Database Organization

Project Report

Member: Liner Zhang

CUG No: 20231003496

IIT No: A20563408

Teacher: Miss. Feng

Date: January 12, 2025

Contents

1 Problem Description — Performance Organization System

1.1 Database Requirements

Design a performance management system that includes multiple tables, such as performance, actors, theaters, users, and tickets, ensuring data integrity, consistency, and proper relational structure between the tables.

1.2 Application Requirements

In this system, users can flexibly query performance information, actor details, remaining ticket information, and make purchases, but they cannot modify system data. Only administrators are allowed to perform operations such as adding, modifying, or deleting performance, actor, and theater information. Additionally, administrators can view and manage all ticketing information.

2 Database Design

2.1 Table Structure

I first designed three separate tables to store the basic information for the performance system, **actors** table, **dramas** table, and **theaters** table. Each table has an ’id” as the primary key, and these ’id’s are auto-incremented at the application layer using the ‘SERIAL‘ data type. Since gender can only be selected as either ”male” or ”female”, I used the ‘ENUM‘ type to enforce this constraint.

Next, I started designing the relationships between these tables. Each performance needs to be associated with a specific drama and venue, so I used the **performances** table to link the two. Each performance also involves different actors, so I created the **performace_actors** table to represent the relationship between performances and actors. Additionally, to record the performance times, I designed the **schedules** table, which functions like a container, organizing each performance in chronological order. At this point, the basic structure for storing performance information is complete.

Considering the ticketing system, I designed the **tickets**, **users**, and **orders** tables, following a similar approach as the previous airplane reservation system. To differentiate between users and administrators, I added a password field and created the **staff** table to distinguish between the two roles, enabling proper system functionality and login identification.

My detailed table structure is shown below.

1. **actors Table**: Stores information about the actors.
    - **actor_id** (Primary Key): A unique identifier for each actor, automatically generated.
    - **actor_name**: The full name of the actor.
    - **gender**: The gender of the actor, ”male” or ”female”.
    - **birthday**: The actor’s birthdate.
    - **biography**: A textual description of the actor’s background and career.
2. **theaters Table**: Stores information about the theaters.
    - **theater_id** (Primary Key): A unique identifier for each theater, automatically generated.
    - **theater_name**: The name of the theater.
    - **address**: The physical address of the theater.
    - **seats**: The total number of seats in the theater.
    - **contact_num**: The contact number for the theater.
3. **dramas Table**: Stores information about the dramas.
    - **drama_id** (Primary Key): A unique identifier for each drama, automatically generated.
    - **drama_name**: The name of the drama.
    - **description**: A textual description of the drama.
    - **duration**: The duration of the drama in minutes.
4. **performances Table**: Stores information about specific performances of dramas in theaters.
    - **performance_id** (Primary Key): A unique identifier for each performance, automatically generated.
    - **drama_id** (Foreign Key): References the **drama_id** in the **dramas** table to identify which drama is being performed.
    - **theater_id** (Foreign Key): References the **theater_id** in the **theaters** table to identify where the performance is being held.
5. **performance_actors Table**: Represents the relationship between performances and actors.
    - **performance_id** (Foreign Key): References the **performance_id** in the **performances** table.

- **role**: The role that the actor plays in the performance.
- **actor_id** (Foreign Key): References the **actor_id** in the **actors** table.

6. **schedules Table**: Stores the schedules for performances in theaters.
   - **theater_id** (Foreign Key): References the **theater_id** in the **theaters** table.
   - **performance_id** (Foreign Key): References the **performance_id** in the **performances** table.
   - **start_time**: The scheduled start time of the performance.
   - **end_time**: The scheduled end time of the performance.
7. **tickets Table**: Stores information about tickets available for performances.
   - **ticket_id** (Primary Key): A unique identifier for each ticket, automatically generated.
   - **performance_id** (Foreign Key): References the **performance_id** in the **performances** table to associate the ticket with a specific performance.
   - **price**: The price of the ticket.
   - **left_tickets**: The number of remaining tickets available for sale.
   - **class**: The class of the ticket, including ”VIP” and ”Basic”.
8. **users Table**: Stores information about users of the system.
   - **user_id** (Primary Key): A unique identifier for each user, automatically generated.
   - **user_name**: The name of the user.
   - **gender**: The gender of the user.
   - **birthday**: The birthdate of the user.
   - **country**: The country where the user resides.
   - **password**: The password for the user's account.
9. **orders Table**: Represents the orders made by users for tickets.
   - **order_id** (Primary Key): A unique identifier for each order, automatically generated.
   - **user_id** (Foreign Key): References the **user_id** in the **users** table.
   - **ticket_id** (Foreign Key): References the **ticket_id** in the **tickets** table.
   - **seat_id**: The seat number for the order.

- **status**: The current status of the order, with a default value of ”pending”. The possible values are ”pending”, ”paid”, or ”cancelled”.

10. **staff Table**: Stores information about staff members managing the system.
    - **work_id** (Primary Key): A unique identifier for each staff member, automatically generated.
    - **work_name**: The name of the staff member.
    - **password**: The password for the staff member’s account.

2.2 ER Model

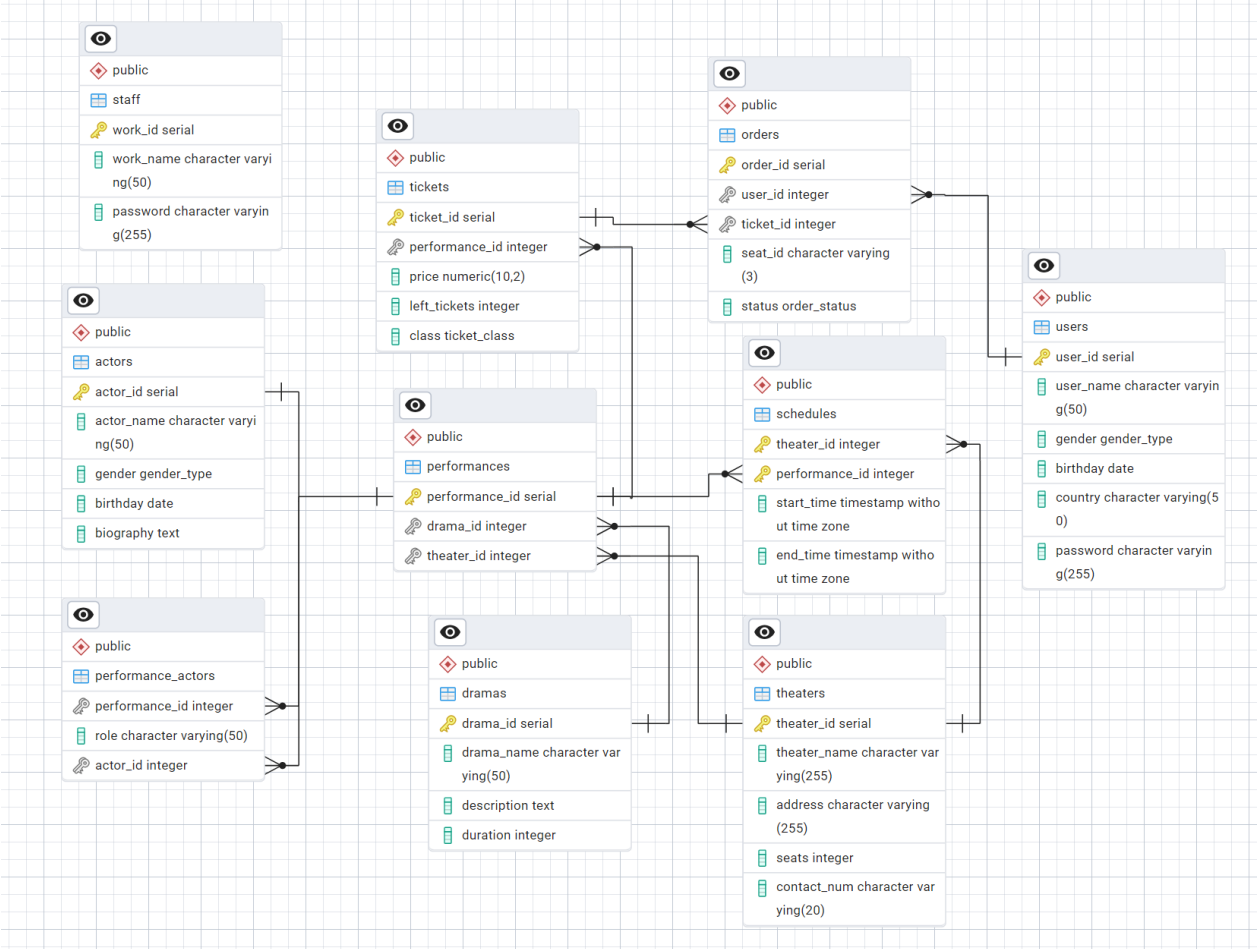

Figure 1: ER Model

3 Function Design

I have divided the program into two main functional modules: one for users and one for staff.

Users can log in and register, view all available information about dramas, actors, and theaters, and choose and purchase tickets based on personal preferences. They can also modify their personal information.

To ensure information security, staff can only log in and cannot register an account. Staff have the authority to modify, add, or delete drama information, actor information, theater information, and performance-related details.

The specific functional flowchart is shown in the figure below. I will omit the UI description and provide a detailed analysis of the database application part of the program.

