# CS 166 Project Report

## Group Information

Tony Trieu - netID: ttrie003
Ricardo Galeano - netID: rgale008

## Implementation Description

Our Pizza Store management system effectively implements a database-driven application that allows customers, drivers, and managers to interact with our relational database through a command-line interface. The system features secure user authentication, role-based access control, and order management functionality. Customers can register accounts, browse and filter the menu, place orders from their preferred stores, and track their order history. Drivers have additional privileges to update order statuses, while managers possess comprehensive administrative capabilities to update menu items and manage user accounts. We implemented database optimization through strategic indexing to improve query performance, particularly for order lookups and menu filtering operations. The application handles data validation robustly, preventing SQL injection vulnerabilities and ensuring data integrity through input validation. Our implementation also includes helpful features like warning users when ordering from closed stores and providing clear, user-friendly feedback for all operations.

## Tony’s part:

I implemented several key functions for our pizza ordering system. For **CreateUser**, I built a secure registration process that validates all fields, checking that phone numbers are digits-only using regex, and preventing duplicate usernames through database queries. The function assigns new users a "customer" role by default, creating a solid foundation for our system's user management.

```java
public static void CreateUser(PizzaStore esql) {
  try {
     System.out.println("\n=== USER REGISTRATION ===");
     System.out.print("Enter your login: ");
     String login = in.readLine().trim();
     System.out.print("Enter your password: ");
     String password = in.readLine().trim();
     System.out.print("Enter your phone number: ");
     String phoneNum = in.readLine().trim();

     if (login.isEmpty() || password.isEmpty() || phoneNum.isEmpty()) {
        System.out.println("Error: All fields must be filled out.");
        return;
     }
     if (!phoneNum.matches("\\d+")) {
        System.out.println("Error: Phone number must contain only numeric digits (0-9).");
        return;
     }
     // Check if the username already exists in the database
     String checkQuery = "SELECT login FROM Users WHERE login = '" + login.replace("'", "''") + "'";
     List<List<String>> result = esql.executeQueryAndReturnResult(checkQuery);
     if (!result.isEmpty()) {
        System.out.println("Error: This username already exists. Please choose another one.");
        return;
     }

     // Set default role for new users
     String role = "customer";
     String query = "INSERT INTO Users (login, password, role, favoriteItems, phoneNum) VALUES ('"
                  + login + "', '"
                  + password + "', '"
                  + role + "', NULL, '"
                  + phoneNum + "')";
     esql.executeUpdate(query);
     System.out.println("\nSuccess! User '" + login + "' has been registered as a customer.");
     System.out.println("You can now log in with your credentials.");

  } catch (SQLException e) {
     System.err.println("Database error during user creation: " + e.getMessage());
  } catch (Exception e) {
     System.err.println("Error processing your request: " + e.getMessage());
  }
}
```

In the **LogIn** function, I set up session management that verifies credentials and stores the user's role, enabling different permissions for customers, drivers, and managers throughout the application. This allows for properly restricted access to sensitive features.

```
411     public static String LogIn(PizzaStore esql){
412       try {
413          System.out.println("Login");
414          System.out.println("-----");
415          System.out.print("Enter username: ");
416          String login = in.readLine();
417          System.out.print("Enter password: ");
418          String password = in.readLine();
419
420          // Check if user exists and password matches
421          String query = String.format("SELECT * FROM Users WHERE login = '%s' AND password = '%s'", login, password);
422          int userCount = esql.executeQuery(query);
423          if (userCount == 1) {
424             // Get the user's role for permission checks later
425             String roleQuery = String.format("SELECT role FROM Users WHERE login = '%s'", login);
426             List<List<String>> result = esql.executeQueryAndReturnResult(roleQuery);
427             currentRole = result.get(0).get(0);
428             currentUser = login;
429             System.out.println("Login successful!");
430             System.out.println("Welcome, " + login + "! (Role: " + currentRole + ")");
431             return login;
432          } else {
433             System.out.println("Error: Invalid username or password.");
434             return null;
435          }
436       } catch (Exception e) {
437          System.err.println(e.getMessage());
438          return null;
439       }
440     }//end
```

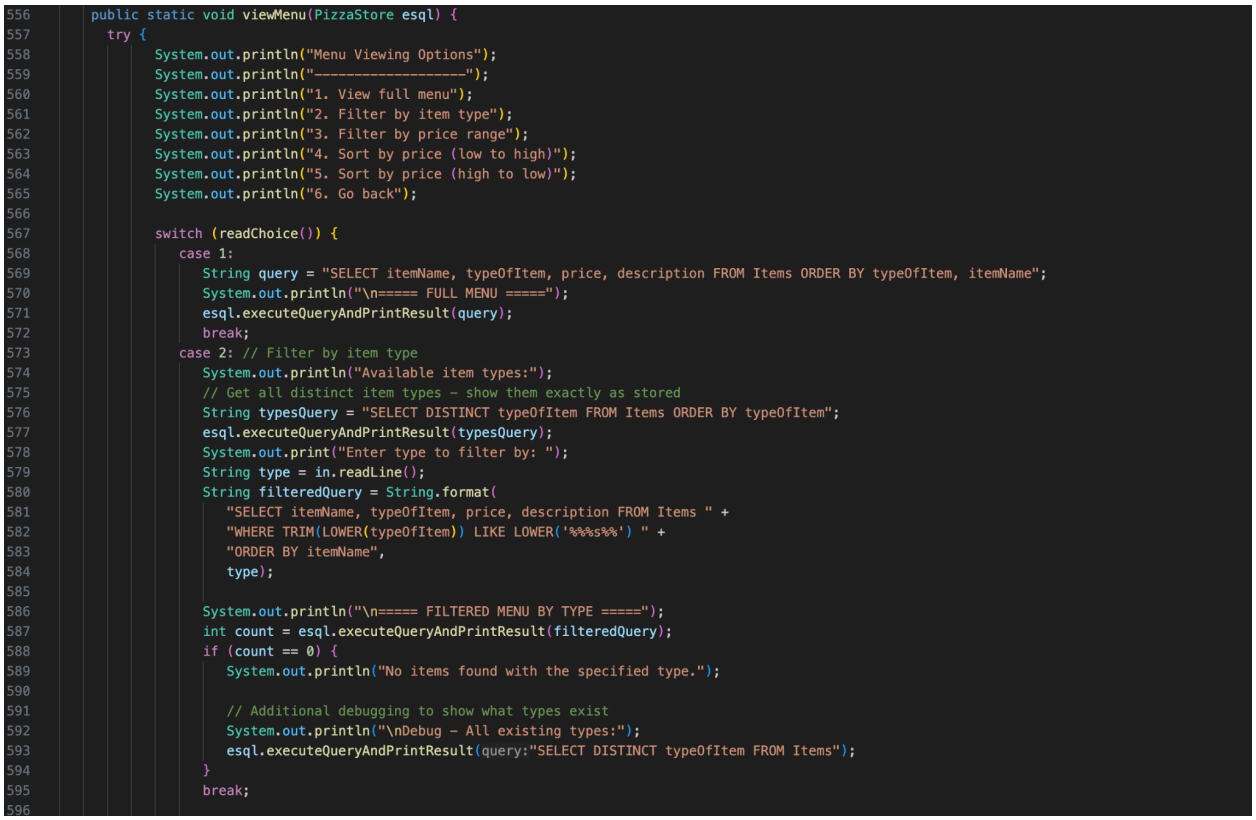
My profile management functions include **viewProfile**, which neatly displays user information, and **updateProfile**, which offers a menu to update passwords (with length validation), favorite items (verifying they exist in the menu), and phone numbers. These functions give users control while maintaining data integrity.

```
445     public static void viewProfile(PizzaStore esql) {
446       try {
447          System.out.println("User Profile");
448          System.out.println("------------");
449          if (currentUser == null) {
450             System.out.println("Error: No user is currently logged in.");
451             return;
452          }
453          String query = String.format("SELECT login, role, favoriteItems, phoneNum FROM Users WHERE login = '%s'", currentUser);
454          List<List<String>> result = esql.executeQueryAndReturnResult(query);
455          if (result.size() > 0) {
456             List<String> user = result.get(0);
457             System.out.println("Username: " + user.get(0));
458             System.out.println("Role: " + user.get(1));
459             System.out.println("Favorite Items: " + (user.get(2) == null || user.get(2).isEmpty() ? "None" : user.get(2)));
460             System.out.println("Phone Number: " + user.get(3));
461          } else {
462             System.out.println("Error: Could not retrieve user profile information.");
463          }
464       } catch (Exception e) {
465          System.err.println("Error viewing profile: " + e.getMessage());
466       }
467     }
```

