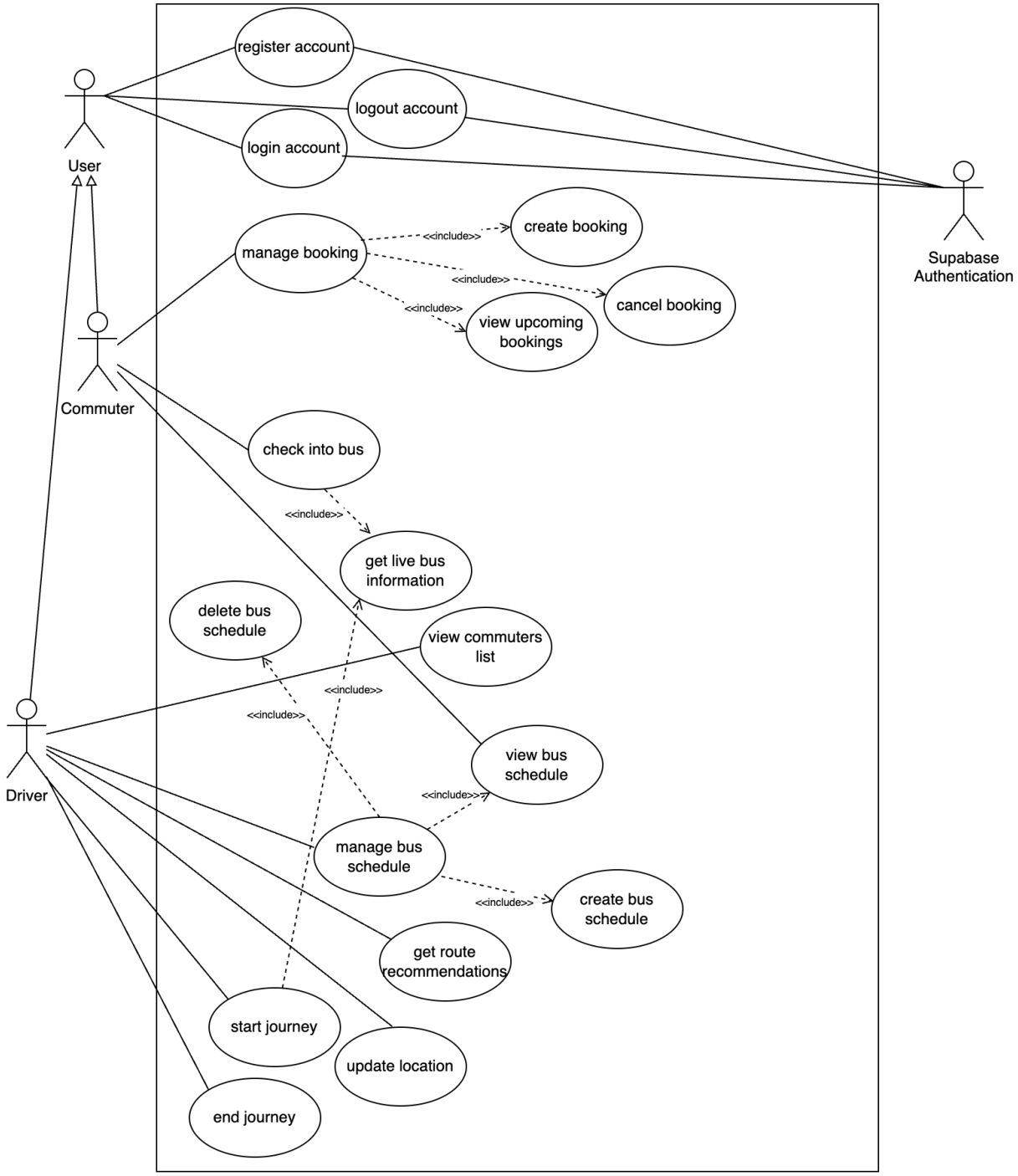
2. Reliability
    2.1. Ensure that the real-time traffic data is accurate to provide real-time travel estimation. (System shall be able to retrieve updated traffic information 3 seconds after being queried.)
    2.2. User informations should be safe and secure (password, credit card information, etc)
    2.3. The system should have minimal downtime unless notified

3. Performance
    3.1. The booking system should be able to handle multiple users and bookings without much significant performance degradation. (Information will be displayed within 3 seconds after user queried)
    3.2. The booking system needs to be able to handle multiple bookings while keeping real time update of the seats availability
    3.3. The GPS system needs to be accurate in tracking the bus locations to notify the users when the bus is arriving

4. Supportability

    4.1. The code need to be well documented and easy to maintain
    4.2. Documentation of the booking system should be done

5. Use Case Model



6. Use Case Descriptions

6.1 Account Management

Use Case ID:	1.1		
Use Case Name:	Register Account		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	9 Apr 2025

Actor:	User, Supabase Authentication
Description:	The “Register Account” use case is used to issue a user an identity so that we know their role in the system and we can attach a unique identifier to them to store their specific data.
Preconditions:	1. The system must be operational 2. Supabase Authentication must be operational 3. Users must have access to the client
Postconditions:	1. Supabase Authentication will have a new record for the newly registered user
Priority:	High
Frequency of Use:	Low
Flow of Events:	1. The user clicks on the “Register an account” button at the bottom to bring up the registration form 2. The user enters their name, email, password, confirm password, and user type 3. The user clicks on the “Register” button to confirm their registration 4. The user will be redirected to the respective homepage once successfully registered
Alternative Flows:	1.2.AF.1 Password and Confirm Password fields do not match 1. User clicks on “Register” button 2. Client verifies if passwords match 3. Client feedbacks to user that passwords do not match 1.2.AF.2. Password is not between 6 to 20 characters 1. User clicks on “Register” button 2. Client verified password not between 6 to 20 characters 3. Client feedbacks to user that password is not long enough 1.2.AF.3 Email address has already been registered 1. User clicks on “Register” button 2. Client verifies if email exist 3. Client feedbacks to user that email has been used 1.2.AF.4 Username must be between 3 to 6 characters 1. User clicks on “Register” button 2. Client verifies if username is between 3 to 6 characters

	3. Client feedbacks to user that username is not long enough
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	1.2		
Use Case Name:	Login Account		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	9 Apr 2025

Actor:	User, Supabase Authentication
Description:	The “Login Account” use case is to authenticate the user so that they can access their user-specific data in the system
Preconditions:	1. User has an existing account recorded in Supabase Authentication 2. The system must be operational 3. Supabase Authentication must be operational 4. Users must have access to the client
Postconditions:	1. User will be considered logged in on Supabase Authentication
Priority:	High
Frequency of Use:	Low
Flow of Events:	1. The user enters their email and password 2. The user clicks on the “Login” button 3. The user will be redirected to their respective homepage once successfully login
Alternative Flows:	
Exceptions:	1.3.EX.1 User not found in Supabase Authentication 1. The user click “Login” 2. The client verifies with Supabase Authentication 3. Supabase Authentication returns “User not found” 4. The client informs user of the exception
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	1.3		
Use Case Name:	Logout Account		
Created By:	Roger Kwek	Last Updated By:	Roger Kwek
Date Created:	11 Apr 2025	Date Last Updated:	11 Apr 2025

Actor:	User, Supabase Authentication
Description:	The “Logout Account” use case is to terminate the user’s active session and log them out of the application.
Preconditions:	1. User has an existing account recorded in Supabase Authentication and is currently logged in 2. The system must be operational 3. Supabase Authentication must be operational 4. Users must have access to the client
Postconditions:	1. The user’s session will be terminated 2. The user is logged out of Supabase Authentication 3. Client will be redirected to the login page
Priority:	High
Frequency of Use:	Low
Flow of Events:	1. The user clicks on the logout button 2. The client sends a signout request 3. Supabase Authentication terminates the user’s session 4. The client redirects the user to the login page
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

6.2 Booking Management

Use Case ID:	2.1		
Use Case Name:	Manage Booking		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	1 Feb 2025

Actor:	Commuter, Database
Description:	The “Manage Booking” use case is essential for allowing commuters to create, cancel, and view their upcoming bookings. It ensures that commuters can manage their trip plans and have access to relevant information about their upcoming trips.
Preconditions:	1. The user must be logged in as a commuter in the system 2. The system must have available bus schedules for booking 3. The Database must be operational
Postconditions:	1. Database is updated accordingly 2. The data present in the client is up-to-date on the modified details
Priority:	High
Frequency of Use:	Moderate to High, depending on commuter’s travel frequency
Flow of Events:	1. The commuter navigates to the booking management section in the client 2. The commuter initiates the desired action 3. The system executes the desired action
Alternative Flows:	
Exceptions:	
Includes:	Create Booking, Cancel Booking, View Upcoming Bookings
Special Requirements:	
Assumptions:	Bookings cannot be updated. They must be cancelled and re-created.
Notes and Issues:	

Use Case ID:	2.2		
Use Case Name:	Create Booking		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	9 Apr 2025

Actor:	Commuter, Database
Description:	The “Create Booking” use case is for commuters to book a seat on a bus during a scheduled timing
Preconditions:	1. The user must be logged in as a commuter in the system 2. The system must have available bus schedules for booking 3. The Database must be operational
Postconditions:	1. Database is updated accordingly 2. The data on the client displays the newly created booking
Priority:	High
Frequency of Use:	Moderate to High, depending on commuter’s travel frequency
Flow of Events:	1. The commuter navigates to “Home” bottom navigation 2. The commuter search for the schedule they want 3. The client will display the list of schedule based on the search 4. The commuter clicks on an available schedule that they want to book 5. The commuter chooses a seat to book on the bus 6. The commuter confirms their selection by clicking on the “Confirm Booking” button 7. The client feedbacks that the commuter’s booking is successful
Alternative Flows:	AF-S3 Commuters Booking the Same Seat at the Same Time (race condition) 1. The system confirms the first request 2. The system rejects the subsequent requests 3. The system sends an error message to the rejected commuters 4. The client displays the error message 5. The client reflects the booked seat on the interface
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	2.3		
Use Case Name:	Cancel Booking		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	9 Apr 2025

Actor:	Commuter, Database
Description:	The “Cancel Booking” use case is for commuters to cancel a booked seat on a bus for a scheduled timing
Preconditions:	1. The user must be logged in as a commuter in the system 2. The booking already exists 3. The Database must be operational
Postconditions:	3. Database is updated accordingly 4. The data on the client removes the booking from the interface
Priority:	High
Frequency of Use:	Low to Moderate
Flow of Events:	1. The commuter navigates to the “Ticket” bottom navigation tab 2. The commuter clicks the “Details” button on the booking that he wants to cancel 3. The commuter clicks the “Cancel Booking” button on the booking to be cancelled 4. The client displays a confirmation message to confirm the cancellation 5. The commuter clicks on “Yes, Cancel” to confirm cancellation 6. The booking is removed from the user interface
Alternative Flows:	AF-S2. Commuter Denies Booking Cancellation 1. The driver clicks on the “Cancel Booking” button on the booking he wants to delete 2. The client displays a confirmation message to confirm the cancellation 3. The driver clicks “No” 4. The system returns to the previous screen without making any changes
Exceptions:	2.3.EX.1 Commuter tries to cancel 30 minutes or less before departure 1. The commuter navigates to the booking management section in the client 2. The client clicks the “Cancel Booking” button on the booking to be cancelled 3. The system displays “Cancellation must be done at least 30 minutes before departure”
Includes:	
Special Requirements:	

Assumptions:	Commuter cannot cancel booking 30 minutes before the departure time
Notes and Issues:	

Use Case ID:	2.4		
Use Case Name:	View Upcoming Bookings		
Created By:	Lim Li Ping Joey	Last Updated By:	Ong Hong Xun
Date Created:	31 Jan 2025	Date Last Updated:	7 Feb 2025

Actor:	Commuter, Database
Description:	The “View Upcoming Bookings” use case is for commuters to check which seats they booked and for which scheduled bus timing
Preconditions:	1. The user must be logged in as a commuter in the system 2. The booking(s) already exist 3. The Database must be operational
Postconditions:	
Priority:	High
Frequency of Use:	High
Flow of Events:	1. The commuter navigates to the “Ticket” page 2a. If at least one booking exists: The table of bookings will be displayed in the booking management section 2b. If no booking exists: The booking management section will display a “No booking(s) found” message
Alternative Flows:	AF-S2a. Commuter has upcoming booking(s) today 1. Commuter opens the app 2. The client displays the home page 3. The client displays today’s upcoming booking(s)
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

6.3 Bus Schedule Management

Use Case ID:	3.1		
Use Case Name:	Manage Bus Schedule		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	3 Feb 2025

Actor:	Driver, Database
Description:	The “Manage Bus Schedule” use case is essential for allowing drivers to create, delete, and view their bus schedule. It ensures that drivers can manage their schedules and have access to relevant information about their upcoming shifts.
Preconditions:	1. The user must be logged in as a driver in the system 2. The Database must be operational
Postconditions:	1. Database is updated accordingly 2. The data present in the client is up-to-date on the modified details
Priority:	High
Frequency of Use:	High
Flow of Events:	1. The driver navigates to the bus schedule management section in the client 2. The driver initiates the desired action 3. The system executes the desired action
Alternative Flows:	
Exceptions:	
Includes:	View Bus Schedule, Create Bus Schedule, Delete Bus Schedule, Search for Bus Schedule
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	3.2		
Use Case Name:	View Bus Schedule		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	3 Feb 2025

Actor:	Driver, Commuter, Database
Description:	The “View Bus Schedule” use case is for drivers and commuters to view all scheduled bus rides available in the system. This is important for the driver not to clash with other drivers, and important for commuters to know when they can book a seat.
Preconditions:	1. The user must be logged in either as driver or commuter in the system 2. The database must be operational
Postconditions:	
Priority:	High
Frequency of Use:	High
Flow of Events:	**Commuter** 1. The commuter opens the client 2. The commuter will search for the bus schedule that they want 3. The client will display the appropriate bus schedule based on the required search 4. Alternatively, the commuter can click on “Show All Schedules” to view the whole list of schedules available **Driver** 1. The driver navigates to “Account” bottom navigation tab 2. The driver selects on “All Upcoming Trips” button 3. The system displays all the bus schedules with the schedule information
Alternative Flows:	AF-S3. No available bus schedules 1. The driver clicks on the “All Upcoming Trips” button. 2. The system displays “No upcoming trips available”.
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	3.3		
Use Case Name:	Create Bus Schedule		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	3 Feb 2025

Actor:	Driver, Database
Description:	The “Create Bus Schedule” use case is for drivers to indicate to commuters when they are available. It also provides commuters details about the seats available, pickup point, drop off point, and departure time of the scheduled ride.
Preconditions:	1. The user must be logged in as driver in the system 2. The database must be operational
Postconditions:	1. Database is updated accordingly 2. The data on the client displays the newly created bus schedule
Priority:	High
Frequency of Use:	High
Flow of Events:	1. The driver navigates to the bus schedules screen 2. The driver enters information such as the pickup point, drop-off point, departure date and time 3. The driver clicks the “Submit Journey” button 4. The system displays “Journey added successfully” 5. The client reflects the newly created bus schedule in the user interface with status “CONFIRMED”
Alternative Flows:	AF-S3a. Driver does not fill in all fields. 1. The driver leaves some of the fields empty. 2. The driver clicks the “Submit Journey” button. 3. The system displays “Please fill in all fields”. AF-S3b. Driver chooses the same pickup and destination points. 1. The driver selects identical pickup and destination points. 2. The driver clicks the “Submit Journey” button. 3. The system displays “Pickup and destination points have to be different”. AF-S3c. Driver chooses the date time before time now. 1. The driver selects the date time that is before time now. 2. The driver clicks the “Submit Journey” button. 3. The system displays “Schedule must be in the future”. AF-S3d. Driver has other schedules with less than 5 hours apart on the same day. 1. The driver selects the date time that has less than 5 hours difference with other confirmed schedules on the same day. 2. The driver clicks the “Submit Journey” button. 3. The system displays “You must have at least 5 hours between trips”.
Exceptions:	

Includes:	
Special Requirements:	All fields are required
Assumptions:	The layout of seats will be options provided by the system, not for the driver to manually upload
Notes and Issues:	

Use Case ID:	3.4		
Use Case Name:	Delete Bus Schedule		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	3 Apr 2025

Actor:	Driver, Commuter, Database
Description:	The “Delete Bus Schedule” use case is for drivers to cancel their scheduled bus ride slot, in case of sudden unavailability. Due to this deletion, commuters that booked the scheduled ride will be affected.
Preconditions:	1. The target bus schedule must exist in the system 2. The user must be logged in as driver in the system 3. The database must be operational
Postconditions:	1. Bookings associated with that bus schedule is deleted from the client and Database 1. The bus schedule is deleted from the client and deletion status is updated on Database
Priority:	High
Frequency of Use:	Low
Flow of Events:	1. The driver navigates to the “Home” screen displaying only his upcoming bus schedules 2. The driver selects on the bus schedule he wants to delete 3. The driver clicks on the “Delete Trip” button at the bottom of the “Trip Details” page 4. The system prompts the driver to ask for confirmation 5. The driver clicks “Delete” 6. The system deletes all bookings related to the bus schedule 7. The system deletes the bus schedule and removes from the “Home” page 8. The system displays “Trip deleted successfully” 9. The status of the deleted trip is updated to “CANCELLED” in the “All Upcoming Trips” page
Alternative Flows:	AF-S2. Driver Denies Confirmation Prompt 1. The driver clicks on the “Delete” button on the schedule he wants to delete in the table of schedules 2. The system prompts the driver to ask for confirmation 3. The driver clicks “No” 4. The system returns to the previous screen without making any changes
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	3.5		
Use Case Name:	Search for Bus Schedule		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	1 Feb 2025

Actor:	Commuter, Database
Description:	The “Search Bus Schedule” use case is for commuters to quickly find the bus schedule they need.
Preconditions:	1. The user must be logged in as a commuter in the system 2. The Database must be operational
Postconditions:	
Priority:	Moderate
Frequency of Use:	High
Flow of Events:	1. The commuter filters the search by pickup point, drop-off point, departure date and time 2. The commuter clicks “Search” 3. The client displays bus schedules that match the filters
Alternative Flows:	AF-S2. No Exact Matching Schedule(s) 1. The commuter filters the search by pickup point, drop-off point, departure date and time 2. The commuter clicks “Search” 3. The client displays bus schedules that are similar to the filters
Exceptions:	3.5.EX.1 No Near Matches or Exact Matches Available 1. The commuter filters the search by pickup point, drop-off point, departure date and time 2. The commuter clicks “Search” 4. The client displays “No schedules available for this route and date” informing the commuter that there is lack of bus schedules matching their needs
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

6.4 Bus Management

Use Case ID:	4.1		
Use Case Name:	Get Live Bus Information		
Created By:	Lim Li Ping Joey	Last Updated By:	Roger Kwek
Date Created:	31 Jan 2025	Date Last Updated:	9 Apr 2025

Actor:	Commuter, Driver
Description:	The “Get Live Bus Information” use case is for commuters to track where they currently are during the trip
Preconditions:	1. The user must be logged in as a commuter in the system
Postconditions:	
Priority:	Low
Frequency of Use:	Moderate
Flow of Events:	1. The commuter checks into the bus 2. The driver starts the journey 3. The bus information is streamed from the driver 4. The commuter’s client displays all relevant information about the current bus ride
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	4.2		
Use Case Name:	Update Location		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	2 Feb 2025

