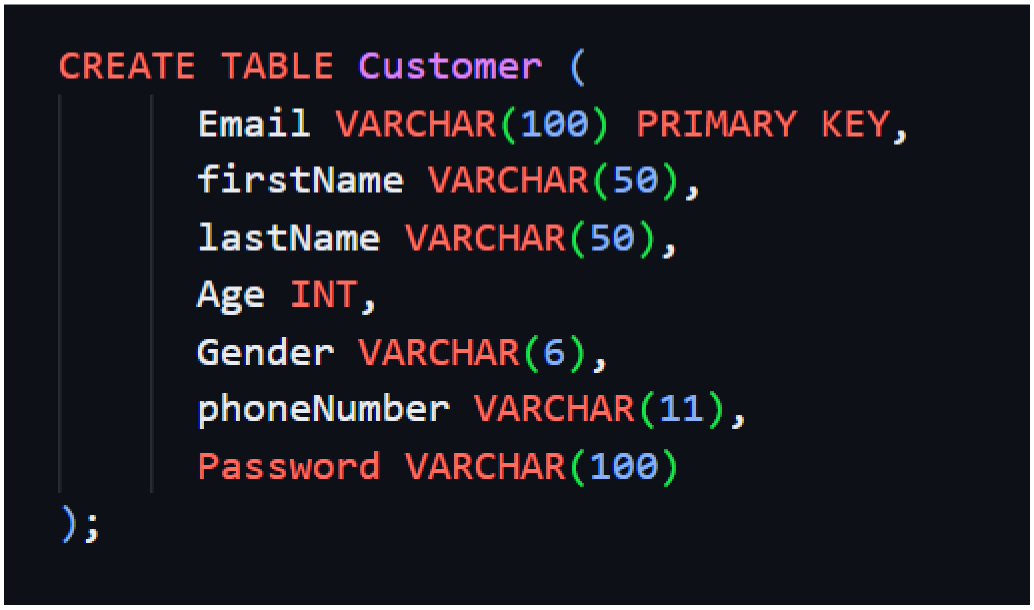
Database Systems

Project Report

Ahmed Hossam Abbas	21P0141
Yehia Mahmoud Lotfy Sakr	21P0166
Mark Saleh	21P0206
Ahmed Tarek Mahmoud	2100561
Hanna Mohamed Bakeer	21P0162
Jana Samir Abdelrahman	21P0255

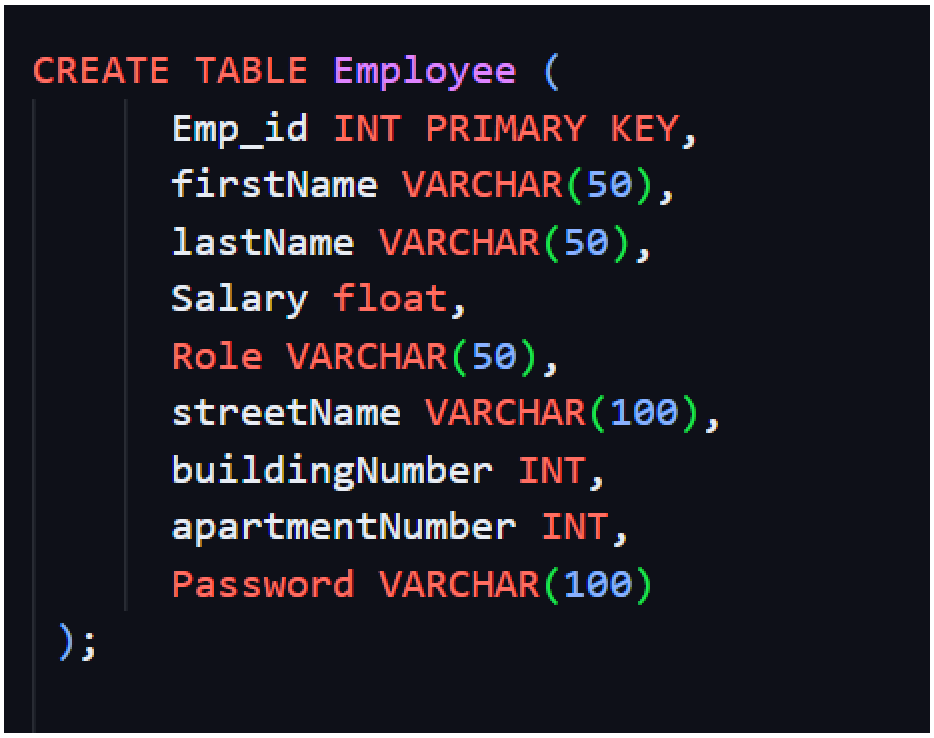
Tables:

Customer table:



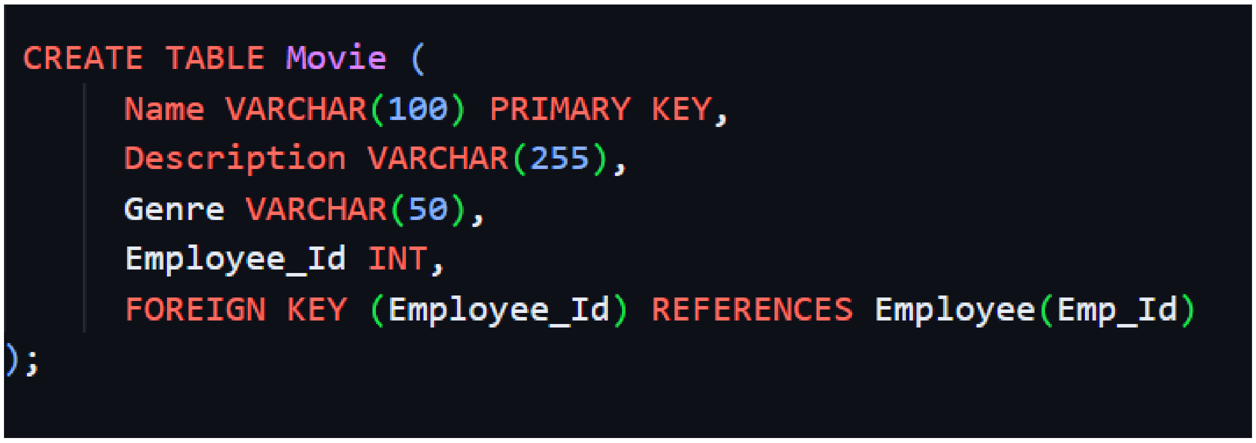
```sql
CREATE TABLE Customer (
        Email VARCHAR(100) PRIMARY KEY,
        firstName VARCHAR(50),
        lastName VARCHAR(50),
        Age INT,
        Gender VARCHAR(6),
        phoneNumber VARCHAR(11),
        Password VARCHAR(100)
);
```

Employee table:



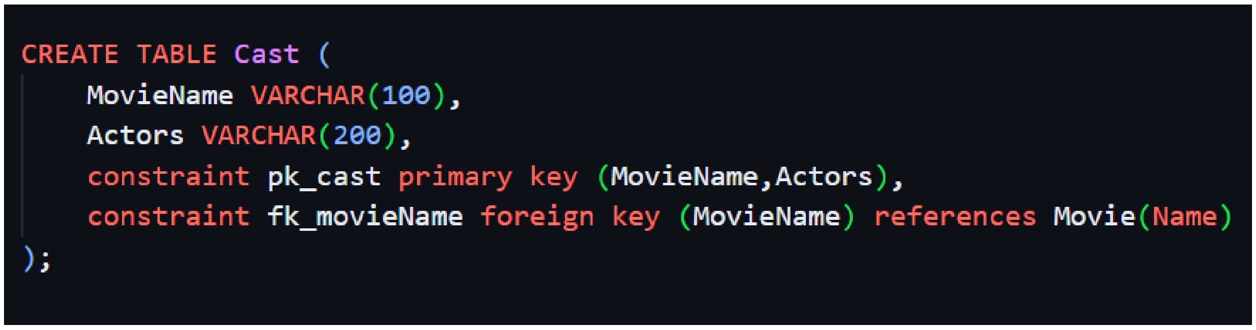
```sql
CREATE TABLE Employee (
        Emp_id INT PRIMARY KEY,
        firstName VARCHAR(50),
        lastName VARCHAR(50),
        Salary float,
        Role VARCHAR(50),
        streetName VARCHAR(100),
        buildingNumber INT,
        apartmentNumber INT,
        Password VARCHAR(100)
);
```

