NYU, Tandon School of Engineering

September 24, 2024

CS-GY 6083

Principles of Database Systems
Section A, Fall 2024

Project Report

Submitted by:

Khwaab Thareja | N15911999 | kt3180

Ansh Harjai | N14452996 | ah7163

Neeha Rathna Janjanam | N10968553 | nj2330

Guided By: Prof Phyllis Frankl

Languages and Frameworks Used

- Backend: **Python using the Flask framework**, chosen for its lightweight nature and ease of integration with SQL databases.
- Database: **MySQL**, selected for its robust relational database capabilities and efficient query execution.
- Frontend: **HTML and CSS** for the user interface, with **JavaScript** to provide dynamic interaction where required.
- Security: Passwords are hashed using **SHA-256** with a **salted hash** for enhanced security, preventing direct storage of sensitive data.

Main Queries and Flow for Features

1. Login & User Session Handling (Neeha Rathna)

Logic/ Flow:

1. **LOGIN**
    - One cannot enter the website to view any other pages until he/she hasn't logged in successfully (i.e. session exists)
    - Once clicked on the Login button, it accepts user credentials (username and password)
    - Hashes the input password with the stored salt.
    - Compare the hashed password with the database record.
    - If valid, initializes a user session with details including username, roles and name of the user (first name and last name). The website is redirected to the dashboard page.
    - If not valid, it prompts the user to re-enter credentials and NO user session is initialized.
2. **REGISTER**
    - Fetches all the valid roles from Role Table

- Prompts the user to enter details including Username, Password, First Name, Last Name, Email, Multiple Phone numbers and Multiple Roles.
- Appropriate form validations are set and is checked once the form is submitted, if any of the fail, an alert message is shown -
    - To ensure all form fields are filled, every field is to be “required”
    - The validity of phone number is also checked (should have country code and phone number like +19296784864)
    - Since the email and phone numbers should be unique in the database for valid users, I have added constraints in the database, mentioned in Query section.
    - A combination of roles is only valid, this is done by a trigger whose logic is mentioned in the Query section.
- On submit, the data is inserted into Person, PersonPhone and Act tables in the order of the Foreign Key constraints.

3. **LOGOUT**
    - This option is only visible to logged in users.
    - Once clicked on Logout, the session is cleared and the website is redirected to the homepage/ index page with options to register or login.

Query:

1. **LOGIN**

- To fetch the person details with username, password and roles

```
SELECT p.*, GROUP_CONCAT(r.roleID) as roles

FROM Person p

LEFT JOIN Act a ON p.username = a.username

LEFT JOIN Role as r ON a.roleID = r.roleID

WHERE p.username = “john_s”

GROUP BY p.username
```

2. **REGISTER**

- To fill in the valid roles in the form

```
SELECT roleID FROM Role
```

- (**Additional**) Trigger added to the database to handle the validity of combination of roles, the reasons of such combinations is also mentioned in the query

```
-- Role combination constraints

-- 1. Staff CANNOT be a volunteer, conflict of interest in order
supervision and delivery

-- 2. Staff CANNOT Be client, Creates conflict of interest if staff can
create orders for themselves

-- 3. Staff CANNOT Be donor, to maintain transparency in donation process
and item valuation and acceptance

-- 4. Volunteer CANNOT be a client or donor, conflict of interest in
service delivery

-- 5. Client CAN be a donor as it promotes community involvement



DELIMITER //



CREATE TRIGGER check_role_combinations

BEFORE INSERT ON Act

FOR EACH ROW

BEGIN

    DECLARE user_roles VARCHAR(100);
```

```
    -- Get existing roles for the user
    SELECT GROUP_CONCAT(roleID) INTO user_roles
    FROM Act
    WHERE userName = NEW.userName;
```

```
   -- Check staff combinations
    IF NEW.roleID = 'staff' AND
       (user_roles LIKE '%volunteer%' OR
        user_roles LIKE '%client%' OR
        user_roles LIKE '%donor%')
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Staff members maintain operational oversight
and cannot hold other roles to prevent conflicts of interest';
    END IF;


    -- Check if user already has staff role
    IF user_roles LIKE '%staff%' AND
       (NEW.roleID = 'volunteer' OR
        NEW.roleID = 'client' OR
        NEW.roleID = 'donor')
```

```
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Staff members cannot take on additional roles
to maintain organizational integrity';
    END IF;
```

```
  -- Check volunteer combinations
    IF NEW.roleID = 'volunteer' AND
       (user_roles LIKE '%client%' OR
        user_roles LIKE '%donor%')
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'A volunteer cannot be a client or donor as it
creates a conflict of interest in service delivery and donation handling';
    END IF;


    -- Check if user has volunteer role when trying to be client or donor
    IF (NEW.roleID = 'client' OR NEW.roleID = 'donor') AND
       user_roles LIKE '%volunteer%'
    THEN
        SIGNAL SQLSTATE '45000'
```

```
        SET MESSAGE_TEXT = 'A volunteer cannot take on client or donor
roles to maintain impartial service delivery';

    END IF;

END //



DELIMITER ;
```

