Java Programming

Project Report

Zest

Submitted by:
Ahmed Hesham
Ziad Osama
Hamdy Ashraf

TA: **Eng. Aya Barakat**
Date: **Monday, December 29, 2025**

Brief Description

Zest is a comprehensive Java-based simulation of a multi-restaurant food delivery platform designed to demonstrate advanced Object-Oriented Programming principles. Unlike basic ordering systems, Zest addresses complex real-world challenges such as data integrity, and collaborative group ordering.

The system allows **Customers** to browse restaurants, place orders, and split bills with friends via a GroupOrder feature. The platform acts as a centralized "Single Source of Truth" (Singleton), ensuring accurate data flow. Key technical highlights include a **Snapshot Pattern** (OrderItem) to freeze item prices at the time of purchase and a **Strategy Pattern** to allow flexible, interchangeable payment methods without modifying core business logic.

Overall Class Structure

The system is architected around a centralized controller and strict domain separation:

- **Platform (Singleton)**: The main entry point that maintains the registry of all Users (Customers/Merchants) and Restaurants. It handles global search and authentication.
- **User Hierarchy**: An abstract User class. This allows specific behaviors to be isolated in subclasses: Customer and Merchant.
- **Order Domain**: The Order class is the transactional core. It utilizes polymorphism to support GroupOrder, which extends Order to add participant tracking and bill-splitting logic.

Class Relations & Design Decisions

We utilized specific relationships to ensure memory management and logical consistency:

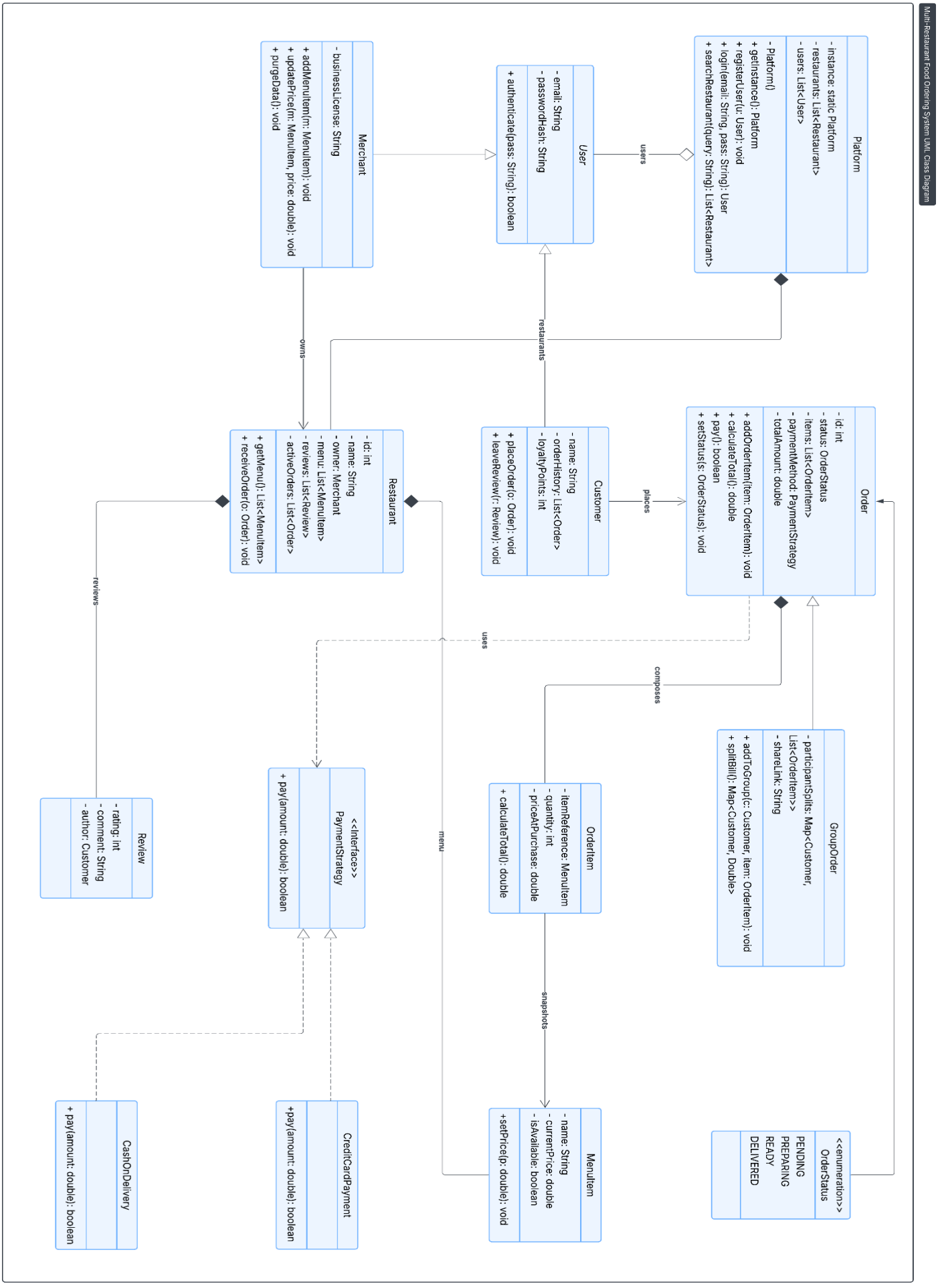
- **Composition (Black Diamond):** Used between Restaurant and MenuItem. This enforces a strict lifecycle dependency; if a Restaurant is deleted from the platform, its Menu and Reviews are automatically destroyed, preventing orphaned data.
- **Snapshot Association:** A critical relationship exists between OrderItem and MenuItem. Instead of an Order referencing a live MenuItem (which changes price), the Order composes OrderItem objects. These objects "snapshot" the price and name at the moment of purchase, ensuring historical financial records remain accurate even if the restaurant updates the menu later.
- **Inheritance:** GroupOrder inherits from Order, allowing the system to treat single and group orders identically during processing (Polymorphism) while enabling unique functionality for the group.

