TABLE OF CONTENT

****List of Figures****

|  |  |  |
| --- | --- | --- |
| I | Ride-Hailing Client–Server Architecture | 3 |
| II | Ride Booking Use Case Diagram | 6 |
| III | Ride-Hailing System Class Diagram | 8 |
| IV | Rider & Driver Management Package Class Diagram | 9 |
| V | Booking and Matching Package Class Diagram | 10 |
| VI | Payment and Commission Package Class Diagram | 11 |
| VII | Ride Booking Activity Diagram with Swim Lane | 14 |
| VIII | Ride Booking Activity Diagram without Swim Lane | 16 |
| IX | Ride Booking Sequence Diagram | 18 |

****Chapter One — Introduction****

|  |  |  |
| --- | --- | --- |
| 1.1.0 | Project Introduction |  |
| 1.2.0 | Problem Statement |  |
| 1.3.0 | Project Objectives |  |
| 1.4.0 | Scope of the Project |  |
| 1.5.0 | Tools and Technologies Used |  |
| 1.6.0 | System Overview Diagram |  |

****Chapter Two — Requirements a****

|  |  |  |
| --- | --- | --- |
| 2.1.0 | System Analysis |  |
| 2.2.0 | Identification of Actors (Rider, Driver, Admin, APIs) |  |
| 2.3.0 | Identification of Use Cases |  |
| 2.4.0 | Use Case Diagram for Ride Booking |  |
| 2.5.0 | Class Diagrams: |  |
| 2.5.1 | Ride-Hailing System Class Diagram |  |
| 2.5.2 | Rider & Driver Management component diagram |  |
| 2.5.3 | Booking and Matching Package Class Diagram |  |
| 2.5.4 | Payment and Commission Package Class Diagram |  |
| 2.6.0 | Flow of Events — Ride Booking |  |
| 2.7.0 | Activity Diagram with Swim Lane |  |
| 2.8.0 | Activity Diagram without Swim Lane |  |

****Chapter Three — Detailed Design and Communication Flow****

|  |  |  |
| --- | --- | --- |
| 3.1.0 | Sequence Diagram — Ride Booking Flow |  |
| 3.2.0 | Sequence Diagram — Payment and Payout Flow |  |
| 3.3.0 | REST API and WebSocket Overview |  |
| 3.4.0 | Main Classes and Attributes |  |
| 3.5.0 | Key Algorithms (Driver Matching, Commission Split, OTP) |  |

****Chapter Four — Implementation Details****

|  |  |  |
| --- | --- | --- |
| 4.1.0 | Frontend (Angular) Design and Navigation Flow |  |
| 4.2.0 | Backend (Spring Boot) Module Structure |  |
| 4.3.0 | Database Schema and Entity Relationships |  |
| 4.4.0 | Integration with Google Maps API and Razorpay Test Mode |  |
| 4.5.0 | Notification Handling (WebSocket, Email, SMS) |  |

# **CHAPTER ONE**

### **1.1.0 PROJECT INTRODUCTION**

Urban X is a **ride-hailing application** designed to bridge the gap between passengers and drivers in urban environments. Much like Uber or Ola, the system provides a platform where users (riders) can book rides through a mobile or web interface, while drivers can accept bookings, complete trips, and receive payments.

The project uses a **client–server architecture**:

* **Frontend (Angular)**: Provides user-friendly interfaces for riders, drivers, and admins.
* **Backend (Spring Boot)**: Implements business logic, APIs, authentication, payment integration, and ride management.
* **Database**: Stores information about users, rides, payments, and vehicles.
* **External APIs**: Google Maps API for GPS tracking and navigation, Razorpay for test-mode payments.

Urban X focuses on **real-time ride matching**, **secure digital payments**, **GPS-enabled navigation**, and **transparent commission processing** for drivers and the platform.

### **1.2.0 PROBLEM STATEMENT**

Urban transportation in developing cities faces several issues:

* **Lack of transparency in fares**: Riders often don’t know the estimated cost before a ride.
* **Unreliable availability**: Finding a safe, verified ride at the right time can be difficult.
* **Safety concerns**: Riders often don’t have emergency support or tracking options.
* **Cash dependency**: Many systems rely heavily on cash, which limits digital record-keeping.
* **Operational inefficiency**: Without a centralized system, admins cannot track vehicles, drivers, or commissions in real time.

Urban X addresses these challenges by offering:

* Automated **fare estimation and digital receipts**.
* **Instant ride matching** using driver proximity and availability.
* **OTP verification** before ride start for rider safety.
* **Wallet, card, and UPI payments** for cashless transactions.
* **Admin dashboards** for complete monitoring of operations.

### **1.3.0 PROJECT OBJECTIVES**

The main objectives of Urban X are:

1. **Enable real-time ride booking** with pickup, drop-off, and ride-type selection.
2. **Provide GPS-based driver tracking** for transparency and security.
3. **Support multiple payment methods** (wallet, card, UPI, cash).
4. **Implement commission management** so the platform automatically deducts its share.
5. **Allow users to rate and review drivers** for quality assurance.
6. **Offer role-based access**: riders, drivers, and admins have different dashboards.
7. **Enhance safety** with SOS features and ride OTP verification.
8. **Ensure scalability** so the platform can grow to handle thousands of concurrent users.

### **1.4.0 SCOPE OF THE PROJECT**

The **scope** of Urban X is both functional and technical.

**Functional Scope:**

* **Riders** can register, book rides, view ride history, make payments, and rate drivers.
* **Drivers** can manage availability, accept rides, update ride status, and track earnings.
* **Admins** can monitor ongoing rides, manage users and vehicles, generate reports, and configure commissions.

**Technical Scope:**

* Web and mobile interface built with **Angular**.
* Backend developed with **Spring Boot** REST APIs.
* Integration with **Google Maps API** for location services.
* Integration with **Razorpay (Test Mode)** for online payments.
* Notifications through **WebSockets, email, and SMS**.

**Out of Scope (for now):**

* Advanced AI-based route optimization.
* Multi-city or international deployment.
* Multi-language support (future enhancement).

### **1.5.0 TOOLS AND TECHNOLOGIES USED**

**Development Tools:**

* **Spring Boot (Java)**: Backend framework for APIs, authentication, and payment handling.
* **Angular (TypeScript)**: Frontend framework for building interactive, responsive UI.
* **Hibernate/JPA**: ORM for database interaction.
* **MySQL/PostgreSQL**: Relational database for storing rides, users, payments.

**APIs & Integrations:**

* **Google Maps API** – Location tracking, route estimation, and map visualization.
* **Razorpay (Test Mode)** – Payment processing for digital transactions.
* **Twilio / SMTP Email / SMS Gateway** – Notifications and alerts.
* **WebSocket** – Real-time updates for ride booking and status.

