# **Car Rental System**

## Features

### User Management

1. **User Registration and Login**
   * Register a new user with necessary details.
   * Login with credentials.
   * Logout functionality.
   * Password reset and change password features.
2. **User Profile Management**
   * View and edit profile information.
   * View rental history.
   * View and manage current reservations.

### Car Inventory Management

1. **Car Information**
   * Add, remove, and update car details (model, brand, type, license plate, etc.).
   * Update car availability status.
   * View detailed information about a specific car.
2. **Car Categorization**
   * Categorize cars based on type (SUV, Sedan, Hatchback, etc.).
   * Categorize cars based on price range.

### Reservation Management

1. **Car Search and Filter**
   * Search for available cars based on location, type, dates, etc.
   * Apply filters like car type, price range, brand, etc.
2. **Reservation Handling**
   * Make a new reservation.
   * Check car availability for desired dates.
   * Confirm or cancel reservations.
   * Modify existing reservations (change dates, car type, etc.).

### Rental Management

1. **Rental Process**
   * Initiate rental at the start time.
   * Record rental start and end times.
   * Calculate rental cost based on duration and car type.
2. **Rental History**
   * View past rentals.
   * Download rental receipts.
3. **Payment Handling**
   * Process payments (multiple payment options like credit card, PayPal, etc.).
   * Handle payment statuses (pending, completed, failed).
   * Generate invoices and receipts.

### Notification System

1. **Notifications**
   * Send notifications for reservation confirmations, cancellations, rental start/end, payment receipts, etc.
   * Support different notification types (email, SMS, push notifications).
2. **Notification Management**
   * View past notifications.
   * Enable or disable notifications.

### Admin Features

1. \*\*Inventory Management

### Admin Features

1. **Inventory Management**
   * Add new cars to the inventory.
   * Remove cars from the inventory.
   * Update car details (status, availability, maintenance schedules).
2. **User Management**
   * View and manage user profiles.
   * Deactivate or suspend user accounts if necessary.
3. **Reporting and Analytics**
   * Generate reports on car usage, rentals, reservations, and payments.
   * Analyze trends in car rentals (popular models, peak times, etc.).
4. **Promotions and Discounts**
   * Create and manage promotional offers and discounts.
   * Apply discounts to specific cars or rental periods.

### Additional Features

1. **Location Management**
   * Add and manage car rental locations.
   * Track car availability by location.
   * Transfer cars between locations.
2. **Feedback and Reviews**
   * Allow users to rate and review cars and rental experiences.
   * View and manage user feedback.
3. **Customer Support**
   * Integrate a helpdesk or support system.
   * Allow users to raise support tickets or contact customer service.
4. **Insurance and Add-ons**
   * Offer insurance options for rentals.
   * Provide additional services like GPS, child seats, etc.

## Class Responsibilities and Methods

Here is an extended look at the responsibilities and methods of key classes in the system.

#### User Management Classes

* **User**

class User {

String userID;

String name;

String email;

String phoneNumber;

List<Reservation> reservations;

List<Rental> rentals;

void register();

void login();

void logout();

void updateProfile(String newName, String newEmail, String newPhoneNumber);

void resetPassword(String newPassword);

}

Admin

class Admin extends User {

void addCar(Car car);

void removeCar(Car car);

void updateCarDetails(Car car);

void generateReport();

void managePromotions(Promotion promotion);

}

Authentication

class Authentication {

boolean authenticate(String userID, String password);

void resetPassword(String userID);

}

#### Car Inventory Management Classes

* **Car**

class Car {

String carID;

String licensePlate;

CarType type;

String model;

String brand;

boolean isAvailable;

void updateAvailability(boolean availability);

void updateDetails(String newModel, String newBrand, CarType newType);

}

CarType

class CarType {

String typeID;

String name; // e.g., SUV, Sedan, Hatchback

double pricePerDay;

}

CarInventory

class CarInventory {

Map<CarType, List<Car>> cars;

List<Car> getAvailableCars(CarType type);

void addCar(Car car);

void removeCar(Car car);

void updateCar(Car car);

}

#### Reservation Management Classes

* **Reservation**

class Reservation {

String reservationID;

User user;

Car car;

LocalDateTime reservationStart;

LocalDateTime reservationEnd;

