Rydex Ride-Sharing Application - Project Documentation

Version: 1.0

Authors: Rydex Development Team

Date: 11-12-2025

Team:

Jawad Ahmad$_{\text{FA24-BCS-093}}$

Bilawal Sohail$_{\text{FA23-BCS-012}}$

Table of Contents

Chapter 1 — Introduction

1.1 Introduction

Rydex is a university-level ride-sharing application implemented in Java with a MySQL backend. It aims to connect riders and drivers with a simple booking workflow, supporting payments, feedback, vehicle and shifts management, and two UI modes: a Swing-based GUI and a CLI version.

This documentation provides comprehensive technical and design-level details that will help developers, maintainers, testers, and stakeholders understand, extend, and deploy the system.

1.2 Project Overview

Key technologies:

Java (backend),

MySQL (database), JDBC

Swing (desktop UI),

DAO pattern,

Core features:

- User management (Driver, Rider)

- Driver registration and vehicle management

- Ride creation, search and booking

- Driver shifts scheduling

- Payments and transaction recording

- Feedback and ratings

- CLI and Swing frontends

System boundaries: Desktop application connecting to a single MySQL instance. No web or mobile clients in the current scope.

1.3 Problem Statement

Urban and campus transportation can be inefficient, costly, and fragmented. Rydex aims to provide an organized, low-cost ride-sharing platform suitable for small communities or campus environments where quick setup and straightforward operation are required.

Specific problems addressed:

- Lack of a coordinated system to match riders to drivers.

- Manual handling of bookings and payments leads to errors and overhead.

- No centralized record of vehicle availability, driver shifts, and rider feedback.

1.4 Scope and Objectives

Primary objectives:

- Build a reliable ride booking system with clear separation of concerns (DAO pattern).

- Provide both GUI and CLI interfaces for flexibility.

- Persist all critical information in a normalized MySQL schema.

Out of scope (for current version):

- Real-time GPS tracking and map integration.

- Push notifications and SMS.

- Scalable multi-instance server deployment and microservices.

1.5 Stakeholders

- Students and campus riders (end users)
- Drivers (service providers)
- Admin / Maintainers (DBA, developers)
- QA/Testers