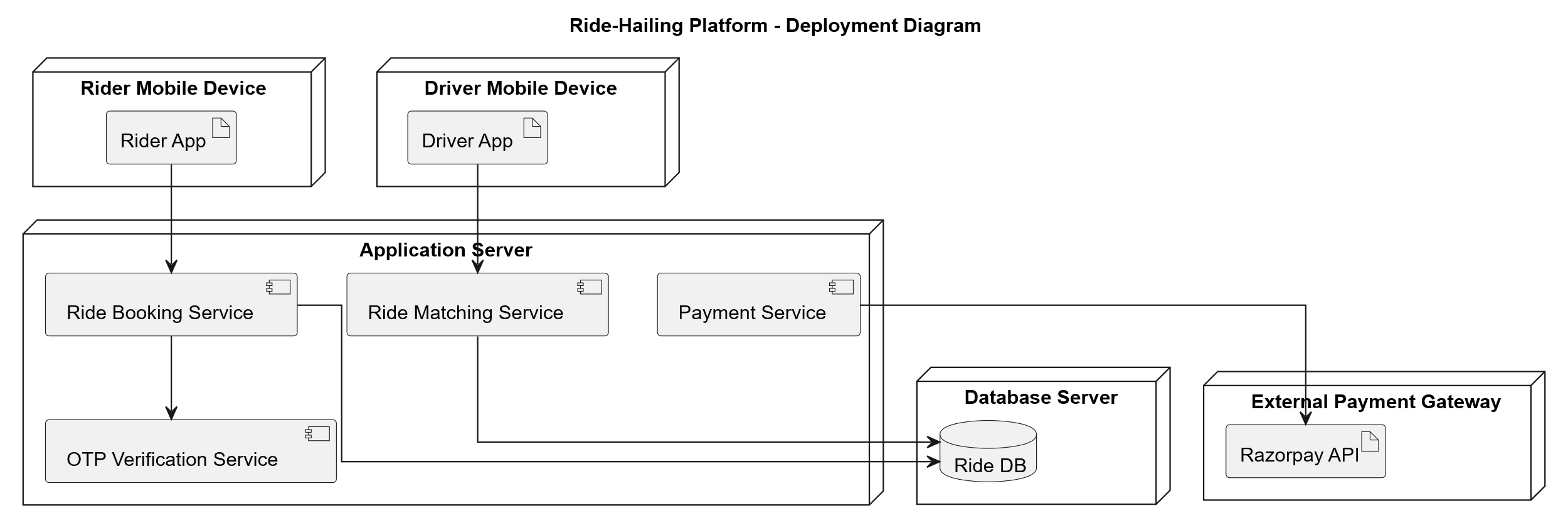
**Supporting Tools:**

* **Draw.io / Lucidchart** – UML diagrams.
* **GitHub / GitLab** – Version control and collaboration.
* **Postman** – API testing.

### **1.6.0 SYSTEM OVERVIEW DIAGRAM**

The Urban X system is designed as a **distributed client–server architecture**. The system consists of three main user groups: **Riders**, **Drivers**, and **Admins**. Each interacts with the system through dedicated interfaces built with Angular, while the backend (Spring Boot) provides the business logic and manages secure communication with the database and third-party APIs.

* **Rider App (Angular)**: Allows riders to register, request rides, make payments, and track trip progress.
* **Driver App (Angular)**: Enables drivers to register, manage vehicle details, accept or reject ride requests, and receive payouts.
* **Admin Dashboard (Angular)**: Provides tools for admins to manage rides, users, payments, and commissions.
* **Spring Boot Backend**: Handles core logic, ride matching, OTP verification, commission calculation, payments, and notifications.
* **Database (MySQL/PostgreSQL)**: Stores persistent data including users, vehicles, rides, payments, and ratings.
* **External Services**:
  + Google Maps API for navigation and live tracking
  + Razorpay for payment integration
  + SMS/Email gateways for notifications



**System Overview Architecture Diagram**

# **CHAPTER TWO — Requirements Analysis**

### **2.1.0 SYSTEM ANALYSIS**

The Urban X platform is designed to address the limitations of traditional taxi-hailing systems. By combining **real-time location tracking**, **automated fare calculation**, and **digital payments**, Urban X ensures an efficient, transparent, and user-friendly experience.

Key considerations in the analysis:

* **Performance**: APIs must respond within 2–3 seconds to ensure smooth booking flow.
* **Scalability**: The platform should scale horizontally to support thousands of simultaneous ride requests.
* **Security**: Secure user authentication (JWT), encrypted payment data, and OTP-based ride verification are required.
* **Usability**: Both rider and driver applications must provide simple navigation and real-time ride status updates.
* **Reliability**: System must be available 24/7 with 99.5% uptime.

This analysis highlights the need for a modular architecture with clearly separated components (booking, payments, notifications, and reporting), each maintained independently for scalability.

### **2.2.0 IDENTIFICATION OF ACTORS (RIDER, DRIVER, ADMIN, APIS)**

The main **actors** in the Urban X system are:

1. **Rider**
   * Registers and maintains profile.
   * Requests rides by entering pickup and destination.
   * Cancels rides if needed.
   * Makes payments (wallet, card, UPI, or cash).
   * Rates and reviews drivers after completion.
2. **Driver**
   * Registers with valid documents and vehicle details.
   * Accepts or rejects ride requests.
   * Starts and ends rides using OTP confirmation.
   * Views ride history and earnings.
3. **Admin**
   * Manages user accounts (rider/driver).
   * Oversees live rides in progress.
   * Configures commission percentage.
   * Generates reports (earnings, cancellations, complaints).
4. **External APIs**
   * **Google Maps API** → Provides route, distance, ETA, and GPS updates.
   * **Razorpay Payment Gateway** → Handles payment authorization, confirmation, and refunds.
   * **Notification Services** → SMS/Email/WebSocket for updates to riders and drivers.

### **2.3.0 IDENTIFICATION OF USE CASES**

The primary **use cases** in Urban X are:

#### **For Rider:**

* UC-01: Register/Login.
* UC-02: Book Ride (select type, pickup, drop, confirm OTP).
* UC-03: Cancel Ride.
* UC-04: Make Payment.
* UC-05: Rate and Review Driver.
* UC-06: View Ride History.
* UC-07: Raise SOS/Emergency Alert.

#### **For Driver:**

* UC-08: Register/Login with Vehicle Details.
* UC-09: Accept/Reject Ride.
* UC-10: Start Ride with OTP Verification.
* UC-11: End Ride and Confirm Fare.
* UC-12: View Earnings and Commission Deductions.
* UC-13: Manage Vehicle Details.