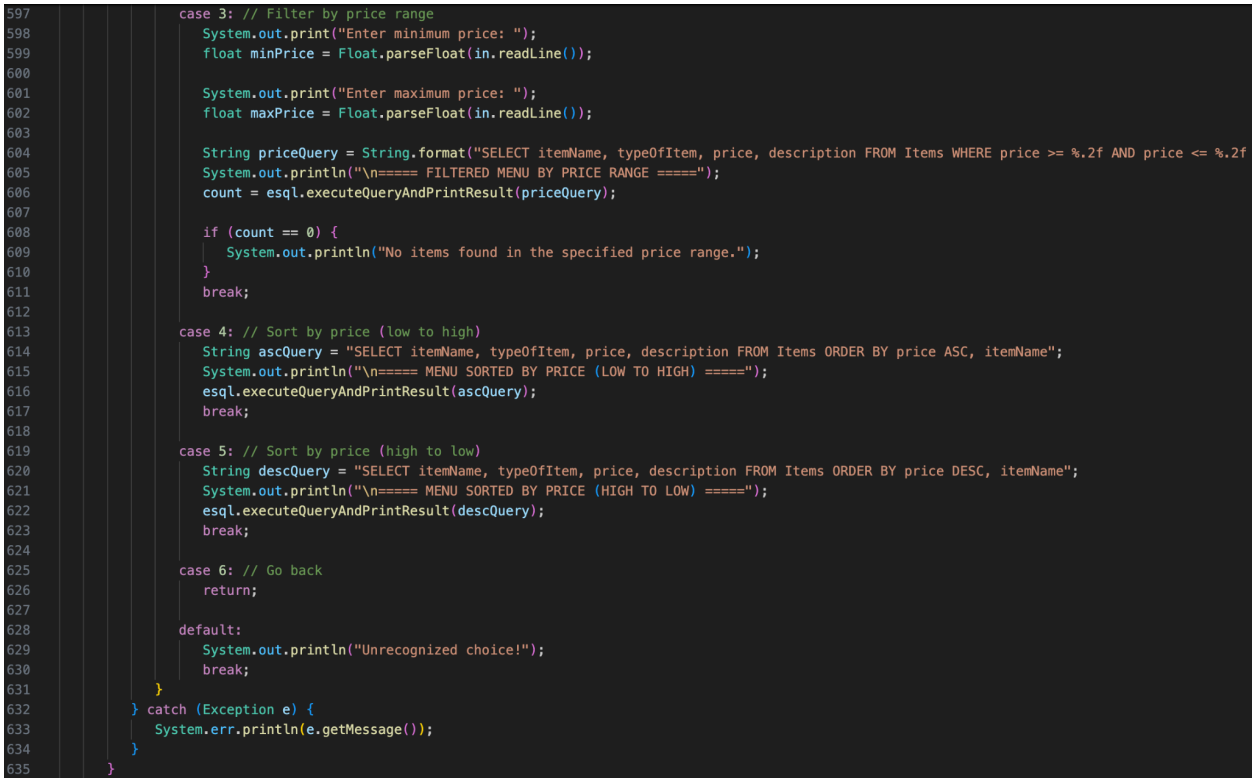
```java
    public static void updateProfile(PizzaStore esql) {
        try {
            if (currentUser == null) {
                System.out.println("Error: You must be logged in to update your profile.");
                return;
            }

            System.out.println("\nUpdate Profile");
            System.out.println("1. Password  2. Favorite Items  3. Phone Number  4. Go back");

            switch (readChoice()) {
                case 1: // Password
                    System.out.print("Current password: ");
                    String currentPassword = in.readLine();

                    // Verify password
                    String verifyQuery = String.format("SELECT login FROM Users WHERE login = '%s' AND password = '%s'", currentUser, currentPassword);
                    List<List<String>> result = esql.executeQueryAndReturnResult(verifyQuery);

                    if (result.isEmpty()) {
                        System.out.println("Error: Incorrect password.");
                        return;
                    }

                    System.out.print("New password (it has to be three characters or longer): ");
                    String newPassword = in.readLine();

                    if (newPassword.length() < 3) {
                        System.out.println("Error: Password too short.");
                        return;
                    }

                    // Update password
                    String updateQuery = String.format("UPDATE Users SET password = '%s' WHERE login = '%s'", newPassword, currentUser);
                    esql.executeUpdate(updateQuery);
                    System.out.println("Password updated successfully.");
                    break;

                    case 2: // Favorite items
                        System.out.print("Enter your favorite item: ");
                        String favoriteItem = in.readLine();
```

```java
                        // Check if the item exists
                        String checkItemQuery = String.format("SELECT itemName FROM Items WHERE itemName = '%s'", favoriteItem);
                        result = esql.executeQueryAndReturnResult(checkItemQuery);
                        if (result.isEmpty()) {
                            System.out.println("Error: Item not found in menu.");
                            return;
                        }

                        // Update favorite item
                        updateQuery = String.format("UPDATE Users SET favoriteItems = '%s' WHERE login = '%s'",
                            favoriteItem, currentUser);
                        esql.executeUpdate(updateQuery);
                        System.out.println("Favorite item updated successfully.");
                        break;

                    case 3: // Phone number
                        System.out.print("New phone number: ");
                        String phoneNum = in.readLine();

                        if (phoneNum.length() < 10) {
                            System.out.println("Error: Enter a valid phone number that's 10 digits or longer.");
                            return;
                        }
                        // Update phone number
                        updateQuery = String.format("UPDATE Users SET phoneNum = '%s' WHERE login = '%s'", phoneNum, currentUser);
                        esql.executeUpdate(updateQuery);
                        System.out.println("Phone number updated successfully.");
                        break;

                    case 4: // Go back
                        return;
                    default:
                        System.out.println("Invalid choice.");
                }
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }
    }
```