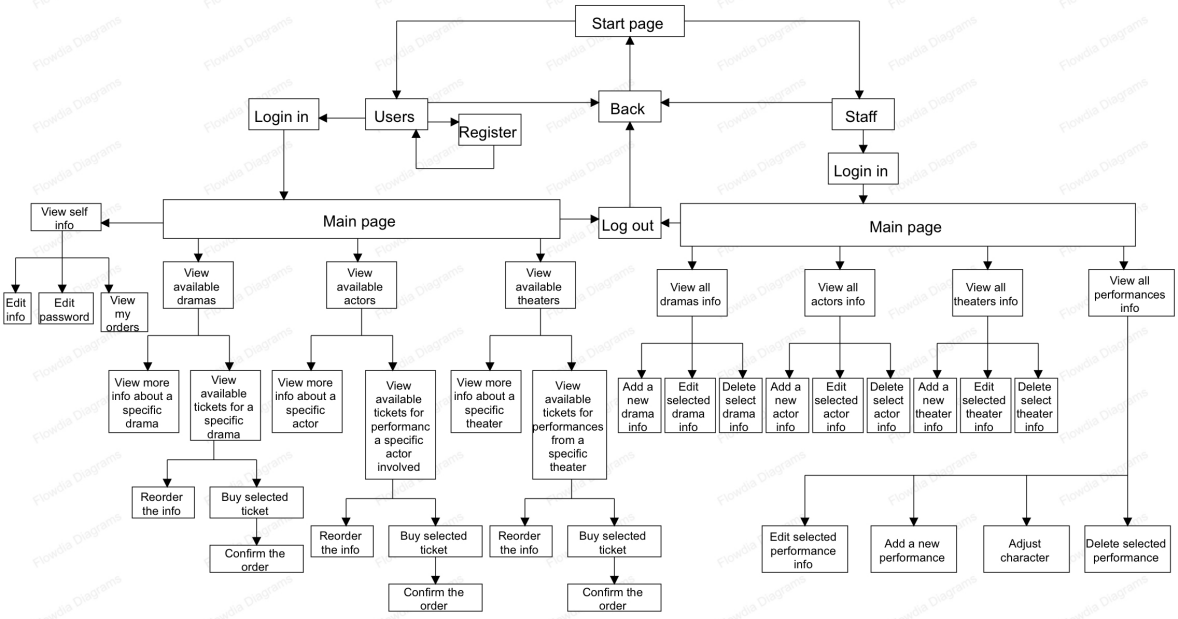

Figure 2: Functional flowchart

3.1 Users Function

3.1.1 Login Process Query

During the login and registration process, the **users** table will be utilized.

Since the **user_id** is set as an auto-incrementing field when the table is created, users only need to input information other than the **user_id**. After performing basic checks on this information, such as verifying whether all required fields are filled, ensuring the format is correct, and checking if

the username already exists, the information will be written to the database using an **INSERT** statement.

Excerpt of User Registration Code

```
existing_user = db.fetch_one("""SELECT user_name
    FROM users WHERE user_name = %s""", (user_name,))
if existing_user:
    show_error_message("""Username already exists""")
else:
    db.execute_query("""
        INSERT INTO users
        (password, user_name, gender, birthday, country)
        VALUES (%s, %s, %s, %s, %s)
        RETURNING user_id""",
        (password, user_name, gender, birthdate, country))
    user_id=db.fetch_one("""SELECT LASTVAL();""")[0]
    show_success_message(f"""Registration successful! Your
       User ID is {user_id}. \nPlease remember your ID for
       the next login.""")
```

During login, the system first uses a **SELECT** statement to retrieve all user information. It first checks whether the **user_id** entered by the user exists in the users table. If it does, the system then compares the entered password with the **password** corresponding to that **user_id** in the database.

Excerpt of User Login Code

```
user = db.fetch_one("""SELECT * FROM users WHERE user_id = %s
    """, (user_id,))
stored_password = user[5]
if password == stored_password:
    success_label = ctk.CTkLabel(window, text="""{}, welcome!
        """.format(user[1]), font=("Arial", 20), text_color="
        green")
    success_label.pack(pady=10)
    window.after(800, lambda: user_dashboard(window,user_id,
        db, pics,pics1,pics2))
else:
    messagebox.showerror("""Login Error""", """Incorrect
        password!\nPlease try again!""")
```

3.1.2 Drama-Based Query

During this process, the **dramas**, **performances**, **tickets**, **orders** tables will be utilized.

First, use a **SELECT** statement to retrieve all the drama information from the database, and then display it to the user combining both images and text.

Excerpt of Search Dramas Code

```
dramas = db.fetch_all("""SELECT * FROM dramas""")
```

When the user selects a particular drama, a **SELECT** statement is used to retrieve the ticket information related to that drama through the relationship between **tickets** and **performances**. Then, using the relationships between **performances** and **schedules**, as well as between **performances** and **theaters**, all the ticket information is comprehensively displayed to the user. Additionally, the information can be sorted and filtered based on the user's desired sorting order and region.

Excerpt of Search Tickets Code

```
region_condition = ""
if region:
    region_condition = f"""AND H.address LIKE %s"""
    region = f"%{region}%"  # Use SQL LIKE pattern
query = f"""
        SELECT T.ticket_id, H.theater_name, S.start_time, S.
            end_time, T.class, T.price, T.left_tickets
        FROM tickets AS T
        JOIN performances AS P ON P.performance_id = T.
            performance_id
        JOIN Theaters AS H ON P.theater_id = H.theater_id
        JOIN schedules AS S ON P.performance_id = S.
            performance_id
        WHERE P.drama_id = %s
        {region_condition}
        ORDER BY {sort_column_map.get(sort_column, 'H.
            theater_name')} {sort_order_sql}
        """
tickets = db.fetch_all(query, (drama_id, region) if region
    else (drama_id,))
```

After the user selects the ticket type, a **SELECT** statement is first used to retrieve all the existing orders of the user to check whether the user has already purchased the ticket, in order to prevent duplicate purchases. Once the user confirms the ticket purchase, an **UPDATE** statement is used to update the remaining **tickets**, decreasing the ticket count by one. Then, an **INSERT** statement is used to add the new order information to **orders**.

Excerpt of Buy Ticket Code

```
db.execute_query(
    """UPDATE tickets
        SET left_tickets = left_tickets - 1
        WHERE ticket_id = %s""",
        (ticket_id,))
seat_id = generate_seat_id(ticket_class)
db.execute_query(
    """INSERT INTO orders(user_id, ticket_id, seat_id, status
       )
        VALUES (%s, %s, %s, 'paid')""",
        (user_id, ticket_id, seat_id))
messagebox.showinfo("Success", f"""Successfully purchased one
    ticket! \nYour seat ID is {seat_id}\nYou can check your
   order in the <My Orders> page.""")
```

3.1.3 Actor-Based Query

During this process, the **actors**, **performances**, **tickets**, **orders** and **performance_actors** tables will be utilized.

When the user wants to query **actors**, a **SELECT** statement is used to retrieve all the actor information, which is then displayed through a combination of images and text.

Excerpt of Search Actors Code

```
actors= db.fetch_all("""SELECT * FROM actors""")
```

When the user selects an actor, a **SELECT** statement retrieves all ticket information for the **performances** the actor has participated in, based on the relationships between **tickets**, **performances**, and **performance_actors**. The ticket details are then displayed using the relationships between **performances**, **schedules**, and **theaters**. Additionally, the information can be sorted and filtered by the user's preferred order and region.

Excerpt of Search Tickets Code

```
region_condition = ""
if region:
    region_condition = f"""AND H.address LIKE %s"""
    region = f"%{region}%"
query = f"""SELECT T.ticket_id, D.drama_name,PA.role,H.
   theater_name, S.start_time, S.end_time, T.class, T.price,
   T.left_tickets
    FROM tickets AS T
    JOIN performances AS P ON P.performance_id = T.
       performance_id
    JOIN theaters AS H ON P.theater_id = H.theater_id
    JOIN schedules AS S ON P.performance_id = S.
       performance_id
    JOIN dramas AS D ON D.drama_id = P.drama_id
    JOIN performance_actors AS PA ON PA.performance_id = P.
       performance_id
    JOIN actors AS A ON A.actor_id = PA.actor_id
    WHERE A.actor_id = %s {region_condition}
    ORDER BY {sort_column_map.get(sort_column, 'H.
       theater_name')} {sort_order_sql}"""
tickets = db.fetch_all(query, (actor_id, region) if region
   else (actor_id,))
```

The database used for subsequent ticket purchases is the same as the one described earlier, so it is omitted.

3.1.4 Theater-Based Query

During this process, the **theaters**, **performances**, **tickets**, **orders** tables will be utilized.

When the user wants to query **theaters**, a **SELECT** statement is used to retrieve all the theater information, which is then displayed through a combination of images and text.

Excerpt of Search Theaters Code

```
theaters=db.fetch_all("""SELECT * FROM theaters""")
```

When the user selects a theater, a **SELECT** statement retrieves all ticket information for performances at that theater, based on the relationships between **theaters**, **performances**, and **tickets**. The details are displayed using the relationships with **dramas** and **schedules**. The information can be sorted and filtered by the user's preferences. Unlike previous sorting processes, the **LIKE** statement is not used for region filtering, as the theater already defines the location.