#### **For Admin:**

* UC-14: Manage Users (approve/reject drivers, suspend riders).
* UC-15: Monitor Live Rides and Status.
* UC-16: Configure Commission Rates.
* UC-17: Generate Reports (earnings, cancellations, complaints).

#### **For External APIs:**

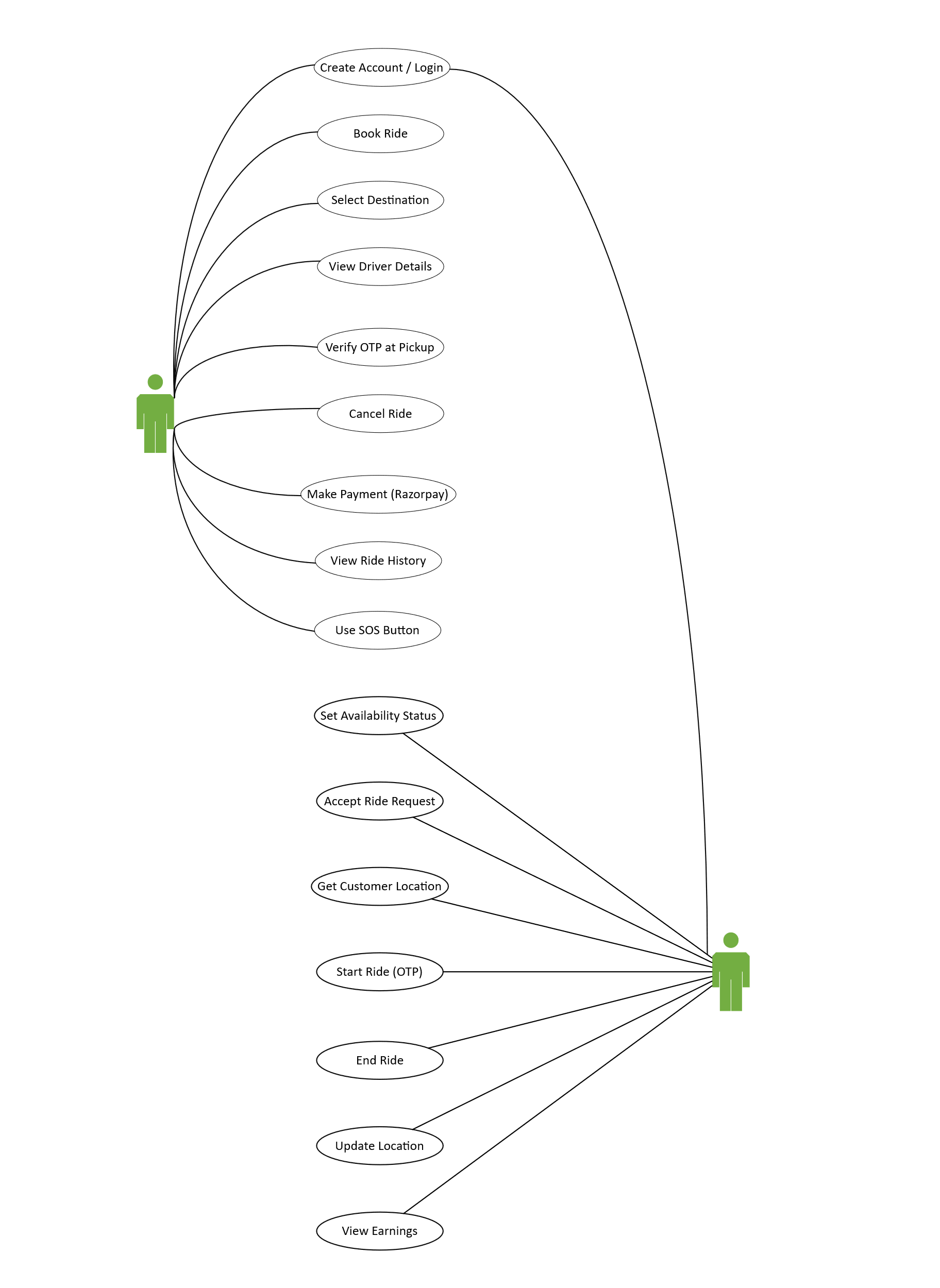
* UC-18: Process Payments (Razorpay).
* UC-19: Provide Navigation & GPS Tracking (Google Maps API).
* UC-20: Send Notifications (Email/SMS/WebSocket).

### **2.4.0 USE CASE DIAGRAM FOR RIDE BOOKING**

The **Ride Booking Use Case Diagram** illustrates the interaction between **Rider**, **Driver**, and the **System** during the ride booking process.

Key actions include:

* Rider initiates a ride request.
* System finds nearest driver.
* Driver accepts/rejects the request.
* System generates OTP for secure pickup.
* Ride begins once OTP is verified.
* Ride ends with payment processing.