For **viewMenu**, I created versatile browsing options including full menu display, case-insensitive filtering by item type or price range, and price-based sorting. The use of SQL's TRIM and LOWER functions makes searches user-friendly regardless of how they format their queries.



```
public static void viewMenu(PizzaStore esql) {
  try {
        System.out.println("Menu Viewing Options");
        System.out.println("--------------------");
        System.out.println("1. View full menu");
        System.out.println("2. Filter by item type");
        System.out.println("3. Filter by price range");
        System.out.println("4. Sort by price (low to high)");
        System.out.println("5. Sort by price (high to low)");
        System.out.println("6. Go back");

        switch (readChoice()) {
           case 1:
              String query = "SELECT itemName, typeOfItem, price, description FROM Items ORDER BY typeOfItem, itemName";
              System.out.println("\n===== FULL MENU =====");
              esql.executeQueryAndPrintResult(query);
              break;
           case 2: // Filter by item type
              System.out.println("Available item types:");
              // Get all distinct item types - show them exactly as stored
              String typesQuery = "SELECT DISTINCT typeOfItem FROM Items ORDER BY typeOfItem";
              esql.executeQueryAndPrintResult(typesQuery);
              System.out.print("Enter type to filter by: ");
              String type = in.readLine();
              String filteredQuery = String.format(
                 "SELECT itemName, typeOfItem, price, description FROM Items " +
                 "WHERE TRIM(LOWER(typeOfItem)) LIKE LOWER('%%%s%%') " +
                 "ORDER BY itemName",
                 type);

              System.out.println("\n===== FILTERED MENU BY TYPE =====");
              int count = esql.executeQueryAndPrintResult(filteredQuery);
              if (count == 0) {
                 System.out.println("No items found with the specified type.");

                 // Additional debugging to show what types exist
                 System.out.println("\nDebug - All existing types:");
                 esql.executeQueryAndPrintResult(query:"SELECT DISTINCT typeOfItem FROM Items");
              }
              break;
```



```
                case 3: // Filter by price range
                    System.out.print("Enter minimum price: ");
                    float minPrice = Float.parseFloat(in.readLine());

                    System.out.print("Enter maximum price: ");
                    float maxPrice = Float.parseFloat(in.readLine());

                    String priceQuery = String.format("SELECT itemName, typeOfItem, price, description FROM Items WHERE price >= %.2f AND price <= %.2f
                    System.out.println("\n===== FILTERED MENU BY PRICE RANGE =====");
                    count = esql.executeQueryAndPrintResult(priceQuery);

                    if (count == 0) {
                        System.out.println("No items found in the specified price range.");
                    }
                    break;

                case 4: // Sort by price (low to high)
                    String ascQuery = "SELECT itemName, typeOfItem, price, description FROM Items ORDER BY price ASC, itemName";
                    System.out.println("\n===== MENU SORTED BY PRICE (LOW TO HIGH) =====");
                    esql.executeQueryAndPrintResult(ascQuery);
                    break;

                case 5: // Sort by price (high to low)
                    String descQuery = "SELECT itemName, typeOfItem, price, description FROM Items ORDER BY price DESC, itemName";
                    System.out.println("\n===== MENU SORTED BY PRICE (HIGH TO LOW) =====");
                    esql.executeQueryAndPrintResult(descQuery);
                    break;

                case 6: // Go back
                    return;

                default:
                    System.out.println("Unrecognized choice!");
                    break;
            }
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
```

The **placeOrder** function guides users through a complete ordering workflow with validation at each step. It warns about closed stores, verifies items exist, tracks running totals, and generates unique order IDs by incrementing from the current maximum, giving users immediate feedback throughout the process.

```java
    public static void placeOrder(PizzaStore esql) {
        try {
            if (currentUser == null) {
                System.out.println("Error: You must be logged in to place an order.");
                return;
            }

            System.out.println("Place New Order");
            System.out.println("---------------");

            // Show all stores - make sure to treat isOpen as a string
            System.out.println("Available stores:");
            String storeQuery = "SELECT storeID, address, city, state, isOpen FROM Store";
            esql.executeQueryAndPrintResult(storeQuery);
            // Select store
            System.out.print("Enter the store ID you want to order from: ");
            int storeID = Integer.parseInt(in.readLine());

            // Verify store exists
            String storeCheckQuery = String.format("SELECT storeID, isOpen FROM Store WHERE storeID = %d", storeID);
            List<List<String>> storeResult = esql.executeQueryAndReturnResult(storeCheckQuery);

            if (storeResult.isEmpty()) {
                System.out.println("Error: Invalid store selection.");
                return;
            }

            // Get isOpen as a string and check its value
            String isOpenStatus = storeResult.get(0).get(1);
            System.out.println("Store open status: " + isOpenStatus);

            // Warn users if store is not open (assuming "true"/"false" or "yes"/"no" strings)
            if (isOpenStatus.equalsIgnoreCase("false") || isOpenStatus.equalsIgnoreCase("no") || isOpenStatus.equalsIgnoreCase("closed") || isOpenStatus.equals("0")) {
                System.out.println("WARNING: This store appears to be closed. Do you still want to place an order? Type yes or no.");
                String userChoice = in.readLine();
                if (!userChoice.equalsIgnoreCase("yes")) {
                    System.out.println("Order cancelled.");
                    return;
                }
            }
```

```java

            int orderID = 0; // Default if no orders exist
            String orderIDQuery = "SELECT MAX(orderID) FROM FoodOrder";
            List<List<String>> result = esql.executeQueryAndReturnResult(orderIDQuery);

            if (result.size() > 0 && result.get(0).get(0) != null) {
                orderID = Integer.parseInt(result.get(0).get(0)) + 1;
            }

            float totalPrice = 0.0f;
            ArrayList<String> orderedItems = new ArrayList<>();
            ArrayList<Integer> itemQuantities = new ArrayList<>();

            // Add items to order
            boolean addingItems = true;
            while (addingItems) {
                // Display menu
                System.out.println("\nMenu:");
                String menuQuery = "SELECT itemName, price FROM Items ORDER BY itemName";
                esql.executeQueryAndPrintResult(menuQuery);

                System.out.print("Enter item name (or 'done' to finish): ");
                String itemName = in.readLine();

                if (itemName.equalsIgnoreCase("done")) {
                    addingItems = false;
                    continue;
                }

                // Verify item exists
                String itemCheckQuery = String.format("SELECT price FROM Items WHERE itemName = '%s'", itemName);
                List<List<String>> itemCheck = esql.executeQueryAndReturnResult(itemCheckQuery);

                if (itemCheck.isEmpty()) {
                    System.out.println("Error: Item not found on menu.");
                    continue;
                }

                System.out.print("Enter quantity: ");
                int quantity = Integer.parseInt(in.readLine());