- (**Additional**) Constraints for Unique Phone and Email (Highlighted with Yellow

```
CREATE TABLE Person (

   userName VARCHAR(50) NOT NULL,

   userName VARCHAR(50) NOT NULL,

   password VARCHAR(100) NOT NULL,

   password VARCHAR(100) NOT NULL,

   fname VARCHAR(50) NOT NULL,

   fname VARCHAR(50) NOT NULL,

   lname VARCHAR(50) NOT NULL,

   lname VARCHAR(50) NOT NULL,

   email VARCHAR(100) NOT NULL,

   email VARCHAR(100) NOT NULL UNIQUE,

   PRIMARY KEY (userName)

   PRIMARY KEY (userName)

);
```

```
CREATE TABLE PersonPhone (
   userName VARCHAR(50) NOT NULL,
   userName VARCHAR(50) NOT NULL,
   phone VARCHAR(20) NOT NULL,
   phone VARCHAR(20) NOT NULL UNIQUE,
   PRIMARY KEY (userName, phone),
   PRIMARY KEY (userName, phone),
   FOREIGN KEY (userName) REFERENCES Person(userName)
   FOREIGN KEY (userName) REFERENCES Person(userName)
);
```

- Insert into Person Table

```
INSERT INTO Person(username, password, fname, lname, email)
VALUES ("john_s", "john_s", "John", "Smith", "john.s@welcome.com")
```

- Insert into PersonPhone Table

```
INSERT INTO PersonPhone(username, phone) VALUES ("john_s", "+15551234567")
INSERT INTO PersonPhone(username, phone) VALUES ("john_s", "+15551234568")
```

- Insert into Act Table

```
INSERT INTO Act(username, roleID) VALUES ("emma_p", "client")

INSERT INTO Act(username, roleID) VALUES ("emma_p", "donor")
```

3. **LOGOUT**

- There are no SQL queries for it, the session is only cleared.

---

2. Find Single Item (Khwaab)

Logic:

- The application asks the user to input an `item_id`.
- The `item_id` is the key used to query the database for item details and locations.

Query:

Execute a query (`piece_check_query`) on the Piece table to count the number of pieces associated with the entered `item_id`:

```
# Check if the item has pieces

        piece_check_query = "SELECT COUNT(*) AS piece_count FROM Piece
WHERE ItemID = %s"
```

Based on the result:

- If `piece_count > 0`: Proceed to fetch the storage locations of all pieces.
- If `piece_count == 0`: Notify the user that the item has no pieces and return no location details.

```
# Check if no pieces then fetch basic item details.

SELECT ItemID, iDescription FROM Item WHERE ItemID = %s
```

If the item has pieces, executed a query to fetch:

`Item details (iDescription).`

`Piece details (pDescription, roomNum, shelfNum).`

`Location details (shelf, shelfDescription).`

```
# If pieces exist, fetch item and piece details

                SELECT

                    i.ItemID,

                    i.iDescription,

                    p.pDescription,

                    p.roomNum,

                    p.shelfNum,

                    l.shelf,

                    l.shelfDescription

                FROM Item i

                LEFT JOIN Piece p ON i.ItemID = p.ItemID

                LEFT JOIN Location l ON p.roomNum = l.roomNum

                AND p.shelfNum = l.shelfNum

                WHERE i.ItemID = %s
```

Flow:

1. User enters the `item_id`.The application prompts the user to input an `item_id`.
2. The entered `item_id` is validated for correct format and non-emptiness.
3. A query checks the number of pieces for the given `item_id` in the Piece table
4. The application proceeds to fetch the storage locations of all pieces.
5. Inform the user that the item has no associated pieces.
6. Execute a query to retrieve item, piece, and location details.
7. Show all storage locations, including room numbers, shelf numbers, and descriptions.
8. Display an error message and prompt the user to re-enter a valid `item_id`.
9. Implemented error handling to prevent application failures.