**Use Case Diagram for Ride Booking**

## **2.5.0 CLASS DIAGRAMS**

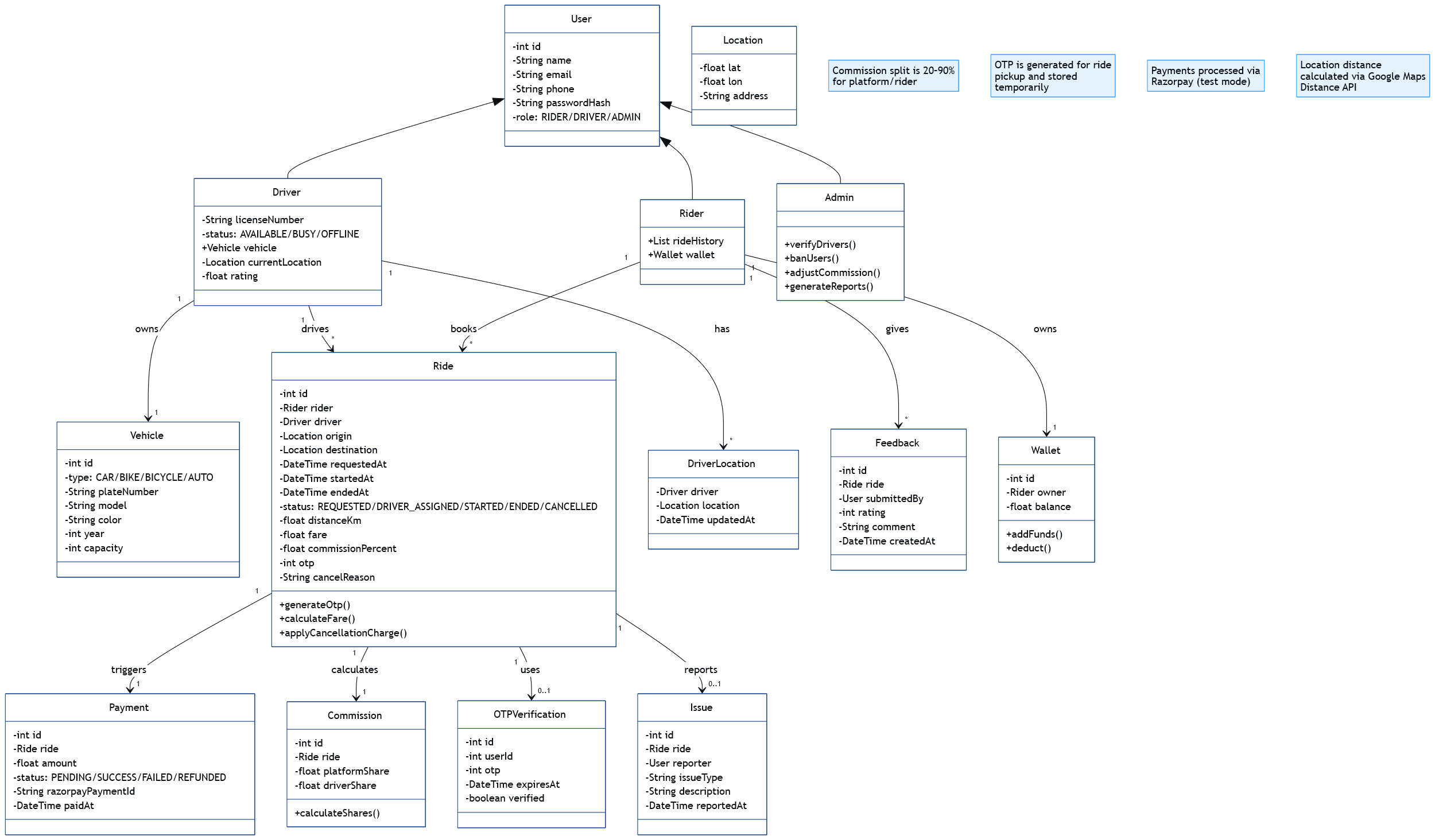
Class diagrams are one of the most important UML representations for the Urban X system. They capture the **static structure** of the application by showing entities, their attributes, and relationships among them. Since Urban X is built on **Spring Boot (backend)** and **Angular (frontend)**, the class diagrams help developers clearly visualize the backend model classes, frontend DTOs, and how they are mapped to database entities.

### **2.5.1 RIDE-HAILING SYSTEM CLASS DIAGRAM**

This diagram illustrates the **overall system entities** and how they are connected. The central classes include **User**, **Rider**, **Driver**, **Vehicle**, **Ride**, **Payment**, **CommissionRecord**, and **Location**.

* **User** acts as a parent class for Rider and Driver, containing basic details such as id, name, phone, email, and role.
* **Rider** extends User and includes methods like requestRide() and cancelRide().
* **Driver** extends User and includes methods like acceptRide() and completeRide().
* **Ride** holds attributes such as rideId, status, fare, pickupLocation, dropoffLocation, and otp.
* **Payment** links directly with Ride to process transactions using different payment methods.
* **CommissionRecord** ensures that every completed ride has a percentage of fare deducted for the platform.
* **Location** keeps track of GPS data during the trip.

This diagram provides the **foundation of the database schema** and is critical for mapping JPA entities in Spring Boot.

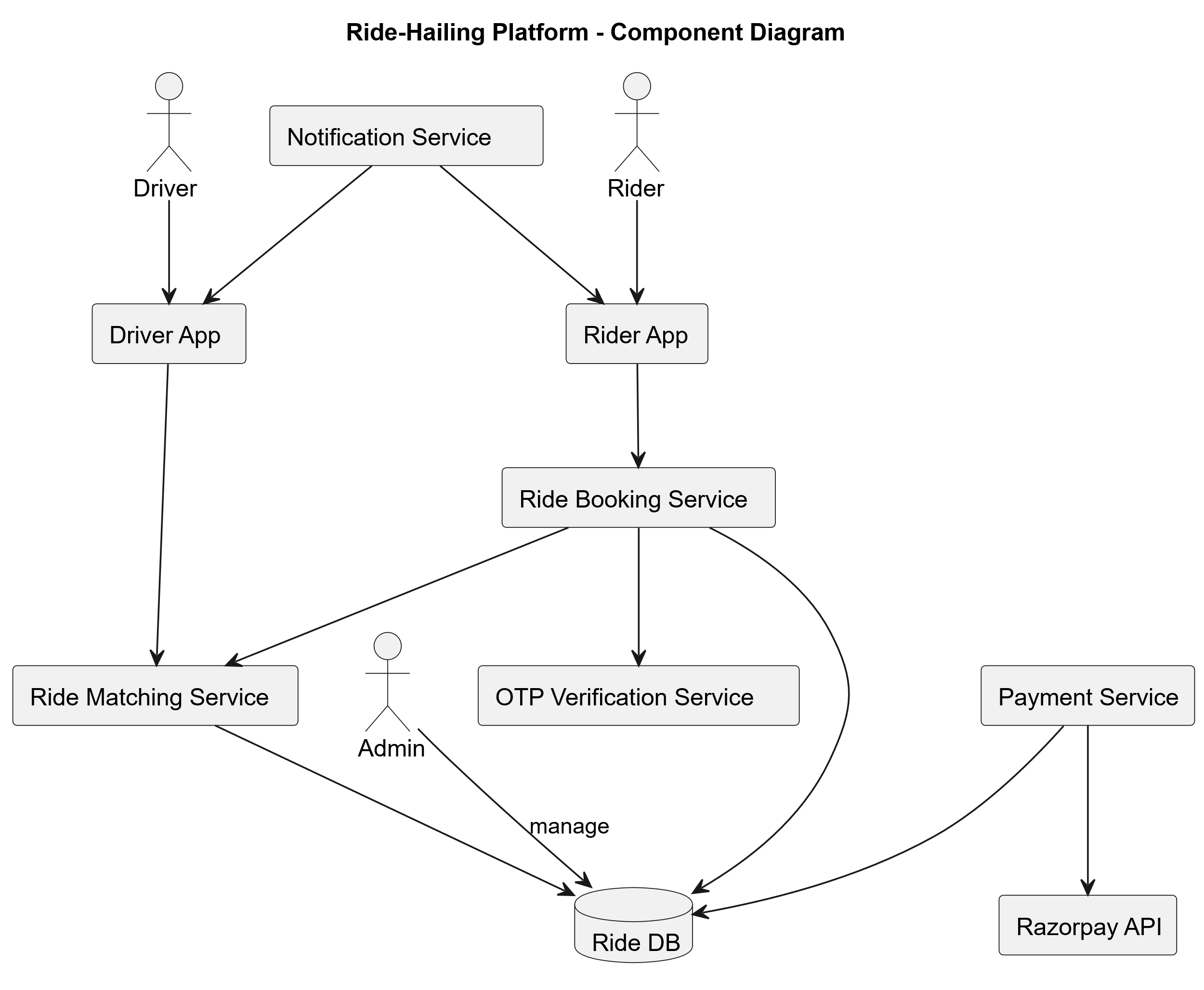
**Class Diagram:**

### **2.5.2 RIDER & DRIVER MANAGEMENT COMPONENT DIAGRAM**

This package diagram focuses on **user registration, authentication, and vehicle management**. It covers how the **Rider** and **Driver** classes interact with supporting classes for account creation and role-specific operations.