Usage of Interfaces

To satisfy the Open/Closed Principle (Open for extension, closed for modification), we implemented the **Strategy Pattern** for payments:

- **PaymentStrategy (Interface):** Defines the contract method pay(double amount).
- **Implementations:** CreditCardPayment and CashOnDelivery implement this interface with specific validation logic.
- **Benefit:** This allows the system to support new payment types (e.g., Digital Wallet, Crypto) in the future without changing a single line of code in the Order class.

UML Diagram


Multi-Restaurant Food Ordering System UML Class Diagram
Platform
- instance: static Platform
- restaurants: List<Restaurant>
- users: List<User>
- Platform()
+ getInstance(): Platform
+ registerUser(u: User): void
+ login(email: String, pass: String): User
+ searchRestaurant(query: String): List<Restaurant>
User
- email: String
- passwordHash: String
+ authenticate(pass: String): boolean
Merchant
- businessLicense: String
+ addMenuItem(m: MenuItem): void
+ updatePrice(m: MenuItem, price: double): void
+ purgeData(): void
users
owns
restaurants
Restaurant
- id: int
- name: String
- owner: Merchant
- menu: List<MenuItem>
- reviews: List<Review>
- activeOrders: List<Order>
+ getMenu(): List<MenuItem>
+ receiveOrder(o: Order): void
Customer
- name: String
- orderHistory: List<Order>
- loyaltyPoints: int
+ placeOrder(o: Order): void
+ leaveReview(r: Review): void
places
Order
- id: int
- status: OrderStatus
- items: List<OrderItem>
- paymentMethod: PaymentStrategy
- totalAmount: double
+ addOrderItem(item: OrderItem): void
+ calculateTotal(): double
+ pay(): boolean
+ setStatus(s: OrderStatus): void
uses
composes
reviews
menu
snapshots
Review
- rating: int
- comment: String
- author: Customer
<<interface>>
PaymentStrategy
+ pay(amount: double): boolean
OrderItem
- itemReference: MenuItem
- quantity: int
- priceAtPurchase: double
+ calculateTotal(): double
GroupOrder
- participantSplits: Map<Customer, List<OrderItem>>
- shareLink: String
+ addToGroup(c: Customer, item: OrderItem): void
+ splitBill(): Map<Customer, Double>
MenuItem
- name: String
- currentPrice: double
- isAvailable: boolean
+setPrice(p: double): void
<<enumeration>>
OrderStatus
PENDING
PREPARING
READY
DELIVERED
CashOnDelivery
+ pay(amount: double): boolean
CreditCardPayment
+pay(amount: double): boolean

Key Implementation Details

A. Integration of Patterns (Controller Logic)

The CheckoutController serves as the central integration point. It demonstrates the Strategy Pattern by dynamically selecting a payment method and the Snapshot Pattern by capturing item prices at the moment of purchase before saving to the database.

File: CheckoutController.java

```
@FXML
private void handlePlaceOrder() {
    PaymentStrategy paymentStrategy;
    if (paypalRadio.isSelected()) {
        paymentStrategy = new PayPalAdapter(); /
    } else {
        paymentStrategy = new CashPayment();
    }

    Order order = new Order();
    order.setPaymentMethod(paymentStrategy);

    for (Map.Entry<MenuItem, Integer> entry : cart.getItemQuantities().entrySet()) {
        MenuItem item = entry.getKey();
        double priceAtPurchase = item.getPrice();

        OrderItem orderItem = new OrderItem(item, entry.getValue(), priceAtPurchase);
        order.addOrderItem(orderItem);
    }

    if (order.pay()) {
        dataService.saveOrder(order);
    }
}
```

B. The Snapshot Pattern (Data Integrity)

To ensure financial accuracy, the OrderItem class decouples the historical order price from the live menu price. The priceAtPurchase field is immutable after instantiation.

File: OrderItem.java

```
public class OrderItem {
    private MenuItem itemReference;
    private double priceAtPurchase;

    public OrderItem(MenuItem itemReference, int quantity, double priceAtPurchase) {
        this.itemReference = itemReference;
        this.quantity = quantity;
        this.priceAtPurchase = priceAtPurchase;
    }

    public double calculateTotal() {
        return priceAtPurchase * quantity;
    }
}
```

C. The Singleton Pattern (Database Connectivity)

To manage resources efficiently and prevent connection conflicts, the DBConnection class ensures only one instance of the database connection exists throughout the application lifecycle.

File: DBConnection.java

```
public class DBConnection {
    // The single static instance
    private static DBConnection instance;
    private Connection connection;

    // Private constructor prevents direct instantiation
    private DBConnection() {
        try {
            // SQLite driver auto-loads; connection is established once
            this.connection = DriverManager.getConnection(DB_URL);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // Global access point with thread-safety
    public static synchronized DBConnection getInstance() {
        if (instance == null) {
            instance = new DBConnection();
        }
        return instance;
    }
}
```

D. The Strategy & Adapter Pattern (Payment Flexibility)

This section defines the contract for payments and demonstrates the Adapter Pattern, which allows incompatible classes (like a third-party PayPal API) to work with your system.

Files: PaymentStrategy.java & PayPalAdapter.java

```
public interface PaymentStrategy {
    boolean pay(double amount);
}

public class PayPalAdapter implements PaymentStrategy {
    private XPayPalApi payPalApi;

    public PayPalAdapter() {
        this.payPalApi = new XPayPalApi();
    }

    @Override
    public boolean pay(double amount) {
        System.out.println("Adapting payment for PayPal...");
        payPalApi.processPayment(amount, "EGP");
        return true;
    }
}
```

E. Advanced Business Logic (Group Ordering)

This snippet proves you implemented the "complex real-world challenge" mentioned in your objectives: allowing users to split bills in a group order.