---

3. Find Order Items (Ansh)

Logic:

- **User Authentication**: Verify the user is logged in and authorized.
- **Retrieve Order ID**: Extract OrderID from the form.
- **Query Database**: Fetch item and location details for the given OrderID using joins across relevant tables.
- **Render Data**: Pass query results to find_orders.html for display.

Query:

```
SELECT
    i.ItemID,
    i.iDescription AS ItemDescription,
    p.pieceNum,
    p.pDescription AS PieceDescription,
    p.roomNum,
    p.shelfNum,
    l.shelf AS ShelfName,
    l.shelfDescription AS ShelfDescription
FROM
    ItemIn ii
JOIN
    Item i ON ii.ItemID = i.ItemID
```

```
LEFT JOIN
    Piece p ON i.ItemID = p.ItemID
LEFT JOIN
    Location l ON p.roomNum = l.roomNum AND p.shelfNum = l.shelfNum
WHERE
    ii.orderID = 1;
```

Flow:

1. **Authenticate**: Ensure the user has permission.
2. **Input Validation**: Retrieve and validate OrderID from the form.
3. **Execute Query:** Use a parameterized query to fetch order details securely.
4. **Render Response:** Display results in find_orders.html or an error message if no data is found.
5. **Clean Up:** Close the database connection in a finally block.

---

4. Accept Donation (Khwaab)

Logic:

- Before allowing access to the “Accept Donation” page, the application verifies whether the logged-in user is a staff member.
- The verification is done by checking the user's role in the database.
- Once verified as a staff member, the application prompts for the donor’s ID.
- A query is executed to check if the donor is registered in the system.
- If the donor is not registered, the application informs the user that the donor is invalid.
- If the donor is a valid user it prompts the staff member to input the following donor details and donor information as mentioned below
    - Donated item information: item name, description, main category, sub-category, condition (isNew or used).
    - Storage details: room number, shelf number, or any other storage details for storing the item.
    - donorID and associated metadata like first name, last name, etc.
- Check if the item_id is unique by querying the `Item` table.
- If the item_id is not provided by the user, use an **auto-increment mechanism** in the database to generate a new item_id.

- Insert the details into the respective tables:
- Insert item details into the `Item` table.
- Insert donation details into the `DonatedBy` table, linking it to the donor.
- Insert storage details into the Location table or update existing entries if the item already exists.
- Confirm the successful addition of the donation by displaying a message to the staff and display the database entries for validation.

Queries:

```
#query is executed to check if the donor is registered in the system.

SELECT userName FROM Person WHERE userName = %s

SELECT roleID FROM Act WHERE userName = %s AND roleID = 'donor'
```

```
#If donor is registered in system then fetch below details

SELECT CONCAT(fname, ' ', lname) AS fullName FROM Person WHERE userName = %s,

#check if category and subcategory combination exist

SELECT 1 FROM Category WHERE mainCategory = %s AND subCategory = %s

#insert items into item table

INSERT INTO Item (iDescription, photo, color, isNew, hasPieces, material,
mainCategory, subCategory) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)

""", (description, photo_data, colors[idx], is_news[idx] == 'true',
has_pieces_list[idx] == 'true', materials[idx], main_categories[idx],
sub_categories[idx]))

#fetch the last recent ItemId

SELECT LAST_INSERT_ID() AS ItemID
```

```
#insert into DonatedBy Table the items donated by that donor.

INSERT INTO DonatedBy (ItemID, userName, donateDate) VALUES (%s, %s, NOW())



#check if roomNum and Shelfnum already exists

SELECT 1 FROM Location WHERE roomNum = %s AND shelfNum = %s



#Insert the roomNum and ShelfNum into Location table if doesn't exist already

INSERT INTO Location (roomNum, shelfNum, shelf, shelfDescription)

                                    VALUES (%s, %s, %s, %s)

                                """, (room_nums[j], shelf_nums[j],
f"Shelf-{shelf_nums[j]}", "Auto-created location"))



#insert pieces into piece table if the item has pieces

INSERT INTO Piece (ItemID, pieceNum, pDescription, length, width, height, roomNum,
shelfNum, pNotes)

                                VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)

                            """, (item_id, j + 1, p_description, lengths[j], widths[j],
heights[j], room_nums[j], shelf_nums[j], p_notes[j]))
```

```
#successful addition of the donation by displaying a message to the staff and
displaying the database entries for validation.



SELECT d.*, i.iDescription, i.mainCategory, i.subCategory, p.fname,
p.lname, i.isNew FROM DonatedBy d JOIN Item i ON d.ItemID = i.ItemID
JOIN Person p ON d.userName = p.userName ORDER BY d.donateDate DESC,
i.ItemID DESC
```

