DATABASE MANAGEMENT SYSTEM
UE23CS351A

Level 2 (Orange problem/ Mini-project) Report

Title: Food Delivery Management System

Course: UE23CS351A
Subject: Database Management System Project

Team Members:

- Anjali Gunti – PES2UG23CS075
- Avani Ajith – PES2UG23CS110

Abstract

The Food Delivery Management System is a comprehensive desktop application designed to manage the operations of a food delivery service. The system is built using Python with a Tkinter GUI for the frontend and a MySQL database for the backend. Key functionalities include full CRUD (Create, Read, Update, Delete) management for customers, staff (chefs, employees), and inventory (food, drinks). The system's core feature is its order processing module, which allows users to place new orders for customers. This process is heavily automated using database triggers: placing an order automatically decrements item stock, updates inventory availability status, and calculates the final payment amount, ensuring data integrity and reducing manual error. The application also provides a dashboard to view all order details and assign delivery drivers.

User Requirement Specification

Functional Requirements:

- **User Management:**
  - The system shall allow an admin to add new customers, staff (employees), and chefs to the database.
  - The system shall allow users to view lists of all customers, staff, and chefs.
- **Inventory Management:**
  - The system shall allow an admin to add new food and drink items, including their name, price, and initial quantity.
  - The system shall automatically update an item's status to "Out of Stock" (Availability='No') when its quantity reaches 0, enforced by a database trigger.
  - The system shall prevent duplicate food or drink items from being created, instead updating the quantity of the existing item.
- **Order Management:**
  - A user shall be able to create a new order for a selected customer.
  - Users shall be able to add multiple food and drink items to a "cart" for a single order.
  - The system shall prevent a user from adding more items to the cart than are available in stock.
  - When an order is placed, the system shall automatically (via a trigger) reduce the Quantity of the ordered items in the Food and Drink tables.
- **Payment & Delivery:**
  - The system shall automatically (via a trigger) calculate the total Amount for a new Payment record based on the items in the Contains table.
  - A user shall be able to view all past orders on a central dashboard.
  - A user shall be able to assign a driver to an order that has not yet been assigned one.
- **Admin & Relations:**
  - An admin shall be able to link chefs to the food items they prepare.
  - An admin shall be able to assign corporate customers to specific employees.

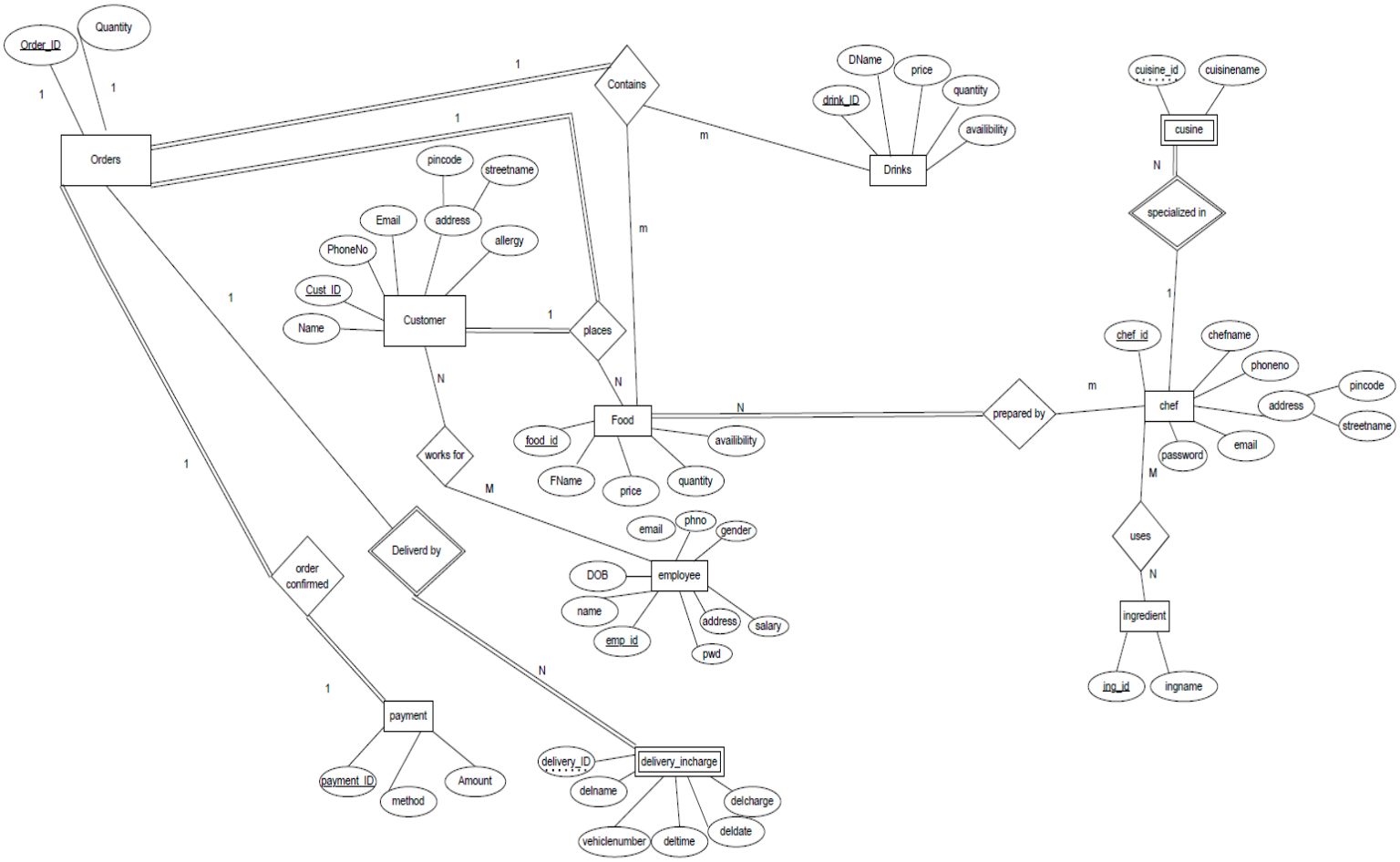
Non-Functional Requirements:

- **Data Integrity:** The database must enforce relationships using foreign keys. Deleting a parent record (like a Customer) should cascade to delete their related records (like Orders). Deleting a menu item (like a Food) should not break sales history.
- **Usability:** The system must have a clean, user-friendly graphical interface (GUI).
- **Concurrency:** The database must be able to handle order placements by using transactions to prevent data corruption.
- **Automation:** The system must use database triggers to handle business logic (stock, payment) automatically.

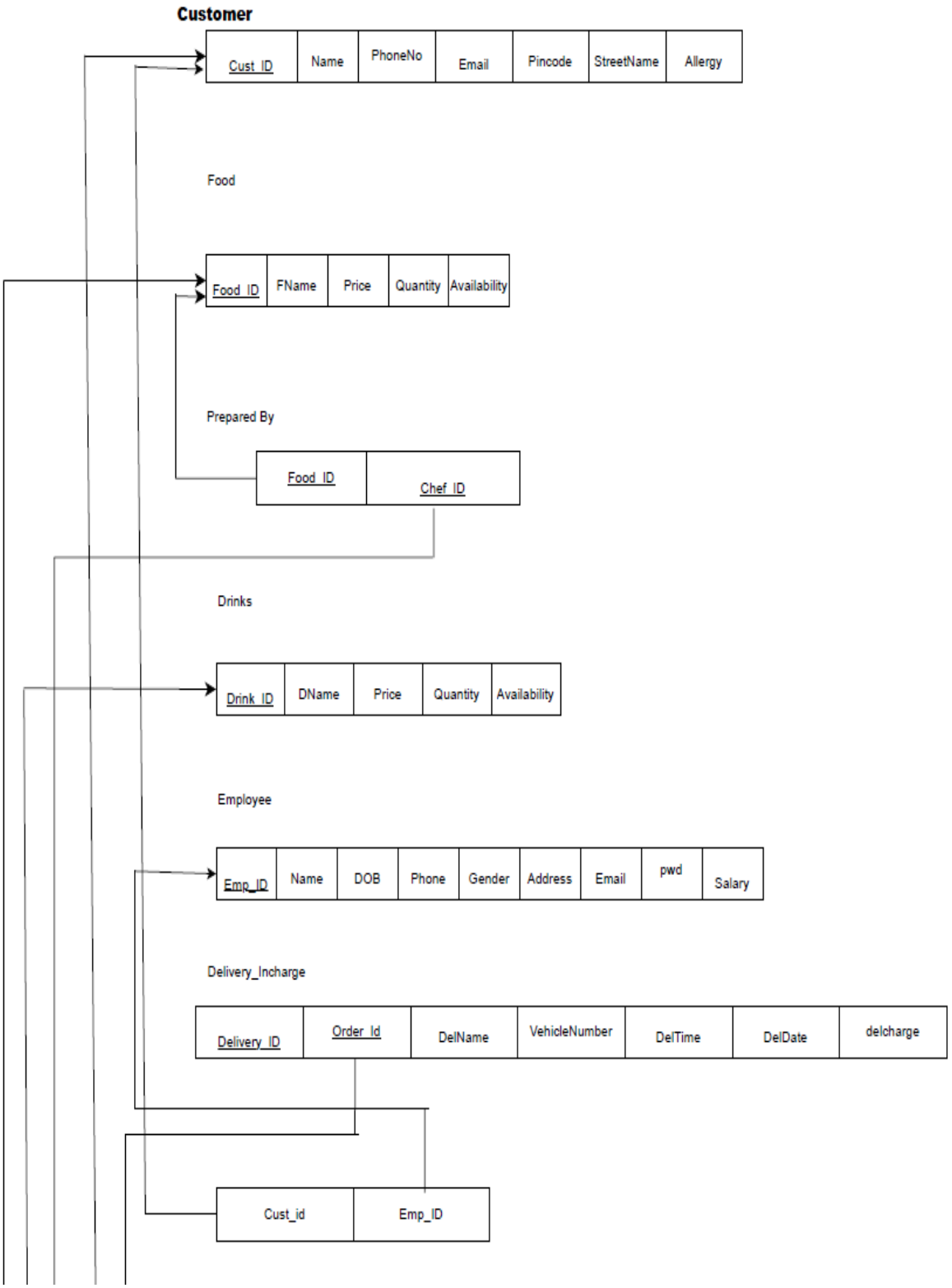
Software, Tools, and Languages

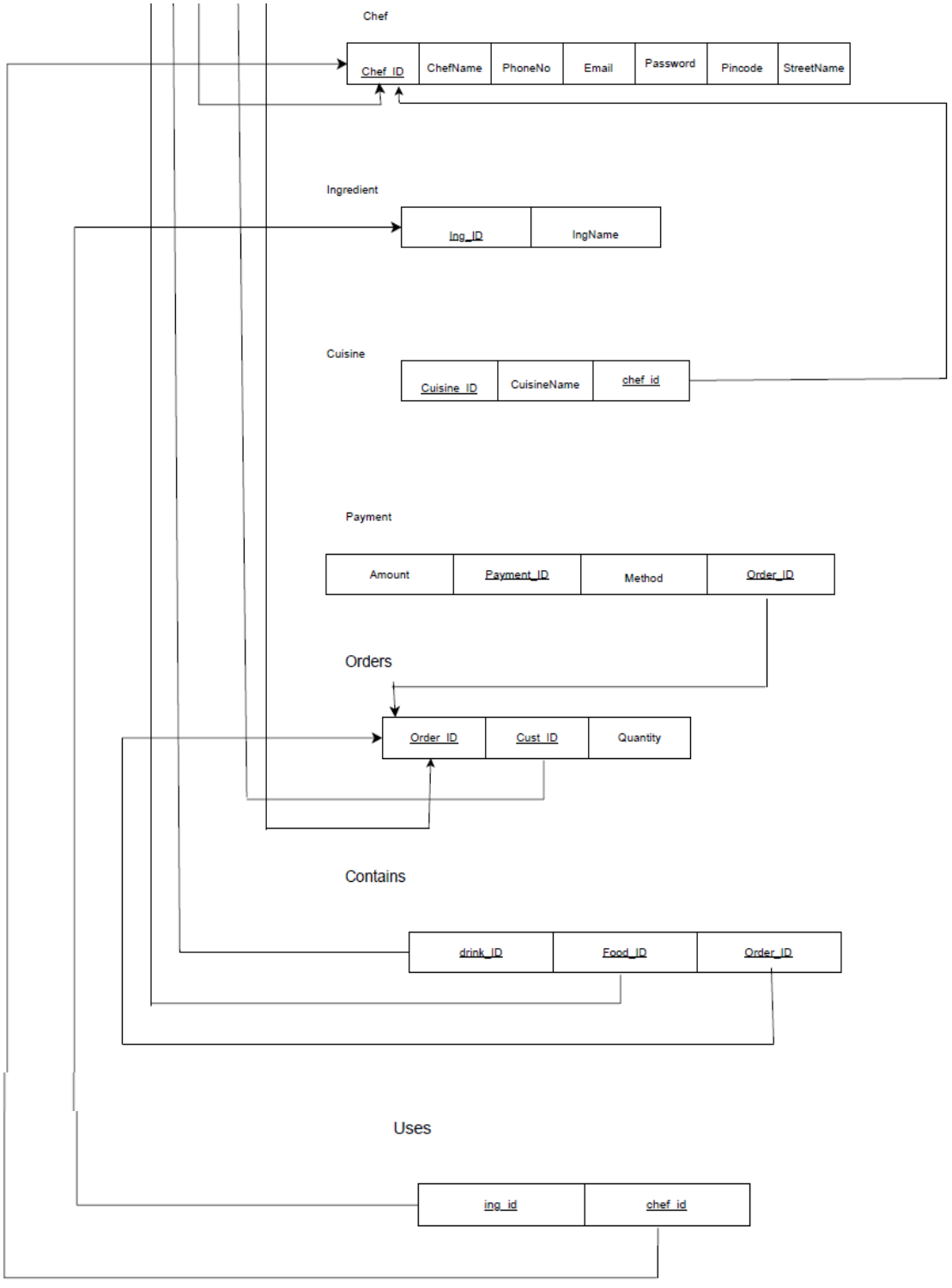
- Programming Language: Python 3
- Database: MySQL Server
- Database Management: MySQL Workbench
- Core Libraries (Python):
    - tkinter (and tkinter.ttk): For the core GUI.
    - mysql-connector-python: For connecting the Python application to the MySQL database.
    - sv_ttk: For modern styling of the Tkinter theme.
    - Pillow (PIL): For handling icons in the GUI.

