CS 166 Project Report

Group Information

Include group number.
Include each team member's name and netid
Ryan Noghani: rnogh001
Nashwaan Ali Khan: nkhan040

Implementation Description

Include high-level description of your implementation (1 paragraph max)

We consider what the role and login of the current user is, and based on that give them certain permissions for what they can do in each function. Each function typically consists of taking input from the user, creating an sql query, executing it, and displaying the results in a way that's readable for the user.

Include screenshots and/or code snippets for each query. In addition, explain how you implemented each query and what purpose it fulfills. (no more than 3-4 sentences per query)

```
System.out.print("Enter phone number: ");
String phoneNum = in.readLine();

String query = String.format(
    "INSERT INTO Users (login, password, role, favoriteItems, phoneNum) VALUES ('%s', '%s', '%s', '%s', '%s');",
    login, password, role, favoriteItems, phoneNum
);
```

So for this inserts into what we are doing in this query we make a new user and a corresponding table after we give customer to the user, and we do the prompts that we need to do basically and for implementation we get login details and make sure it's right; contact details, and through esql object

```
String query = String.format(
   "SELECT Users.* FROM Users WHERE login = '%s' AND password = '%s';",
   login, password
);
```

So for this query all we are doing purpose wise is checking, whenever a user logs in, if that log in attempt is connected to a user who actually exists into the system, and LogIn

with esql object as a parameter

```
try {
   System.out.print("YOUR PROFILE\n");
   String query = String.format("SELECT favoriteItems, phoneNum FROM Users WHERE login = '%s';", login);
   esql.executeQueryAndPrintResult(query);
}
catch (Exception e) {
   System.err.println("Error retrieving profile: " + e.getMessage());
}
```

So what this query does is as shown gets favorite items in addition and in simultaneously with the phone number of the user and it has to match the user who's logging in and the implementation is in viewProfile which takes in esql and login.

```
switch (input) {
   case 1:
      System.out.print("Enter new favorite items: ");
      newFavoriteItems = in.readLine();
      updateQuery = String.format("UPDATE Users SET favoriteItems = '%s' WHERE login = '%s';", newFavoriteItems, login);
      esql.executeUpdate(updateQuery);
      System.out.println("Profile updated successfully!");
      break;
   case 2:
      System.out.print("Enter new phone number: ");
      newPhoneNumber = in.readLine();
      updateQuery = String.format("UPDATE Users SET phoneNum = '%s' WHERE login = '%s';", newPhoneNumber, login);
      esql.executeUpdate(updateQuery);
      System.out.println("Profile updated successfully!");
      break;
   case 3:
      System.out.print("Enter new password: ");
      newPassword = in.readLine();
      updateQuery = String.format("UPDATE Users SET password = '%s' WHERE login = '%s';", newPassword, login);
      esql.executeUpdate(updateQuery);
      System.out.println("Profile updated successfully!");
      break;
   default:
      System.out.println("Invalid choice.");
      return;
}
```

So these 3 updates are for updating the favoriteItems, phoneNum and passwords respectively. We have to make sure that it lines up with the login, so that no malicious stuff happens. Then we do this implementation in updateProfile which takes the esql object, login and role (so nobody does funny stuff).

```
case 1:
    query = "SELECT * FROM ITEMS;";
        rowCount = esql.executeQueryAndPrintResult(query);
        System.out.println("Total items found: " + rowCount);
    break;

case 2:
    System.out.print("Enter Type\n");
    String typeOfItem = in.readLine();
    query = String.format("SELECT * FROM ITEMS WHERE typeOfItem = '%s';", typeOfItem);
    rowCount = esql.executeQueryAndPrintResult(query);
        System.out.println("Total items found: " + rowCount);
        break;

case 3:
    System.out.print("Enter Price\n");
        double price = Double.parseDouble(in.readLine());
        query = String.format("SELECT * FROM ITEMS WHERE price = %.2f;", price);
        rowCount = esql.executeQueryAndPrintResult(query);
        System.out.println("Total items found: " + rowCount);
        break;

case 4:
    query = "SELECT * FROM ITEMS ORDER BY price DESC;";
    rowCount = esql.executeQueryAndPrintResult(query);
        System.out.println("Total items found: " + rowCount);
        break;

case 5:
    query = "SELECT * FROM ITEMS ORDER BY price ASC;";
        rowCount = esql.executeQueryAndPrintResult(query);
        System.out.println("Total items found: " + rowCount);
        break;

case 6: break;

default : System.out.println("Unrecognized choice!"); break;
```