* **RiderProfile**: Contains rider-specific attributes (rideHistory, walletBalance, SOS settings).
* **DriverProfile**: Includes licenseNo, vehicleAssigned, and driverRating.
* **Vehicle**: Connected to DriverProfile, storing information such as vehicleId, type, registration number, and insurance details.
* **AuthenticationService**: Manages login, JWT token issuance, and role-based landing pages.

This diagram ensures separation of concerns: Rider and Driver features are clearly managed in distinct subpackages while still inheriting from a common **User** entity.

**Rider & Driver Management component diagram:**

### **2.5.3 BOOKING AND MATCHING PACKAGE CLASS DIAGRAM**

The booking and matching package diagram describes how ride requests are created, matched with drivers, and tracked in the system.

* **BookingRequest**: Represents a new ride request containing pickup, destination, ride type, and OTP.
* **MatchingEngine**: Implements the algorithm to find the nearest available driver using GPS coordinates.
* **RideManager**: Controls ride lifecycle events — requested, accepted, in-progress, completed, cancelled.
* **NotificationService**: Sends updates to riders and drivers via WebSocket, SMS, or email.

By separating booking and matching into its own package, the architecture allows future improvements such as **AI-based driver assignment** or **dynamic surge pricing** without affecting the rest of the system.

### **2.5.4 PAYMENT AND COMMISSION PACKAGE CLASS DIAGRAM**

The payment and commission package ensures that all financial transactions are processed safely and that driver payouts are handled correctly.

* **PaymentProcessor**: Connects with Razorpay API to initiate, confirm, or cancel payments.
* **Wallet**: Stores prepaid rider balances and allows refunds or top-ups.
* **CommissionCalculator**: Deducts platform fees automatically after ride completion.
* **PayoutManager**: Maintains driver earnings after commission deductions and initiates weekly/monthly payouts.
* **TransactionLog**: Keeps an immutable record of all completed payments for auditing.

This package ensures financial transparency and security by separating business logic from payment API integration.

**Payment & Commission Package Diagram:**  
[[[[ Payment and Commission Package Class Diagram.svg ]]]]

## **2.6.0 FLOW OF EVENTS — RIDE BOOKING**

The ride booking use case begins when a **Rider** initiates a request by entering pickup and destination details. The **system** then estimates fare, assigns the nearest available driver, and generates an **OTP** to confirm identity at pickup. Once verified, the ride proceeds until completion, followed by payment and feedback.

**Normal Flow:**

1. Rider logs in and selects ride type (Bike, Car, SUV).
2. Rider enters pickup and destination locations.
3. The system calculates estimated fare and ETA.
4. Rider confirms booking.
5. MatchingEngine identifies the nearest driver.
6. Driver accepts the request.
7. The system generates OTP and shares with Rider.
8. At pickup, Rider shares OTP with Driver to start ride.
9. Ride is tracked in real-time via Google Maps API.
10. Ride ends, fare finalized, payment processed.
11. Commission deducted and driver earnings updated.
12. Rider and Driver both submit ratings.

**Alternate Flow:**

* If no driver is available, the system notifies Rider with “No drivers nearby.”
* If Rider cancels before confirmation, no payment is processed.
* If Driver cancels after acceptance, system reassigns another driver.

**Activity Diagram with Swim Lane:**

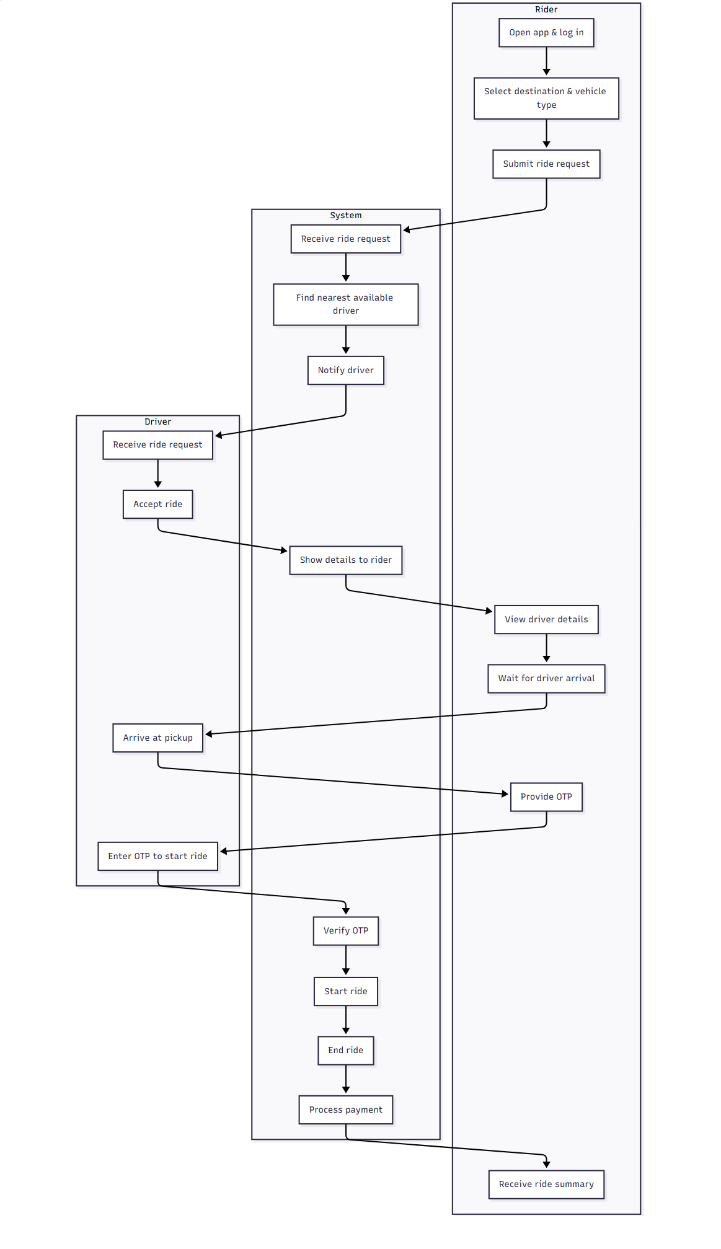
## **2.7.0 ACTIVITY DIAGRAM WITH SWIM LANE**

The **Activity Diagram with Swim Lane** demonstrates the **step-by-step flow of a ride booking** in Urban X and clearly separates responsibilities among different actors: **Rider, Driver, Backend System, and Payment Gateway**. Each swim lane shows actions specific to one actor, making it easy to understand how responsibilities are distributed.