Excerpt of Search Tickets Code

```
query = f"""SELECT T.ticket_id, D.drama_name, H.theater_name,
    S.start_time, S.end_time, T.class, T.price, T.
   left_tickets
    FROM tickets AS T
    JOIN performances AS P ON P.performance_id = T.
       performance_id
    JOIN theaters AS H ON P.theater_id = H.theater_id
    JOIN schedules AS S ON P.performance_id = S.
       performance_id
    JOIN dramas AS D ON D.drama_id = P.drama_id
    WHERE H.theater_id = %s
    ORDER BY {sort_column_map.get(sort_column, 'H.
       theater_name')} {sort_order_sql}
tickets = db.fetch_all(query, (theater_id,))
```

The database used for subsequent ticket purchases is the same as the one described earlier, so it is omitted.

3.1.5 Personal Information Query

When the user views his or her information, the corresponding data is retrieved from the **users** table based on the **user_id** and displayed.

Excerpt of Search Personal Info Code

```
user_info = db.fetch_one("""SELECT * FROM users WHERE user_id
    =%s""", (user_id,))
```

When the user edits their personal information, the submitted content is briefly reviewed, and the **users** is updated using an **UPDATE** statement.

Excerpt of Edit Personal Info Code

```
db.excute_query("""UPDATE users SET user_name=%s, gender=%s,
    birthday=%s, country=%s WHERE user_id=%s""",
     (new_name, new_gender, new_birthdate, new_country,
        user_id))
```

When the user changes their password, both the new and old passwords must be provided. A **SELECT** statement retrieves the current password for comparison to ensure account security.

Excerpt of Edit Password Code

```
if old_password != current_password:
    messagebox.showerror("Error","""Old password is incorrect
        ! Please try again.""")
elif not new_password:
    messagebox.showerror("Error", """New password cannot be
        empty! Please try again.""")
else:
    db.execute_query("""UPDATE users
    SET password=%s WHERE user_id=%s""", (new_password,
        user_id))
    messagebox.showinfo("Success", """Password changed
        successfully!""")
```

When the user queries all their orders, a **SELECT** statement is used to query the corresponding orders in the **orders** table. Then, based on the relationships with **performances**, **theaters**,

dramas, and **schedules**, the complete order details are output, providing the user with the most comprehensive information.

Excerpt of Search Personal Orders Code

```
query = """
    SELECT orders.order_id, orders.seat_id,
    tickets.price, dramas.drama_name,
    dramas.duration, schedules.start_time,
    schedules.end_time, theaters.theater_name,
    theaters.address
    FROM orders
    RIGHT JOIN tickets ON tickets.ticket_id = orders.
        ticket_id
    RIGHT JOIN performances ON performances.performance_id =
        tickets.performance_id
    RIGHT JOIN dramas ON dramas.drama_id = performances.
        drama_id
    RIGHT JOIN schedules ON schedules.performance_id =
        performances.performance_id
    RIGHT JOIN theaters ON theaters.theater_id = performances
        .theater_id
    WHERE orders.user_id = %s"""
orders = db.fetch_all(query, (user_id,))
```

3.2 Staff Function

3.2.1 Login Process Query

During the login process, the **staff** table will be utilized. Staff can only log in, and the process is the same as the user login process.

Excerpt of Staff Login Code

```
staff = db.fetch_one("""SELECT * FROM staff WHERE work_id = %
    s""", (work_id,))
if staff is None:
    messagebox.showerror("Error", """Invalid Work ID!""")
else:
    stored_password = staff[2]
    if password == stored_password:
        success_label = ctk.CTkLabel(window, text="""{},
            welcome!""".format(staff[1]), font=("Arial", 20),
            text_color="green")
        success_label.pack(pady=10)
        window.after(800, lambda: staff_dashboard(window,
            work_id, db))
    else:
        messagebox.showerror("Error", """Incorrect password!\
            nPlease try again!""")
```

3.2.2 Drama Management Query

The **dramas** table is used to manage the drama information.

When the staff selects the drama section, all the information from the **dramas** table needs to be displayed.

When a new drama is to be added, the staff's input is briefly reviewed, and the information is inserted into the dramas table using an **INSERT** statement.

Excerpt of Drama Add Code

```
query = """INSERT INTO dramas (drama_name, description,
    duration) VALUES (%s, %s, %s)"""
db.execute_query(query, (name_entry, description_entry,
    duration_entry))
```

When the user needs to delete the selected drama, the related foreign key data is deleted

first, and then the corresponding information is removed from the **dramas** table using a **DELETE** statement.

Excerpt of Drama Deletion Code

```
drama_id = treeview.item(selected_item[0])['values'][0]
performance_ids = db.fetch_all("""SELECT performance_id FROM
    performances WHERE drama_id = %s""", (drama_id,))
query3 = """DELETE FROM tickets WHERE performance_id = %s"""
query4 = """DELETE FROM performance_actors WHERE
    performance_id = %s"""
query5 = """DELETE FROM schedules WHERE performance_id = %s
    """
query1 = """DELETE FROM performances WHERE drama_id = %s"""
query2 = """DELETE FROM dramas WHERE drama_id = %s"""
for performance in performance_ids:
    performance_id = performance
    db.execute_query(query5, (performance_id,))
    db.execute_query(query4, (performance_id,))
    db.execute_query(query3, (performance_id,))
db.execute_query(query1, (drama_id,))
db.execute_query(query2, (drama_id,))
```

When the user needs to modify the selected drama, the staff's input is reviewed, and the changes are applied to the **dramas** table using an **UPDATE** statement.

Excerpt of Drama Edit Code

```
query = """UPDATE dramas SET drama_name = %s, description = %
    s, duration = %s WHERE drama_id = %s"""
db.execute_query(query, (drama_name, description, duration,
    drama_id))
```

3.2.3 Actor Management Query

The **actors** table is used to manage the actor information.

When the staff selects the actor section, all the information from the **actors** table needs to be displayed.

When a new actor's information is to be added, the staff's input is briefly reviewed, and the information is inserted into the **actors** table using an **INSERT** statement.

Excerpt of Actor Add Code

```
query = """INSERT INTO actors (actor_name, biography, gender,
    birthday) VALUES (%s, %s, %s,%s)"""
db.execute_query(query, (name_entry, description_entry,
    gender_entry,birthdate_entry))
```

When the staff needs to delete a selected actor, the related foreign key data is deleted first using a **DELETE** statement, followed by the deletion of the actor's information from the **actors** table.

Excerpt of Actor Deletion Code

```
actor_id = treeview.item(selected_item[0])['values'][0]
query1 = """DELETE FROM performance_actors WHERE actor_id = %
    s"""
query2 = """DELETE FROM actors WHERE actor_id = %s"""
db.execute_query(query1, (actor_id,))
db.execute_query(query2, (actor_id,))
```

When the actor's information needs to be modified, an **UPDATE** statement is used.

Excerpt of Actor Edit Code

```
query = """UPDATE actors SET actor_name = %s, gender = %s,
    birthday = %s, biography = %s WHERE actor_id = %s"""
db.execute_query(query, (actor_name, gender, birthdate,
    biography, actor_id))
```

3.2.4 Theater Management Query

The **theaters** table is used to manage the theater information.

When the staff selects the theater section, all the information from the **theaters** table needs to be displayed.

When a new theater's information is to be added, the staff's input is briefly reviewed, and the information is inserted into the **theaters** table using an **INSERT** statement.

Excerpt of Theater Add Code

```
query = """INSERT INTO theaters (theater_name, address, seats
    , contact_num) VALUES (%s, %s, %s, %s)"""
db.execute_query(query, (name_entry, address_entry,
    seats_entry, contact_entry))
```

When the staff needs to delete a selected theater, the related foreign key data is deleted first using a **DELETE** statement, followed by the deletion of the theater's information from the **theaters** table.

Excerpt of Theater Deletion Code

```
theater_id = treeview.item(selected_item[0])['values'][0]
performance_ids = db.fetch_all("""SELECT performance_id FROM
    performances WHERE theater_id = %s""", (theater_id,))
query_schedule = """DELETE FROM schedules WHERE theater_id =
    %s"""
query_ticket = """DELETE FROM tickets WHERE performance_id =
    %s"""
query_performance_actors = """DELETE FROM performance_actors
    WHERE performance_id = %s"""
query_performance ="""DELETE FROM performances WHERE
    performance_id = %s"""
query_theater = """DELETE FROM theaters WHERE theater_id = %s
    """
db.execute_query(query_schedule, (theater_id,))
for performance in performance_ids:
    performance_id = performance[0]
    db.execute_query(query_ticket, (performance_id,))
    db.execute_query(query_performance_actors, (
        performance_id,))
    db.execute_query(query_performance, (performance_id,))
db.execute_query(query_theater, (theater_id,))
```

When the theater's information needs to be modified, an **UPDATE** statement is used.

Excerpt of Theater Edit Code

```
query = """UPDATE theaters SET theater_name = %s, address = %
    s, seats = %s, contact_num = %s WHERE theater_id = %s"""
db.execute_query(query, (theater_name, address, seats,
    contact, theater_id))
```

3.2.5 Performance Management Query

This process involves the **performances**, **actors**, **performance_actors**, **theaters**, **dramas**, and **schedules** tables, as a performance's information includes a combination of details such as the drama, performance time, and venue.

When displaying all performance information, the data from multiple tables is first combined using the **performance_id** to generate the performance details, and then the **performance_actors** table is used to link the information to the actors.