Actor:	Driver
Description:	The “Update Location” use case is for drivers to update their locations for commuters to see the location of the bus relative to the pickup and drop-off points
Preconditions:	1. The user must be logged in as a commuter in the system 2. The driver must have started the journey
Postconditions:	
Priority:	Low
Frequency of Use:	Moderate
Flow of Events:	1. The driver starts the journey on the client 2. The client will retrieve the driver’s location 3. The client will update the location to the supabase 4. Steps 2 and 3 will repeat every minute until the driver ends the journey on the client
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	4.3		
Use Case Name:	Check into Bus		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	2 Feb 2025

Actor:	Commuter, Database
Description:	The “Check into bus” use case is for commuters to confirm that they are on the bus before departure. By checking into the bus, the commuter will be able to subscribe to the live bus information
Preconditions:	1. The user must be logged in as a commuter in the system 2. The Database must be operational 3. It must be 15 minutes or less before departure time
Postconditions:	1. The Database is updated accordingly
Priority:	Moderate
Frequency of Use:	Moderate
Flow of Events:	1. The commuter opens the client 2. The commuter navigates to “Ticket” bottom navigation tab 3. The commuter clicks on the “Details” button for the booking he wants to check into 4. The commuter clicks the “Check In” button 5. The client displays a confirmation message
Alternative Flows:	
Exceptions:	
Includes:	Get Live Bus Information
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	4.4		
Use Case Name:	Start Journey		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	2 Feb 2025

Actor:	Driver
Description:	The “Start Journey” use case is for drivers to broadcast to the commuters’ clients that it will be sending live bus information updates. It also starts the routing recommendation system.
Preconditions:	1. The user must be logged in as a driver in the system
Postconditions:	
Priority:	High
Frequency of Use:	Moderate
Flow of Events:	1. The driver opens the client 2. The driver clicks the “Navigate” button 3. The client navigates to the “Ride” page to displays the map 4. The driver clicks on “Start Journey” button to display the route recommendations 5. The status of the journey is updated to “IN PROGRESS” at the “Home” page
Alternative Flows:	
Exceptions:	
Includes:	Get Live Bus Information
Special Requirements:	
Assumptions:	The bus driver can start the journey from 5 minutes before departure time onwards
Notes and Issues:	

Use Case ID:	4.5		
Use Case Name:	End Journey		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	2 Feb 2025

Actor:	Driver
Description:	The “End Journey” use case is for drivers to end the trip and stop broadcasting the live bus information. It stops the route recommendations.
Preconditions:	1. The user must be logged in as a driver in the system
Postconditions:	
Priority:	High
Frequency of Use:	Moderate
Flow of Events:	1. The driver clicks on the “Stop Journey” button 2. The route recommendation screen is destroyed 3. The driver is brought to the Home Screen
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	The driver will be able to end the journey at any time, regardless if they are close in proximity to the dropoff point. This is to accommodate situations where a diversion of path is needed due to bus faults, etc.
Notes and Issues:	

Use Case ID:	4.6		
Use Case Name:	View Commuters List		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	2 Feb 2025

Actor:	Driver, Database
Description:	The “View Commuters List” allows the driver to see who booked the scheduled ride and who has checked in to gauge whether he can leave early or has to wait a bit longer.
Preconditions:	1. The user must be logged in as a driver in the system 2. The Database must be operational
Postconditions:	
Priority:	Low
Frequency of Use:	Low
Flow of Events:	1. The driver opens the client 2. The driver navigate to the “Home” bottom navigation tab 3. The driver selects the schedule that he wants to check 4. A list of commuters who booked the scheduled time slot, along with their checked in status represented as a orange star, will be displayed
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

6.5 Route Recommendations

Use Case ID:	5.1		
Use Case Name:	Get Route Recommendations		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	10 Feb 2025

Actor:	Driver
Description:	The “Get Route Recommendations” use case is for the driver to get a suggested route to get to the dropoff point, based on their current location
Preconditions:	1. The user must be logged in as a driver in the system 2. The user need to start a journey
Postconditions:	
Priority:	High
Frequency of Use:	Moderate
Flow of Events:	1. The driver starts the journey 2. The Route Recommendation screen shows up 3. The screen shows a recommended route to the driver
Alternative Flows:	AF-S1 The system detects a disruption and a new route was found 1. The system finds a new route 2. The new route is sent to the driver’s client 3. The route is updated on the driver’s client 4. The client informs the driver of the disruption and the reason for the rerouting
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	The driver only gets route recommendations once at the start, and if there are any subsequent disruptions to traffic
Notes and Issues:	

Use Case ID:	5.2		
Use Case Name:	Return Routing		
Created By:	Lim Li Ping Joey	Last Updated By:	Lim Li Ping Joey
Date Created:	31 Jan 2025	Date Last Updated:	10 Feb 2025

Actor:	Driver
Description:	The “Return Routing” use case is for the driver to re-navigate back to the map with the recommended route for the journey in the scenario where the user leaves the app.
Preconditions:	1. The user must be logged in as a driver in the system 2. The user need to start a journey
Postconditions:	
Priority:	Moderate
Frequency of Use:	Moderate
Flow of Events:	1. The driver returns back to the app 2. The driver clicks on “Navigate” button 3. The client will direct the user back to the ride page with recommended route
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	The driver only gets route recommendations once at the start, and if there are any subsequent disruptions to traffic
Notes and Issues:	

7. Data Dictionary

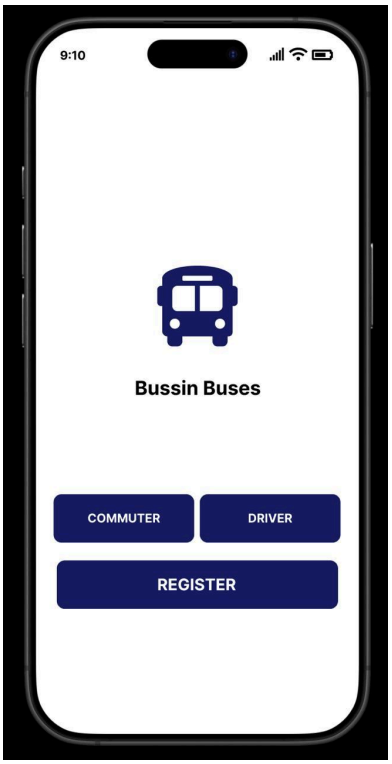
Term	Definition
Account	A registered user's personal profile associated with an application. It may include personal information, contact details, etc.
Application	A mobile application installed on users' smartphones that relies on an internet connection to fully utilize all of its intended features. The app may request certain system permissions to enable specific functionalities.
AuthenticationServer	A server responsible for managing user login credentials, verifying identity and handling password recovery requests.
Availability	Bus seats currently available for booking.
Booking	A confirmed reservation of the bus seat including details such as destination and time.
Bus	A transport vehicle that follows a fixed schedule and route to shuttle NTU students.
BusDriver	A designated individual responsible for operating the bus according to the assigned schedules and routes.
BusFare	The cost charged to users for booking a seat on a bus, based on factors such as route, distance, and demand.
BusSchedule	A predefined timetable specifying bus departure and arrival times along different routes.
Commuter	A NTU student using the application to book and track buses.

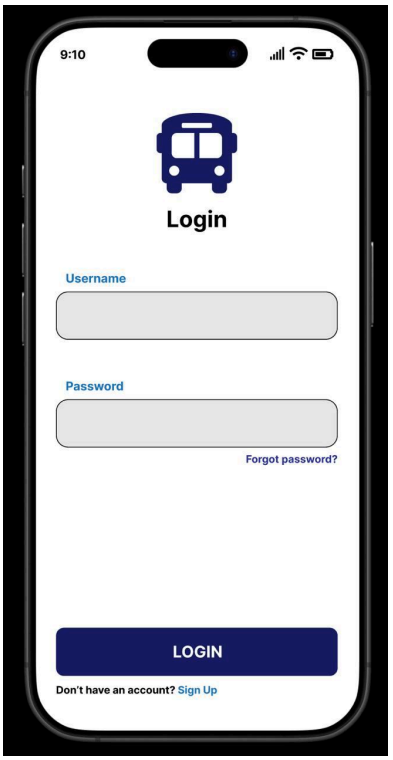
Credentials	Unique identifiers (e.g., username, email, password) used to verify a user's identity for login.
Database	Database refers to an organized collection of structured information, or data, typically stored electronically in a computer system.
Driver	The individual responsible for operating the bus along the assigned route and schedule
Destination	The final drop-off point for a bus journey as determined by a commuter’s booking.
Drop-off-Location	The designated place where a commuter exits the bus.
EstimatedArrivalTime	The estimated time for the bus to arrive at its drop off location.
GPS	GPS, or the Global Positioning System, is a global navigation satellite system that provides the bus location, speed and time synchronization.
LiveTracking	A feature that allows commuters to monitor the real-time location of their booked bus.
Notification	An alert sent to uses about bookings, updates, delays or cancellations
PickupPoint	The designated spot where a commuter boards the bus.
RealTimeUpdates	Instant information provided to commuters and drivers about bus location, estimated arrival time and delays.
Route	The bus's predetermined route, which includes destinations, departure locations, and traffic-adjusted directions.

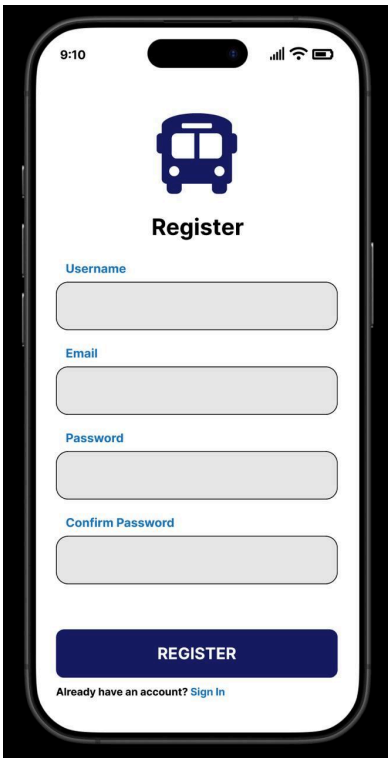
Status	Current status of the bus status such as “delayed”, “cancelled”, “available” and “unavailable”.
TrafficMonitoringSystem	Real-time traffic data used to optimize bus routes and provide accurate estimated arrival times.
User	Anyone who has created an account and uses the bus booking mobile app.

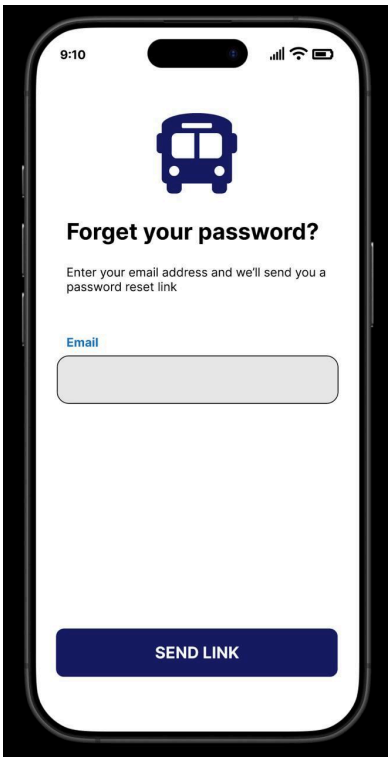
8. UI Mockup

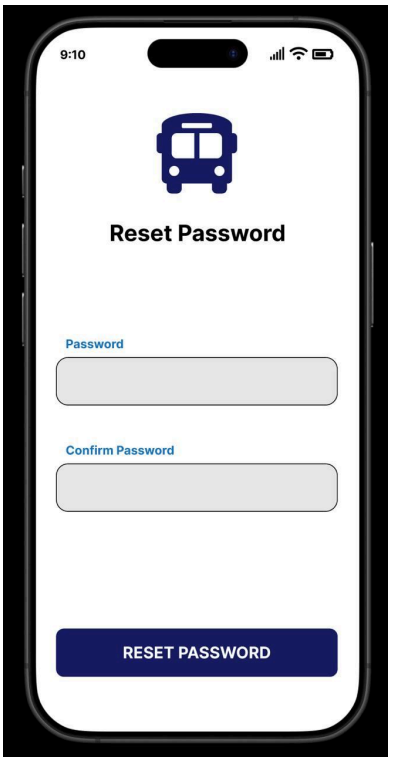
8.1 User Management



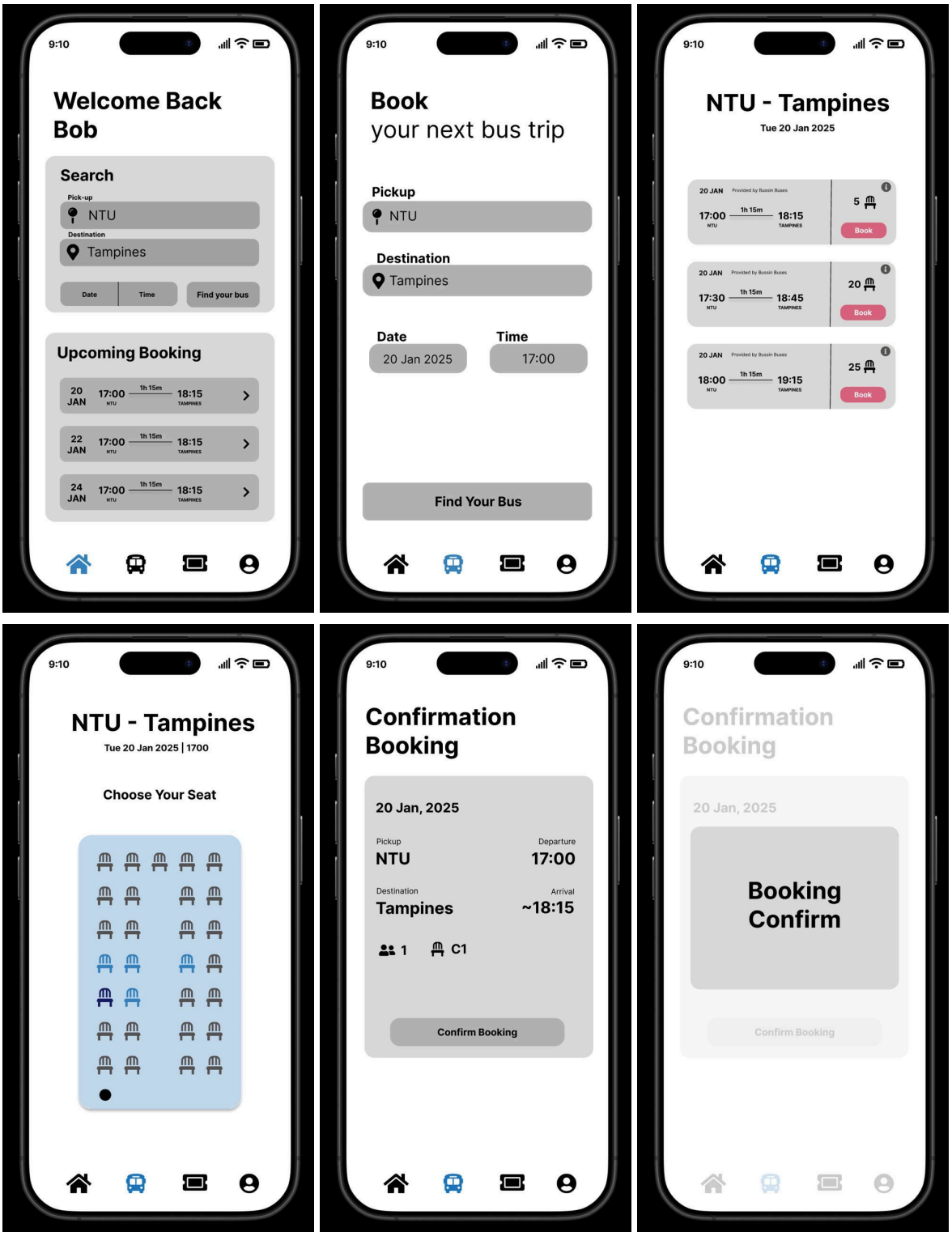




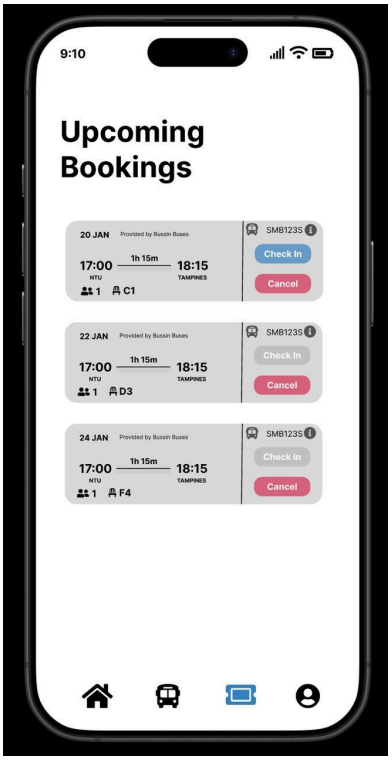


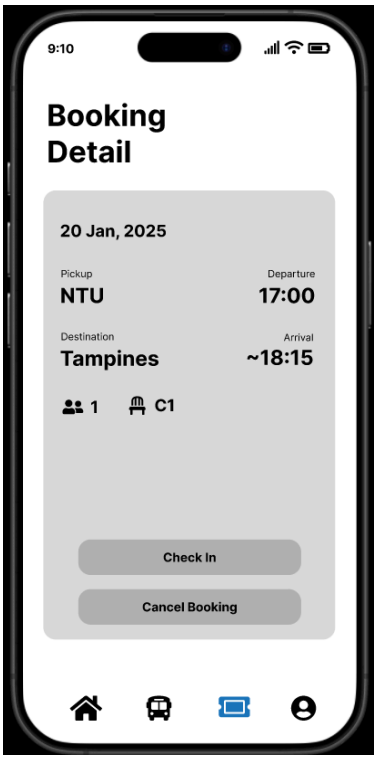


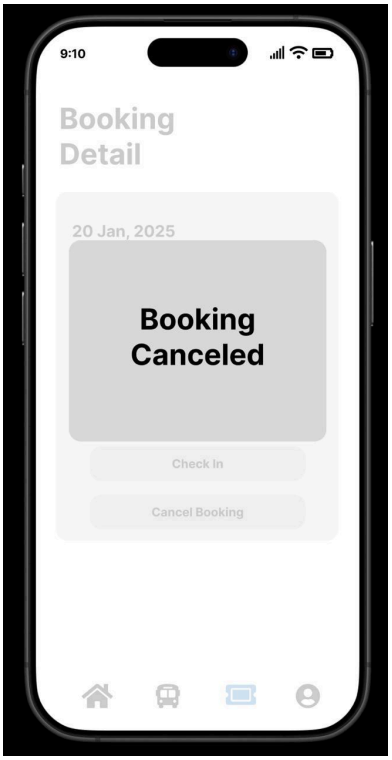
8.2 Commuters - Home Page + Booking of Bus Trip

