SWINBURNE UNIVERSITY OF TECHNOLOGY

School of Science, Computing and Engineering Technologies

ASSIGNMENT AND PROJECT COVER SHEET

Subject Code: SWE30003 Unit Title: Software Architectures and Design

Assignment number and title: 3. Design Implementation Due date: 2nd Apr 2025

Tutorial Day and Time: 13:00 - 16:00 Thu Project Group: 5

Tutor: Dr. Do Thi Bich Ngoc

To be completed as this is a group assignment
We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

ID Number	Name	Signature
104182779	Do Duy Hung	Hung
104221559	Doan Trung Nghia	Nghia
104181857	Luong Minh Duc	Duc
104175326	Do Quang Minh	Minh

Marker's comments:

Total Mark:

Extension certification:

This assignment has been given an extension and is now due on

Signature of Convener:

Introduction

This project focuses on the comprehensive design and execution of an Online Smart Parking Space (OSPS) system for SmartParking, a firm on Duy Tan Street. The primary aim is to enhance and expand on the first concept to provide a thorough, implementable solution that tackles the inefficiencies in the present parking system. This includes making feedback-driven changes to the class structure, relationships, and interactions, as well as constructing a functioning system prototype.

This paper describes the improved design, implementation details, and thoughts on the original concept, emphasizing improvements and lessons learned. The installation includes essential areas such as account administration, reservation processing, billing, and parking advice, resulting in a complete smart parking solution.

SWINBURNE UNIVERSITY OF TECHNOLOGY **1**
ASSIGNMENT AND PROJECT COVER SHEET **1**
Introduction **2**
1, Summary of Design Revision **2**
2, Detailed Design **3**
2.1 Overall changes at class level 5
2.2 Responsibilities and Collaborators 7
2.3 Changes to dynamic aspects (bootstrap, interaction scenarios) 10
3, Design Quality **11**
3.1 Good aspect 11
3.2 Missing aspects 12
3.3 Error aspects of the original design 12
3.4 Level of interpretation 12
3.5 Lesson learnt 13
4, Implementation and Testing **13**
4.1 Mapping design to code 13
4.2 Compilation and Execution 13
4.3 Testing 17
5, Appendix **17**

1, Summary of Design Revision

After conducting an in-depth analysis of our Assignment 2, we have revised the initial object-oriented design and improved the classes, the table below states changes made during the revision as well as the UML diagram of classes.

Aspect	Planned in CRC Cards (Assignment 2)	Revised in Code Implementation (Assignment 3)	Justification/Impact
User Class	• Attributes: userID, name, phoneNumber, email. • Responsibilities: Store basic user info; authenticate credentials.	• Attributes: userID, name, email, password. • Functionality is limited to registration and authentication.	• Simplified attributes (phoneNumber removed) to focus on core login and identification. • Serves as a base class for role-specific classes (Passenger, Driver).
Passenger Class	• Responsibilities: Create and manage ride requests, view ride history, leave feedback. • Collaborators: RideRequest, RatingReview.	• Implements dedicated methods: createPassenger(), isPassenger(), getPassengerID(), and authenticatePassenger().	• Role-specific functionality is separated from User. This improves clarity by ensuring that Passenger objects always reflect the passenger record (with its own ID) as stored in the database. • Authentication now verifies that a User has an associated Passenger record.
Driver Class	• Responsibilities: Accept ride requests, update vehicle information, maintain ratings. • Collaborators: RideRequest, RatingReview, Vehicle.	• Implements dedicated methods: createDriver(), isDriver(), and authenticateDriver().	• Similar to Passenger, role-specific methods ensure that a Driver object reflects a record from the driver table (with a distinct driver ID). • This separation avoids confusion and allows role-based operations to be implemented independently.
RideRequest Class	• Responsibilities: Store pickup/drop-off location, request time, fare, status. • Originally planned to “inherit” from	• Revised to use composition rather than inheritance. • Provides methods to create a Location record and to insert a ride request using the passenger’s ID retrieved from the Passenger class.	• Using composition (“has-a” relationship) better models the real-world scenario where a RideRequest is associated with a Passenger, rather than being a specialized type of Passenger.

	Passenger to access passenger info.		• This change improves flexibility and avoids the pitfalls of inheritance misuse.
Ride Class	• Responsibilities: Represent a confirmed ride between a passenger and driver; track ride status; compute final fare.	• Implements methods such as getPendingRides() to retrieve unassigned ride requests and startRide() to update ride status and assign a driver.	• Clearly distinguishes between a ride request (pre-ride) and a confirmed ride (in-progress or completed). • The design now reflects the two-phase process (request then confirmation) for better clarity.
Location Class	• Responsibilities: Store coordinates and address information for pickup and drop-off.	• Provides a dedicated Location class with attributes for pickup_address, dropoff_address, pickup_lat, pickup_long, dropoff_lat, dropoff_long; includes a static create() method for inserting records into the database.	• Centralizing all location-related data in one class improves reusability and data integrity. It avoids scattering address and coordinate handling across multiple classes.
Other Classes (Vehicle, Payment, PaymentMethod, RatingReview)	• Planned as part of the initial candidate list to cover all aspects of ride management, payment processing, and feedback.	• Our implementation (in the zip file) includes the core classes (User, Passenger, Driver, RideRequest, Ride, Location) and supports the intended functionality. Extraneous classes like Calculator and DataVisualization were removed, and Payment and RatingReview are implemented to the extent required for core functionality.	• Removing or merging unnecessary classes streamlines the design, focuses on core features, and makes the system easier to maintain.
Relationships & Associations	• CRC cards showed various relationships (inheritance, composition, aggregation) between classes like User, Passenger, Driver,	• Revised relationships: Role-specific classes no longer inherit from User via RideRequest; instead, RideRequest uses composition to reference Passenger. • Associations now correctly represent interactions, e.g., RideRequest retrieves passenger ID from Passenger, and Location	• Clarifies dependencies and responsibilities. Using composition/aggregation rather than incorrect inheritance improves modularity and allows each class to evolve independently.

	RideRequest, and Payment.	is managed independently.	
Design Patterns	• Initial design mentioned using Factory, Singleton, Observer, and Strategy patterns.	• The Database class consistently implements the Singleton pattern. Factory methods are used in role-specific creation (e.g., Passenger::createPassenger()). Observer and Strategy patterns are documented and applied in parts of the code (e.g., dynamic pricing calculation, real-time routing updates).	• Ensures a consistent and scalable approach to object creation, state management, and real-time updates. Patterns were refined to match the actual system needs rather than overengineering the design.

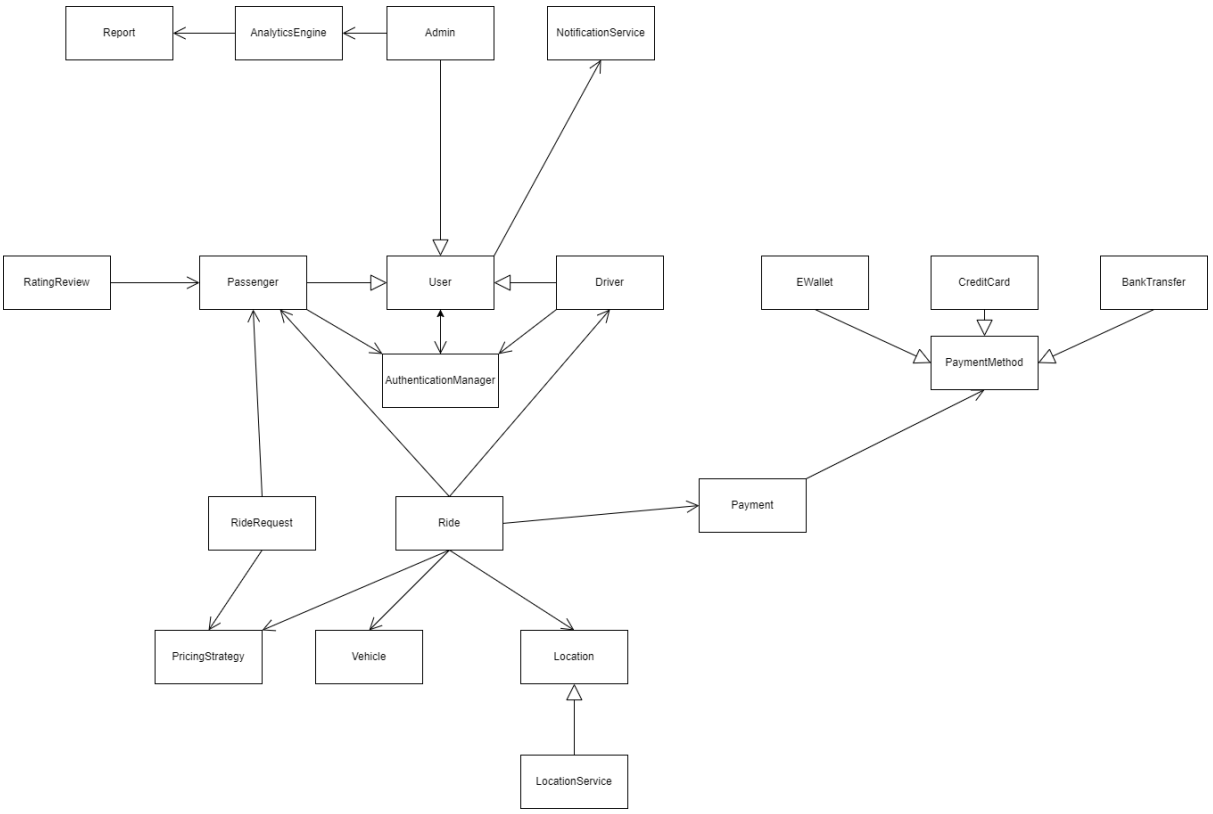

Figure 1 - Initial Design from assignment 2

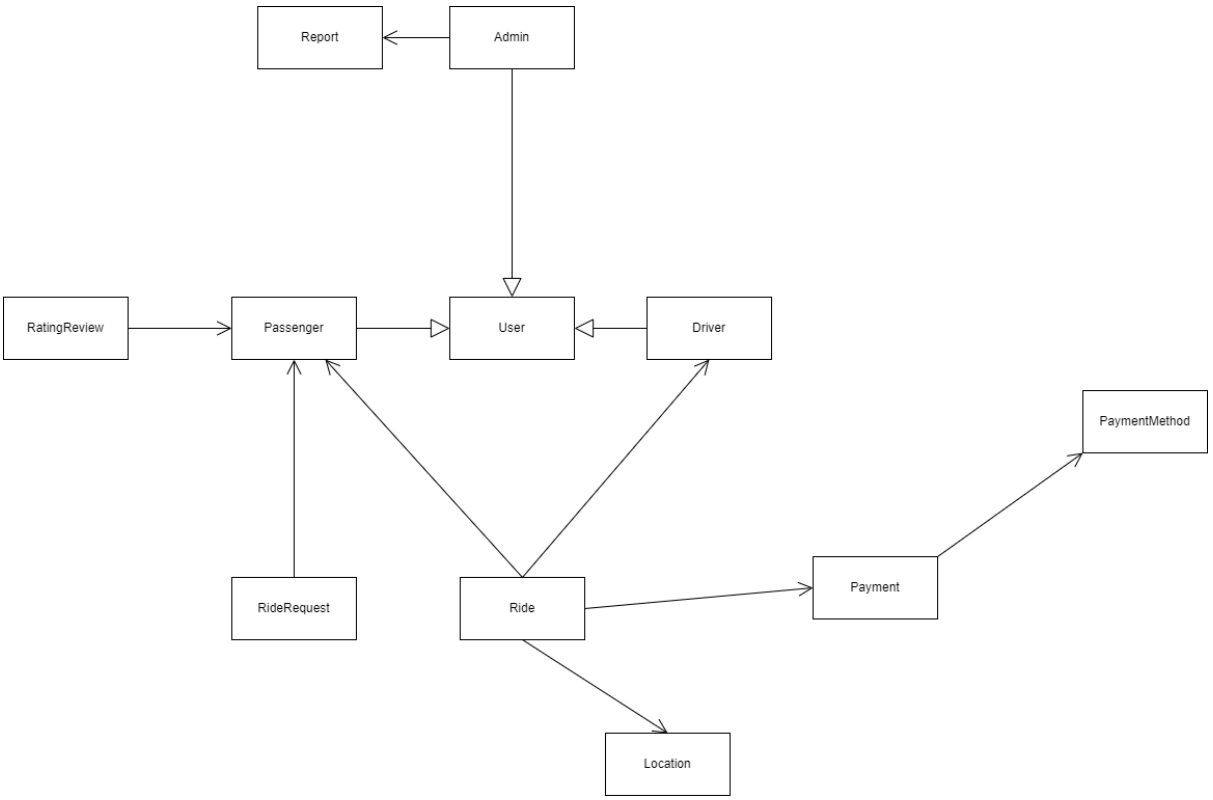

Figure 2 - Refined Design

2, Detailed Design

The UML Class Diagram for SmartRide comprises several key classes that model the system's core functionality. The abstract *User* class encapsulates common attributes like id, name, email, and password, along with essential methods such as login and logout. This base class is extended by *Driver* and *Passenger* classes; the Driver class includes additional properties such as license number and vehicle information, as well as methods to update location and confirm ride completion, while the Passenger class features a rating attribute and the ability to book rides. The *Admin* class is responsible for overseeing the system by accessing comprehensive driver and passenger details, as well as database records. The *Ride* class models a trip with attributes for ride identification, start and end locations, and status, and it is

closely linked to the *RideRequest* class, which captures the details of ride bookings made by passengers. Other integral classes include *Location*, which handles geographical coordinates and address retrieval; *Payment*, which processes transactions using various methods like Credit Card, Cash, or MasterCard; *RatingReview*, which allows for the submission of ride feedback; and *Database*, which manages data connections and queries, ensuring that sensitive data is viewable only by the Admin.