ReservationStatus status;

boolean checkAvailability();

void confirmReservation();

void cancelReservation();

void modifyReservation(LocalDateTime newStart, LocalDateTime newEnd, Car newCar);

}

ReservationRecord

class ReservationRecord {

List<Reservation> reservations;

void addReservation(Reservation reservation);

List<Reservation> getUserReservations(String userID);

}

ReservationStatus (Enum)

enum ReservationStatus {

PENDING,

CONFIRMED,

CANCELLED

}

#### Rental Management Classes

* **Rental**

class Rental {

String rentalID;

User user;

Car car;

LocalDateTime rentalStart;

LocalDateTime rentalEnd;

double totalCost;

RentalStatus status;

void calculateTotalCost();

void startRental();

void endRental();

}

RentalRecord

class RentalRecord {

List<Rental> rentals;

void addRental(Rental rental);

List<Rental> getUserRentals(String userID);

}

Payment

class Payment {

String paymentID;

Rental rental;

double amount;

PaymentStatus status;

void processPayment();

}

PaymentStatus (Enum)

enum PaymentStatus {

PENDING,

COMPLETED,

FAILED

}

#### 4. Reservation Management

* **Reservation**

class Reservation {

String reservationID;

User user;

Car car;

LocalDateTime reservationStart;

LocalDateTime reservationEnd;

boolean checkAvailability();

void confirmReservation();

void cancelReservation();

}

ReservationRecord

class ReservationRecord {

List<Reservation> reservations;

void addReservation(Reservation reservation);

List<Reservation> getUserReservations(String userID);

}

#### 5. Notification System

* **Notification**

class Notification {

String notificationID;

User user;

String message;

NotificationType type;

void sendNotification();

}

**NotificationType (Enum)**

enum NotificationType {

EMAIL,

SMS

}

NotificationService

class NotificationService {

void sendNotification(Notification notification);

}

### Interactions

1. **User makes a reservation:**
   * User searches for available cars.
   * User selects a car and makes a reservation.
   * System checks car availability.
   * If available, reservation is confirmed and a notification is sent to the user.
2. **User rents a car:**
   * User confirms the rental at the start time.
   * System updates car availability.
   * Rental is recorded, and payment is processed.
   * A notification is sent confirming the rental.
3. **Admin manages inventory:**
   * Admin adds/removes cars from the inventory.
   * Admin updates car details and availability.

### Sequence Diagram

Here's a simplified sequence diagram for the reservation process:

User -> System: Search for available cars

System -> CarInventory: Get available cars

CarInventory -> System: Return list of available cars

System -> User: Display available cars

User -> System: Select a car and make reservation

System -> Reservation: Create reservation

Reservation -> CarInventory: Check availability

CarInventory -> Reservation: Confirm availability

Reservation -> System: Reservation confirmed

System -> NotificationService: Send notification

NotificationService -> User: Notification sent

## **Design Pattern**

Design patterns are reusable solutions to common software design problems. For a car rental system, various design patterns can be applied to enhance modularity, scalability, and maintainability. Here are some key design patterns and how they can be applied to the car rental system:

### 1. Singleton Pattern

**Use Case:** To ensure that only one instance of a class is created and provide a global point of access to it.  
**Application:**

* **CarInventory:** Ensure that there is a single inventory of cars that can be accessed and modified by various parts of the system.
* **NotificationService:** Ensure that notifications are managed by a single service instance.

public class CarInventory {

private static CarInventory instance;

private Map<CarType, List<Car>> cars;

private CarInventory() {

cars = new HashMap<>();

}

public static synchronized CarInventory getInstance() {

if (instance == null) {

instance = new CarInventory();

}

return instance;

}

// Other methods to manage inventory

}

### 2. Factory Pattern

**Use Case:** To create objects without specifying the exact class of object that will be created.  
**Application:**

* **CarFactory:** Create different types of car objects based on the car type.

public abstract class Car {

String carID;

String licensePlate;

CarType type;

String model;

String brand;

// Common methods

}

public class SUV extends Car {

// Specific methods for SUV

}

public class Sedan extends Car {

// Specific methods for Sedan

}

public class CarFactory {

public static Car createCar(CarType type) {

switch (type) {

case SUV:

return new SUV();

case SEDAN:

return new Sedan();

// Add other car types

default:

throw new IllegalArgumentException("Car type not supported.");

