Assignment 3

SWE30003 - Software Architecture and Design

Project title: Online Ride Sharing Platform

Member:
Pham Tuan Bach- 10435403
Tran Hoang Duy Linh - 104222060
Nguyen Hoang Nguyen - 1041754342
Nguyen Thanh An - 104221672

Introduction

This document presents the design and implementation of the SmartRide ride-sharing platform, developed using React.js for the frontend and ASP.NET for the backend. The system incorporates key software design patterns and utilizes SignalR for real-time communication. The platform is tailored to provide a seamless experience for both drivers and passengers, ensuring efficient ride-matching and communication. The implementation focuses on scalability, reliability, and user-centric functionality.

Executive Summary

SmartRide is an online ride-sharing application designed to facilitate convenient and efficient transportation. The platform supports separate interfaces for drivers and passengers, enabling real-time ride requests and matching through Google Cloud APIs. SignalR integration ensures instant ride status updates. Key features include a structured request-handling mechanism, role-based user interfaces, and a robust backend for managing ride-related data. The implementation follows industry best practices to deliver a reliable and scalable solution.

Detailed Design

Refinement of Original Design

Original Design Summary

The original design established core classes including User (with Driver, Passenger, and Manager subclasses), Ride, RideRequest, Invoice, Rate, Notification, PaymentMethod, Database, and Report. These classes formed a foundation for the ride-sharing system but lacked implementation details necessary for actual development.

Design Refinement Approach

Our detailed design refines these classes by:

1. Adding specific attributes and methods with proper access modifiers
2. Introducing interface implementations for enhanced flexibility
3. Creating UI classes that follow object-oriented principles
4. Developing a database schema that supports persistence requirements
5. Enhancing security through authentication mechanisms
6. Adding exception handling and validation

Changes from Initial Design

The following adjustments were made to refine the initial design into a detailed, implementation-ready architecture:

Enhancement of RideRequest Class

- Description: RideRequest was extended to include fare estimation logic and driver matching based on proximity, availability, and ratings.
- Reason: The initial design vaguely defined RideRequest responsibilities, omitting critical features like fare calculation and advanced matching criteria, which are essential for a functional ride-sharing system.
- Impact:
    - Static Structure: Increases RideRequest collaborations with Location and Rate.
    - Interaction Patterns: Adds runtime logic for fare computation and driver selection, affecting sequence diagrams (example: Booking a Ride).
- Justification: Improves system accuracy and user satisfaction by ensuring fair pricing and optimal driver allocation, while keeping the class modular.

Removal of Ride Tracking Functionality

- Description: Real-time ride tracking, previously included in the Ride and Location classes, has been removed from the scope of this design.
- Reason: Implementing real-time ride tracking requires two or more real devices (example: A driver's phone and a passenger's phone) with actual GPS integration, which exceeds the project's current resources and simulation capabilities. The initial design's simulated GPS was insufficiently detailed for practical use.
- Impact:
    - Static Structure: Reduces the responsibilities of Ride and Location; removes dynamic tracking collaborations with Notification.
    - Class Responsibilities: Ride now focuses solely on ride lifecycle management (start, ongoing, completed) without location updates.
    - Interaction Patterns: Eliminates the "Ride Tracking" scenario from verification; simplifies runtime behavior to static ride status updates.
- Justification: Keeps the design feasible within the project's constraints, avoiding complexity that cannot be tested or implemented effectively. Tracking can be reintroduced in future iterations with proper hardware support.

Non-Changes from Initial Design

The following elements from Assignment 2 were retained in the detailed design due to their adequacy and alignment with the system's goals:

Core Class Hierarchy (User, Driver, Passenger, Manager)

- Description: The inheritance structure with User as the superclass and Driver, Passenger, and Manager as subclasses remains unchanged.
- Reason: This hierarchy effectively encapsulates shared user behaviors (example: registration, login) while allowing role-specific responsibilities, supporting the Factory pattern for user creation.
- Impact: No changes to UML class relationships or bootstrap process; responsibilities remain evenly distributed.
- Justification: The structure is robust, scalable, and avoids redundancy, requiring no adjustment for implementation.

Notification Class Responsibilities

- Description: The Notification class retains its role in sending ride updates, confirmations, and payment alerts without modification.
- Reason: The initial responsibilities were well-defined and sufficient for in-app notifications, aligning with the simplification of excluding SMS/email integration.
- Impact: No changes to collaborations with Passenger, Driver, and PaymentMethod.
- Justification: Maintains simplicity and focus on core functionality, with the Observer pattern enhancing its scalability for future notification types.

PaymentMethod Class Structure

- Description: The PaymentMethod class continues to handle payment processing and validation as originally designed.
- Reason: The initial design's focus on an in-app credit system and non-refundable transactions remains practical and simplifies financial logic.
- Impact: No alterations to its collaborations with Invoice and DatabaseManager.
- Justification: Keeps transaction management straightforward and consistent with the system's scope, avoiding unnecessary complexity.

Candidate Class List

User Classes

1. **User** - Abstract base class for all system users with common authentication and profile functionality

2. **Driver** - Provides transportation services, accepts ride requests, and receives payments

3. **Passenger** - Requests rides, tracks drivers, makes payments, and provides ratings

4. **Manager** - Oversees system operations, generates reports, and monitors performance

Core Functionality Classes

5. **Ride** - Represents active transportation between driver and passenger with real-time tracking

6. **RideRequest** - Handles ride bookings, driver matching, and fare estimation

7. **Invoice** - Manages billing, fare calculation, and payment confirmations

Support Classes

8. **Rate** - Stores and calculates driver performance ratings and feedback

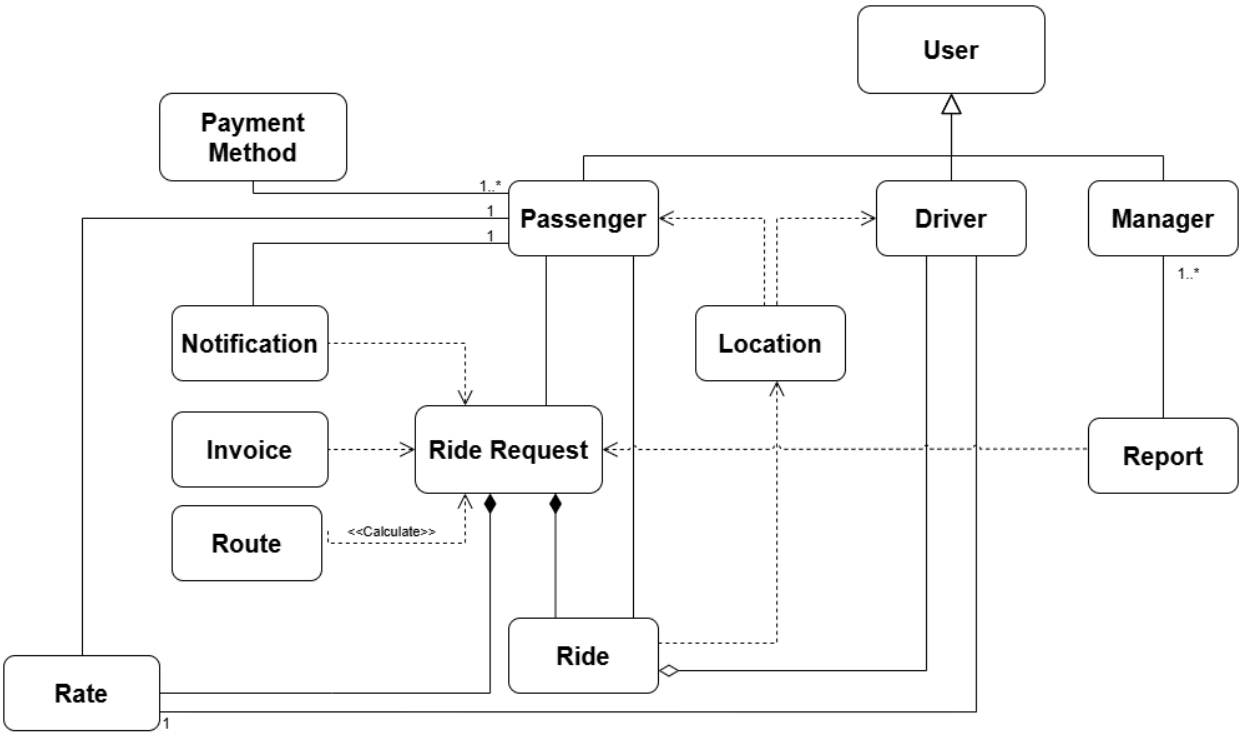
9. **Notification** - Delivers system communications to users about rides and payments

10. **PaymentMethod** - Processes transactions and handles payment validation

11. **Database** - Central data storage and retrieval system

12. **Report** - Generates analytical insights and performance metrics

UML diagram



CRC Cards

User Classes

User:

Class Name	**User**
Super Class	
Description	Abstract base class for all system users with common authentication and profile functionality
Responsibilities	Collaborators

Know basic information like ID, email address, username, password	--
Can authenticate login credentials	Database
Can update profile information	Database

Driver:

Class Name	**Driver**
Super Class	User
Description	Provides transportation services, accepts ride requests, and receives payments
Responsibilities	Collaborators
Know driver-specific details (vehicle type, license number)	--
Can accept or decline ride requests	RideRequest
Can track active ride status	Ride
Can receive payment after ride completion	Invoice, PaymentMethod

Passenger:

Class Name	**Passenger**
Super Class	User
Description	Requests rides, tracks drivers, makes payments, and provides ratings
Responsibilities	Collaborators
Know passenger-specific details preferred payment method)	--

Can request a ride	RideRequest
Can track driver location and ride progress	Ride
Can make payment for ride	Invoice, PaymentMethod
Can rate driver performance	Rate

Manager:

Class Name	**Manager**
Super Class	User
Description	Oversees system operations, generates reports, and monitors performance
Responsibilities	Collaborators
Know authorization level	--
Can monitor system performance (ride completions, disputes)	Report
Can generate analytical reports	Report, Database
Can manage user accounts (suspend, review)	Database

Core Functionality Classes

Ride:

Class Name	**Ride**
Super Class	
Description	Represents active transportation between driver and passenger with real-time tracking
Responsibilities	Collaborators

Know ride details (start time, end time, pickup/drop-off locations)	--
Can track ride progress in real-time	Driver, Passenger
Can calculate ride duration and distance	--
Can update ride status (started, completed)	Database

RideRequest:

Class Name	**RideRequest**
Super Class	
Description	Handles ride bookings, driver matching, and fare estimation
Responsibilities	Collaborators
Know request details (pickup location, destination, time)	--
Can match request to available drivers	Driver
Can estimate fare based on distance and time	--
Can create a ride instance upon acceptance	Ride, Database

Invoice:

Class Name	**Invoice**
Super Class	
Description	Manages billing, fare calculation, and payment confirmations
Responsibilities	Collaborators