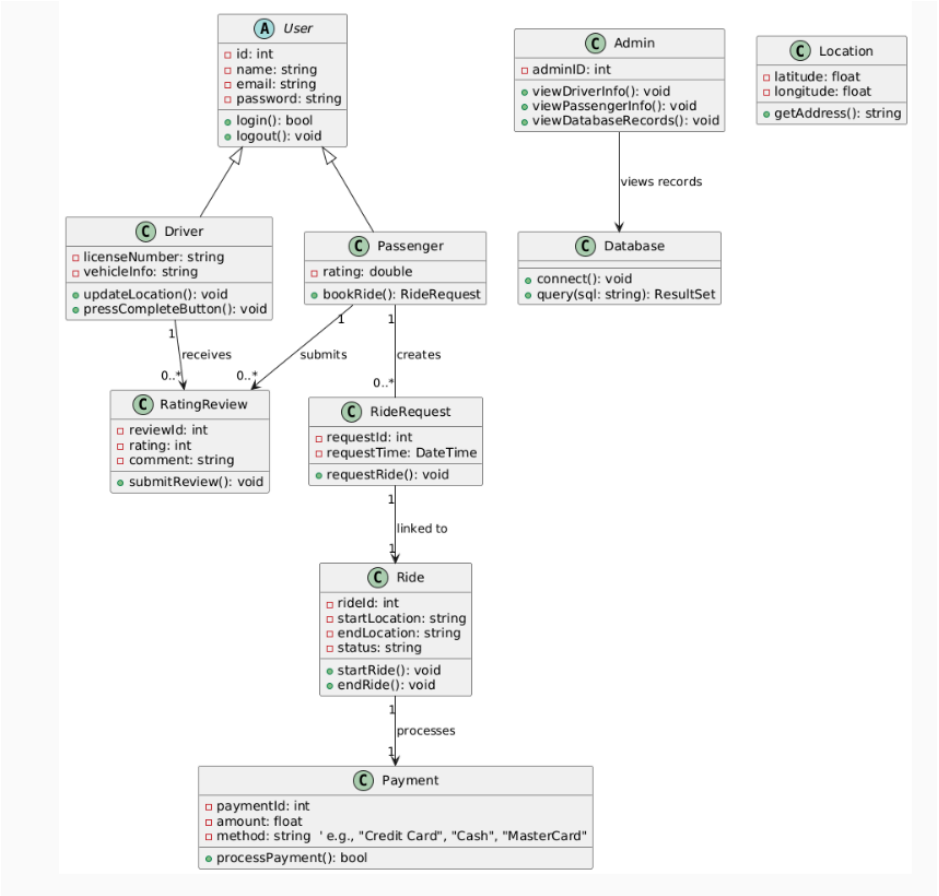

Figure 3 - Class diagram for SmartRide

The Payment Sequence Diagram in SmartRide outlines the step-by-step process for handling ride payments. In this diagram, the Passenger initiates a ride request and selects a payment method, prompting the RideRequestController to begin the payment process via the PaymentModule. Depending on the selected method—Credit Card, Cash, or MasterCard—the module requests the necessary payment details from the Passenger. Once the payment information is provided, the transaction is saved to the Database, and a confirmation is sent back to the controller, which in turn notifies the Driver. After the ride, the Driver confirms completion by pressing a complete button, leading the PaymentModule to update the ride status in the Database. Additionally, the Admin can access all stored records, ensuring full oversight, while the LocationService displays the current positions of both the Driver and Passenger.

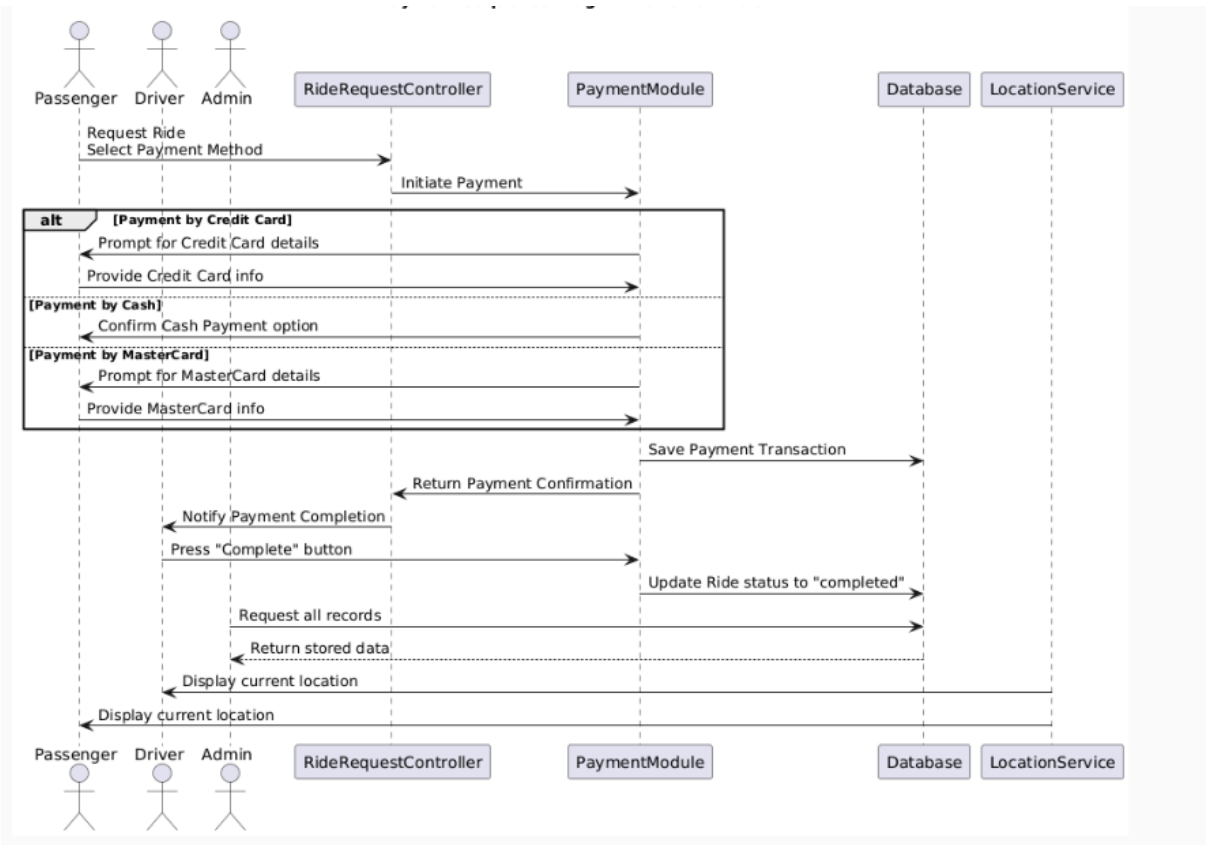

Figure 4 - Payment Sequence Diagram for SmartRide

2.1 Overall changes at class level

The updated design for SmartRide has been refined to clearly define the responsibilities of each candidate class while ensuring that their relationships accurately represent real-world interactions within the system.

Refined User Hierarchy and Relationships:

- The abstract *User* class now serves as the base for all user types. Its subclasses—*Passenger*, *Driver*, and *Admin*—inherit common attributes (id, name, email, password) and behaviors (login, logout).
- *Passenger* maintains a strong composition relationship with *RideRequest*, emphasizing that ride requests cannot exist independently of the passenger who creates them.
- *Driver* is associated with *Vehicle* and *RatingReview*, representing that drivers manage their vehicle details and receive independent ratings from passengers.

Payment Structure and Method Handling:

- The *Payment* class interacts with *PaymentMethod* in an aggregation relationship. Instead of inheriting payment methods, *Payment* references a specific *PaymentMethod* (e.g., CreditCard, EWallet, or BankTransfer) to handle transactions.
- This design reflects that while the payment process is centrally managed, the methods used for payment are independent components that can be modified or replaced without affecting the overall *Payment* class.
- Additionally, *Payment* is directly associated with *Ride*, ensuring that payment details are linked to ride transactions and financial records.

Ride and Request Dynamics:

- *RideRequest* and *Ride* are now explicitly separated. A *RideRequest* represents the initial booking with essential details such as pickup and drop-off locations and request time, whereas a *Ride* is created once a driver accepts the request.
- *Ride* is associated with both *Passenger* and *Driver*, reflecting the collaborative nature of the trip. It also maintains associations with *Vehicle*, *Payment,* and *RatingReview*, enabling independent management of each aspect of the ride.
- *Ride* interacts with *PricingStrategy* to calculate fares dynamically, allowing fare computations to be modified independently of the ride logic.

Supporting Service Relationships:

- Core service classes such as *LocationService* and *PricingStrategy* are treated as independent components that are aggregated or associated with relevant domain classes.
- For example, *LocationService* provides real-time tracking data used by *RideRequest* and *Ride*, while *PricingStrategy* is engaged by these classes for dynamic fare calculation.
- *Database* remains the central persistence layer with associations to key classes like *User*, *RideRequest*, *Ride*, *Payment*, and *RatingReview,* emphasizing that these classes depend on the centralized database for data integrity and storage.

Admin and System Oversight:

- The *Admin* class is designed to aggregate system-wide services. It now has direct associations with *Database* to provide oversight and reporting capabilities, ensuring that all aspects of the system can be monitored and managed effectively.

This revised design, with the removed classes omitted, results in a leaner architecture where each candidate class has clearly defined responsibilities and relationships that accurately reflect their roles in the SmartRide system.

2.2 Responsibilities and Collaborators

a. User

• Responsibilities:

o Original: Manages basic user info (userID, name, phone number, email) and authenticates credentials.

o Updated: Manages user login credentials (username, password), supports login/logout, and authenticates via *AuthenticationManager.*

• Collaborators: No changed

b. Passenger

• Responsibilities:

o Original: Knows default payment method, creates and manages ride requests, views ride history, and leaves feedback.

o Updated: Creates and manages ride requests, views detailed ride summaries and fare breakdowns, processes payment, and confirms payment status.

• Collaborators:

o Original: RideRequest, Ride, RatingReview.

o Updated: RideRequest, Ride, Payment, RatingReview.

c. Driver

• Responsibilities:

o Original: Manages availability, updates vehicle information, accepts ride requests, and maintains ratings.

o Updated: Manages availability, updates vehicle information, accepts ride

requests, and confirms ride completion via a dedicated action (pressCompleteButton).

• Collaborators: No changed

d. Admin

• Responsibilities:

o Original: Manages user accounts, generates and reviews reports, and oversees overall system operations.

o Updated: Manages user accounts, reviews comprehensive driver and passenger information, accesses database records, and oversees system operations.

• Collaborators:

o Original: User, Report, AuthenticationManager.

o Updated: User, Database, AuthenticationManager.

e. RideRequest

• Responsibilities:

o Original: Stores pickup/drop-off locations, request time, and status; links to the passenger; interacts with PricingStrategy for fare estimates.

o Updated: Captures detailed booking data (including pickup/drop-off locations, request time, and fare estimation) and serves as the initial reservation linked to a specific passenger.

• Collaborators:

o Original: Passenger, PricingStrategy.

o Updated: Passenger, PricingStrategy, LocationService.

f. Ride

• Responsibilities:

o Original: Represents a confirmed trip, tracking assigned driver, passenger, vehicle, locations, ride status, and fare calculation; allows feedback submission.

o Updated: Represents a confirmed trip with defined start/end locations, explicit ride lifecycle methods (startRide, endRide), integrated payment processing, and supports feedback collection.

• Collaborators:

o Original: Driver, Passenger, Vehicle, Location, PricingStrategy, Payment,

RatingReview.

o Updated: Driver, Passenger, Vehicle, Payment, RatingReview, NotificationService, PricingStrategy.

g. Vehicle

• Responsibilities:

o Original: Holds vehicle information (make, model, type) and associates with a driver.

o Updated: Maintains and updates vehicle details (make, model, type).