Movie table:



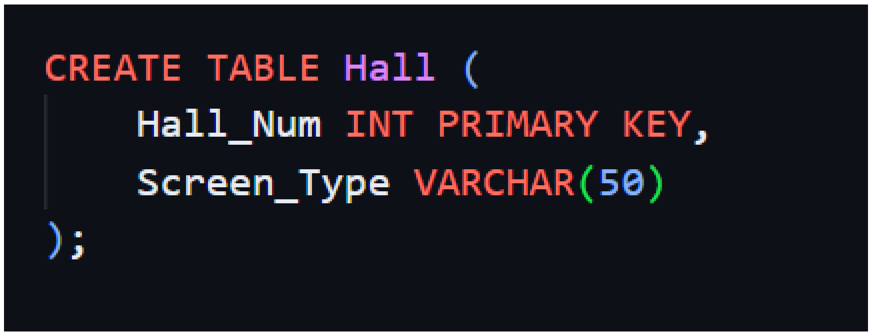
```
CREATE TABLE Movie (
    Name VARCHAR(100) PRIMARY KEY,
    Description VARCHAR(255),
    Genre VARCHAR(50),
    Employee_Id INT,
    FOREIGN KEY (Employee_Id) REFERENCES Employee(Emp_Id)
);
```

Cast table:



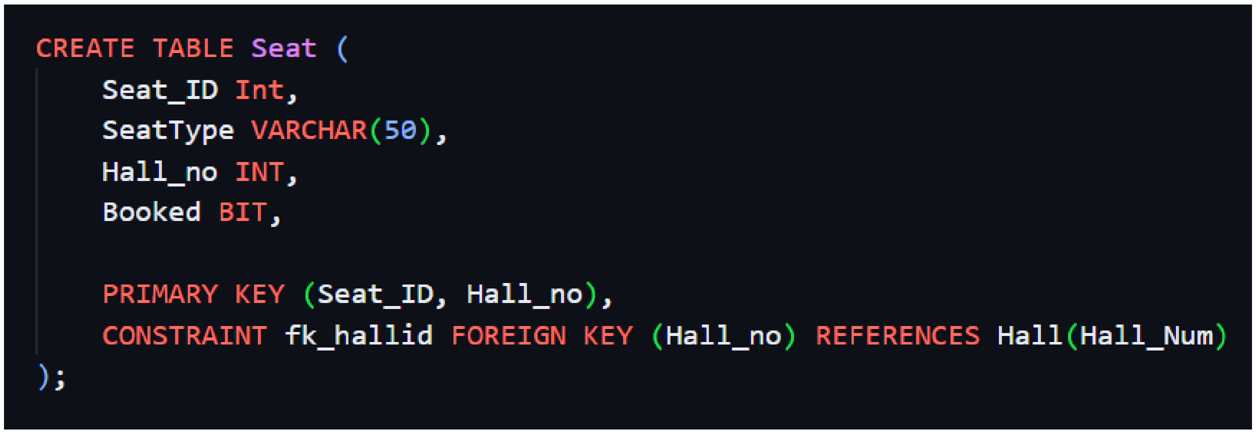
```
CREATE TABLE Cast (
    MovieName VARCHAR(100),
    Actors VARCHAR(200),
    constraint pk_cast primary key (MovieName,Actors),
    constraint fk_movieName foreign key (MovieName) references Movie(Name)
);
```