ER Diagram:


Order_ID
Quantity
Orders
Contains
DName
price
drink_ID
quantity
availibility
Drinks
cuisine_id
cuisinename
cusine
specialized in
pincode
streetname
Email
address
allergy
PhoneNo
Cust_ID
Name
Customer
places
chef_id
chefname
phoneno
pincode
address
streetname
chef
email
password
prepared by
Food
food_id
availibility
FName
price
quantity
works for
Deliverd by
order confirmed
email
phno
gender
DOB
employee
name
address
salary
emp_id
pwd
uses
ingredient
ing_id
ingname
payment
payment_ID
method
Amount
delivery_ID
delivery_incharge
delname
delcharge
deldate
vehiclenumber
deltime

Relational Schema:




Chef
Chef_ID
ChefName
PhoneNo
Email
Password
Pincode
StreetName
Ingredient
Ing_ID
IngName
Cuisine
Cuisine_ID
CuisineName
chef_id
Payment
Amount
Payment_ID
Method
Order_ID
Orders
Order_ID
Cust_ID
Quantity
Contains
drink_ID
Food_ID
Order_ID
Uses
ing_id
chef_id

DDL Commands (Data Definition Language)

```
CREATE TABLE Customer (Cust_ID INT PRIMARY KEY, Cust_Name
VARCHAR(100) NOT NULL,PhoneNo VARCHAR(15) NOT NULL,Email
VARCHAR(100) NOT NULL UNIQUE,
StreetName VARCHAR(100) NOT NULL,Pincode VARCHAR(10) NOT NULL,Allergy
VARCHAR(100));

CREATE TABLE Employee (Emp_ID INT PRIMARY KEY,Emp_Name
VARCHAR(100) NOT NULL,DOB DATE,Email VARCHAR(100) NOT NULL
UNIQUE,
PhoneNo VARCHAR(15)NOT NULL, Gender VARCHAR(10),Address
VARCHAR(200),Salary DECIMAL(10,2),Password VARCHAR(50) NOT NULL
UNIQUE);

CREATE TABLE Chef (Chef_ID INT PRIMARY KEY,Chef_Name
VARCHAR(100)NOT NULL,PhoneNo VARCHAR(15) NOT NULL ,Email
VARCHAR(100) NOT NULL UNIQUE,
StreetName VARCHAR(100) NOT NULL,Pincode VARCHAR(10) NOT
NULL,Password VARCHAR(50)NOT NULL UNIQUE);

CREATE TABLE Cuisine (Cuisine_ID INT,Chef_ID INT,Cuisine_Name
VARCHAR(100),PRIMARY KEY (Cuisine_ID, Chef_ID),
FOREIGN KEY (Chef_ID) REFERENCES Chef(Chef_ID));

CREATE TABLE Ingredient (Ing_ID INT PRIMARY KEY,Ing_Name
VARCHAR(100));

CREATE TABLE Food (Food_ID INT PRIMARY KEY,FName VARCHAR(100),Price
DECIMAL(10,2),Quantity INT,Availability VARCHAR(10));

CREATE TABLE Drink (Drink_ID INT PRIMARY KEY,DName VARCHAR(100),Price
DECIMAL(10,2),Quantity INT,Availability VARCHAR(10));

CREATE TABLE Orders (Order_ID INT PRIMARY KEY,Quantity INT, Cust_ID INT,
FOREIGN KEY (Cust_ID) REFERENCES Customer(Cust_ID));

CREATE TABLE Delivery_Incharge (Delivery_ID INT,Del_Name VARCHAR(100)
NOT NULL,VehicleNumber VARCHAR(20) NOT NULL UNIQUE,
DelCharge DECIMAL(10,2), DelDate DATE,DelTime TIME, Order_ID INT,PRIMARY
KEY(Delivery_ID,Order_ID),
FOREIGN KEY (Order_ID) REFERENCES Orders(Order_ID));
```

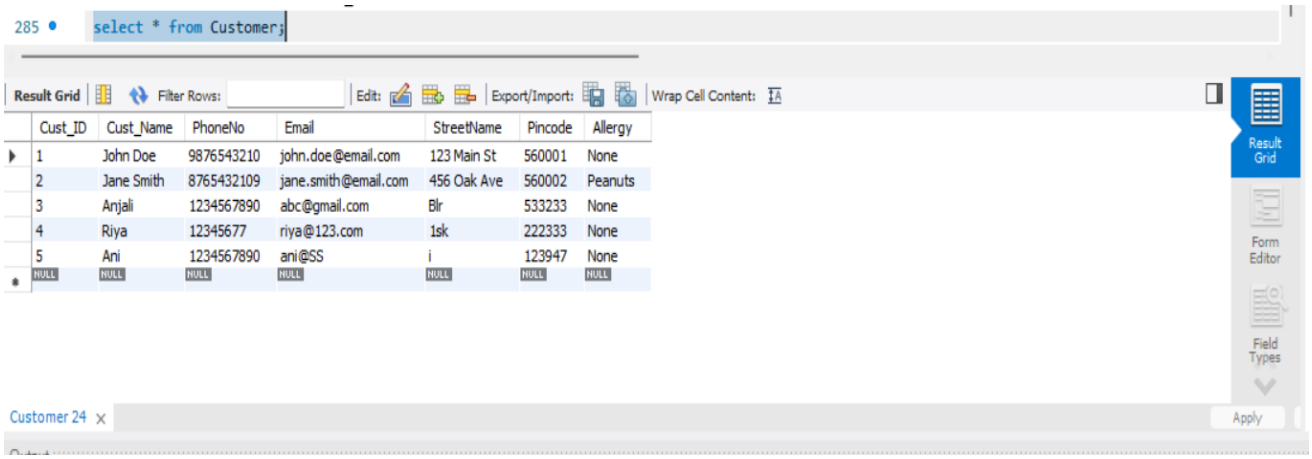
```
CREATE TABLE Payment (Payment_ID INT PRIMARY KEY,Method
VARCHAR(50),Amount DECIMAL(10,2),Order_ID INT UNIQUE,
 FOREIGN KEY (Order_ID) REFERENCES Orders(Order_ID));

CREATE TABLE Prepared_by (Food_ID INT,Chef_ID INT,PRIMARY KEY (Food_ID,
Chef_ID),
FOREIGN KEY (Food_ID) REFERENCES Food(Food_ID),FOREIGN KEY (Chef_ID)
REFERENCES Chef(Chef_ID));

CREATE TABLE Uses (Chef_ID INT,Ing_ID INT,PRIMARY KEY (Chef_ID, Ing_ID),
FOREIGN KEY (Chef_ID) REFERENCES Chef(Chef_ID),FOREIGN KEY (Ing_ID)
REFERENCES Ingredient(Ing_ID));

CREATE TABLE Contains (Order_ID INT,Food_ID INT,Drink_ID INT,PRIMARY
KEY(Order_ID,Drink_ID,Food_ID),
FOREIGN KEY (Order_ID) REFERENCES Orders(Order_ID), FOREIGN KEY
(Food_ID) REFERENCES Food(Food_ID),
FOREIGN KEY (Drink_ID) REFERENCES Drink(Drink_ID));

CREATE TABLE Works_For (Cust_ID INT,Emp_ID INT, PRIMARY
KEY(Cust_ID,Emp_ID),
FOREIGN KEY (Cust_ID) REFERENCES Customer(Cust_ID), FOREIGN KEY
(Emp_ID) REFERENCES Employee(Emp_ID));
```

CRUD operation Screenshots

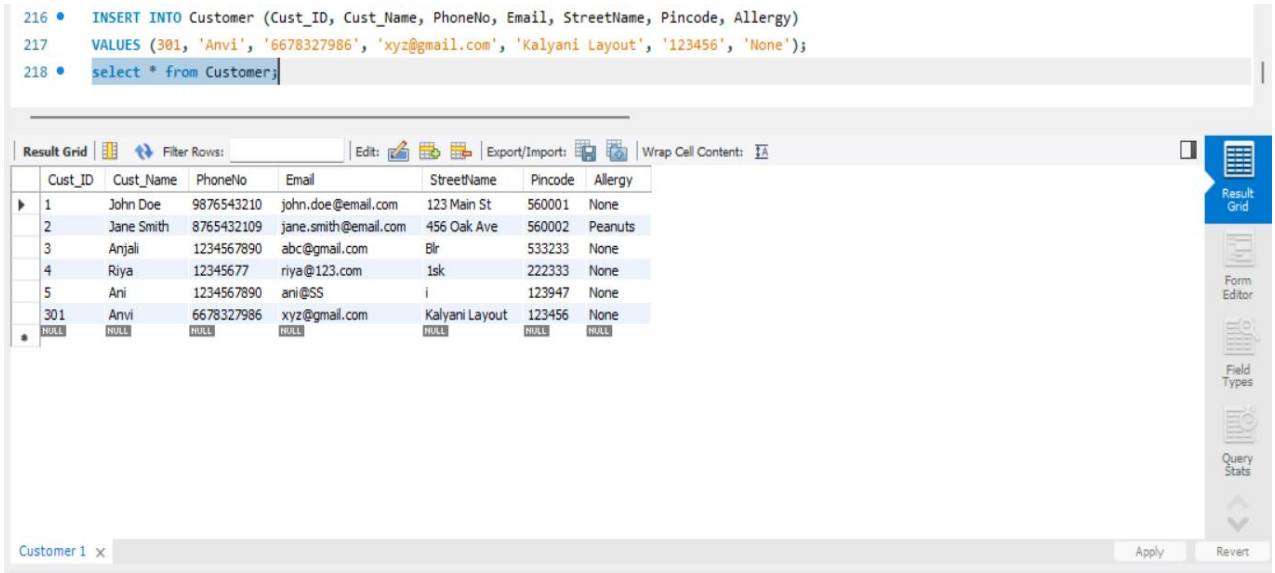
1. Add New Customer (Create Operation)

Description: The “Add New Customer” form allows the admin to create a new customer record by entering details such as Name, Phone, Email, Street, Pincode, and Allergy. On clicking “Add Customer”, the application executes an SQL INSERT query into the Customer table.

Before:



After:



2. Add New Food Item (Create + Trigger)

Purpose: Demonstrates creation of a new food record; trigger automatically sets Availability = 'Yes' because quantity > 0.

Before:

```
227 •  SELECT * FROM Food ;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Food_ID	FName	Price	Quantity	Availability
0	No Food Item	0.00	999	Yes
301	Margherita Pizza	350.00	17	Yes
302	Chicken Curry	450.00	11	Yes
303	Fish Fry	12.00	9	No
304	Mango Juice	100.00	8	Yes
305	Mango Juice	100.00	1	Yes
501	Margherita Pizza	250.00	11	Yes
502	Onion Pizza	250.00	12	Yes
NULL	NULL	NULL	NULL	NULL

Result Grid | Form Editor | Field Types | Food 25 | Apply | Conte

After:

```
220
221 •  INSERT INTO Food (Food_ID, FName, Price, Quantity, Availability)
222    VALUES (503, 'Paneer Pizza', 250.00, 12, 'Yes');
223 •  SELECT * FROM Food ;
224
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Food_ID	FName	Price	Quantity	Availability
0	No Food Item	0.00	999	Yes
301	Margherita Pizza	350.00	17	Yes
302	Chicken Curry	450.00	11	Yes
303	Fish Fry	12.00	9	No
304	Mango Juice	100.00	8	Yes
305	Mango Juice	100.00	1	Yes
501	Margherita Pizza	250.00	12	Yes
502	Onion Pizza	250.00	12	Yes
503	Paneer Pizza	250.00	12	Yes
NULL	NULL	NULL	NULL	NULL

Result Grid | Form Editor | Field Types | Food 4 | Apply | Revert

3. Add / Update Drink Item (Update Operation)

Before:

```
228 •  SELECT * FROM Drink;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Drink_ID	DName	Price	Quantity	Availability
0	No Drink Item	0.00	999	Yes
401	Coke	50.00	94	Yes
402	Lemonade	70.00	49	No
NULL	NULL	NULL	NULL	NULL

Result Grid | Form Editor | Field Types | Drink 5 | Apply | Revert

After:

```
226 •  UPDATE Drink
227    SET Price = 60.00, Quantity = Quantity + 10
228    WHERE DName = 'Coke';
229 •  SELECT * FROM Drink;
230
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Drink_ID	DName	Price	Quantity	Availability
0	No Drink Item	0.00	999	Yes
401	Coke	60.00	104	Yes
402	Lemonade	70.00	49	No
NULL	NULL	NULL	NULL	NULL

Drink 6 | Apply | Revert

4. Place New Order (Create + Automated Triggers)

Purpose: Demonstrates automatic business logic.

When a new order is placed, triggers:

- reduce item quantity in Food / Drink
- compute total Amount in Payment

Before:

Orders table:

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Order_ID	Quantity	Cust_ID
1001	2	1
1002	1	2
1004	1	1
1005	1	1
1007	2	2
1008	1	2
1020	1	2
1021	1	2
1022	2	3
1023	1	4
NULL	NULL	NULL

Orders 11 | Apply | Revert

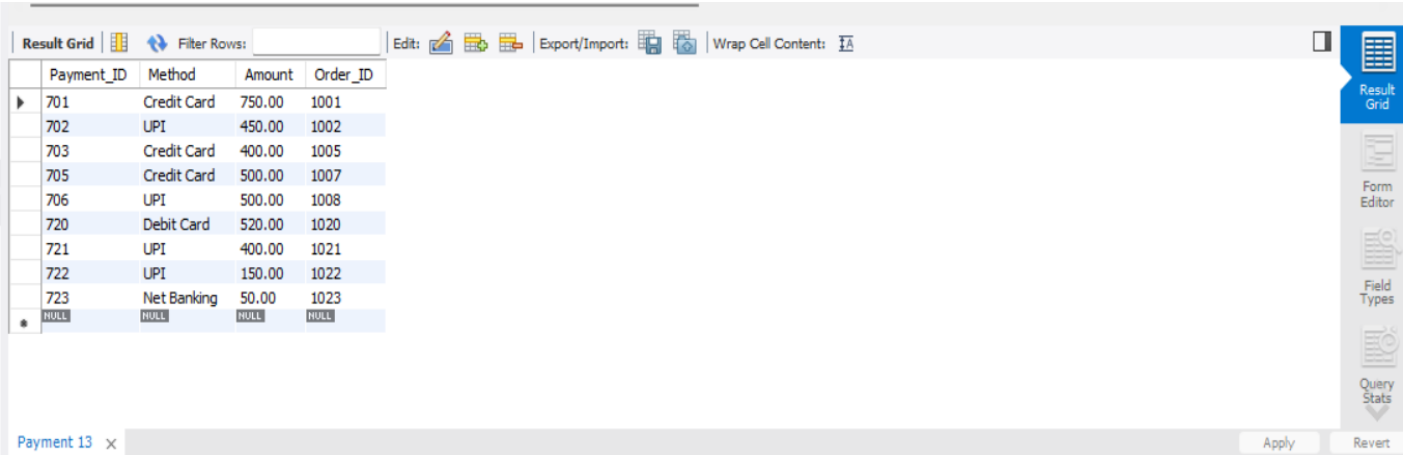
Contains table:

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Order_ID	Food_ID	Drink_ID
1023	0	401
1001	301	401
1002	301	0
1004	301	0
1005	301	401
1021	301	401
1001	302	0
1002	302	401
1007	302	401
1008	302	401
1020	302	402
1022	305	0
NULL	NULL	NULL