}

}

}

### 3. Observer Pattern

**Use Case:** To define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.  
**Application:**

* **NotificationService:** Notify users when a reservation is confirmed, a rental is started, or a payment is processed.

public interface Observer {

void update(String message);

}

public class User implements Observer {

String userID;

String name;

String email;

@Override

public void update(String message) {

// Send notification to user

}

}

public class NotificationService {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers(String message) {

for (Observer observer : observers) {

observer.update(message);

}

}

}

### 4. Strategy Pattern

**Use Case:** To define a family of algorithms, encapsulate each one, and make them interchangeable.  
**Application:**

* **Payment processing:** Implement different payment strategies (credit card, PayPal, etc.).

public interface PaymentStrategy {

void pay(double amount);

}

public class CreditCardPayment implements PaymentStrategy {

@Override

public void pay(double amount) {

// Process credit card payment

}

}

public class PayPalPayment implements PaymentStrategy {

@Override

public void pay(double amount) {

// Process PayPal payment

}

}

public class Payment {

private PaymentStrategy paymentStrategy;

public Payment(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void processPayment(double amount) {

paymentStrategy.pay(amount);

}

}

### 5. Template Method Pattern

**Use Case:** To define the skeleton of an algorithm in a method, deferring some steps to subclasses.  
**Application:**

* **Rental process:** Define a template for the rental process where specific steps can be implemented by different rental types.

public abstract class RentalProcess {

public final void processRental() {

selectCar();

confirmRental();

processPayment();

startRental();

}

protected abstract void selectCar();

protected abstract void confirmRental();

protected abstract void processPayment();

protected abstract void startRental();

}

public class OnlineRentalProcess extends RentalProcess {

@Override

protected void selectCar() {

// Online car selection

}

@Override

protected void confirmRental() {

// Online rental confirmation

}

@Override

protected void processPayment() {

// Online payment processing

}

@Override

protected void startRental() {

// Start rental process

}

}

### 6. Command Pattern

**Use Case:** To encapsulate a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.  
**Application:**

* **Reservation and rental operations:** Encapsulate actions like create, cancel, and modify reservations/rentals as command objects.

public interface Command {

void execute();

}

public class CreateReservationCommand implements Command {

private Reservation reservation;

public CreateReservationCommand(Reservation reservation) {

this.reservation = reservation;

}

@Override

public void execute() {

// Logic to create a reservation

}

}

public class CancelReservationCommand implements Command {

private Reservation reservation;

public CancelReservationCommand(Reservation reservation) {

this.reservation = reservation;

}

@Override

public void execute() {

// Logic to cancel a reservation

}

}

public class ReservationInvoker {

private List<Command> commandHistory = new ArrayList<>();

public void executeCommand(Command command) {

command.execute();

commandHistory.add(command);

}

}

## Interactions and Sequence Diagrams

#### User makes a reservation:

1. User searches for available cars.
2. User selects a car and makes a reservation.
3. System checks car availability.
4. If available, reservation is confirmed and a notification is sent to the user.

User -> System: Search for available cars

System -> CarInventory: Get available cars

CarInventory -> System: Return list of available cars

System -> User: Display available cars

User -> System: Select a car and make reservation

System -> Reservation: Create reservation

Reservation -> CarInventory: Check availability

CarInventory -> Reservation: Confirm availability

Reservation -> System: Reservation confirmed

System -> NotificationService: Send notification

NotificationService -> User: Notification sent

#### User rents a car:

1. User confirms the rental at the start time.
2. System updates car availability.
3. Rental is recorded, and payment is processed.
4. A notification is sent confirming the rental.

User -> System: Confirm rental

System -> CarInventory: Update car availability

CarInventory -> System: Car marked as unavailable

System -> RentalRecord: Record rental details

System -> Payment: Process payment

Payment -> PaymentGateway: Payment transaction

PaymentGateway -> Payment: Payment confirmed

Payment -> System: Update payment status

System -> NotificationService: Send rental confirmation notification

NotificationService -> User: Notification sent

#### Admin manages inventory:

1. Admin adds/removes cars from the inventory.
2. Admin updates car details and availability.

Admin -> System: Add car

System -> CarInventory: Add car to inventory

CarInventory -> System: Car added

Admin -> System: Update car details

System -> CarInventory: Update car details

CarInventory -> System: Car details updated

Admin -> System: Remove car

System -> CarInventory: Remove car from inventory

CarInventory -> System: Car removed