• Collaborators: No changed

h. Payment

• Responsibilities:

o Original: Records payment transaction details (paymentID, amount, status), processes payment, and verifies payment.

o Updated: Represents a generalized concept for various payment types, settles fees, processes payment, notifies on successful transactions, and stores payment details.

• Collaborators: No changed

i. RatingReview

• Responsibilities:

o Original: Records feedback (rating and comment) from a passenger to a driver.

o Updated: Records ride feedback with a rating and review, optionally linked to a specific ride.

• Collaborators: No changed

j. Location

• Responsibilities:

o Original: Stores coordinates and address information for navigation.

o Updated: Maintains geographical data (latitude, longitude) and provides a method to retrieve a formatted address.

• Collaborators: No changed

k. Database

• Responsibilities:

o Original: Acts as the central persistence layer, storing, retrieving, and updating system data.

o Updated: Serves as the core data management component, providing connection and query functionality for users, rides, payments, etc.

• Collaborators: No changed

l. AuthenticationManager

• Responsibilities:

o Original: Handles user authentication, manages login/logout, and maintains sessions.

o Updated: Manages user authentication and secure session control, verifies credentials, and enforces role-based access.

• Collaborators: No changed

2.3 Changes to dynamic aspects (bootstrap, interaction scenarios)

a. User and Role Initialization:

• Instances of the abstract *User* class are created, along with its derived classes—*Passenger*, *Driver*, and *Admin*—to handle user management. These instances manage authentication, access control, and role-specific permissions within the system.

• *Note: Passenger*, *Driver*, and *Admin* inherit common functionality from *User*.

b. Ride and Request Setup:

• When a *Passenger* initiates a booking, a *RideRequest* instance is created that captures pickup/drop-off details, request time, and fare estimation.

• Once a *Driver* accepts the request, a Ride instance is instantiated to represent the confirmed trip, linking the *Passenger*, *Driver*, and relevant ride details.

• *Note:* The strong composition relationship between *Passenger* and *RideRequest* ensures that ride requests cannot exist without an originating *Passenger*.

c. Payment Processing Initialization:

• After ride completion, a Payment instance is generated to process the transaction based on the selected *PaymentMethod* (e.g., CreditCard, EWallet, BankTransfer).

• The *Payment* module manages fee settlement, confirms payment status, and notifies the system upon successful transaction.

• *Note:* Payment interacts with *PaymentMethod* through an aggregation relationship, providing flexibility for handling different payment types.

d. Supporting Services Initialization:

• Core services such as *Database*, *LocationService*, and *PricingStrategy* are initialized to ensure data persistence, real-time navigation, and dynamic fare calculation.

• *Note:* The *Database* acts as a singleton to maintain data integrity, while *LocationService* and *PricingStrategy* operate as independent modules that support the dynamic aspects of ride booking.

e. Dynamic Interaction Scenarios:

• Following ride confirmation, the *Driver* completes the ride using a dedicated action (pressCompleteButton), triggering the *Payment* process.

• Upon successful payment, the system prompts for feedback through the *RatingReview* component, enabling passengers to rate and review their ride experience.

• *Note:* These interactions reflect real-time communication between components, ensuring that each phase of the ride—from booking to payment and feedback—is seamlessly managed

3, Design Quality

3.1 Good aspect

In our initial design for SmartRide, these are several strong aspects contributed to its overall quality:

- Modular Class Structure: Your design includes distinct classes like User, Driver, Passenger, Ride, Payment, Vehicle, etc., which aligns well with separation of concerns — each class having its own responsibility.
- Defined Interactions: The system defines flows like booking a ride, registering, making a payment, etc., and reflects this in both frontend and backend implementations — similar to the good bootstrap and scenario-based testing in the reference.
- Mapping from Design to Code: Your implementation matches many class responsibilities (like Ride, Payment, and NotificationService) with actual code

logic. This follows the principle of “mapping design to code” praised in the reference PDF.

3.2 Missing aspects

Despite the strengths, these are some crucial aspects were missing in our initial design:

- Lack of class relationships: The design did not clearly show how key classes are connected. For example, there should be clear links between Ride and Invoice, between Payment and Receipt, and between Driver and Vehicle.
- Missing interaction details: While we had classes listed, we did not fully describe how they work together. It would be better to include diagrams or descriptions showing how one class uses another.
- No CRC cards: We did not prepare Class Responsibility Collaborator (CRC) cards. These would help explain what each class does and who it works with.

3.3 Error aspects of the original design

Several errors were identified in the initial design, which required significant modifications:

- Wrong use of inheritance: Some classes used inheritance when it was not needed. For example, different payment methods could be handled better with composition or separate strategies instead of class inheritance.
- Unclear or missing UML arrows: The design diagrams did not always use correct arrows to show relationships. This made it harder to understand the structure and flow between classes.
- Extra or confusing classes: Some classes like NotificationService were too simple or generic and might not need to be separate. They could be merged or simplified to avoid confusion.

3.4 Level of interpretation

Our design in Assignment 2 has notably poor interpretability. As was previously said, the main cause of this problem is our UML diagram's improper usage of connection arrows. The links between classes in our original design are muddled by this usage, making it difficult to understand. The design should have been simple to describe because of the strong underlying idea, even with the connection arrows issue. Decomposing the original domain entities into components and then abstracting related functionality into higher-level classes formed the foundation of the design. As a result, the design needs to have been simple and intuitive.

3.5 Lesson learnt

After completing this task, our team has also developed a deeper understanding of how to structure class responsibilities effectively. One of the key takeaways is the importance of cohesion within each class. We learned that each class should be responsible for a single, well-defined functionality, which helps reduce confusion and improves maintainability. Additionally, we recognized the value of proper naming conventions. Clear and descriptive class and method names greatly enhance readability, especially when the project scales. Lastly, we now understand that isolating reusable logic into utility or service classes can significantly reduce code duplication and make the overall architecture more modular and scalable.

4, Implementation and Testing

During the Assignment 3 implementation stage, we create the SmartRide system as a web-based application with four primary functionalities: establishing user accounts, booking rides, processing payments, and gathering ride feedback. To guarantee that these four essential operating areas work together flawlessly, we reduced the design so that each is entirely suited for the ride-sharing environment.

4.1 Mapping design to code

During the assignment 3 implementation stage, we create a web-based application with four key functionalities: establishing client accounts, reservations, bills, and receipts.
To effectively manage the system's four operational domains, we streamlined the design and ensured complete functionality.

Design	Code Implementation
User, Passenger, Driver	- **Files:** • *`controllers/register_controller.php`* (handles user registration and role selection) • *`controllers/login_controller.php`* (manages login, session handling) • *`passenger_dashboard.php, driver_dashboard.php`* (UI pages for each role) - **Description:** These files implement user registration,

	role-based login, and dashboard functionality. They validate inputs (e.g., name, email, password) and interact with the *Database* class to store and retrieve user data.
RideRequest	- **Files:** • `book_a_ride.php` (presents a form to book a ride, including pickup/drop-off locations) • `controllers/ride_controller.php` (handles creation and validation of ride requests) - **Description:** The `book_a_ride.php` page collects ride details (pickup, destination, time) from passengers. `ride_controller.php` creates a `RideRequest` record in the database and performs checks such as availability or pricing.
Ride	- **Files:** • `models/Ride.php` (defines the Ride class attributes and methods) • `controllers/ride_controller.php` (assigns driver, updates ride status) • `passenger_dashboard.php` / `driver_dashboard.php` (UI to view current rides) - **Description:** The `Ride` class in `models/Ride.php` tracks assigned driver, passenger, location, and status. The `ride_controller.php` updates ride progress (started, in-progress, completed) and manages real-time status changes.
Payment	- **Files:** • `models/Payment.php` (defines attributes like paymentId, amount, status, and methods like `processPayment()`)

	• *controllers/payment_controller.php* (coordinates payment logic, including reading user input for payment method) • *database/smartride.sql* (tables for storing payment transactions) - **Description:** The *Payment* model encapsulates transaction details and processing logic. *payment_controller.php* ensures the selected payment method is charged, updates the payment status, and returns success or error messages.
PaymentMethod (e.g., CreditCard, EWallet, BankTransfer)	- **Files:** • *models/PaymentMethod.php* (abstract or interface defining the contract for different payment methods) • *models/CreditCard.php*, *models/EWallet.php*, *models/BankTransfer.php* (concrete implementations, if still used) - **Description:** These classes detail how each payment method is processed. In some cases, you might have merged them into a single method field in *Payment*—if so, only a single *Payment* class or method field would appear in the code.
RatingReview	- **Files:** • *models/RatingReview.php* (class definition storing ratingID, rating score, comment) • *controllers/rating_controller.php* (handles saving passenger reviews, retrieving driver ratings) • *driver_dashboard.php* (displays driver's average rating) - **Description:** The *RatingReview* model captures and stores feedback from passengers to

	drivers, and the controller ensures reviews are inserted or retrieved from the database.
Location, LocationService	- **Files:** • *`models/Location.php`* (stores latitude, longitude, address) • *`controllers/location_service.php`* or integrated code in *`ride_controller.php`* (fetches map data, calculates distance/ETA) - **Description:** The *`Location`* model represents geographic data. *`location_service.php`* (or integrated logic) might call external APIs (e.g., Google Maps) to compute distances and provide route info for rides.
PricingStrategy	- **Files:** • *`models/PricingStrategy.php`* (defines *`calculateFare()`* logic) • *`controllers/ride_controller.php`* (applies surge pricing or discount rates) - **Description:** The *`PricingStrategy`* model encapsulates fare calculation. The ride controller leverages it to compute or update ride costs dynamically based on distance, time, or demand.
Database	- **Files:** • *`models/Database.php`* (class handling DB connection, queries) • *`database/smartride.sql`* (schema for tables like *`users, rides, payments, reviews`*) - **Description:** The *`Database`* class in *`models/Database.php`* manages

	connections (possibly via Singleton), executes queries, and returns results. The `.sql` file initializes the database structure.
Admin	- **Files:** • `models/Admin.php` (class definition if distinct from User) • `admin_dashboard.php` (webpage to view system metrics, user accounts, ride records) • `controllers/admin_controller.php` (manages admin-specific actions like user management or reporting) - **Description:** The `Admin` class extends `User` and has elevated privileges. The admin dashboard provides an interface to monitor drivers, passengers, rides, and payments, often querying the database or calling relevant controllers.

4.2 Compilation and Execution

You can find our project via GitHub:
https://github.com/duyhung-0203/swe30003.git

The program is built on XAMPP, using MySQL for database administration and Apache for server hosting. Visual Studio Code is used for development, with PHP and JavaScript providing object-oriented programming (OOP) for backend logic and interaction. CSS and the Bootstrap toolkit are used to create a responsive and contemporary user interface that offers a seamless experience.
To compile and run source code, first install XAMPP and start Apache and MySQL services.

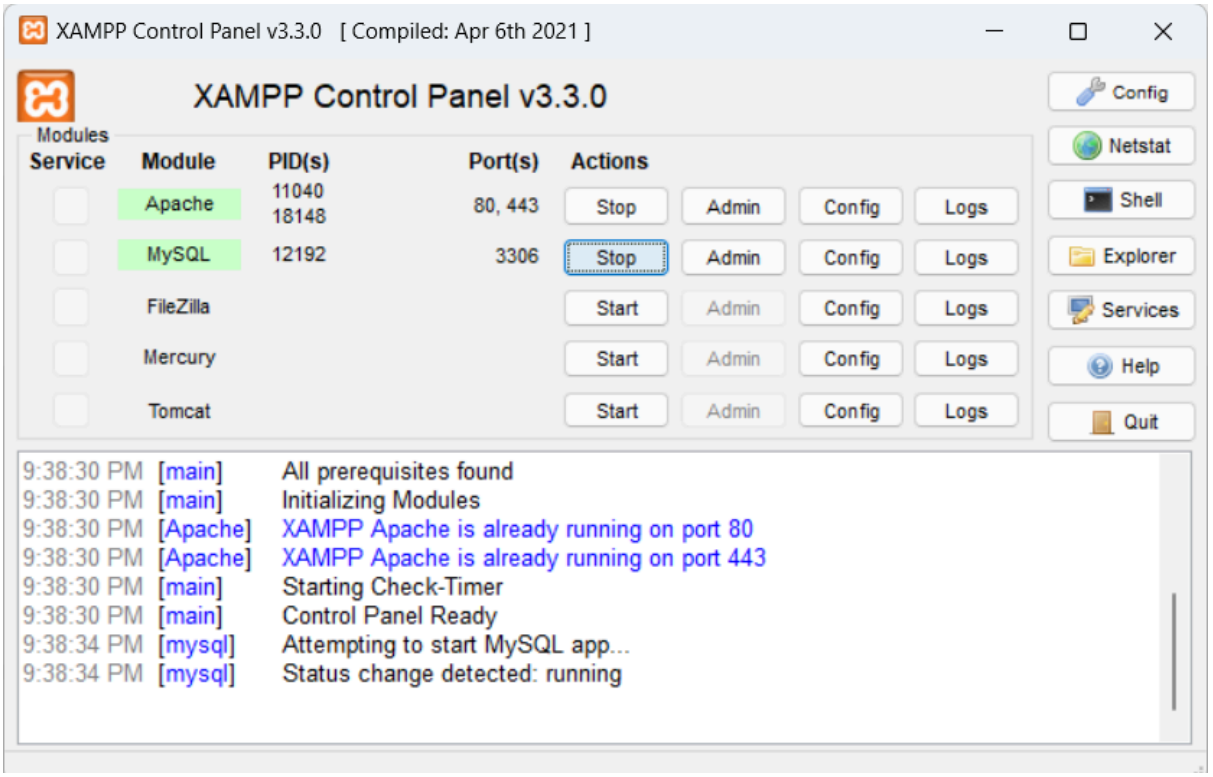

Figure 5 - Running localhost successfully

We saved the source code, including PHP, JavaScript, CSS files in the assets folder, to the htdocs directory of our XAMPP installation.