Contains 12 | Apply | Revert

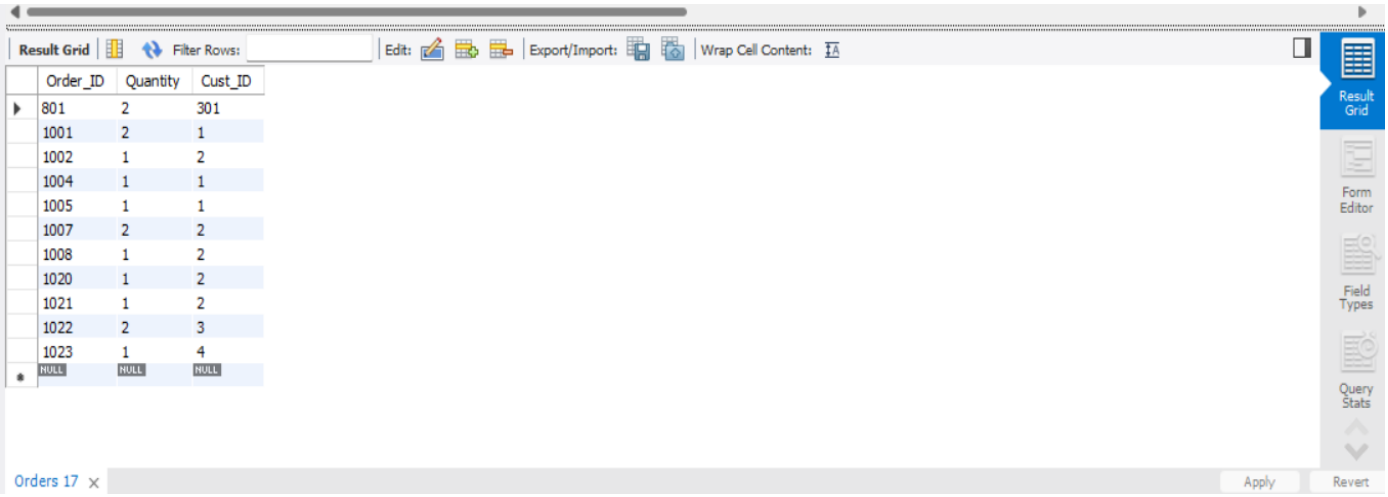
Payment table:



Payment_ID	Method	Amount	Order_ID
701	Credit Card	750.00	1001
702	UPI	450.00	1002
703	Credit Card	400.00	1005
705	Credit Card	500.00	1007
706	UPI	500.00	1008
720	Debit Card	520.00	1020
721	UPI	400.00	1021
722	UPI	150.00	1022
723	Net Banking	50.00	1023
NULL	NULL	NULL	NULL

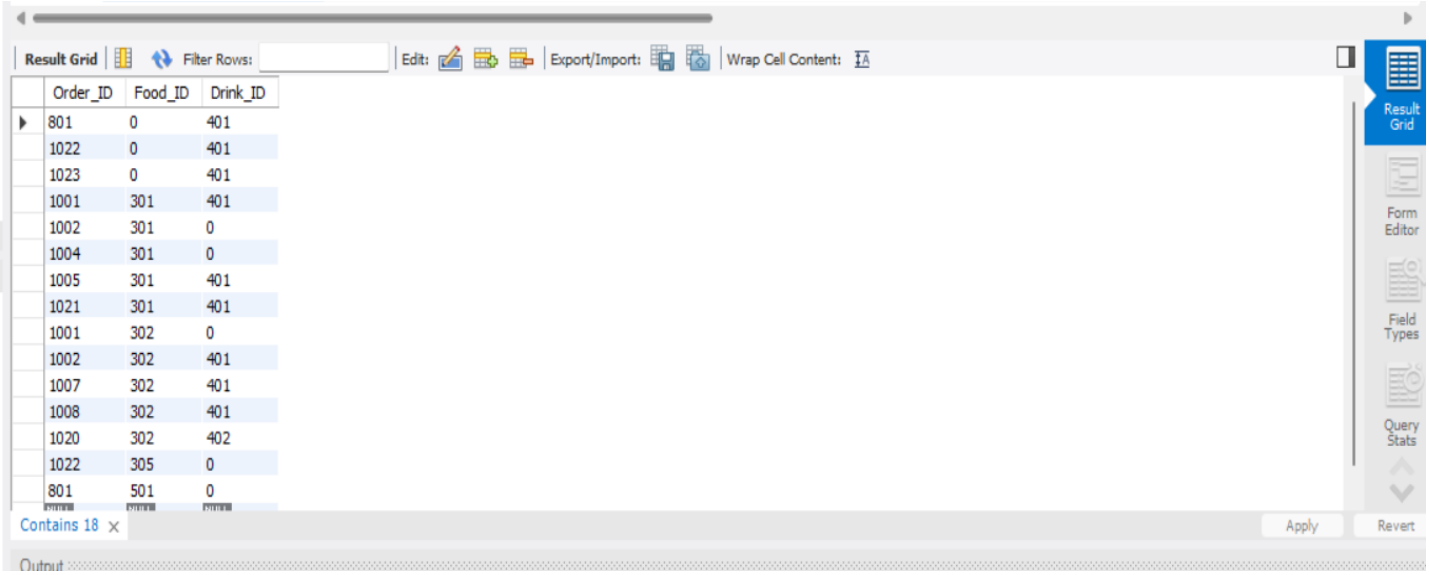
After:

Orders table:



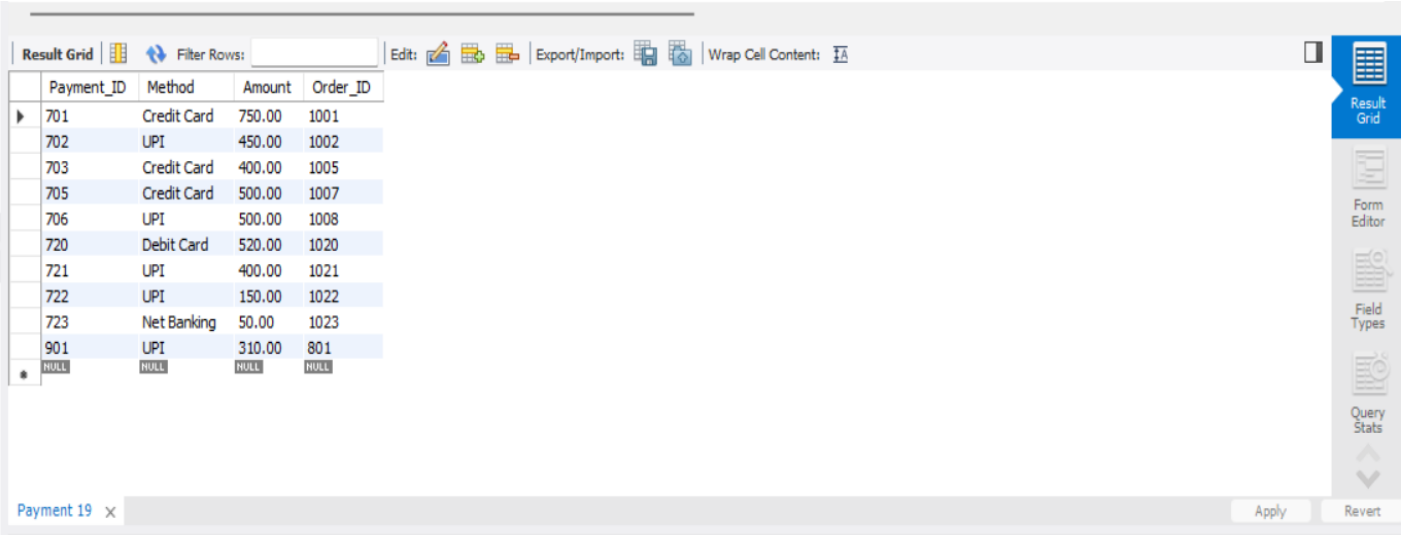
Order_ID	Quantity	Cust_ID
801	2	301
1001	2	1
1002	1	2
1004	1	1
1005	1	1
1007	2	2
1008	1	2
1020	1	2
1021	1	2
1022	2	3
1023	1	4
NULL	NULL	NULL

Contains table:



Order_ID	Food_ID	Drink_ID
801	0	401
1022	0	401
1023	0	401
1001	301	401
1002	301	0
1004	301	0
1005	301	401
1021	301	401
1001	302	0
1002	302	401
1007	302	401
1008	302	401
1020	302	402
1022	305	0
801	501	0

Payments table:



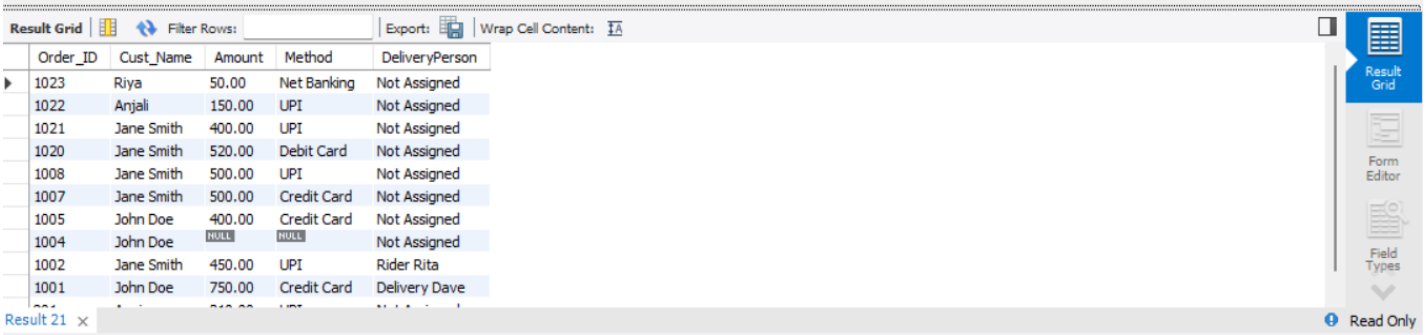
Payment_ID	Method	Amount	Order_ID
701	Credit Card	750.00	1001
702	UPI	450.00	1002
703	Credit Card	400.00	1005
705	Credit Card	500.00	1007
706	UPI	500.00	1008
720	Debit Card	520.00	1020
721	UPI	400.00	1021
722	UPI	150.00	1022
723	Net Banking	50.00	1023
901	UPI	310.00	801
NULL	NULL	NULL	NULL

This SQL script demonstrates the Create (Order) operation.It inserts a new order for customer 301, links the selected food (501) and drink (401) items in the Contains table, and automatically generates a payment entry via database triggers.After execution, the Orders, Contains, and Payment tables show the new Order ID 801 with updated stock and total amount.

5. View Orders Dashboard (Read Operation)

Purpose: Displays all orders with customer, payment, and delivery details using JOINs.

```
253 •  SELECT o.Order_ID,
254           c.Cust_Name,
255           p.Amount,
256           p.Method,
257           COALESCE(d.Del_Name,'Not Assigned') AS DeliveryPerson
258    FROM Orders o
259    JOIN Customer c ON o.Cust_ID = c.Cust_ID
260    LEFT JOIN Payment p ON o.Order_ID = p.Order_ID
261    LEFT JOIN Delivery_Incharge d ON o.Order_ID = d.Order_ID
262    ORDER BY o.Order_ID DESC;
```

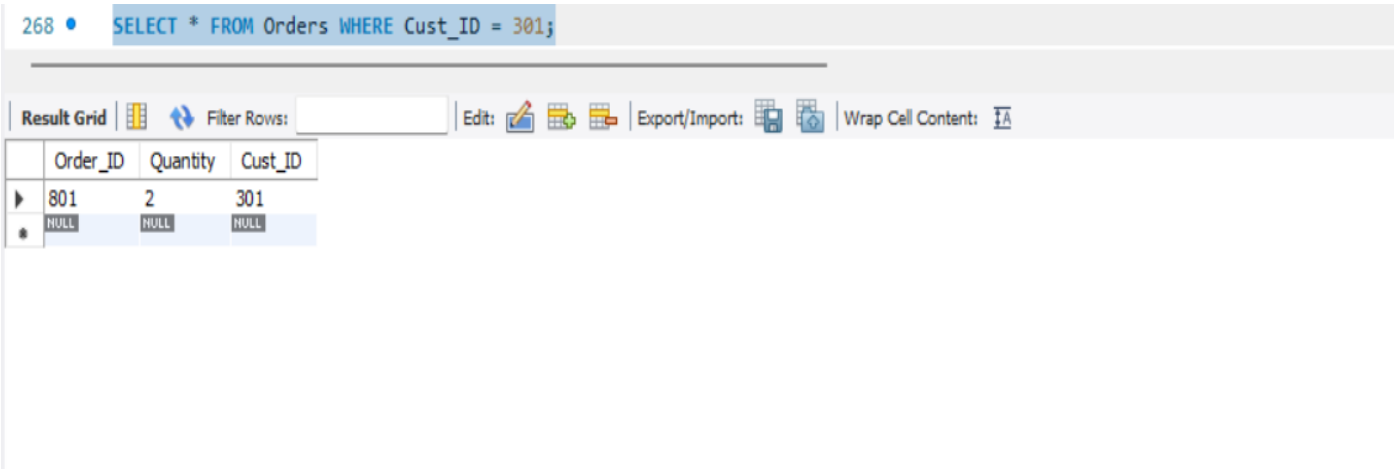


Order_ID	Cust_Name	Amount	Method	DeliveryPerson
1023	Riya	50.00	Net Banking	Not Assigned
1022	Anjali	150.00	UPI	Not Assigned
1021	Jane Smith	400.00	UPI	Not Assigned
1020	Jane Smith	520.00	Debit Card	Not Assigned
1008	Jane Smith	500.00	UPI	Not Assigned
1007	Jane Smith	500.00	Credit Card	Not Assigned
1005	John Doe	400.00	Credit Card	Not Assigned
1004	John Doe	NULL	NULL	Not Assigned
1002	Jane Smith	450.00	UPI	Rider Rita
1001	John Doe	750.00	Credit Card	Delivery Dave

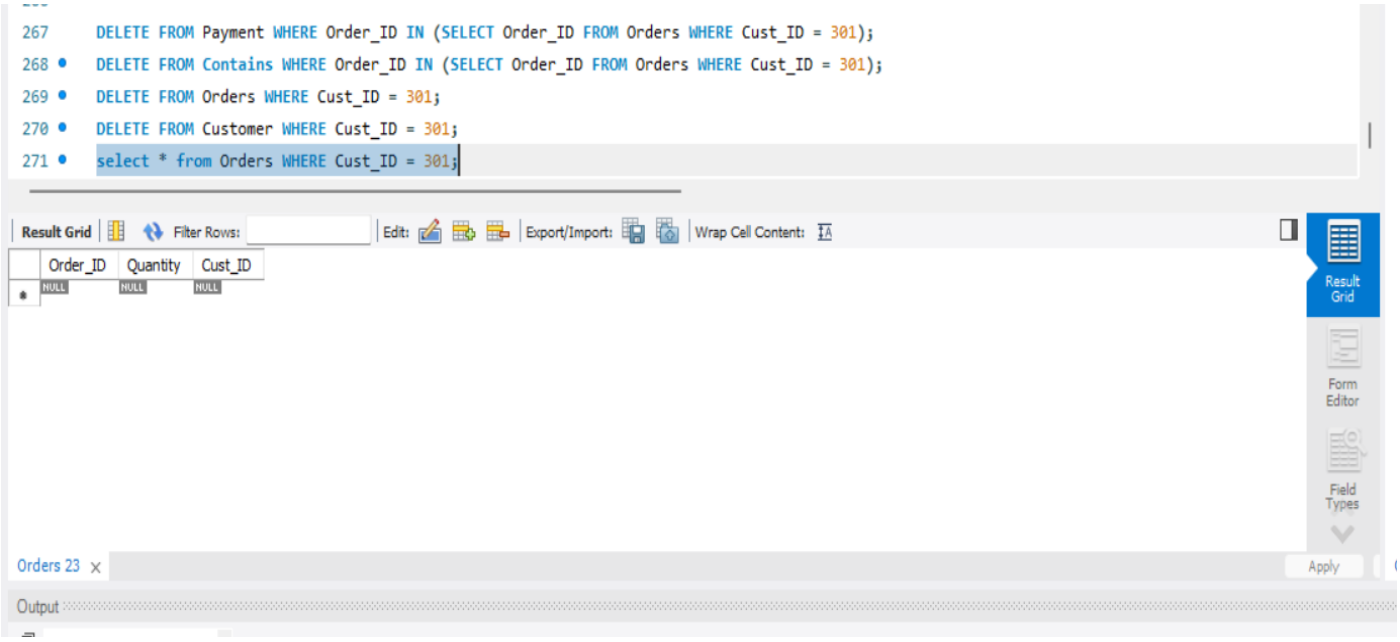
6. Delete Record (Delete Operation)

Purpose: Demonstrates deletion and cascading effect.
Deleting a customer automatically removes related orders and payments due to foreign-key constraints.