Know ride cost and payment status	Ride
Can calculate final fare (base fare, surge pricing)	Ride
Can generate billing document for passenger	Passenger
Can confirm payment completion	PaymentMethod, Database

Support Classes

Rate:

Class Name	**Rate**
Super Class	
Description	Stores and calculates driver performance ratings and feedback
Responsibilities	Collaborators
Know rating score and feedback comments	--
Can calculate average driver rating	--
Can store rating data for historical analysis	Database
Can provide rating summary to manager	Manager

Notification:

Class Name	**Notification**
Super Class	
Description	Delivers system communications to users about rides and payments

Responsibilities	Collaborators
Know notification content (ride confirmation, payment receipt)	--
Can send messages to users (email, SMS)	User (Driver, Passenger)
Can log notification history	Database

PaymentMethod:

Class Name	**PaymentMethod**
Super Class	
Description	Processes transactions and handles payment validation
Responsibilities	Collaborators
Know payment type (credit card, digital wallet)	--
Can process payment transactions	Invoice
Can validate payment details (card number, expiry)	--
Can confirm payment success to system	Database

Database:

Class Name	**Database**
Super Class	
Description	Central data storage and retrieval system
Responsibilities	Collaborators
Know connection details (host, credentials)	--

Can execute queries for data storage and retrieval	User, Ride, RideRequest, Invoice, Rate, Notification, PaymentMethod
Can ensure data consistency (ACID compliance)	--

Report:

Class Name	**Report**
Super Class	
Description	Generates analytical insights and performance metrics
Responsibilities	Collaborators
Know report parameters (time range, metrics)	--
Can aggregate data for analysis (ride frequency, revenue)	Database
Can generate formatted reports for managers	Manager

Design Patterns

In designing the SmartRide system, we have deliberately chosen not to incorporate formal design patterns such as Singleton, Factory, Strategy, or Composite. This decision was driven by two key factors: time constraints and the nature of our backend architecture, which relies on API-based interactions rather than deep object-oriented structures.

Absence of Design Patterns

Description:
Unlike traditional object-oriented designs that rely on design patterns to address recurring problems, the SmartRide system follows a straightforward API-driven architecture. For instance, we avoided using **any** pattern for database management

(instead relying on stateless API calls) and did not implement the Strategy pattern for payment processing, as API-based payment gateways already handle method variations.

Justification:
Design patterns, while useful in certain contexts, offer limited benefits in our backend design since services communicate via RESTful or GraphQL APIs rather than through complex object relationships. Additionally, given the project's timeline, prioritizing functional development over architectural patterns ensures faster delivery without unnecessary abstraction.

Benefits of This Approach

- **Faster Development:** Avoiding patterns reduces design overhead, allowing the team to focus on delivering functional APIs within the project timeline.

- **Simplified Maintenance:** Developers can easily modify and extend the system without adhering to strict pattern conventions.

- **Optimized for APIs:** Since backend services interact via APIs rather than in-memory objects, patterns like Factory or Composite provide minimal advantages.

Trade-Offs Considered

- **Scalability:** While patterns could standardize object creation or behavior handling, our modular service-based architecture already ensures flexibility and scalability.

- **Reusability:** API-driven services are inherently reusable across multiple clients, making traditional object-oriented reuse less relevant.

- **Alignment with Project Goals:** SmartRide's primary objective is to build a scalable and maintainable ride-sharing platform. By focusing on API efficiency rather than design patterns, we align with this goal while ensuring timely delivery.

Quality of Assignment 2 Design

Good aspects

Each section focuses on one task, preventing unnecessary complexity. Design patterns add flexibility, enabling changes without impacting other areas. Data flows smoothly, ensuring quick access to ride, user, and payment info. Responsibilities are balanced, with minimal dependencies between sections.

Missing from original design

Plans for handling heavy traffic are unclear. Steps for continuous location updates are undefined. Integration with external payment providers is absent. Error responses for payment failures or ride cancellations are not covered.

Flawed aspects of original design

Security lacks a second layer of protection for user accounts. Ride matching is basic, missing efficiency and fairness improvements. Notifications are limited to the app, which isn't helpful for offline users. Management has limited control over user settings and system configurations.

Level of Interpretation Required

Developers need to assume how GPS tracking will work, as real-world mapping details are missing. Ride assignment requires extra logic for better efficiency. The service startup is outlined broadly, without clear guidance on implementation choices, creating gaps for real-world application.

Lesson Learnt

System Design and Flexibility: Through this project, we learned the importance of designing a flexible system. For future improvements, we will focus on ensuring that our system is adaptable to new requirements, such as adding new ride types or integrating additional payment methods without significant code changes.

Clear Class Responsibilities: We recognized the need for clear and focused class responsibilities. By ensuring that each class addresses a specific aspect of the system,

the system becomes easier to maintain and scale. Future systems will benefit from a more granular approach to class responsibilities, improving code readability and maintainability.

Error Handling and Validation: We learned the significance of robust error handling and validation in ride-sharing platforms. Addressing edge cases such as booking conflicts, failed payments, or service downtime is crucial. Ensuring thorough validation in every component will enhance system reliability and provide a seamless user experience.

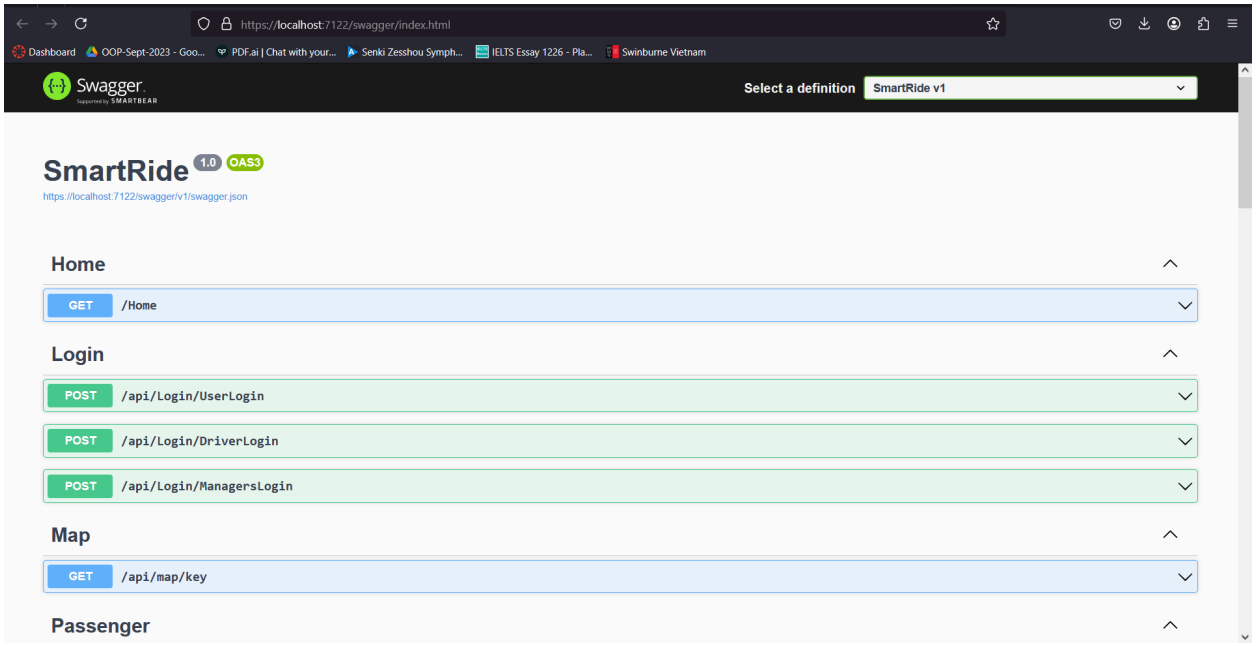
Testing and Simulation: Testing has proven to be an essential part of ensuring system functionality. We discovered that simulating real-world scenarios during testing helps uncover potential issues early. In future projects, we will allocate more time for comprehensive testing, including user load tests and edge-case simulations.

Real-Time Features: Implementing real-time features, such as ride status updates, was a challenging but valuable learning experience. We learned the importance of optimizing for real-time performance, especially in high-traffic situations. For future systems, we will invest in more advanced real-time technologies to ensure a smooth user experience.

Implementation

- The front end of the project is coded in React JS
- The back end of the project is coded in ASP NET C#
- The database is hosted on MicrosoftSQL server management studio
- For the backend, we create APIs for the front end to use.
- As the implementation of the backend is repetitive, we will only make some examples of the backend code and frontend, as in how APIs are created and used.