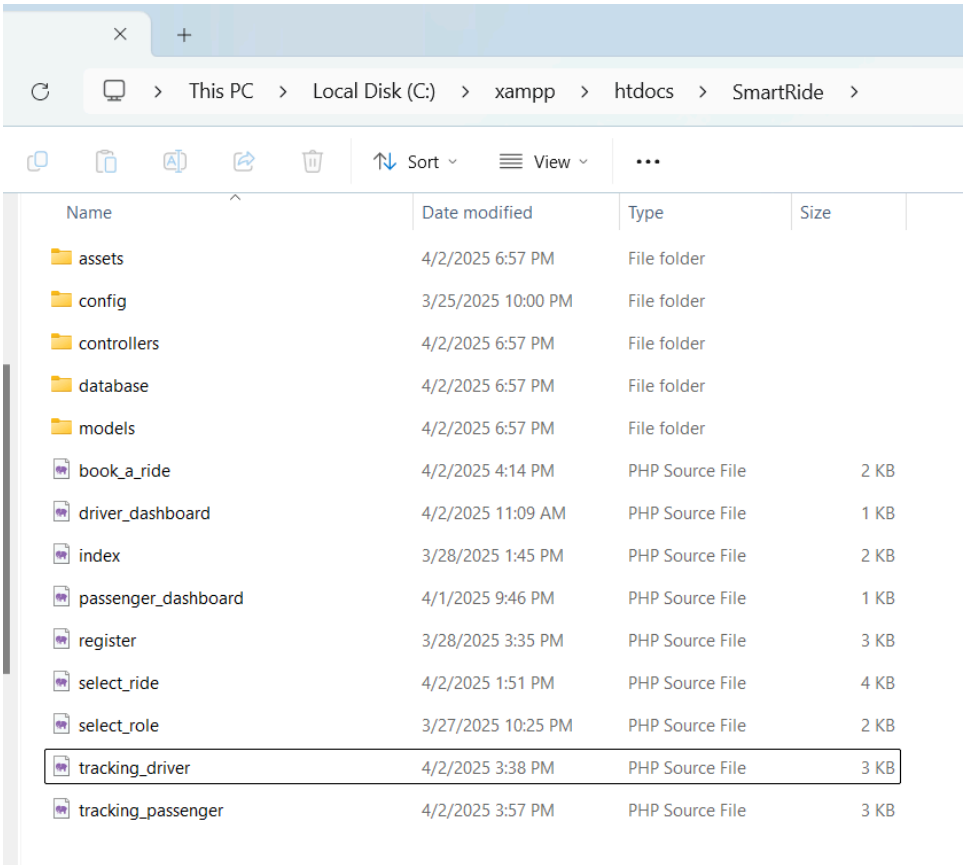

Figure 6 - Running localhost successfully

Before using the app, you must register as a member if you don’t already have an account, and select whether you are a driver or a customer using the service.

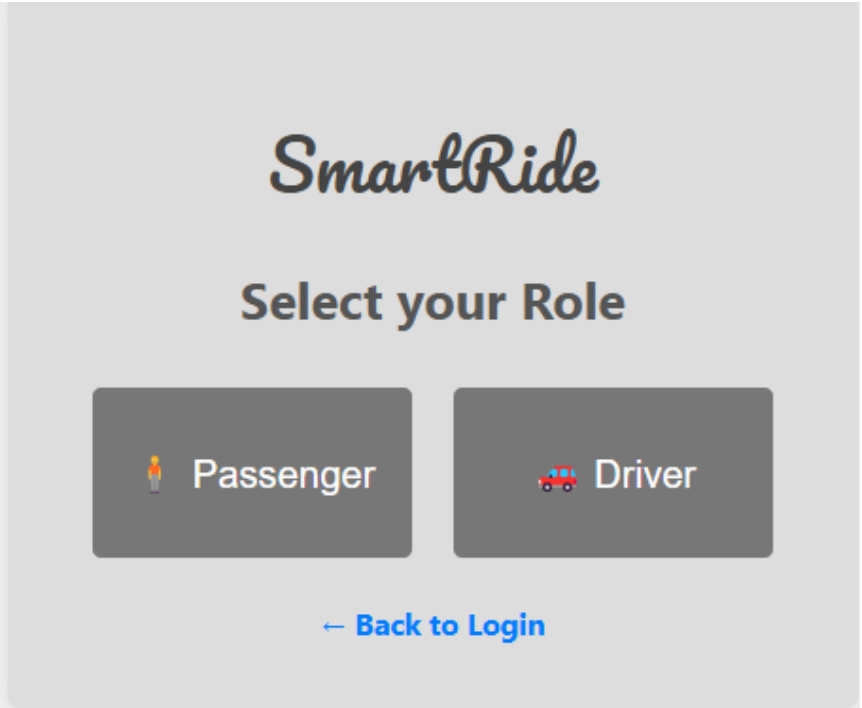

Figure 7 - Select the role

After choosing your role, you will be directed to the member registration section where you fill in your name, email, and password; if your role is driver, you must verify that you have a valid driver's license.

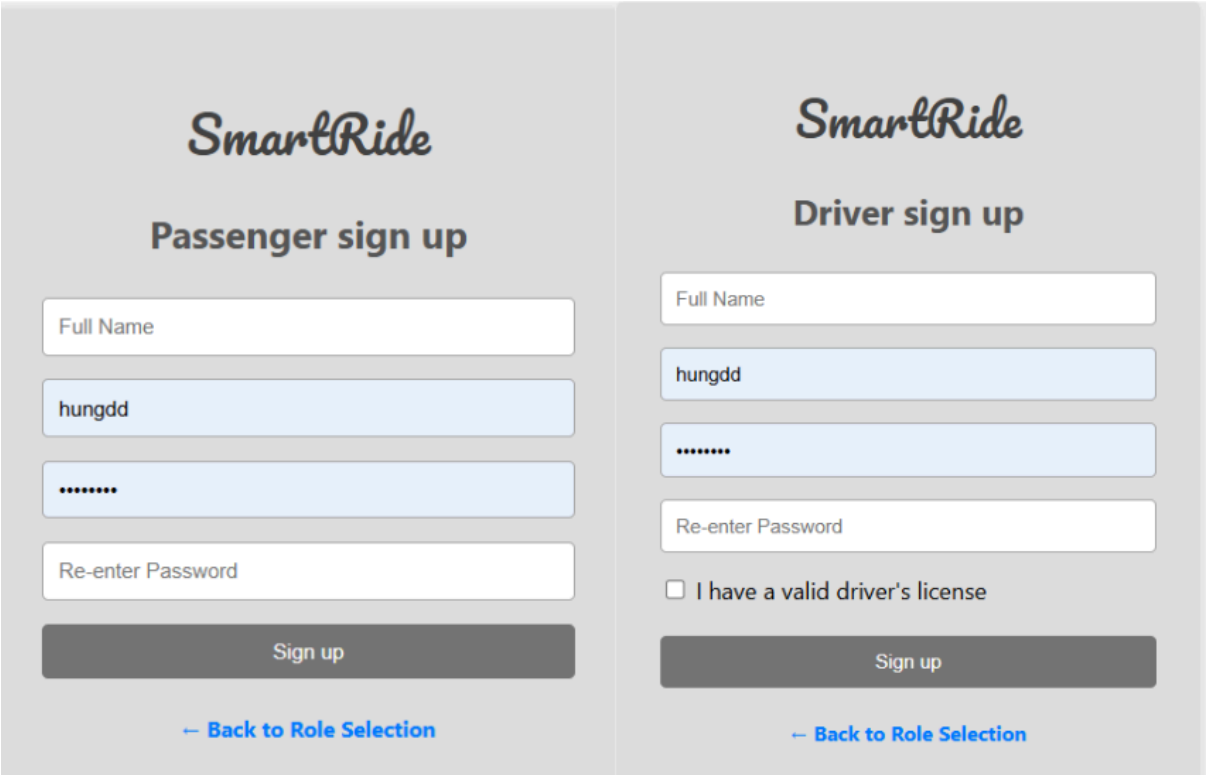

Figure 8 - Sign up in 2 different role

After that, log in with your account, and my dashboard is shown in the Testing part.

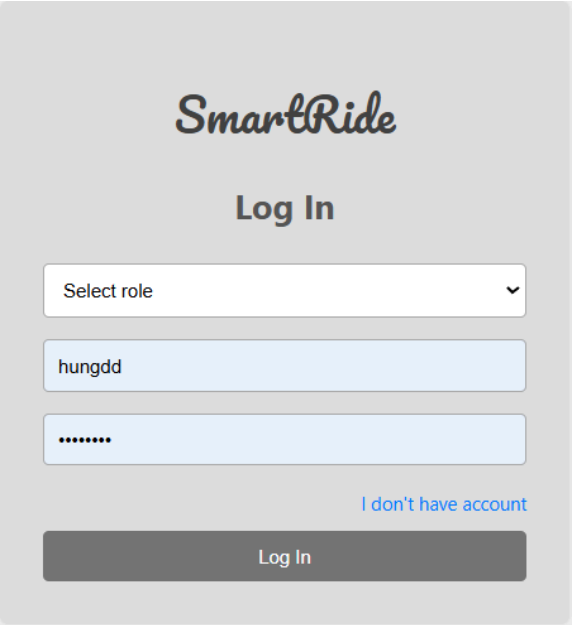

Figure 9 - Log in to account

4.3 Testing

Scenario 1: Creating driver/passenger account:

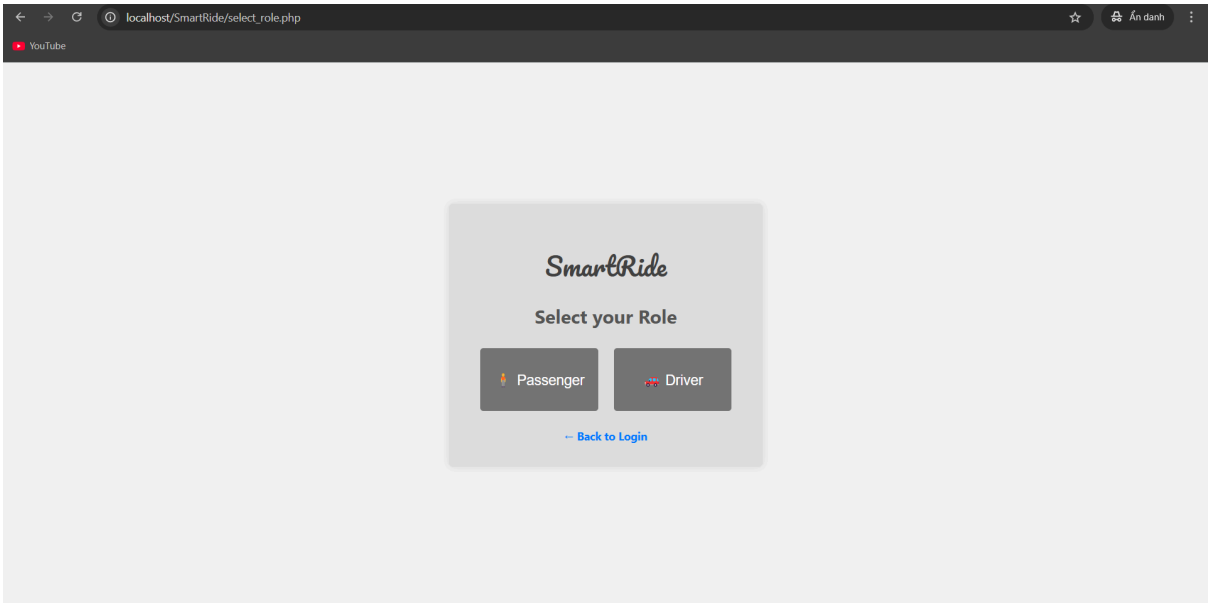

Figure 10 - Role selection UI

This is a screen for users to select roles to register with 2 option passenger and driver.