**Flow Description:**

1. The **Rider** initiates the process by logging in and selecting pickup and destination.
2. The **System** estimates fare and ETA, then sends a booking request to available drivers.
3. The **Driver** receives the request and either accepts or rejects it.
4. If accepted, the **System** generates and shares an OTP with the Rider.
5. At pickup, the **Driver** enters the OTP to start the ride.
6. The ride is tracked in real-time, with continuous location updates.
7. On ride completion, the **Payment Gateway** processes the payment and updates driver earnings.
8. Finally, both Rider and Driver provide ratings.

This diagram makes it visually clear how multiple stakeholders coordinate during the ride booking and payment process.

**Activity Diagram with Swim Lane:**

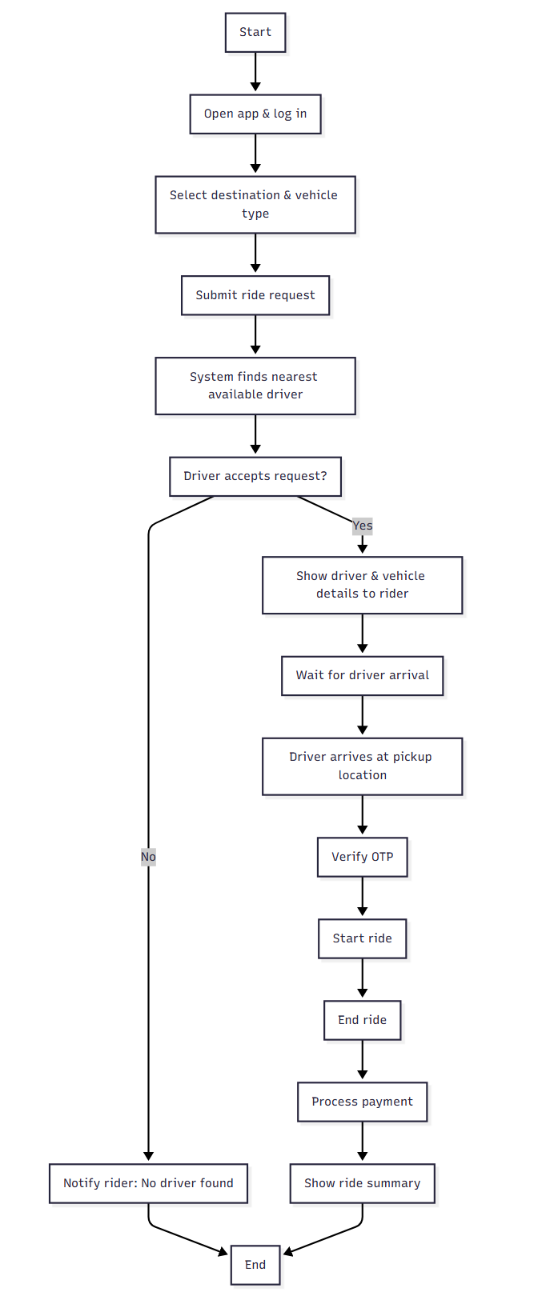
## **2.8.0 ACTIVITY DIAGRAM WITHOUT SWIM LANE**

The **Activity Diagram without Swim Lane** presents the same ride booking process but without dividing responsibilities into lanes. Instead, it shows the workflow as a **single continuous sequence of actions**, focusing on the **logical flow of events** rather than which actor performs each step.

**Flow Description:**

1. Rider logs in and requests a ride.
2. The system estimates fare and finds the nearest available driver.
3. Driver accepts request, OTP is generated.
4. OTP verification occurs at pickup.
5. Ride is started, tracked, and then completed.
6. Payment is processed automatically.
7. Ride ends with ratings from both Rider and Driver.

This version of the activity diagram is helpful for stakeholders who want to understand the **overall system flow** without going into actor-specific details.

**Activity Diagram without Swim Lane:**

# **CHAPTER THREE — DETAILED DESIGN AND COMMUNICATION FLOW**

## **3.1.0 SEQUENCE DIAGRAM — RIDE BOOKING FLOW**

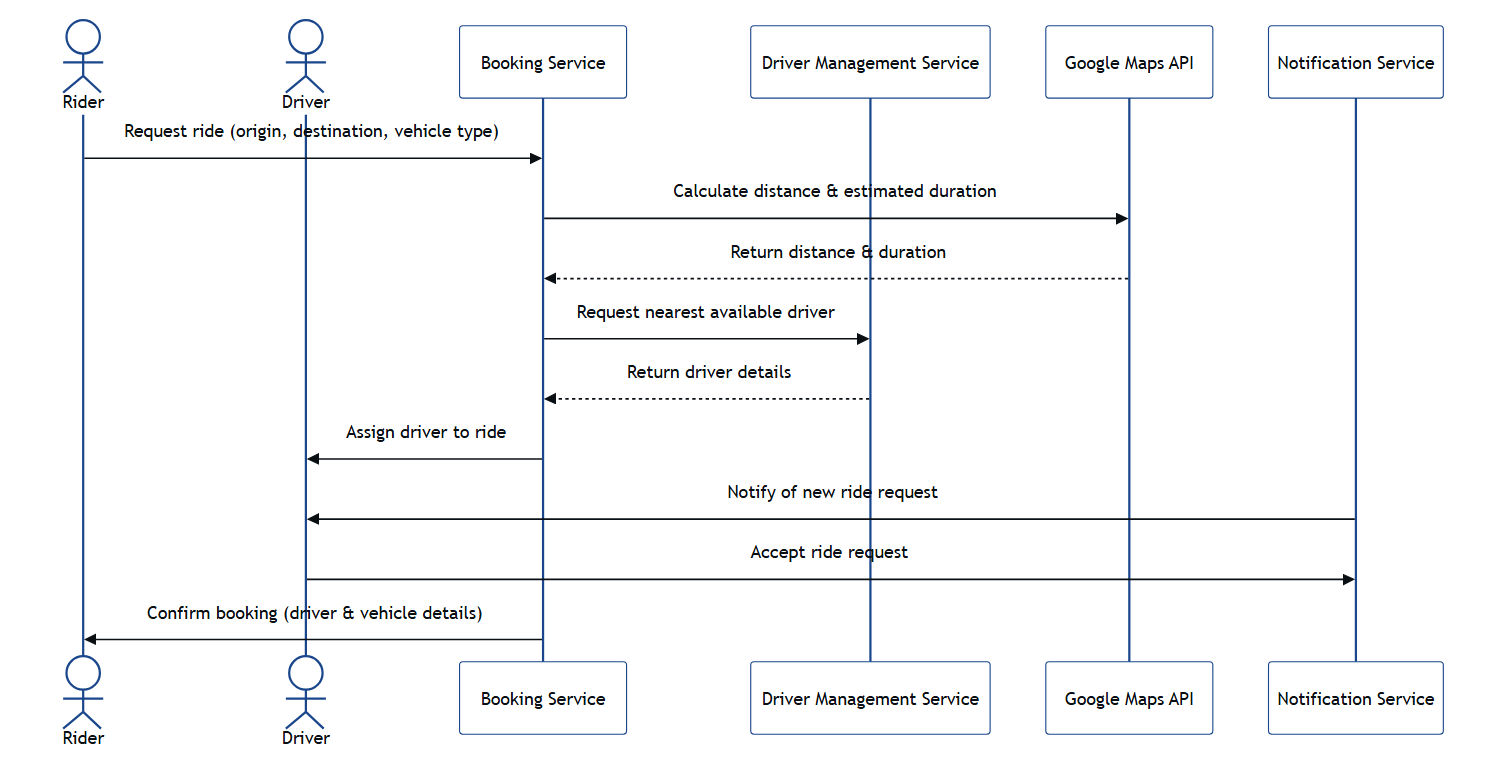
The **Ride Booking Flow Sequence Diagram** illustrates the chronological interaction between **Rider, System (Spring Boot backend), Driver, and External APIs** during the process of booking a ride.