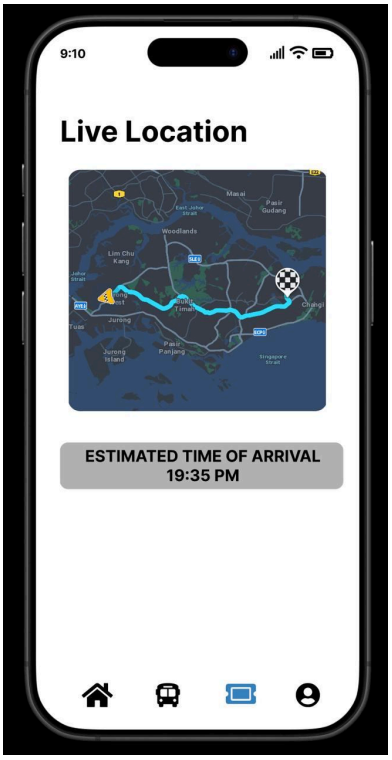


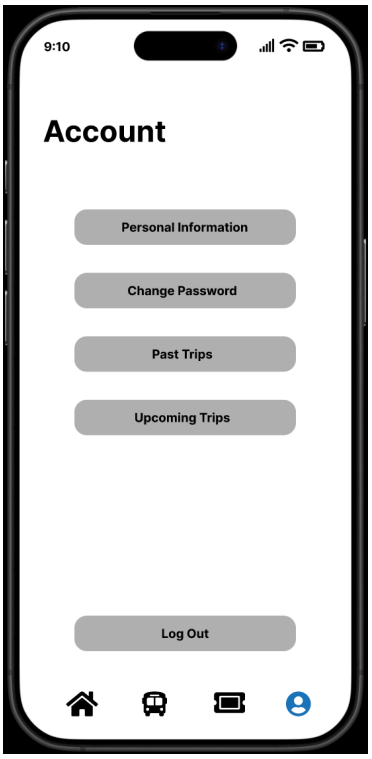
8.3 Commuter - Viewing Upcoming Booking + Live Tracking + Account Management





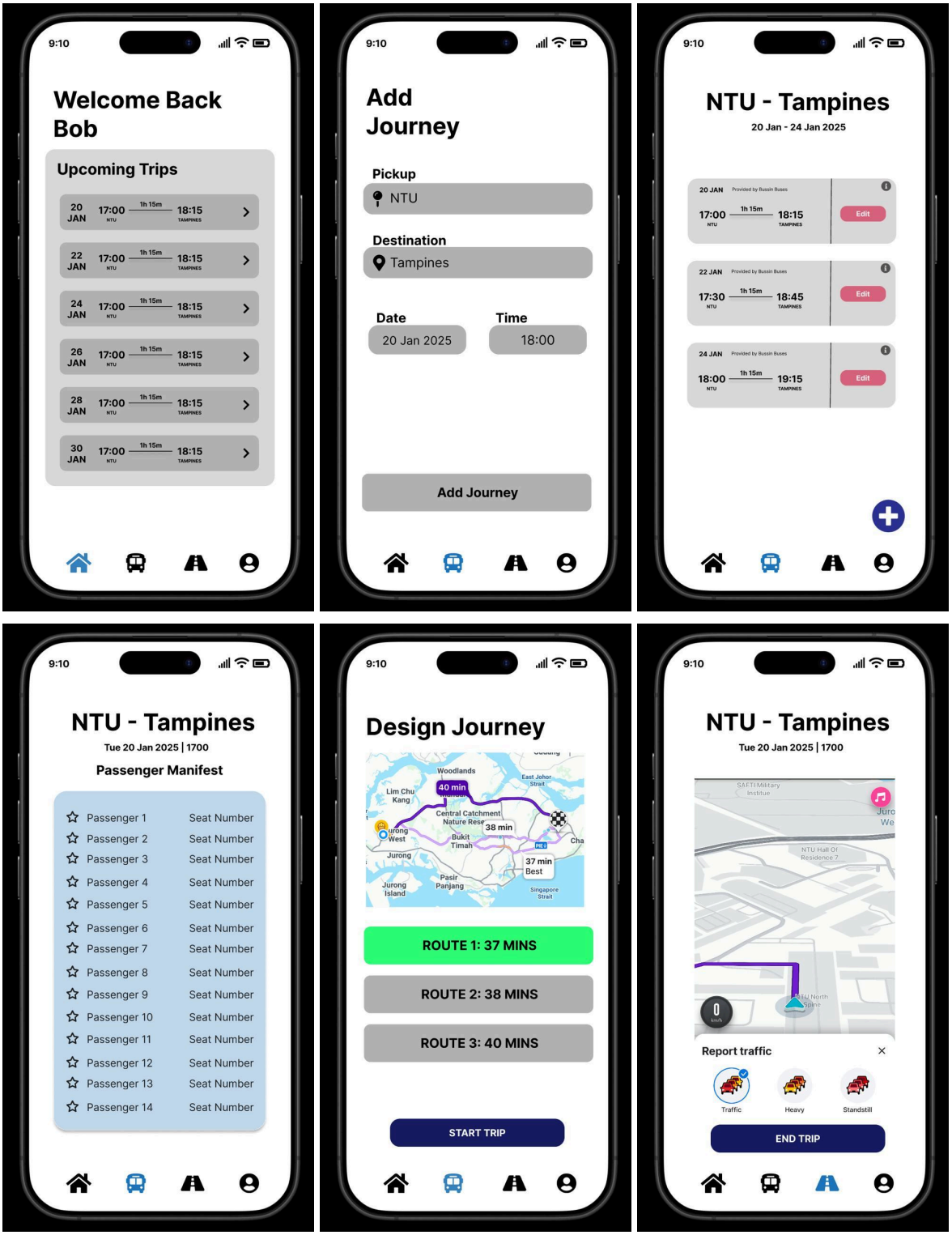






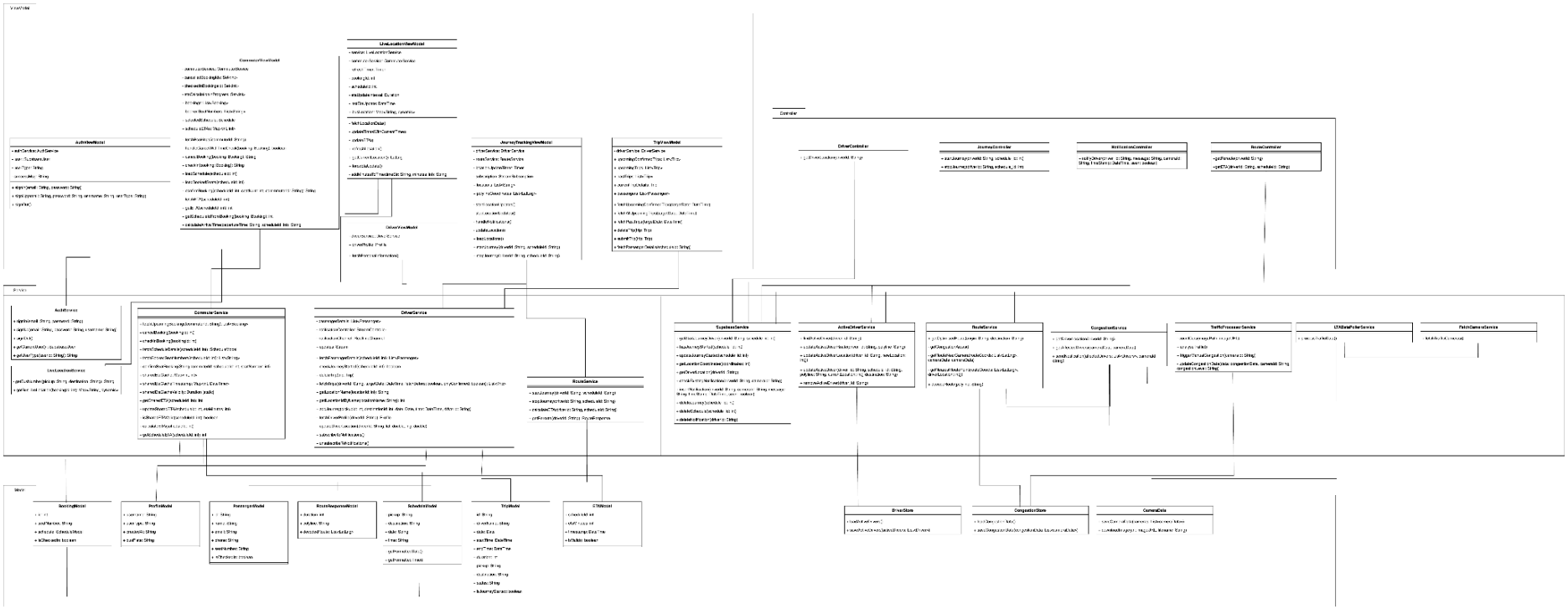
2

8.4 Driver - Homepage + Add Journey + Start & End Journey

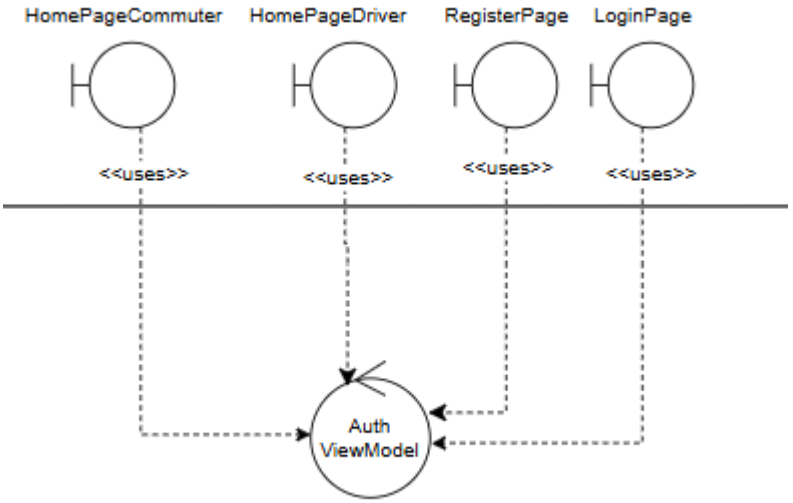


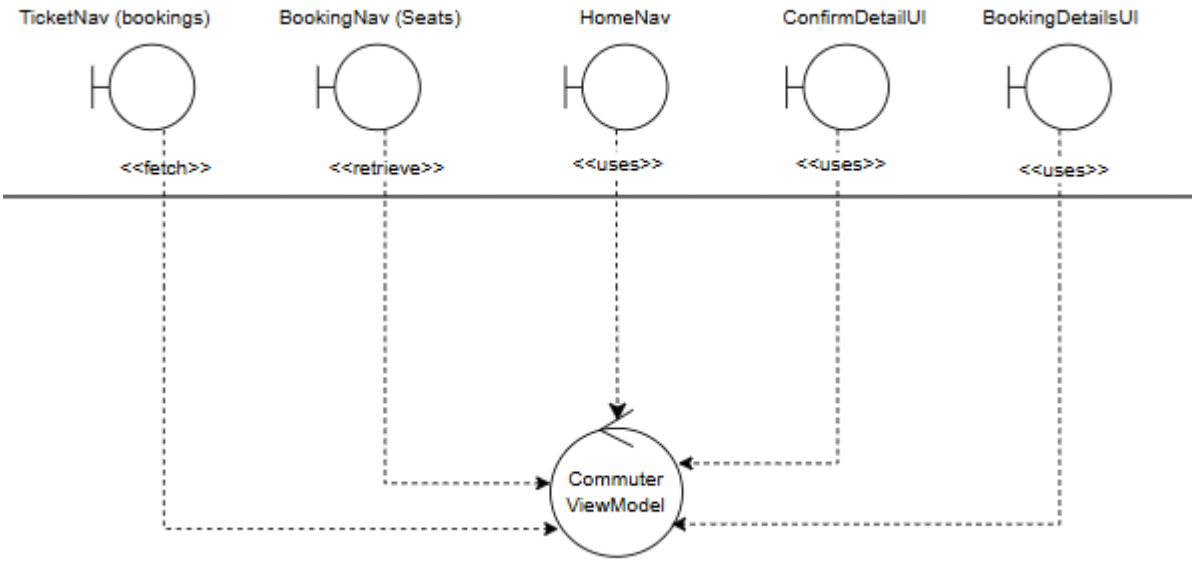
9. Class Diagram

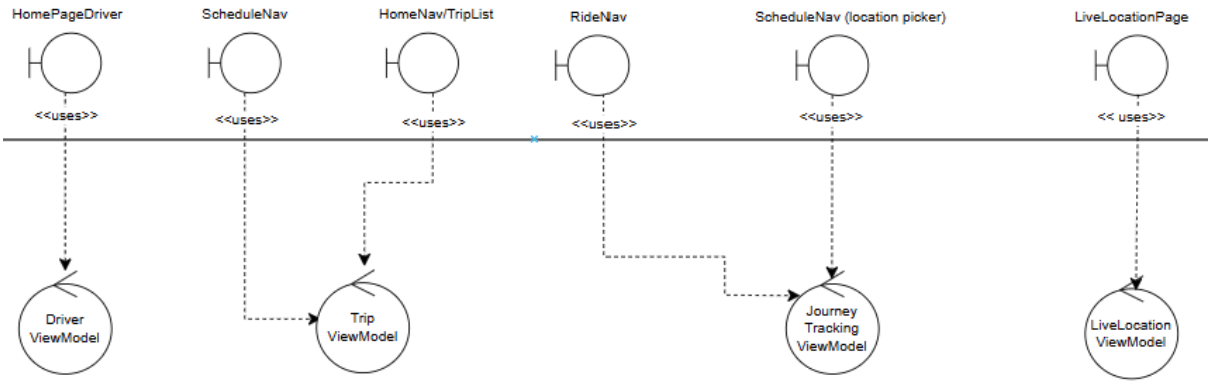
View in higher quality in our GitHub submission:
https://github.com/softwarelab3/2006-SCSD-C2/blob/main/Deliverables/Lab%205/Class%20Diagram.drawio.png



10. Key boundary classes and control classes

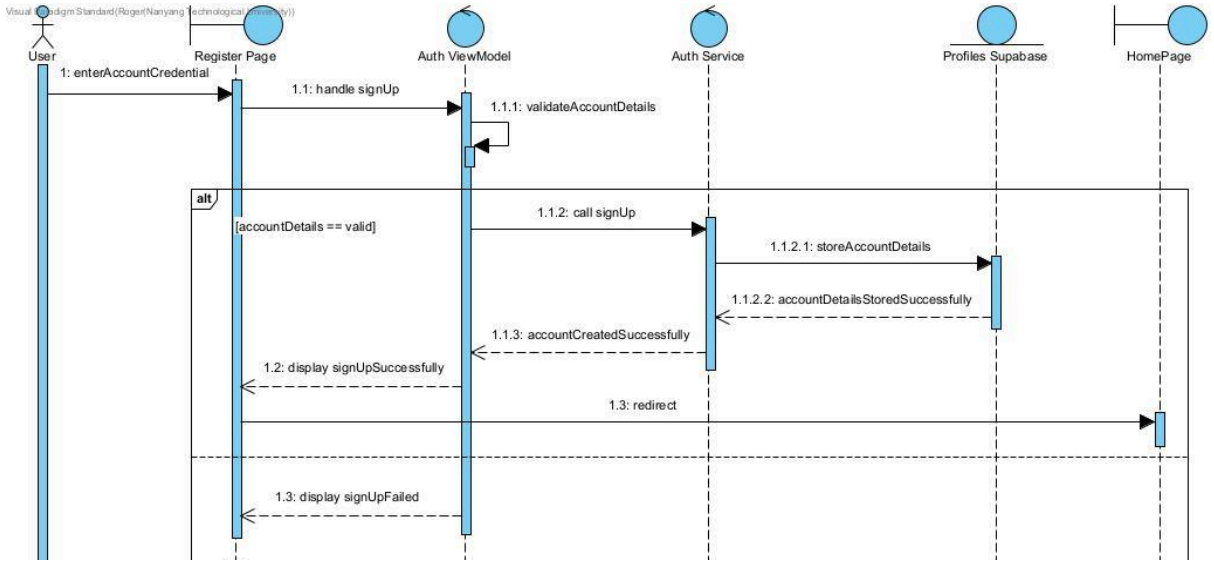




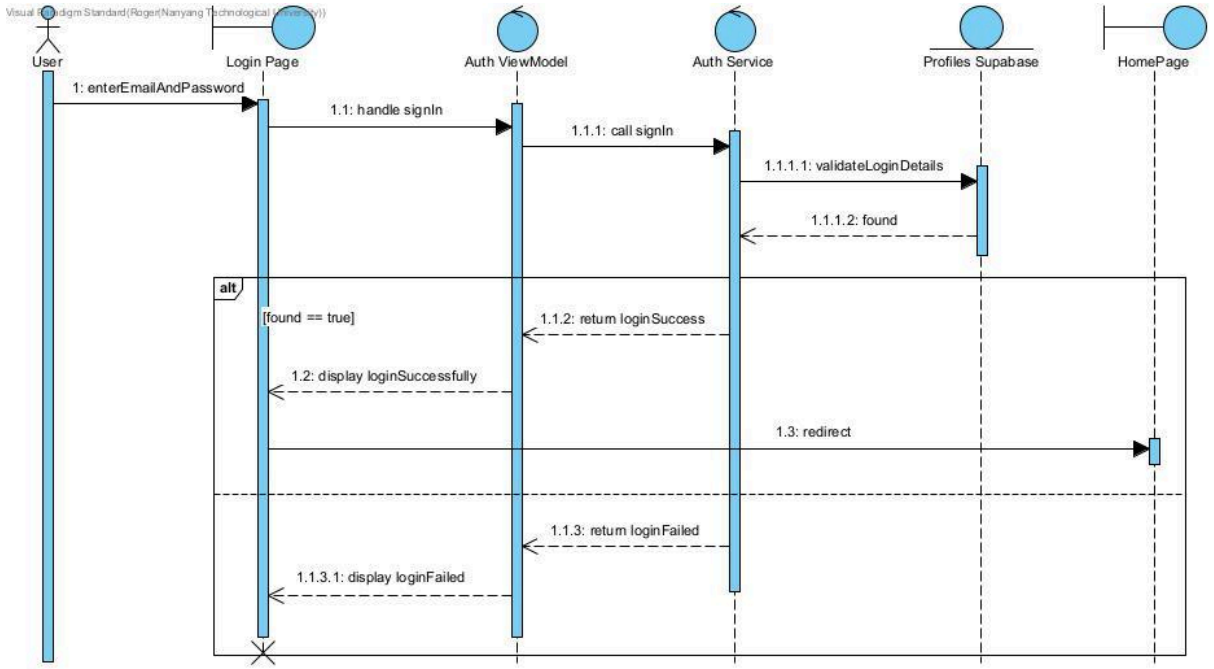


11. Sequence Diagrams

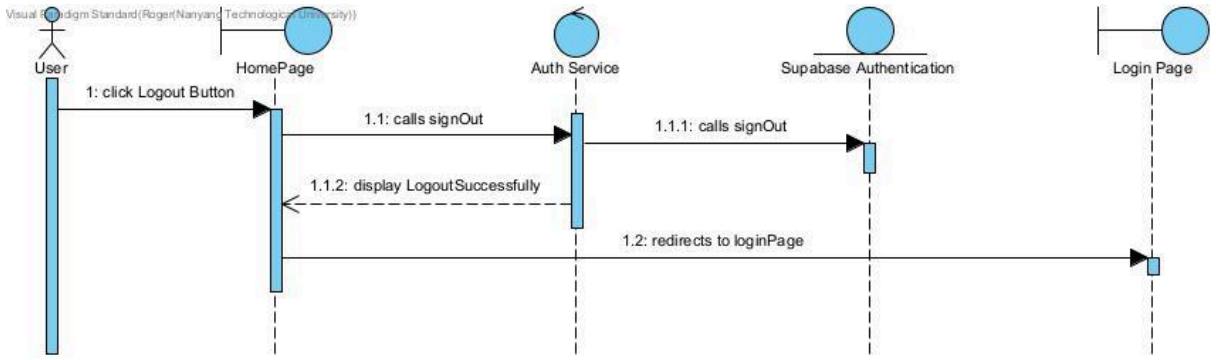
Use Case ID: 1.1
Use Case Name: Register Account



