System Design Document

RideEase

Group 11

High-Level Class Description using CRC Cards

Class: RideRequest

- **Parent Class (if any):** None
- **Subclasses (if any):** ScheduledRideRequest, InstantRideRequest
- **Responsibilities:**
  - Create ride requests with location, time, and passenger details.
  - Validate ride request parameters.
  - Track request status (pending, accepted, completed).
- **Collaborators:**
  - RideMatcher
  - UserProfile

Class: UserProfile

- **Parent Class (if any):** None
- **Subclasses (if any):** DriverProfile, PassengerProfile
- **Responsibilities:**
  - Store user information (e.g., name, contact, payment details).
  - Manage user preferences.
  - Track ride history.
- **Collaborators:**
  - RideRequest
  - PaymentProcessor

Class: RideMatcher

- **Parent Class (if any):** None
- **Subclasses (if any):** None
- **Responsibilities:**
  - Match passengers with drivers based on location and preferences.
  - Optimize matches using algorithms (e.g., shortest route, lowest cost).

- Handle re-matching in case of driver cancellation.
- **Collaborators:**
    - RideRequest
    - UserProfile

Class: PaymentProcessor

- **Parent Class (if any):** None
- **Subclasses (if any):** None
- **Responsibilities:**
    - Handle fare calculations.
    - Process payments via credit card or digital wallets.
    - Manage refunds and dispute resolutions.
- **Collaborators:**
    - UserProfile
    - RideRequest

Class: NotificationSystem

- **Parent Class (if any):** None
- **Subclasses (if any):** None
- **Responsibilities:**
    - Send notifications to users about ride status.
    - Notify drivers of incoming ride requests.
    - Handle alerts for cancellations or system issues.
- **Collaborators:**
    - RideRequest
    - UserProfile

System Interaction with the Environment

The RideEase application relies on the following dependencies and assumptions for operation:

- **Operating System:** The system will primarily operate on Android and iOS mobile platforms. It also requires backend servers running Linux.
- **Programming Languages and Frameworks:**
    - Frontend: React Native for cross-platform compatibility.
    - Backend: Python (Django/Flask) or Node.js.
- **Database:** A relational database such as PostgreSQL to store user profiles, ride requests, and transaction details.
- **Network Configuration:** The system requires stable internet connectivity to communicate with cloud services and APIs.
- **Third-party APIs:**
    - Google Maps API for location and routing services.
    - Firebase API for user authentication.
- **Error Handling Assumptions:**
    - Valid user input is expected; invalid input will prompt error messages.
    - System will retry failed network requests up to 3 times before notifying the user.

System Architecture

The system uses a three-tier architecture:

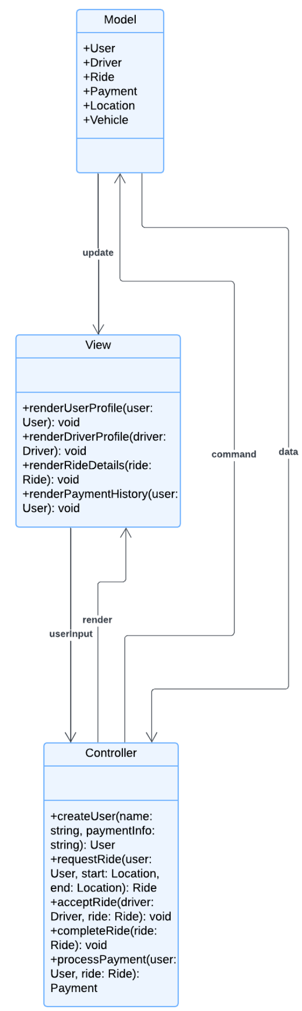
1. **Presentation Layer:**
    - Composed of the mobile application interface.
    - Includes user registration, ride request creation, and payment options.
2. **Business Logic Layer:**
    - Contains core functionalities such as ride matching, payment processing, and notification management.
    - Handles algorithms for driver-passenger pairing and route optimization.
3. **Data Layer:**
    - Manages the database with tables for user profiles, ride requests, and transaction logs.
    - Ensures data consistency and secure storage.

Architectural Diagram:

- [Frontend] --[HTTPS]--> [Backend API Server] --[SQL Queries]--> [Database]
- Backend also integrates with third-party services like Google Maps and Firebase.

Error and Exception Handling Strategy

Anticipated Errors:



1. **Invalid User Input:**
    - Strategy: Input validation on the frontend and backend to prevent invalid requests.
    - Response: Display user-friendly error messages and suggestions for corrections.
2. **Network Failures:**
    - Strategy: Implement retries with exponential backoff for API calls.
    - Response: Notify users of connectivity issues and save actions locally for retry.
3. **External System Failures (e.g., API downtime):**
    - Strategy: Monitor API health using heartbeat checks.
    - Response: Provide fallback mechanisms or alternative options (e.g., cached map data).
4. **Driver/Passenger Cancellations:**
    - Strategy: Update ride status in real-time and initiate re-matching procedures.
    - Response: Notify affected parties and adjust fare calculations if applicable.
5. **Payment Processing Issues:**
    - Strategy: Validate payment methods and transaction details before processing.
    - Response: Retry failed transactions and notify users of success or failure.

Backend

RideEase Backend Architecture

The backend of RideEase, a ridesharing application, is built using **Python** and the **Flask** framework. This architecture is designed to ensure scalability, reliability, and ease of maintenance, with a focus on facilitating seamless communication between passengers and drivers.

Core Components

1. **Flask Framework**
    - Flask serves as the primary web framework, providing a lightweight and flexible foundation for building RESTful APIs. Its modular nature allows for easy integration with various components and libraries, making it ideal for this application.

2. **Database**
    - An **SQL database** is used to store and manage data. This relational database ensures efficient querying and data consistency across the application.
    - The database schema includes tables for:
        - **Users**: Stores user information such as name, email, phone number, and role (driver or passenger).
        - **Rides**: Details of ride requests and offers, including origin, destination, time, fare, and status.
        - **Vehicles**: Information about the drivers' vehicles.
        - **Transactions**: Logs payment details and ride history.

3. **Database Setup File (database_setup.py)**
    - The database_setup.py file defines the database schema and initializes the database structure.
    - Key functionalities include:
        - Creating tables and defining their relationships using an ORM (e.g., SQLAlchemy).
        - Setting up primary and foreign keys to ensure referential integrity.
        - Providing utilities for database connection and migration.

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base
from dotenv import load_dotenv
import os

# Load environment variables from the .env file
load_dotenv()

class Database:
    """Class to encapsulate the database setup and session management."""

    # Define the base class for your SQLAlchemy models
    Base = declarative_base()

    def __init__(self):
        # Fetch the database URL from the environment variables
        self.database_url = os.getenv("DATABASE_URL")

        # Ensure the URL is provided
        if not self.database_url:
            raise ValueError("DATABASE_URL is not set in the environment variables")

        # Initialize the database engine and session
        self.engine = create_engine(self.database_url, echo=True, future=True)
        self.SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=self.engine)

    def initialize_db(self):
        """Initializes the database connection and creates tables if they don't exist."""
        try:
            # Create all tables in the database
            self.Base.metadata.create_all(bind=self.engine)
            print("Database connection successful and tables created!")
        except Exception as e:
            print("Failed to initialize the database!")
            print(e)

# Initialize the database object
db = Database()

if __name__ == "__main__":
    db.initialize_db()
```