So here we do these select queries to get the menu. The where clauses help guide us in terms of filtering by getting a specific item type or specific price. We use the ORDER BY for ordering our menu in descending or ascending order. The implementation is in the viewMenu function which only takes our esql object as a param.

```
String orderQuery = String.format("INSERT INTO FoodOrder (orderID, login, storeID, totalPrice, orderTimestamp, orderStatus) VALUES (%d, '%s', %d, %.2f, '%s', '%s');", orderID, login, storeID, totalPrice, ord
esql.executeUpdate(orderQuery);

// Insert each item into ItemsInOrder table
for (int i = 0; i < itemNames.size(); i++) {
    String insertItemQuery = String.format(
        "INSERT INTO ItemsInOrder (orderID, itemName, quantity) VALUES (%d, '%s', %d);",
        orderID, itemNames.get(i), quantities.get(i));
    esql.executeUpdate(insertItemQuery);
}

System.out.println("Order placed successfully! Total Price: $" + totalPrice);

}
```

So here we do these inserts into queries, primarily focusing on putting an order which will correspondingly update the FoodOrder, and the second one, it updates ItemsInOrder, by the respective parameters. ItemsInOrder can experience multiple insertions during a user's order. This implementation is carried out in the placeOrder function which takes an esql object and login as params.

```
public static void viewAllOrders(PizzaStore esql, String login, String role) {
 try {
    String query;

    if (role.equals("customer")) {
        query = String.format("SELECT * FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp DESC;", login);
    }
    else {
        System.out.println("Enter login of user who's order history you want to see: ");
        String update_login = in.readLine();
        query = String.format("SELECT * FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp DESC;", update_login);
    }

    int rowCount = esql.executeQueryAndPrintResult(query);
    if (rowCount == 0) {
        System.out.println("No orders found.");
    }


 } catch (Exception e) {
    System.err.println("Error retrieving order history: " + e.getMessage());
 }
```

Basically we have to select queries. One of them will get all the FoodOrders that the user has ever ordered. The other one is for managers to access a specific user's (or their own) orders. This second query is making sure that the inputted login aligns with a user who has an order history. We pass in the role to make sure some funny stuff doesn't occur, like the customer trying to snoop other people's order history. Also the implementation is for viewAllOrders which takes the esql object, login, and role as parameters.

```java
public static void viewRecentOrders(PizzaStore esql, String login, String role) {

  try {
     String query;

     if (role.equals("customer")) {
        query = String.format("SELECT * FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp DESC LIMIT 5;", login);
     }
     else {
        System.out.println("Enter login of user who's order history you want to see: ");
        String update_login = in.readLine();
        query = String.format("SELECT * FROM FoodOrder WHERE login = '%s' ORDER BY orderTimestamp DESC LIMIT 5;", update_login);
     }

     int rowCount = esql.executeQueryAndPrintResult(query);
     if (rowCount == 0) {
        System.out.println("No orders found.");
     }


}
catch (Exception e) {
    System.err.println("Error retrieving order history: " + e.getMessage());
}
}
```

As you can see this select query will get the FoodOrders assuming the login aligns with the user currently trying to enter the system. The orderTimeStamp gets the history and makes sure it's in the last 5 orders, and our implementation is in viewRecentOrders, which takes the esql object, login, and role.

```java
public static void viewOrderInfo(PizzaStore esql, String login, String role) {
   try {
      String existsQuery = "";
      role = role.trim();
      int orderID;
      int count;
      if (role.equals("customer")) {

         System.out.print("Enter Order ID: ");
         orderID = Integer.parseInt(in.readLine());
         existsQuery = String.format("SELECT * FROM FoodOrder WHERE orderID = %d AND login = '%s';", orderID, login);
         count = esql.executeQuery(existsQuery);

         if (count == 0) {
            System.out.println("Order ID not found! Returning to menu.");
            return;
         }

         esql.executeQueryAndPrintResult(existsQuery);


         existsQuery = String.format("SELECT * FROM itemsinorder WHERE orderID = %d;", orderID);
         esql.executeQueryAndPrintResult(existsQuery);
      }
```