Before:

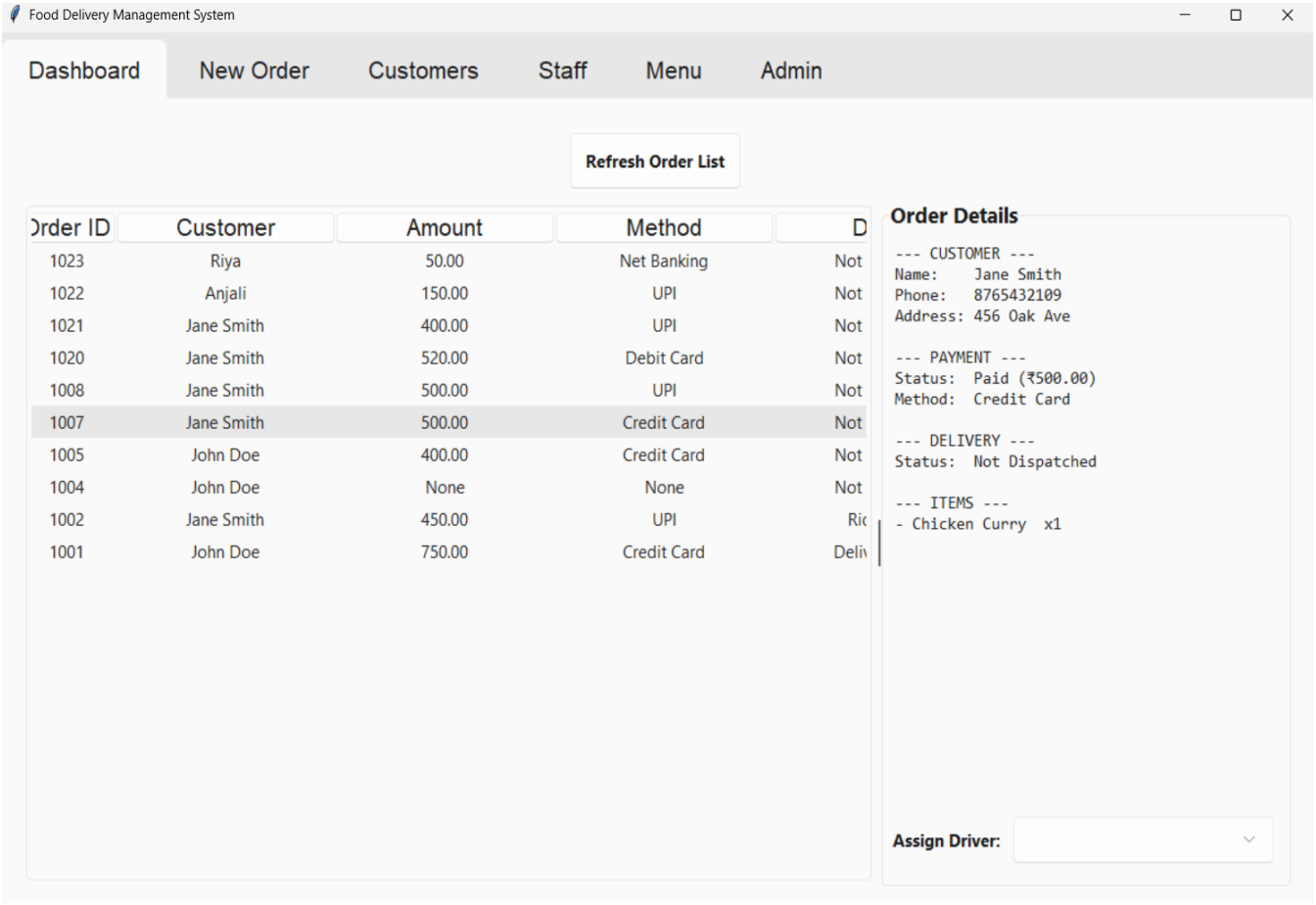


After:



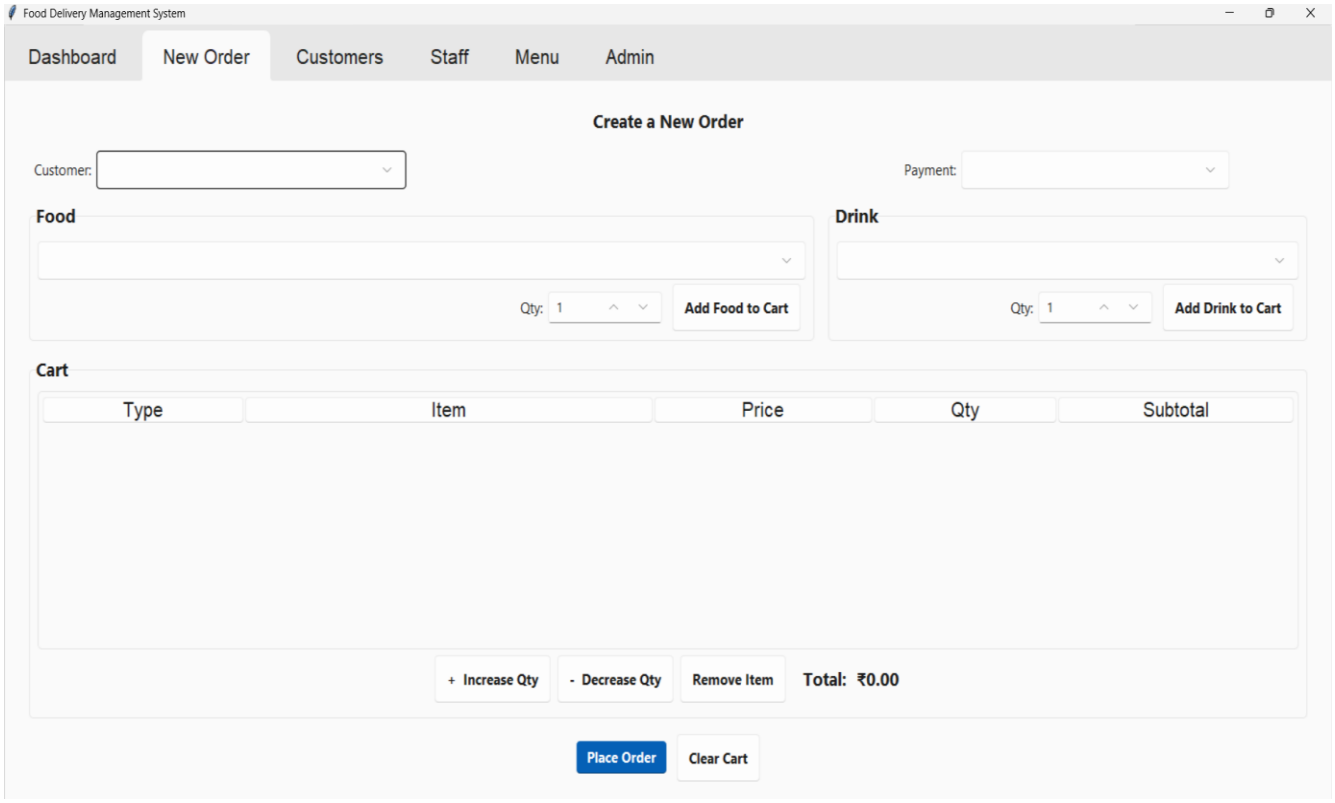
Application Features and UI Showcase

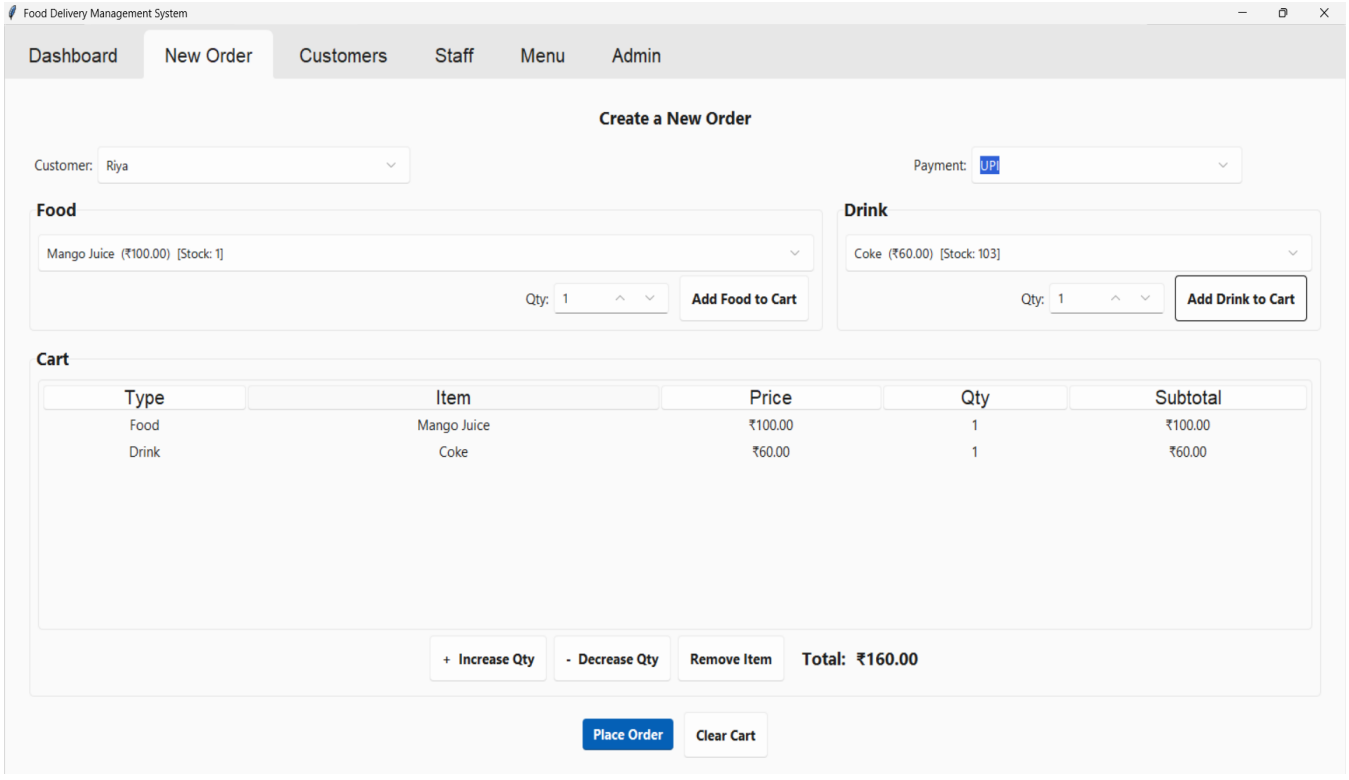
1. Dashboard – Displays all orders, their payment method, total amount, and assigned delivery driver.



The Dashboard serves as the central control panel of the Food Delivery Management System. It displays all existing orders along with their customer names, payment methods, total amounts, and assigned delivery drivers.
Users can select any order to view detailed information such as customer contact, delivery status, and ordered items.
This interface allows quick monitoring and management of all ongoing and completed deliveries in real time.

2. Creating a new order by adding food and drink items to the cart. Before placing an order:



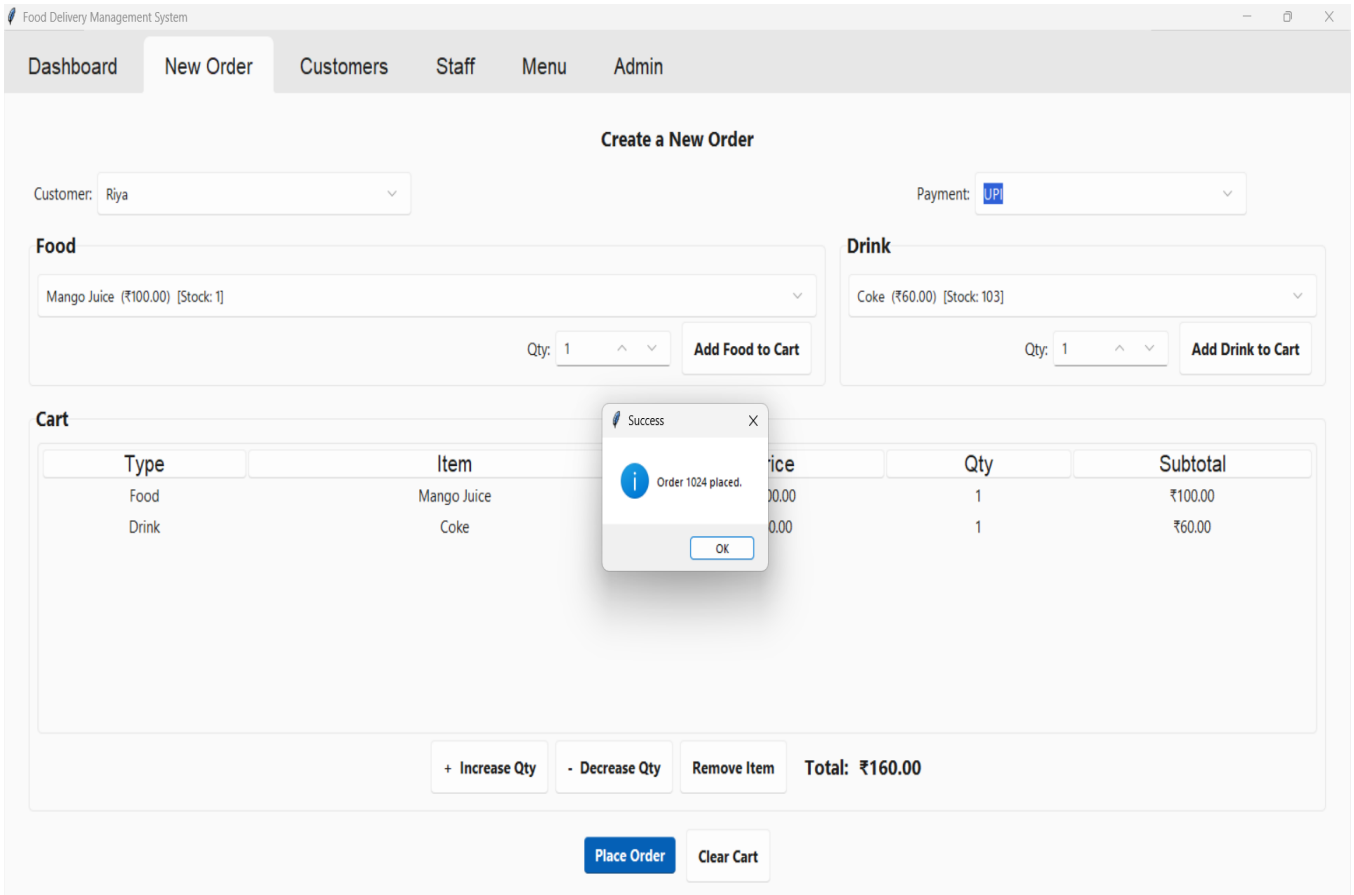


The New Order tab allows users to create a new order for any registered customer. Customers are selected from a dropdown list, and available food and drink items (along with their price and stock quantity) are displayed dynamically from the database.