File: GroupOrder.java

```
public class GroupOrder extends Order {
    private Map<Customer, List<OrderItem>> participantSplits;

    public Map<Customer, Double> splitBill() {
        Map<Customer, Double> billSplit = new HashMap<>();

        for (Map.Entry<Customer, List<OrderItem>> entry : participantSplits.entrySet()) {
            double customerTotal = entry.getValue().stream()
             .mapToDouble(OrderItem::calculateTotal)
                .sum();

            billSplit.put(entry.getKey(), customerTotal);
        }
        return billSplit;
    }
}
```

F. Security & Inheritance (User Hierarchy)

This demonstrates Inheritance (Abstract Base Class) and basic security practices (Password Hashing) used in your system.

File: User.java

```
public abstract class User {
    protected String email;
    protected String passwordHash;

    public User(String email, String passwordHash) {
        this.email = email;
        this.passwordHash = passwordHash;
    }

    public boolean authenticate(String inputPassword) {
        if (inputPassword == null) return false;

        String hashedInput = hashPassword(inputPassword);
        return passwordHash.equals(hashedInput);
    }

}
```

G. Transaction Management (Data Integrity)

This snippet from your Data Access Layer proves you are handling database reliability. It shows how you use setAutoCommit(false), commit(), and rollback() to ensure that an order is only saved if all its items are also saved successfully.

File: DataService.java

```
public int saveOrder(Order order) {
    Connection conn = dbConnection.getConnection();
    try {
        conn.setAutoCommit(false);
        String itemSql = "INSERT INTO order_items (order_id, price_at_purchase, ...) VALUES (?, ?, ...)";
        try (PreparedStatement itemStmt = conn.prepareStatement(itemSql)) {
            for (OrderItem item : order.getItems()) {
                itemStmt.setDouble(2, item.getPriceAtPurchase());
                itemStmt.addBatch();
            }
            itemStmt.executeBatch();
        }

        conn.commit();
        return orderId;

    } catch (SQLException e) {
        try { conn.rollback(); } catch (SQLException ex) { ex.printStackTrace(); }
        return -1;
    }
}
```

H. Centralized State Management (The Cart)

This demonstrates a functional Singleton. Unlike DBConnection (which is just a resource handle), CartManager holds live application state. It shows business logic validation (preventing items from different restaurants).

File: CartManager.java

```
public class CartManager {
    private static CartManager instance;
    private Map<MenuItem, Integer> itemQuantities;
    private Integer currentRestaurantId;
    public static CartManager getInstance() {
        if (instance == null) instance = new CartManager();
        return instance;
    }
    public boolean addItem(MenuItem item) {
        if (currentRestaurantId != null && currentRestaurantId != item.getRestaurantId()) {
            return false;
        }

        if (currentRestaurantId == null) {
            currentRestaurantId = item.getRestaurantId();
        }

        itemQuantities.put(item, itemQuantities.getOrDefault(item, 0) + 1);
        return true;
    }
}
```

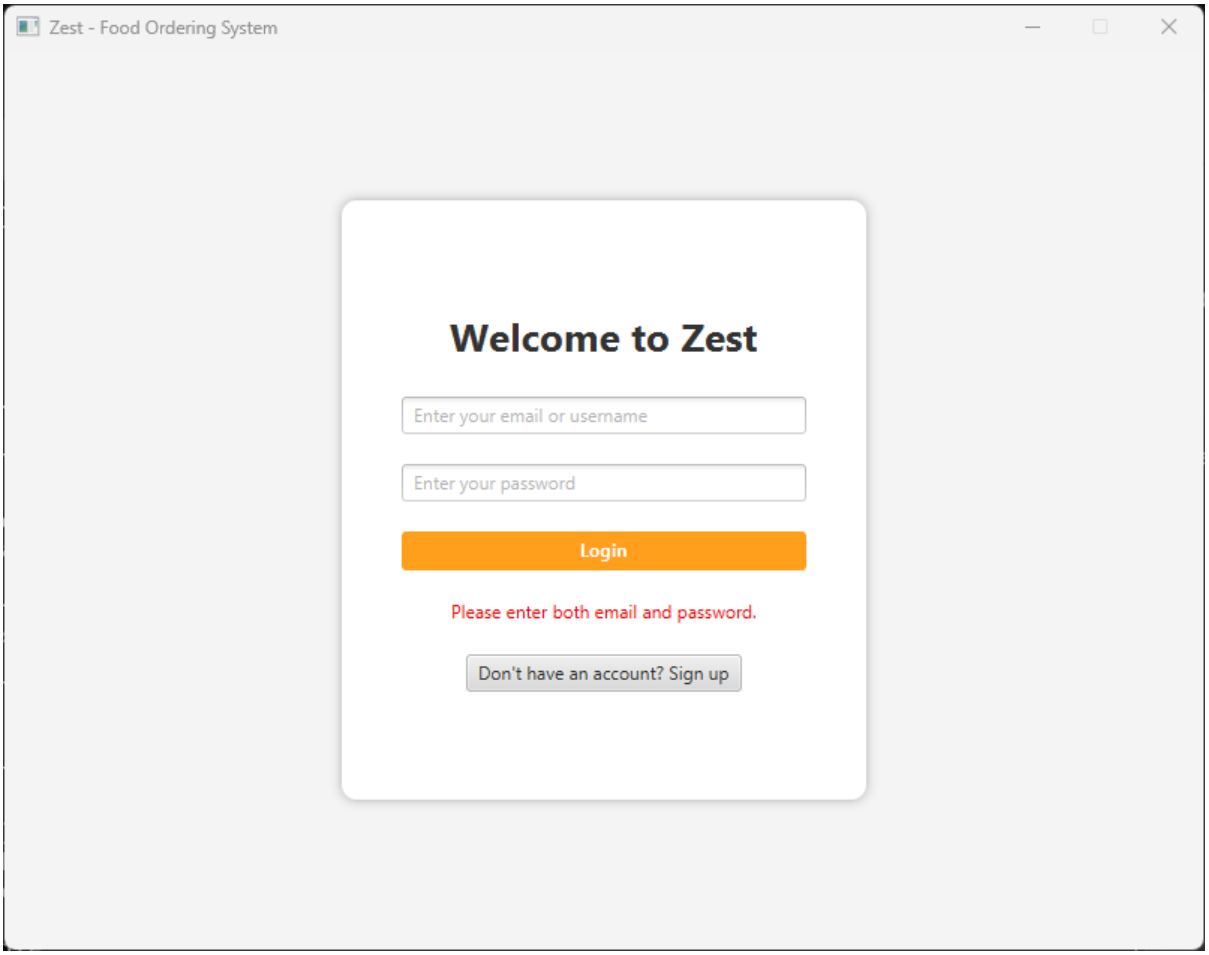
I. Multi-Threading (Concurrency)