Excerpt of Performance show Code

```
performances = db.fetch_all("""
    SELECT
    performances.performance_id, dramas.drama_name,
    theaters.theater_name, actors.actor_name,
    performance_actors.role, schedules.start_time,
    schedules.end_time
    FROM performances
    LEFT JOIN theaters ON performances.theater_id = theaters.
        theater_id
    LEFT JOIN dramas ON dramas.drama_id = performances.
        drama_id
    LEFT JOIN performance_actors ON performance_actors.
        performance_id = performances.performance_id
    LEFT JOIN actors ON actors.actor_id = performance_actors.
        actor_id
    LEFT JOIN schedules ON performances.performance_id =
        schedules.performance_id;""")
```

When modifying selected performance details, such as the drama name, venue, and time, the staff only needs to provide the updated information. The primary key of the new data is then found in the corresponding table and updated in the **performances** table, followed by updating the corresponding time for that performance_id in the **schedules** table.

Excerpt of Performance Edit Code

```
performance_info = treeview.item(selected_item[0])['values']
performance_id, drama_name, theater_name, actor, role,
    start_time, end_time = performance_info
theater_id = db.fetch_one("""SELECT theater_id FROM theaters
    WHERE theater_name=%s""", (updated_theater_name,))[0]
drama_id = db.fetch_one("""SELECT drama_id FROM dramas WHERE
    drama_name=%s""", (updated_drama_name,))[0]
query_performance = """
    UPDATE performances, schedules
    JOIN theaters ON performances.theater_id = theaters.
        theater_id
    JOIN schedules ON schedules.performance_id = performances
        .performance_id
    SET performances.theater_id = %s,
        performances.drama_id = %s,
        schedules.start_time = %s,
        schedules.end_time = %s
    WHERE performances.performance_id = %s"""
db.execute_query(query_performance, (theater_id, drama_id,
    updated_start_time, updated_end_time, performance_id))
```

When adding a performance, the staff needs to provide all relevant performance information, but does not need to include actor role information. Then, the corresponding primary key is found and the data is added to the **performances** table using an **INSERT** statement.

Excerpt of Performance Add Code

```
drama_id = db.fetch_one("""SELECT drama_id FROM dramas WHERE
    drama_name=%s""", (drama_name,))[0]
theater_id = db.fetch_one("""SELECT theater_id FROM theaters
    WHERE theater_name=%s""",(theater_name))[0]
query = """
    INSERT INTO performances (drama_id, theater_id)
    VALUES (%s, %s)"""
db.execute_query(query, (drama_id, theater_id,))
```

When the staff wants to adjust the role information for a performance, they need to input the performance_id, actor name, and role name, and then update or add the corresponding data to the **performance_actors** table.

Excerpt of Charator Adjust Code

```
if role in existing_roles:
    confirm = messagebox.askyesno("Confirm", """The role
        exists.\nDo you want to replace it?""")
    if confirm:
        db.execute_query("""UPDATE performance_actors
            SET actor_id = (SELECT actor_id FROM actors WHERE
                actor_name = %s)
            WHERE performance_id = %s AND role = %s
            """, (actor_name, performance_id, role))
else:
    actor_id = db.fetch_one("""SELECT actor_id FROM actors
       WHERE actor_name = %s""", (actor_name,))
    if actor_id is None:
        messagebox.showerror("Error", """Actor not found in
            the database!""")
        return
    actor_id = actor_id[0]
    db.execute_query("""
        INSERT INTO performance_actors (performance_id, role,
            actor_id)
         VALUES (%s, %s, %s)
        """, (performance_id, role, actor_id))
```

When deleting a performance, all related foreign key information must be deleted first, followed by the deletion of the performance itself.

Excerpt of Performance Delete Code

```
erformance_id = treeview.item(selected_item[0])['values'][0]
query1 = """DELETE FROM performances WHERE performance_id = %
    s"""
query2 = """DELETE FROM tickets WHERE performance_id = %s"""
query3 = """DELETE FROM schedules WHERE performance_id = %s
    """
query4 = """DELETE FROM performance_actors WHERE
    performance_id = %s"""
db.execute_query(query4, (performance_id,))
db.execute_query(query3, (performance_id,))
db.execute_query(query2, (performance_id,))
db.execute_query(query1, (performance_id,))
```

4 Implementation

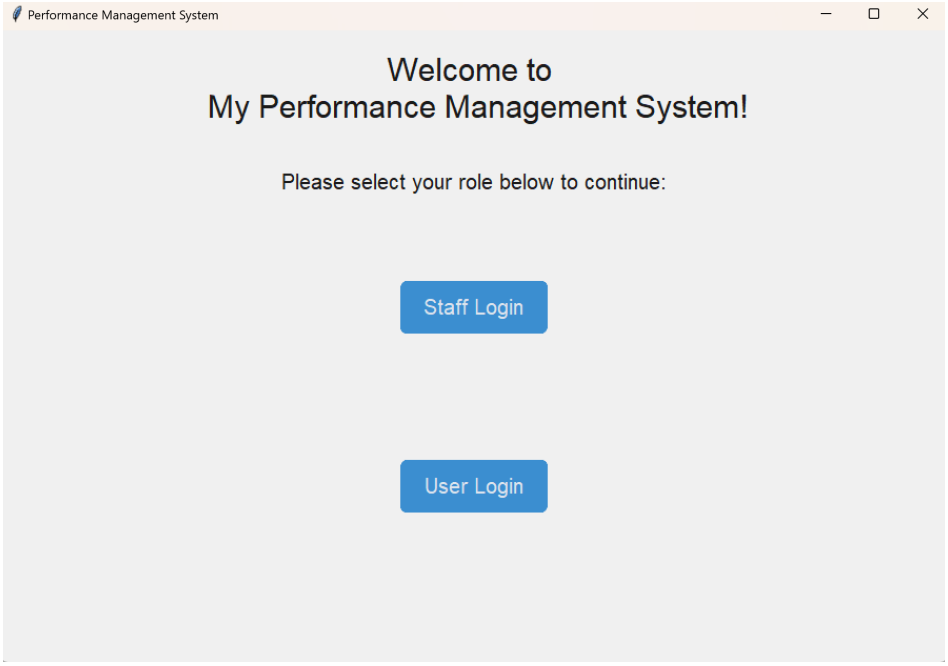

Figure 3: Initial interface

4.1 User Usage

Users can log in using their username and password. Upon successful login, the username will be displayed to welcome the user. If the login fails, an error message will pop up, prompting the user to re-enter the information. Users without an account can click on 'Register' and fill in the information in the registration popup to sign up. Users can also click the 'Back' button in the top left corner to return and select a different role.

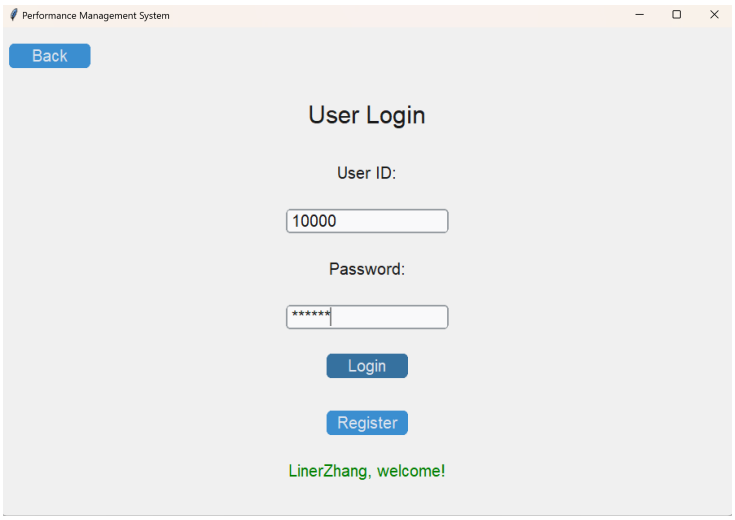

Figure 4: User login

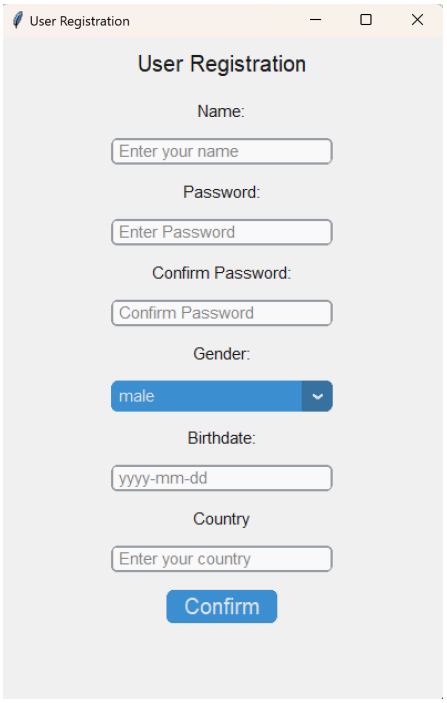

Figure 5: User registration

After logging in, the interface is divided into three main sections. The top row contains two buttons: the left button allows the user to log out and return to the initial screen, while the right button allows the user to view and manage their personal information. The left section contains three buttons for querying and displaying different types of information. The right section is the display area, which by default shows drama information, with two items per page and the ability to paginate.