Hall table:



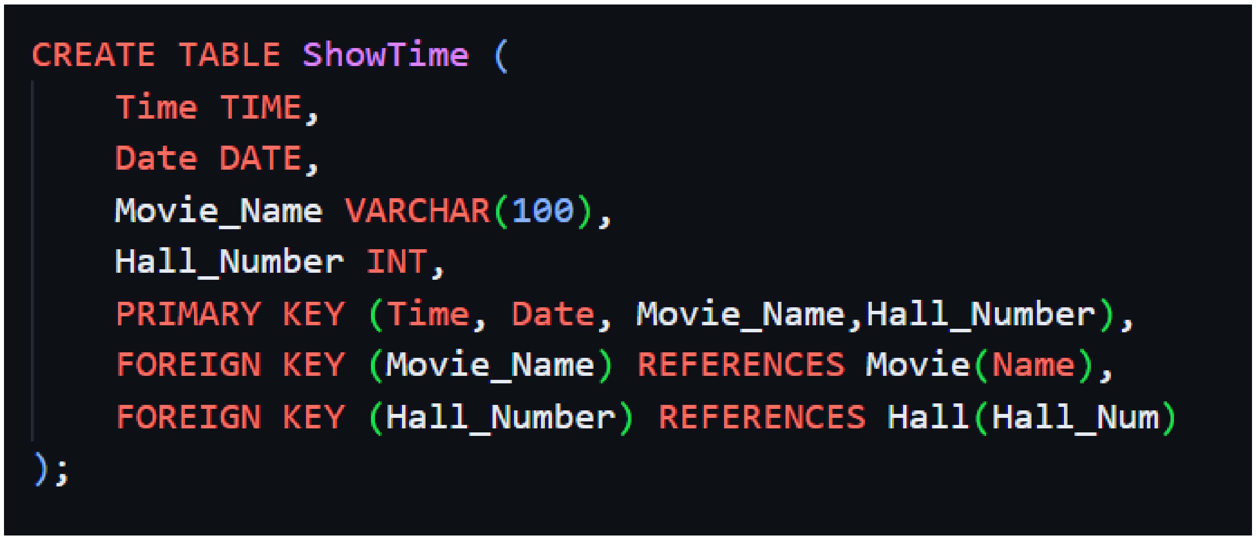
```
CREATE TABLE Hall (
    Hall_Num INT PRIMARY KEY,
    Screen_Type VARCHAR(50)
);
```

Seat table:



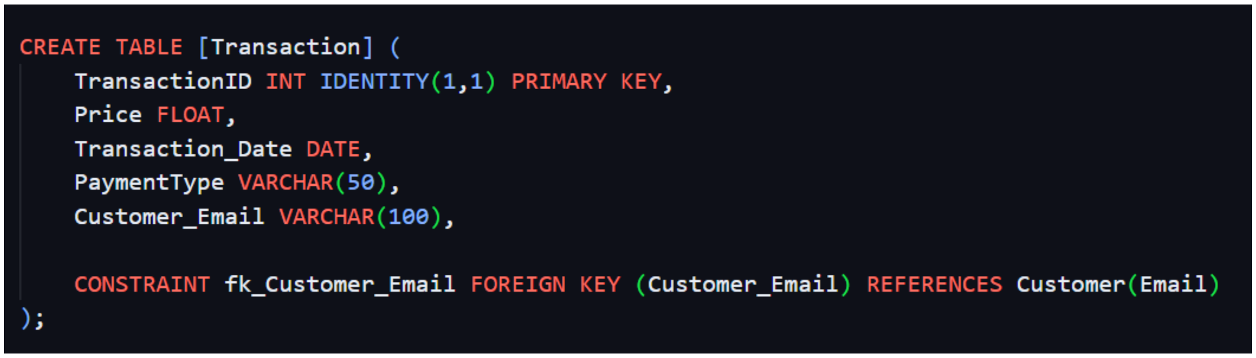
```sql
CREATE TABLE Seat (
    Seat_ID Int,
    SeatType VARCHAR(50),
    Hall_no INT,
    Booked BIT,

    PRIMARY KEY (Seat_ID, Hall_no),
    CONSTRAINT fk_hallid FOREIGN KEY (Hall_no) REFERENCES Hall(Hall_Num)
);
```