The first query checks whether the user has a specific order under their name. If they do, then that order information is displayed to them. Otherwise, they are told the order ID they’re looking for wasn’t found.

```java
public static void viewStores(PizzaStore esql) {

   try {
      String query = "SELECT * FROM Store;";
      esql.executeQueryAndPrintResult(query);
   }

   catch (Exception e) {
     System.err.println("Error viewing stores: " + e.getMessage());
   }
}
```

This query is pretty straight forward. As you can see it just selects every single item from the Store table, the implementation is in viewStores function which only takes the esql object as a parameter.

```java
public static void updateOrderStatus(PizzaStore esql) {

   try {
      System.out.println("Enter Order ID: ");
      int orderID = Integer.parseInt(in.readLine());
      String existsQuery = String.format("SELECT * FROM FoodOrder WHERE orderID = %d;", orderID);
      int count = esql.executeQuery(existsQuery);
      if (count == 0) {
         System.out.println("OrderID not found! Returning to menu.");
         return;
      }


      System.out.println("Enter New Order Status: ");
      String newStatus = in.readLine();
      String updateQuery = String.format("UPDATE FoodOrder SET orderStatus = '%s' WHERE orderID = %d;", newStatus, orderID);
      esql.executeQuery(updateQuery);
      System.out.println("Status Updated Successfully!");


   }

   catch (Exception e) {
     System.err.println("Error viewing stores: " + e.getMessage());
   }

}
```

Essentially what we are doing here is that we update the status of a specific order by using the orderID to track it. Our implementation can be seen in updateOrderStatus which takes the esql object as a parameter. The first query is for checking if the order even exists, while the second query does the actual updating when we know the order exists.

```java
String updateQuery = "";
switch (option) {
    case 1:
        System.out.print("Enter new price: ");
        double newPrice = Double.parseDouble(in.readLine());
        updateQuery = String.format("UPDATE Items SET price = %.2f WHERE itemName = '%s';", newPrice, itemName);
        break;
    case 2:
        System.out.print("Enter new type: ");
        String newType = in.readLine();
        updateQuery = String.format("UPDATE Items SET type = '%s' WHERE itemName = '%s';", newType, itemName);
        break;
    case 3:
        System.out.print("Enter new description: ");
        String newDesc = in.readLine();
        updateQuery = String.format("UPDATE Items SET description = '%s' WHERE itemName = '%s';", newDesc, itemName);
        break;

    case 4:
        System.out.print("Enter new ingredients: ");
        String newIngredients = in.readLine();
        updateQuery = String.format("UPDATE Items SET ingredients = '%s' WHERE itemName = '%s';", newIngredients, itemName);
        break;

    case 5:
        System.out.print("Enter new name: ");
        String newName = in.readLine();
        String checkQuery = String.format("SELECT * FROM Items WHERE itemName = '%s';", newName);
        if (esql.executeQuery(checkQuery) > 0) {
            System.out.println("Item name already taken! Returning to menu.");
            break;
        }
        updateQuery = String.format("UPDATE Items SET itemName = '%s' WHERE itemName = '%s';", newName, itemName);
        break;

    default:
        System.out.println("Invalid choice! Returning to menu.");
        break;
}
```

These queries for updating items are all about updating the menu by choosing to update one of its attributes. For example we give the user the choice to set the price, type, description, ingredients etc. We want to implement this with the updateMenu function which only takes the esql object as a parameter. As with our previous examples, the SELECT queries are being used to check for the existence of a certain Item, and the UPDATE is being used once we know the item exists.