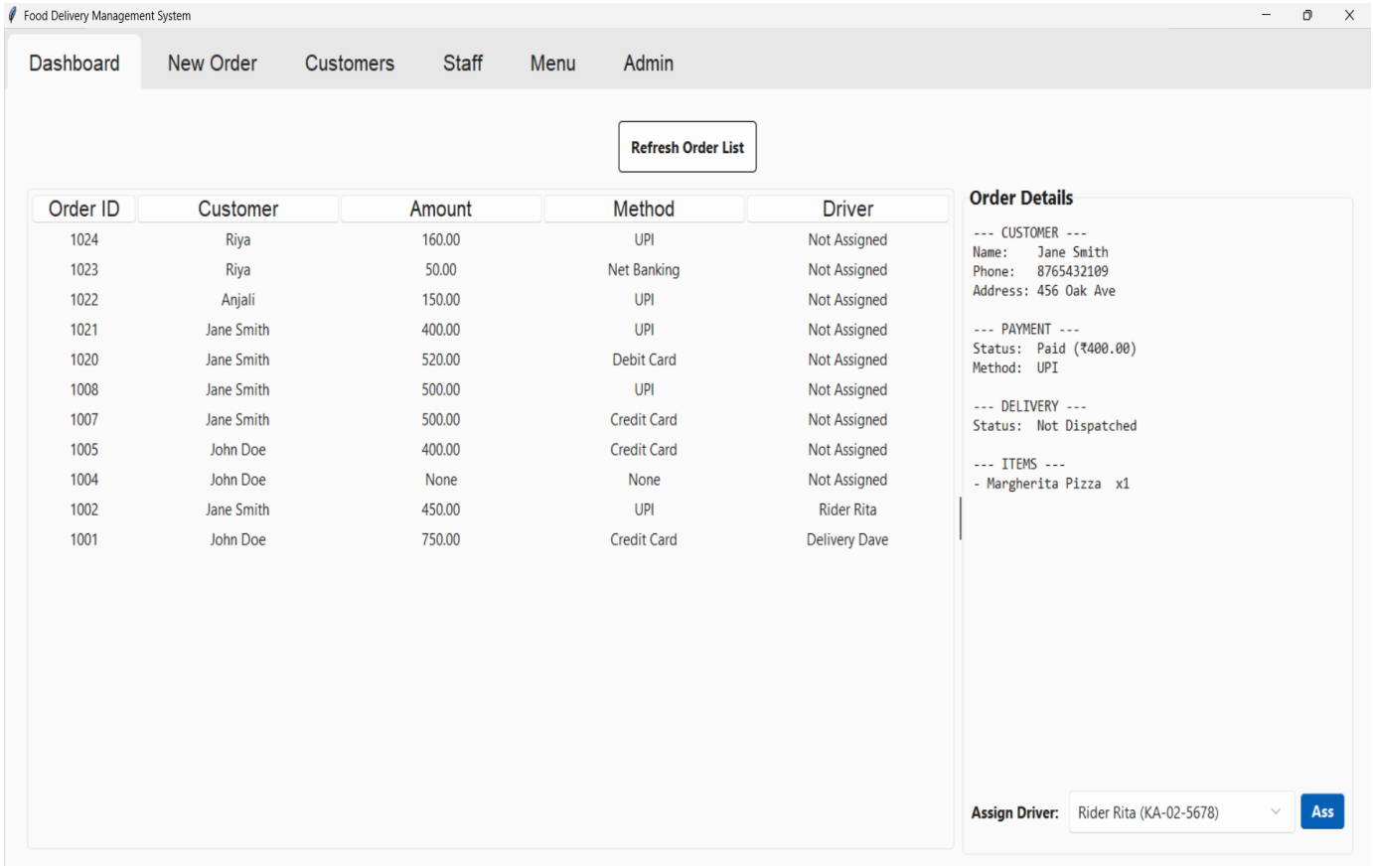
Users can add multiple items to the cart with chosen quantities, modify or remove them, and the total order amount is automatically calculated in real time.

Once the cart is finalized, the “Place Order” button inserts the order details into the database the Orders, Contains, and Payment tables are updated.
Backend triggers then automatically deduct stock quantities from the Food and Drink tables and compute the total payment amount, ensuring accurate and consistent data without manual input.

3. **Order placed successfully — backend triggers automatically update stock and calculate payment.**



New order update added to the dashboard:



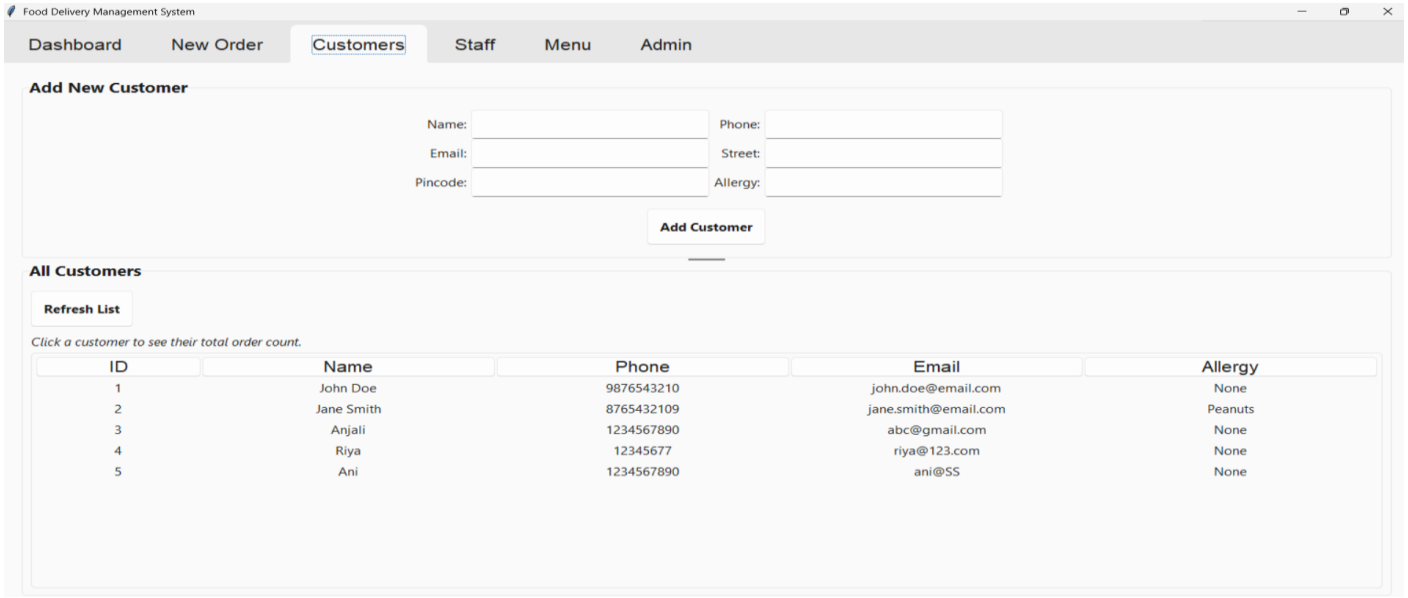
When the user places an order, the system automatically records it in the database and initiates multiple backend operations through triggers.

These triggers deduct the ordered quantities from the Food and Drink tables, update item availability status if stock reaches zero, and calculate the total payable amount in the Payment table.

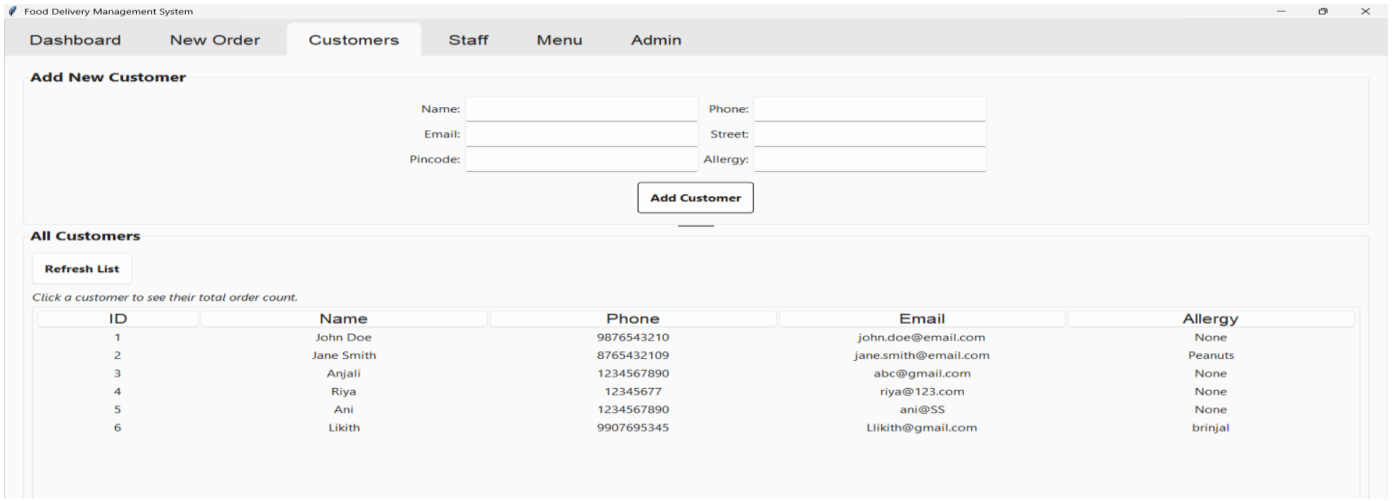
This automation ensures accuracy, eliminates manual calculation errors, and maintains data integrity across all related tables.

4. Adding a new customer and viewing the customer list.

Before adding a new Customer:



After adding customer



The Customers tab enables the admin to register new customers by entering their details such as name, phone number, email, address, pincode, and allergy information.

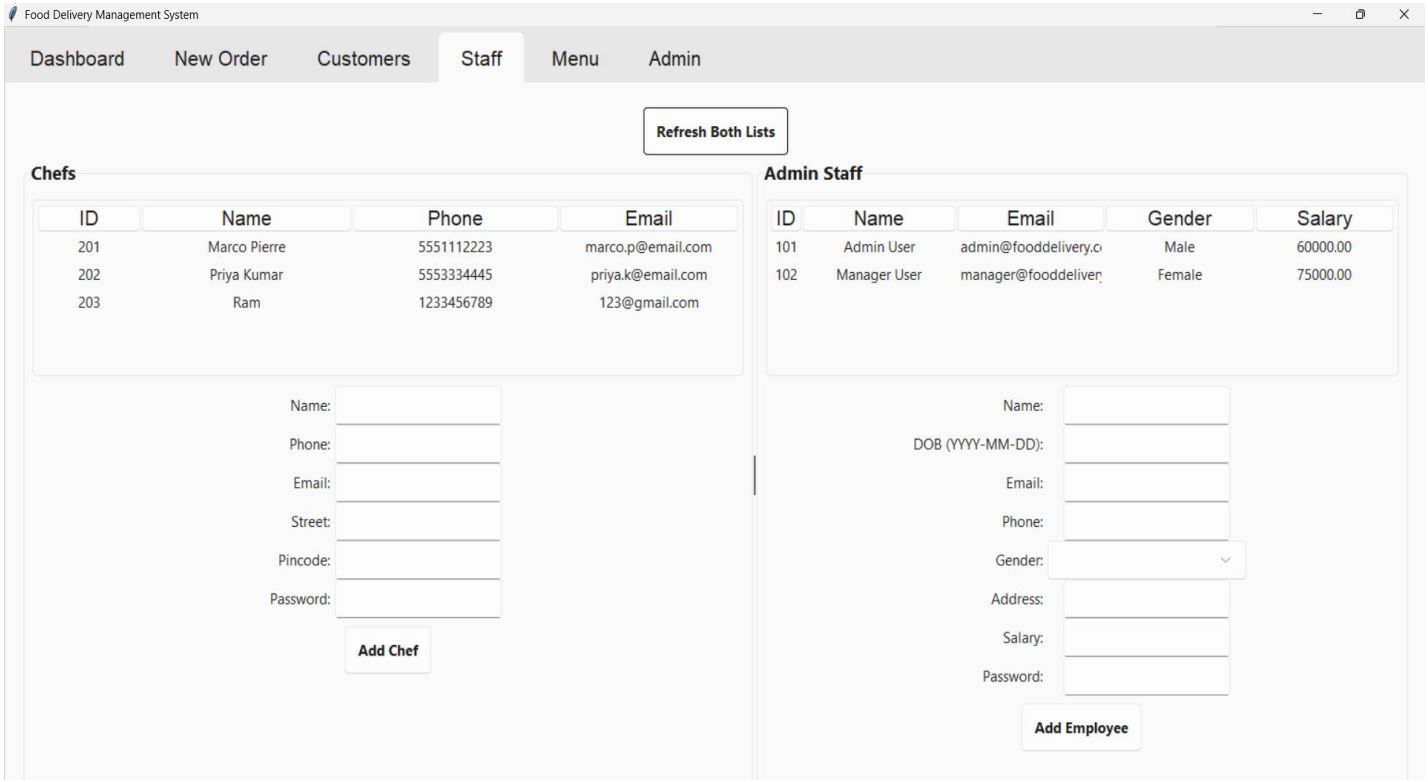
Once the “Add Customer” button is clicked, the details are validated and inserted into the Customer table in the database.