```

```
                if (quantity <= 0) {
                    System.out.println("Error: Quantity must be greater than zero.");
                    continue;
                }

                // Add to order
                orderedItems.add(itemName);
                itemQuantities.add(quantity);

                // Update total price
                float itemPrice = Float.parseFloat(itemCheck.get(0).get(0));
                totalPrice += (itemPrice * quantity);

                System.out.println("Item added. Current total: $" + String.format("%.2f", totalPrice));
                System.out.print("Add another item? (y/n): ");
                String another = in.readLine();
                if (!another.equalsIgnoreCase("y")) {
                    addingItems = false;
                }
            }

            if (orderedItems.isEmpty()) {
                System.out.println("Order cancelled - no items selected.");
                return;
            }

            java.util.Date date = new java.util.Date();
            Timestamp timestamp = new Timestamp(date.getTime());
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String formattedTimestamp = sdf.format(timestamp);

            // Create order in database
            String orderQuery = String.format(
                "INSERT INTO FoodOrder (orderID, login, storeID, totalPrice, orderTimestamp, orderStatus) " +
                "VALUES (%d, '%s', %d, %.2f, '%s', 'Placed')",
                orderID, currentUser, storeID, totalPrice, formattedTimestamp);

            esql.executeUpdate(orderQuery);
```

```
            // Add items to order
            for (int i = 0; i < orderedItems.size(); i++) {
                String itemOrderQuery = String.format(
                    "INSERT INTO ItemsInOrder (orderID, itemName, quantity) VALUES (%d, '%s', %d)",
                    orderID, orderedItems.get(i), itemQuantities.get(i));
                esql.executeUpdate(itemOrderQuery);
            }
            System.out.println("\nOrder placed successfully!");
            System.out.println("Order ID: " + orderID);
            System.out.println("Total: $" + String.format("%.2f", totalPrice));
            System.out.println("Status: Placed");

        } catch (Exception e) {
            System.err.println("Error placing order: " + e.getMessage());
            // Print the complete stack trace for debugging
            e.printStackTrace();
        }
    }
```

Finally, **viewAllOrders** implements role-based access, letting managers and drivers see all orders while restricting customers to their own history. The function displays orders newest-first and leverages our custom index for fast performance even with large order volumes.

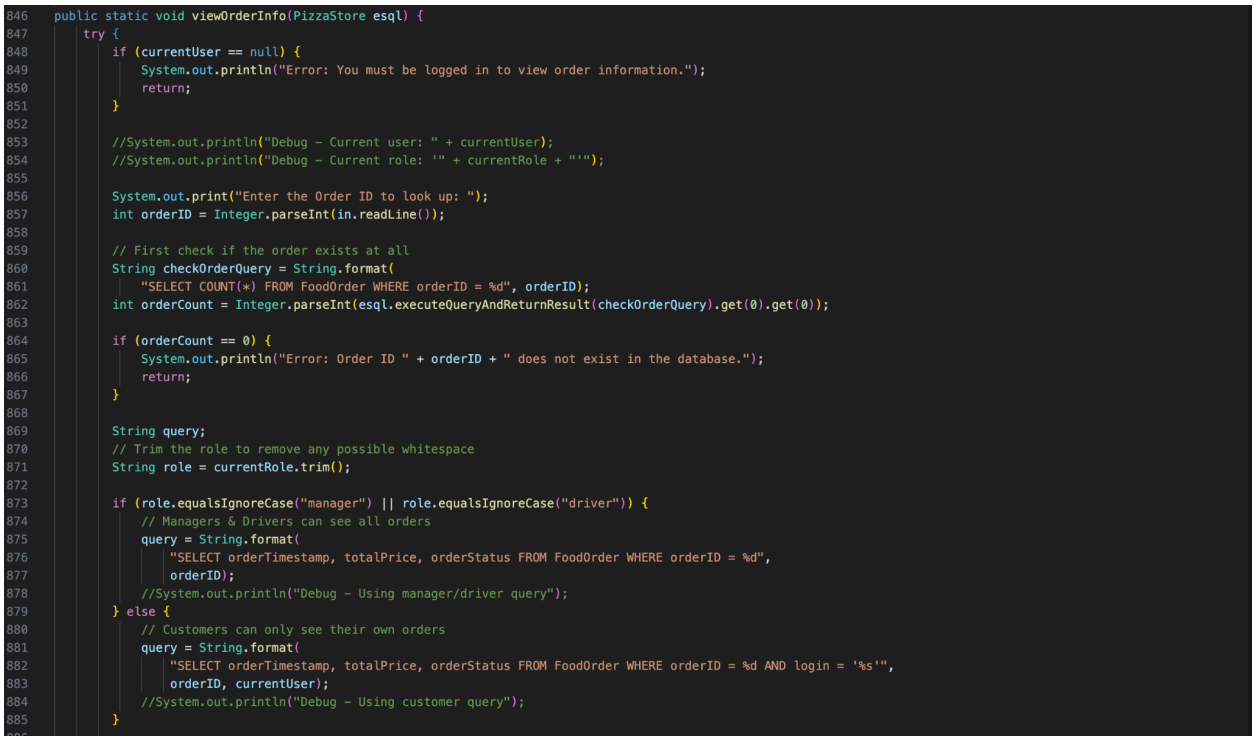
```java
public static void viewAllOrders(PizzaStore esql) {
   try {
      System.out.println("Order History");
      System.out.println("-------------");
      String query;

      // Check user role for permissions
      if (currentRole.equalsIgnoreCase("Manager") || currentRole.equalsIgnoreCase("Driver")) {
         // Managers and drivers can see every order
         System.out.println("1. View all orders in the system");
         System.out.println("2. View only my orders");
         System.out.print("Enter choice: ");
         int choice = Integer.parseInt(in.readLine());

         if (choice == 1) {
            query = "SELECT orderID, login, storeID, totalPrice, orderTimestamp, orderStatus FROM FoodOrder ORDER BY orderTimestamp DESC";
            System.out.println("\n===== ALL ORDERS IN SYSTEM =====");
         } else {
            query = String.format("SELECT orderID, storeID, totalPrice, orderTimestamp, orderStatus FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp D
            System.out.println("\n===== YOUR ORDERS =====");
         }
         } else {
            // Regular customers can only see their own orders
            query = String.format("SELECT orderID, storeID, totalPrice, orderTimestamp, orderStatus FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp D
            System.out.println("\n===== YOUR ORDERS =====");
         }

         int result = esql.executeQueryAndPrintResult(query);
         if (result == 0) {
            System.out.println("No orders found.");
         } else {
            System.out.println("\nTotal orders: " + result);
         }
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
      }
   }
```

## Ricardo's part

In my part of the project, I created **viewRecentOrders** to give users quick access to their latest activity, showing just their five most recent orders. The function uses our login-timestamp index for speed and the LIMIT clause to maintain performance as the database grows.

```java
public static void viewRecentOrders(PizzaStore esql) {
   try {
      if(currentUser == null) {
         System.out.print("Error: You must be logged in to view recent orders.");
         return;
      }
      System.out.println("\n===== Your 5 Most recent orders =====");

      String query = String.format(
         "SELECT orderID, storeID, totalPrice, orderTimestamp, orderStatus " +
         "FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp DESC LIMIT 5",
         currentUser
      );

      int resultCount = esql.executeQueryAndPrintResult(query);

      if(resultCount == 0) {
         System.out.println("No recent orders found.");
      }
   }
   catch (Exception e) {
         System.err.println("Error retrieving recent orders: " + e.getMessage());
      }
}
```

For **viewOrderInfo**, I built in security checks that first confirm the order exists, then verify the user has permission to view it based on their role. I handled char field whitespace with trim() and used two queries - one for order details and another for the items - to give users complete information about their purchases.