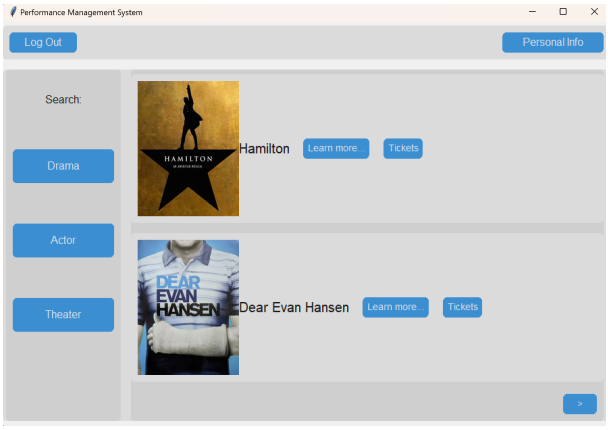

Figure 6: User dashboard

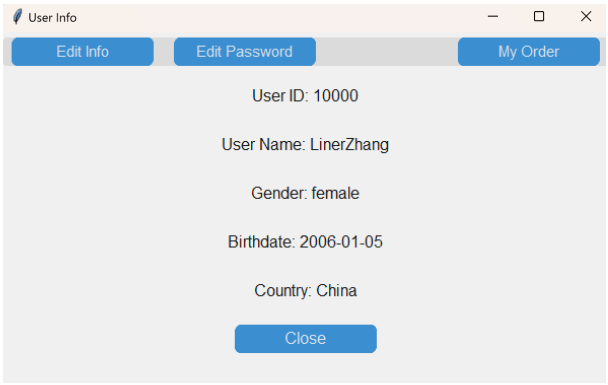

Figure 7: Personal info

In the Personal Info section, the user's personal information is displayed. Clicking 'Edit Personal Info' will open a new popup where the current information can be modified. Clicking 'Edit Password' will open a new popup prompting the user to enter both the old and new passwords, with the password only being updated if the old password is correct. Clicking 'My Orders' allows the user to view all the tickets they have purchased, along with the related viewing times and locations, providing convenience for the user.

Figure 8: Edit info

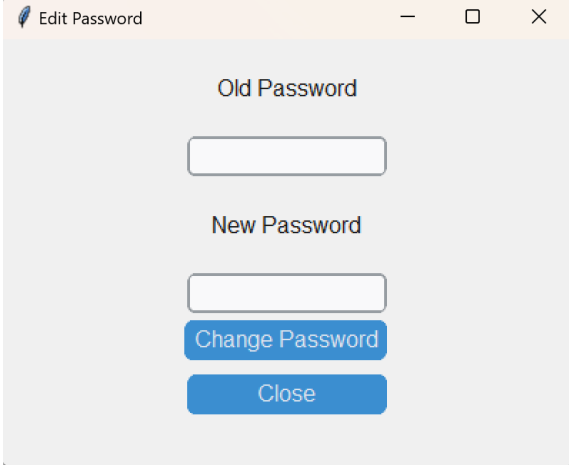

Figure 9: Edit password

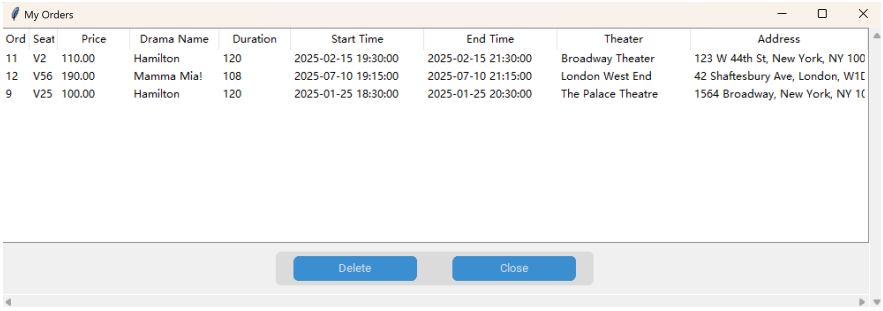

Figure 10: My orders

4.1.1 Drama Display

Clicking the 'Learn More' button of any drama that interests the user will display a brief summary of the drama's main content.$Figure11-14$

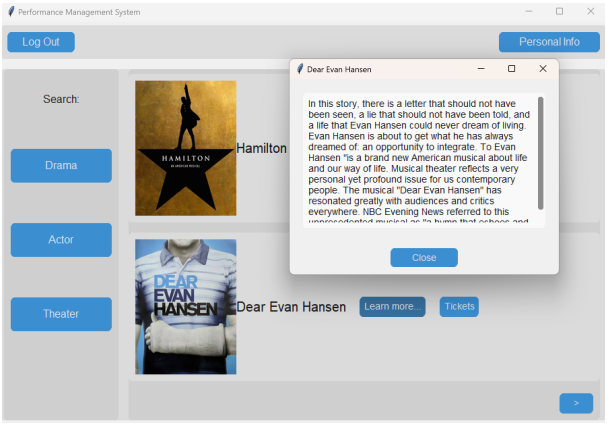

Figure 11: Learn more example

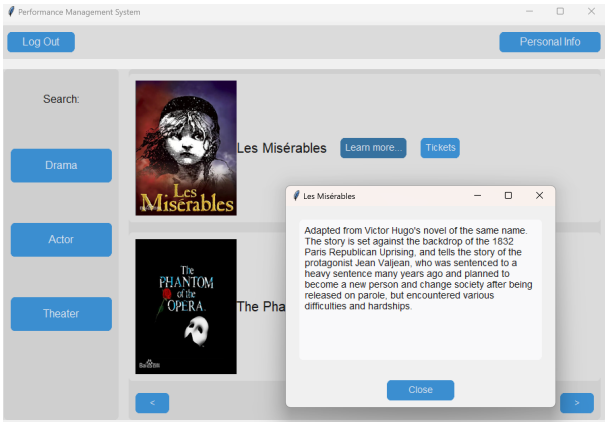

Figure 12: Learn more example

When displaying tickets, the user can choose the sorting method and type for the display. After clicking on the ticket they wish to purchase, clicking 'Buy It Now' will display the complete order details, allowing the user to confirm the purchase.

