# **LLD for a "Ola/ Uber" system**

### Basic Features

1. **User Management**: Register, login, and manage user profiles.
2. **Ride Booking**: Book a ride, track ride status, and manage ride history.
3. **Driver Management**: Register and manage driver profiles.
4. **Payment**: Handle payment for rides.
5. **Notifications**: Send notifications to users and drivers.

### Core Classes and Interactions

#### 1. **User**

public class User {

private String userId;

private String name;

private String email;

private String phoneNumber;

private List<Ride> rideHistory;

public User(String userId, String name, String email, String phoneNumber) {

this.userId = userId;

this.name = name;

this.email = email;

this.phoneNumber = phoneNumber;

this.rideHistory = new ArrayList<>();

}

// Getters and setters

// Other methods like register, login, etc.

}

#### 2. **Driver**

public class Driver {

private String driverId;

private String name;

private String licenseNumber;

private String carDetails;

private boolean isAvailable;

public Driver(String driverId, String name, String licenseNumber, String carDetails) {

this.driverId = driverId;

this.name = name;

this.licenseNumber = licenseNumber;

this.carDetails = carDetails;

this.isAvailable = true;

}

// Getters and setters

// Other methods like updateAvailability, etc.

}

#### 3. **Ride**

public class Ride {

private String rideId;

private User user;

private Driver driver;

private String source;

private String destination;

private RideStatus status;

private double fare;

public Ride(String rideId, User user, Driver driver, String source, String destination) {

this.rideId = rideId;

this.user = user;

this.driver = driver;

this.source = source;

this.destination = destination;

this.status = RideStatus.REQUESTED;

}

// Getters and setters

// Other methods like startRide, completeRide, calculateFare, etc.

}

#### 4. **RideStatus (Enum)**

public enum RideStatus {

REQUESTED,

IN\_PROGRESS,

COMPLETED,

CANCELLED;

}

#### 5. **Payment**

public class Payment {

private String paymentId;

private Ride ride;

private double amount;

private PaymentStatus status;

public Payment(String paymentId, Ride ride, double amount) {

this.paymentId = paymentId;

this.ride = ride;

this.amount = amount;

this.status = PaymentStatus.PENDING;

}

// Getters and setters

// Other methods like processPayment, etc.

}

#### 6. **PaymentStatus (Enum)**

public enum PaymentStatus {

PENDING,

COMPLETED,

FAILED;

}

#### 7. **Notification**

public class Notification {

private String notificationId;

private String message;

private User user;

private Driver driver;

public Notification(String notificationId, String message, User user, Driver driver) {

this.notificationId = notificationId;

this.message = message;

this.user = user;

this.driver = driver;

}

// Getters and setters

// Other methods like sendNotification, etc.

}

### Design Patterns

#### 1. **Singleton Pattern**

Used for classes that manage connections to external services like databases, payment gateways, etc.

public class DatabaseConnection {

private static DatabaseConnection instance;

private DatabaseConnection() {

// Initialize connection

}

public static DatabaseConnection getInstance() {

if (instance == null) {

instance = new DatabaseConnection();

}

return instance;

}

// Other methods for database operations

}

#### 2. **Observer Pattern**

Used for the notification system to notify users and drivers about ride status changes.

#### public interface Observer {

#### void update(String message);

#### }

#### public class RideStatusNotifier {

#### private List<Observer> observers;

#### public RideStatusNotifier() {

#### observers = new ArrayList<>();

#### }

#### public void addObserver(Observer observer) {

#### observers.add(observer);

#### }

#### public void notifyObservers(String message) {

#### for (Observer observer : observers) {

#### observer.update(message);

#### }

#### }

#### }

#### public class User implements Observer {

#### // Other fields and methods

#### @Override

#### public void update(String message) {

#### System.out.println("User received notification: " + message);

#### }

#### }

#### public class Driver implements Observer {

#### // Other fields and methods

#### @Override

#### public void update(String message) {

#### System.out.println("Driver received notification: " + message);

#### }

#### }

### Interactions

1. **User Registration/Login**: User registers or logs in, and the system creates or retrieves a User object.
2. **Ride Booking**: User requests a ride, and the system creates a Ride object and assigns an available Driver.
3. **Ride Management**: Driver starts/completes the ride, updating the Ride status, and notifying User and Driver.
4. **Payment Processing**: Once the ride is completed, a Payment object is created and processed.
5. **Notifications**: Notifications are sent to User and Driver about ride status changes.