# **Food Delivery App**

## Features

### 1. User Management

#### Requirements

* **Register User**: The system should allow new users to register by providing their name, email, phone number, and address.
* **Get User Details**: The system should allow retrieving details of a user by their user ID.
* **List Users**: The system should support listing all registered users.

#### Methods

* User registerUser(String name, String email, String phone, String address)
* User getUser(String userId)

### 2. Restaurant Management

#### Requirements

* **Register Restaurant**: The system should allow new restaurants to register by providing their name, address, and menu items.
* **Get Restaurant Details**: The system should allow retrieving details of a restaurant by its restaurant ID.
* **List Restaurants**: The system should support listing all registered restaurants.

#### Methods

* Restaurant registerRestaurant(String name, String address, List<MenuItem> menu)
* Restaurant getRestaurant(String restaurantId)

### 3. Menu Management

#### Requirements

* **Add Menu Item**: The system should allow adding menu items to a restaurant's menu.
* **Get Menu Item Details**: The system should allow retrieving details of a menu item by its item ID.
* **List Menu Items**: The system should support listing all menu items of a restaurant.

#### Methods

* MenuItem addMenuItem(String name, double price)
* MenuItem getMenuItem(String itemId)

### 4. Order Management

#### Requirements

* **Place Order**: The system should allow users to place orders by selecting menu items from a restaurant.
* **Get Order Details**: The system should allow retrieving details of an order by its order ID.
* **Update Order Status**: The system should allow updating the status of an order (e.g., "Placed", "Dispatched", "Delivered").
* **List Orders**: The system should support listing all orders placed by a user or for a restaurant.

#### Methods

* Order placeOrder(User user, Restaurant restaurant, List<MenuItem> items)
* Order getOrder(String orderId)
* void updateOrderStatus(String orderId, String status)

### 5. Delivery Management

#### Requirements

* **Assign Delivery**: The system should allow assigning a delivery person to an order.
* **Get Delivery Details**: The system should allow retrieving details of a delivery by its delivery ID.
* **Update Delivery Status**: The system should allow updating the status of a delivery (e.g., "Assigned", "In Transit", "Delivered").
* **List Deliveries**: The system should support listing all deliveries assigned to a delivery person.

#### Methods

* Delivery assignDelivery(Order order, String deliveryPerson)
* Delivery getDelivery(String deliveryId)
* void updateDeliveryStatus(String deliveryId, String status)

### 6. Sample Use Case

#### Register a User

* **Input**: User details (name, email, phone, address)
* **Process**: Call UserService.registerUser(name, email, phone, address)
* **Output**: New User object

#### Register a Restaurant

* **Input**: Restaurant details (name, address, menu)
* **Process**: Call RestaurantService.registerRestaurant(name, address, menu)
* **Output**: New Restaurant object

#### Place an Order

* **Input**: User, Restaurant, List of Menu Items
* **Process**: Call OrderService.placeOrder(user, restaurant, items)
* **Output**: New Order object

#### Assign a Delivery

* **Input**: Order, Delivery Person
* **Process**: Call DeliveryService.assignDelivery(order, deliveryPerson)
* **Output**: New Delivery object

## Design Patterns

1. **Singleton Pattern**: To ensure only one instance of service classes.
2. **Factory Pattern**: To abstract object creation logic.
3. **Observer Pattern**: To notify various parts of the system when an order status changes.
4. **Strategy Pattern**: To handle different payment methods dynamically.
5. **Command Pattern**: To encapsulate actions as objects and support undo/redo operations.
6. **Builder Pattern**: To simplify the construction of complex objects.

### 1. Singleton Pattern

**Use Case**: To ensure that only one instance of a service class (like UserService, RestaurantService, OrderService, DeliveryService) is created throughout the application.

**Implementation**:

public class UserService {

private static UserService instance;

private List<User> users;

private UserService() {

users = new ArrayList<>();

}

public static synchronized UserService getInstance() {

if (instance == null) {

instance = new UserService();

}

return instance;

}

// Other methods...

}

### 2. Factory Pattern

**Use Case**: To create objects like User, Restaurant, MenuItem, Order, and Delivery without exposing the instantiation logic to the client and to provide a layer of abstraction.