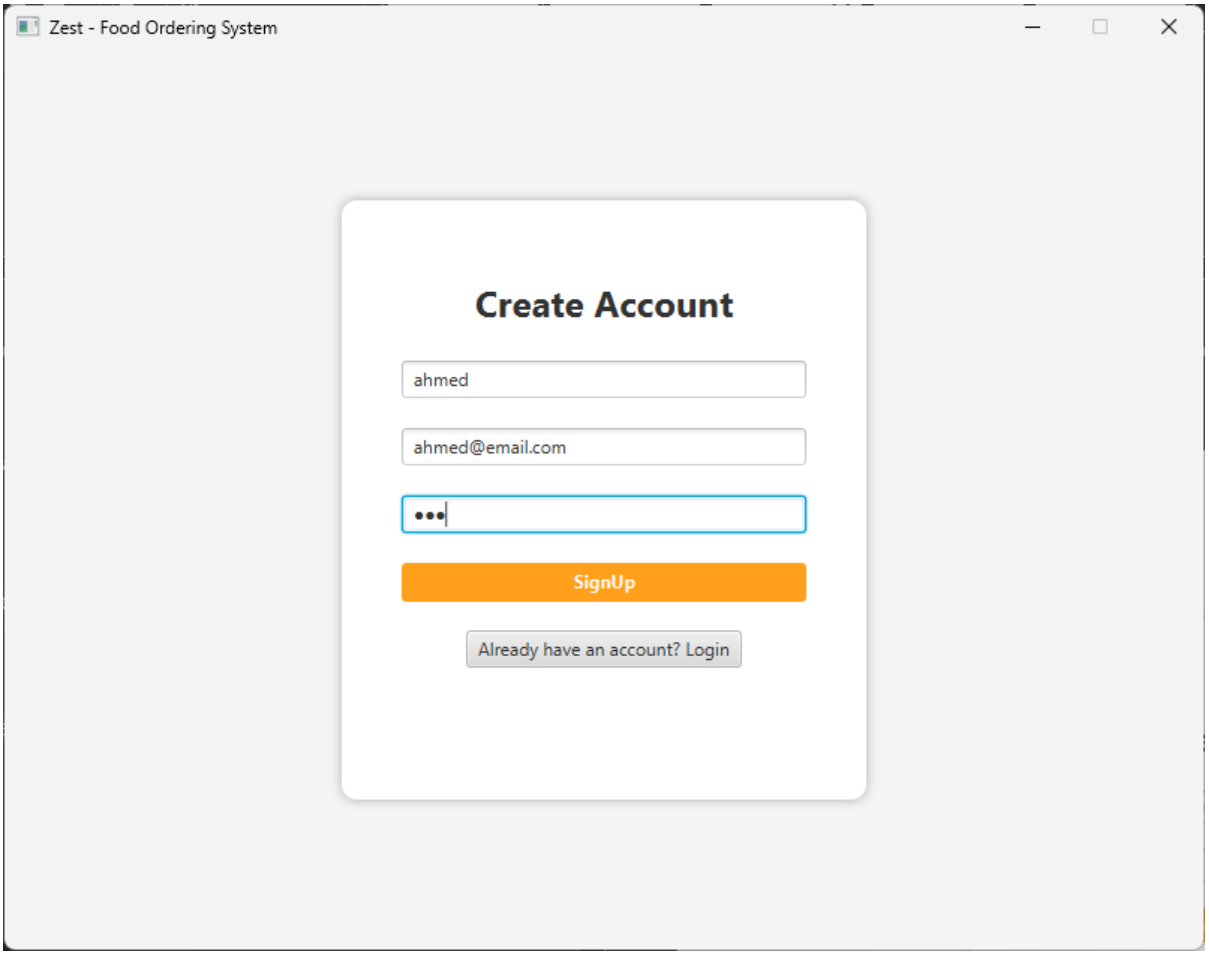
This is a "Bonus" technical highlight. It shows you know how to keep the GUI responsive by loading data on a background thread instead of blocking the main UI thread.