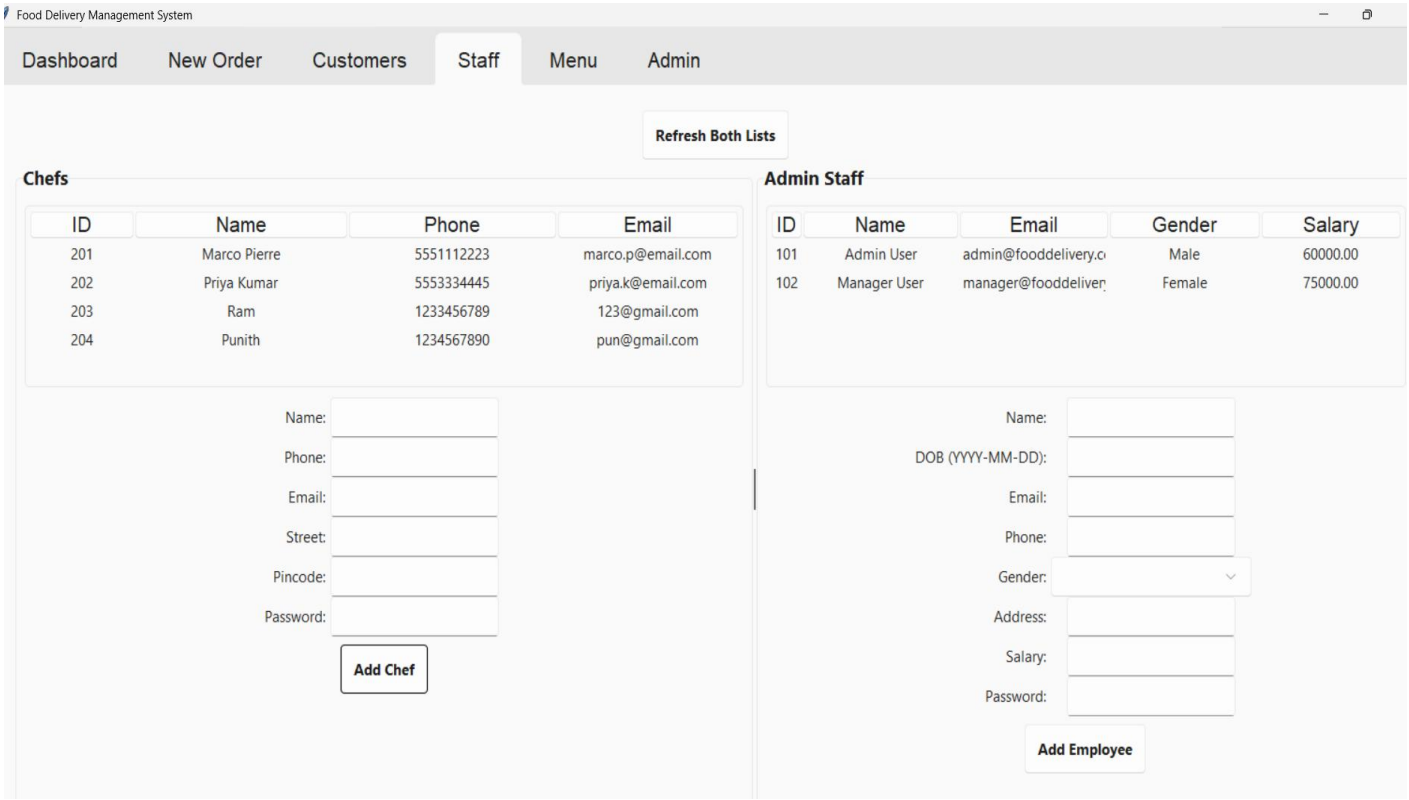
The system then refreshes the customer list in real time, displaying all existing customer records in a table view.

When a customer is selected from the list, their total number of orders is automatically retrieved using the GetCustomerOrderCount function, providing quick insights into their activity.

5. Staff management — adding new chefs and employees

Before adding Staffs:

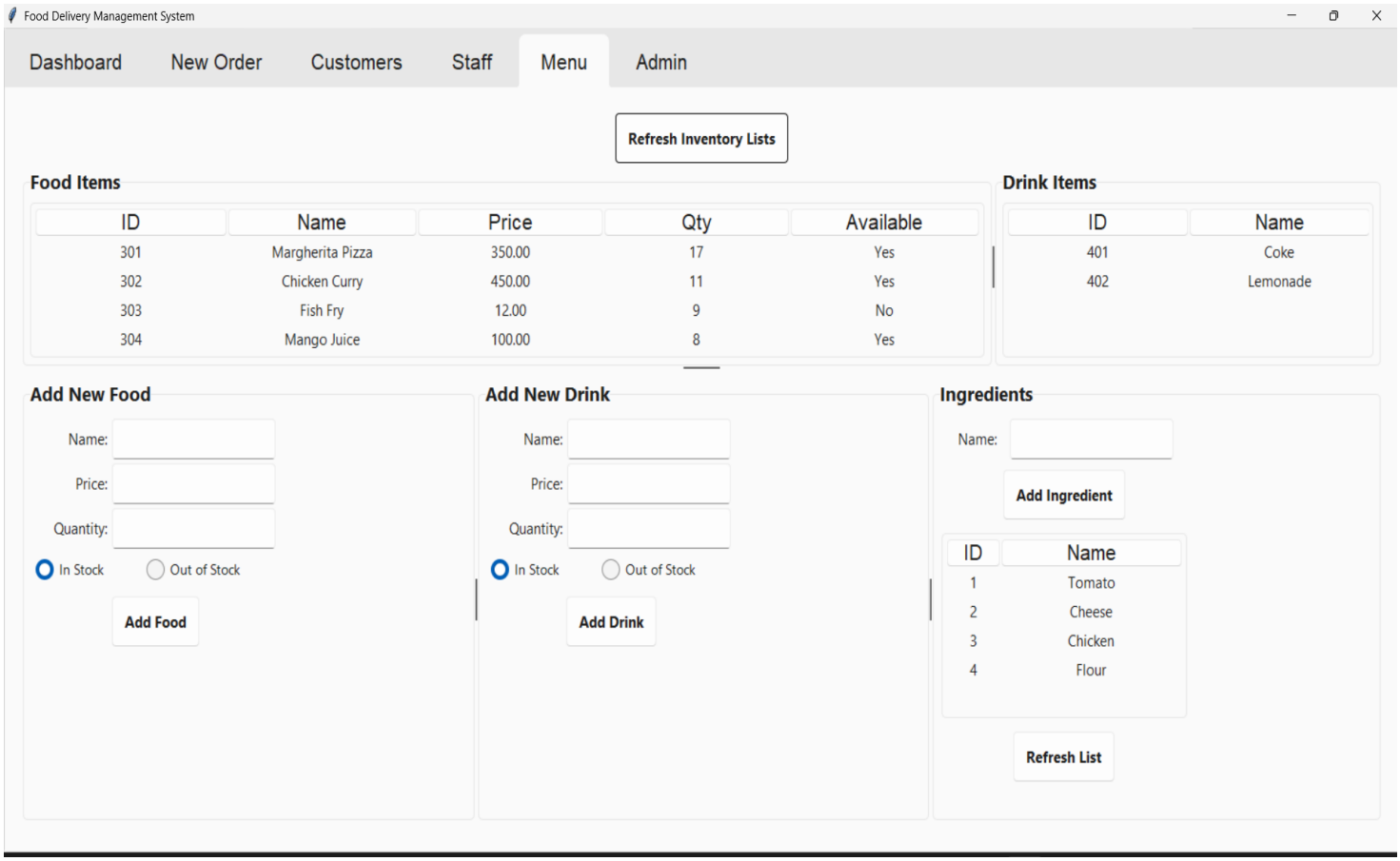


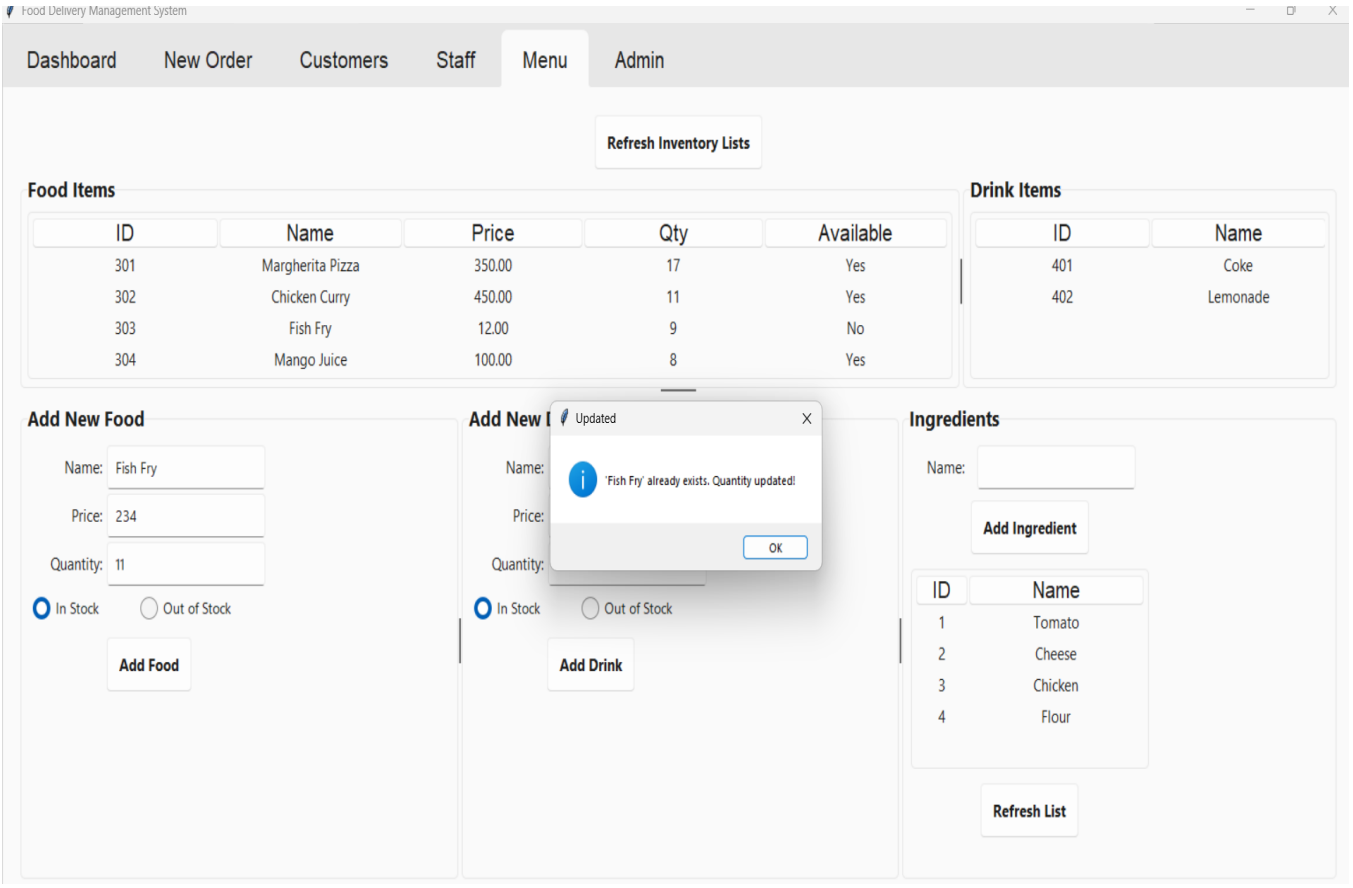


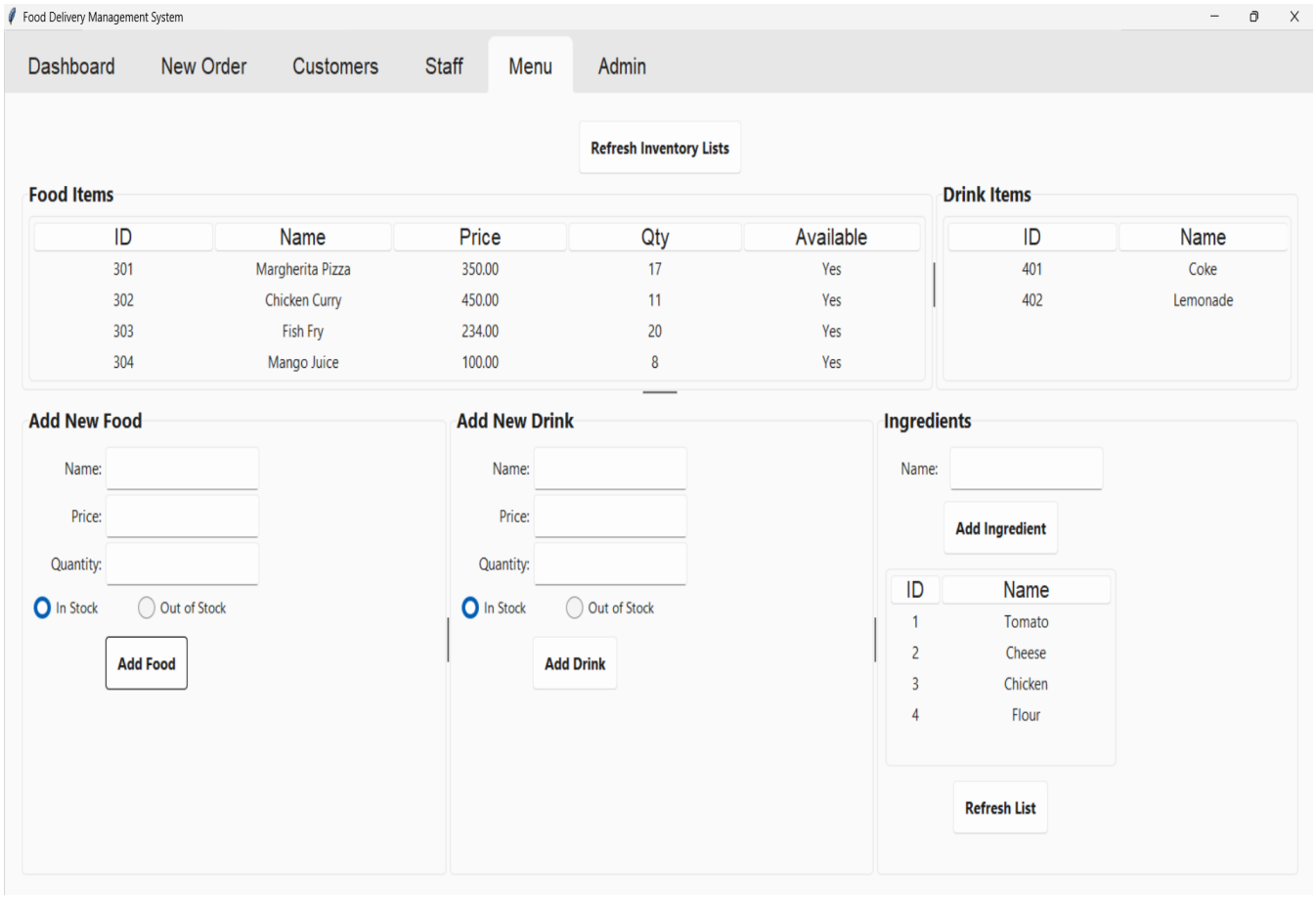
The Staff tab is designed to manage both chefs and administrative employees within the system.
Before adding a new staff member, the interface displays the current list of chefs and employees fetched directly from the Chef and Employee tables.
Each section allows entry of essential details such as name, contact information, email, address, date of birth, salary, and password.
When the “Add Chef” or “Add Employee” button is clicked, the system automatically generates a unique ID and inserts the record into the respective table.
After successful insertion, the lists refresh instantly, reflecting the newly added staff member and ensuring up-to-date workforce information.