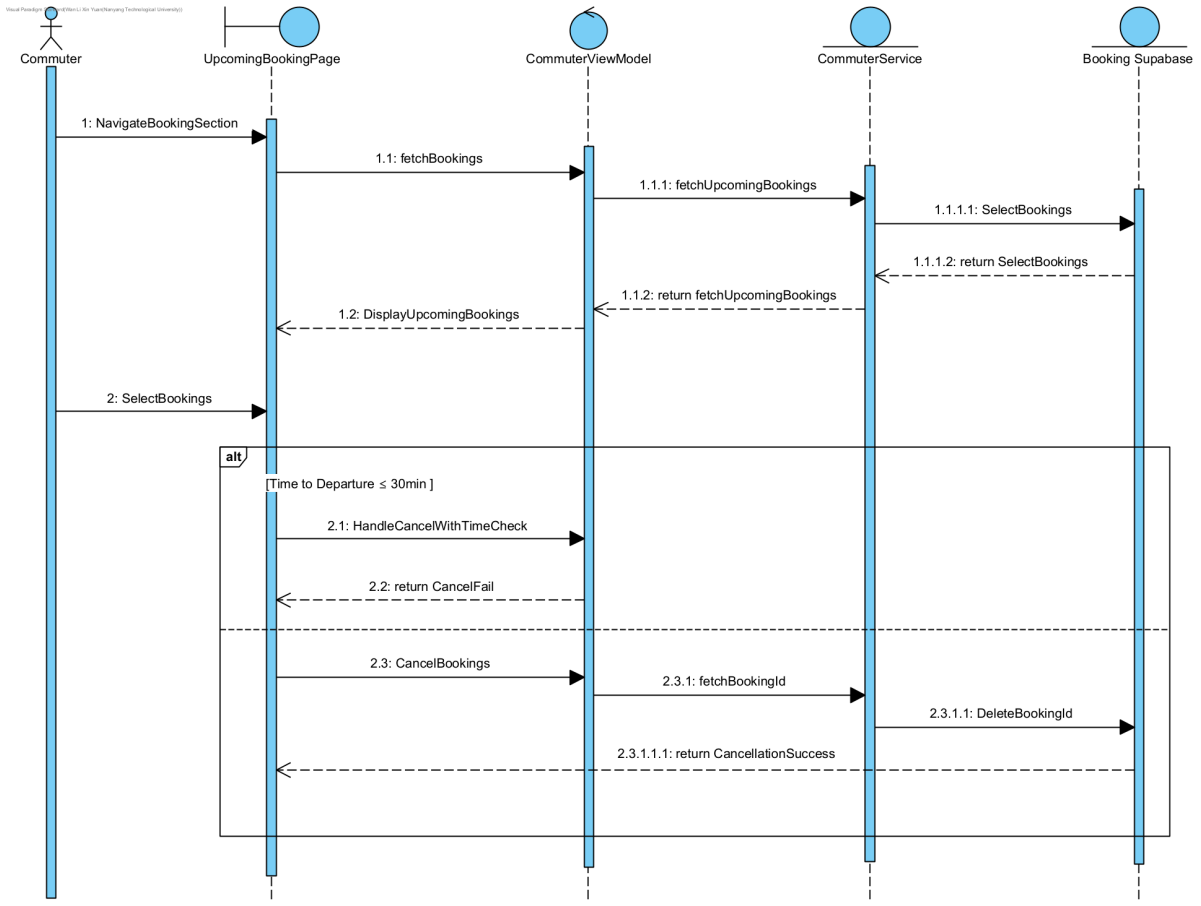
Use Case ID: 1.2
Use Case Name: Login Account



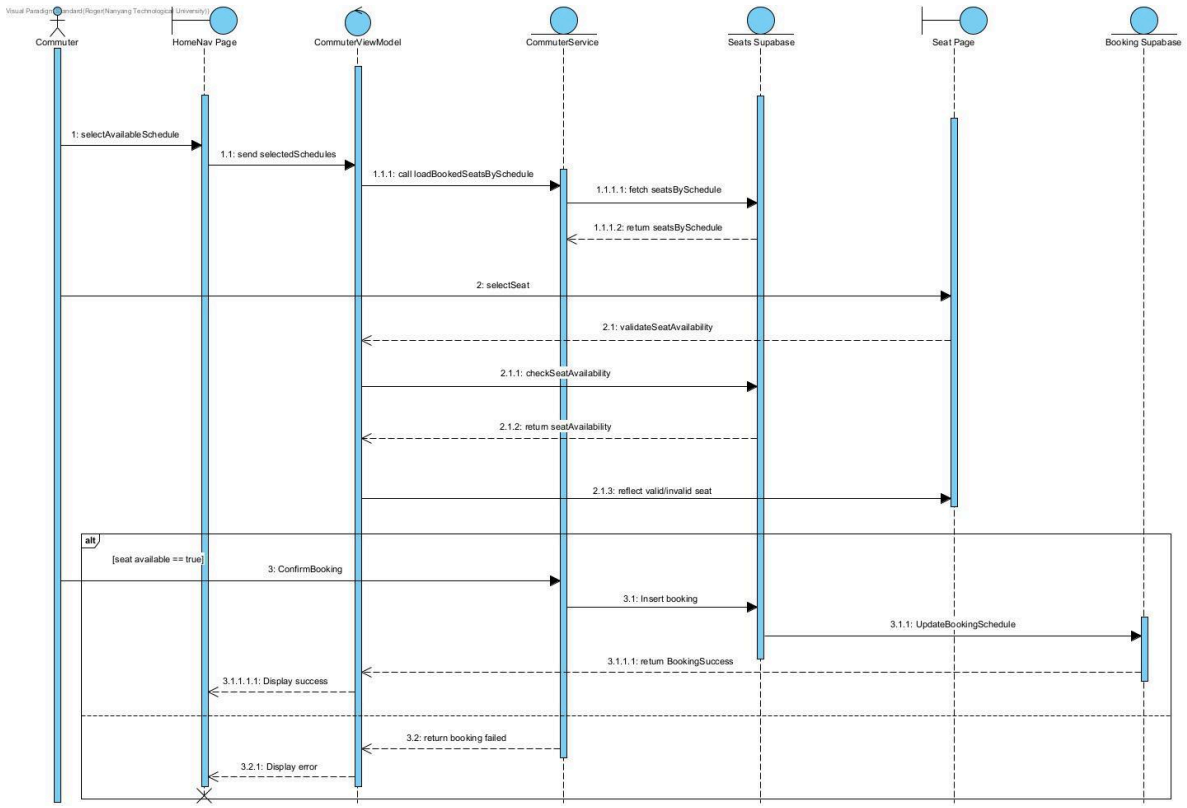
Use Case ID: 1.3
Use Case Name: Logout Account



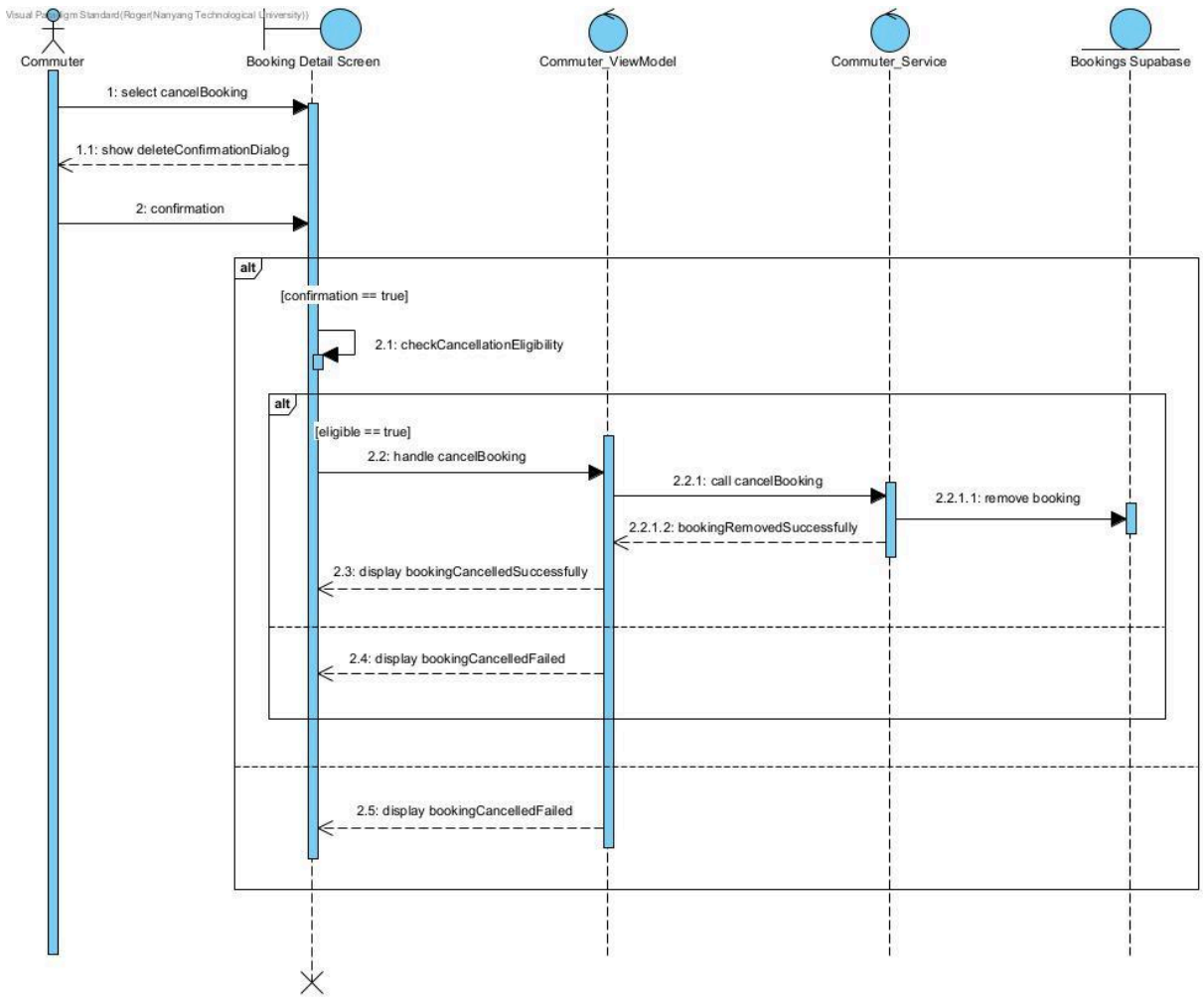
Use Case ID: 2.1
Use Case Name: Manage Booking



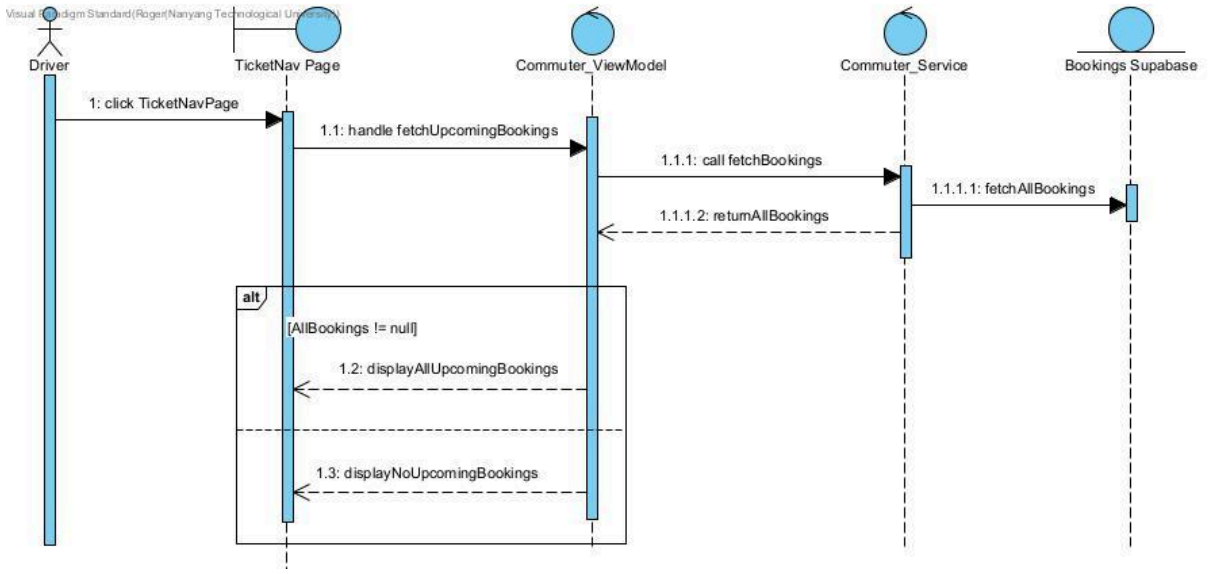
Use Case ID: 2.2
Use Case Name: Create Booking



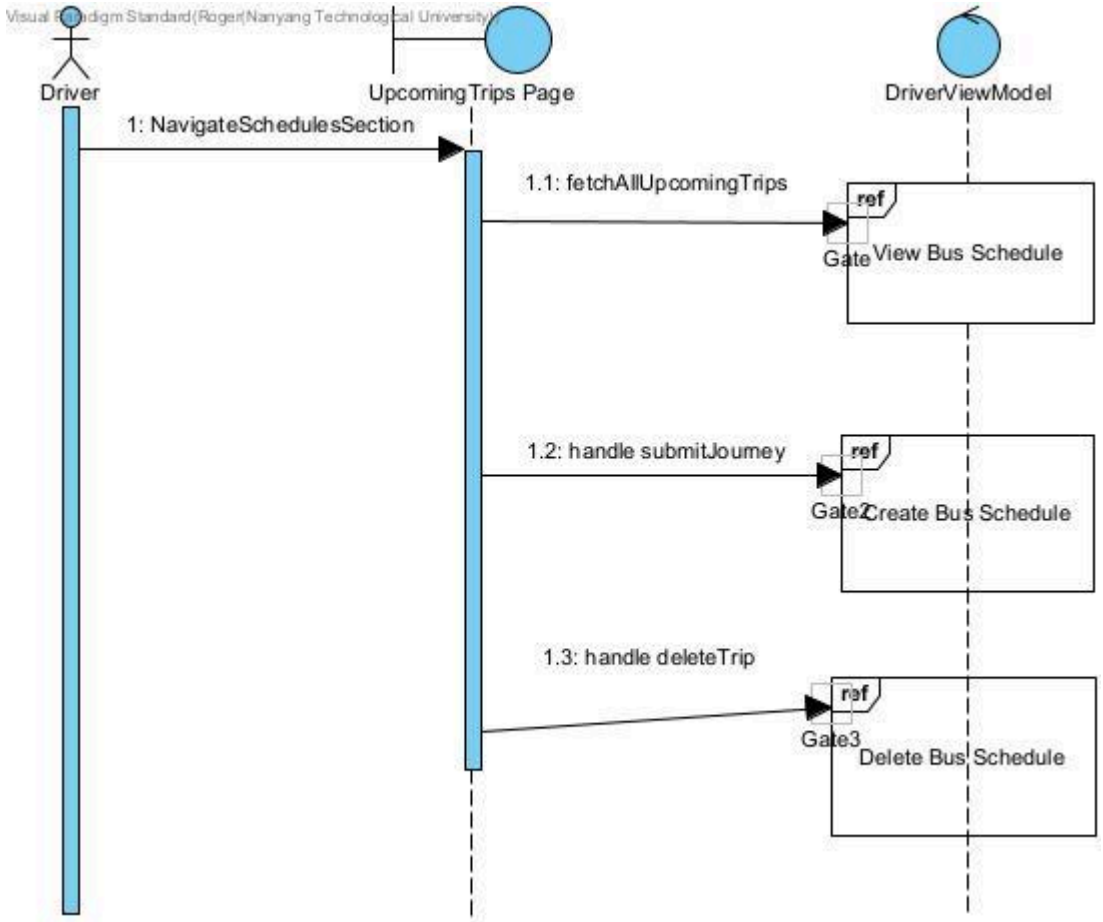
Use Case ID: 2.3
Use Case Name: Cancel Booking



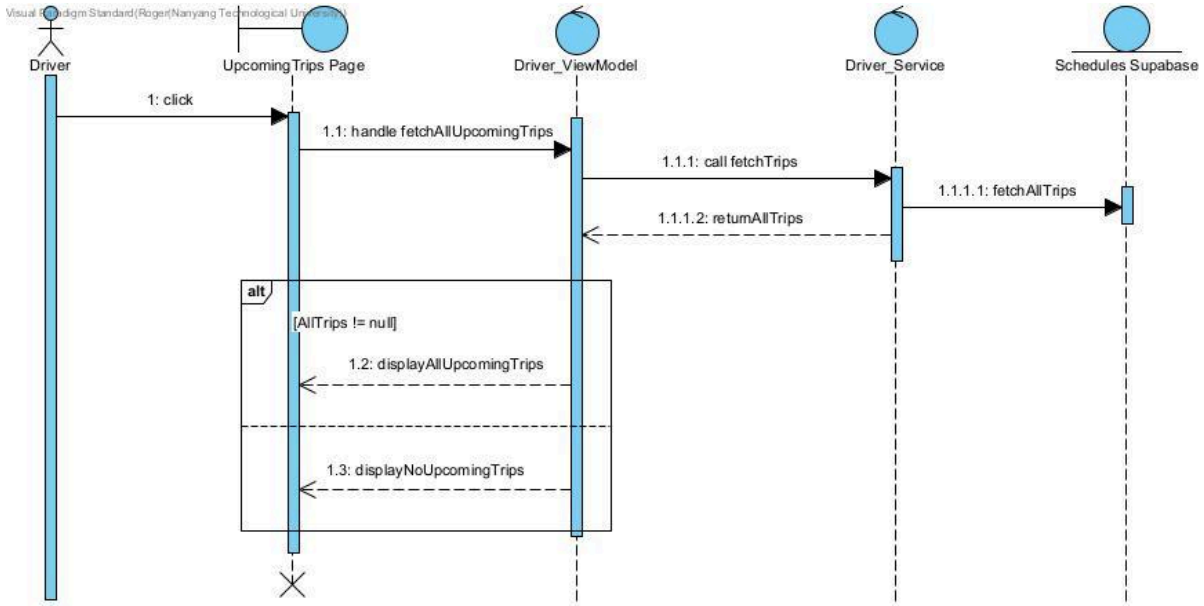
Use Case ID: 2.4
Use Case Name: View Upcoming Booking