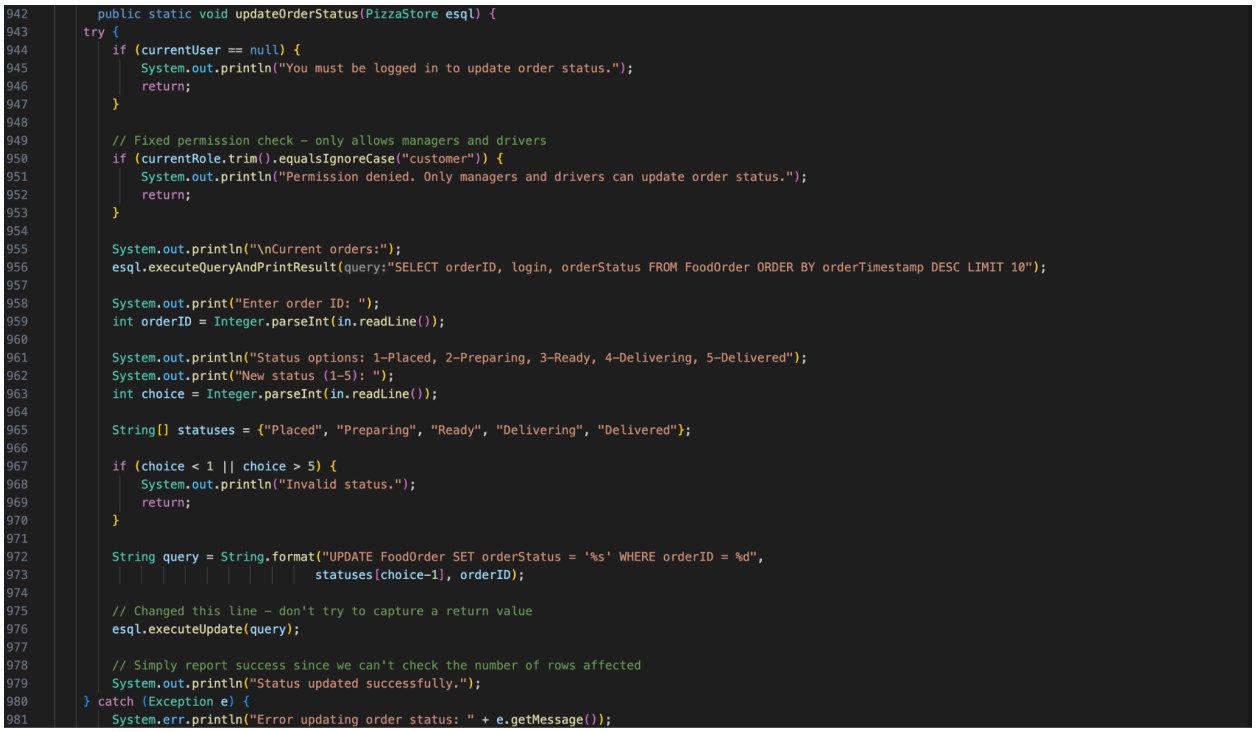
```
public static void viewOrderInfo(PizzaStore esql) {
    try {
        if (currentUser == null) {
            System.out.println("Error: You must be logged in to view order information.");
            return;
        }

        //System.out.println("Debug - Current user: " + currentUser);
        //System.out.println("Debug - Current role: '" + currentRole + "'");

        System.out.print("Enter the Order ID to look up: ");
        int orderID = Integer.parseInt(in.readLine());

        // First check if the order exists at all
        String checkOrderQuery = String.format(
            "SELECT COUNT(*) FROM FoodOrder WHERE orderID = %d", orderID);
        int orderCount = Integer.parseInt(esql.executeQueryAndReturnResult(checkOrderQuery).get(0).get(0));

        if (orderCount == 0) {
            System.out.println("Error: Order ID " + orderID + " does not exist in the database.");
            return;
        }

        String query;
        // Trim the role to remove any possible whitespace
        String role = currentRole.trim();

        if (role.equalsIgnoreCase("manager") || role.equalsIgnoreCase("driver")) {
            // Managers & Drivers can see all orders
            query = String.format(
                "SELECT orderTimestamp, totalPrice, orderStatus FROM FoodOrder WHERE orderID = %d",
                orderID);
            //System.out.println("Debug - Using manager/driver query");
        } else {
            // Customers can only see their own orders
            query = String.format(
                "SELECT orderTimestamp, totalPrice, orderStatus FROM FoodOrder WHERE orderID = %d AND login = '%s'",
                orderID, currentUser);
            //System.out.println("Debug - Using customer query");
        }
```

```
        //System.out.println("Debug - Query: " + query);

        List<List<String>> orderDetails = esql.executeQueryAndReturnResult(query);

        if (orderDetails.isEmpty()) {
            if (role.equalsIgnoreCase("customer")) {
                System.out.println("Error: You do not have permission to view this order.");
            } else {
                System.out.println("Error: Could not retrieve order details. Please contact technical support.");
            }
            return;
        }

        // Display Order Info
        List<String> order = orderDetails.get(0);
        System.out.println("\n===== Order Details =====");
        System.out.println("Timestamp: " + order.get(0));
        System.out.println("Total Price: $" + order.get(1));
        System.out.println("Status: " + order.get(2).trim()); // Trim to remove padding on char fields

        // Retrieve order items
        String itemsQuery = String.format(
            "SELECT itemName, quantity FROM ItemsInOrder WHERE orderID = %d",
            orderID);

        System.out.println("\n===== Order Items =====");
        int itemCount = esql.executeQueryAndPrintResult(itemsQuery);

        if (itemCount == 0) {
            System.out.println("No items found for this order.");
        }
    } catch (NumberFormatException e) {
        System.err.println("Error: Invalid order ID format. Please enter a numeric value.");
    } catch (Exception e) {
        System.err.println("Error retrieving order information: " + e.getMessage());
        e.printStackTrace(); // Print full stack trace for debugging
    }
}
```

The **viewStores** function provides a clean display of all store information including location, status, and reviews, sorted by ID for consistency. This straightforward approach helps customers quickly find where they want to order from.