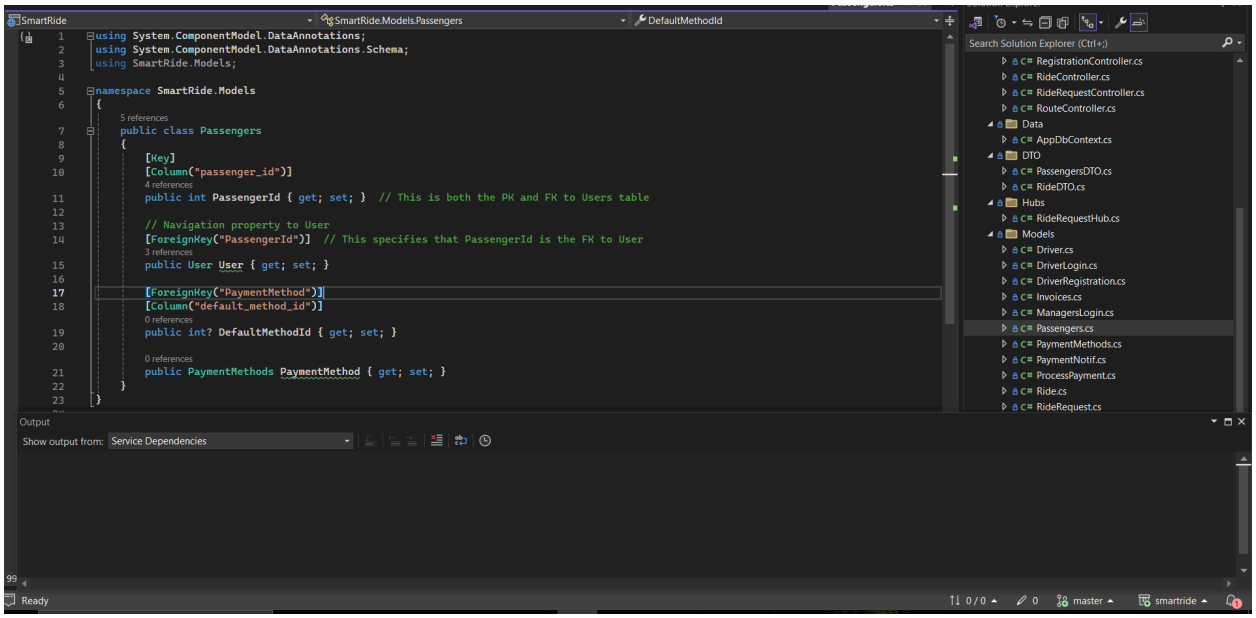
Backend Implementation



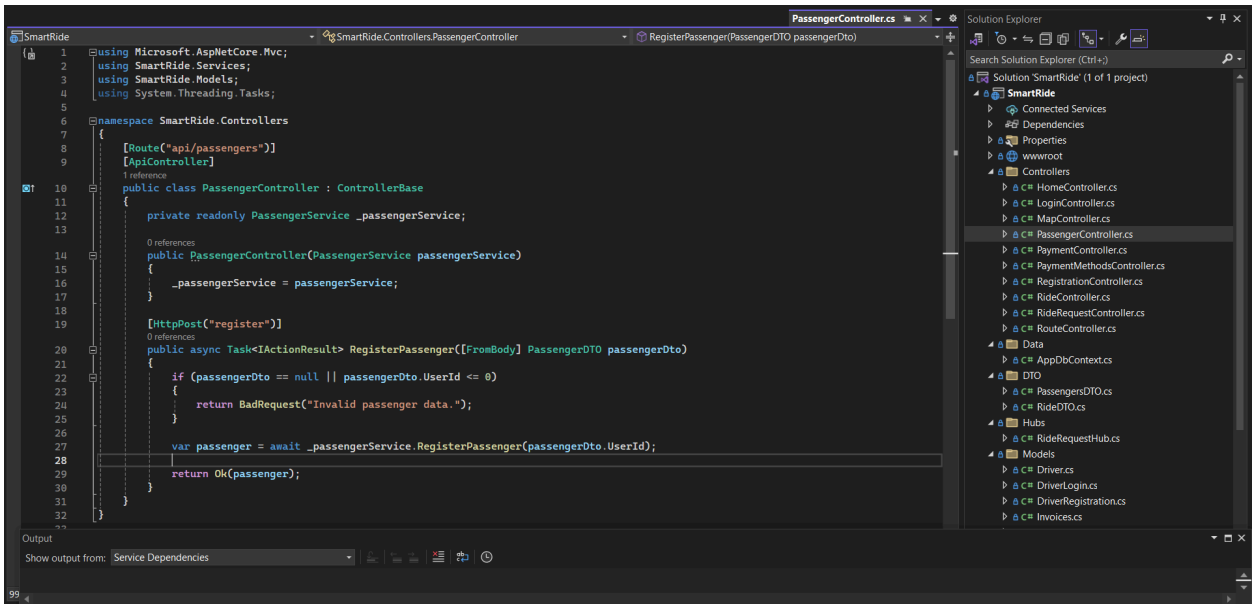
- The API is written as HTTP methods to interact with the front end.
- For each of the API, we need 2 cs files for it. They use Entity Framework to interact with the database.

Passenger API

- Let's take an example of the Passenger API. This API maps the corresponding column in the database as variables to use in the code.

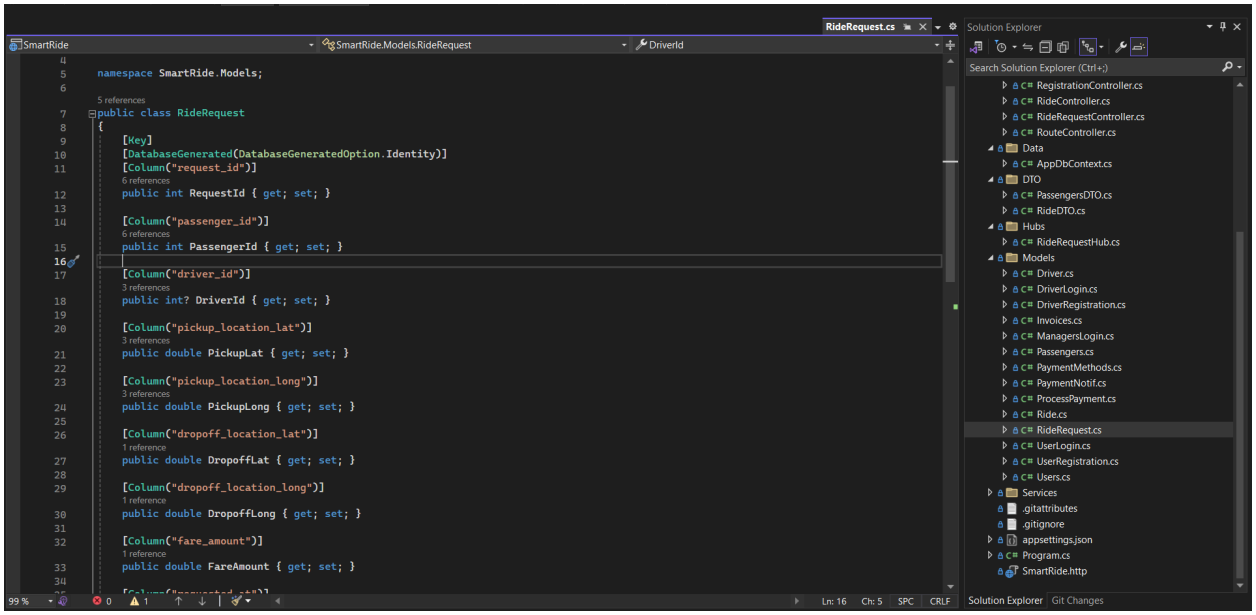


- Next is the PassengerController, this file is used to provide an API endpoint for adding new passenger's info.

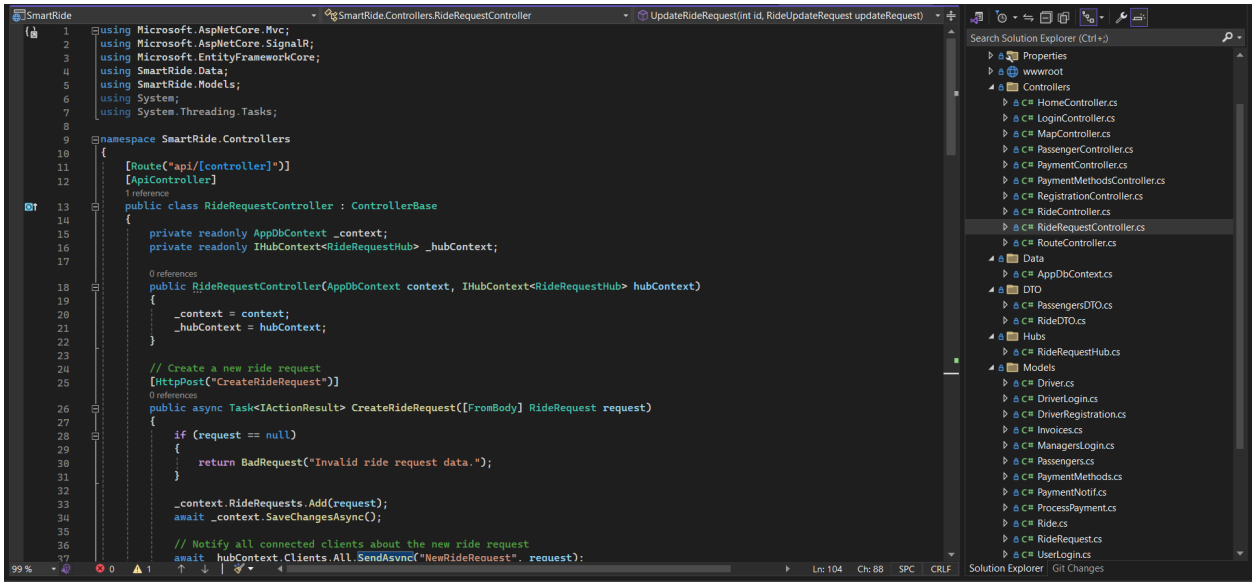


RideRequest API

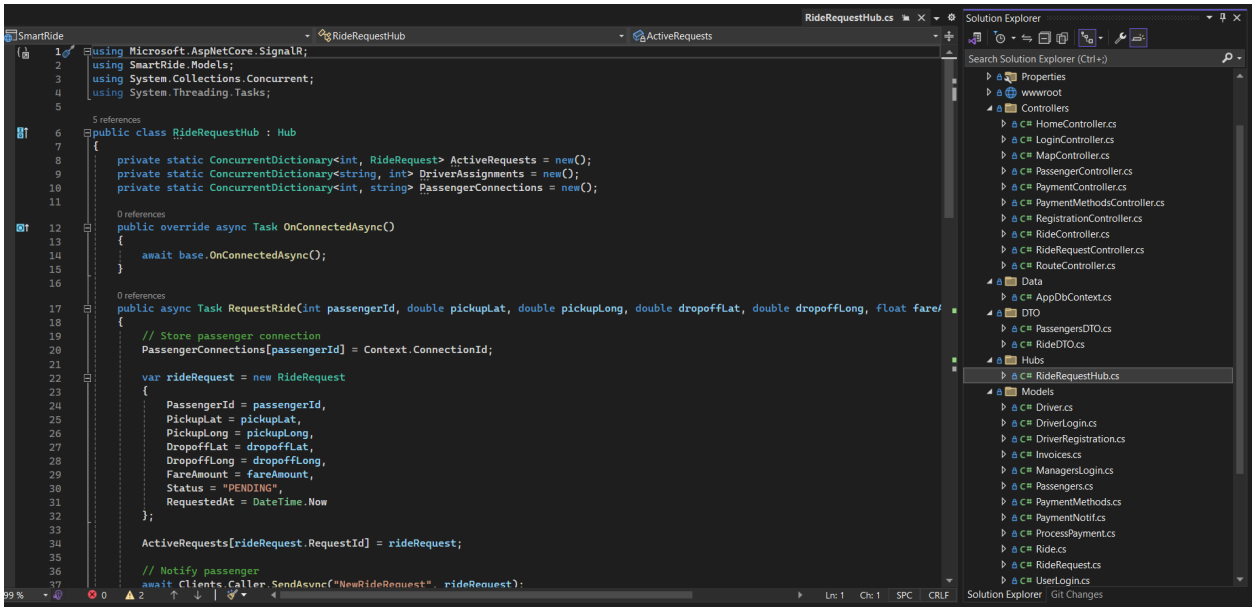
- The model represents a ride request made by a passenger.



- The RideRequestController creates API endpoints for the front end to use.