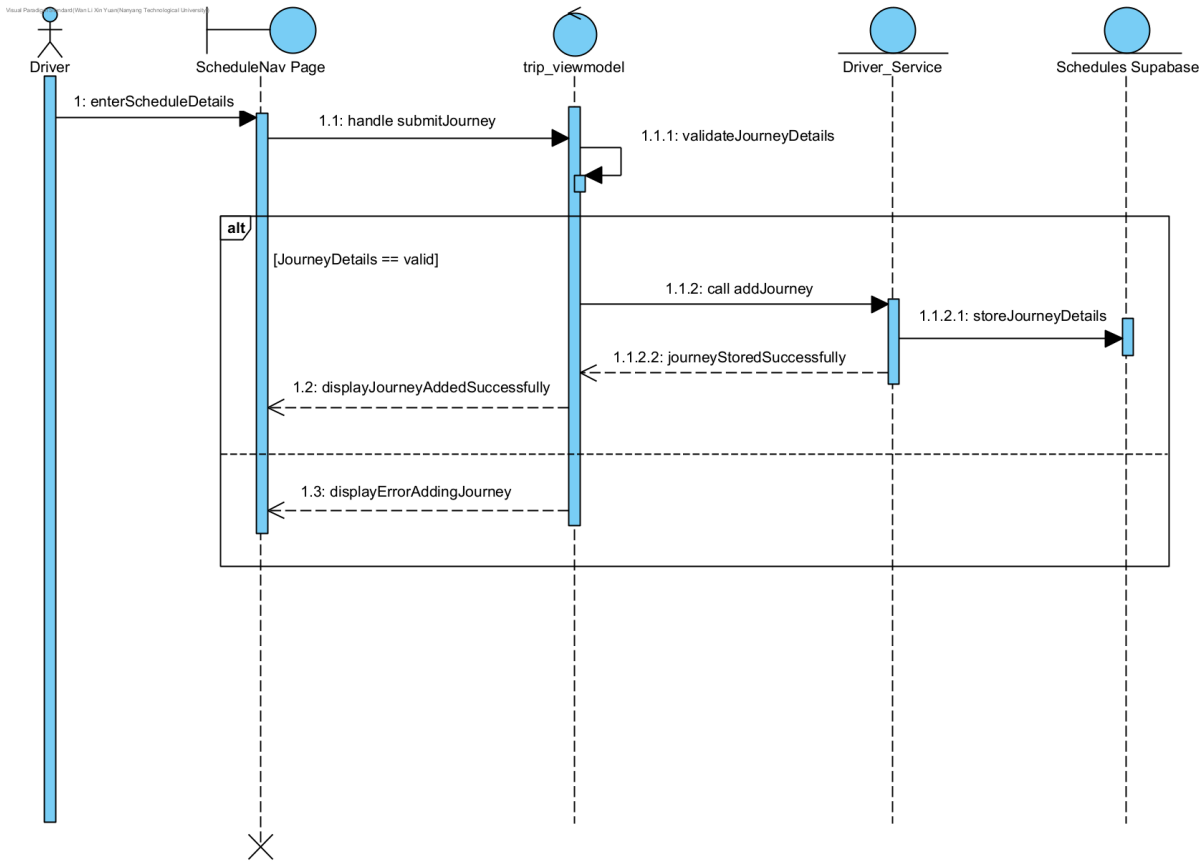
Use Case ID: 3.1
Use Case Name: Manage Bus Schedule



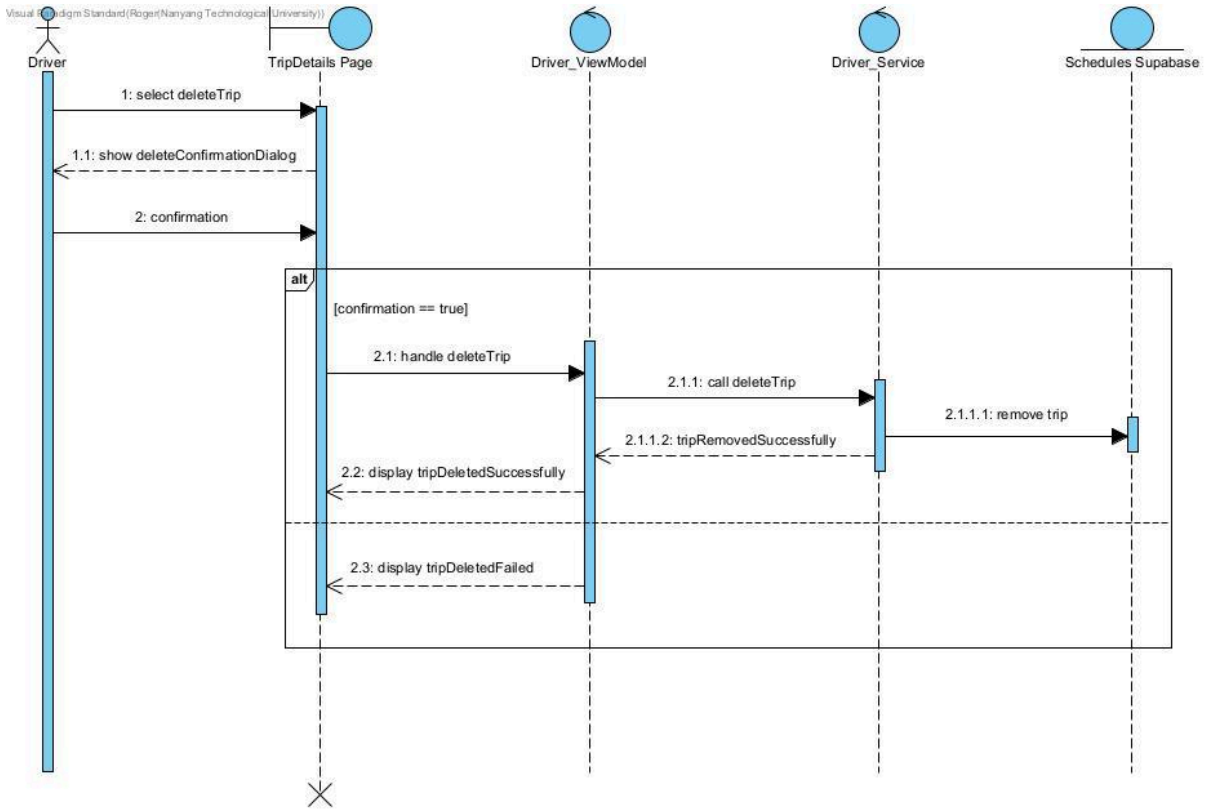
Use Case ID: 3.2
Use Case Name: View Bus Schedule



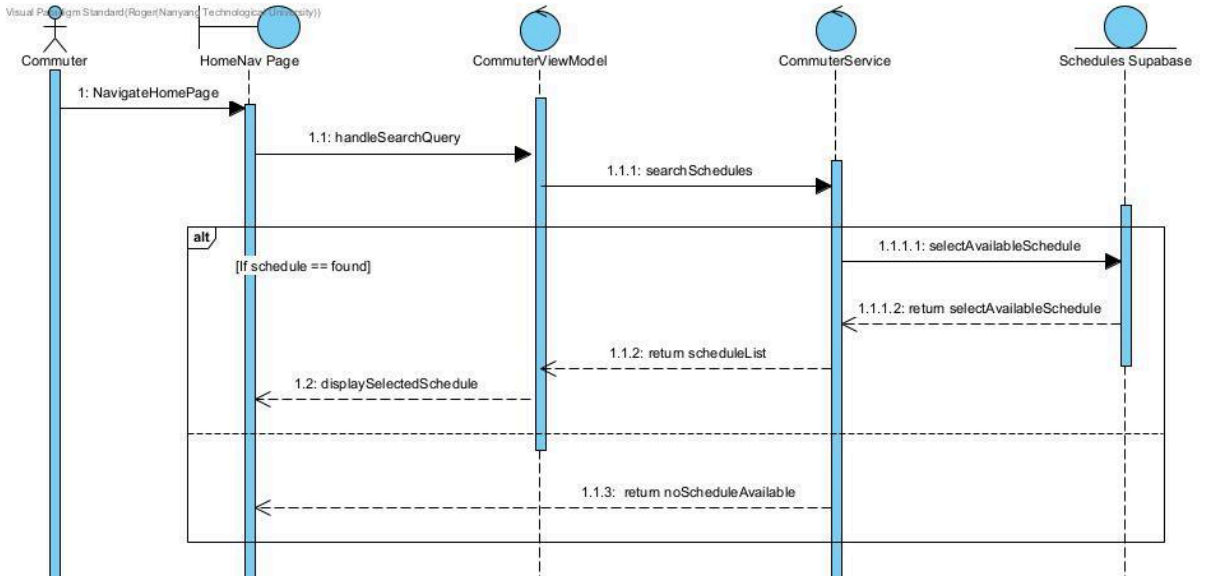
Use Case ID: 3.3
Use Case Name: Create Bus Schedule



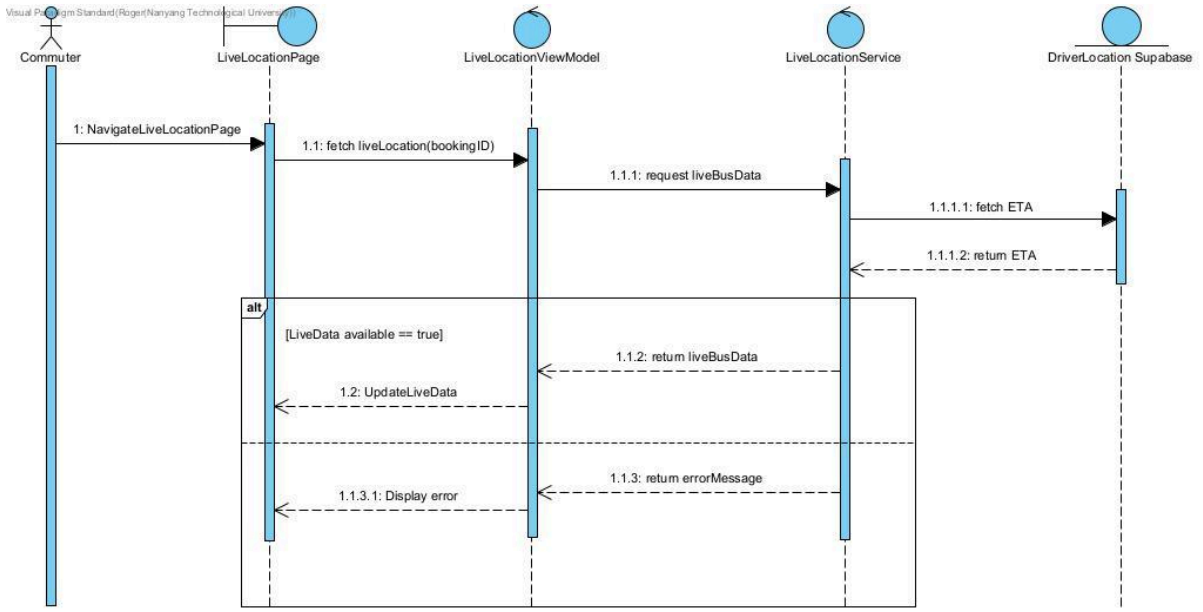
Use Case ID: 3.4
Use Case Name: Delete Bus Schedule



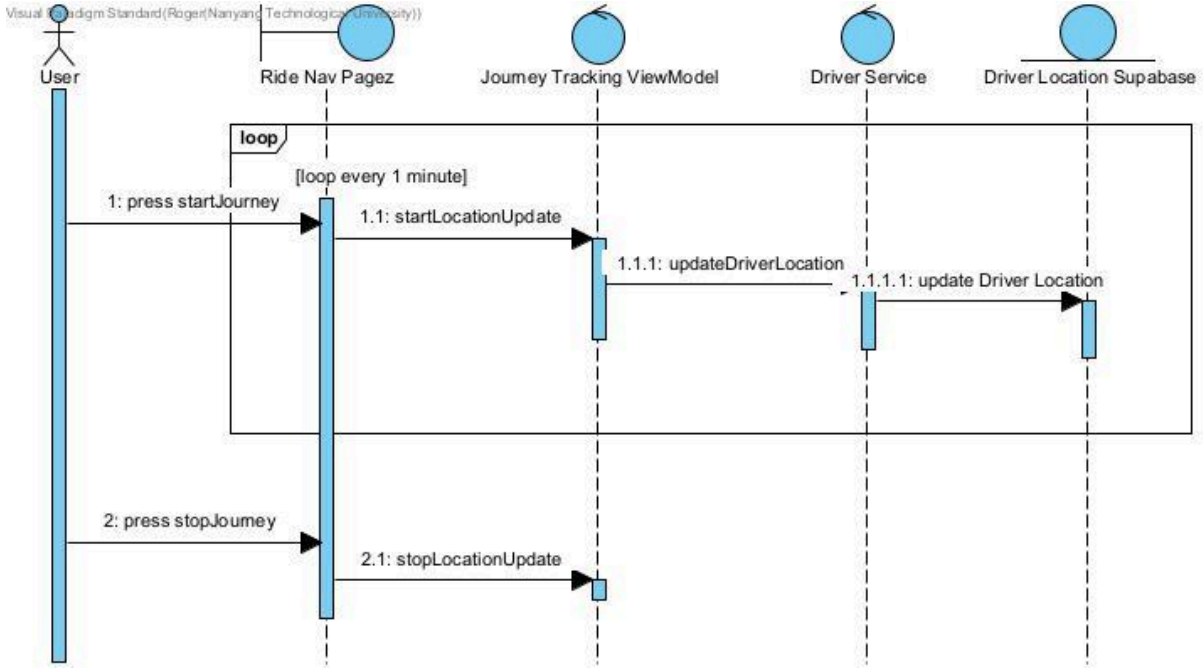
Use Case ID: 3.5
Use Case Name: Search for Bus Schedule



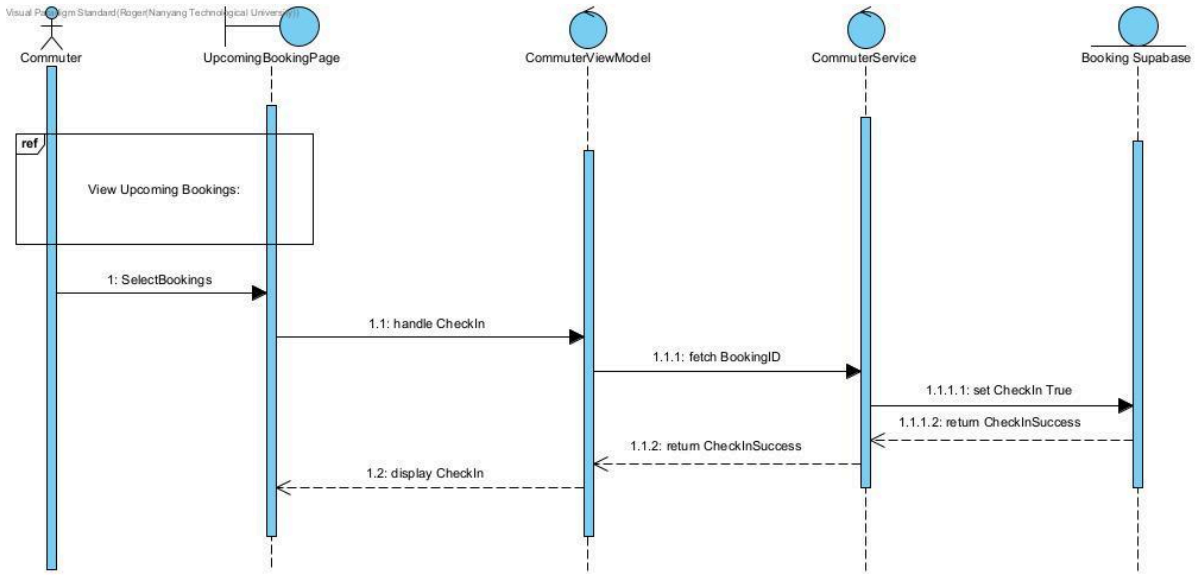
Use Case ID: 4.1
Use Case Name: Get Live Bus Information



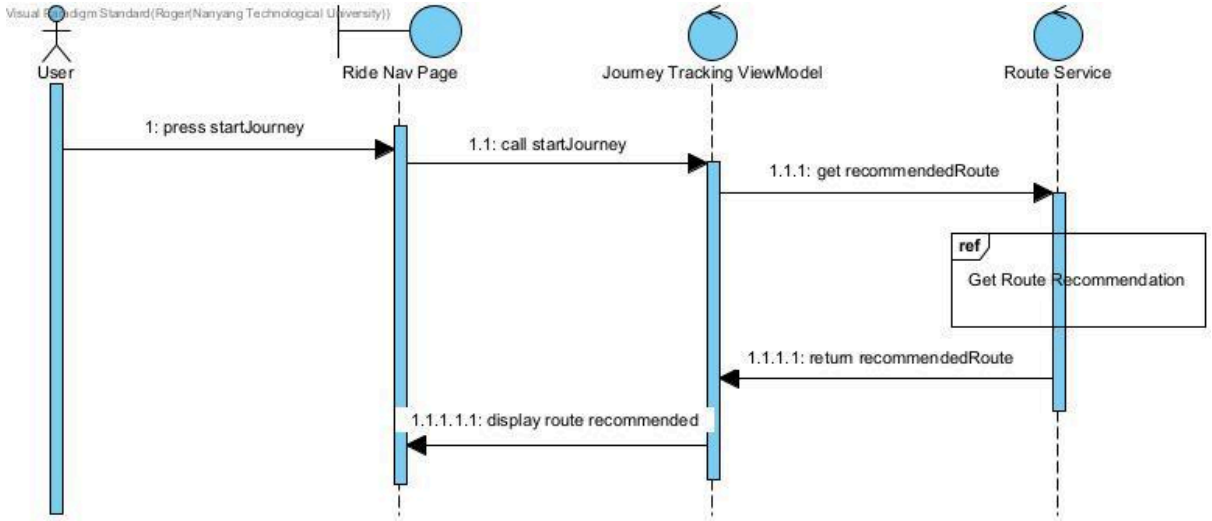
Use Case ID: 4.2
Use Case Name: Update Location



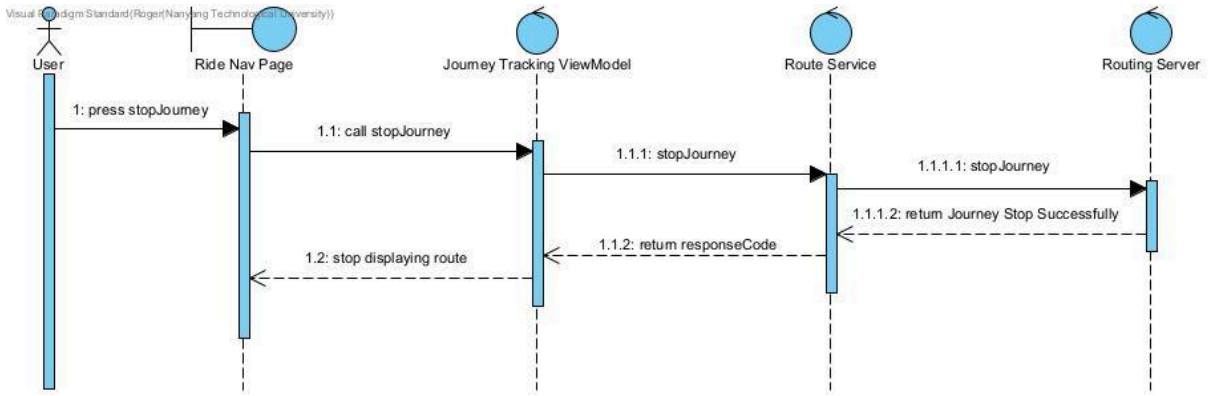
Use Case ID: 4.3
Use Case Name: Check into Bus