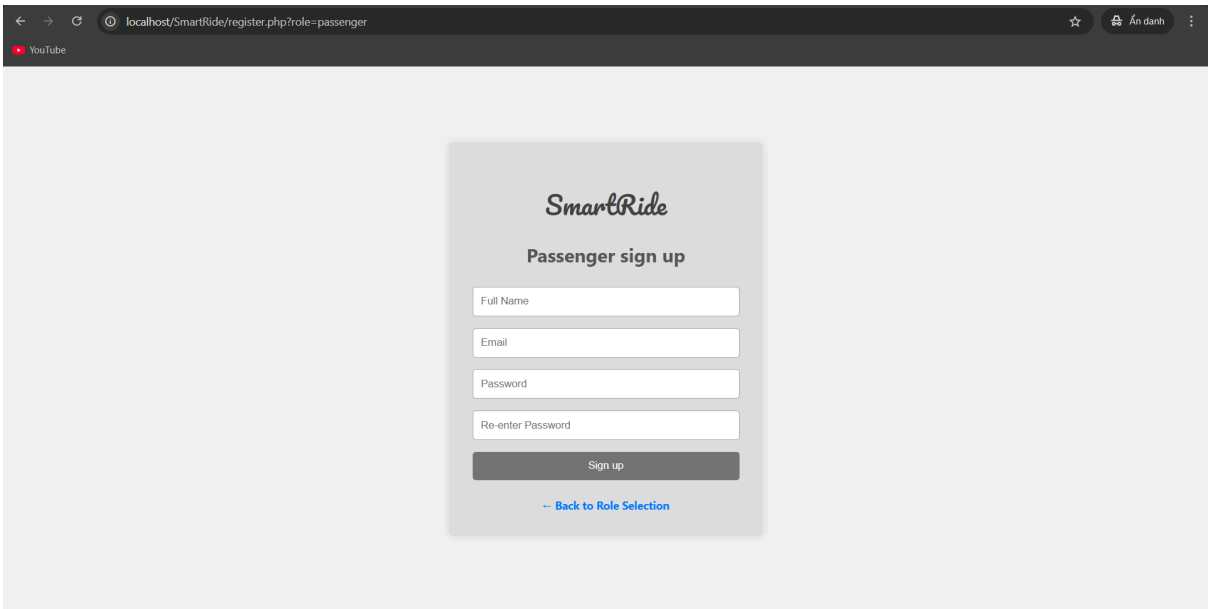

Figure 11 - Passenger signup

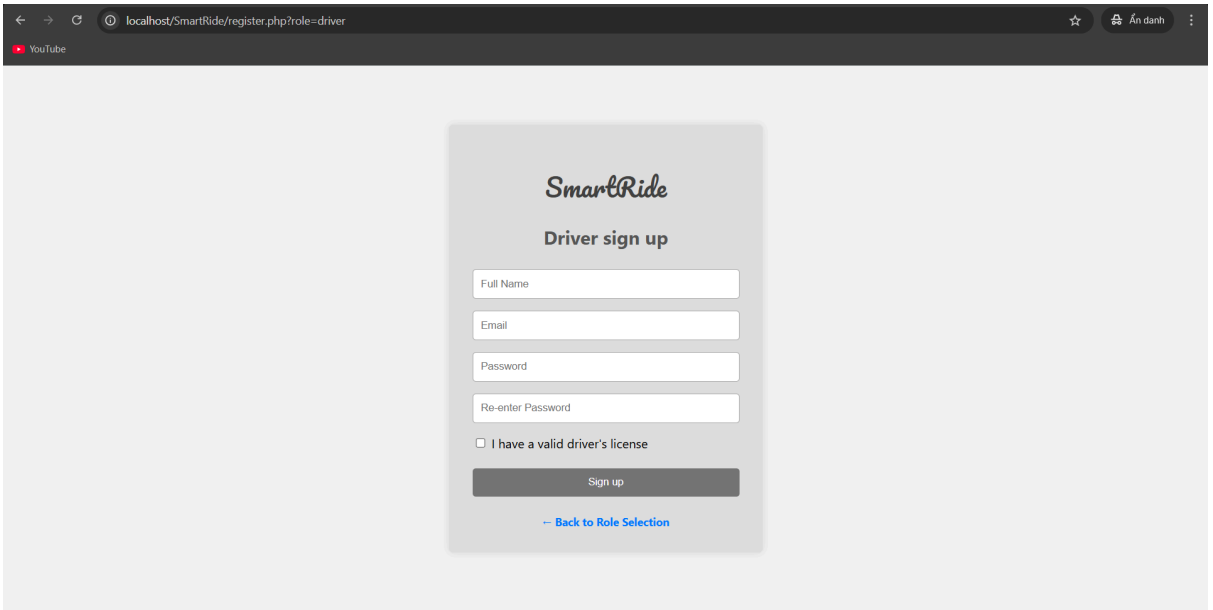

FIgure 12 - Driver signup

This is the passenger and driver signup UI. Users need to fill in their full name, email, password and re-password and also tick to confirm having a driver's license (in driver signup).

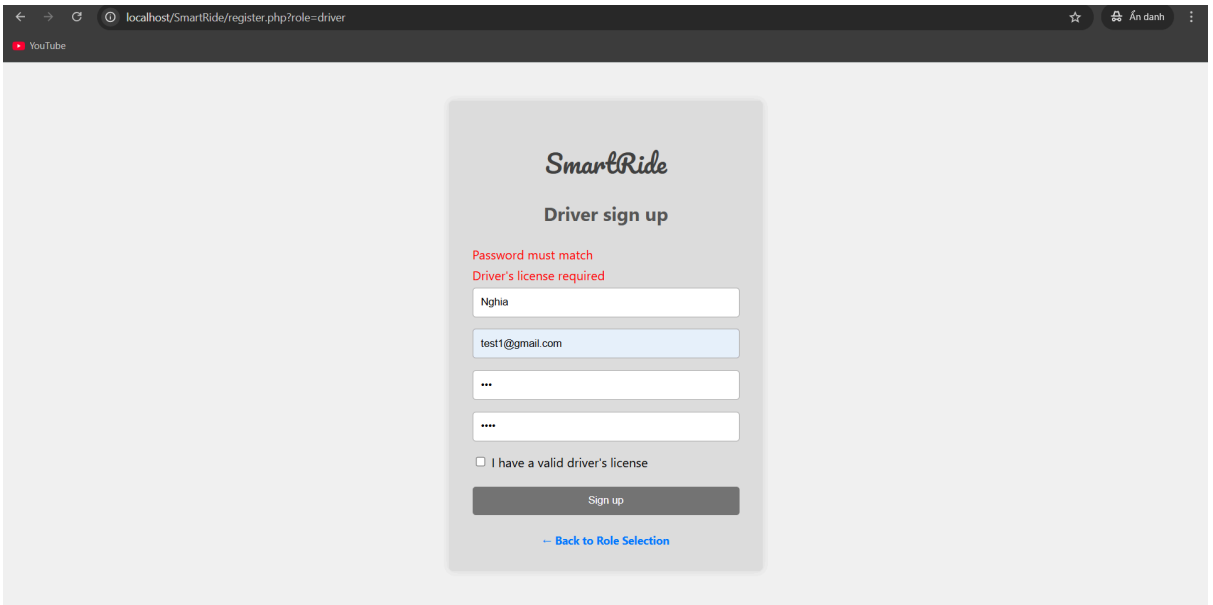

FIgure 13 - Input error

Users fill in incorrect input leading to error notification.

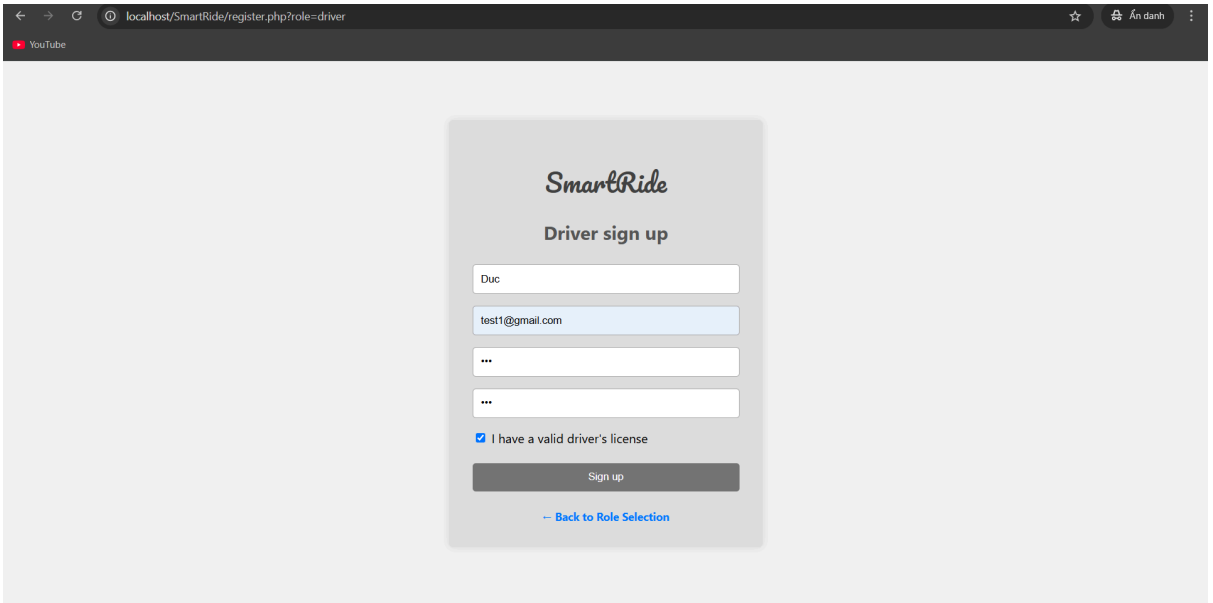

FIgure 14 - Example of an correct signup input

Users fill in correct information (with a non-existent email), then it will automatically navigate to index.php (login page)

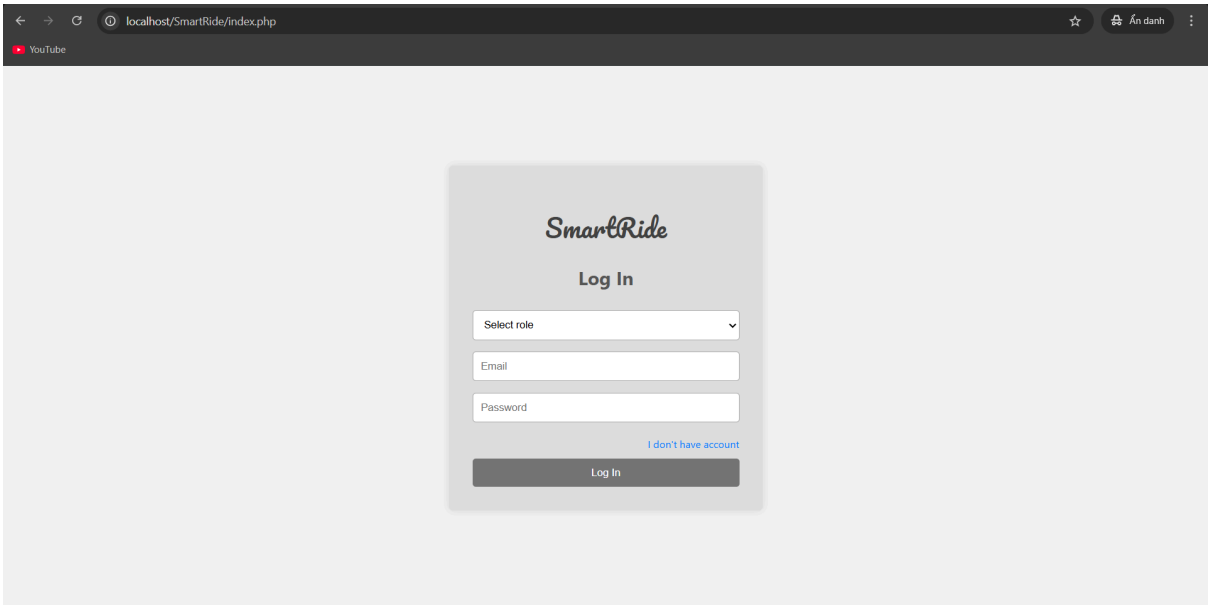

FIgure 15 - Log in

After logging in as passenger or driver through the form, it will verify the data from the database.