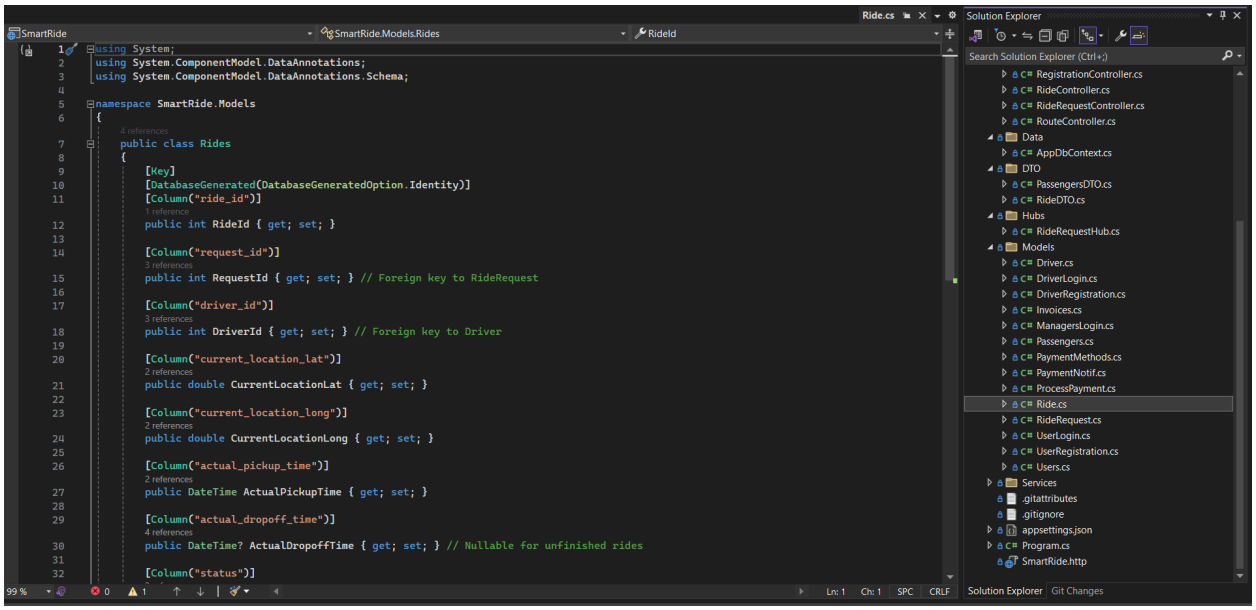


- As some of the API of the ride request needs to be updated in real time (example: When a passenger creates a request, it will appear in the driver dashboard in real time), we use SignalR for real-time communication. For this feature, we add a RideRequestHub file to provide real-time updates.



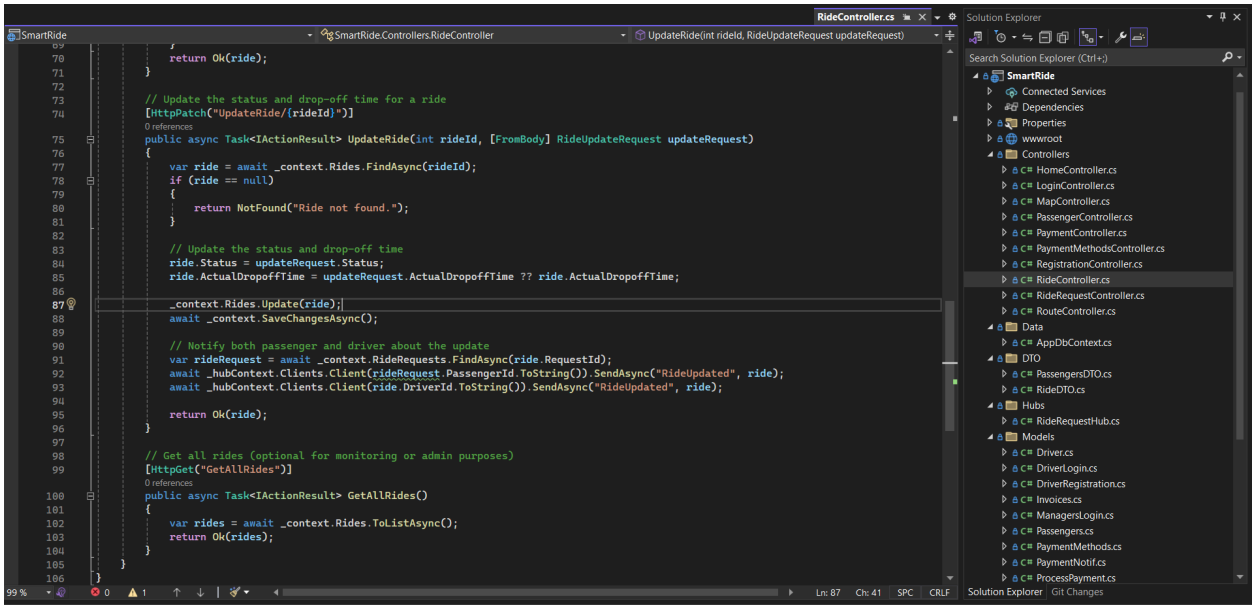
Ride API

- This model represents a ride.



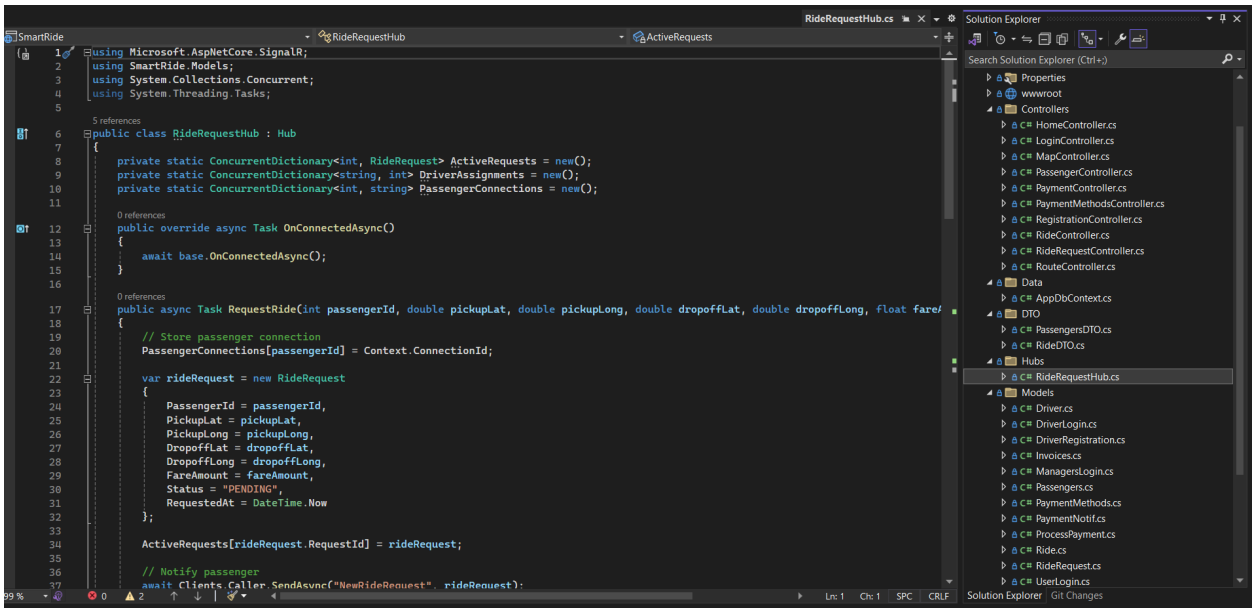
```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace SmartRide.Models
{
    public class Rides
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        [Column("ride_id")]
        public int RideId { get; set; }

        [Column("request_id")]
        public int RequestId { get; set; } // Foreign key to RideRequest

        [Column("driver_id")]
        public int DriverId { get; set; } // Foreign key to Driver

        [Column("current_location_lat")]
        public double CurrentLocationLat { get; set; }

        [Column("current_location_long")]
        public double CurrentLocationLong { get; set; }

        [Column("actual_pickup_time")]
        public DateTime ActualPickupTime { get; set; }

        [Column("actual_dropoff_time")]
        public DateTime? ActualDropoffTime { get; set; } // Nullable for unfinished rides

        [Column("status")]
```

- The RideController exposes API for the front end to use.



```
            return Ok(ride);
        }

        // Update the status and drop-off time for a ride
        [HttpPatch("UpdateRide/{rideId}")]
        public async Task<IActionResult> UpdateRide(int rideId, [FromBody] RideUpdateRequest updateRequest)
        {
            var ride = await _context.Rides.FindAsync(rideId);
            if (ride == null)
            {
                return NotFound("Ride not found.");
            }

            // Update the status and drop-off time
            ride.Status = updateRequest.Status;
            ride.ActualDropoffTime = updateRequest.ActualDropoffTime ?? ride.ActualDropoffTime;

            _context.Rides.Update(ride);
            await _context.SaveChangesAsync();

            // Notify both passenger and driver about the update
            var rideRequest = await _context.RideRequests.FindAsync(ride.RequestId);
            await _hubContext.Clients.Client(rideRequest.PassengerId.ToString()).SendAsync("RideUpdated", ride);
            await _hubContext.Clients.Client(ride.DriverId.ToString()).SendAsync("RideUpdated", ride);

            return Ok(ride);
        }

        // Get all rides (optional for monitoring or admin purposes)
        [HttpGet("GetAllRides")]
        public async Task<IActionResult> GetAllRides()
        {
            var rides = await _context.Rides.ToListAsync();
            return Ok(rides);
        }
    }
}
```

- As the status of the ride and its location, etc, will be updated in real time, it also needs its own SignalR file to create a hub session for real-time update. This file is reused from the RideRequestHub.