6. Managing the food and drink inventory with stock updates and availability toggling.

Before updating availability of Food Item

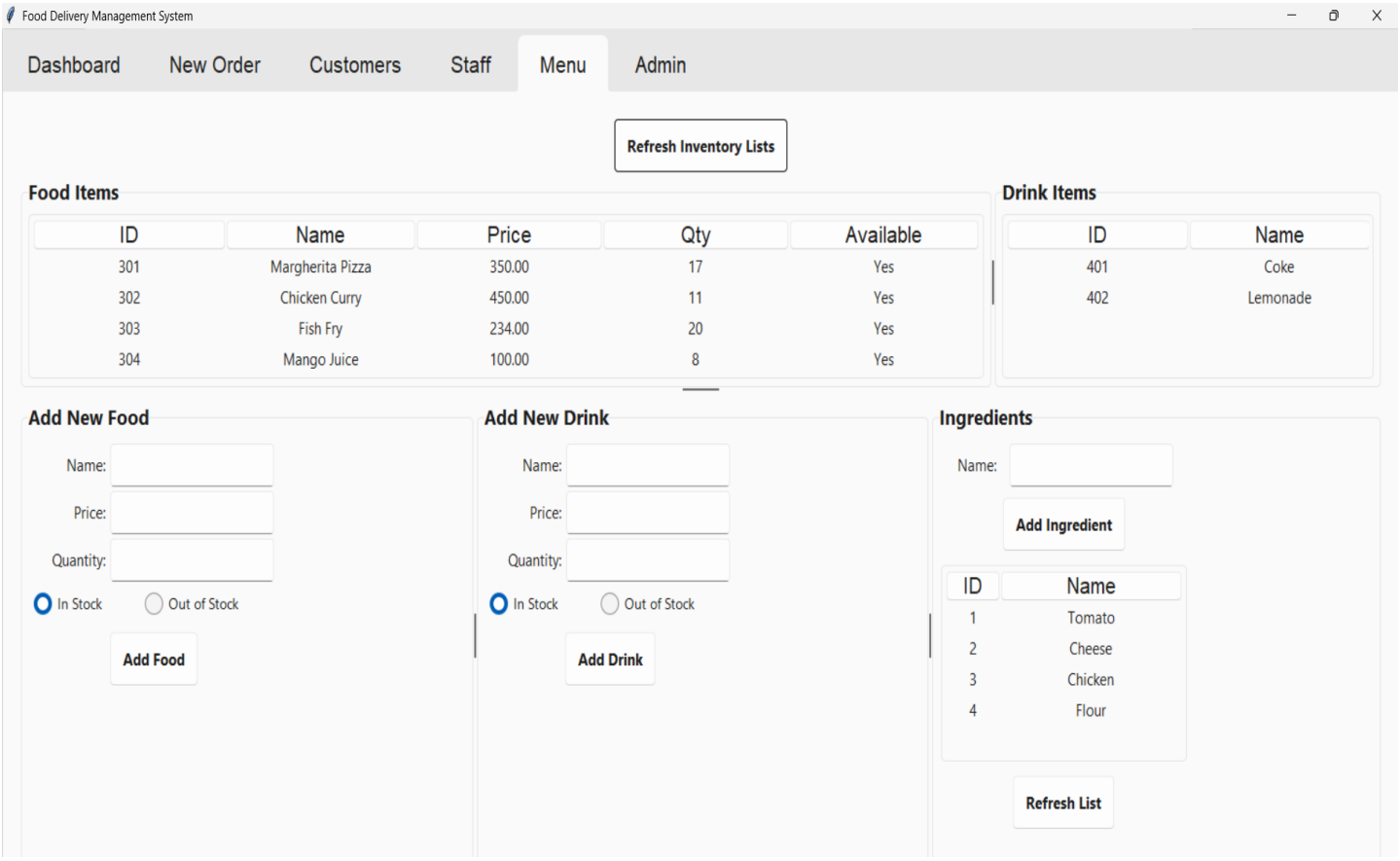


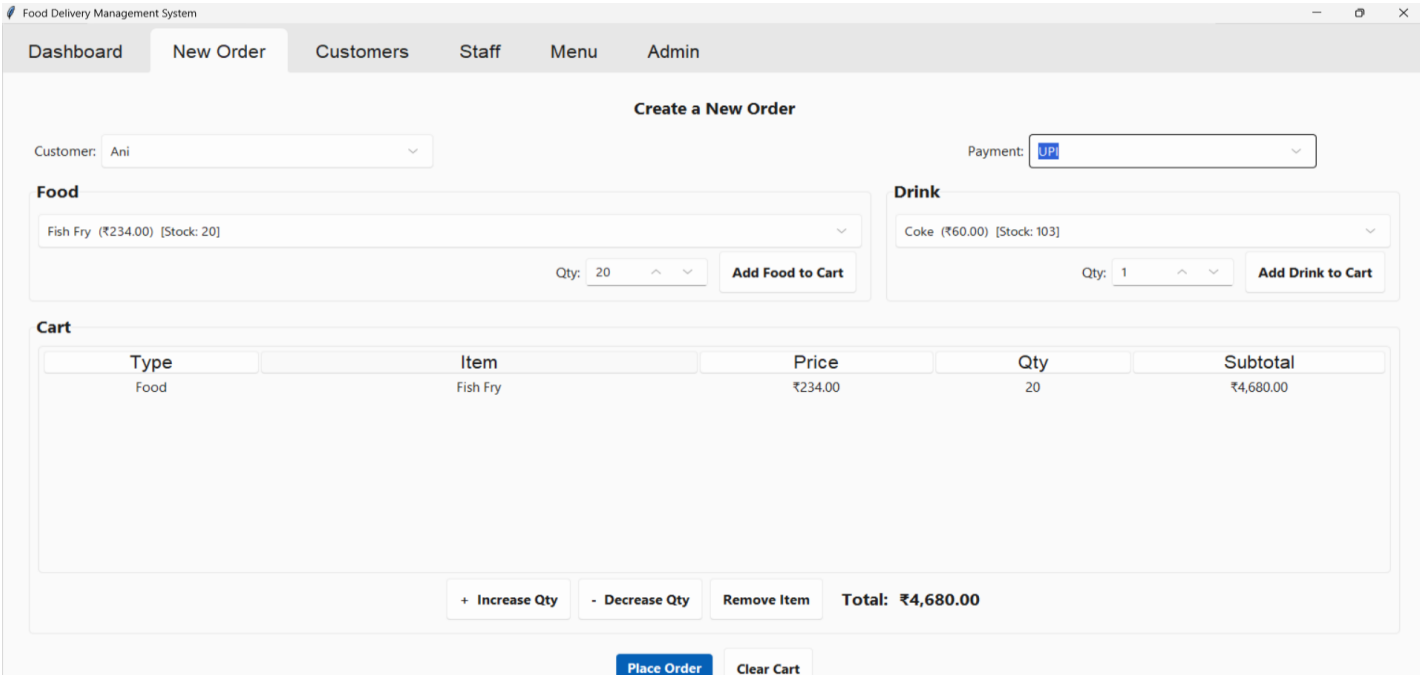

Food Delivery Management System
Dashboard
New Order
Customers
Staff
Menu
Admin
Refresh Inventory Lists
Food Items
ID
Name
Price
Qty
Available
301
Margherita Pizza
350.00
17
Yes
302
Chicken Curry
450.00
11
Yes
303
Fish Fry
12.00
9
No
304
Mango Juice
100.00
8
Yes
Drink Items
ID
Name
401
Coke
402
Lemonade
Add New Food
Name:
Fish Fry
Price:
234
Quantity:
11
In Stock
Out of Stock
Add Food
Add New D
Updated
'Fish Fry' already exists. Quantity updated!
OK
Name:
Price:
Quantity:
In Stock
Out of Stock
Add Drink
Ingredients
Name:
Add Ingredient
ID
Name
1
Tomato
2
Cheese
3
Chicken
4
Flour
Refresh List


Food Delivery Management System
Dashboard
New Order
Customers
Staff
Menu
Admin
Refresh Inventory Lists
Food Items
ID
Name
Price
Qty
Available
301
Margherita Pizza
350.00
17
Yes
302
Chicken Curry
450.00
11
Yes
303
Fish Fry
234.00
20
Yes
304
Mango Juice
100.00
8
Yes
Drink Items
ID
Name
401
Coke
402
Lemonade
Add New Food
Name:
Price:
Quantity:
In Stock
Out of Stock
Add Food
Add New Drink
Name:
Price:
Quantity:
In Stock
Out of Stock
Add Drink
Ingredients
Name:
Add Ingredient
ID
Name
1
Tomato
2
Cheese
3
Chicken
4
Flour
Refresh List

The Menu tab allows the admin to manage all food and drink items available for ordering.
Users can add new items by specifying their name, price, and quantity, which are then stored in the Food or Drink tables.
If an item already exists, its quantity and price are updated instead of creating duplicates.
Availability is automatically handled through triggers — when stock becomes zero, the item's status changes to "Out of Stock."
Additionally, admins can manually mark items as In Stock or Out of Stock using dedicated buttons, ensuring accurate real-time inventory tracking in the system

7. *Quantity gets reduced as orders for that item is made:*





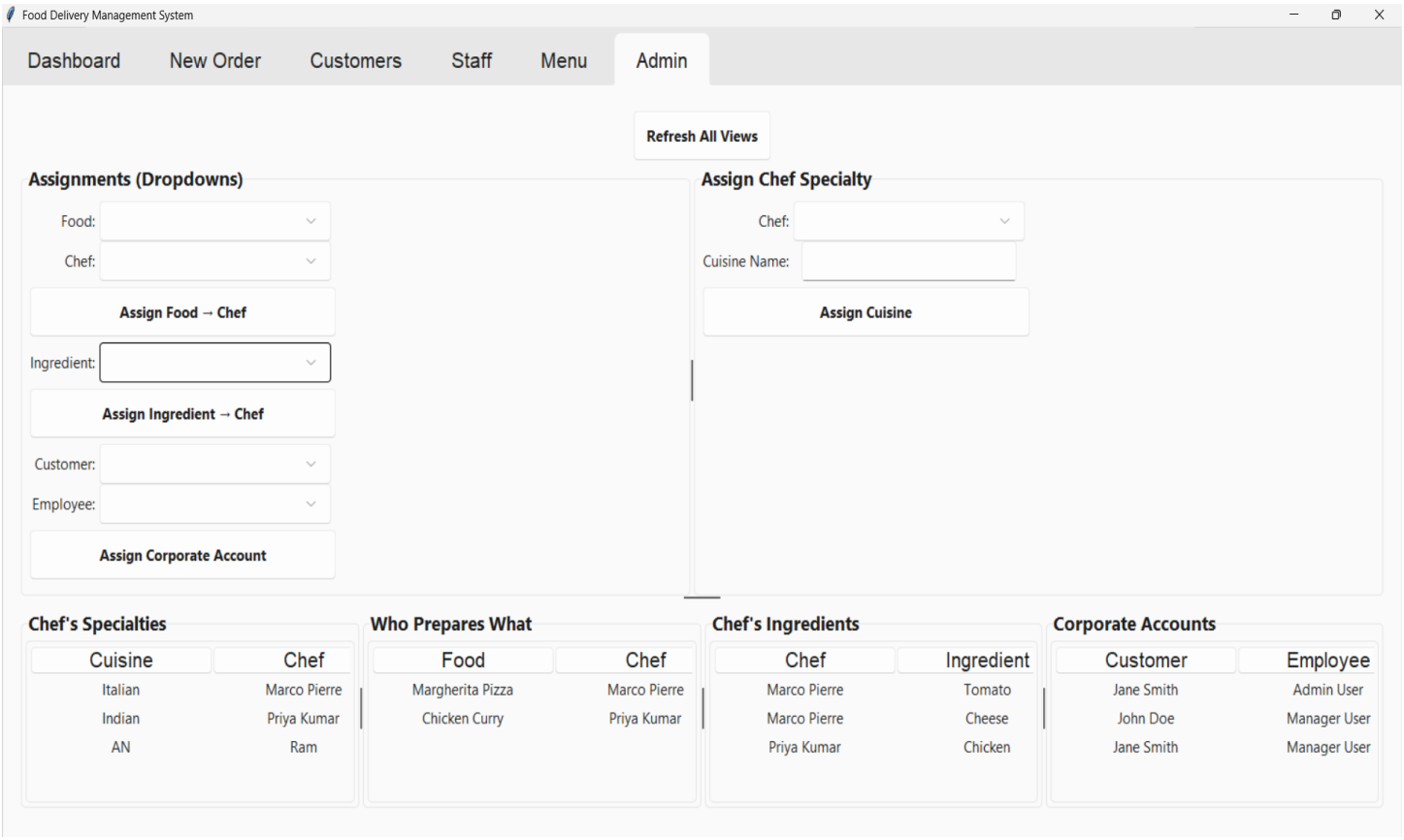
Whenever a new order is placed, the system automatically updates the stock levels of the ordered items.