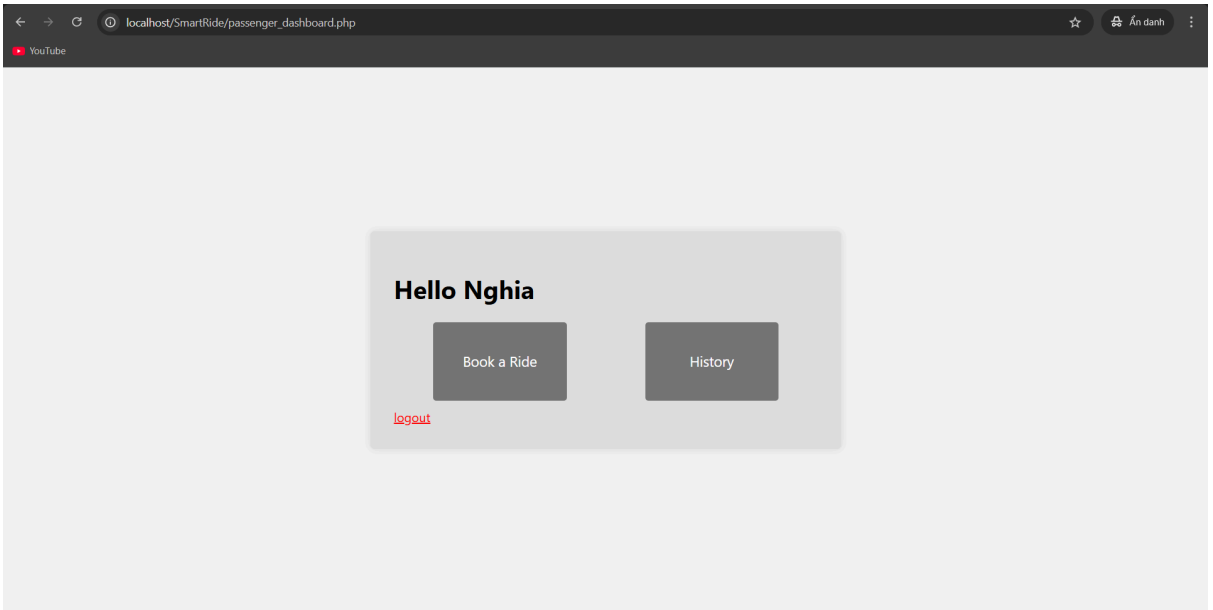

FIgure 16 - Passenger Dashboard

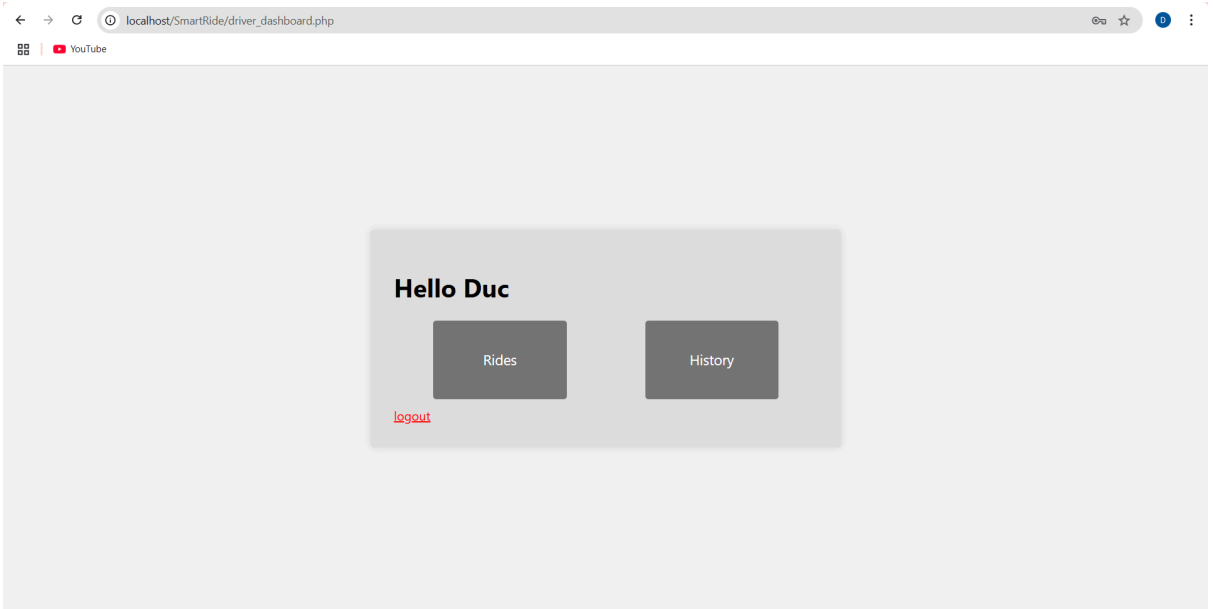

FIgure 17 - Driver Dashboard

Scenario 2: Book a ride:

After selecting “Book a ride” from the passenger dashboard, now users can see a map with 2 inputs of enter start location and end location.

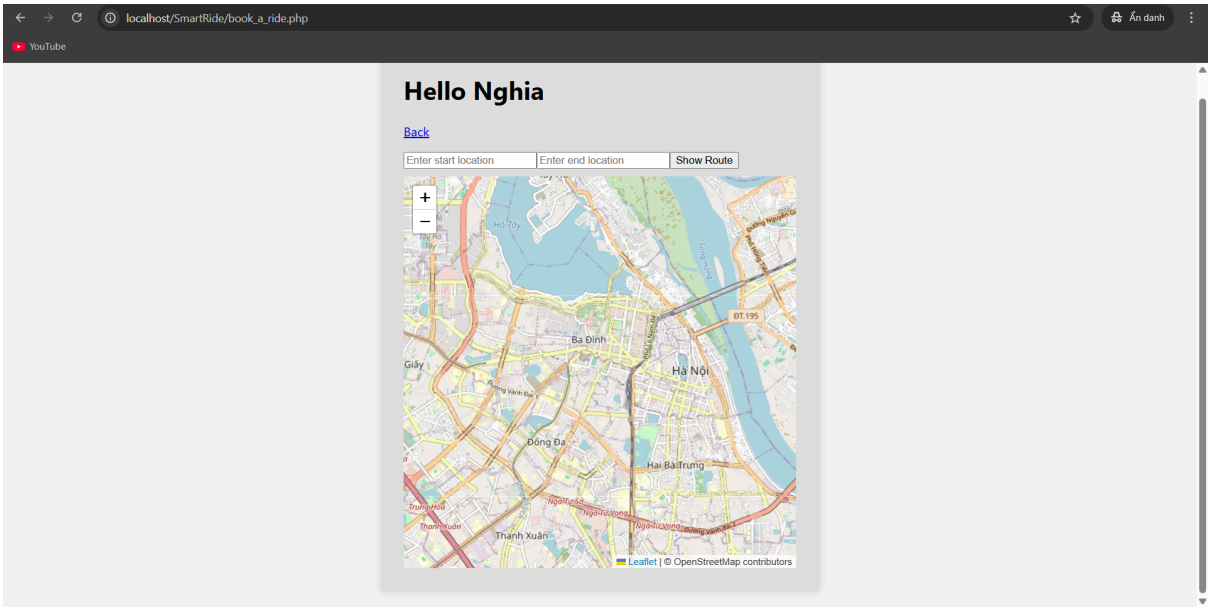

FIgure 18 - Passenger book ride

Error occurs when leaving blank inputs.

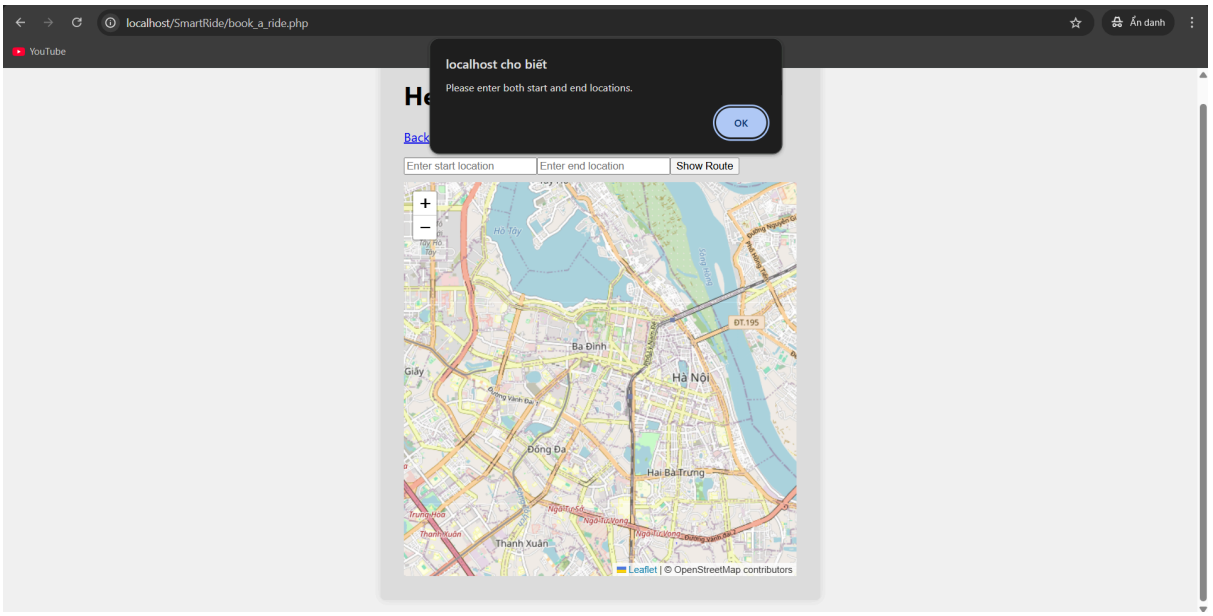

FIgure 19 - Error when not inputting

Users enter start location and end location, then click show results → a route that contains start point and end point showing in the map, along with distance and fare.

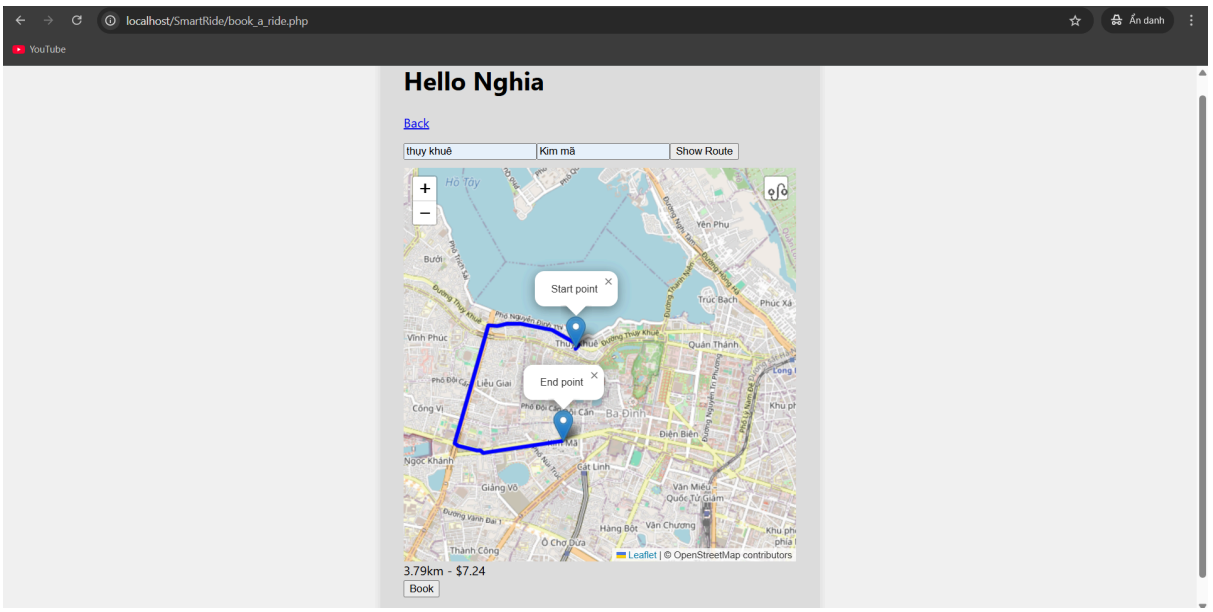

FIgure 20 - Show route

Users can also select the route guideline on the top right of the map.