RideRequestHub.cs
SmartRide
RideRequestHub
ActiveRequests
using Microsoft.AspNetCore.SignalR;
using SmartRide.Models;
using System.Collections.Concurrent;
using System.Threading.Tasks;
5 references
public class RideRequestHub : Hub
{
private static ConcurrentDictionary<int, RideRequest> ActiveRequests = new();
private static ConcurrentDictionary<string, int> DriverAssignments = new();
private static ConcurrentDictionary<int, string> PassengerConnections = new();
0 references
public override async Task OnConnectedAsync()
{
await base.OnConnectedAsync();
}
0 references
public async Task RequestRide(int passengerId, double pickupLat, double pickupLong, double dropoffLat, double dropoffLong, float fare
{
// Store passenger connection
PassengerConnections[passengerId] = Context.ConnectionId;
var rideRequest = new RideRequest
{
PassengerId = passengerId,
PickupLat = pickupLat,
PickupLong = pickupLong,
DropoffLat = dropoffLat,
DropoffLong = dropoffLong,
FareAmount = fareAmount,
Status = "PENDING",
RequestedAt = DateTime.Now
};
ActiveRequests[rideRequest.RequestId] = rideRequest;
// Notify passenger
await Clients.Caller.SendAsync("NewRideRequest", rideRequest);
99 %
0
2
Ln: 1
Ch: 1
SPC
CRLF
Solution Explorer
Search Solution Explorer (Ctrl+;)
Properties
wwwroot
Controllers
HomeController.cs
LoginController.cs
MapController.cs
PassengerController.cs
PaymentController.cs
PaymentMethodsController.cs
RegistrationController.cs
RideController.cs
RideRequestController.cs
RouteController.cs
Data
AppDbContext.cs
DTO
PassengersDTO.cs
RideDTO.cs
Hubs
RideRequestHub.cs
Models
Driver.cs
DriverLogin.cs
DriverRegistration.cs
Invoices.cs
ManagersLogin.cs
Passengers.cs
PaymentMethods.cs
PaymentNotif.cs
ProcessPayment.cs
Ride.cs
RideRequest.cs
UserLogin.cs
Solution Explorer
Git Changes

Frontend Implementation

Ride Request Frontend

- It will give inputs to the RideRequestAPI by inputting in information of the ride request, and then the API will handle all the features.

```
useEffect(() => {
  listenForRideRequests((newRequest) => {
    console.log("New ride request received", newRequest);

    setRideRequests(prevRequests => {
      const isDuplicate = prevRequests.some(
        request => request.request_id === newRequest.request_id
      );

      if (isDuplicate) {
        return prevRequests;
      }

      return [...prevRequests, {
        request_id: newRequest.request_id,
        pickup_lat: newRequest.pickupLat,
        pickup_long: newRequest.pickupLong,
        dropoff_lat: newRequest.dropoffLat,
        dropoff_long: newRequest.dropoffLong,
        fare: newRequest.fareAmount,
      }];
    });
  });
}, []);

useEffect(() => {
  async function fetchKey() {
    try {
      const key = await fetchMapKey();
      if (key) {
        setSubscriptionKey(key);
        loadMapScript();
      } else {
        console.error("Failed to get map key");
      }
```

Ride Frontend

- It accepts the API of Ride and input in the info selected from the screen by the passenger. An instance of ride is only created when the ride requests status is changed from “PENDING” to “ACCEPTED” in the database.

```
useEffect(() => {
  if (subscriptionKey && mapRef.current && !mapInstance.current) {
    const script = document.createElement("script");
    script.src =
      "https://atlas.microsoft.com/sdk/javascript/mapcontrol/2/atlas.min.js";
    script.async = true;
    script.onload = () => {
      mapInstance.current = new window.atlas.Map(mapRef.current, {
        center: [105.91715, 21.043769],
        zoom: 10,
        view: "Auto",
        authOptions: {
          authType: "subscriptionKey",
          subscriptionKey: subscriptionKey,
        },
      });

      mapInstance.current.events.add("ready", () => {
        datasourceRef.current = new window.atlas.source.DataSource();
        mapInstance.current.sources.add(datasourceRef.current);
        mapInstance.current.layers.add(
          new window.atlas.layer.LineLayer(datasourceRef.current)
        );

        mapInstance.current.events.add("click", (event) => {
          if (!rideRequested) {
            const position = event.position;
            const coordinates = mapInstance.current.pixelsToPositions([
              position,
            ])[0];
            setDropoffLat(coordinates[1].toFixed(6));
            setDropoffLon(coordinates[0].toFixed(6));
```

Execution and Operation

How to run:

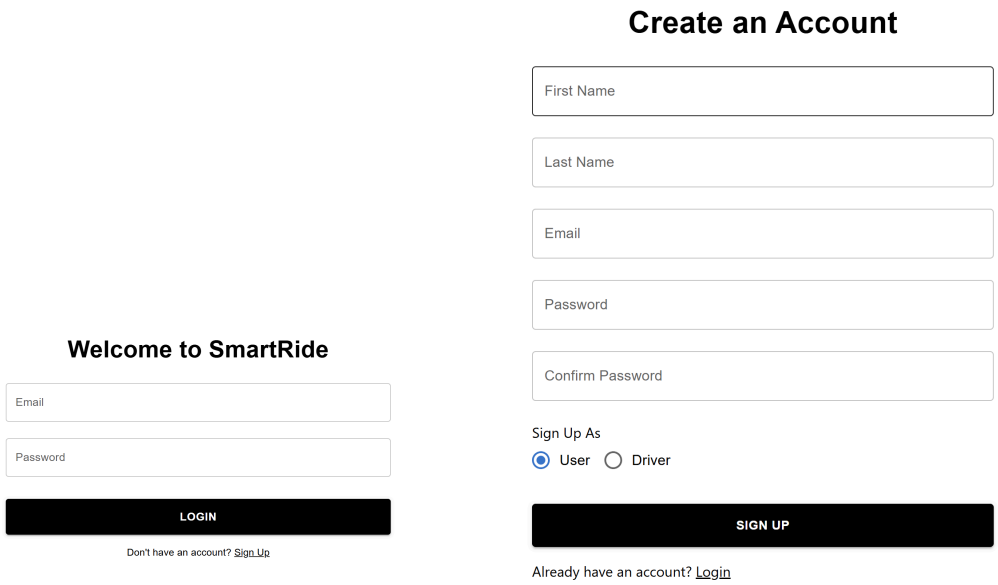
- Download Microsoft SQL management studio.
- Open it and connect to the database
- Create the database, named “smartride”, and then run the query from the SQL query file provided in the code (sql folder).
- Run the backend on Visual Studio Community 2022.
- Open the front end project in VS Code, then run npm install, then npm start on the terminal to open the server.

Scenario 1: User and Driver Registration Process

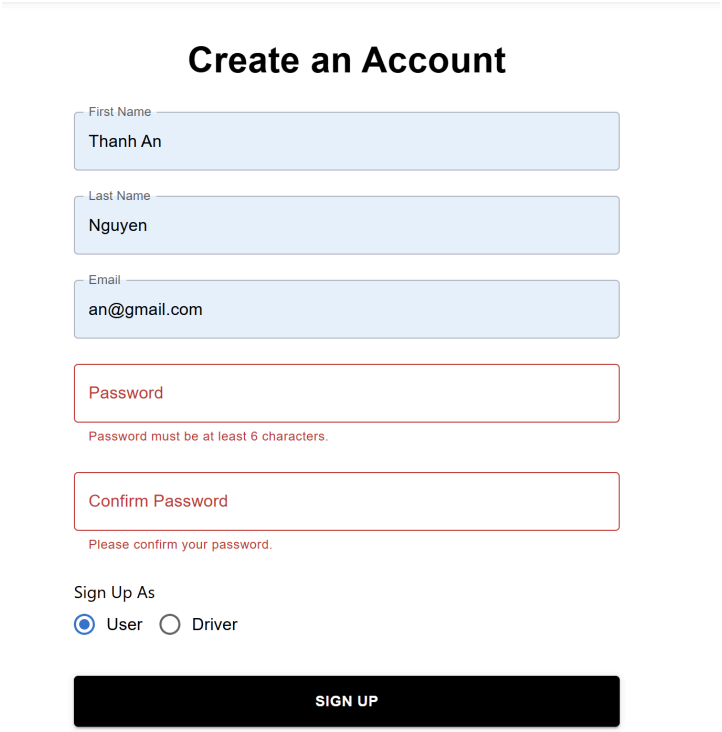
Description: This scenario demonstrates the complete registration process for both regular passengers and drivers in the SmartRide application, showing the different steps and information required for each user type.

Steps:

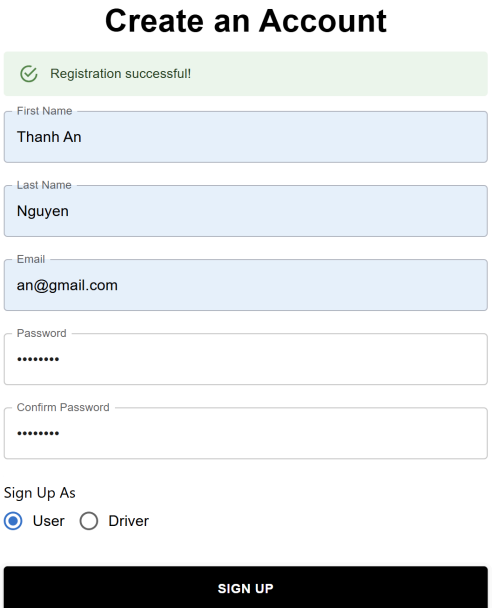
1. New user opens the SmartRide application showing the empty login/registration screen



2. User enters basic account information (first name, last name, email, password, confirm password)
3. The system validates document format (shows error if format is incorrect)



4. User selects account type: "User" or "Driver"
5. For user registration:
    - User press the “Sign Up” button to complete registration
    - System confirms user account creation



6. For driver registration:
    - User enters driver's license information
    - User provides vehicle details
    - User receives confirmation of successful submission

7. The system navigates to Login session to log into the application

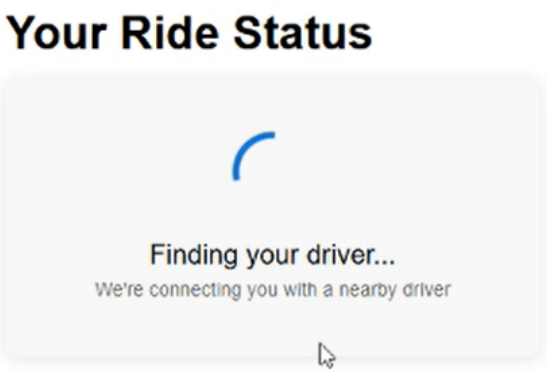
Scenario 2: User Requesting a Ride

Description: This scenario demonstrates how a registered user can request a ride through the SmartRide application, from entering locations to submitting the request.