Chapter 2 — System Modelling

2.1 Enhanced Entity-Relationship (EER)

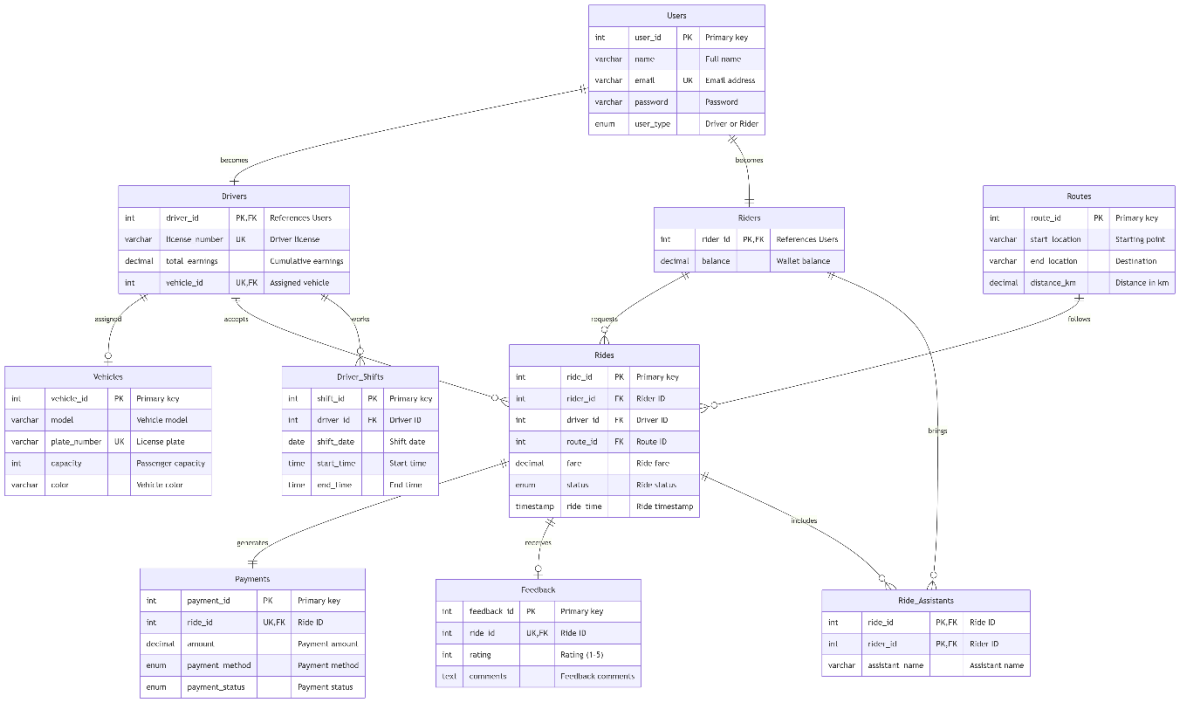


Figure 1EER Diagram

Relationship Details:

1. Users → Drivers / Riders (1:1)

- **Cardinality**: One-to-One (Exclusive)
- **Description**: Each user is either a Driver OR a Rider, not both
- **Foreign Key**: Drivers.driver_id and Riders.rider_id reference Users.user_id

2. Drivers → Vehicles (1:0..1)

- **Cardinality**: One-to-Zero-or-One
- **Description**: A driver may have one assigned vehicle, or none initially
- **Foreign Key**: Drivers.vehicle_id → Vehicles.vehicle_id
- **Constraint**: UNIQUE ensures one vehicle per driver

3. Riders → Rides (1:N)

- **Cardinality**: One-to-Many
- **Description**: One rider can request multiple rides
- **Foreign Key**: Rides.rider_id → Riders.rider_id

4. Drivers → Rides (1:N)

- **Cardinality**: One-to-Many
- **Description**: One driver can accept multiple rides
- **Foreign Key**: Rides.driver_id → Drivers.driver_id

5. Routes → Rides (1:N)

- **Cardinality**: One-to-Many
- **Description**: One route (start/end) can be used for multiple rides
- **Foreign Key**: Rides.route_id → Routes.route_id

6. Rides → Payments (1:1)

- **Cardinality**: One-to-One
- **Description**: Each ride generates exactly one payment record
- **Foreign Key**: Payments.ride_id → Rides.ride_id
- **Constraint**: UNIQUE ensures one payment per ride

7. Rides → Feedback (1:0..1)

- **Cardinality**: One-to-Zero-or-One

- **Description**: A ride may receive one feedback, or none if not rated
- **Foreign Key**: Feedback.ride_id → Rides.ride_id

8. Drivers → Driver_Shifts (1:N)

- **Cardinality**: One-to-Many
- **Description**: A driver can work multiple shifts
- **Foreign Key**: Driver_Shifts.driver_id → Drivers.driver_id

9. Rides → Ride_Assistants (1:N)

- **Cardinality**: One-to-Many
- **Description**: One ride can include multiple assistants
- **Foreign Key**: Ride_Assistants.ride_id → Rides.ride_id
- **Composite Key**: Primary key includes both ride_id and rider_id

10. Riders → Ride_Assistants (1:N)

- **Cardinality**: One-to-Many
- **Description**: A rider can bring multiple assistants on different rides
- **Foreign Key**: Ride_Assistants.rider_id → Riders.rider_id

Business Rules Implied:

1. **User Type Enforcement**: A user cannot be both Driver and Rider
2. **Vehicle Assignment**: One vehicle per driver maximum
3. **Payment Tracking**: Every ride must have a payment record
4. **Feedback System**: Optional ratings (1-5 stars) with comments
5. **Shift Management**: Drivers can have scheduled working hours
6. **Assistant Tracking**: Support for additional passengers
7. **Ride Status Flow**: Pending → Confirmed → In Progress → Completed/Cancelled
8. **Payment Flow**: Pending → Completed/Failed/Refunded