**Steps in the Flow:**

1. The **Rider** initiates a ride booking by submitting pickup and destination details.
2. The **System** calculates fare and finds available drivers.
3. A **Driver** is notified and can accept or reject the request.
4. If accepted, the **System** generates an OTP and shares it with the Rider.
5. Ride is confirmed and tracking begins until completion.

This sequence diagram helps visualize the exact request–response communication across all components.

**Sequence Diagram:**



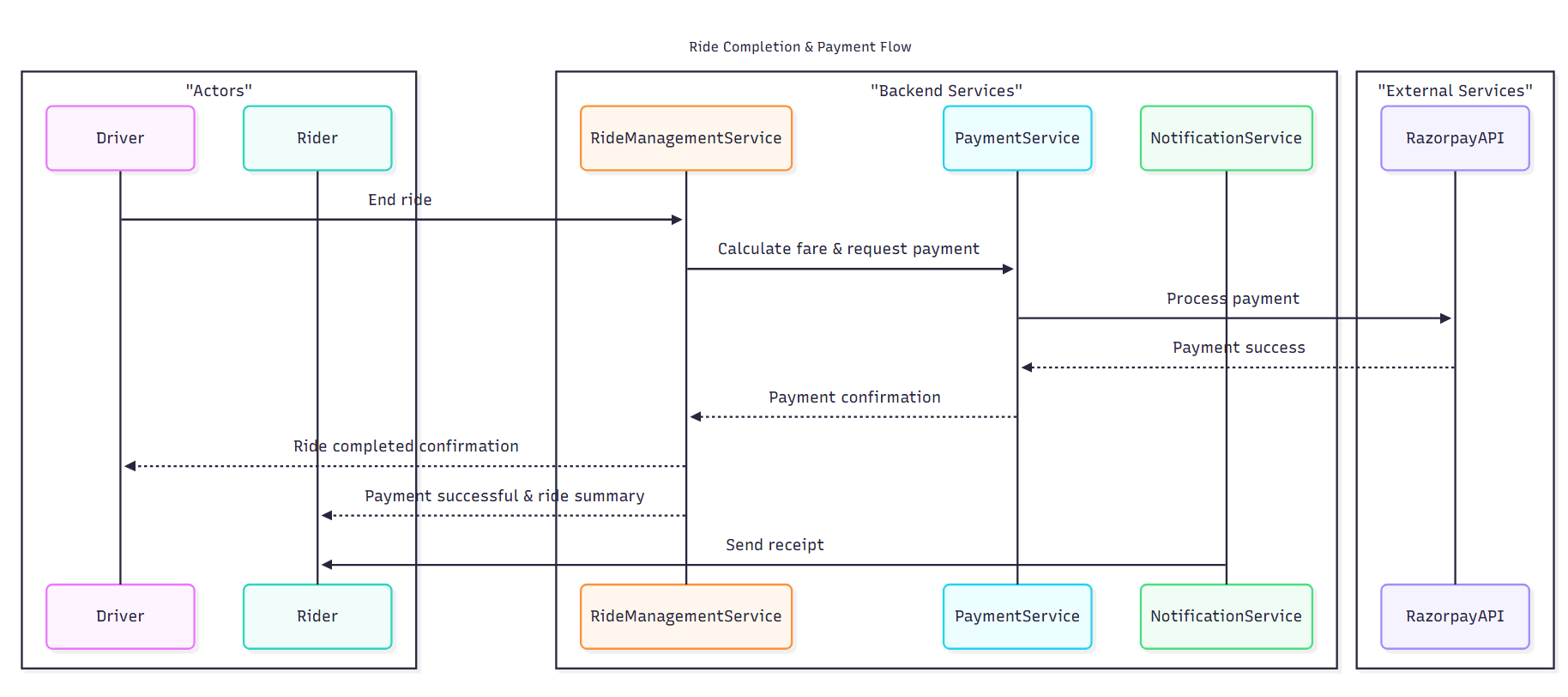
## **3.2.0 SEQUENCE DIAGRAM — PAYMENT AND PAYOUT FLOW**

The **Payment and Payout Flow Sequence Diagram** describes how the **Rider, Payment Gateway, System, and Driver** interact during fare settlement.

**Steps in the Flow:**

1. After ride completion, the **System** finalizes the fare.
2. Rider selects payment mode (wallet, card, UPI, or cash).
3. The **System** requests transaction authorization from the **Payment Gateway** (Razorpay).
4. On success, payment confirmation is sent to Rider and Driver.
5. The **Commission Module** deducts the platform fee.
6. Driver’s earnings are updated and stored for future payout cycles.

This ensures **secure transactions**, compliance with financial standards, and automatic **commission handling**.

**Sequence Diagram:**

## **3.3.0 REST API AND WEBSOCKET OVERVIEW**

Urban X is designed as a **REST-based system** with additional **WebSocket support** for real-time updates.