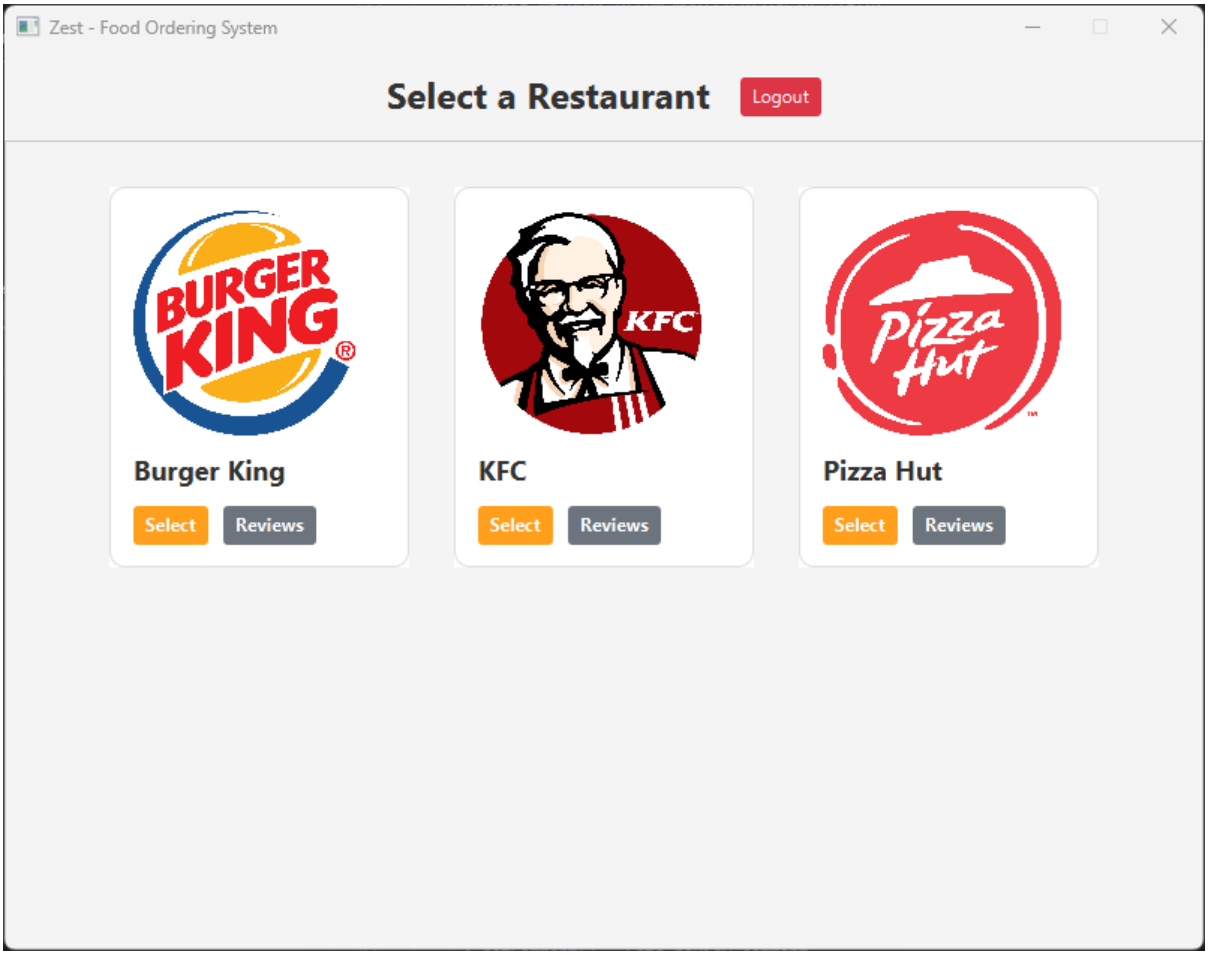
File: HomeController.java

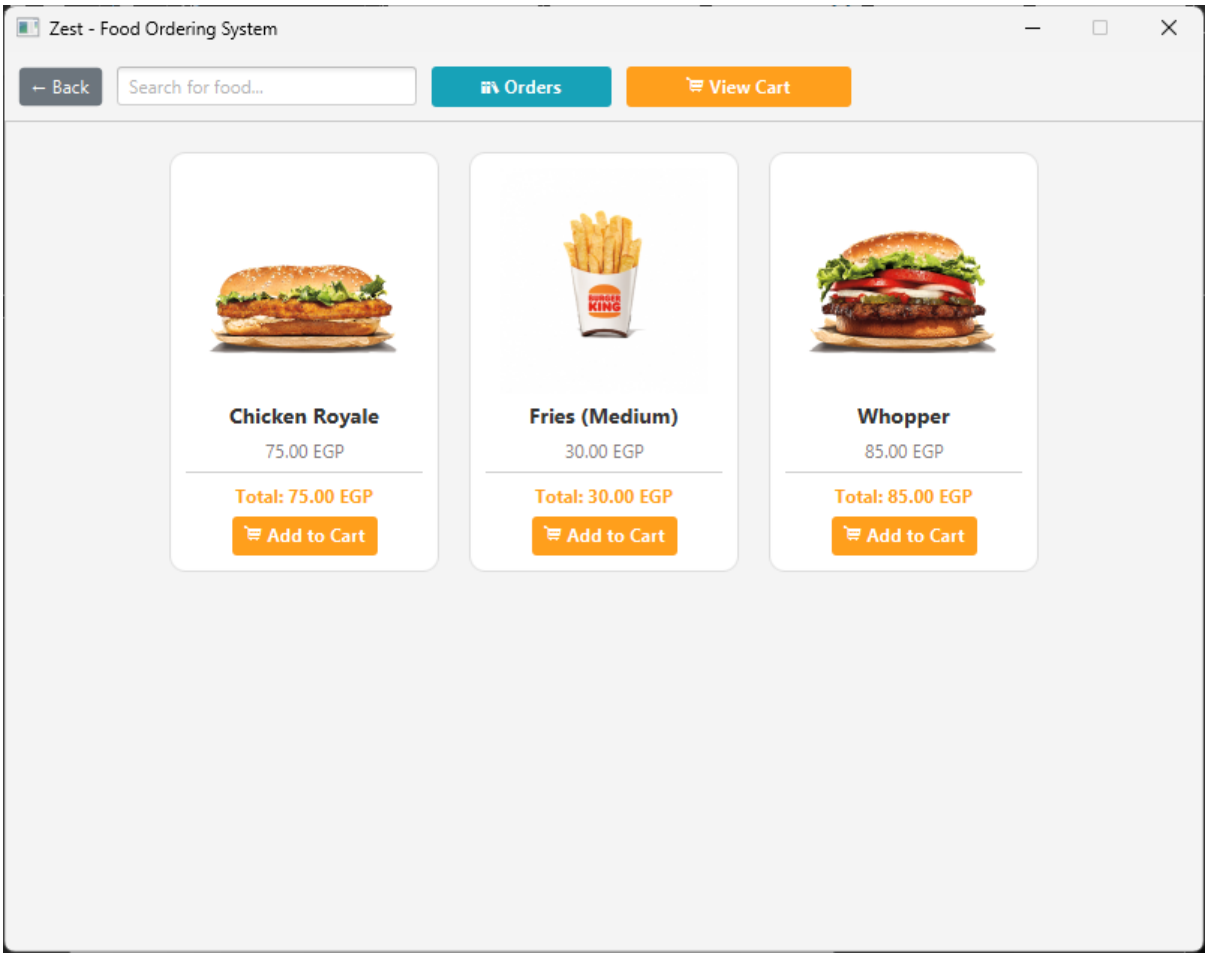
```
private void loadMenu() {
    ExecutorService executor = Executors.newSingleThreadExecutor();

    executor.submit(() -> {
        try {

            List<MenuItem> items = dataService.getMenuItemsByRestaurant(restaurantId);

                menuContainer.getChildren().clear();
                for (MenuItem item : items) {              }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
}
```