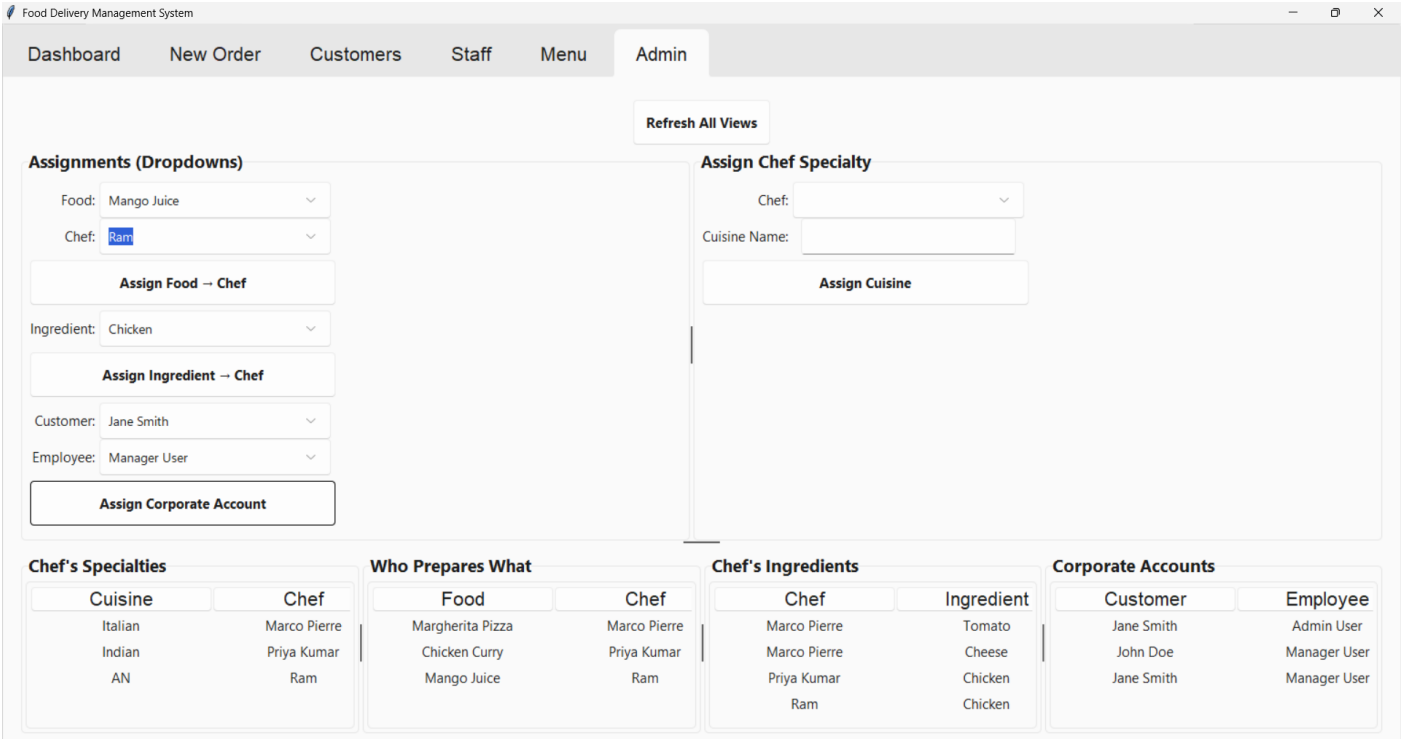
This functionality is handled through a database trigger that decrements the Quantity field in the Food and Drink tables each time an item is added to the Contains table.

If the remaining quantity reaches zero, the item's Availability status is instantly updated to "No", indicating it is out of stock.

This automated mechanism ensures that the inventory remains accurate and synchronized with real-time order activity, eliminating the need for manual stock adjustments.

8. *Ingredient management — maintaining the ingredients used by chefs.*





The Ingredients section allows the admin to manage and monitor all ingredients available in the kitchen database.
New ingredients can be added through the interface and are stored in the Ingredient table.
Using the Admin tab, each ingredient can be linked to a specific chef through the Uses relation table, showing which chef utilizes which ingredients.
This feature ensures transparency in kitchen operations and helps maintain an organized record of ingredient usage for inventory and preparation planning.

Triggers, Procedures/Functions, Nested query, Join, Aggregate queries

Triggers:

- ```
  DELIMITER $$
  CREATE TRIGGER update_food_status
  BEFORE UPDATE ON Food
  FOR EACH ROW
  BEGIN
    IF NEW.Quantity <= 0 THEN
      SET NEW.Availability = 'No';
    ELSE
      SET NEW.Availability = 'Yes';
    END IF;
  END$$
  DELIMITER ;
  ```

- ```
  DELIMITER $$
  CREATE TRIGGER update_drink_status
  BEFORE UPDATE ON Drink
  FOR EACH ROW
  BEGIN
    IF NEW.Quantity <= 0 THEN
      SET NEW.Availability = 'No';
    ELSE
      SET NEW.Availability = 'Yes';
    END IF;
  END$$
  DELIMITER ;
  ```

- ```
  DELIMITER $$
  CREATE TRIGGER check_food_status_on_insert
  BEFORE INSERT ON Food
  FOR EACH ROW
  BEGIN
    IF NEW.Quantity <= 0 THEN
      SET NEW.Availability = 'No';
    ELSE
      SET NEW.Availability = 'Yes';
    END IF;
  ```
  ```

```
END$$
DELIMITER ;
```

- ```
  DELIMITER $$
  CREATE TRIGGER check_drink_status_on_insert
  BEFORE INSERT ON Drink
  FOR EACH ROW
  BEGIN
   IF NEW.Quantity <= 0 THEN
     SET NEW.Availability = 'No';
   ELSE
     SET NEW.Availability = 'Yes';
   END IF;
  END$$
  DELIMITER ;
  ```

- ```
  DELIMITER $$
  CREATE TRIGGER before_payment_insert_calculate_total
  BEFORE INSERT ON Payment FOR EACH ROW
  BEGIN
     DECLARE total_cost DECIMAL(10, 2) DEFAULT 0.00;
     SELECT SUM(COALESCE(f.Price, 0) + COALESCE(d.Price, 0)) INTO
  total_cost
     FROM Contains c
     LEFT JOIN Food f ON c.Food_ID = f.Food_ID
     LEFT JOIN Drink d ON c.Drink_ID = d.Drink_ID
     WHERE c.Order_ID = NEW.Order_ID;
     SET NEW.Amount = COALESCE(total_cost, 0.00);
  END$$
  DELIMITER ;
  ```

- ```
  DELIMITER $$
  CREATE TRIGGER after_contains_insert_update_inventory
  AFTER INSERT ON Contains FOR EACH ROW
  BEGIN
     IF NEW.Food_ID IS NOT NULL AND NEW.Food_ID != 0 THEN
        UPDATE Food SET Quantity = Quantity - 1 WHERE Food_ID =
  NEW.Food_ID;
     END IF;
     IF NEW.Drink_ID IS NOT NULL AND NEW.Drink_ID != 0 THEN
        UPDATE Drink SET Quantity = Quantity - 1 WHERE Drink_ID =
  NEW.Drink_ID;
     END IF;
  ```
  ```

```
END$$
DELIMITER ;
```

Procedures:

- ```
  DELIMITER $$
  CREATE PROCEDURE PlaceCompleteOrder(IN p_Cust_ID INT,IN
  p_Food_ID INT,IN p_Drink_ID INT,IN p_PaymentMethod VARCHAR(50))
  BEGIN
     DECLARE new_OrderID INT;
     DECLARE new_PaymentID INT;
     SELECT COALESCE(MAX(Order_ID), 1000) + 1 INTO new_OrderID
  FROM Orders;
     SELECT COALESCE(MAX(Payment_ID), 700) + 1 INTO new_PaymentID
  FROM Payment;
     INSERT INTO Orders (Order_ID, Quantity, Cust_ID) VALUES
  (new_OrderID, 1, p_Cust_ID);
     INSERT INTO Contains (Order_ID, Food_ID, Drink_ID) VALUES
  (new_OrderID, p_Food_ID, p_Drink_ID);
     INSERT INTO Payment (Payment_ID, Method, Order_ID, Amount)
  VALUES (new_PaymentID, p_PaymentMethod, new_OrderID, 0);
     SELECT new_OrderID AS 'NewOrderID';
  END$$
  DELIMITER ;
  ```

Functions:

- ```
  DELIMITER $$
  CREATE FUNCTION GetCustomerOrderCount(p_Cust_ID INT)
  RETURNS INT
  DETERMINISTIC
  BEGIN
     DECLARE order_count INT;
     SELECT COUNT(*) INTO order_count FROM Orders WHERE Cust_ID =
  p_Cust_ID;
     RETURN order_count;
  END$$
  DELIMITER ;
  ```

Nested Queries:

- SELECT Cust_Name, Email
  FROM Customer
  WHERE Cust_ID IN (
  SELECT o.Cust_ID
  FROM Orders o
  JOIN Contains c ON o.Order_ID = c.Order_ID
  WHERE c.Food_ID = 301
  );

  SELECT (COALESCE(MAX(Cust_ID),0)+1) AS id FROM Customer;

Join:

- SELECT o.Order_ID, c.Cust_Name, p.Amount, p.Method, COALESCE(d.Del_Name, 'Not Assigned') AS Del_Name FROM Orders o JOIN Customer c ON o.Cust_ID = c.Cust_ID LEFT JOIN Payment p ON o.Order_ID = p.Order_ID LEFT JOIN Delivery_Incharge d ON o.Order_ID = d.Order_ID ORDER BY o.Order_ID DESC

  SELECT c.Cuisine_Name, ch.Chef_Name FROM Cuisine c JOIN Chef ch ON c.Chef_ID = ch.Chef_ID

Aggregate Queries:

- SELECT c.Cust_Name, SUM(p.Amount) AS Total_Spent_By_Customer
  FROM Customer c
  JOIN Orders o ON c.Cust_ID = o.Cust_ID
  JOIN Payment p ON o.Order_ID = p.Order_ID
  GROUP BY c.Cust_Name
  ORDER BY Total_Spent_By_Customer DESC;

  SELECT COUNT(*) INTO order_count FROM Orders WHERE Cust_ID = p_Cust_ID;

  SELECT COALESCE(MAX(Order_ID),1000)+1 FROM Orders;

Code snippets for invoking the Procedures/Functions/Trigger

1. Invoking a Stored Procedure:

```
def place_order():
    cust_name = po_cust_combo.get().strip()
    paym      = po_payment_method_combo.get().strip()

    if not cust_name:
        messagebox.showwarning("Input", "Please select a customer.")
        return
    if not cart_items:
        messagebox.showwarning("Cart", "Cart is empty. Add items first.")
        return
    if not paym:
        messagebox.showwarning("Payment", "Select a payment method.")
        return

    cust_id = customer_map.get(cust_name)
    if cust_id is None:
        messagebox.showerror("Selection Error", "Invalid customer.")
        return
    cn = get_db_connection()
    if cn is None: return
    try:
        cur = cn.cursor()
        cur.execute("SELECT COALESCE(MAX(Order_ID),1000)+1 FROM Orders")
        new_order_id = cur.fetchone()[0]

        cur.execute("SELECT COALESCE(MAX(Payment_ID),700)+1 FROM Payment")
        new_payment_id = cur.fetchone()[0]
        tot_items = sum(it['qty'] for it in cart_items)
        cur.execute("INSERT INTO Orders (Order_ID, Quantity, Cust_ID) VALUES (%s,%s,%s)",
                    (new_order_id, tot_items, cust_id))
        for it in cart_items:
            if it['type']=="Food":
                for _ in range(it['qty']):
                    cur.execute("INSERT INTO Contains (Order_ID, Food_ID, Drink_ID) VALUES (%s,%s,%s)",
                                (new_order_id, it['id'], 0))
            else:
                for _ in range(it['qty']):
                    cur.execute("INSERT INTO Contains (Order_ID, Food_ID, Drink_ID) VALUES (%s,%s,%s)",
                                (new_order_id, 0, it['id']))

        cur.execute("INSERT INTO Payment (Payment_ID, Method, Order_ID, Amount) VALUES (%s,%s,%s,%s)",
                    (new_payment_id, paym, new_order_id, 0))

        cn.commit()
        messagebox.showinfo("Success", f"Order {new_order_id} placed.")
        set_status(f"Order {new_order_id} placed. Method: {paym}")
```

Note: The Python application's place_order function implements the order logic manually. It runs the same set of INSERT statements as the PlaceCompleteOrder stored procedure, but it does not use a CALL statement. The stored procedure was tested separately in the SQL script

2. Invoking a Function:

```
def on_customer_select(_e=None):
    sel = customer_tree.focus()
    if not sel:
        return
    vals = customer_tree.item(sel)['values']
    if not vals:
        return
    cust_id = vals[0]
    r = execute_query("SELECT GetCustomerOrderCount(%s) AS c", (cust_id,))
    if r:
        cust_order_count_label.config(text=f"Total Orders for selected customer: {r[0]['c']}")
```

3. Invoking a Trigger:

```
# Payment: Amount will be auto-set by your BEFORE INSERT trigger
cur.execute("INSERT INTO Payment (Payment_ID, Method, Order_ID, Amount) VALUES (%s,%s,%s,%s)",
            (new_payment_id, paym, new_order_id, 0))
```

```
# Contains: insert one row per qty (works with your existing triggers)
for it in cart_items:
    if it['type']=="Food":
        for _ in range(it['qty']):
            cur.execute("INSERT INTO Contains (Order_ID, Food_ID, Drink_ID) VALUES (%s,%s,%s)",
                        (new_order_id, it['id'], 0))
    else:
        for _ in range(it['qty']):
            cur.execute("INSERT INTO Contains (Order_ID, Food_ID, Drink_ID) VALUES (%s,%s,%s)",
                        (new_order_id, 0, it['id']))
```

```
cur.execute("INSERT INTO Payment (Payment_ID, Method, Order_ID, Amount) VALUES
                    (new_payment_id, paym, new_order_id, 0))

        cn.commit()
        messagebox.showinfo("Success", f"Order {new_order_id} placed.")
        set_status(f"Order {new_order_id} placed. Method: {paym}")
```

```
next_id = execute_query("SELECT COALESCE(MAX(Food_ID), 300) + 1 AS next_id FROM Food")[0]['next_id']

        # Manually set availability based on the new qty
        avail_status = 'Yes' if qty_val > 0 else 'No'

        insert_query = "INSERT INTO Food (Food_ID, FName, Price, Quantity, Availability) VALUES (%s,%s,%s,%s,%s)"
        execute_query(insert_query, (next_id, name, price_val, qty_val, avail_status), fetch=False)
        messagebox.showinfo("Success", f"New food '{name}' added!")
```

```
nxt = execute_query("SELECT (COALESCE(MAX(Drink_ID),400)+1) AS id FROM Drink")
        next_id = nxt[0]['id']

        # Manually set availability based on the new qty
        avail_status = 'Yes' if q > 0 else 'No'

        ok = execute_query(
            "INSERT INTO Drink (Drink_ID,DName,Price,Quantity,Availability) VALUES (%s,%s,%s,%s,%s)",
            (next_id, name, p, q, avail_status), fetch=False
        )
        if ok:
            messagebox.showinfo("Success", "Drink added.")
            set_status(f"Drink '{name}' added (ID {next_id}) with stock: {q}.")
```

```
ok = execute_query(f"UPDATE {table} SET Availability=%s WHERE {id_col}=%s", (val, item_id), fetch=False)

    if ok:
        set_status(f"{table[:-1]} ID {item_id} marked {'In Stock' if in_stock else 'Out of Stock'}.")
        refresh_inventory()
```

```
update_query = "UPDATE Food SET Price=%s, Quantity=%s, Availability=%s WHERE Food_ID=%s"
        execute_query(update_query, (price_val, new_qty, avail_status, food_id), fetch=False)
        messagebox.showinfo("Updated", f"'{name}' already exists. Quantity updated!")
```

Complete SQL Script

DBMS_miniproject_Group22.sql

(Click the above icon for opening the sql file)