**Implementation**:

public class EntityFactory {

public static User createUser(String name, String email, String phone, String address) {

return new User(/\* generate userId \*/, name, email, phone, address);

}

public static Restaurant createRestaurant(String name, String address, List<MenuItem> menu) {

return new Restaurant(/\* generate restaurantId \*/, name, address, menu);

}

public static MenuItem createMenuItem(String name, double price) {

return new MenuItem(/\* generate itemId \*/, name, price);

}

public static Order createOrder(User user, Restaurant restaurant, List<MenuItem> items) {

double totalAmount = items.stream().mapToDouble(MenuItem::getPrice).sum();

return new Order(/\* generate orderId \*/, user, restaurant, items, totalAmount, "Placed");

}

public static Delivery createDelivery(Order order, String deliveryPerson) {

return new Delivery(/\* generate deliveryId \*/, order, deliveryPerson, "Assigned");

}

}

### 3. Observer Pattern

**Use Case**: To notify various parts of the system when an order status changes. For example, notifying the user, the restaurant, and the delivery service.

**Implementation**:

import java.util.ArrayList;

import java.util.List;

// Observer interface

interface OrderStatusObserver {

void update(String orderId, String status);

}

// Concrete observer for User

public class User implements OrderStatusObserver {

private String userId;

private String name;

private String email;

private String phone;

private String address;

// Constructors, getters, and setters

@Override

public void update(String orderId, String status) {

System.out.println("User " + name + " notified: Order " + orderId + " is now " + status);

}

}

// Concrete observer for Restaurant

public class Restaurant implements OrderStatusObserver {

private String restaurantId;

private String name;

private String address;

private List<MenuItem> menu;

// Constructors, getters, and setters

@Override

public void update(String orderId, String status) {

System.out.println("Restaurant " + name + " notified: Order " + orderId + " is now " + status);

}

}

// Subject class

public class Order {

private String orderId;

private User user;

private Restaurant restaurant;

private List<MenuItem> items;

private double totalAmount;

private String status;

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

// Constructors, getters, and setters

public void addObserver(OrderStatusObserver observer) {

observers.add(observer);

}

public void setStatus(String status) {

this.status = status;

notifyAllObservers();

}

private void notifyAllObservers() {

for (OrderStatusObserver observer : observers) {

observer.update(orderId, status);

}

}

}

### 4. Strategy Pattern

**Use Case**: To handle different payment methods (e.g., credit card, PayPal, cash) dynamically without changing the order processing logic.

**Implementation**:

// Strategy interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete strategy for Credit Card

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

private String expiryDate;

private String cvv;

// Constructors, getters, and setters

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

// Concrete strategy for PayPal

public class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

// Constructors, getters, and setters

@Override

public void pay(double amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

// Context class

public class Order {

private String orderId;

private User user;

private Restaurant restaurant;

private List<MenuItem> items;

private double totalAmount;

private String status;

private PaymentStrategy paymentStrategy;

// Constructors, getters, and setters

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay() {

paymentStrategy.pay(totalAmount);

}

}

### 5. Command Pattern

**Use Case**: To handle actions like placing an order, canceling an order, or updating an order status in a way that allows for undo/redo operations.

**Implementation**:

// Command interface

interface Command {

void execute();

}

// Concrete command for placing an order

public class PlaceOrderCommand implements Command {

private OrderService orderService;

private User user;

private Restaurant restaurant;

private List<MenuItem> items;

// Constructor

public PlaceOrderCommand(OrderService orderService, User user, Restaurant restaurant, List<MenuItem> items) {

this.orderService = orderService;

this.user = user;

this.restaurant = restaurant;

this.items = items;

}

@Override

public void execute() {

orderService.placeOrder(user, restaurant, items);

}

}

// Invoker class

public class OrderInvoker {

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

public void executeCommand(Command command) {

command.execute();

commandHistory.add(command);

}

// Method to undo commands if needed

}

## Define the Core Components

First, identify the core components of the system:

* **User**: The customer using the app.
* **Restaurant**: The entity providing food.
* **Menu**: The list of food items a restaurant offers.
* **Order**: The customer's order.
* **Delivery**: The process of delivering the order.

## Create the Core Classes

Define the core classes and their attributes.