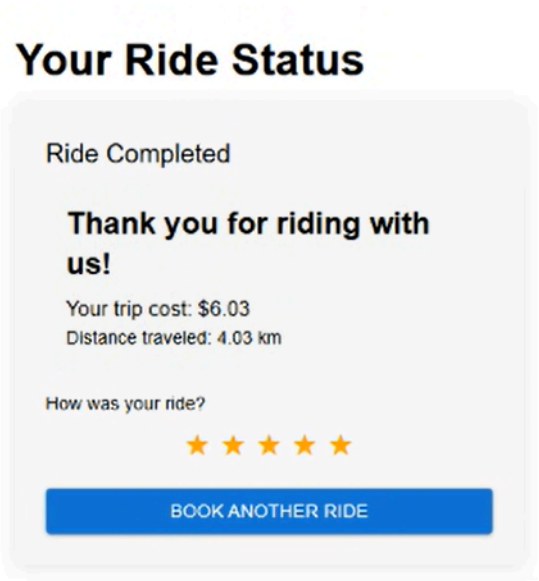
Steps:

1. User logs in to see the dashboard
2. Users enters pickup and drop-off location (tracking based on device's location)

3. The system calculates the estimated price
4. User click "Confirm Ride" button
5. The system assigns driver



6. When the ride is completed, the system shows the summary of the ride

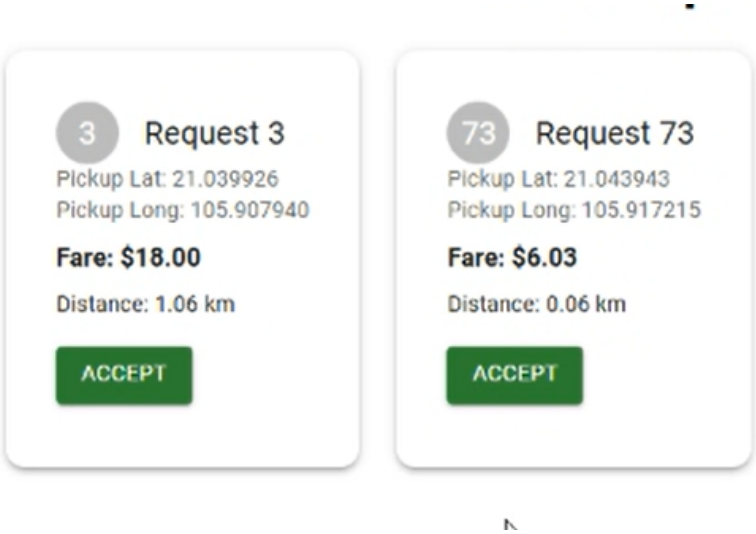


Scenario 3: Driver Accepting and Starting a Ride

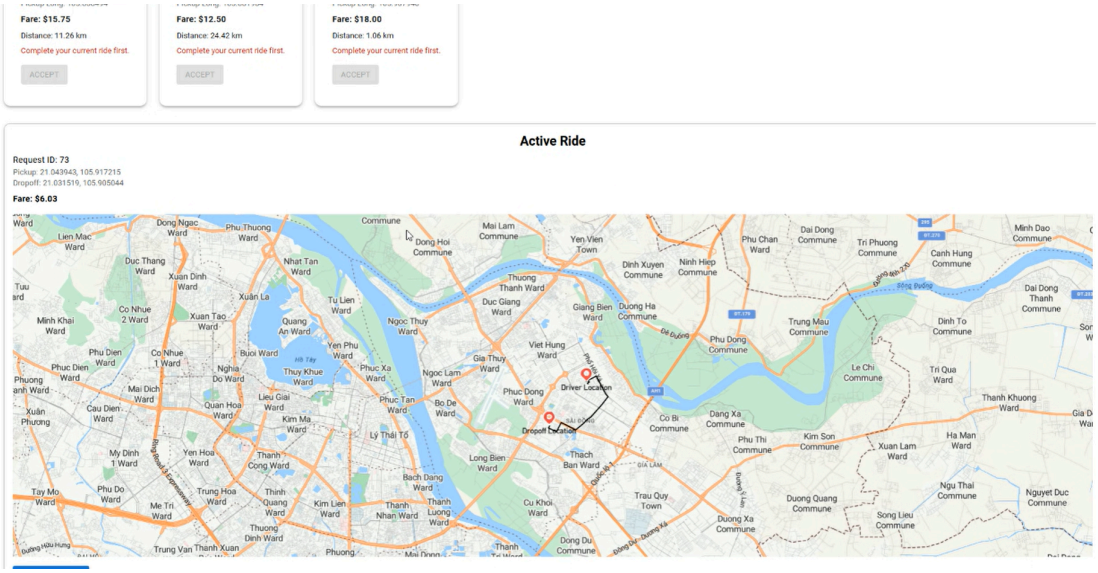
Description: This scenario demonstrates how a driver receives ride requests, accepts them, navigates to the pickup location, and starts the ride.

Steps:

1. Driver logs in to see the driver dashboard
2. Driver receives notification of a new ride request
3. System displays request details:
    - Passenger name
    - Pickup location
    - Destination
    - Estimated fare



4. Driver reviews request information
5. Driver has option to accept or decline
6. Driver follows navigation to pickup location, and destination



7. Driver complete the ride

Appendix

SWE30003 - SOFTWARE ARCHITECHTURE AND DESIGN

OBJECT-ORIENTED DESIGN **DOCUMENT**

GROUP 10
NGUYEN HOANG NGUYEN - 1041754342
NGUYEN THANH AN - 104221672
PHAM TUAN BACH - 104354303
TRAN HOANG DUY LINH - 104222060

LECTURER: DR. DO THI BICH NGOC

Table of Content

Executive Summary

SmartRide is a ride-sharing platform developed to enhance urban transportation by digitally connecting riders with available drivers. The current manual system results in inefficiencies such as long wait times, difficulties in driver allocation, and manual payment processing. This report outlines an object-oriented design for an automated system that addresses these issues. The proposed solution includes automated ride matching, real-time location tracking, a simplified payment handling system, and ride history management. By following object-oriented design principles, the system ensures modularity, flexibility, and future scalability.

1. Introduction

This Object Design Document outlines the object-oriented design (OOD) for the SmartRide Online Ride-Sharing Platform (ORSP). It serves as a technical reference for developers, system architects, and stakeholders, detailing the problem analysis, class identification, design quality, applied design patterns, and system verification.

This document is structured as follows:

- **Section 2**: Problem analysis, assumptions, and design trade-offs.
- **Section 3**: Candidate class identification, including **UML diagrams and CRC cards**.
- **Section 4**: Design quality evaluation based on responsibilities, heuristics, and modularity.
- **Section 5**: Applied **design patterns**, ensuring system scalability and maintainability.
- **Section 6**: The **bootstrap process**, outlining system initialization steps.
- **Section 7**: **Use case verification**, demonstrating system functionality.

1.2 Outlook of the Solution

The proposed solution replaces the manual ride-matching and payment process with an automated, digital platform. The system will provide:

- **Automated driver allocation**, reducing passenger wait times.
- **Real-time ride tracking** through simulated GPS integration.
- **A structured payment system**, replacing cash transactions with an in-system balance system.

- **A scalable object-oriented design**, ensuring **maintainability** and future enhancements.

The system relies on a database for user management, ride data storage, and payment tracking. Notifications and ride updates are processed asynchronously, ensuring smooth communication between passengers and drivers.

1.3 Trade-offs and Object Design

Certain trade-offs were made to keep the system scope manageable within the available time and resources:

1.3.1 Naming Conventions

- Class names follow PascalCase (e.g., RideRequest, PaymentMethod).
- Variable names use camelCase (e.g., rideStatus, paymentAmount).
- Interface names begin with "I" (e.g., INotifiable).

1.3.2 Boundary Cases

- Maximum **ride distance** is predefined in the system to avoid edge cases with long-distance travel.
- **Account creation requires unique phone numbers**, avoiding duplicate registrations.

1.3.3 Authentication & Security

- **Temporary account freeze** after multiple failed login attempts, ensuring **basic security**.
- No **multi-factor authentication (MFA)** due to complexity constraints.

1.3.4 Data Handling & Exceptions

- The system assumes **all notifications** are received successfully, without external SMS/email integration.
- Payments are **non-refundable**, simplifying transaction handling.

1.4 Definitions, Acronyms, and Abbreviations

Term	Definition
ORSP	Online Ride-Sharing Platform
GPS	Global Positioning System
OOD	Object-Oriented Design
UML	Unified Modeling Language

2. Problem Analysis

The current manual ride-booking system used by SmartRide is inefficient, leading to long wait times, poor ride assignment, and inconvenient payments. This section analyzes key system challenges and presents assumptions and simplifications to keep the design feasible.

2.1 Assumptions

The following assumptions were made to simplify implementation while keeping the system functional:

2.1.1 User Registration & Roles

- Managers do not register manually; accounts are provided by the system admin.
- All passengers and drivers must register before using the platform.

2.1.2 Ride Management

- A driver can only handle one ride at a time to prevent scheduling conflicts.

- A passenger can only have one active ride at any time.

2.1.3 GPS & Tracking

- Real-time GPS tracking is simulated, as integrating real-world tracking APIs would be complex.
- ETA calculations are based on static distance-time calculations instead of live traffic updates.

2.1.4 Payment System

- The system uses an in-app credit system instead of real-world payment gateways.
- Refunds are not processed—transactions are final once a ride is completed.

2.1.5 Notifications

- Only in-app notifications are implemented; no SMS/email support.
- The system assumes all notifications are successfully received.

2.1.6 Security & Authentication

- Basic authentication with password-based login is implemented.
- Temporary accounts freeze after multiple failed logins for security.

2.2 Simplifications

To **reduce complexity** while maintaining system integrity, the following simplifications were applied:

Component	Simplification	Reason
Ride Matching	Uses driver availability & proximity instead of AI-based demand prediction.	Keeps implementation manageable.

Driver Earnings Tracking	No commission-based calculations, only total earnings tracking.	Reduces financial complexity.
Payment Processing	No partial payments or refunds. The system only processes full ride payments.	Simplifies transaction management.
Authentication	Only basic password-based login, no two-factor authentication (2FA).	Security kept minimal for ease of use.
Ride Demand Monitoring	Managers manually track ride demand, no automated AI-based forecasting.	AI-based demand prediction is out of scope.

2.3 Design Justification

The SmartRide system design follows object-oriented principles, ensuring a modular and extensible architecture. Key design decisions include:

1. **Database-Centric Approach**
    - The database stores all ride, user, and payment details.
    - The business logic interacts with the database through structured queries.
2. **Separation of Concerns**
    - The system is **modular**, with classes handling **specific** responsibilities.
    - **RideRequest**, **Ride**, and **Payment** are independent, avoiding tightly coupled logic.
3. **Minimal Data Redundancy**
    - All user and ride-related data are **stored efficiently**, ensuring **third normal form (3NF) compliance**.