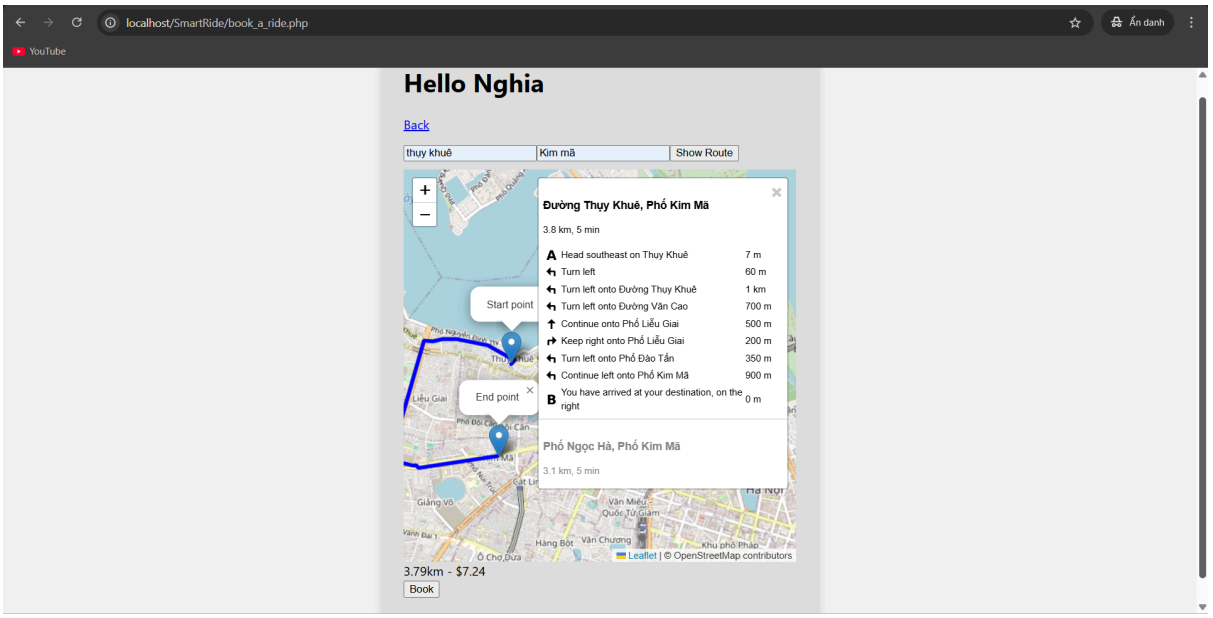

FIgure 21 - Route guideline

When passengers are ready to book the ride, click the book button then a status of “Finding driver….” will occur. For now, passengers just wait for the driver.

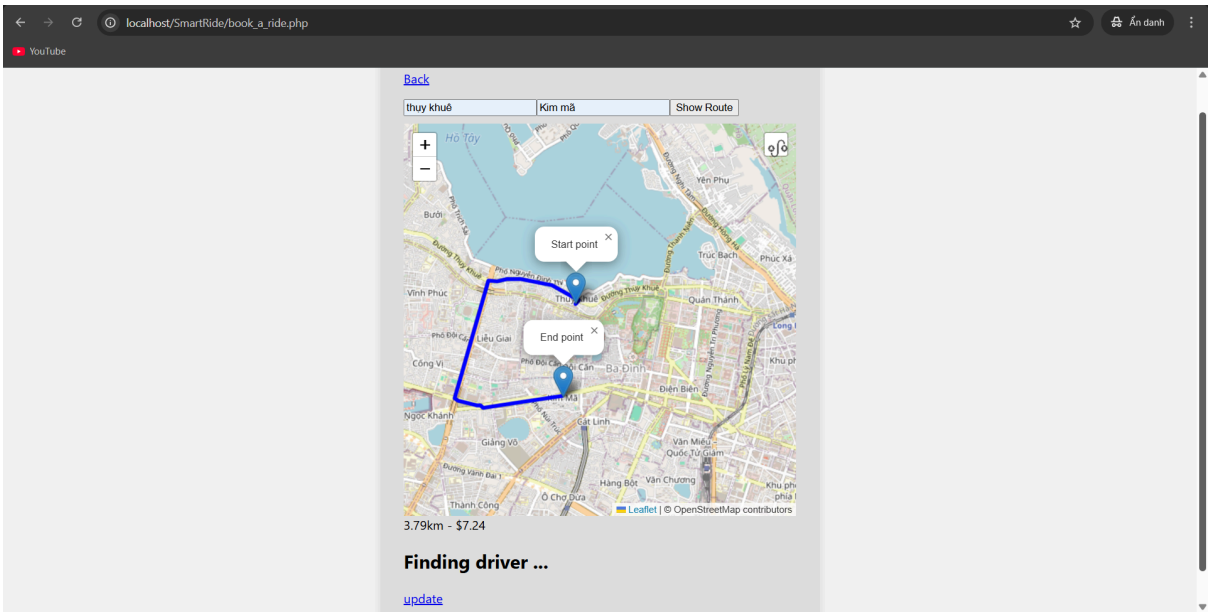

FIgure 22 - Finding driver

Scenario 3: Confirm a ride:

Since the passenger has booked a ride, so now in driver side, it will show ride information like passenger name, pickup address, dropoff address, fare, distance, and an accept ride button.

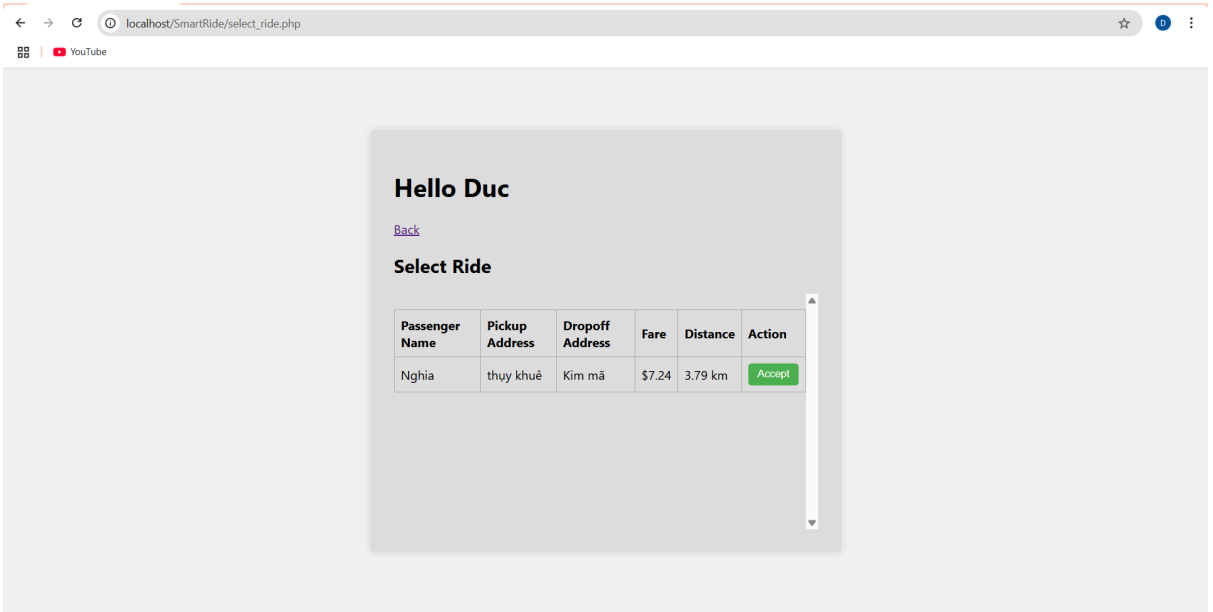

FIgure 23 - Accept ride

Driver has accepted the ride then now it will show a screen that contains a button to show the route.

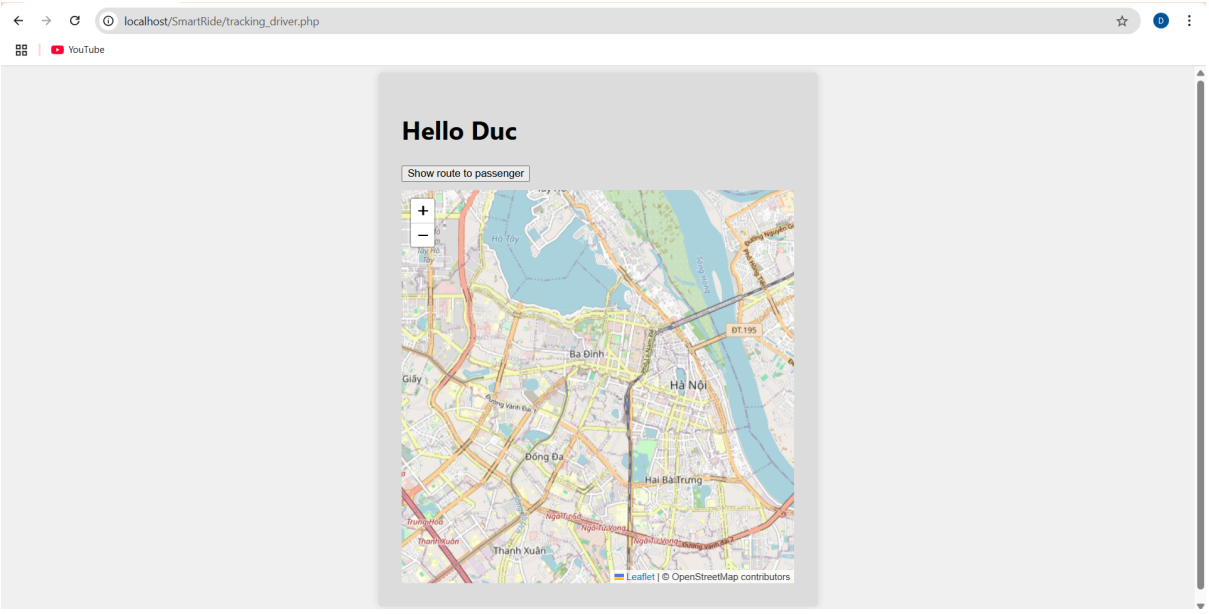

FIgure 24 - Start tracking

After clicking the show route, it will present 3 points which are start, end, and driver location, along with 2 lines which are the route from driver to start point (red), and route from start point to end point (blue). Also it shows the distance and fare

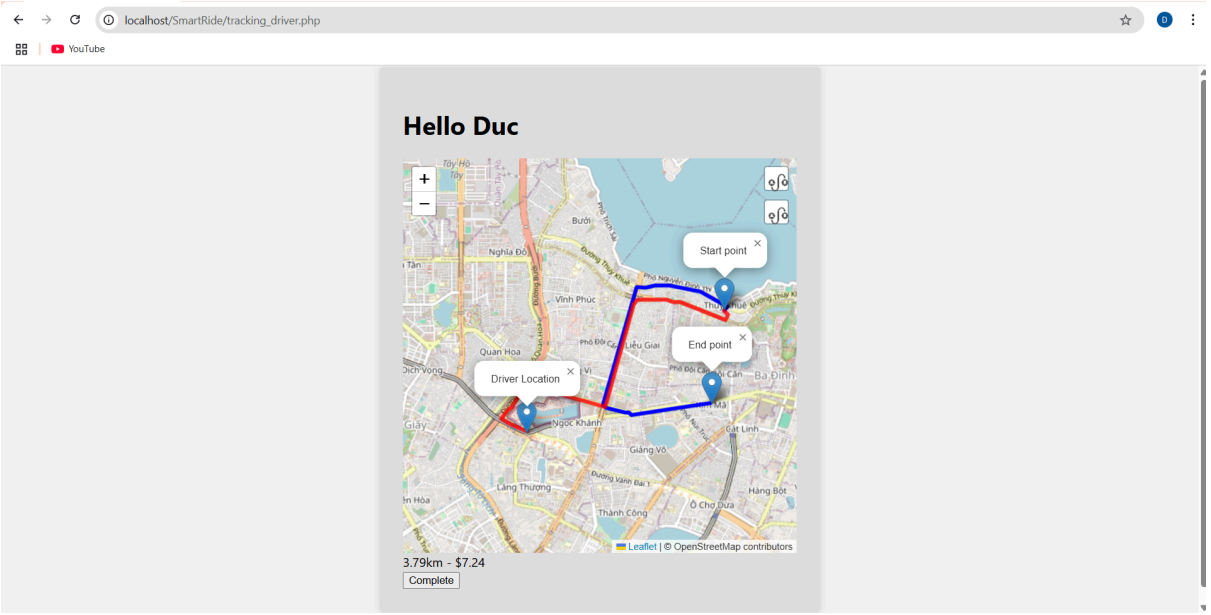

FIgure 25 - show route

The ride keeps continuing until drivers pick passengers up to the end point. Now drivers click the complete button then ask passengers for payment.