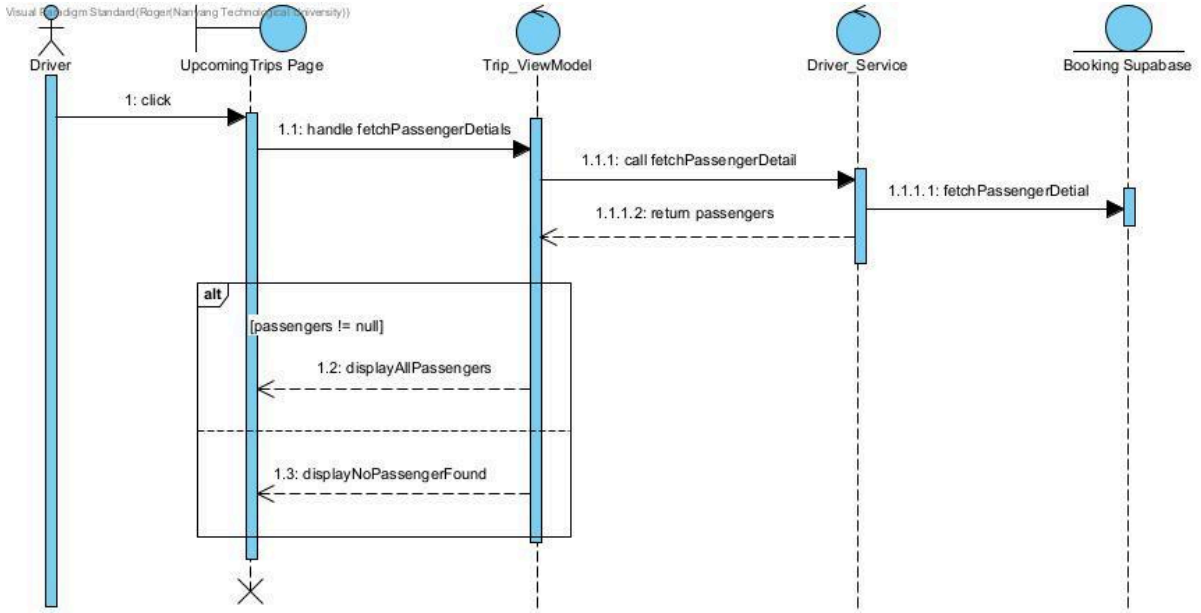
Use Case ID: 4.4
Use Case Name: Start Journey



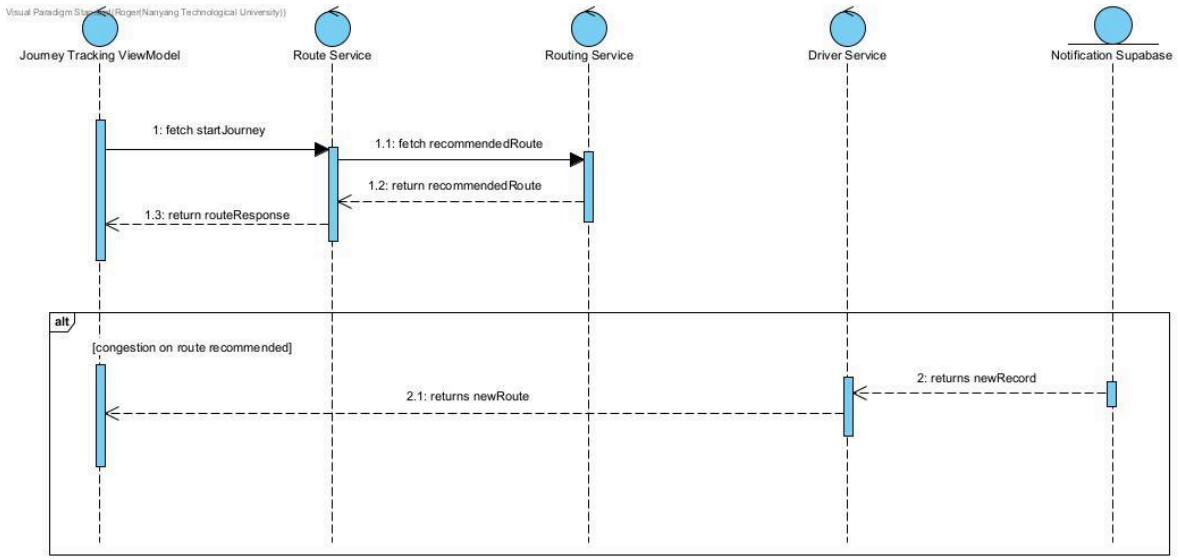
Use Case ID: 4.5
Use Case Name: End Journey



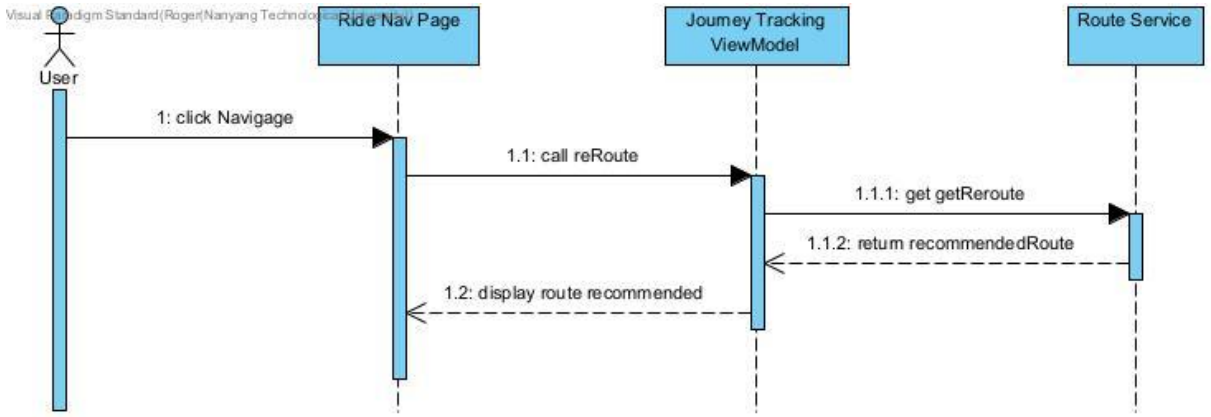
Use Case ID: 4.6
Use Case Name: View Commuters List



Use Case ID: 5.1
Use Case Name: Get Route Recommendations

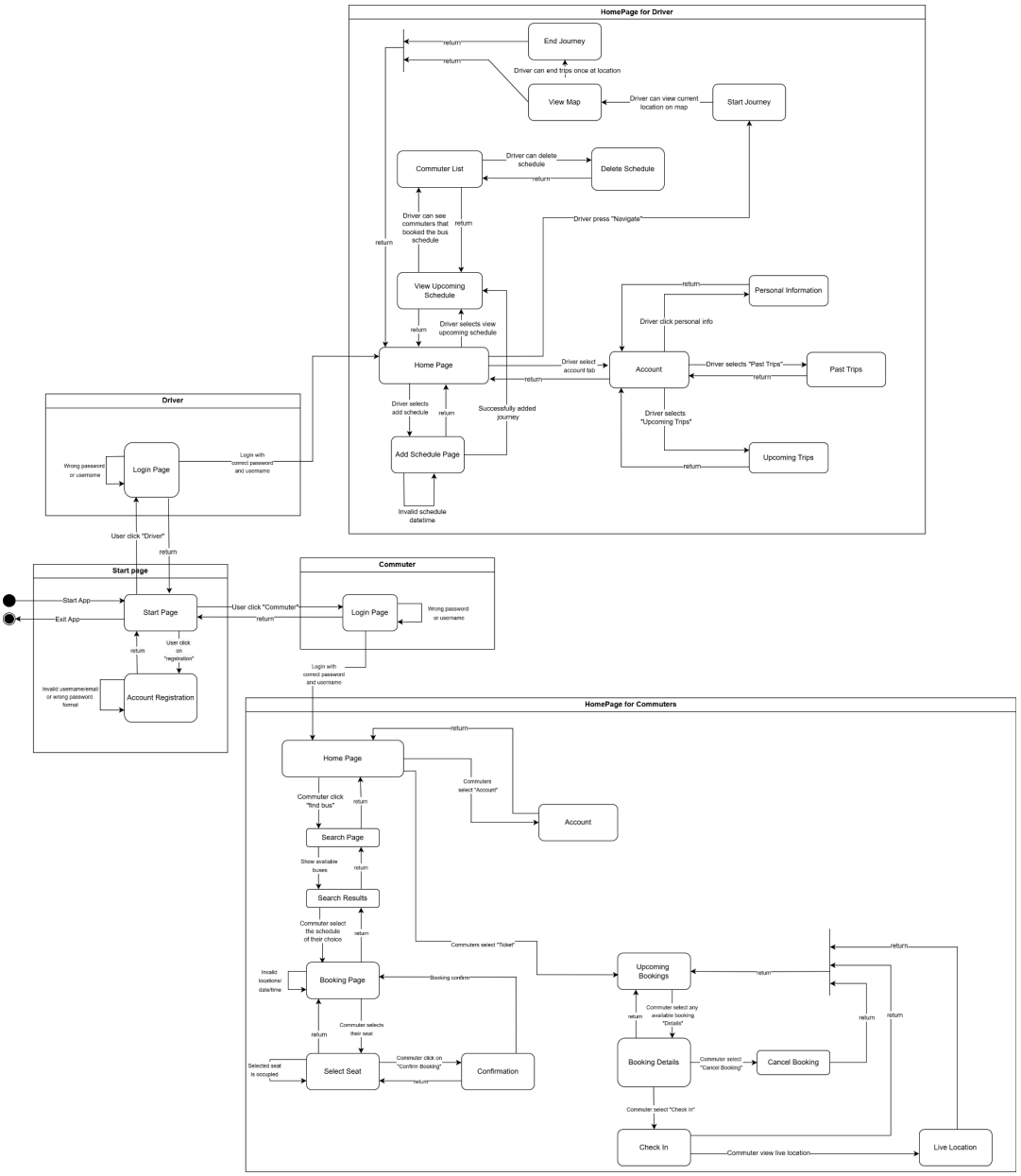


Use Case ID: 5.2
Use Case Name: Return Routing



12. Dialog Map

View in higher quality in our GitHub:
https://github.com/softwarelab3/2006-SCSD-C2/blob/main/Deliverables/Lab%205/Dialog%20Map.drawio.png



13. System Architecture Diagram

View in higher quality in our GitHub here:
https://github.com/softwarelab3/2006-SCSD-C2/blob/main/Deliverables/Lab%205/System%20Architecture.drawio.png
Our project adopts a hybrid architectural design: MVVM (Model-View-ViewModel) for the frontend and MCS (Model-Controller-Service) for the backend.

On the frontend, the MVVM pattern ensures a clean separation of UI and business logic.

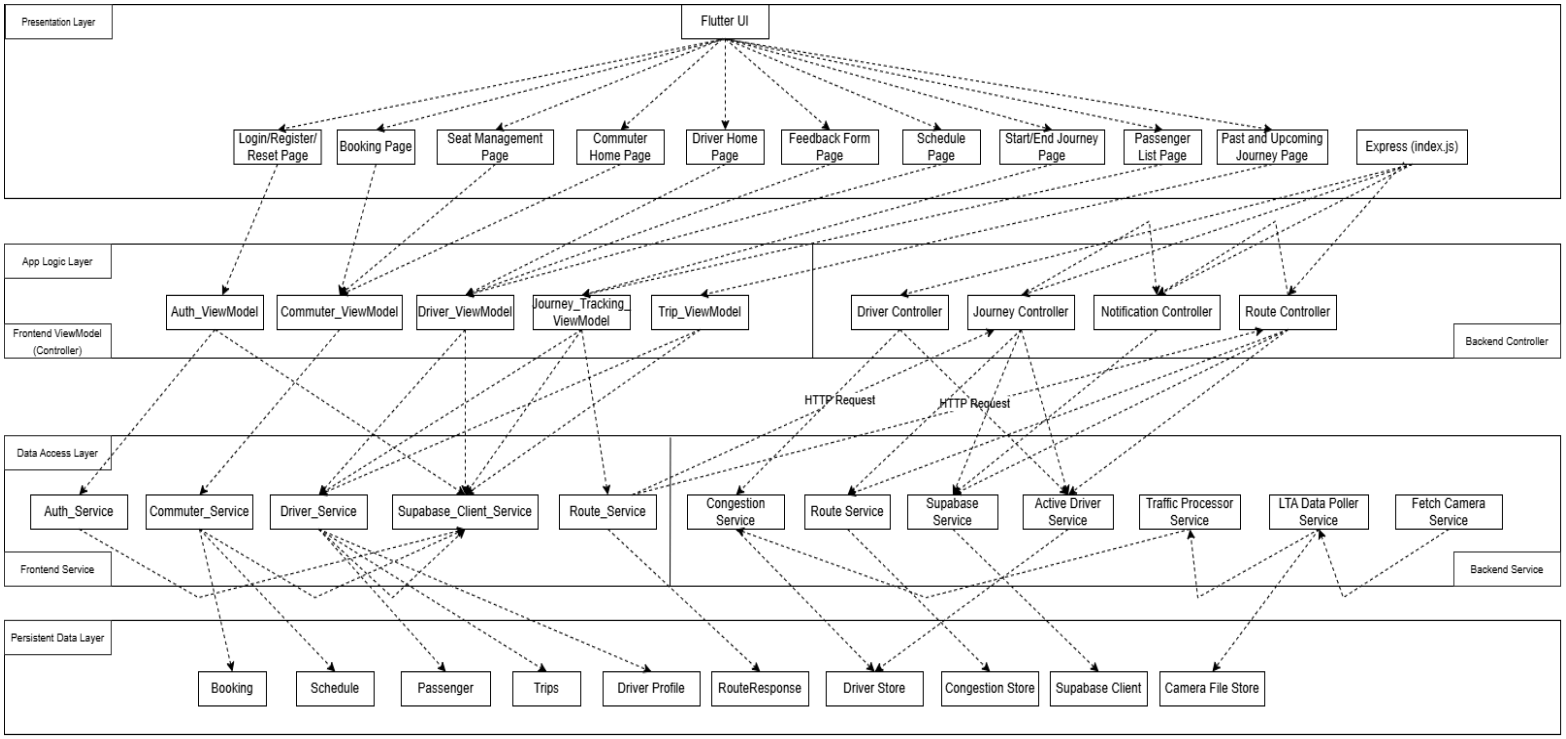
- Model represents the data layer
- ViewModel handles UI-related logic and state management
- View comprises the visual components rendered to the user

On the backend, we structured it using the MCS architecture.

- Model represents the data schema and handles interactions with the database
- Service layer contains business logic and reusable functionalities
- Controller acts as the entry point for handling incoming API requests

Although we did not explicitly include a “View” component in our backend architecture diagram, it is important to clarify that the backend does have a presentation layer. This layer functions as the point of contact with the frontend, typically responsible for formatting and sending responses such as JSON objects. In our implementation, this “View” functionality is handled by the index.js file, which routes requests to the appropriate controllers.

Given that this presentation layer is minimal and consists of only a single file, we made a design decision to integrate its responsibilities into the Controller layer. As a result, while our architecture is documented as MCS (Model-Controller-Service), the Controller implicitly covers the presentation logic, fulfilling the role of a view layer in practice.



Note: The frontend follows MVVM (Model-View-ViewModel) architecture, while the backend is structured with MCS (Model-Controller-Service). Each layer in this system architecture reflects the respective responsibilities.

14. System Design

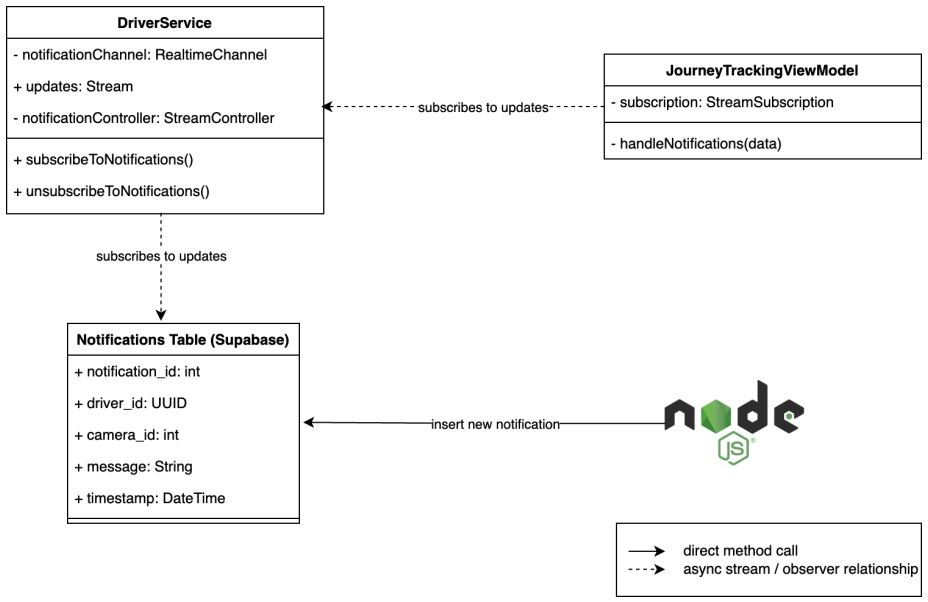
14.1 Observer Pattern

In order for our mobile app to receive real-time congestion updates, we implemented a reactive subscription system based on the Observer Pattern.

The DriverService component acts as a subject, and exposes a stream of congestion-related events. This allows the JourneyTrackingViewModel to subscribe to the stream using a StreamSubscription, and implement a handler (handleNotifications(data)) to handle the incoming stream of data.

The DriverService component itself is also an observer as it is listening to our “notifications” Supabase through a RealtimeChannel. The “notifications” table is updated by our backend routing server whenever congestion is high in a specific driver’s path.

Upon receiving an event, the DriverService component pushes it to the stream, which triggers the handler in the JourneyTrackingViewModel and pushes a notification to the driver’s screen and calls the backend routing server to reroute.



Using the Observer Pattern this way promotes loose coupling and makes our system event-driven, allowing the frontend to respond to the backend prompt without requiring polling every minute which could be expensive.

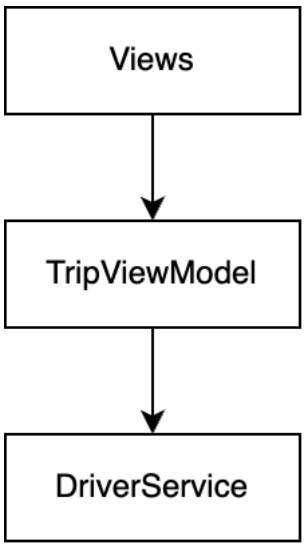
14.2 Facade Pattern

In our app, we followed the MVVM architecture pattern, which naturally introduces separation of concern between the User Interface (View) and ViewModel.