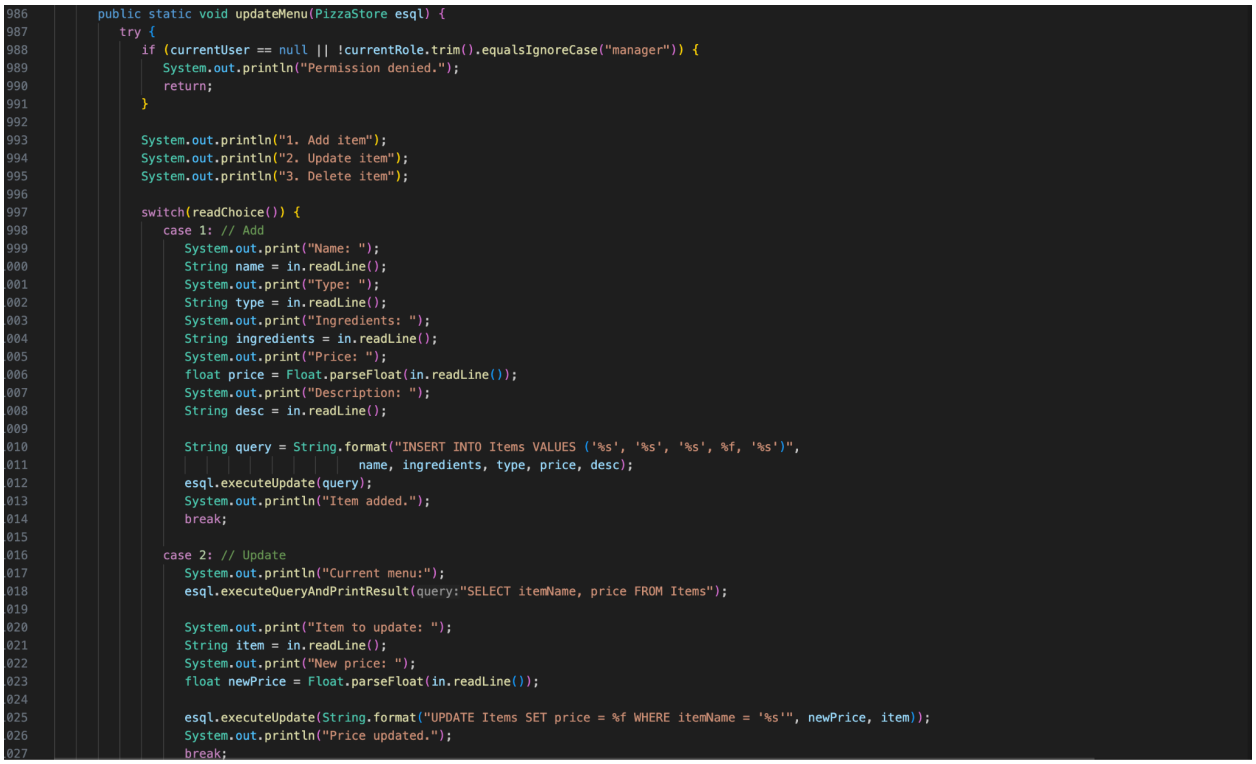
```
// Shows all store information
public static void viewStores(PizzaStore esql) {
    try {
        System.out.println("\n===== STORES =====");
        String query = "SELECT * FROM Store ORDER BY storeID";
        int count = esql.executeQueryAndPrintResult(query);

        if (count == 0) System.out.println("No stores found.");
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
}
```

With **updateOrderStatus**, I implemented role restrictions so only drivers and managers can change order statuses. The function displays recent orders for context, offers standardized status options through a simple menu, and provides clear feedback on the update result.



```
    public static void updateOrderStatus(PizzaStore esql) {
try {
    if (currentUser == null) {
        System.out.println("You must be logged in to update order status.");
        return;
    }

    // Fixed permission check - only allows managers and drivers
    if (currentRole.trim().equalsIgnoreCase("customer")) {
        System.out.println("Permission denied. Only managers and drivers can update order status.");
        return;
    }

    System.out.println("\nCurrent orders:");
    esql.executeQueryAndPrintResult(query:"SELECT orderID, login, orderStatus FROM FoodOrder ORDER BY orderTimestamp DESC LIMIT 10");

    System.out.print("Enter order ID: ");
    int orderID = Integer.parseInt(in.readLine());

    System.out.println("Status options: 1-Placed, 2-Preparing, 3-Ready, 4-Delivering, 5-Delivered");
    System.out.print("New status (1-5): ");
    int choice = Integer.parseInt(in.readLine());

    String[] statuses = {"Placed", "Preparing", "Ready", "Delivering", "Delivered"};

    if (choice < 1 || choice > 5) {
        System.out.println("Invalid status.");
        return;
    }

    String query = String.format("UPDATE FoodOrder SET orderStatus = '%s' WHERE orderID = %d",
                                 statuses[choice-1], orderID);

    // Changed this line - don't try to capture a return value
    esql.executeUpdate(query);

    // Simply report success since we can't check the number of rows affected
    System.out.println("Status updated successfully.");
} catch (Exception e) {
    System.err.println("Error updating order status: " + e.getMessage());
```

For **updateMenu**, I created comprehensive management options allowing managers to add, update, or remove menu items. I included confirmation steps for deletions and input validation throughout to maintain data integrity.



```
public static void updateMenu(PizzaStore esql) {
   try {
      if (currentUser == null || !currentRole.trim().equalsIgnoreCase("manager")) {
         System.out.println("Permission denied.");
         return;
      }

      System.out.println("1. Add item");
      System.out.println("2. Update item");
      System.out.println("3. Delete item");

      switch(readChoice()) {
         case 1: // Add
            System.out.print("Name: ");
            String name = in.readLine();
            System.out.print("Type: ");
            String type = in.readLine();
            System.out.print("Ingredients: ");
            String ingredients = in.readLine();
            System.out.print("Price: ");
            float price = Float.parseFloat(in.readLine());
            System.out.print("Description: ");
            String desc = in.readLine();

            String query = String.format("INSERT INTO Items VALUES ('%s', '%s', '%s', %f, '%s')",
                                         name, ingredients, type, price, desc);
            esql.executeUpdate(query);
            System.out.println("Item added.");
            break;

         case 2: // Update
            System.out.println("Current menu:");
            esql.executeQueryAndPrintResult(query:"SELECT itemName, price FROM Items");

            System.out.print("Item to update: ");
            String item = in.readLine();
            System.out.print("New price: ");
            float newPrice = Float.parseFloat(in.readLine());

            esql.executeUpdate(String.format("UPDATE Items SET price = %f WHERE itemName = '%s'", newPrice, item));
            System.out.println("Price updated.");
            break;
```

```
         case 3: // Delete
            System.out.println("Current menu:");
            esql.executeQueryAndPrintResult(query:"SELECT itemName FROM Items");

            System.out.print("Item to delete: ");
            String delItem = in.readLine();

            System.out.print("Confirm delete (y/n): ");
            if (in.readLine().equalsIgnoreCase("y")) {
               esql.executeUpdate(String.format("DELETE FROM Items WHERE itemName = '%s'", delItem));
               System.out.println("Item deleted.");
            }
            break;
      }
   }
   catch (Exception e) {
      System.err.println(e.getMessage());
   }
}
```