2.2 Class Diagrams (UML)

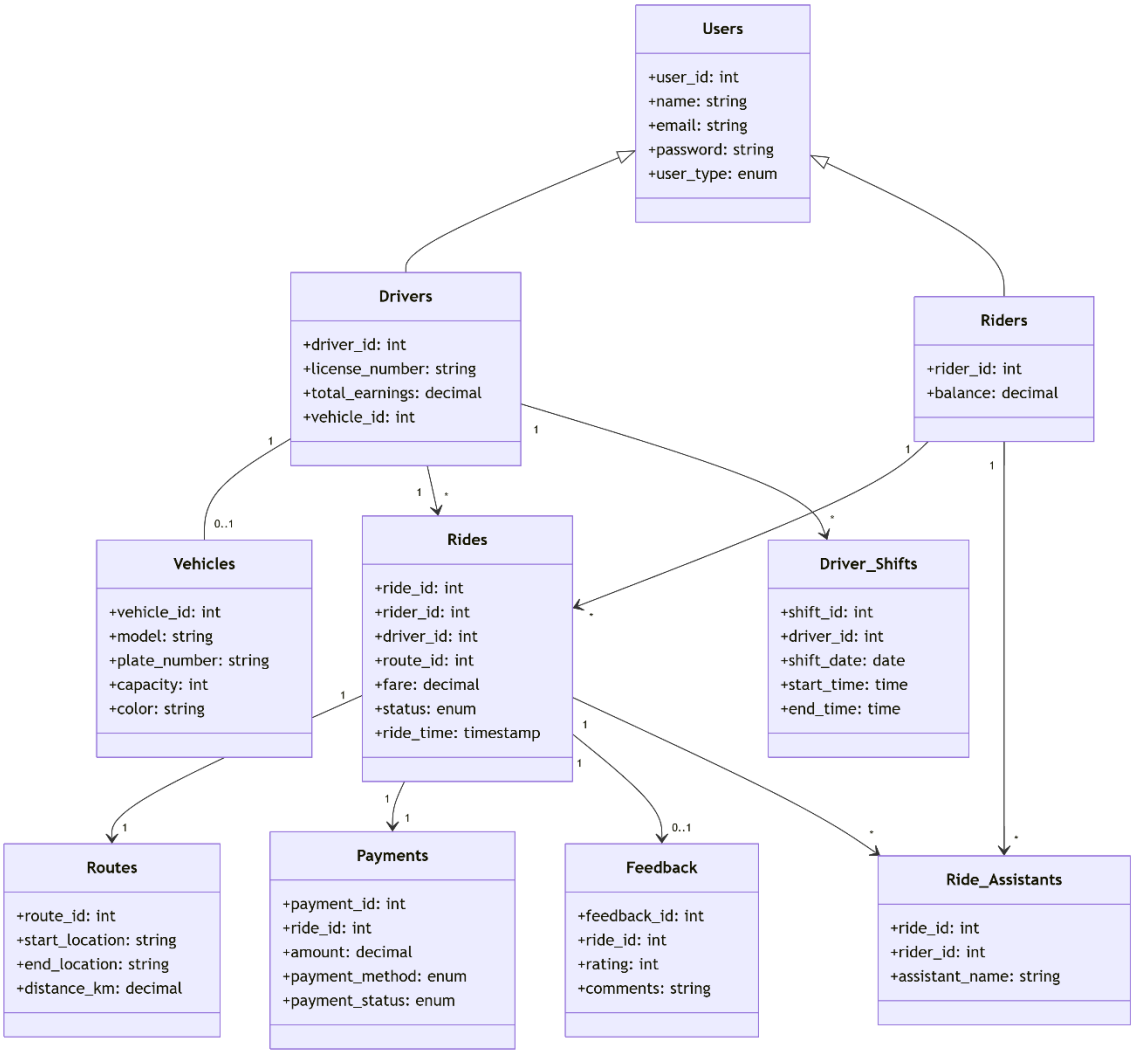


Figure 2 UML Diagram

2.3 Data Dictionary

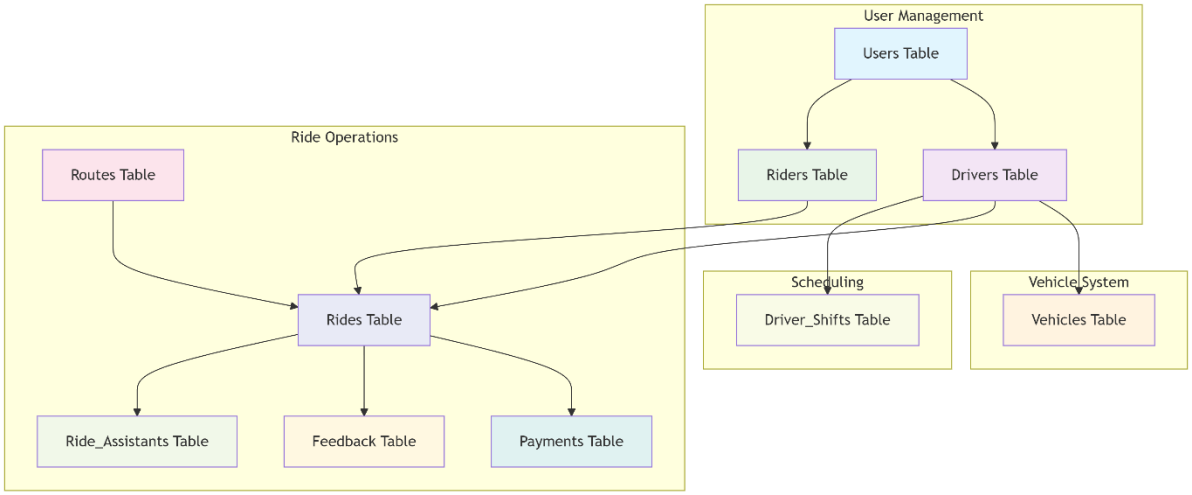


Figure 3 Class Flow

Chapter 3 — Methodology

3.1 Development Methodology

The project follows a lightweight Agile approach with iterative sprints. Each sprint focuses on delivering one vertical slice (e.g., user auth, booking flow, payment integration). Use version control (Git) with feature branches and PR reviews.

3.2 Architecture & Design Decisions

Design patterns used:
- DAO (Data Access Object) pattern for database separation
- MVC-like separation between UI (Swing/CLI), business logic (services/DAOs), and persistence (Database Manager)

Why these choices:
- DAO allows testing and swapping DB implementations.
- Swing keeps the UI desktop-native; CLI enables headless testing and debugging.

3.3 Detailed Module Descriptions

3.3.1 Authentication & User Management

- Login flow: LoginScreen calls UserDAO.authenticate(username, password).
- Passwords: currently stored as plaintext

3.3.2 Driver & Vehicle Management

- CRUD operations for drivers and vehicles via DriverDAO and VehicleDAO.
- Drivers have shifts recorded in Driver_Shifts.

3.3.3 Ride Matching & Booking

- Basic matching: riders search for available drivers by route/time.
- Booking creates a Ride record with status = REQUESTED.
- Driver accepts -> status = ACCEPTED -> when started IN_PROGRESS -> upon completion COMPLETED and payment flow triggers.

3.3.4 Payments

- Payments recorded in Payments table. Offline payment methods should still create a transaction record.

3.3.5 Feedback

- Riders and drivers can leave ratings and comments linked to rides.

3.4 Database Design & Normalization

All tables are normalized to at least 3NF. Foreign keys enforce referential integrity. Use indexes on frequently searched columns (e.g., username, driver_id, ride status) to improve read performance.

Sample normalization rationale:

- Separation of Users from Drivers/Riders avoids data duplication and allows role switching.

- Vehicles separate because drivers may have multiple vehicles.

3.5 Algorithms & Sequence Flows

Ride booking sequence (high level):

1. Rider searches for route/time.
2. System queries available drivers (matching route/shift).
3. Rider selects a driver and creates booking; system inserts Ride with REQUESTED.
4. Driver receives request (in current offline model, polls or checks GUI) and accepts -> ACCEPTED.
5. On ride completion, system updates Ride status and creates Payment and Feedback entries.