If you did any extra credit, provide screenshots and/or code snippets. Explain how you implemented the extra credit. (triggers/stored procedures, performance tuning, etc)

```sql
-- Many of our queries need to find the Users' login, hence why we have an index for it.
DROP INDEX IF EXISTS User_Login_Index;
CREATE UNIQUE INDEX User_Login_Index ON Users (login);

-- The query we use for logging in will be used very often, so we made an index for retrieving the login and password.
DROP INDEX IF EXISTS User_Login_Password_Index;
CREATE INDEX User_Login_Password_Index ON Users (login, password);

-- This index is for speeding up queries that filter the menu based on a specific item type.
DROP INDEX IF EXISTS Type_Of_Item_Index;
CREATE INDEX Type_Of_Item_Index ON Items (typeOfItem);

-- We frequently access orders by their order id to either update order status or creating a new, unique order id.
DROP INDEX IF EXISTS Order_ID_Index;
CREATE INDEX Order_ID_Index ON ItemsInOrder (orderID);

-- Item name is used frequently for when the user wants to order an item.
DROP INDEX IF EXISTS Item_Name_Index;
CREATE INDEX Item_Name_Index ON Items (itemName);
```

```
[rnogh001@xe-10 cs166_project_phase_3]$ source sql/scripts/create_db.sh
NOTICE:  drop cascades to constraint foodorder_login_fkey on table foodorder
DROP TABLE
NOTICE:  drop cascades to constraint itemsinorder_itemname_fkey on table itemsinorder
DROP TABLE
NOTICE:  drop cascades to constraint itemsinorder_orderid_fkey on table itemsinorder
DROP TABLE
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
NOTICE:  index "user_login_index" does not exist, skipping
DROP INDEX
CREATE INDEX
NOTICE:  index "user_login_password_index" does not exist, skipping
DROP INDEX
CREATE INDEX
NOTICE:  index "type_of_item_index" does not exist, skipping
DROP INDEX
CREATE INDEX
NOTICE:  index "order_id_index" does not exist, skipping
DROP INDEX
CREATE INDEX
NOTICE:  index "item_name_index" does not exist, skipping
DROP INDEX
CREATE INDEX
COPY 1000
COPY 27
COPY 1000
COPY 4
COPY 6
```

We made indexes for our queries. We chose these indexes based on which queries we thought would most be used in this database. For example, we know logging in will be one of the most used features, so we created an index for the login and password. Although we didn't notice any major improvements in the runtime of our

queries after adding indexes, we know this isn't due to the indexes being flawed, but rather the database being so small that indexes aren't really necessary. However, we know that if the database were to grow, these indexes would make a noticeable improvement on our queries' runtime.

```
*********************************************************
              User Interface
*********************************************************

Connecting to database...Connection URL: jdbc:postgresql://localhost:26761/rnogh001_project_phase_3_DB

Done
MAIN MENU
---------
1. Create user
2. Log in
9. < EXIT
Please make your choice: 2
Enter login: mfarrears0
Enter password: dshimoni0
Login successful! Welcome, mfarrears0
MAIN MENU
---------
1. View Profile
2. Update Profile
3. View Menu
4. Place Order
5. View Full Order ID History
6. View Past 5 Order IDs
7. View Order Information
8. View Stores
9. Update Order Status
10. Update Menu
.........................
20. Log out
Please make your choice: 2
Enter the login of the profile you would like to update:
lbeldom3
Which part of the profile would you like to update?
1. Favorite Items
2. Phone Number
3. Password
4. Login
5. Role
Enter your choice: 5
Enter new role: manager
Profile updated successfully!
MAIN MENU
---------
1. View Profile
2. Update Profile
3. View Menu
4. Place Order
5. View Full Order ID History
6. View Past 5 Order IDs
7. View Order Information
8. View Stores
9. Update Order Status
10. Update Menu
.........................
20. Log out
Please make your choice: 2
```

```
Enter the login of the profile you would like to update:
lbeldom3
Which part of the profile would you like to update?
1. Favorite Items
2. Phone Number
3. Password
4. Login
5. Role
Enter your choice: 5
Update denied. Cannot demote managers!
```

```java
case 5:
    String checkQuery = String.format("SELECT * FROM Users WHERE login = '%s' AND role = 'manager';", update_login);
    int count2 = esql.executeQuery(checkQuery);
    if (count2 != 0) {
        System.out.println("Update denied. Cannot demote managers!");
        return;
    }
    System.out.print("Enter new role: ");
    newRole = in.readLine();

    if (!(newRole.equals("manager") || newRole.equals("driver") || newRole.equals("customer"))) {
        System.out.println("Not a valid role. Returning to menu.");
        return;
    }
    updateQuery = String.format("UPDATE Users SET role = '%s' WHERE login = '%s';", newRole, update_login);
    esql.executeUpdate(updateQuery);
    System.out.println("Profile updated successfully!");
    break;

default:
    System.out.println("Invalid choice.");
    return;
```