#### User Class

public class User {

private String userId;

private String name;

private String email;

private String phone;

private String address;

// Constructors, getters, and setters

}

#### Restaurant Class

public class Restaurant {

private String restaurantId;

private String name;

private String address;

private List<MenuItem> menu;

// Constructors, getters, and setters

}

#### MenuItem Class

public class MenuItem {

private String itemId;

private String name;

private double price;

// Constructors, getters, and setters

}

#### Order Class

public class Order {

private String orderId;

private User user;

private Restaurant restaurant;

private List<MenuItem> items;

private double totalAmount;

private String status; // e.g., "Placed", "Dispatched", "Delivered"

// Constructors, getters, and setters

}

#### Delivery Class

public class Delivery {

private String deliveryId;

private Order order;

private String deliveryPerson;

private String status; // e.g., "Assigned", "In Transit", "Delivered"

// Constructors, getters, and setters

}

## Create Services for Business Logic

Define services to handle business logic.

#### UserService

public class UserService {

private List<User> users;

public User registerUser(String name, String email, String phone, String address) {

User user = new User(/\* generate userId \*/, name, email, phone, address);

users.add(user);

return user;

}

public User getUser(String userId) {

for (User user : users) {

if (user.getUserId().equals(userId)) {

return user;

}

}

return null;

}

// Additional methods for user operations

}

RestaurantService

public class RestaurantService {

private List<Restaurant> restaurants;

public Restaurant registerRestaurant(String name, String address, List<MenuItem> menu) {

Restaurant restaurant = new Restaurant(/\* generate restaurantId \*/, name, address, menu);

restaurants.add(restaurant);

return restaurant;

}

public Restaurant getRestaurant(String restaurantId) {

for (Restaurant restaurant : restaurants) {

if (restaurant.getRestaurantId().equals(restaurantId)) {

return restaurant;

}

}

return null;

}

// Additional methods for restaurant operations

}

OrderService

public class OrderService {

private List<Order> orders;

public Order placeOrder(User user, Restaurant restaurant, List<MenuItem> items) {

double totalAmount = items.stream().mapToDouble(MenuItem::getPrice).sum();

Order order = new Order(/\* generate orderId \*/, user, restaurant, items, totalAmount, "Placed");

orders.add(order);

return order;

}

public Order getOrder(String orderId) {

for (Order order : orders) {

if (order.getOrderId().equals(orderId)) {

return order;

}

}

return null;

}

public void updateOrderStatus(String orderId, String status) {

Order order = getOrder(orderId);

if (order != null) {

order.setStatus(status);

}

}

// Additional methods for order operations

}

DeliveryService

public class DeliveryService {

private List<Delivery> deliveries;

public Delivery assignDelivery(Order order, String deliveryPerson) {

Delivery delivery = new Delivery(/\* generate deliveryId \*/, order, deliveryPerson, "Assigned");

deliveries.add(delivery);

return delivery;

}

public Delivery getDelivery(String deliveryId) {

for (Delivery delivery : deliveries) {

if (delivery.getDeliveryId().equals(deliveryId)) {

return delivery;

}

}

return null;

}

public void updateDeliveryStatus(String deliveryId, String status) {

Delivery delivery = getDelivery(deliveryId);

if (delivery != null) {

delivery.setStatus(status);

}

}

// Additional methods for delivery operations

}

## Main Class to Demonstrate Functionality

public class FoodDeliveryApp {

public static void main(String[] args) {

UserService userService = new UserService();

RestaurantService restaurantService = new RestaurantService();

OrderService orderService = new OrderService();

DeliveryService deliveryService = new DeliveryService();

// Register users

User user1 = userService.registerUser("Alice", "alice@example.com", "1234567890", "123 Street, City");

// Register restaurants

List<MenuItem> menu = new ArrayList<>();

menu.add(new MenuItem(/\* generate itemId \*/, "Burger", 5.99));

menu.add(new MenuItem(/\* generate itemId \*/, "Pizza", 8.99));

Restaurant restaurant1 = restaurantService.registerRestaurant("Food Place", "456 Avenue, City", menu);

// Place an order

List<MenuItem> orderItems = new ArrayList<>(menu);

Order order1 = orderService.placeOrder(user1, restaurant1, orderItems);

// Assign delivery

Delivery delivery1 = deliveryService.assignDelivery(order1, "John Doe");

// Update order and delivery status

orderService.updateOrderStatus(order1.getOrderId(), "Dispatched");

deliveryService.updateDeliveryStatus(delivery1.getDeliveryId(), "In Transit");

// Print order and delivery details

System.out.println("Order Details: " + order1);

System.out.println("Delivery Details: " + delivery1);

}

}