One example is our TripViewModel acting as a facade for multiple screens for the driver. TripViewModel simplifies access to underlying trip-related services and exposes high-level methods like fetchUpcomingConfirmedTrips(), fetchPastTrips(), etc. This allows the Views to use those services without needing to know how the data is handled.

Internally, TripViewModel delegates lower-level operations like queries to Supabase, time validation, conflict checking, etc. to the DriverService.

This design decouples the Views from the backend-specific logic and provides a clean and maintainable interface.



15. Black Box Testing

The control class we have selected for testing will be the **SignUp** control.

During the user sign up process, the user has to input their email, username, password, confirm password and select their role (commuter or driver). After a successful sign up, the user's details will be stored in the database.

However, the email and confirm password checks depend on format and matching values, not how long the text is. So, the boundary values for these 2 checkings cannot be determined. Moreover, the role selection will be commuter by default. Hence, there are no invalid cases for the role field.

15.1 Equivalence Class Testing & Boundary Value Testing

Username

	Invalid	**Valid**	**Invalid**
Equivalence Class	0 - 2 characters	3 - 10 characters	>= 11 characters

Valid Equivalence Class

Value on boundary values	3, 10 characters
Value just below boundary values	2 characters
Value just above boundary values	11 characters

Invalid Equivalence Class

Value on boundary values	0, 2 characters
Value just below boundary values	-1 characters (does not make sense)
Value just above boundary values	3 characters

Invalid Equivalence Class

Value on boundary values	11 characters
Value just below boundary values	10 characters
Value just above boundary values	Infinity (does not make sense)

Email

	Valid	**Invalid**
Equivalence Class	Has a proper format (x@domain.com) Not registered by others	Invalid email format Registered by others

Password

	Invalid	**Valid**	**Invalid**
Equivalence Class	0 - 5 characters	6 - 20 characters	>= 21 characters

Valid Equivalence Class

Value on boundary values	6, 20 characters
Value just below boundary values	5 characters
Value just above boundary values	21 characters

Invalid Equivalence Class

Value on boundary values	0, 5 characters
Value just below boundary values	-1 characters (does not make sense)
Value just above boundary values	6 characters

Invalid Equivalence Class

Value on boundary values	21 characters
Value just below boundary values	20 characters
Value just above boundary values	Infinity (does not make sense)

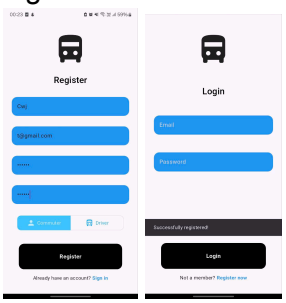
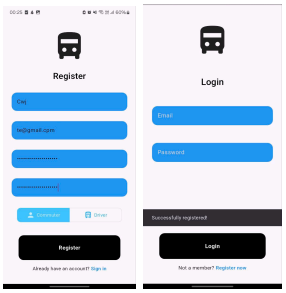
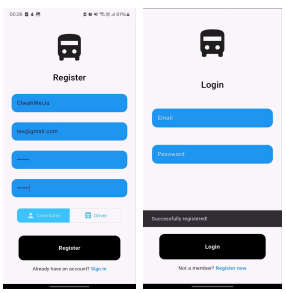
Confirm Password

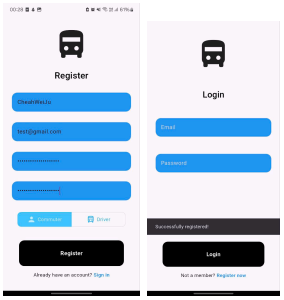
	Valid	**Invalid**
Equivalence Class	Password is identical to Confirm Password	Password is not identical to Confirm Password

15.2 Test Cases and Testing Result

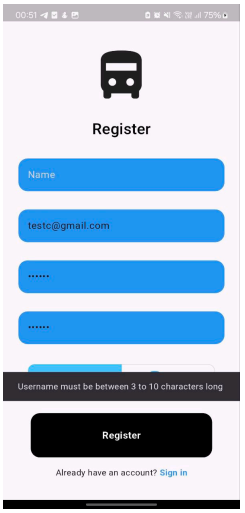
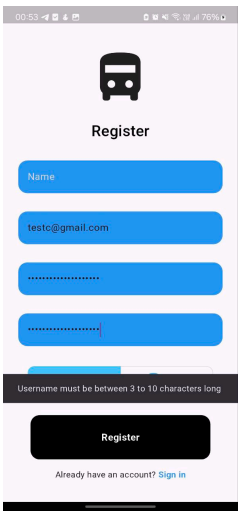
[Higher resolution of the picture can be found in Github]

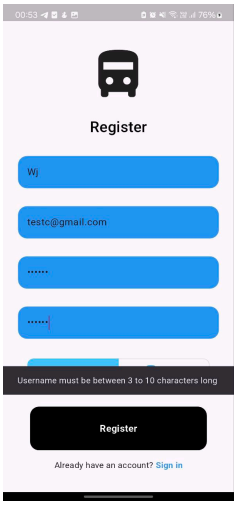
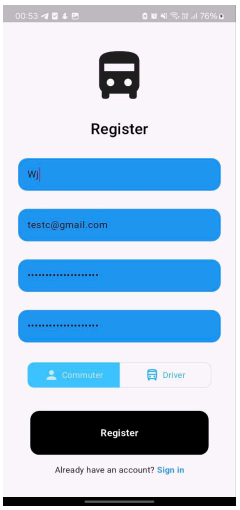
Valid Username (3, 10 characters) +
Valid Email (x@domain.com & not registered) +
Valid Password (6, 20 characters) +
Valid Confirm Password (ConfirmPassword = Password)

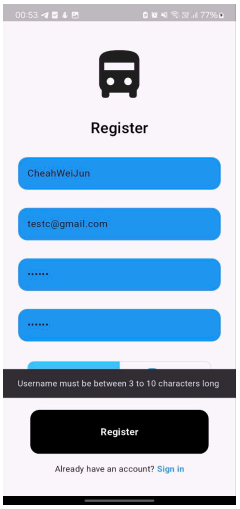
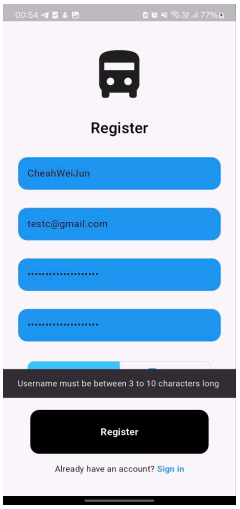
Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Signup-01	String Username: Cwj **(3 char)** String Email: t@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Successfully registered!	Successfully registered! 	Pass
Signup-02	String Username: Cwj **(3 char)** String Email: te@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Successfully registered!	Successfully registered! 	Pass
Signup-03	String Username: CheahWeiJu **(10 char)** String Email: tes@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Successfully registered!	Successfully registered! 	Pass

Signup-04	String Username: CheahWeiJu **(10 char)** String Email: test@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Successfully registered!	Successfully registered! 	Pass

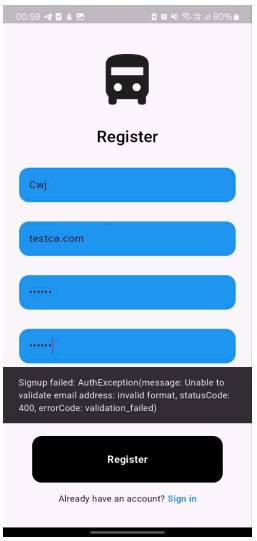
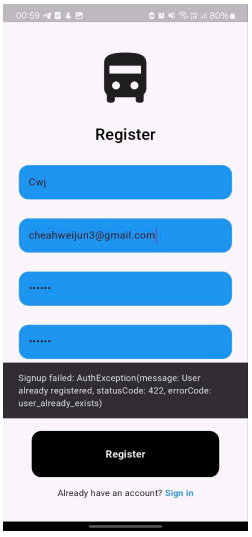
Invalid Username (0, 2, 11 characters) +
Valid Email (x@domain.com & not registered) +
Valid Password (6, 20 characters) +
Valid Confirm Password (ConfirmPassword = Password)

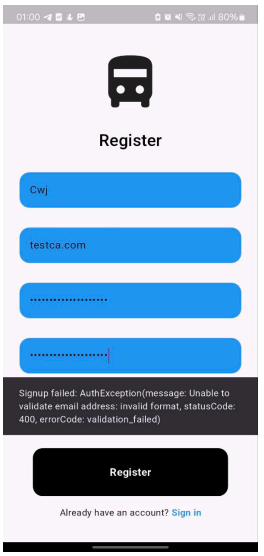
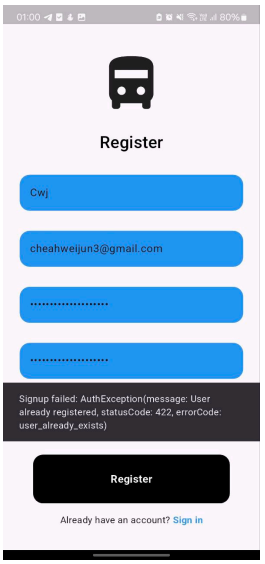
Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Signup-05	String Username: **(0 char)** String Email: testc@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Username must be between 3 to 10 characters long	Username must be between 3 to 10 characters long 	Pass
Signup-06	String Username: **(0 char)** String Email: testc@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Username must be between 3 to 10 characters long	Username must be between 3 to 10 characters long 	Pass

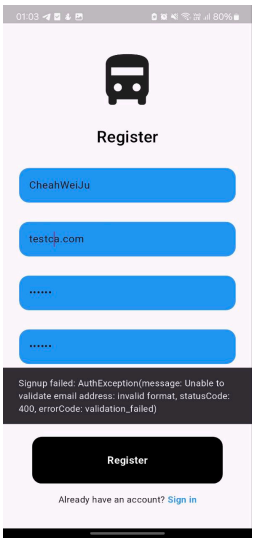
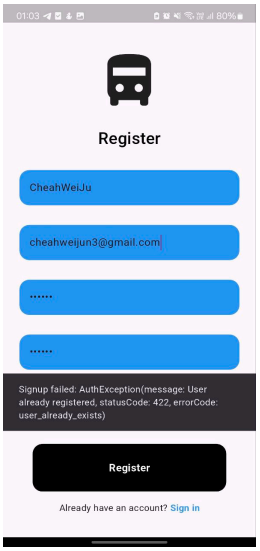
Signup-07	String Username: Wj **(2 char)** String Email: testc@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Username must be between 3 to 10 characters long	Username must be between 3 to 10 characters long 	Pass
Signup-08	String Username: Wj **(2 char)** String Email: testc@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Username must be between 3 to 10 characters long	Username must be between 3 to 10 characters long 	Pass

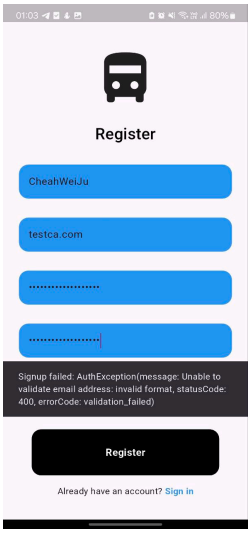
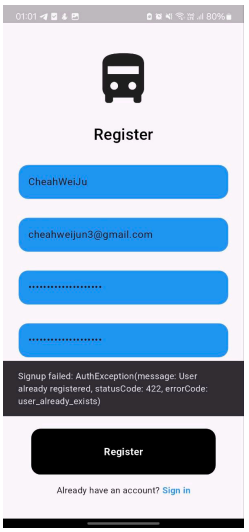
Signup-09	String Username: CheahWeiJun **(11 char)** String Email: testc@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Username must be between 3 to 10 characters long	Username must be between 3 to 10 characters long 	Pass
Signup-10	String Username: CheahWeiJun **(11 char)** String Email: testc@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Username must be between 3 to 10 characters long	Username must be between 3 to 10 characters long 	Pass

Valid Username (3, 10 characters) +
Invalid Email (x.com, registered) +
Valid Password (6, 20 characters) +
Valid Confirm Password (ConfirmPassword = Password)

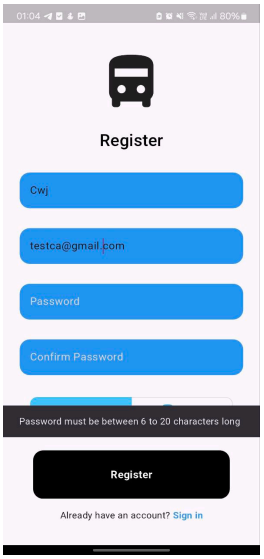
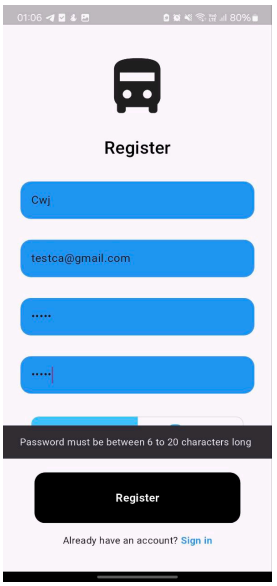
Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Signup-11	String Username: Cwj **(3 char)** String Email: testca.com **(invalid format)** String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Sign up Failed: Unable to validate email address	Sign up Failed: Unable to validate email address 	Pass
Signup-12	String Username: Cwj **(3 char)** String Email: cheahweijun3@gmail.com **(registered)** String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Sign up Failed: User already registered	Sign up Failed: User already registered 	Pass

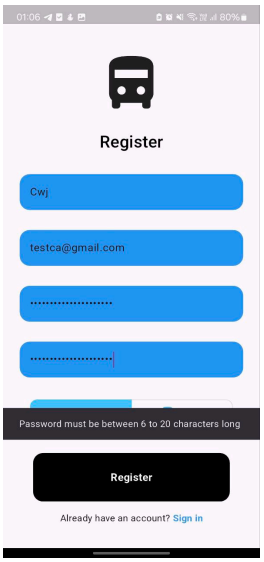
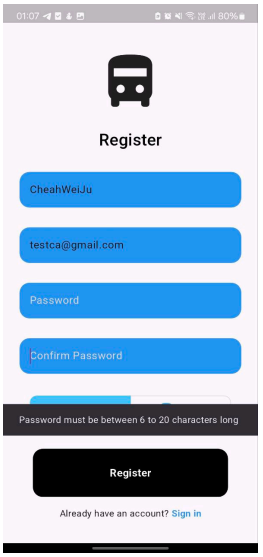
Signup-13	String Username: Cwj **(3 char)** String Email: testca.com **(invalid format)** String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Sign up Failed: Unable to validate email address	Sign up Failed: Unable to validate email address 	Pass
Signup-14	String Username: Cwj **(3 char)** String Email: cheahweijun3@gmail.com **(registered)** String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Sign up Failed: User already registered	Sign up Failed: User already registered 	Pass

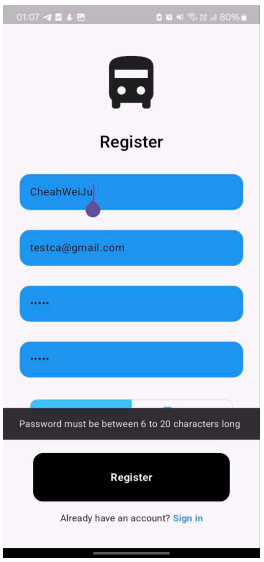
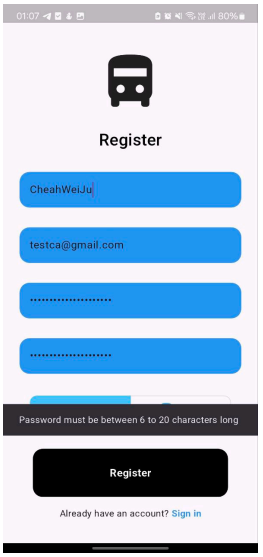
Signup-15	String Username: CheahWeiJu **(10 char)** String Email: testca.com **(invalid format)** String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Sign up Failed: Unable to validate email address	Sign up Failed: Unable to validate email address 	Pass
Signup-16	String Username: CheahWeiJu **(10 char)** String Email: cheahweijun3@gmail.com **(registered)** String Password: Cwj123 **(6 char)** String Confirm Password: Cwj123 String Role: Commuter	Sign up Failed: User already registered	Sign up Failed: User already registered 	Pass

Signup-17	String Username: CheahWeiJu **(10 char)** String Email: testca.com **(invalid format)** String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Sign up Failed: Unable to validate email address	Sign up Failed: Unable to validate email address 	Pass
Signup-18	String Username: CheahWeiJu **(10 char)** String Email: cheahweijun3@gmail.com **(registered)** String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: Cwj12345678909876543 String Role: Commuter	Sign up Failed: User already registered	Sign up Failed: User already registered 	Pass

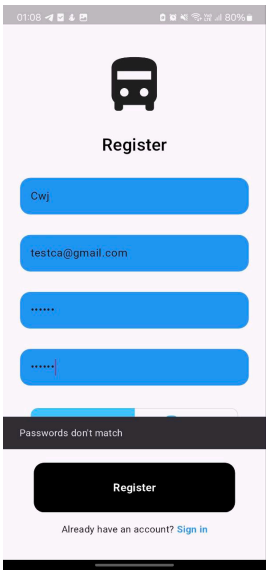
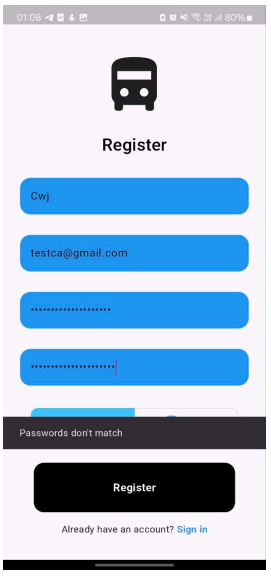
Valid Username (3, 10 characters) +
Valid Email (x@domain.com & not registered) +
Invalid Password (0, 5, 21 characters) +
Valid Confirm Password (ConfirmPassword = Password)

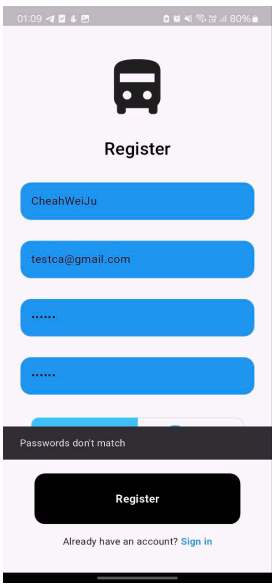
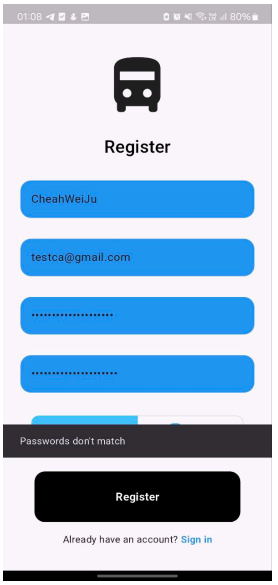
Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Signup-19	String Username: Cwj **(3 char)** String Email: testca@gmail.com String Password: **(0 char)** String Confirm Password: String Role: Commuter	Password must be between 6 to 20 characters long	Password must be between 6 to 20 characters long 	Pass
Signup-20	String Username: Cwj **(3 char)** String Email: testca@gmail.com String Password: Cwj12 **(5 char)** String Confirm Password: Cwj12 String Role: Commuter	Password must be between 6 to 20 characters long	Password must be between 6 to 20 characters long 	Pass