Flow:

1. When a user tries to access the “Accept Donation” page, the application validates that the user is a staff member.
2. If the user is not a staff member, deny access with an appropriate error message.
3. The application prompts the staff to enter the donor’s donorID.
4. A query checks if the entered donorID exists in the Donors table:
5. If the donor is not registered, the application displays an error message, halting further operations.
6. The application prompts the staff to input the details of the donated items, including:
    - Item description (`iDescription`).
    - Main category and sub-category.
    - Condition (`isNew`).
    - Storage locations (room and shelf numbers).
7. If an item_id is provided, the application validates its uniqueness by checking against the Item table.
8. If an item_id is not provided, the system generates a new item_id using the database’s auto-increment feature and thus, ensuring that every new item automatically gets a unique item_id assigned by the database.
9. Since users cannot manually assign item_ids, the database ensures no duplicate IDs are created.
10. The application inserts the item details into the Item table.
11. The donation details are added to the DonatedBy table:
12. Storage details are added or updated in the Location table:
13. The application displays a success message to the user, confirming that the donation has been recorded.

14. The new database entries are shown for validation purposes, displaying the DonatedBy, Item, and Location tables.
15. Ensure that the donated item is now available for retrieval using the "Find Single Item" feature.
16. Test the system by querying the database to confirm the item appears in relevant searches.

---

5. Start an Order (Khwaab)

Logic:

- The system checks the role of the logged-in user to confirm they are a staff member.
- Only staff members can access the “Start an Order” page or perform order-related operations.
- The system prompts the staff to enter the username of the client for whom the order is being created.
- A query checks if the provided username exists in the `Client` table and ensures it does not match the username of the logged-in staff member:
- If the username is invalid or matches the staff’s username, the system prohibits the order creation and displays an error message.
- Once the client username is validated, the system inserts a new record into the `Ordered` table:
- The `client` is the provided username, and `supervisor` is the logged-in staff member’s username.
- The database assigns an `orderID` using an auto-increment mechanism.
- After inserting the new order, the system retrieves the `orderID` of the newly created entry using the query:
- An initial entry is inserted into the `Delivered` table to track the status of the order:
  sql
  Copy code
- The logged-in staff member is recorded as the user who initiated the order.
- The `orderID` is saved in a session variable for subsequent operations or updates related to the order.

Query:

```
# Create a new order
                    INSERT INTO Ordered (client, supervisor, orderDate)
                    VALUES (%s, %s, NOW())
```

```
# Get the newly created order ID
                ("SELECT LAST_INSERT_ID() AS orderID")
```

```
#Insert an initial status entry in the Delivered table
                cursor.execute("""
                    INSERT INTO Delivered (userName, orderID, status,
date)
                    VALUES (%s, %s, 'InProgress', NOW())
                """, (session['username'], order_id))
                conn.commit()
```

Flow:

- Verify the logged-in user is a staff member; deny access and display an error message if they are not.
- Prompt staff to enter the client's username and validate it against the `Client` table; halt the process if the username is invalid or matches the staff's username.
- Create a new entry in the `Ordered` table with the client's username, supervisor's username, and current date.
- Use the query `SELECT LAST_INSERT_ID()` to get the auto-generated `orderID` of the new order.
- Add an entry to the `Delivered` table with the logged-in user, `orderID`, "InProgress" status, and current date.
- Store the retrieved `orderID` in a session variable for tracking the current order.
- Confirm successful creation of the order, showing the `orderID` and updated details of the `Ordered` and `Delivered` tables for verification.

---

6. Add to Current Order (Khwaab)

Logic:

- Displays available items by category.
- Allows the user to select items and add them to an order.
- Show details of the current order, including:
    - Order Date, Order Notes, Supervisor, Client, and Order ID.