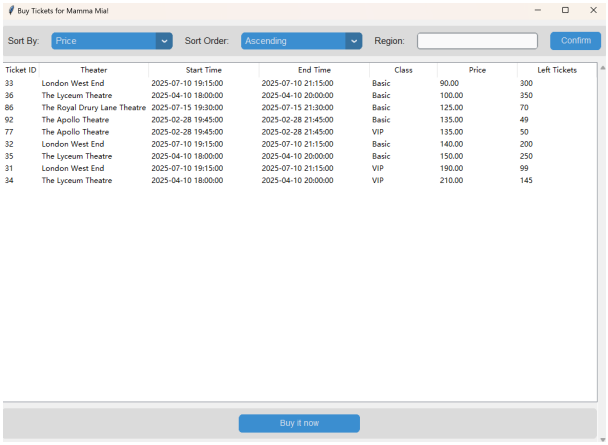

Figure 13: Tickets example

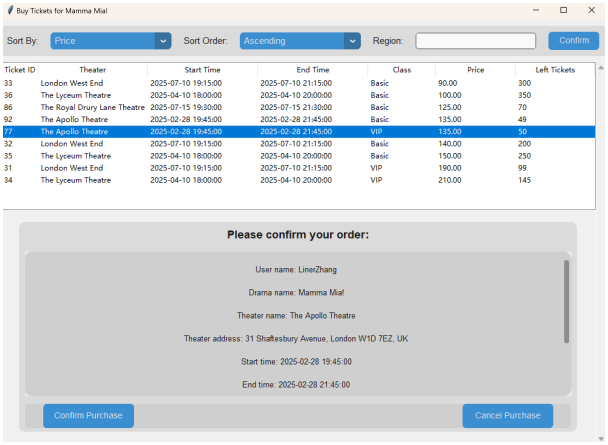

Figure 14: Buy it now

4.1.2 Actor Display

The process is the same as before, so it is omitted.$Figure 15-18$

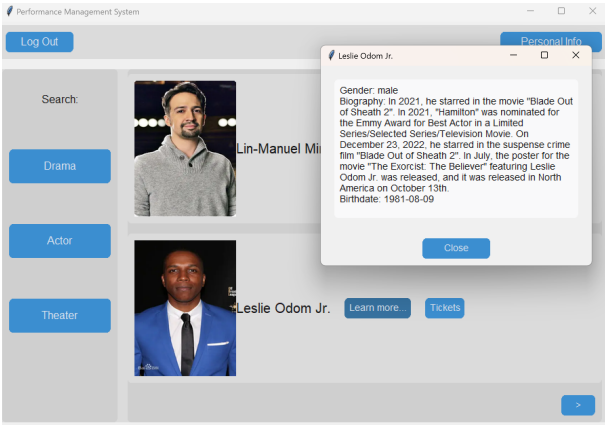

Figure 15: Learn more example

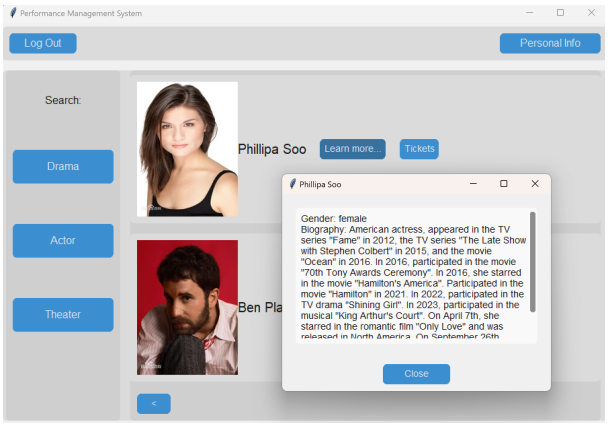

Figure 16: Learn more example

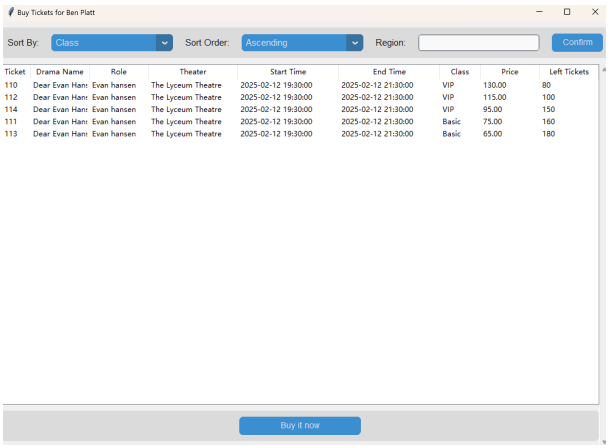

Figure 17: Tickets example

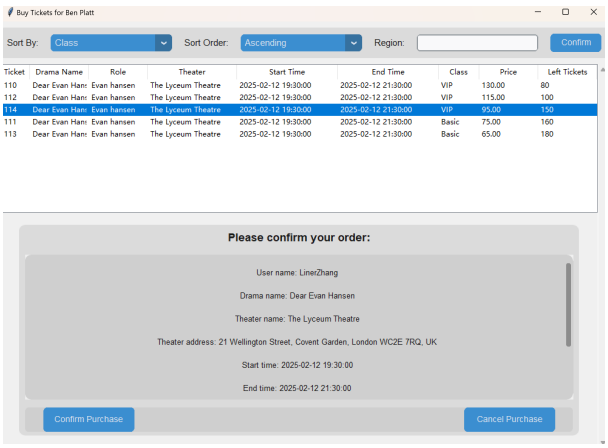

Figure 18: Buy it now

4.1.3 Theater Display

The process is the same as before, so it is omitted.$Figure 19-22$

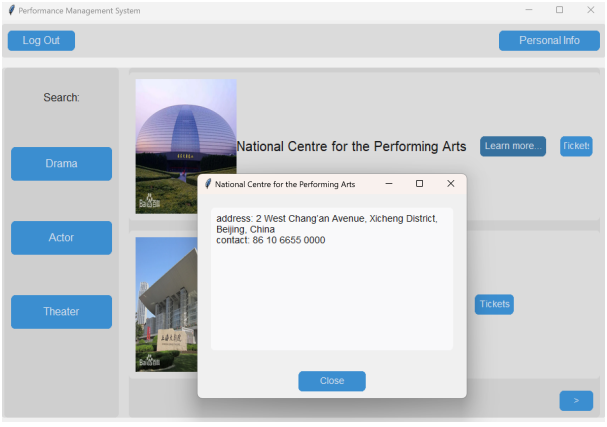

Figure 19: Learn more example

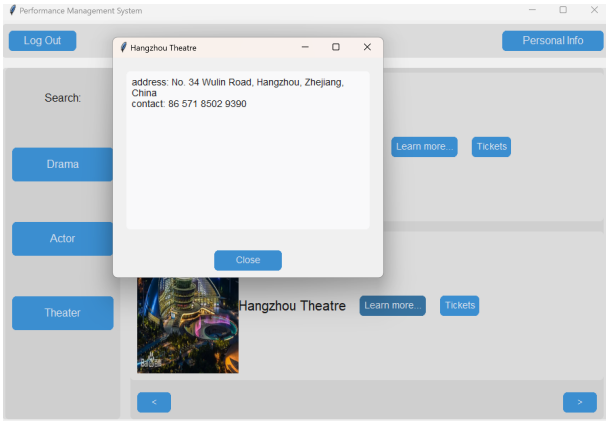

Figure 20: Learn more example

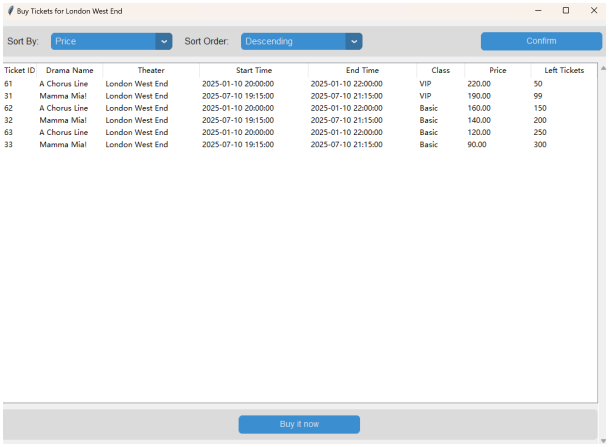

Figure 21: Tickets example

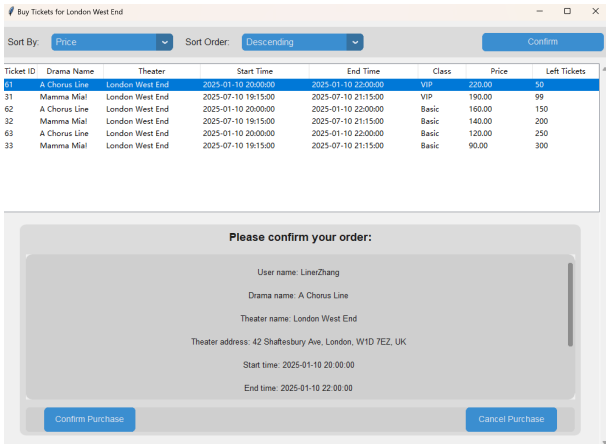

Figure 22: Buy it now

4.2 Staff Usage

The login process is the same as before, except without the registration option, so it is omitted. available staff account:

- Work ID: 10000
- Password: 123456

After logging in, the interface is still divided into three main sections. The top row contains only a logout button. The left section has four buttons, each triggering a different type of information. The right section is the information display area.$Figure 23-24$

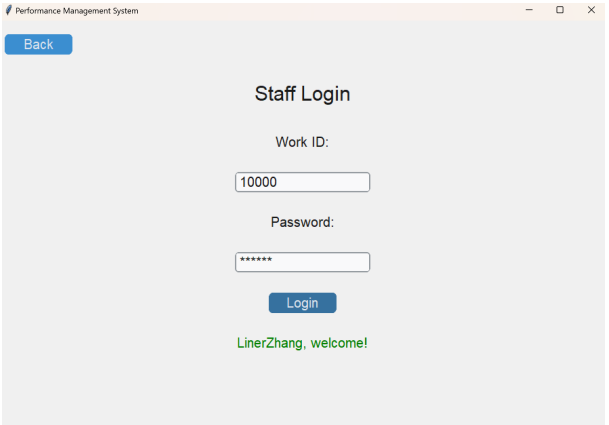

Figure 23: Staff login

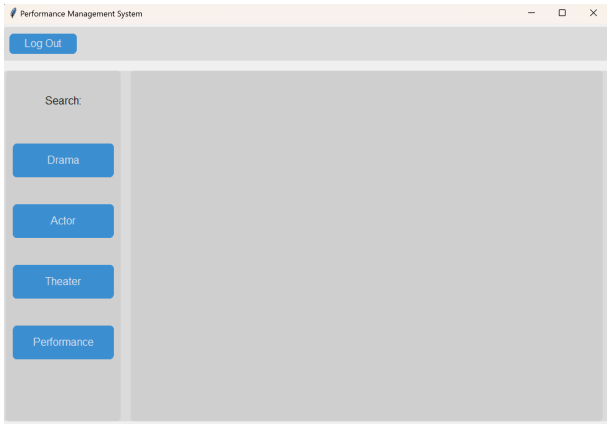

Figure 24: Staff dashboard

4.2.1 Drama Management

Clicking on 'Drama' displays all the drama information, with options to add, modify, or delete drama information using the corresponding buttons.$Figure 25-30$