3.6 Testing Strategy

- **Unit tests:**

- Test DAOs with an in-memory or dedicated test database. Use transactions + rollbacks for isolation.
- Model tests for validation logic (e.g., fare calculation).

- **Integration tests:**

- Full flow tests: registration -> booking -> acceptance -> payment -> feedback.

- **Manual/GUI tests:**

- Run Swing UI tests using Fest or AssertJ-Swing for regression.

- **Test data:**

- Provide SQL fixtures for test users, drivers, vehicles, and routes.

3.7 Deployment & Maintenance

Local deployment steps:

1. Install Java (11+ recommended) and JDBC.

2. Install MySQL and run the provided schema.sql to create tables.

3. Update DatabaseManager with DB credentials.

4. Build and run

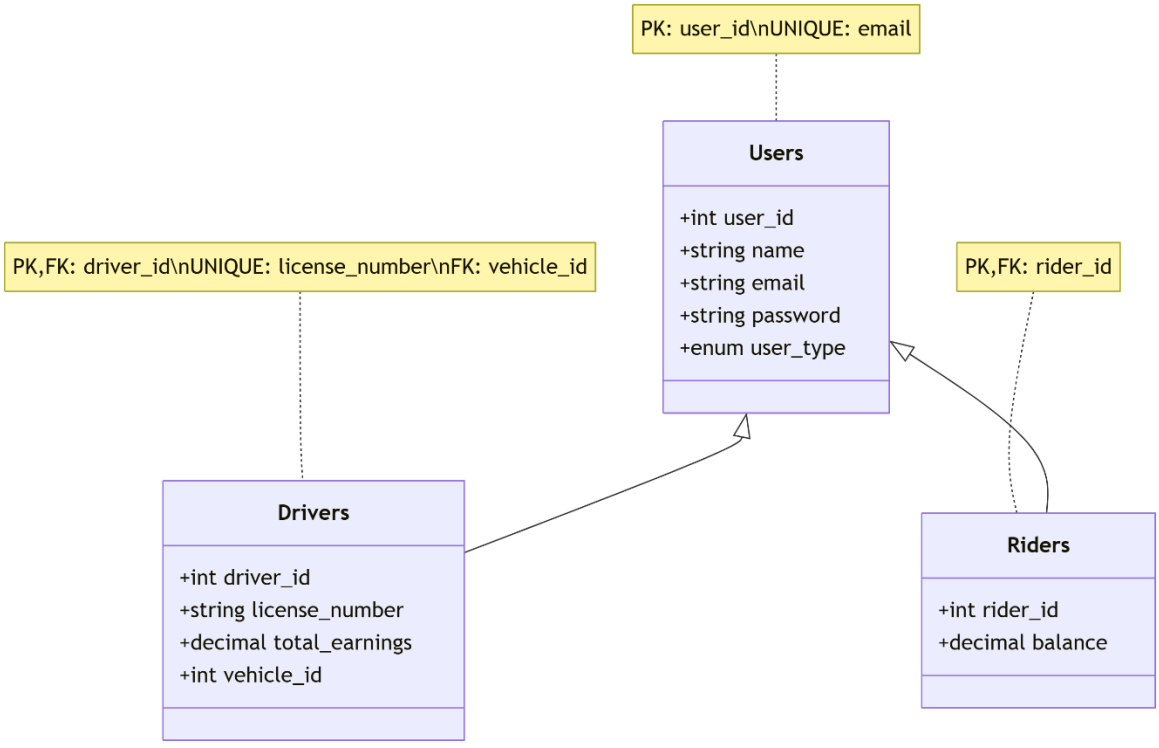
Backup and maintenance:

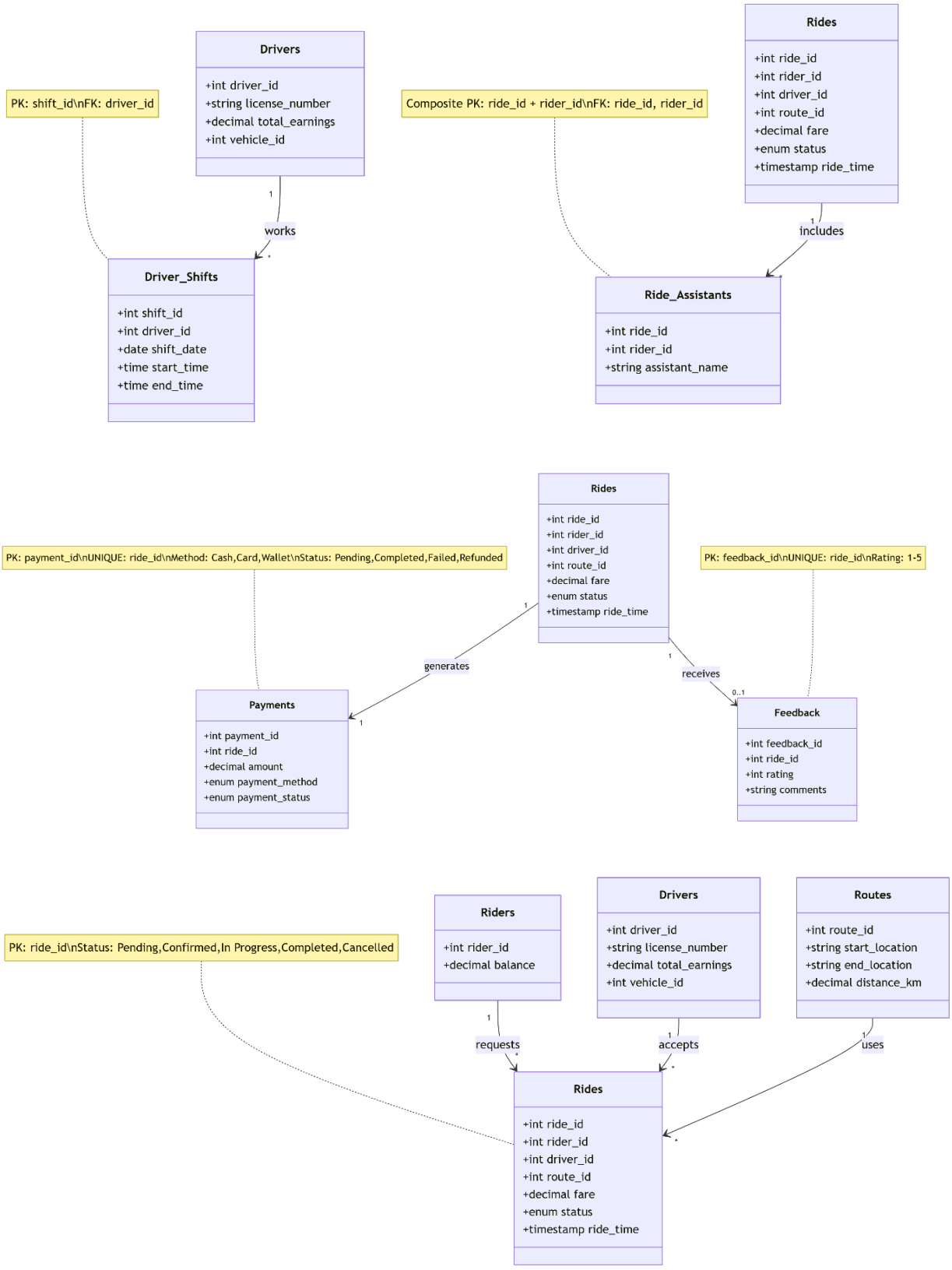
- Regular DB backups

- Monitor table sizes for Payments and Feedback and archive older data if needed.