- Retrieve these details using the session-stored `orderID` and the `Ordered` table:
  Sql
- Query the database to retrieve all distinct `mainCategory` values for display in a dropdown menu
- When a user selects a category, fetch the associated subcategories
- Validate the category-subcategory relationship to ensure valid selections:
  sql
- Fetch all items belonging to the selected category and subcategory that are not already part of any order
- When a user selects an item, check if the item is already associated with any order:
  Sql.
- If the item is not part of any order, add it to the current order using the session-stored `orderID`:
- Once the item is added to the order, its availability is updated in the system, ensuring it is excluded from future queries for available items in the same category.
- Display the updated details of the current order.
- Confirm that the items added to the current order are no longer available when the same category is searched again.

Queries:

```
# Fetch categories for dropdown
            cursor.execute("SELECT DISTINCT mainCategory FROM Category")
```

```
cursor.execute("""
                SELECT subCategory FROM Category WHERE mainCategory = %s
            """, (main_category,))
```

```
# Validate category-subcategory relationship
                with conn.cursor() as cursor:
                    cursor.execute(
                        """
                        SELECT 1
                        FROM Category
                        WHERE mainCategory = %s AND subCategory = %s
                        LIMIT 1
                        """,
                        (selected_category, selected_subcategory)
                    )
```

```
# Fetch items for the category and subcategory,
                # excluding items already in any order
                with conn.cursor() as cursor:
                    cursor.execute(
                        """
                        SELECT i.ItemID, i.iDescription
                        FROM Item i
                        LEFT JOIN ItemIn ii ON i.ItemID = ii.ItemID
                        WHERE ii.orderID IS NULL
                        AND i.mainCategory = %s
                        AND i.subCategory = %s
                        """,
                        (selected_category, selected_subcategory)
                    )
```

```
# Fetch subcategories for the selected category
                with conn.cursor() as cursor:
                    cursor.execute(
                        """
                        SELECT subCategory
                        FROM Category
                        WHERE mainCategory = %s
                        """,
                        (selected_category,)
                    )
```

```
# Check if the item is already in any order
                    cursor.execute(
                        """
                        SELECT ii.orderID
                        FROM ItemIn ii
                        WHERE ii.ItemID = %s
                        LIMIT 1
                        """,
                        (item_id,)
```

```
                    )
```

```
# Insert the item into the current order
                    cursor.execute(
                        """
                        INSERT INTO ItemIn (orderID, ItemID)
                        VALUES (%s, %s)
                        """,
                        (session['orderID'], item_id)
                    )
```

Flow:

1.Show details of the current order, including Order Date, Notes, Supervisor, Client, and Order ID.

2. Populate a dropdown menu with `mainCategory` values and dynamically display associated subcategories.

3.Retrieve items in the selected category and subcategory that are not already part of an order.

4. Allow the user to select an item and add it to the current order after verifying it is not already ordered.

5.Mark the added item as unavailable for future queries in the same category or subcategory.

6.Display the updated order details and confirm that added items are no longer available.

7.Provide a confirmation message and display the updated order summary.

---

7. Prepare Order (Ansh)

Logic:

- **Method Check**: Handle POST requests to prepare orders, render the default page for GET.
- **Validate Input**: Retrieve and validate orderID from the form.
- **Fetch Order**: Query the Ordered table for details using orderID. Flash an error if no orders are found.
- **Update Order**:
    - Move pieces to the "Holding Area" (room 4, shelf 3) with delivery notes.
    - Mark items as no longer new.
- **Commit Changes**: Save updates and notify the user of success.
- **Error Handling**: Display errors for invalid input or exceptions, and ensure resources are closed.