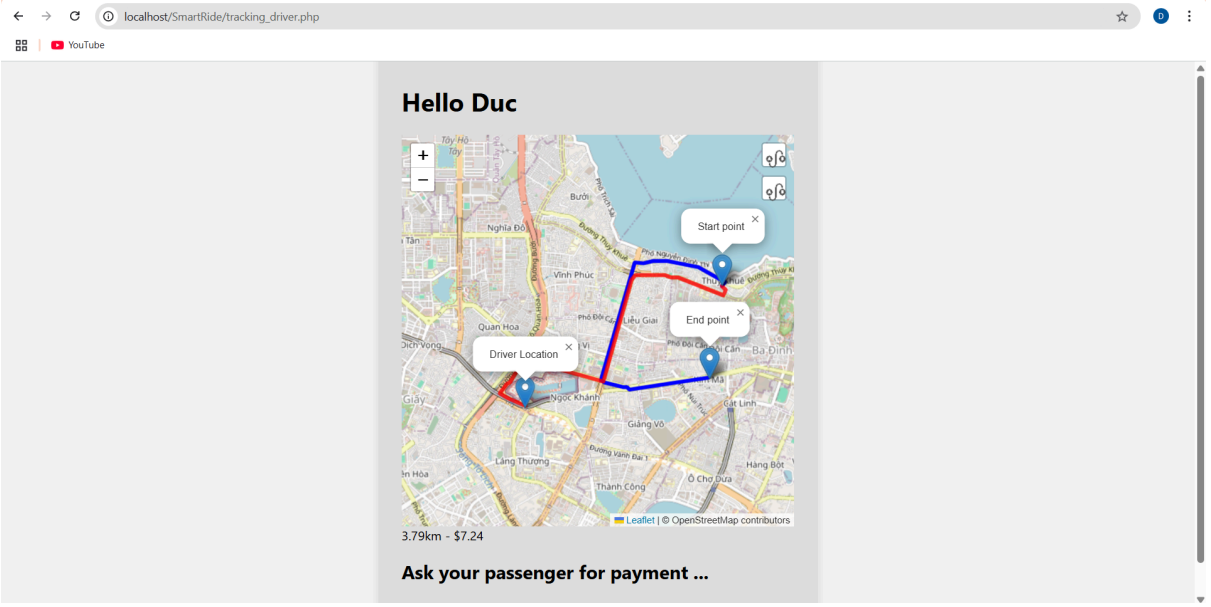

FIgure 26 - Complete ride

Scenario 4: Wait for driver and end ride:

After successfully finding a driver, passengers click update.

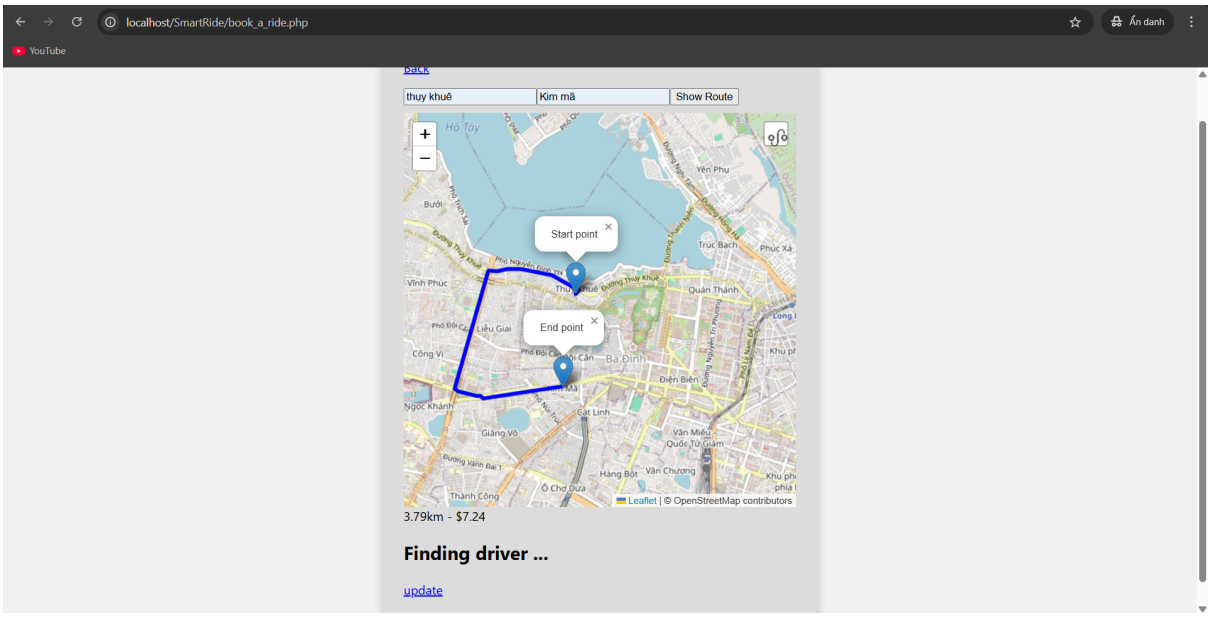

FIgure 27 - Update

At this screen, passengers can track the driver route via the Track driver button.

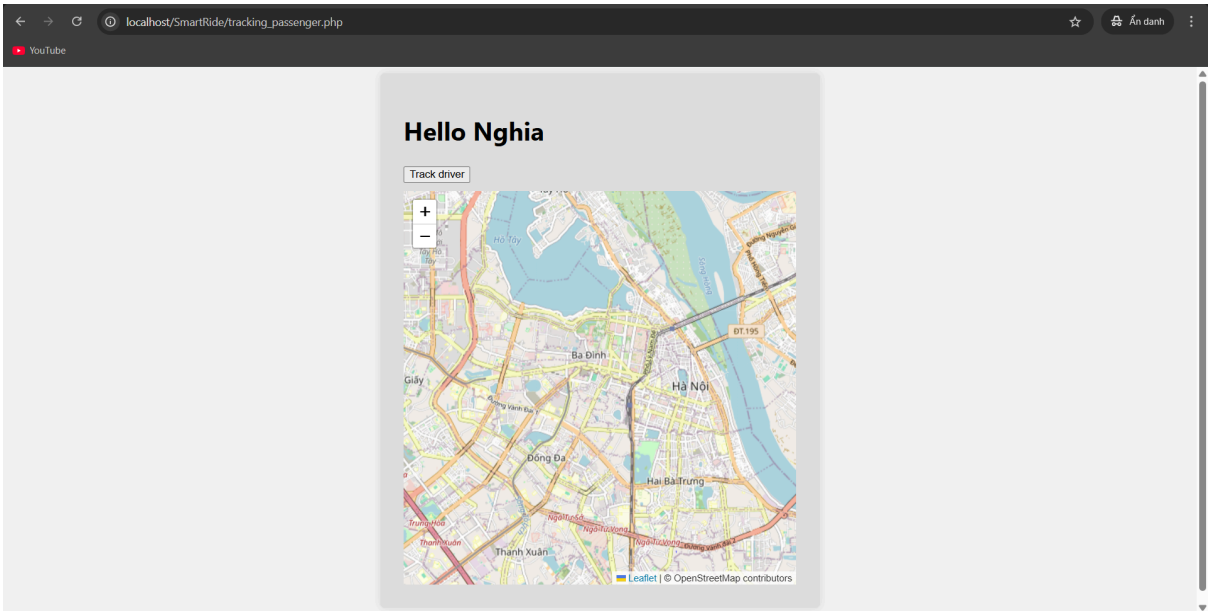

FIgure 28 - Complete ride

The screen will show a map that contains the driver route (red) and start to end point route (blue). Down below is the distance and fare, along with status “Waiting for driver…” which means waiting for the driver to come pick passengers up.

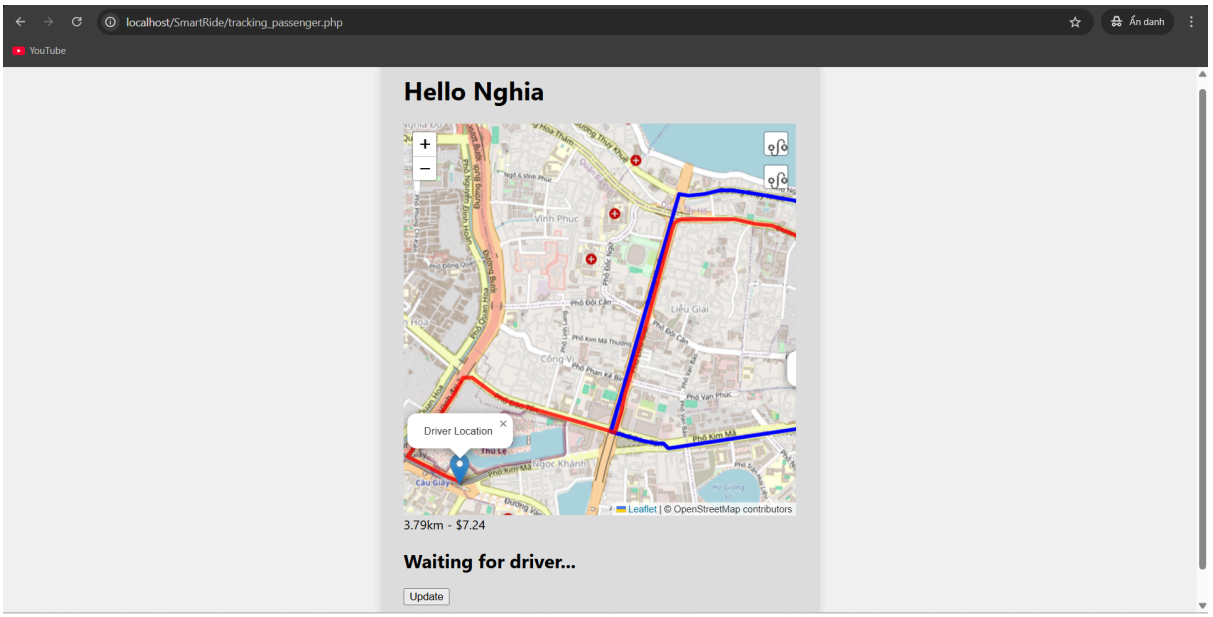

FIgure 29 - Complete ride

After drivers click on the complete ride, then users click the update to update the status. In this screen it's time for passengers to make a payment.

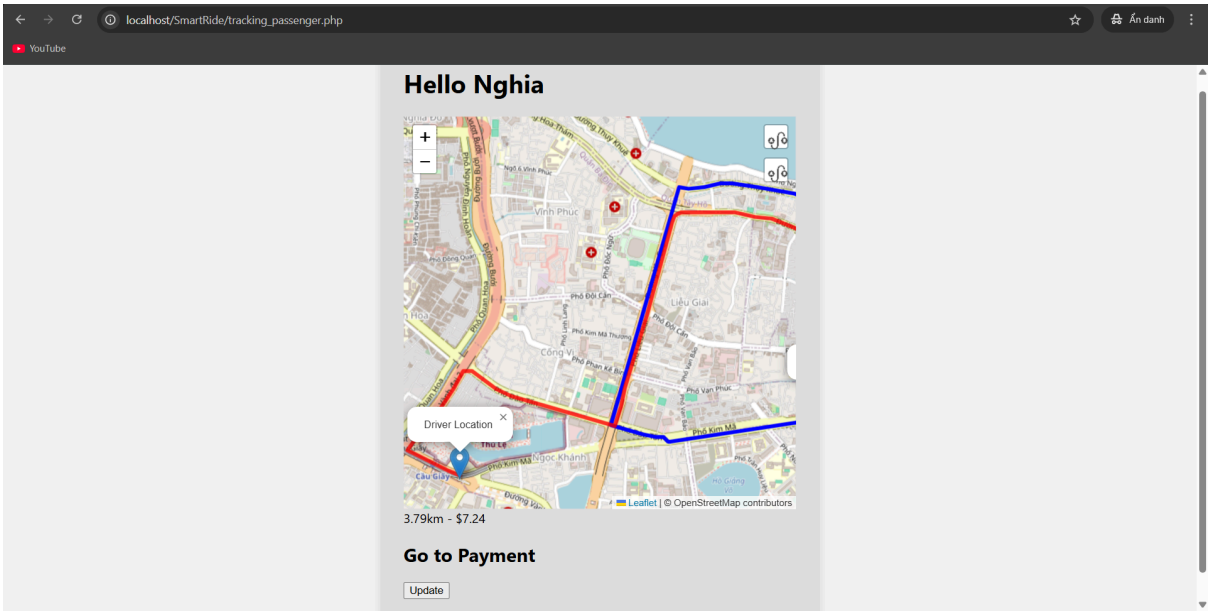

FIgure 30 - Complete ride

Scenario 5: Payment:

This class generates the payment since this assignment just require a basic payment method

```php
models > Payment.php > ...
<?php
// models/Payment.php
require_once 'Database.php';

0 references | 0 implementations
class Payment {
    1 reference
    public $id;
    1 reference
    public $ride_id;
    1 reference
    public $passenger_id;
    1 reference
    public $status;

    0 references | 0 overrides
    public function __construct($id, $ride_id, $passenger_id, $status) {
        $this->id = $id;
        $this->ride_id = $ride_id;
        $this->passenger_id = $passenger_id;
        $this->status = $status;
    }

    // Create a payment record (simulate payment)
    0 references | 0 overrides
    public static function createPayment($ride_id, $passenger_id): int|string {
        $conn = Database::getInstance();
        // We simulate a payment by marking it as 'paid'
        $status = 'paid';
        $stmt = $conn->prepare(query: "INSERT INTO payments (ride_id, passenger_id, status) VALUES (?, ?, ?)");
        $stmt->bind_param(types: "iis", var: &$ride_id, vars: &$passenger_id, $status);
        $stmt->execute();
        $id = $stmt->insert_id;
        $stmt->close();
        return $id;
    }
}
?>
```

FIgure 31 - Payment.php

5, Appendix

E., F. (2004). Head First Design Patterns. O’Reilly Media. Erich, G., Richard, H., Ralph, J., & John, V. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Gang of Four.