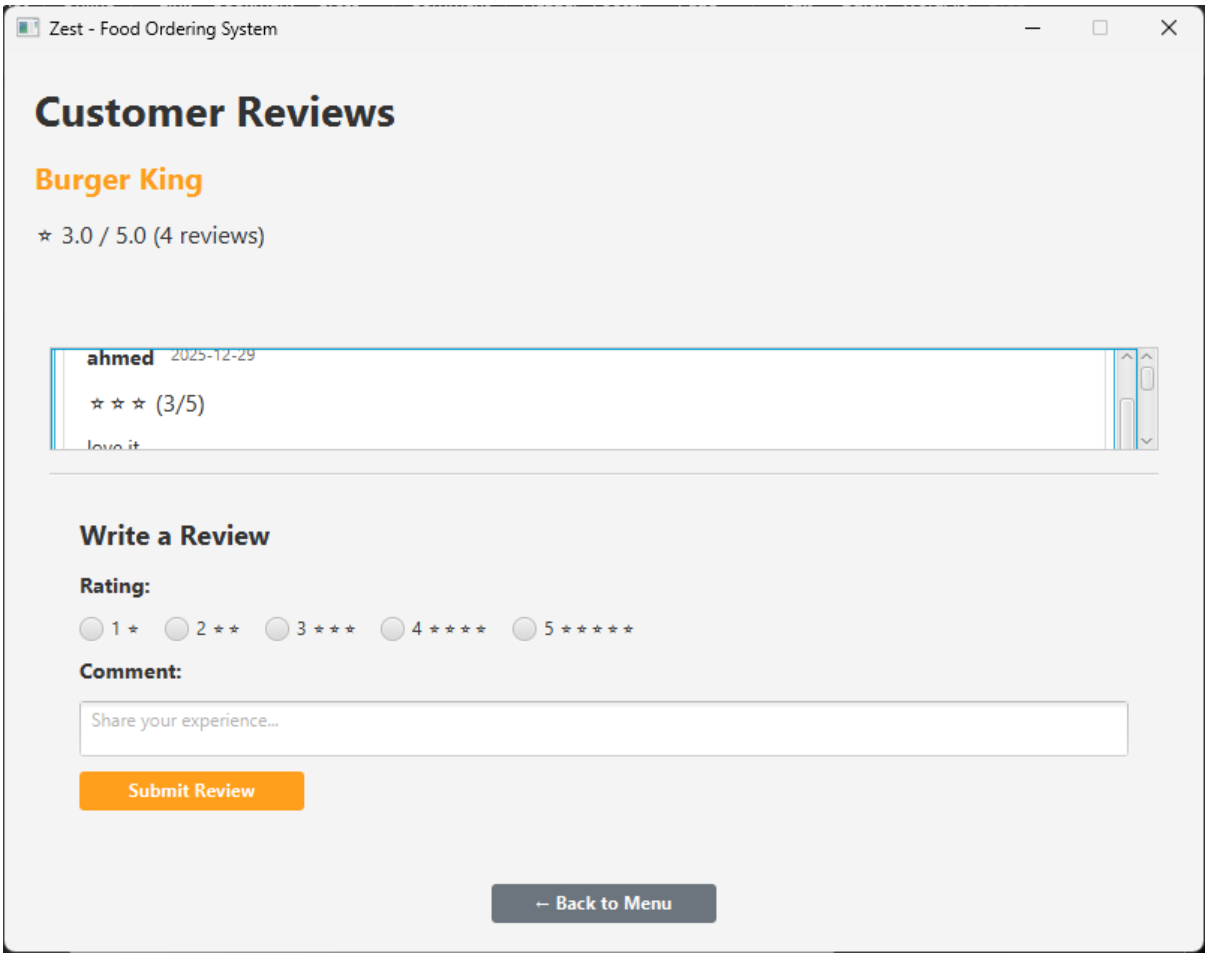
GUI Screenshots

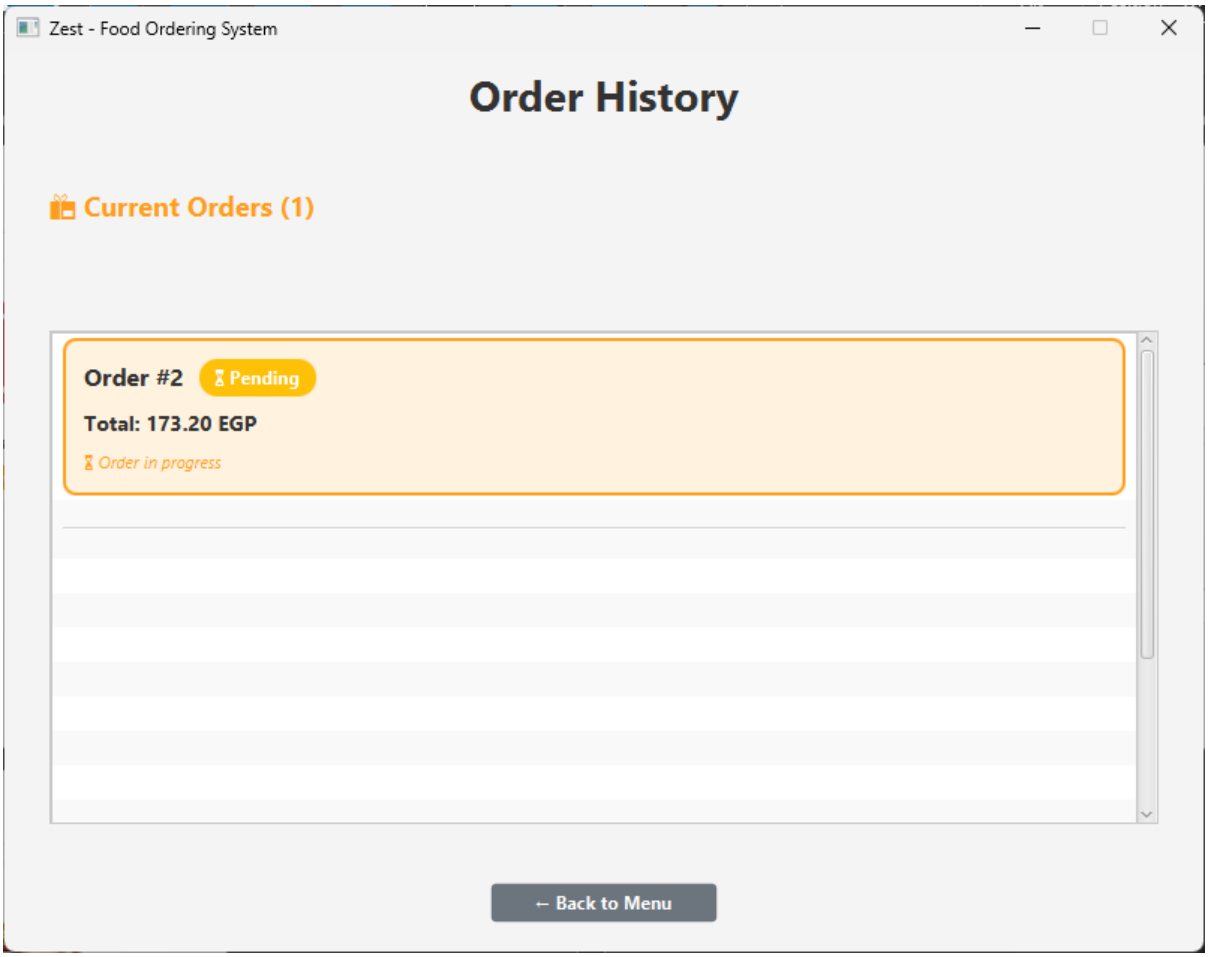


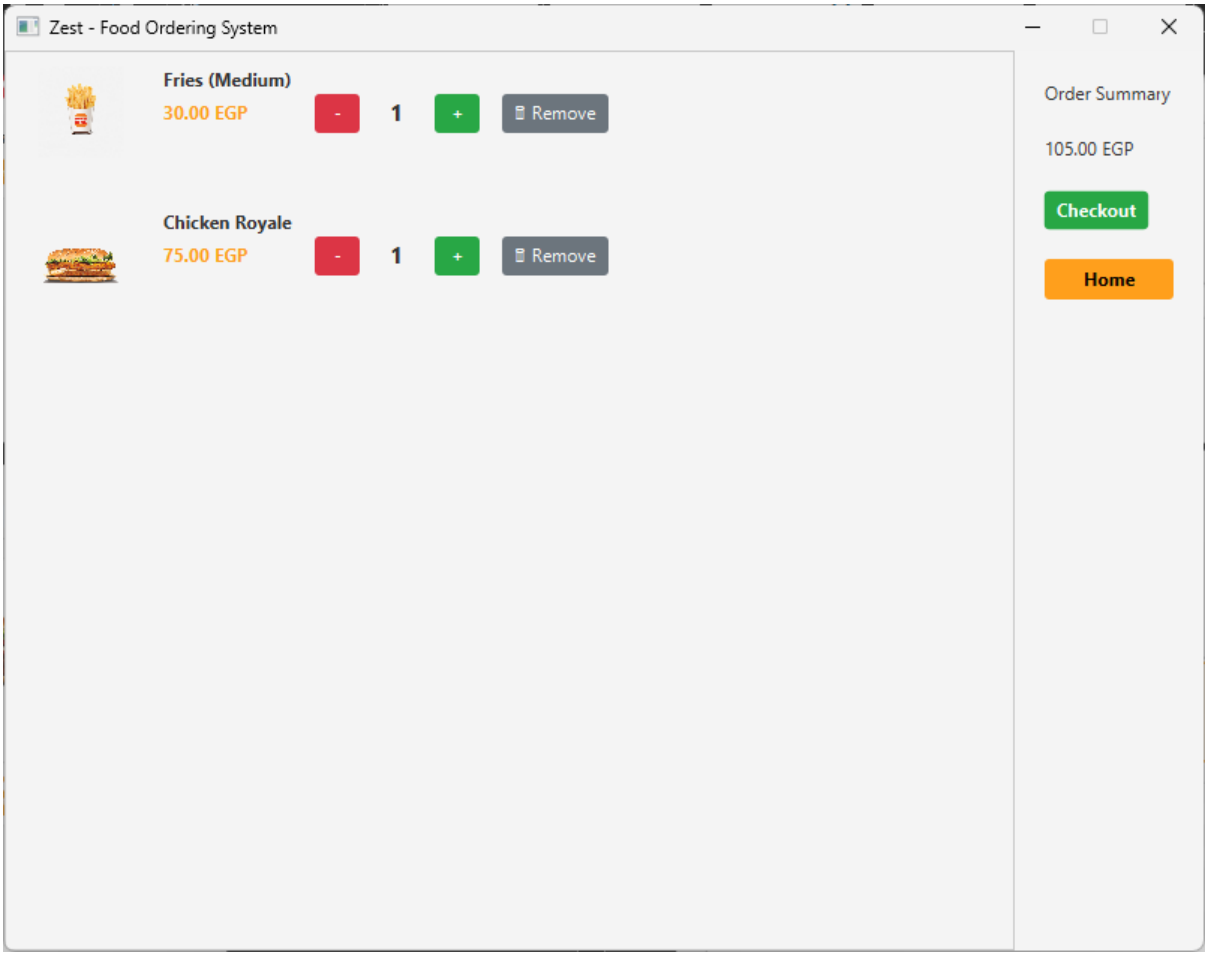


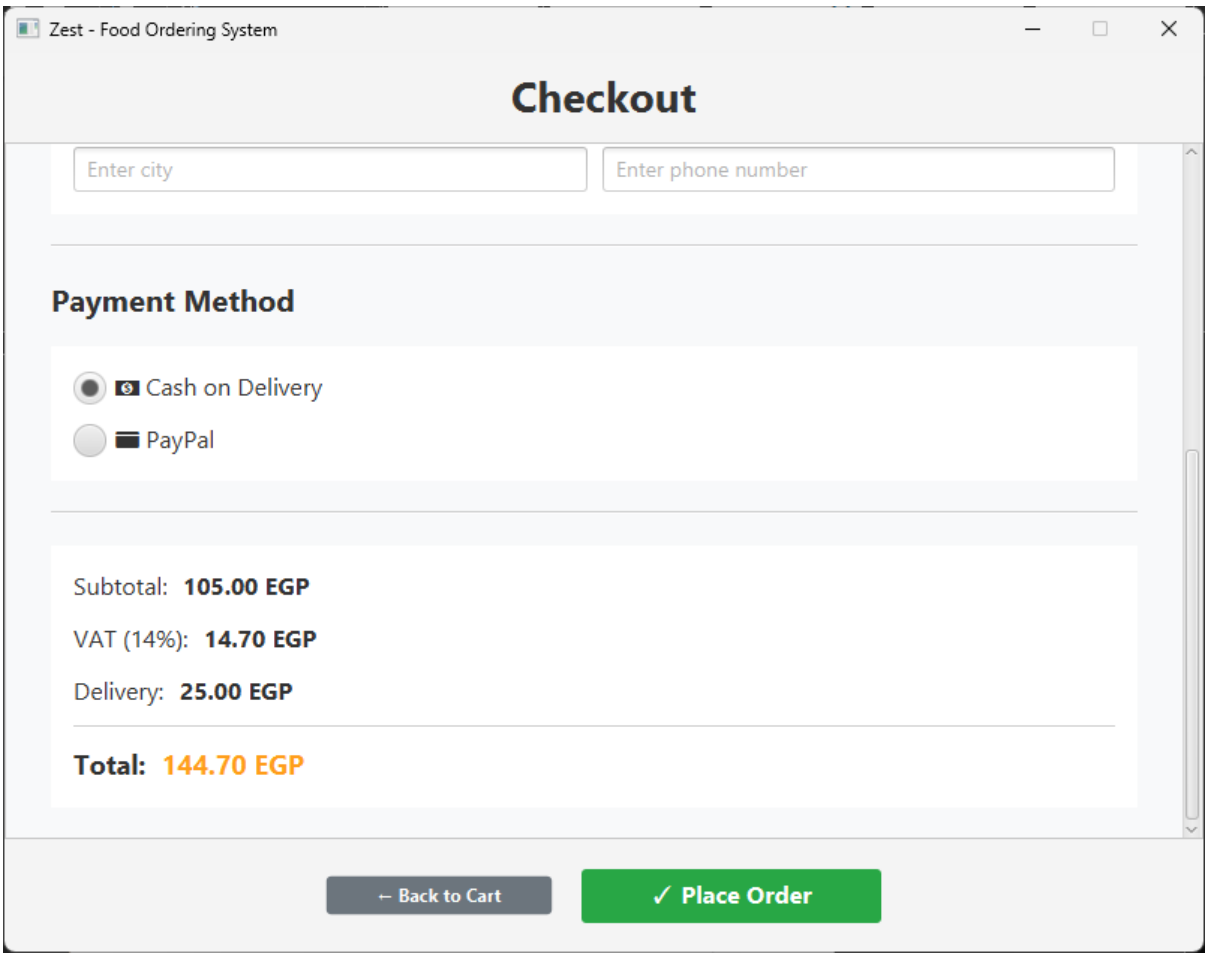

Zest - Food Ordering System
Select a Restaurant
Logout
Burger King
Select
Reviews
KFC
Select
Reviews
Pizza Hut
Select
Reviews


Zest - Food Ordering System
← Back
Search for food...
Orders
View Cart
Chicken Royale
75.00 EGP
Total: 75.00 EGP
Add to Cart
Fries (Medium)
30.00 EGP
Total: 30.00 EGP
Add to Cart
Whopper
85.00 EGP
Total: 85.00 EGP
Add to Cart


Zest - Food Ordering System
Customer Reviews
Burger King
★ 3.0 / 5.0 (4 reviews)
ahmed 2025-12-29
★★★ (3/5)
Write a Review
Rating:
1 ★ 2 ★★ 3 ★★★ 4 ★★★★ 5 ★★★★★
Comment:
Share your experience...
Submit Review
← Back to Menu


Zest - Food Ordering System
Order History
Current Orders (1)
Order #2 Pending
Total: 173.20 EGP
Order in progress
← Back to Menu


Zest - Food Ordering System
Fries (Medium)
30.00 EGP
-
1
+
Remove
Chicken Royale
75.00 EGP
-
1
+
Remove
Order Summary
105.00 EGP
Checkout
Home


Zest - Food Ordering System
Checkout
Enter city
Enter phone number
Payment Method
Cash on Delivery
PayPal
Subtotal: 105.00 EGP
VAT (14%): 14.70 EGP
Delivery: 25.00 EGP
Total: 144.70 EGP
← Back to Cart
✓ Place Order