4. **Scalability**

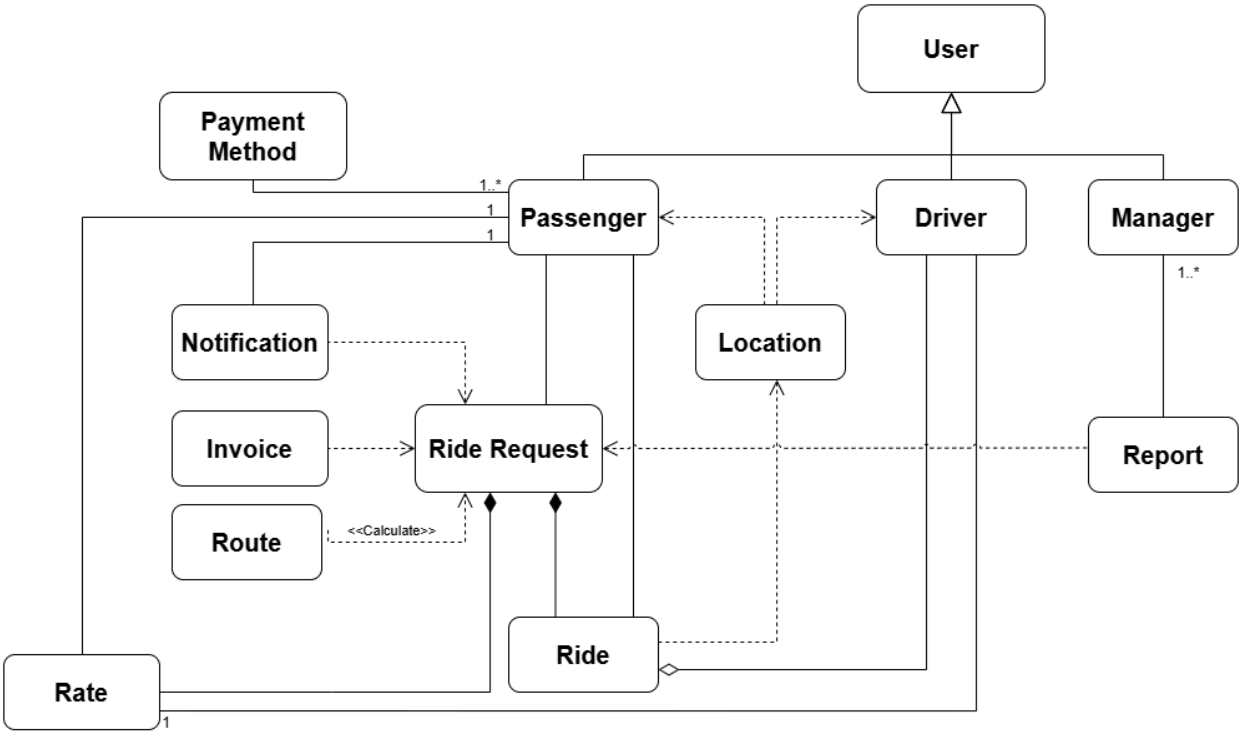
    - The Factory Pattern supports adding new user types (e.g., Admin, SupportStaff) without modifying core logic, should the business continue to expand in the future.
    - The Observer Pattern ensures notifications scale as more event-driven features are added.

3. Candidate Classes

3.1. Candidate classes list

- User:
    + Driver
    + Manager
    + Passenger
- Ride
- Ride Request
- Invoice
- Rate
- Notification
- PaymentMethod
- Database
- Report

3.2. UML Diagram



3.3. CRC Cards

3.3.1. User

Class Name: User **Super Class: -**	
Represents users in the system, including drivers, passengers, and managers.	
Responsibilities	**Collaborators**
● Register and log in to the system	NA
● Manage profile information such as name, contact details, and account settings	NA
● Book and manage rides	Ride, RideRequest
● Accept and complete rides	Ride
● Oversee system operations	Report, Database

3.3.1.1. Driver

Class Name: Driver **Super Class: User**	
Represents drivers who provide transportation services.	
Responsibilities	**Collaborators**
• Accept or decline ride requests from passengers	RideRequest
• Navigate to the passenger's pickup location and drive to the destination	NA
• Update ride status	Ride
• Receive payments through integrated payment methods	PaymentMethod, Invoice
• Maintain ratings based on passenger feedback	Rate

3.3.1.2. Passenger

Class Name: Passenger **Super Class: User**	
Represents passengers who use the system to request rides.	
Responsibilities	**Collaborators**
● Request a ride by entering pickup and destination locations	RideRequest
● Track the driver's location in real-time via GPS	Ride, Notification
● Make payments via the preferred payment method	PaymentMethod, Invoice
● Provide feedback and rate the driver	Rate

3.3.1.3. Manager

Class Name: Manager **Super Class: User**	
Represents system managers responsible for monitoring and overseeing system performance.	
Responsibilities	**Collaborators**
● Generate reports and analytics for business insights	RideRequest
● Monitor ride and driver performance statistics	Report
● Manage system-wide settings and policies	NA

3.3.2. Ride

Class Name: Ride **Super Class: -**	
Represents an active ride between a driver and a passenger.	
Responsibilities	**Collaborators**
• Store details of the ride, including route, driver, passenger, and fare	NA
• Track ride progress in real-time	Notification
• Manage ride lifecycle (start, ongoing, completed, canceled)	Driver, Passenger

3.3.3. RideRequest

Class Name: RideRequest **Super Class: -**	
Represents a request made by a passenger to get a ride.	
Responsibilities	**Collaborators**
● Create and store ride requests from passengers	Passenger
● Capture ride details including pickup, drop-off locations, and passenger information	Passenger
● Find and match an available driver	Driver
● Manage ride request acceptance and rejection	Driver
● Notify the passenger and driver about the request status	Ride, Notification
● Handle fare estimation and pricing calculations	Invoice
● Generate a ride summary upon ride completion	Report, Rate

3.3.4. Invoice

Class Name: Invoice **Super Class: -**	
Represents billing details for a completed ride.	
Responsibilities	**Collaborators**
● Generate a unique invoice ID for each completed ride	NA
● Calculate and store the final fare, including surcharges and discounts	RideRequest
● Link invoice details to the corresponding ride and payment transaction	PaymentMethod
● Send payment confirmation to users	Passenger, Notification
● Store transaction history	Database

3.3.5. Rate

Class Name: Rate **Super Class: -**	
Stores passenger ratings for drivers and feedback comments, helping maintain service quality	
Responsibilities	**Collaborators**
● Knows rating value given by the passenger	Passenger
● Can determine the driver's overall performance status based on rating trends	Report
● Can calculate the average rating for each driver	Database
● Drivers can view their rates	Database, Report

3.3.6. Notification

Class Name: Notification **Super Class: -**	
Handles all system notifications, including ride updates, confirmations, and payment alerts.	
Responsibilities	**Collaborators**
• Can send ride confirmation to passenger	Passenger
• Can notify drivers of new ride request	Driver
• Can send ride status updates	Passenger, Driver
• Can send payment confirmation	PaymentMethod
• Can send ride completion notification	Passenger

3.3.7. PaymentMethod

Class Name: PaymentMethod **Super Class: -**	
Processes ride payments, validate transactions, and handle payment confirmations.	
Responsibilities	**Collaborators**
● Can process and validate payments (handle different methods, verify details, and complete transactions)	Database, Passenger
● Can select different payment methods (debit card, credit card, cash)	Passenger
● Can handle transaction failures and retries	Database
● Can generate and store payment receipts	Invoice, Database
● Can notify passengers about payment status	Notification

3.3.8. Database

Class Name: Database **Super Class: -**	
Stores and manages all system data, including rides, payments, user, and ratings	
Responsibilities	**Collaborators**
• Can stores ride request and active ride data	RideRequest, Ride
• Can store user data (passenger, driver, manager)	Passenger, Driver, Manager
• Can store payment transaction records	PaymentMethod
• Can store ratings and feedback	Rate
• Can retrieve data for reports	Report

3.3.9. Report

Class Name: Report **Super Class: -**	
Generates analytical reports based on ride data, payments, and driver ratings for management	
Responsibilities	**Collaborators**
● Can generate ride history reports	Database
● Can provide financial summaries	Database, PaymentMethod
● Can aggregate driver ratings and feedback	Rate, Database
● Can generate performance reports for managers	Manager
● Can retrieve and format for reports	Database

4. Design Quality

4.1. Responsibilities Coverage

Each class in the SmartRide system is designed to handle specific responsibilities effectively. The key responsibilities covered by the classes include:

- The **User** class manages user information and authentication.
- The **RideRequest** class handles ride bookings and driver assignments.
- The **Ride** class tracks the status of an active ride.
- The **PaymentMethod** class processes ride payments and transactions.
- The **Report** class generates analytics and insights for management.

4.2. Responsibilities Described Consistently and Precisely

The responsibilities of each class are well-defined and consistently described. Each class has a distinct role, avoiding redundancy and overlapping functionalities. For example, the **PaymentMethod** class handles payment processing and transactions, ensuring seamless financial operations.

4.3. Responsibilities Are Evenly Distributed Without Creating 'God' Classes

Responsibilities are evenly distributed among the system's classes, ensuring that no single class handles an excessive number of operations. Each class is designed to focus on a specific aspect of the system, promoting modularity and maintainability.

4.4. Design Heuristics

H1. A class should represent a single concept

H2. Encapsulate data and restrict direct access

H3. Base class attributes should not be exposed

H4. Maintain consistency across system components

H5. Provide clear and actionable error messages

H6. Prevent unintended method overriding

H7. Avoid overloaded 'God' classes

H8. Abstract classes should define common behavior

H9. Abstract classes must only be used as base classes

5. Design Pattern

5.1 Creational Patterns

5.1.1 Factory Method Design Pattern

The Factory Method pattern is useful for creating different types of User objects (Passenger, Driver, Manager). Instead of directly instantiating objects using new, a UserFactory class will handle the creation of user objects based on their role.

This pattern helps manage different user types in a structured way while ensuring scalability. If additional roles (e.g., Admin, SupportStaff) are introduced in the future, they can be integrated seamlessly into the factory without altering other parts of the code.

Implementation:

- Create an abstract User class.
- Implement subclasses: Passenger, Driver, and Manager.
- Use UserFactory with a method createUser(role: String): User to instantiate the correct object.

5.2 Behavioural Patterns

5.2.1 Observer Design Pattern

The Observer pattern is suitable for implementing the NotificationService. This pattern ensures that Passenger and Driver are notified of events such as ride confirmations, cancellations, and updates.

Whenever RideRequest is updated (e.g., assigned to a driver, canceled by a passenger), NotificationService will automatically notify the relevant users without requiring direct method calls.

Implementation:

- RideRequest acts as a **subject** that maintains a list of observers.
- Passenger and Driver implement the observer interface and subscribe to updates.

- When a ride status changes, the observers are notified automatically.

5.3 Structural Patterns

5.3.1 Adapter Design Pattern

The **Adapter Pattern** is useful for integrating third-party or external navigation services (Google Maps API as in this project context) with the existing **Route** and **Ride** classes. Since external APIs may have different interfaces, an adapter acts as a **bridge** between the system and the external service.

This pattern ensures that the system remains flexible when dealing with external dependencies, reducing the impact of future API changes.

Implementation:

- **Create a NavigationAdapter** class that **translates** method calls from the system into the required format for the external service.
- The **NavigationSystem** class interacts with the adapter rather than the external service directly.
- The adapter implements a common interface, allowing easy replacement or extension with other navigation providers in the future.