* **REST APIs (Spring Boot):**
  + POST /api/v1/rides → Create a new ride booking.
  + GET /api/v1/rides/{id} → Fetch ride details.
  + PATCH /api/v1/rides/{id}/cancel → Cancel an existing ride.
  + POST /api/v1/payments → Process payment for a completed ride.
  + GET /api/v1/drivers/{id}/earnings → Get driver’s earnings and commission details.
* **WebSocket Communication:**
  + Used for **real-time notifications** such as “Driver Found”, “Driver Arriving”, “Ride Started”, “Ride Completed”.
  + Enables push-based updates without requiring riders or drivers to refresh screens.
  + Ensures **low latency** for GPS location updates during rides.

This hybrid approach ensures both **scalability and responsiveness** in Urban X.

## **3.4.0 MAIN CLASSES AND ATTRIBUTES**

The key classes implemented in **Spring Boot backend** (mapped to database entities using JPA) are:

* **User**
  + Attributes: userId, name, phone, email, password, role, rating
  + Operations: register(), login(), updateProfile()
* **Rider (extends User)**
  + Attributes: walletBalance, rideHistory
  + Operations: requestRide(), cancelRide()
* **Driver (extends User)**
  + Attributes: licenseNo, vehicleAssigned, driverStatus
  + Operations: acceptRide(), completeRide(), viewEarnings()
* **Vehicle**
  + Attributes: vehicleId, type, registrationNo, capacity
  + Operations: updateDetails()
* **Ride**
  + Attributes: rideId, pickup, dropoff, status, otp, fare
  + Operations: startRide(), endRide(), cancelRide()
* **Payment**
  + Attributes: paymentId, rideId, amount, method, status
  + Operations: initiatePayment(), confirmPayment()
* **CommissionRecord**
  + Attributes: commissionId, rideId, percentage, amount
  + Operations: calculateCommission()

These classes form the **data model** and are directly mapped to backend tables.

## **3.5.0 KEY ALGORITHMS (DRIVER MATCHING, COMMISSION SPLIT, OTP)**

The following algorithms are central to Urban X operations:

1. **Driver Matching Algorithm**
   * Input: Rider’s pickup location, ride type.
   * Process:
     + Fetch available drivers within a given radius.
     + Calculate distance using Google Maps API.
     + Prioritize by **nearest driver** and **driver rating**.
   * Output: Selected driver assigned to ride request.
2. **Commission Split Algorithm**
   * Input: Ride fare, platform commission percentage.
   * Process:
     + Commission = fare × percentage.
     + DriverEarning = fare – commission.
   * Output: Stored CommissionRecord and updated driver balance.
3. **OTP Verification Algorithm**
   * Input: System-generated OTP, Rider-provided OTP.
   * Process: Compare Rider OTP with system value before starting ride.
   * Output: Ride start is allowed only if OTP matches..

****CHAPTER FOUR — IMPLEMENTATION DETAILS****

## **4.1.0 FRONTEND (ANGULAR) DESIGN AND NAVIGATION FLOW**

The frontend of Urban X is developed using **Angular** for its **component-based structure, reusability, and reactive forms**. Angular enables responsive layouts, modular design, and clear separation between rider, driver, and admin interfaces.

**Navigation Flow (simplified):**

* **Rider Application:**
  + Home → Book Ride → Enter Pickup & Destination → Confirm OTP → Live Ride Tracking → Payment → Feedback
* **Driver Application:**
  + Dashboard → Incoming Ride Requests → Accept/Reject → Start Ride (OTP) → Complete Ride → Earnings Dashboard
* **Admin Dashboard:**
  + Login → Manage Users → Monitor Rides → Reports & Analytics → Commission Settings

Angular **Routing** is used to control navigation between components. Services handle API calls to the backend, while guards protect authenticated routes.

## **4.2.0 BACKEND (SPRING BOOT) MODULE STRUCTURE**

The backend is implemented using **Spring Boot**, following a **layered architecture** for maintainability.

**Core Modules:**

1. **Authentication Module:**
   * Handles user registration, login, JWT-based authentication, and role-based authorization.
2. **Ride Management Module:**
   * Handles booking, driver matching, ride status updates, cancellations.
3. **Payment Module:**
   * Processes online payments, records commission splits, updates driver payouts.
4. **Notification Module:**
   * Sends ride status updates via WebSocket, SMS, and email.
5. **Admin Module:**
   * Provides APIs for monitoring, reporting, and managing users and vehicles.

Each module is exposed via REST APIs and integrated with the database through **JPA/Hibernate**.

## **4.3.0 DATABASE SCHEMA AND ENTITY RELATIONSHIPS**

The Urban X system uses **MySQL/PostgreSQL** as the relational database. The schema is designed for **normalization** to avoid redundancy while ensuring **efficient queries** for ride booking and tracking.

**Key Entities:**

* **User** (userId, name, email, phone, role, passwordHash)
* **Rider** (userId FK, walletBalance, rideHistory FK)
* **Driver** (userId FK, licenseNo, vehicleId FK, driverStatus)
* **Vehicle** (vehicleId, driverId FK, registrationNo, type, capacity)
* **Ride** (rideId, riderId FK, driverId FK, pickup, dropoff, otp, fare, status)
* **Payment** (paymentId, rideId FK, method, amount, status, timestamp)
* **CommissionRecord** (commissionId, rideId FK, amount, percentage)
* **Location** (locationId, rideId FK, latitude, longitude, timestamp)

## **4.4.0 INTEGRATION WITH GOOGLE MAPS API AND RAZORPAY TEST MODE**

* **Google Maps API:**
  + Used for live GPS tracking, route optimization, and ETA calculation.
  + APIs integrated: Directions API, Distance Matrix API, and Geocoding API.
  + Continuous location updates are pushed from Driver App to Rider App via WebSocket.
* **Razorpay (Test Mode):**
  + Used for simulating online payments during development.
  + Supports credit/debit cards, UPI, and wallets.
  + Webhooks are configured to confirm payment status.
  + Refunds and cancellations are tested through sandbox API calls.

This integration ensures **real-world testing** while keeping the project development-friendly and cost-free.

## **4.5.0 NOTIFICATION HANDLING (WEBSOCKET, EMAIL, SMS)**

Urban X provides **multi-channel notifications** for real-time communication:

1. **WebSocket:**
   * Used for low-latency updates such as “Driver Arriving”, “Ride Started”, and live location updates.
   * Implemented using **Spring WebSocket** on the backend and **RxJS Observables** on Angular frontend.
2. **Email Notifications:**
   * Sent for account registration, ride invoices, and account alerts.
   * Implemented using **Spring Boot Mail** service with SMTP integration.
3. **SMS Notifications:**
   * Sent for OTP verification, ride alerts, and SOS messages.
   * Integrated via third-party SMS gateways (e.g., Twilio).

This hybrid approach ensures **maximum reachability** and enhances user trust by providing updates even if one communication channel fails.