Signup-21	String Username: Cwj **(3 char)** String Email: testca@gmail.com String Password: Cwj123456789098765432 **(21 char)** String Confirm Password: Cwj123456789098765432 String Role: Commuter	Password must be between 6 to 20 characters long	Password must be between 6 to 20 characters long 	Pass
Signup-22	String Username: CheahWeiJu **(10 char)** String Email: testca@gmail.com String Password: **(0 char)** String Confirm Password: String Role: Commuter	Password must be between 6 to 20 characters long	Password must be between 6 to 20 characters long 	Pass

Signup-23	String Username: CheahWeiJu **(10 char)** String Email: testca@gmail.com String Password: Cwj12 **(5 char)** String Confirm Password: Cwj12 String Role: Commuter	Password must be between 6 to 20 characters long	Password must be between 6 to 20 characters long 	Pass
Signup-24	String Username: CheahWeiJu **(10 char)** String Email: testca@gmail.com String Password: Cwj123456789098765432 **(21 char)** String Confirm Password: Cwj123456789098765432 String Role: Commuter	Password must be between 6 to 20 characters long	Password must be between 6 to 20 characters long 	Pass

Valid Username (3, 10 characters) +
Valid Email (x@domain.com) +
Valid Password (6, 20 characters) +
Invalid Confirm Password (ConfirmPassword != Password)

Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Signup-25	String Username: Cwj **(3 char)** String Email: testca@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: 123Cwj String Role: Commuter	Passwords don't match	Passwords don't match 	Pass
Signup-26	String Username: Cwj **(3 char)** String Email: testca@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: 12345678909876543Cwj String Role: Commuter	Passwords don't match	Passwords don't match 	Pass

Signup-27	String Username: CheahWeiJu **(10 char)** String Email: testca@gmail.com String Password: Cwj123 **(6 char)** String Confirm Password: 123Cwj String Role: Commuter	Passwords don't match	Passwords don't match 	Pass
Signup-28	String Username: CheahWeiJu **(10 char)** String Email: testca@gmail.com String Password: Cwj12345678909876543 **(20 char)** String Confirm Password: 12345678909876543Cwj String Role: Commuter	Passwords don't match	Passwords don't match 	Pass

16. White Box Testing

The 2 methods selected for white box testing will be:

1) SubmitJourney
2) FetchPassengerDetails

The **SubmitJourney** method allows drivers to add new journeys into the bus schedules. During the SubmitJourney process, the drivers select pick-up point, destination point, date and time.
This include 5 scenarios:

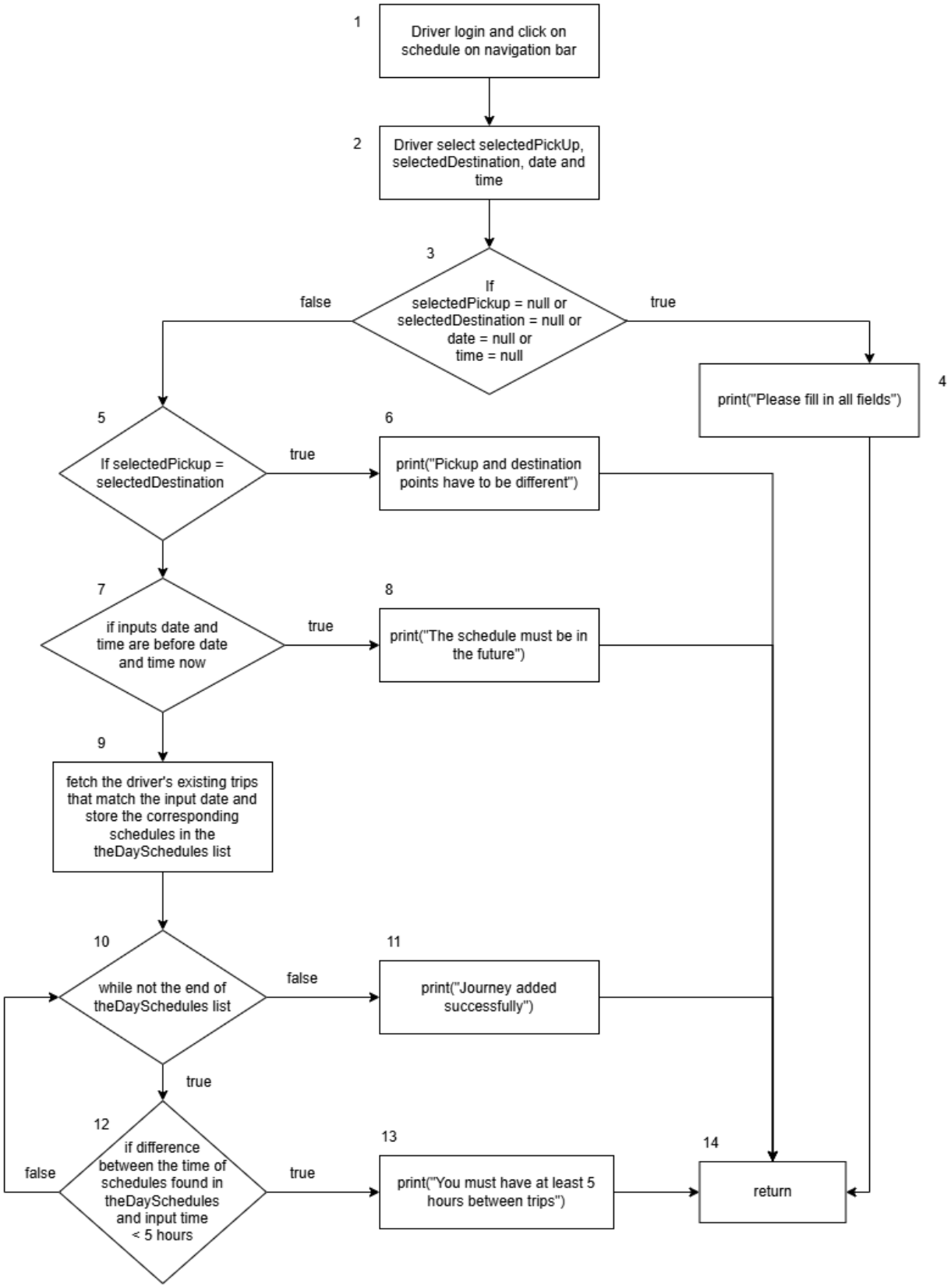
1) Drivers fail to fill in all the required fields.
2) Drivers enter identical pickup and destination points.
3) Drivers input a date and time that is in the past.
4) Drivers attempt to schedule a new trip where they already have another scheduled trip within 5 hours of the new schedule.
5) Drivers correctly fill in all the fields, providing valid information.

The **FetchPassengerDetails** method displays the details of passengers if they exist.
This includes 3 scenarios:

1) No passengers have booked for the trip.
2) Passengers have booked for the trip but have not checked in.
3) Passengers have booked for the trip and have checked in.

16.1 SubmitJourney

16.1.1 Control Flow Graph



16.1.2 Cyclomatic Complexity

Cyclomatic Complexity (CC) = |binarydecisionpoint| + 1 = 5 + 1 = 6

16.1.3 Basis Path

Basis Path #1: 1, 2, 3, 5, 7, 9, 10, 11, 14

Basis Path #2: 1, 2, 3, 4, 14

Basis Path #3: 1, 2, 3, 5, 6, 14

Basis Path #4: 1, 2, 3, 5, 7, 8, 14

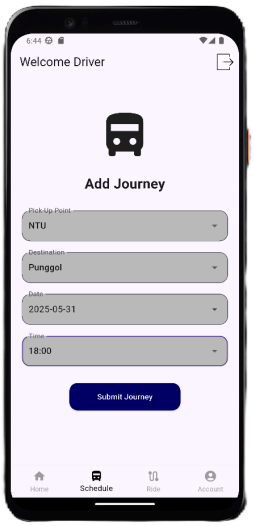
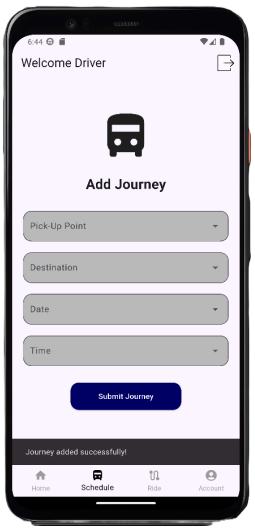
Basis Path #5: 1, 2, 3, 5, 7, 9, 10, 12, 10, 11, 14

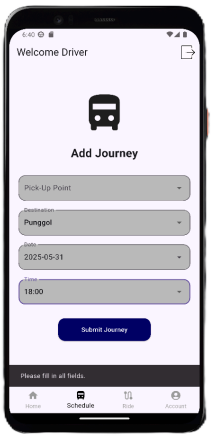
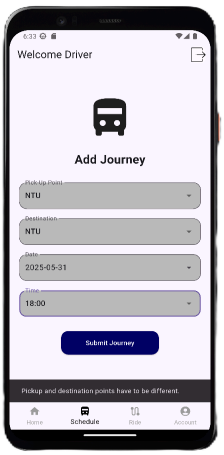
Basis Path #6: 1, 2, 3, 5, 7, 9, 10, 12, 13, 14

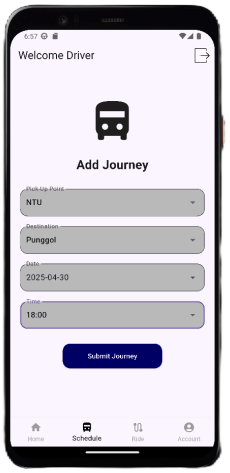
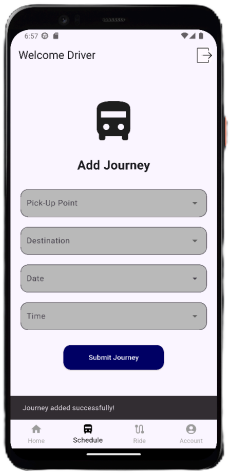
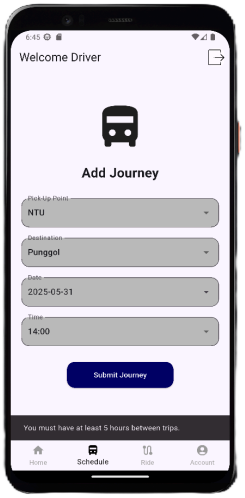
16.1.4 Test Cases and Testing Results

Input Parameters:

1) selectedPickUp
2) selectedDestination
3) date
4) time

Test Case Name	**Test Input**	**Expected Output**	**Actual Output**	**Test Result**
Submit-01	String selectedPickUp = NTU String selectedDestination = Punggol Date date = 2025-05-31 Time time = 18:00	Success Message: “Journey added successfully”	Success Message: “Journey added successfully”   	Pass

Submit-02	String selectedPickUp = null String selectedDestination = Punggol Date date = 2025-05-31 Time time = 18:00	Error Message: “Please fill in all fields”	Error Message: “Please fill in all fields 	Pass
Submit-03	String selectedPickUp = NTU String selectedDestination = NTU Date date = 2025-05-31 Time time = 18:00	Error Message: “Pickup and destination points have to be different”	Error Message: “Pickup and destination points have to be different” 	Pass
Submit-04	String selectedPickUp = NTU String selectedDestination = Punggol Date date = 2025-01-01 Time time = 18:00	Error Message: “The schedule must be in the future”	Error Message: “The schedule must be in the future” 	Pass

Submit-05	String selectedPickUp = NTU String selectedDestination = Punggol Date date = 2025-04-30 Time time = 18:00	Success Message: “Journey added successfully”	Success Message: “Journey added successfully”  	Pass
Submit-06	String selectedPickUp = NTU String selectedDestination = Punggol Date date = 2025-05-31 Time time = 14:00	Error Message: “You must have at least 5 hours between trips”	Error Message: “You must have at least 5 hours between trips” 	Pass

16.2 Fetch Passenger Details

16.2.1 Control Flow Graph

16.2.2 Cyclomatic Complexity

Cyclomatic Complexity (CC) = |binarydecisionpoint| + 1 = 3 + 1 = 4

16.2.3 Basis Path

Basis Path #1: 1, 2, 3, 4, 9, 10, 11, 16

Basis Path #2: 1, 2, 3, 4, 9, 10, 12, 13, 14, 16 **(infeasible)**

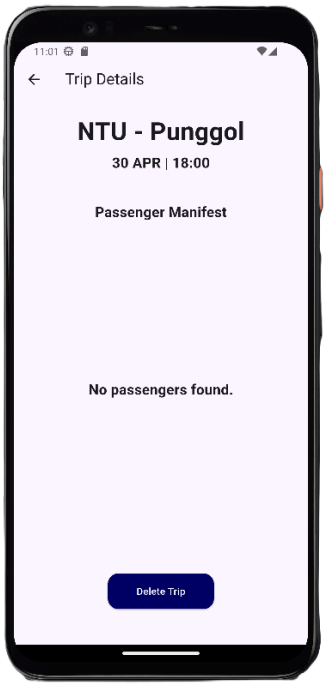
Basis Path #3: 1, 2, 3, 4, 5, 6, 7, 8, 4, 9, 10, 12, 13, 14, 16

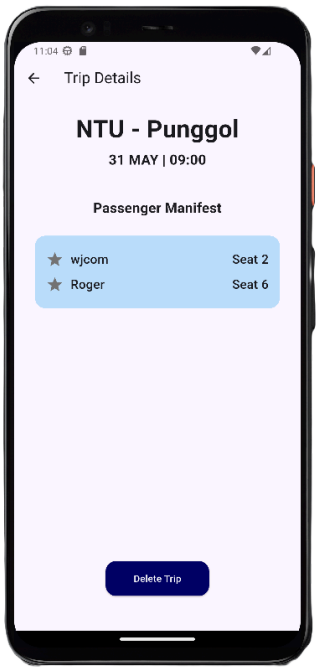
Basis Path #4: 1, 2, 3, 4, 5, 6, 7, 8, 4, 9, 10, 12, 13, 15, 16

16.2.4 Test Cases and Testing Results

Parameters:

1) scheduleId

Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Fetch-01	Int scheduleId = 179	No passengers found	No passengers found 	Pass
Fetch-02	Infeasible Path			

Fetch-03	Int scheduleId = 178	Grey icon, wjcom, Seat 2 Grey icon, Roger, Seat 6	Grey icon, wjcom, Seat 2 Grey icon, Roger, Seat 6 	Pass
Fetch-04	Int scheduleId = 177	Yellow icon, wjcom, Seat 1 Yellow icon, xy, Seat 5	Yellow icon, wjcom, Seat 1 Yellow icon, xy, Seat 5 	Pass

16.3 Demo Script

We apply the Single Responsibility Principle, ensuring that each class is responsible for only one specific functionality. This minimizes the risk of unintended side effects and enhances code maintainability. For instance, our view models are clearly separated based on functionality, such as TripViewModel, which handles all trip-related operations.

We also use a multi-layered architecture consisting of the presentation layer, application logic layer, data access layer, and persistent data layer. For the frontend, we applied the MVVM architecture, where the UI layer (e.g., TripListUI) handles layout and user interactions, the application logic layer, mainly the view models, manages business logic and updates views if there are any changes, and the service layer handles data fetching from or storing into Supabase.

For the backend, it is built using Node.js with an MCS (Model-Controller-Service) architecture. The flow begins when a user (commuter or driver) interacts with the frontend, triggering API requests handled by Express routes. The presentation layer comprises the express + controller classes which handles the request and delegates logic to the service layer. The controller extracts request parameters and returns status codes and data in JSON format. The service layer handles business logic such as checking if a journey has started, generating optimized routes, or notifying drivers of congestion while the data layer is responsible for reading and writing data. In this figure, you can see how the overall backend flows.

For the data layer, We are using Supabase as our database, with data organized into separate tables based on their specific functionalities. This helps to maintain clarity, improves data management, and makes it easier for us to add new features as the system grows.

Some of the good software engineering practices we have adopted include modular code structure, efficient state management, input validation, and error handling. A modular code structure means our project is divided into separate layers following the MVVM architecture as shared just now, which enhances code maintainability and readability. Additionally, we adopted efficient state management using ChangeNotifier and Provider. This approach enables automatic UI updates in response to state changes, eliminating the need for manual refreshes and significantly improving the overall user experience. We also implemented input validation and error handling using Snackbars to display clear and user-friendly error messages when inputs are invalid. This helps reduce the likelihood of system crashes and ensures the application behaves predictably and reliably.

By adopting these good practices, we are able to support future updates more easily. UI changes in the view layers, such as modifying layouts or adding new elements, can be made without impacting the core business logic in the view model layers. This separation enhances maintainability and allows for more flexible development. Additionally, with input validation and error checking in place, issues can be quickly isolated and resolved, making the system more robust. This is especially useful when introducing new features, as they may unintentionally break existing functionality. With built-in error handling mechanisms, the

system is better equipped to manage edge cases, making future development and feature expansion more efficient and reliable.

Moving on to the use case, we would like to highlight the process of creating new bus schedules by the drivers. This use case allows drivers to indicate their availability to commuters, along with key trip details such as pickup and destination points, and the scheduled departure date and time. The process begins when drivers navigate to the ScheduleNav page and enter the schedule details. These details are then passed to the Trip_ViewModel, where validation takes place. If the entered details are valid, the Trip_ViewModel will call Driver_Service to store the details in the schedules table in Supabase. Upon successful storage, a success message ("Journey Stored Successfully") is sent back to the Trip_ViewModel. The Trip_ViewModel then triggers UI updates in the ScheduleNav page to display the newly added schedule. If an error occurs during UI update, an error message is shown.

This is the class diagram of the use case. It begins by triggering the submitTrip function in the Trip_ViewModel, which in turn calls the addJourney function in the Driver Service to store the trip data in Supabase. This use case also utilizes the Trip model, where fields like ID, driver name, date, time, pickup and destination points, and trip status are maintained.

To test if this use case functions correctly, we perform white-box testing, and this is the control flow graph. There are five scenarios to consider:

1. Drivers fail to fill in all the required fields.

2. Drivers enter identical pickup and destination points.

3. Drivers input a date and time that is in the past.

4. Drivers attempt to schedule a new trip where they already have another scheduled trip within 5 hours of the new one.

5. Drivers correctly fill in all the fields with valid information.

Since all decision points are binary, we calculate a cyclomatic complexity of 6.

- The first basis path is the scenario where the driver enters all valid details and does not have any other trips scheduled on the same day. In this case, the driver will receive a success message of "journey added successfully," indicating that the journey has been added successfully.

- The second basis path occurs when the driver does not fill in all the required fields, such as leaving the pickup point blank. In this case, an error message stating "Please fill in all fields" will be displayed to the driver.

- The third basis path is triggered when the driver enters identical pickup and destination points, such as both being "NTU." In this case, the system will display an

error message saying, "Pickup and destination points must be different."

- The fourth basis path occurs when the driver attempts to schedule a trip with a date in the past, such as January 1st. The system will show an error message saying, "The schedule must be in the future."

- The fifth basis path happens when the driver has other trips on the same day, but the time difference between them is more than 5 hours. In this case, the journey will be added successfully, and the driver will receive a success message indicating that the journey has been added.

- The sixth basis path occurs when the driver has other trips scheduled on the same day, and the time difference between the trips is less than 5 hours. In this case, an error message will be shown stating, "You must have at least 5 hours between trips."