Query:

1. **Fetch Order Details.**

```
a. SELECT o.orderID, o.orderDate, o.orderNotes, o.client, o.supervisor
b.                    FROM Ordered o
c.                    WHERE o.orderID = 1;
```

2. **Update Pieces locations to holding area.**

```
a. UPDATE Piece p
b.                    JOIN ItemIn ii ON p.ItemID = ii.ItemID
c.                    SET p.roomNum = 4, p.shelfNum = 3, p.pNotes = 'Ready
   for delivery'
d.                    WHERE ii.orderID = 1;
```

3. **Mark Items as not new.**

```
UPDATE Item i
```

```
                    JOIN ItemIn ii ON i.ItemID = ii.ItemID

                    SET i.isNew = FALSE

                    WHERE ii.orderID = 1;
```

Flow:

1. **Request Type**:
    - If GET: Render the prepare_order.html template.
    - If POST: Validate and process the orderID input.
2. **Input Validation**:
    - Check if orderID is provided.
    - If missing, show an error and redirect to the preparation page.
3. **Fetch Order**:
    - Use orderID to fetch details from the Ordered table.
    - If no orders match, flash an error and redirect.
4. **Update Database**:
    - Update pieces in the order to the "Holding Area" with delivery notes.
    - Mark items as no longer new.
5. **Commit and Notify**:
    - Commit changes to the database.
    - Flash a success message if updates complete successfully.
6. **Error Handling**:
    - Catch exceptions and display an error message.
    - Ensure database connections and cursors are properly closed in a finally block.

---

8. User's Tasks (Neeha Rathna)

Logic/ Flow:

- Displays all orders associated with the current user by giving a personalized view. This is done by using the roles key in the user session.
- A user can have multiple roles, so there is an option to choose how you want to view the orders.
    - Say a person is both a donor and a client, there will be 2 options in the dropdown to choose from.
    - For a client view, the user can see only the orders bought by the logged in user. Details visible include Order ID, Order Date, Order Notes, Supervisor Name, Volunteer(s) Names, Delivery Status and the number of Items in the order.

- For a donor view, the user can see only the orders donated by the logged in user. Details visible include Order ID, Order Date, Donated Date and Delivery Status.
- For a staff view, all the orders are visible but won't be able to update other staff orders. Details visible include Order ID, Order Date, Order Notes, Supervisor Name, Client Name, Volunteer(s) Names, Delivery Status, No of Items and Item Details.
- For a volunteer view, the user can see all the orders which they deliver. Details visible include Order ID, Order Date, Supervisor Name and Item Details.
- There are separate queries called for each of these views, mentioned in Query section

- All the orders are ORDERED descending order by Order Date to see the latest/ most recent orders at the top, this is handled in the query itself, not on the Frontend,

Query:

1. Client View Query

```
SELECT o.orderID, o.orderDate, o.orderNotes,

                        p.fname as SupervisorFName, p.lname as
SupervisorLName,

                        GROUP_CONCAT(DISTINCT CONCAT(v.fname, ' ',
v.lname)) as volunteers,

                        CASE

                            WHEN SUM(CASE WHEN d2.status = 'Pending'
THEN 1 ELSE 0 END) > 0 THEN 'Pending'

                            WHEN SUM(CASE WHEN d2.status = 'InProgress'
THEN 1 ELSE 0 END) > 0 THEN 'InProgress'

                            WHEN COUNT(d2.status) = SUM(CASE WHEN
d2.status = 'Delivered' THEN 1 ELSE 0 END) THEN 'Delivered'

                            ELSE 'Pending'

                        END as delivery_status,

                        COUNT(DISTINCT i.ItemID) as item_count
```

```
                    FROM Ordered o

                    LEFT JOIN Person p ON o.supervisor = p.userName

                    LEFT JOIN Delivered d ON o.orderID = d.orderID

                    LEFT JOIN Person v ON d.userName = v.userName

                    LEFT JOIN ItemIn i ON o.orderID = i.orderID

                    LEFT JOIN Delivered d2 ON o.orderID = d2.orderID

                    WHERE o.client = “emma_p”

                    GROUP BY o.orderID, o.orderDate, o.orderNotes,
p.fname, p.lname

                    ORDER BY o.orderDate DESC, orderID DESC
```

2. Donor View Query

```
SELECT DISTINCT o.orderID, o.orderDate,

                        i.iDescription, db.donateDate,

                        CASE

                            WHEN SUM(CASE WHEN d2.status = 'Pending'
THEN 1 ELSE 0 END) > 0 THEN 'Pending'

                            WHEN SUM(CASE WHEN d2.status = 'InProgress'
THEN 1 ELSE 0 END) > 0 THEN 'InProgress'

                            WHEN COUNT(d2.status) = SUM(CASE WHEN
d2.status = 'Delivered' THEN 1 ELSE 0 END) THEN 'Delivered'

                            ELSE 'Pending'

                        END as order_status
```

```
                    FROM DonatedBy db

                    JOIN Item i ON db.ItemID = i.ItemID

                    JOIN ItemIn ii ON i.ItemID = ii.ItemID

                    JOIN Ordered o ON ii.orderID = o.orderID

                    LEFT JOIN Delivered d ON o.orderID = d.orderID

                    LEFT JOIN Delivered d2 ON o.orderID = d2.orderID

                    WHERE db.userName = “emma_p”

                    GROUP BY o.orderID, o.orderDate, i.iDescription,
db.donateDate

                    ORDER BY o.orderDate DESC, orderID DESC
```