Some Fancy Stuff

One of the fancy features we have is maintaining a hierarchy in our system. This means for example that even if you are the manager, you are not able to have power over other managers. One of the abilities a manager has is to change the role of users. But what if the other user is a manager? Realistically, it would be bad if a rogue manager was able to demote every single user in the system to a customer. The example above shows a manager with login mfarrears0 promoting a customer with login lbeldom3 to a manager. However, when mfarrears0 tries changing lbeldom3's role again, our system prevents this from happening, since now both of them are equals and should not be able to demote each other.

We also hide certain features from the drivers and customers in our menu. This keeps the functions that are strictly for the manager more simple with less cases checking for the role of the logged in user.

```
while (true) {
    System.out.print("Enter Item Name (or type '0' to finish): ");
    String itemName = in.readLine();
    if (itemName.equals("0")) {
        break;
    }

    String itemQuery = String.format("SELECT price FROM Items WHERE itemName = '%s';", itemName);
    List<List<String>> result = esql.executeQueryAndReturnResult(itemQuery);
    if (result.isEmpty()) {
        System.out.println("Item does not exist! Try again.");
        continue;
    }

    System.out.print("Enter Quantity: ");
    int quantity = Integer.parseInt(in.readLine());
    while (quantity <= 0) {
        System.out.println("Invalid quantity entered. Try again!");
        quantity = Integer.parseInt(in.readLine());
    }

    double price = Double.parseDouble(result.get(0).get(0));
    totalPrice += price * quantity;
    itemNames.add(itemName);
    quantities.add(quantity);
}

if (itemNames.isEmpty()) {
    System.out.println("No items selected. Order canceled.");
    return;
}

LocalDateTime now = LocalDateTime.now();
String orderTimestamp = now.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

String orderStatus = "incomplete";


int orderID = 1 + Integer.parseInt(esql.executeQueryAndReturnResult("SELECT MAX(orderID) FROM FoodOrder").get(0).get(0));


String orderQuery = String.format("INSERT INTO FoodOrder (orderID, login, storeID, totalPrice, orderTimestamp, orderStatus) VALUES (%d, '%s', %d, %.2f, '%s', '%s');", orderID, login, storeID, totalPrice, orderTimestamp, orderStatus);
esql.executeUpdate(orderQuery);

// Insert each item into ItemsInOrder table
for (int i = 0; i < itemNames.size(); i++) {
    String insertItemQuery = String.format(
        "INSERT INTO ItemsInOrder (orderID, itemName, quantity) VALUES (%d, '%s', %d);",
        orderID, itemNames.get(i), quantities.get(i));
    esql.executeUpdate(insertItemQuery);
}

System.out.println("Order placed successfully! Total Price: $" + totalPrice);
```

Good User Interface

Our system is also pretty user friendly. We implemented a lot of error checking, whether that be limiting the roles a user inputs to “customer”, “driver”, and “manager”, or preventing a manager from creating a new menu item that already exists on the menu. A good example of our error checking is shown above. The code above is able to limit a user from entering 0 or less as the quantity for an item they’re ordering. Our system will keep prompting them for a quantity until they enter a valid one. Another user friendly feature we have is allowing the user to go back or return to the menu after they begin using some feature. This is good for when the user presses on a feature on accident.

Problems/Findings

Include problems/findings you encountered while working on the project (1-2 paragraphs max)

-The functions were not designed to account for the role and login of the current user, which made it difficult to impose restrictions on using certain queries. Our workaround was to modify the functions' headers by adding login and role parameters.

-Dealing with hierarchical issues in our system, such as preventing a manager from abusing their power. Our workaround was adding more sophisticated conditions that check to see the role of the user that a manager is trying to manipulate.

Contributions
Include descriptions of what each member worked on (1 paragraph max)

Ryan: Functions having to do with viewing/modifying customer data and menus.
Nashwaan: Order adding and menu viewing functions and querying (update, insert into)