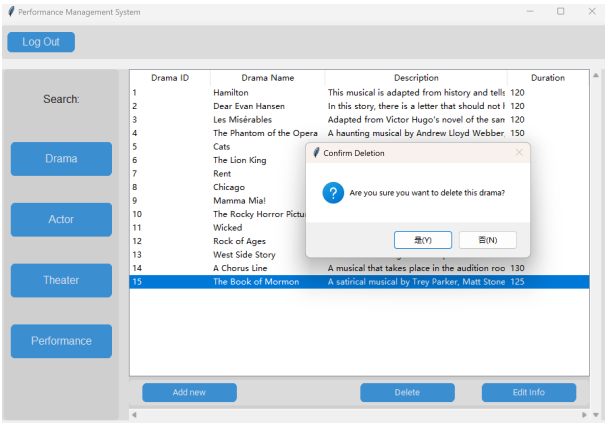

Figure 25: Drama delete

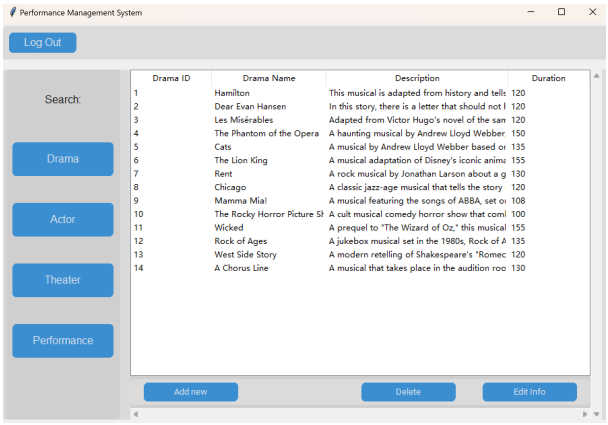

Figure 26: After deletion

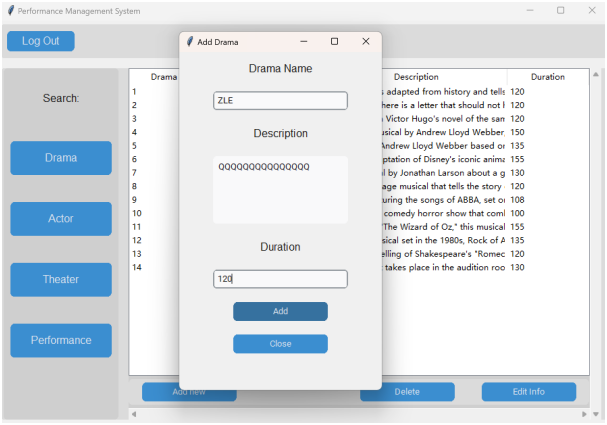

Figure 27: Drama add

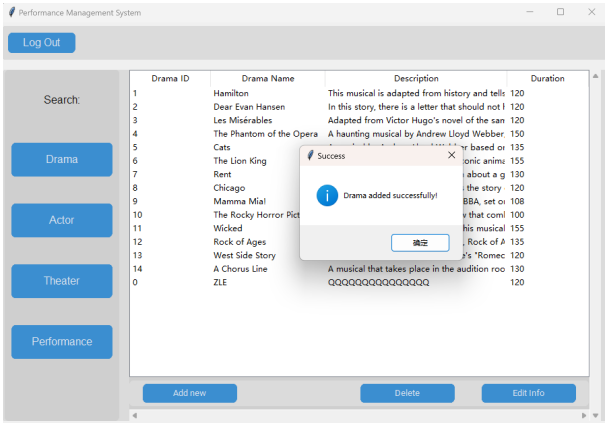

Figure 28: After add

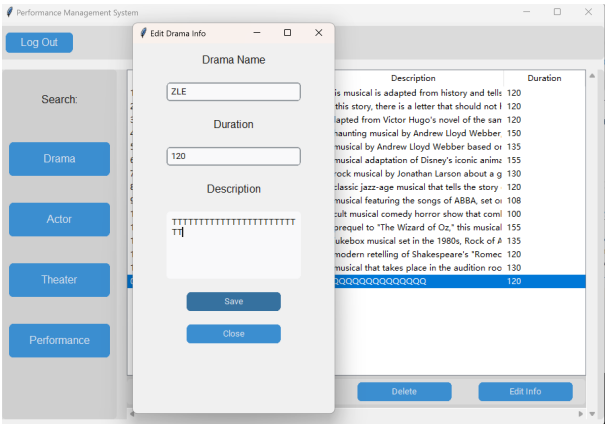

Figure 29: Drama edit

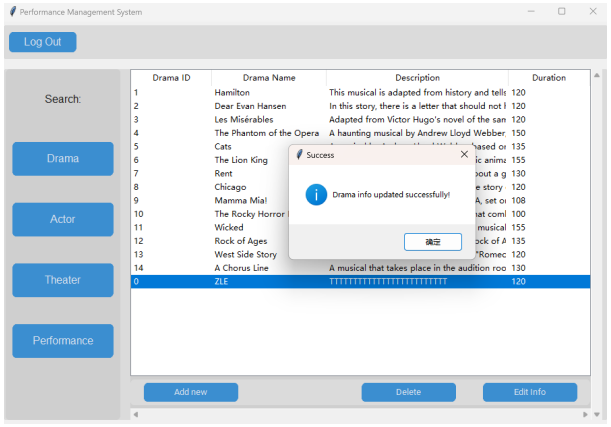

Figure 30: After edit

4.2.2 Actor Management

Clicking on 'Actor' displays all the actor information, with options to add, modify, or delete actor information using the corresponding buttons.$Figure 31-36$

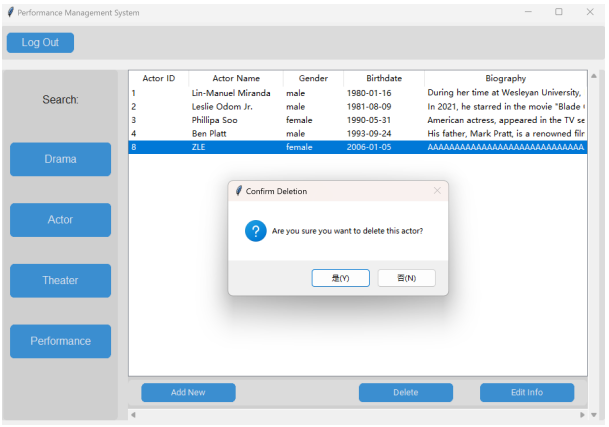

Figure 31: Actor delete

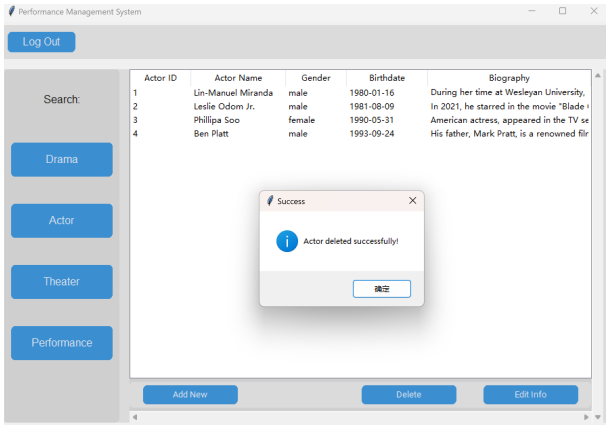

Figure 32: After deletion

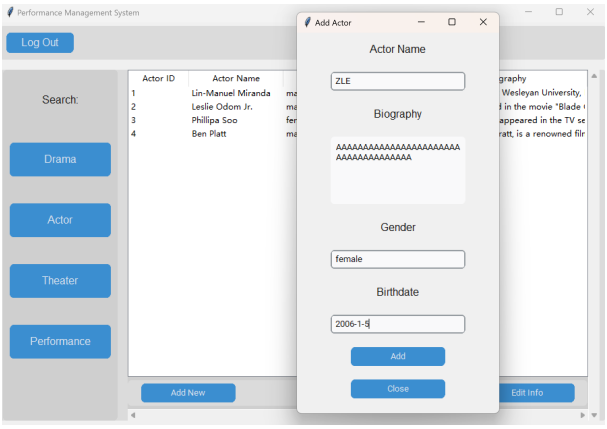

Figure 33: Actor add

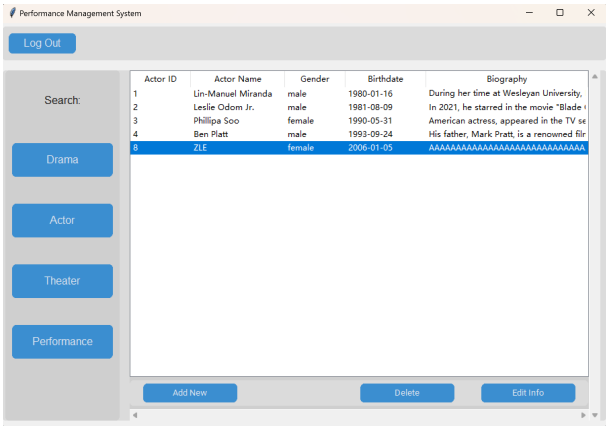

Figure 34: After add

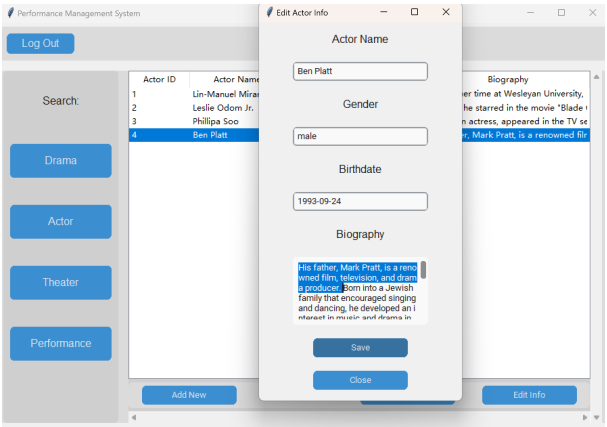

Figure 35: Actor edit

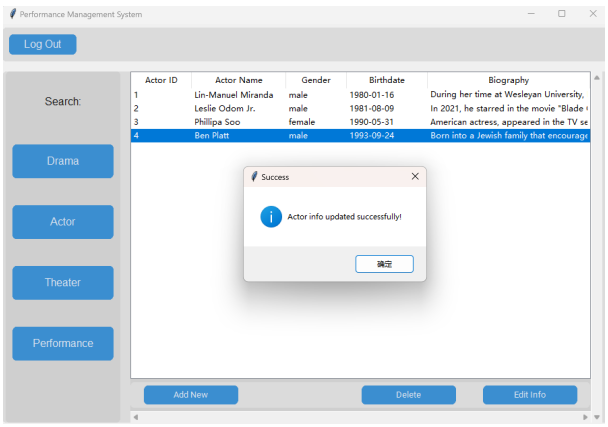

Figure 36: After edit

4.2.3 Theater Management

Clicking on 'Theater' displays all the theater information, with options to add, modify, or delete theater information using the corresponding buttons.$Figure 37-42$