Finally, **updateUser** gives managers controlled access to modify accounts - changing roles between customer, driver, and manager, or resetting passwords when needed. I added role validation to prevent invalid assignments and strict permission checking to protect these administrative functions.

```
1050        public static void updateUser(PizzaStore esql) {
1051           try {
1052              if (currentUser == null || !currentRole.trim().equalsIgnoreCase("manager")) {
1053                 System.out.println("Permission denied.");
1054                 return;
1055              }
1056
1057              System.out.println("Users:");
1058              esql.executeQueryAndPrintResult(query:"SELECT login, role FROM Users");
1059
1060              System.out.print("Username to modify: ");
1061              String user = in.readLine();
1062
1063              System.out.println("1. Change role");
1064              System.out.println("2. Reset password");
1065
1066              switch(readChoice()) {
1067                 case 1:
1068                    System.out.print("New role (customer/driver/manager): ");
1069                    String role = in.readLine();
1070
1071                    if (role.equalsIgnoreCase("customer") ||
1072                       role.equalsIgnoreCase("driver") ||
1073                       role.equalsIgnoreCase("manager")) {
1074
1075                       esql.executeUpdate(String.format("UPDATE Users SET role = '%s' WHERE login = '%s'", role, user));
1076                       System.out.println("Role updated.");
1077                    } else {
1078                       System.out.println("Invalid role.");
1079                    }
1080                    break;
1081
1082                 case 2:
1083                    System.out.print("New password: ");
1084                    String pass = in.readLine();
1085
1086                    esql.executeUpdate(String.format("UPDATE Users SET password = '%s' WHERE login = '%s'", pass, user));
1087                    System.out.println("Password reset.");
1088                    break;
1089              }
1090           }
```

```
1091           catch (Exception e) {
1092              System.err.println(e.getMessage());
1093           }
1094        }
1095     }
```

## Extra credit

### Performance tuning / indexes

sql > src > create_indexes.sql

```sql
CREATE INDEX idx_users_login_password ON Users(login, password);
CREATE INDEX idx_items_type_price_name ON Items(typeOfItem, price, itemName);
CREATE INDEX idx_foodorder_login_timestamp ON FoodOrder(login, orderTimestamp DESC);
CREATE INDEX idx_orderid_itemname ON ItemsInOrder(orderID, itemName);
CREATE INDEX idx_store_location ON Store(city, state);

```

We thought hard about which indexes would help our database run faster based on how people actually use the pizza app. Here's why we picked each one:

The Users table index on (login, password) makes logging in super quick. Since every user has to login before ordering pizza, this happens constantly. Without this index, the system would need to check every single user account when someone tries to log in. And that would be a very slow process / operation because there are over 1000 users.

For the menu, we know that people usually filter by what type of food they want (like "just show me pizzas" or "just show me drinks") and then sort by price. So we made an index on Items(typeOfItem, price, itemName) that follows exactly this pattern. It's like organizing a menu where you first group by category, then by price, with each item name listed last.

When users check their order history, they almost always want to see their most recent orders first. That's why we created an index on FoodOrder(login, orderTimestamp DESC) with that DESC part being crucial - it means "show newest first." This makes the "see your recent 5 orders" feature feel instant instead of slow.

The ItemsInOrder table tracks which food items belong to which orders. Since we're constantly looking up "what items were in order #12345?", we indexed (orderID, itemName) to make these lookups faster. Without this, viewing order details would get painfully slow as our order database grows.

Finally, we added an index on Store(city, state) because we figured people often want to find pizza stores near them. This lets the app quickly show all stores in your city without checking every single store location in the database.

We tried to be smart about not over-indexing too - each index takes up storage space and slows down when you add new data. These five indexes hit the sweet spot where they speed up the most common operations without creating too much overhead.

## Triggers and Procedures

To enforce data integrity and validation rules within our database, we implemented triggers that prevent invalid data from being inserted or updated. These triggers ensure phone number format validation in the Users table and quantity validation in the ItemsInOrder table.

### Phone Number Validation Trigger

To enforce valid phone numbers, Ricardo created a function validate_phone_number(), which checks two conditions:

- The phone number must contain only numeric digits (^[0-9]+$ regex).
- The phone number must be at least 10 digits long.
- If either condition is not met, an error is raised, preventing invalid data from being inserted or updated.

```sql
-- Function to validate phone number format
CREATE OR REPLACE FUNCTION validate_phone_number()
RETURNS "trigger" AS
$BODY$
BEGIN
    -- Check if phone number contains only digits
    IF NEW.phoneNum !~ '^[0-9]+$' THEN
        RAISE EXCEPTION 'Phone number must contain only numeric digits';
    END IF;

    -- Check phone number length
    IF length(NEW.phoneNum) < 10 THEN
        RAISE EXCEPTION 'Phone number must be at least 10 digits long';
    END IF;

    RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE;

-- Create the trigger
DROP TRIGGER IF EXISTS phone_number_validation ON Users;
CREATE TRIGGER phone_number_validation
BEFORE INSERT OR UPDATE ON Users
FOR EACH ROW
EXECUTE PROCEDURE validate_phone_number();
```

**Item Quantity Validation Trigger**

To prevent negative or zero quantities in orders, Ricardo created a trigger function validate_item_quantity(), which enforces that all order quantities must be greater than zero. This trigger ensures that customers cannot place orders with zero or negative quantities, preventing logical errors and maintaining data accuracy in the ItemsInOrder table. If an invalid quantity is entered, an exception is raised, forcing the user to provide a valid quantity.

```
28
29  -- This trigger ensures that order quantities are valid (greater than zero)
30  CREATE OR REPLACE FUNCTION validate_item_quantity()
31  RETURNS "trigger" AS
32  $BODY$
33  BEGIN
34      IF NEW.quantity <= 0 THEN
35          RAISE EXCEPTION 'Item quantity must be greater than zero';
36      END IF;
37
38      RETURN NEW;
39  END;
40  $BODY$
41  LANGUAGE plpgsql VOLATILE;
42
43  -- Create the trigger
44  DROP TRIGGER IF EXISTS item_quantity_validation ON ItemsInOrder;
45  CREATE TRIGGER item_quantity_validation
46  BEFORE INSERT OR UPDATE ON ItemsInOrder
47  FOR EACH ROW
48  EXECUTE PROCEDURE validate_item_quantity();
```

```sql
-- Tony's triggers
CREATE OR REPLACE FUNCTION log_status_change()
RETURNS "trigger" AS
$BODY$
BEGIN
    -- Only update timestamp if status has changed
    IF OLD.orderStatus <> NEW.orderStatus THEN
        NEW.orderTimestamp = CURRENT_TIMESTAMP;

    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE;


DROP TRIGGER IF EXISTS order_status_update_trigger ON FoodOrder;
CREATE TRIGGER order_status_update_trigger
BEFORE UPDATE ON FoodOrder
FOR EACH ROW
WHEN (OLD.orderStatus IS DISTINCT FROM NEW.orderStatus)
EXECUTE PROCEDURE log_status_change();
```