3. Staff View Query

```
SELECT o.orderID, o.orderDate, o.orderNotes, o.supervisor,

               pc.fname as ClientFName, pc.lname as ClientLName,

               GROUP_CONCAT(DISTINCT CONCAT(v.fname, ' ', v.lname)) as
volunteers,

               GROUP_CONCAT(DISTINCT v.userName) as volunteer_usernames,

                CASE

                                WHEN SUM(CASE WHEN d2.status = 'Pending'
THEN 1 ELSE 0 END) > 0 THEN 'Pending'

                                WHEN SUM(CASE WHEN d2.status = 'InProgress'
THEN 1 ELSE 0 END) > 0 THEN 'InProgress'
```

```
                                WHEN COUNT(d2.status) = SUM(CASE WHEN
d2.status = 'Delivered' THEN 1 ELSE 0 END) THEN 'Delivered'

                                ELSE 'Pending'

                            END as delivery_status,

                COUNT(DISTINCT i.ItemID) as item_count,

                GROUP_CONCAT(DISTINCT CONCAT(i.iDescription, ' (',
i.color, ')')) as item_details

                    FROM Ordered o

                JOIN Person pc ON o.client = pc.userName

                LEFT JOIN Delivered d ON o.orderID = d.orderID

                LEFT JOIN Person v ON d.userName = v.userName

                LEFT JOIN ItemIn ii ON o.orderID = ii.orderID

                LEFT JOIN Item i ON ii.ItemID = i.ItemID

                LEFT JOIN Delivered d2 ON o.orderID = d2.orderID

                GROUP BY o.orderID, o.orderDate, o.orderNotes,
o.supervisor, pc.fname, pc.lname

                ORDER BY o.orderDate DESC, o.orderID DESC
```

4. Volunteer View Query

```
SELECT o.orderID, o.orderDate, o.supervisor,

                p.fname as ClientFName, p.lname as ClientLName,

                ps.fname as SupervisorFName, ps.lname as SupervisorLName,

                CASE
```

```
                            WHEN SUM(CASE WHEN d2.status = 'Pending'
THEN 1 ELSE 0 END) > 0 THEN 'Pending'

                            WHEN SUM(CASE WHEN d2.status = 'InProgress'
THEN 1 ELSE 0 END) > 0 THEN 'InProgress'

                            WHEN COUNT(d2.status) = SUM(CASE WHEN
d2.status = 'Delivered' THEN 1 ELSE 0 END) THEN 'Delivered'

                            ELSE 'Pending'

                        END as delivery_status,

                COUNT(DISTINCT i.ItemID) as item_count,

                GROUP_CONCAT(DISTINCT d.userName) as volunteer_usernames,

                GROUP_CONCAT(DISTINCT CONCAT(i.iDescription, ' (',
i.color, ')')) as item_details

                FROM Delivered d

                JOIN Ordered o ON d.orderID = o.orderID

                JOIN Person p ON o.client = p.userName

                JOIN Person ps ON o.supervisor = ps.userName

                LEFT JOIN ItemIn ii ON o.orderID = ii.orderID

                LEFT JOIN Item i ON ii.ItemID = i.ItemID

                LEFT JOIN Delivered d2 ON o.orderID = d2.orderID

                GROUP BY o.orderID, o.orderDate, o.supervisor, p.fname,
p.lname,

                        ps.fname, ps.lname

                ORDER BY o.orderDate DESC, orderID DESC
```

9. Rank System (Ansh)

Logic:

- **Method Check**: Handle POST requests to fetch rankings; render the default page for GET.
- **Validate Input**: Retrieve and validate rank_date from the form.
- **Query Database**: Fetch volunteers' delivery counts since rank_date.
    - Include only users with a Volunteer role.
    - Count deliveries using Delivered table, defaulting to 0 if none exist.
- **Render Data**: Pass results to view_ranking.html for display or show an error message if input is invalid or no data is found.