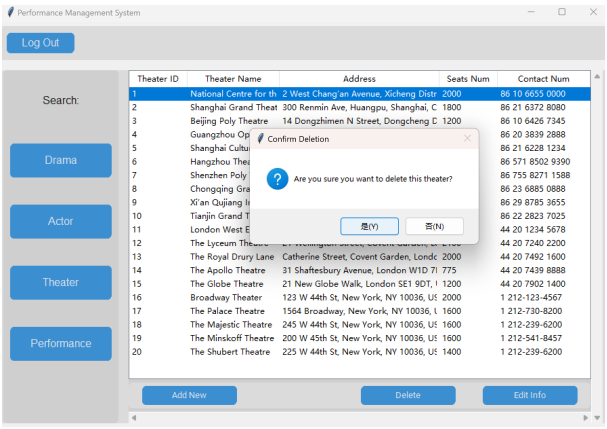

Figure 37: Theater delete

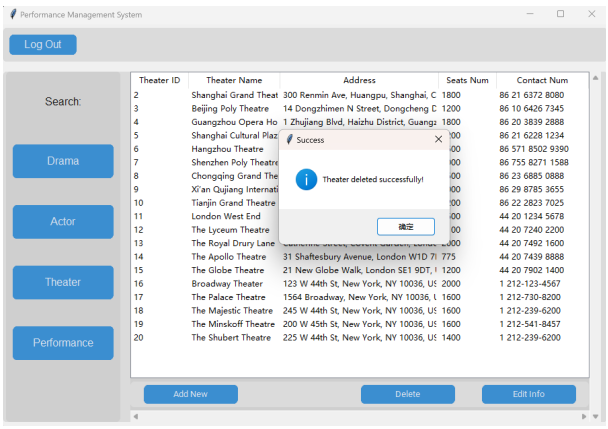

Figure 38: After deletion

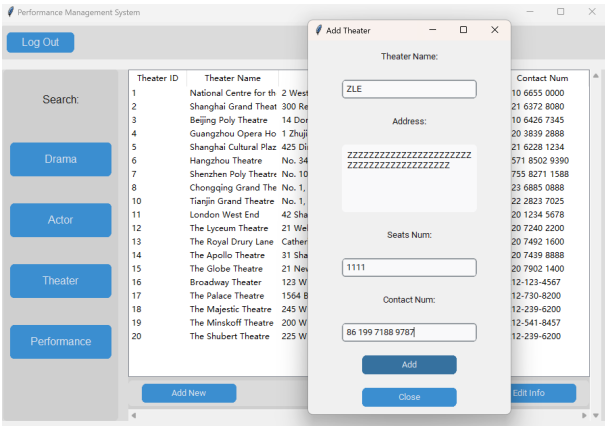

Figure 39: Theater add

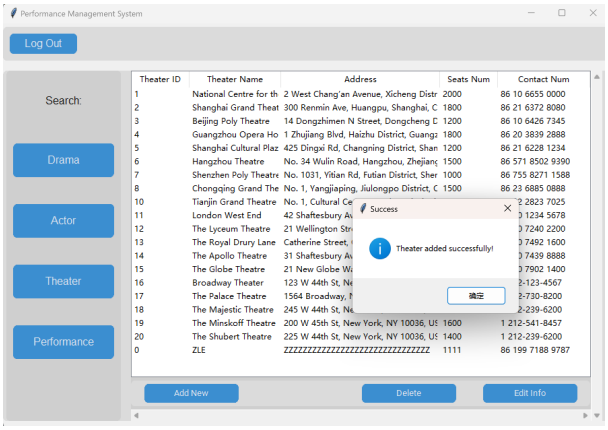

Figure 40: After add

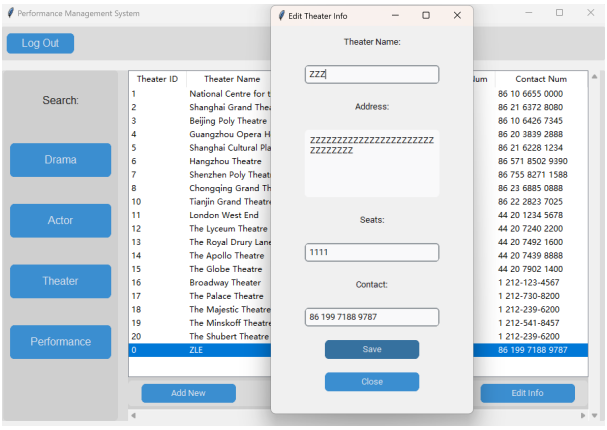

Figure 41: Theater edit

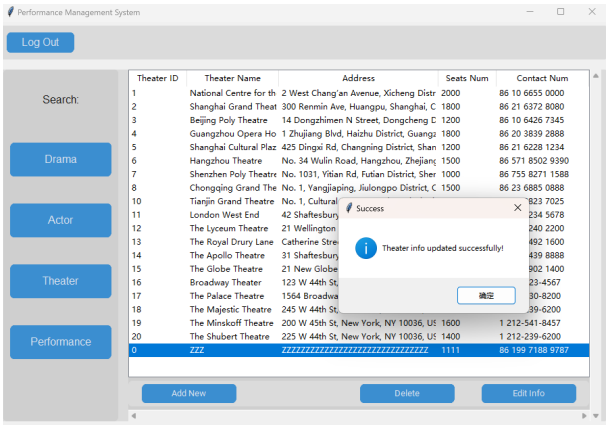

Figure 42: After edit

4.2.4 Performance Management

Clicking on 'Performance' displays all the performance information, with options to add, modify, or delete performance information and adjust charactors information using the corresponding buttons.$Figure 43-46$

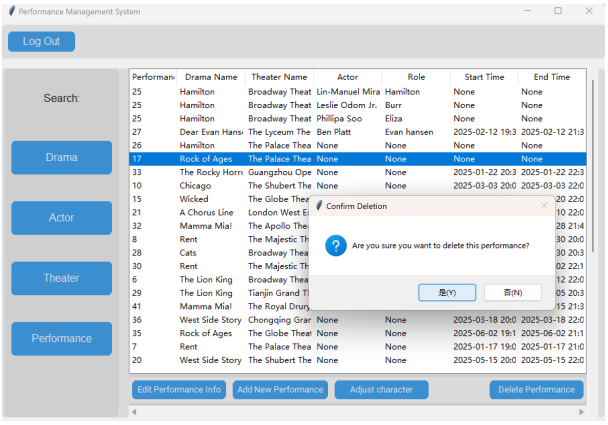

Figure 43: Performance delete

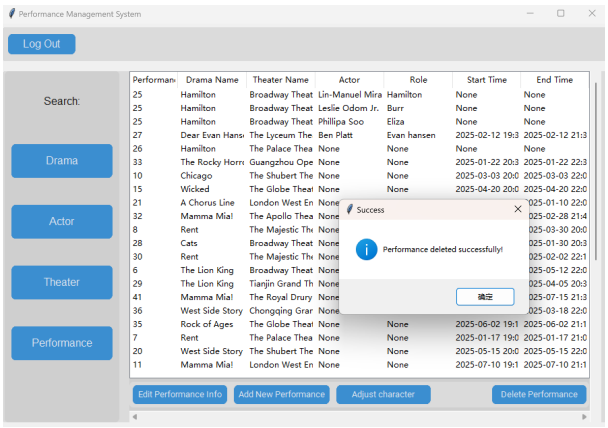

Figure 44: After deletion

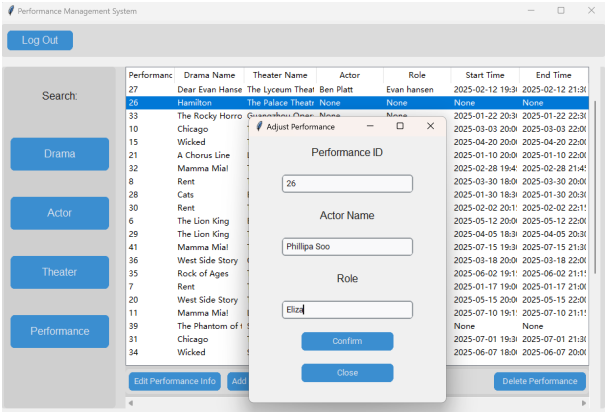

Figure 45: Charactor adjust

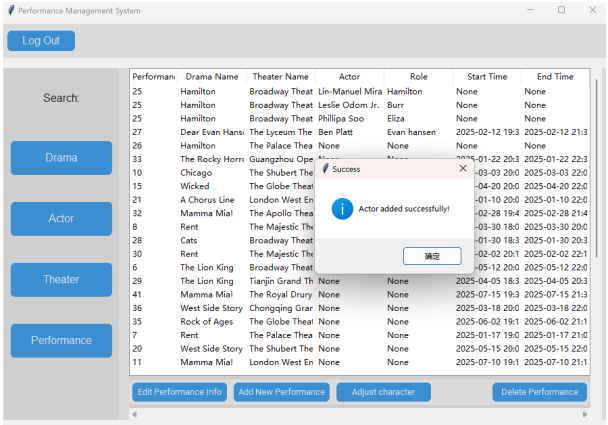

Figure 46: After adjust

5 Conclusion

Although a lot of effort has been put into this project, drawing inspiration from real-world performance systems and striving to improve functionality, time is limited, and there are still many areas that need refinement. For example, issues like the format for writing to the database, whether the performance schedule corresponds to the drama's duration, and certain constraints still need to be addressed. Additionally, some features are missing, such as allowing users to add comments,

share and view comments, and enable ticket refunds. After the course, I will continue to study databases and further improve this program.