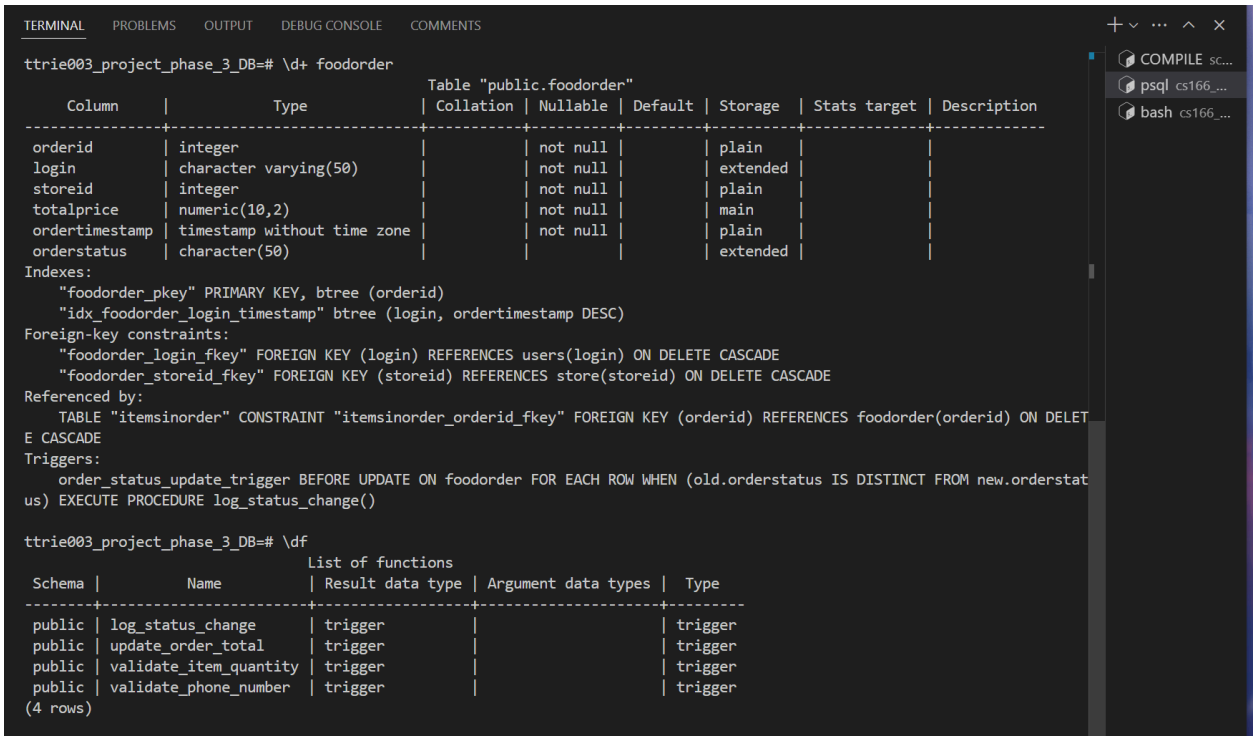
```sql
CREATE OR REPLACE FUNCTION update_order_total()
RETURNS "trigger" AS
$BODY$
DECLARE
    calculated_total DECIMAL(10,2);
BEGIN
    -- Calculate the new total price based on items and quantities
    SELECT COALESCE(SUM(i.price * io.quantity), 0.00)
    INTO calculated_total
    FROM ItemsInOrder io
    JOIN Items i ON io.itemName = i.itemName
    WHERE io.orderID = NEW.orderID;

    -- Update the order with the calculated total
    UPDATE FoodOrder
    SET totalPrice = calculated_total
    WHERE orderID = NEW.orderID;

    RETURN NULL;
END;
$BODY$
LANGUAGE plpgsql VOLATILE;

DROP TRIGGER IF EXISTS calculate_order_total ON ItemsInOrder;
CREATE TRIGGER calculate_order_total
AFTER INSERT OR UPDATE OR DELETE ON ItemsInOrder
FOR EACH ROW
EXECUTE PROCEDURE update_order_total();
```

In my portion of the triggers, I (Tony), carefully implemented two key triggers in our PizzaStore database to automate important workflows and ensure data consistency across related tables.

The first trigger, **order_status_update**, automatically updates the timestamp of an order whenever its status changes. I chose to implement this as a BEFORE UPDATE trigger on the FoodOrder table, which allows me to modify the NEW row values before they're committed to the database. The trigger fires only when there's an actual status change, which I enforced with a WHEN clause that checks if the old and new status values are different. Within the trigger function, I verify the status change condition again and then set the orderTimestamp to the current time. This approach ensures that we maintain an accurate record of when each status transition occurred, which is vital for order tracking and customer service.

The second trigger, **calculate_order_total**, handles the automatic calculation of order totals when items are added to, modified in, or removed from an order. I implemented this as an AFTER trigger on the ItemsInOrder table because it needs to react to changes in the order's line items and then update the related FoodOrder record. The trigger function queries the current items and quantities for the specified order ID, calculates the total price by joining with the Items table to get the current prices, and then updates the totalPrice field in the corresponding FoodOrder record. By returning NULL at the end of the function, I ensure the trigger completes properly for AFTER triggers.

These triggers eliminate the need for manual calculations and updates in the application code, reducing the risk of errors and inconsistencies in the database. They ensure that our order timestamps accurately reflect status changes and that order totals always match the sum of their line items, maintaining data integrity across related tables in our database.




## Problems/Findings

The bug in our updateOrderStatus method involved a permission check that incorrectly denied access to managers and drivers despite them having sufficient privileges. This occurred because PostgreSQL stores values in fixed-length char(20) fields with trailing spaces to pad them to exactly 20 characters (e.g., "manager" becomes "manager "). When our code retrieved the currentRole value and performed a string comparison with equalsIgnoreCase("manager"), the comparison failed because "manager " doesn't equal "manager". The solution was to use the trim() method on the role string before comparison, which removes all leading and trailing whitespace characters, allowing the role check to function correctly. This illustrates how database-specific storage implementations can subtly affect application logic when not properly accounted for.

## Contributions

Tony implemented the createUser, Login, viewProfile, updateProfile, viewMenu, placeOrder, and the viewAllOrders methods. Ricardo implemented the updateUser,

viewRecentOrders, updateMenu, updateOrderStatus, viewStores, and the viewOrderInfo methods. Ricardo and Tony worked on the indexes together. Ricardo worked on the validate phone number and validate item quantity triggers and procedures. Tony worked on the order status update and the calculate order total triggers and procedures.