Chapter 4 — Figures and Descriptions

4.1 Figures




Drivers
+int driver_id
+string license_number
+decimal total_earnings
+int vehicle_id
PK: shift_id\nFK: driver_id
works
Driver_Shifts
+int shift_id
+int driver_id
+date shift_date
+time start_time
+time end_time
Rides
+int ride_id
+int rider_id
+int driver_id
+int route_id
+decimal fare
+enum status
+timestamp ride_time
Composite PK: ride_id + rider_id\nFK: ride_id, rider_id
includes
Ride_Assistants
+int ride_id
+int rider_id
+string assistant_name
Rides
+int ride_id
+int rider_id
+int driver_id
+int route_id
+decimal fare
+enum status
+timestamp ride_time
PK: payment_id\nUNIQUE: ride_id\nMethod: Cash,Card,Wallet\nStatus: Pending,Completed,Failed,Refunded
generates
receives
PK: feedback_id\nUNIQUE: ride_id\nRating: 1-5
Payments
+int payment_id
+int ride_id
+decimal amount
+enum payment_method
+enum payment_status
Feedback
+int feedback_id
+int ride_id
+int rating
+string comments
Riders
+int rider_id
+decimal balance
Drivers
+int driver_id
+string license_number
+decimal total_earnings
+int vehicle_id
Routes
+int route_id
+string start_location
+string end_location
+decimal distance_km
PK: ride_id\nStatus: Pending,Confirmed,In Progress,Completed,Cancelled
requests
accepts
uses
Rides
+int ride_id
+int rider_id
+int driver_id
+int route_id
+decimal fare
+enum status
+timestamp ride_time

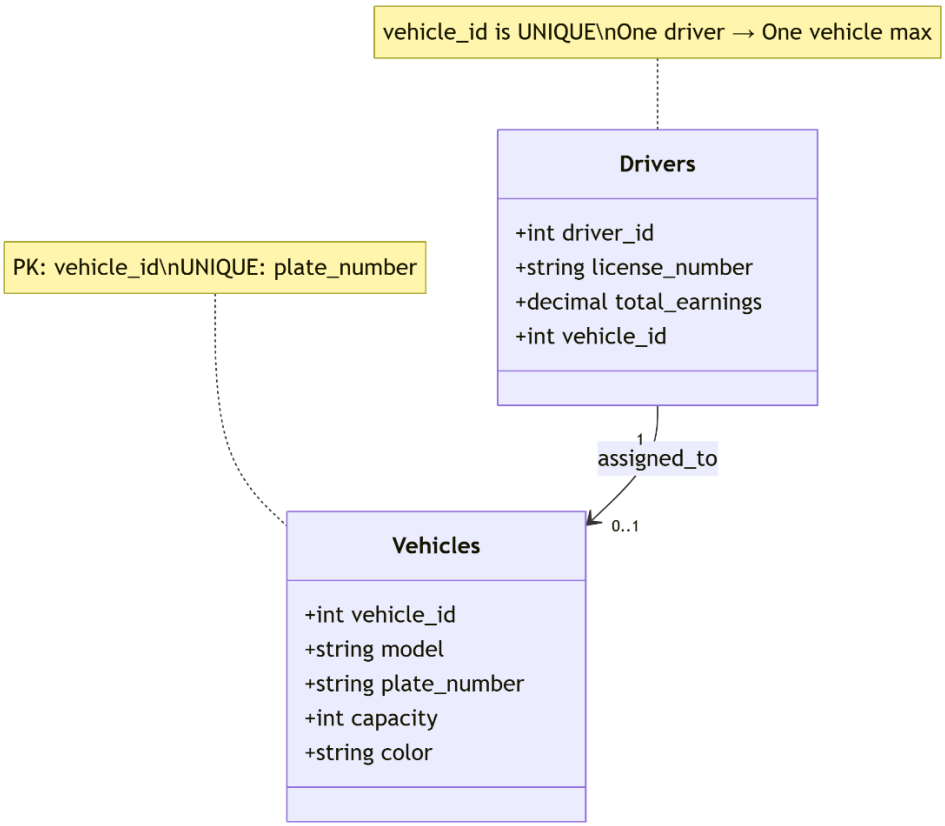


Figure 4 Column level Diagram

4.2 Figure Descriptions

Figure 1 – UML Diagram - Visualizes POJOs and DAO classes with attributes. Helpful for onboarding new developers.

Figure 2 – Class Flow Diagram - Shows the flow of interaction between various classes, how they are linked and how they share the data among themselves

Figure 3 - EER Diagram - Shows entities (Users, Drivers, Riders, Vehicles, Routes, Rides, Payments, Feedback, Driver_Shifts) and their relationships. Primary/foreign keys and cardinalities are annotated. Use this to review normalization and ensure FK constraints match DAO operations.