6. Bootstrap Process

This bootstrapping process reflects a customer booking a ride:

- The system begins with User registration
- Drivers provide their Location information
- Passengers register and set up their profiles
- Passengers establish Payment Methods
- A Passenger initiates a Ride Request
- The system calculate route for Ride Request
- The Ride Request checks for Driver availability based on Location
- Notifications are created and sent to available Driver
- When a Driver accepts, the Ride Request creates a Ride entity
- The Ride maintains a dependency with Location for real-time updates
- The system sends Notifications to the Passenger about ride status
- The system creates an Invoice for the ride

- Upon completion, the Ride triggers payment processing using the Passenger's Payment Method
- The Rate entity is created for both Passenger and Driver to provide feedback

7. Verification

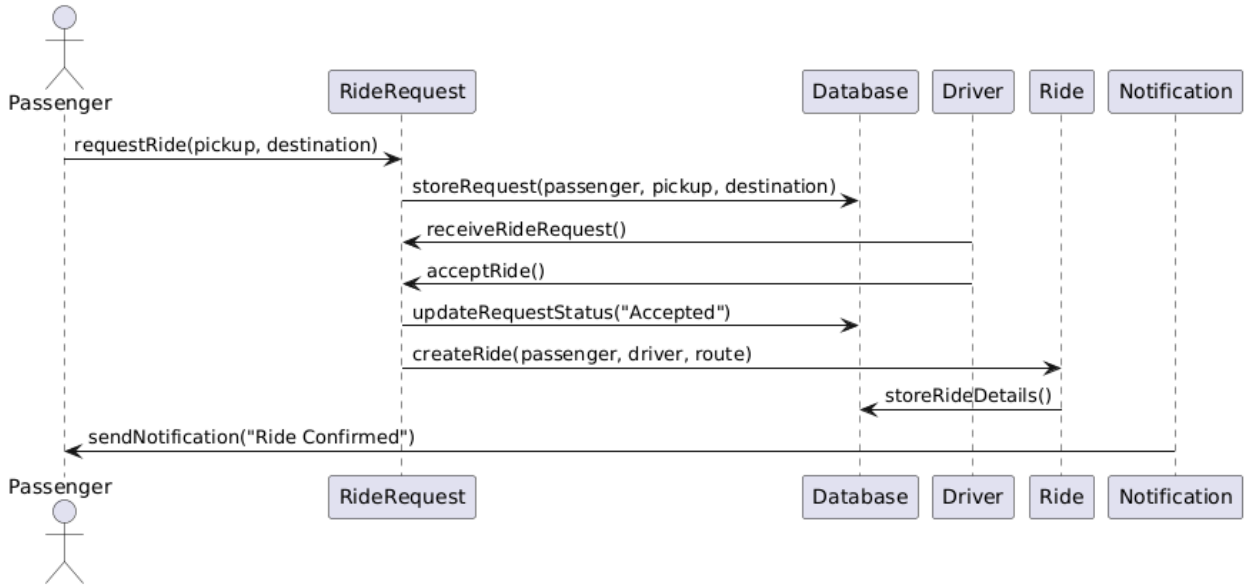
7.1. Scenario 1: Booking a Ride

Description: A passenger requests a ride, and the system assigns a driver. The passenger receives confirmation once a driver accepts.

Steps:

1. The **Passenger** submits a ride request.
2. The **RideRequest** is created and stored in the **Database**.
3. The available **Driver** receives a notification about the ride request.
4. The **Driver** accepts or rejects the request.
    - If rejected, the ride request is reassigned.
5. Once accepted, the **Ride** is created and updated in the **Database**.
6. The **Passenger** receives a confirmation via **Notification**.

UML:



7.2. Scenario 2: Ride Tracking

Description**:** Once the ride starts, the driver's location is tracked and updated for the passenger. The system ensures the driver picks up the passenger and does not exceed the destination.

Steps:

Before Pickup:

1. The **Driver** shares their GPS location with the **Ride**.
2. The **Ride** updates the **Database** with the driver's location.
3. If the driver reaches the pickup location, the **Passenger** receives a notification.
4. The **Passenger** enters the vehicle, and the **Driver** marks the ride as "Started."

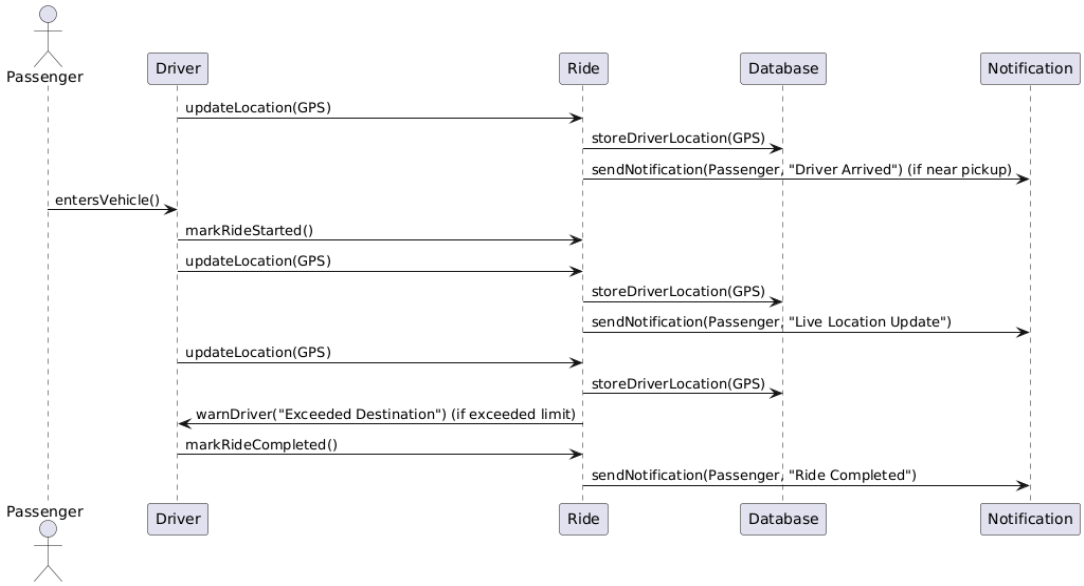
During the Ride:

5. The **Ride** updates the **Database** with the latest driver location.
6. The **Passenger** receives live location updates via **Notification**.

At Drop-Off Location:

7. The **Ride** checks if the driver is near the destination.
8. If the driver exceeds the drop-off point, they receive a warning.
9. The **Driver** marks the ride as "Arrived".
10. The **Passenger** receives a notification confirming ride completion.

UML:



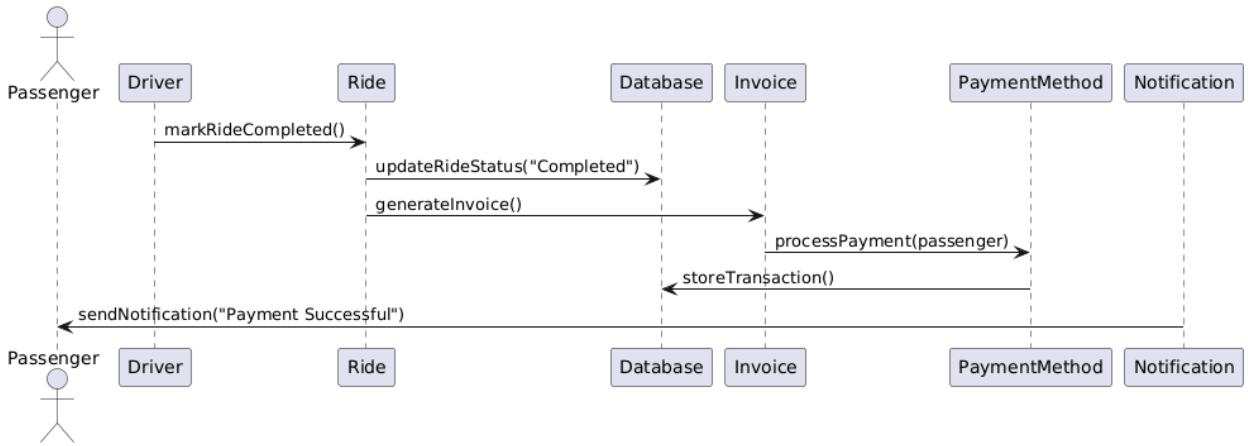
7.3. Scenario 3: Completing the Ride & Payment Processing

Description:After reaching the destination, the system generates an invoice and processes the passenger's payment.

Steps:

1. The **Driver** confirms the ride completion.
2. The **Ride** updates the **Database** with the completed status.
3. The **Invoice** is generated based on the ride details.
4. The **Passenger**'s **PaymentMethod** is charged.
5. The **PaymentMethod** updates the **Database** with the transaction details.
6. The **Passenger** receives a **Notification** with payment confirmation.

UML:



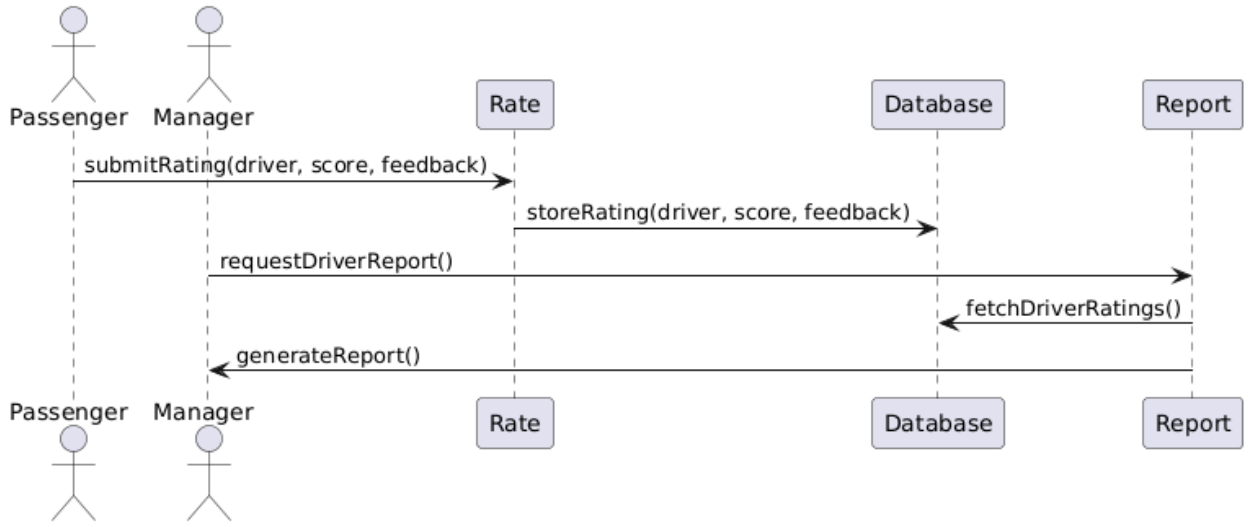
7.4. Scenario 4: Submitting a Rating and Feedback

Description: The passenger provides feedback and a rating for the driver. The manager can generate reports using this data.

Steps:

1. The **Passenger** submits a rating and feedback.
2. The **Rate** stores the rating in the **Database**.
3. The **Manager** requests a report of driver performance.
4. The **Report** fetches the necessary data from the **Database** and generates the report.

UML:



8. Reference

Object-Oriented Design. (n.d.).

https://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/chap03.pdf