Query:

```
SELECT
    p.userName,
    p.fname AS FirstName,
    p.lname AS LastName,
    COALESCE(COUNT(d.orderID), 0) AS DeliveryCount
FROM
    Person p
JOIN
    Act a ON p.userName = a.userName
LEFT JOIN
    Delivered d ON p.userName = d.userName
WHERE
    a.roleID = 'Volunteer' AND d.date >= 2024-12-03
GROUP BY
    p.userName
ORDER BY
    DeliveryCount DESC;
```

Flow:

1. **Request Type**:
    - If GET: Render an empty ranking page.
    - If POST: Validate rank_date from the form.
2. **Input Validation**:
    - If rank_date is missing or invalid, show an error and redirect.
3. **Execute Query**:
    - Use the valid rank_date to fetch volunteer rankings by delivery count.
4. **Render Response**:
    - If data is found, display rankings on view_ranking.html.
    - If no results, flash an error message and reload the page.
5. **Clean Up**: Close database connection and cursor.

---

10. Update Enabled (Neeha Rathna)

Logic/ Flow:

- The logged in user role is checked using the roles key in the user session. If the logged in user doesn't have "staff" or "volunteer" in it, the edit icon isn't visible to update the order status for each order in the My Orders tab.
- Even though the edit button is visible, it won't be enabled for those orders which are already delivered, since no one needs to update its status.
- For the editable orders, a popup appears which shows the Order details and since each item can have its individual delivery status, one can update it for each item - dropdown appears for each of them. The user can then click on Submit/ Cancel the changes. An API is called for Submitting and One for retrieving details for each order at Item level.
- Post a successful submit, based on the new status updated, the delivery status on the My Orders page also is updated simultaneously by the /myorders API call.
    - If any one of the items in the order is in "Pending" status, the whole order is in "Pending" status.
    - If any one of the items in the order is in "In Progress" status, the whole order is in "Pending" status.
    - If all items in the order is "Delivered", the whole order is "Delivered".

- This update will only be allowed for supervisors for their own orders, an alert message comes up if a supervisor or volunteer tries to update the status of an order which he/she doesn't supervise or deliver.

Query:

- Getting item level data for each order based on Order ID

```
SELECT o.orderID, o.orderDate, o.orderNotes, o.supervisor,
                      pc.fname as ClientFName, pc.lname as ClientLName,
                      i.ItemID, i.iDescription, i.color, d.status as
item_status,
                      d.userName as assigned_volunteer
                FROM Ordered o
                JOIN Person pc ON o.client = pc.userName
                LEFT JOIN ItemIn ii ON o.orderID = ii.orderID
                LEFT JOIN Item i ON ii.ItemID = i.ItemID
                LEFT JOIN Delivered d ON ii.ItemID = d.ItemID
                WHERE o.orderID = "10"
                ORDER BY i.ItemID
```

- Checking Authorization before submitting the Update on Database Level too

```
SELECT 1 FROM Delivered
                       WHERE ItemID = %s AND userName = %s
                       OR EXISTS (
                           SELECT 1 FROM Ordered
```

```
                        WHERE orderID = "10" AND supervisor = "john_s"

                    )
```

- Update Delivery status and associated date in delivered Table

```
UPDATE Delivered

SET status = "Delivered", date = CURRENT_DATE()

WHERE ItemID = "12"
```

Difficulties Encountered & Lessons Learned

1. Difficulties

- Integrating secure password hashing required understanding of cryptographic functions.
- Handling duplicate items while avoiding redundancy in the schema.
- Role combinations which are valid so that other features aren't impacted
- The orders view for each role would be different, finalization of the attributes and making it visible on UI took a lot of effort and time.
- Implementing role-based access control for staff and volunteers.

2. Lessons Learned

- Importance of Schema Design: Proper schema planning reduces development complexity.
- Security Best Practices: Using prepared statements and hashing ensures robust security.
- Team Collaboration: Effective task allocation and communication are crucial for success.

Team Contributions

- **Khwaab**: Implemented 2, 4, 5, 6
- **Neeha Rathna**: Implemented Project setup, 1, 4(Photo section), 8, 10
- **Ansh**: Implemented 3, 7, 9