Figure 4 - Column level Diagram - Visualizes primary and foreign keys and other important constraints applied on columns. Helpful for onboarding new developers.

Appendices

A. SQL Schema Excerpt

CREATE TABLE **Users** (

user_id INT AUTO_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(100) NOT NULL UNIQUE,

password VARCHAR(100) NOT NULL,

user_type ENUM('Driver', 'Rider') NOT NULL);

CREATE TABLE **Vehicles** (

vehicle_id INT AUTO_INCREMENT PRIMARY KEY,

model VARCHAR(100) NOT NULL,

plate_number VARCHAR(50) UNIQUE NOT NULL,

capacity INT NOT NULL,

color VARCHAR(50));

CREATE TABLE **Drivers** (

driver_id INT PRIMARY KEY,

license_number VARCHAR(50) UNIQUE NOT NULL,

total_earnings DECIMAL(10,2) DEFAULT 0,

vehicle_id INT UNIQUE,

FOREIGN KEY (driver_id) REFERENCES Users(user_id),

FOREIGN KEY (vehicle_id) REFERENCES Vehicles(vehicle_id) ON DELETE SET NULL);

CREATE TABLE **Riders** (

rider_id INT PRIMARY KEY,

balance DECIMAL(10,2) DEFAULT 0,

FOREIGN KEY (rider_id) REFERENCES Users(user_id));

CREATE TABLE **Routes** (

route_id INT AUTO_INCREMENT PRIMARY KEY,

start_location VARCHAR(150) NOT NULL,

end_location VARCHAR(150) NOT NULL,

distance_km DECIMAL(10,2) NOT NULL);

CREATE TABLE **Rides** (

ride_id INT AUTO_INCREMENT PRIMARY KEY,

rider_id INT NOT NULL,

driver_id INT NOT NULL,

route_id INT NOT NULL,

fare DECIMAL(10,2) NOT NULL,

status ENUM('Pending', 'Confirmed', 'In Progress', 'Completed', 'Cancelled') NOT NULL DEFAULT 'Pending',

ride_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

```
FOREIGN KEY (rider_id) REFERENCES Riders(rider_id),
FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id),
FOREIGN KEY (route_id) REFERENCES Routes(route_id));
CREATE TABLE Payments (
payment_id INT AUTO_INCREMENT PRIMARY KEY,
ride_id INT UNIQUE NOT NULL,
amount DECIMAL(10,2) NOT NULL,
payment_method ENUM('Cash', 'Card', 'Wallet') NOT NULL,
payment_status ENUM('Pending', 'Completed', 'Failed', 'Refunded') NOT NULL DEFAULT 'Pending',
FOREIGN KEY (ride_id) REFERENCES Rides(ride_id));
CREATE TABLE Feedback (
feedback_id INT AUTO_INCREMENT PRIMARY KEY,
ride_id INT UNIQUE NOT NULL,
rating INT NOT NULL CHECK (rating BETWEEN 1 AND 5),
comments TEXT,
FOREIGN KEY (ride_id) REFERENCES Rides(ride_id));
CREATE TABLE Driver_Shifts (
shift_id INT AUTO_INCREMENT PRIMARY KEY,
driver_id INT NOT NULL,
shift_date DATE NOT NULL,
start_time TIME NOT NULL,
end_time TIME NOT NULL,
FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id));
CREATE TABLE Ride_Assistants (
ride_id INT NOT NULL,
rider_id INT NOT NULL,,
assistant_name VARCHAR(100) NOT NULL,
PRIMARY KEY (ride_id, rider_id),
FOREIGN KEY (ride_id) REFERENCES Rides(ride_id),
FOREIGN KEY (rider_id) REFERENCES Riders(rider_id));
```

B. Sample CLI Commands

- javac -cp ".;libs/mysql-connector-j-9.5.0.jar" management*.java
- java -cp ".;libs/mysql-connector-j-9.5.0.jar;management" RideSharingApp