ShowTime table:



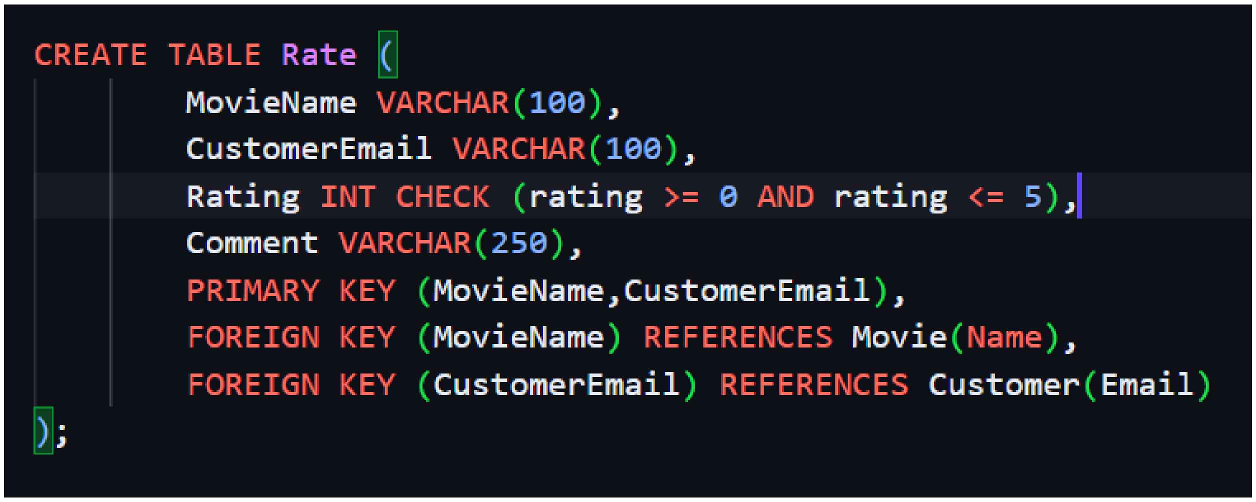
```sql
CREATE TABLE ShowTime (
    Time TIME,
    Date DATE,
    Movie_Name VARCHAR(100),
    Hall_Number INT,
    PRIMARY KEY (Time, Date, Movie_Name,Hall_Number),
    FOREIGN KEY (Movie_Name) REFERENCES Movie(Name),
    FOREIGN KEY (Hall_Number) REFERENCES Hall(Hall_Num)
);
```

Transaction table:



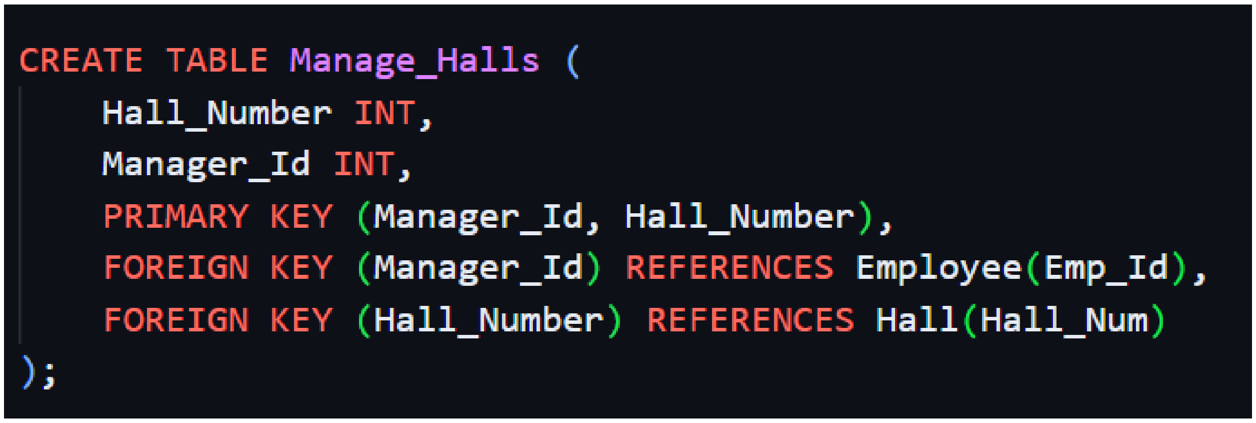
```sql
CREATE TABLE [Transaction] (
    TransactionID INT IDENTITY(1,1) PRIMARY KEY,
    Price FLOAT,
    Transaction_Date DATE,
    PaymentType VARCHAR(50),
    Customer_Email VARCHAR(100),

    CONSTRAINT fk_Customer_Email FOREIGN KEY (Customer_Email) REFERENCES Customer(Email)
);
```

Rate table:



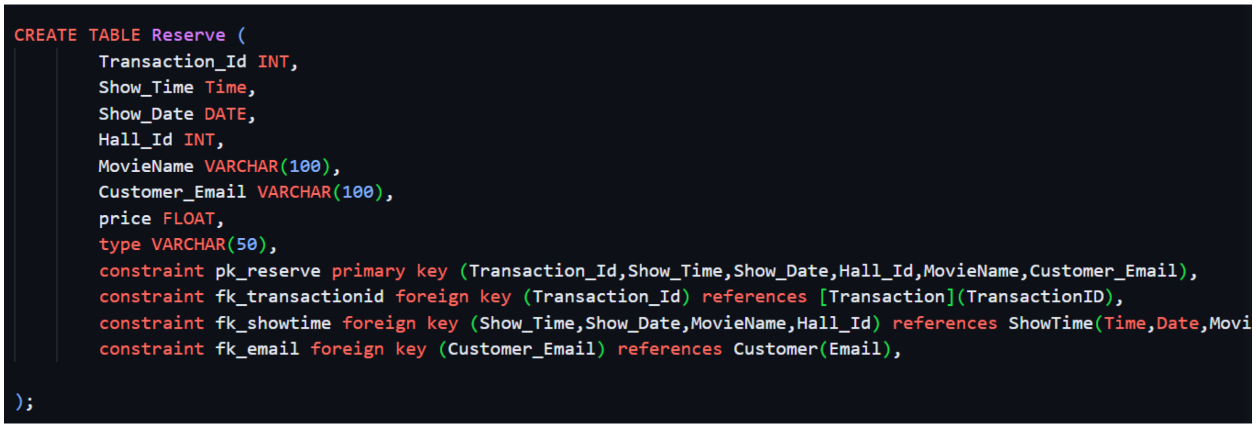
```sql
CREATE TABLE Rate (
        MovieName VARCHAR(100),
        CustomerEmail VARCHAR(100),
        Rating INT CHECK (rating >= 0 AND rating <= 5),
        Comment VARCHAR(250),
        PRIMARY KEY (MovieName,CustomerEmail),
        FOREIGN KEY (MovieName) REFERENCES Movie(Name),
        FOREIGN KEY (CustomerEmail) REFERENCES Customer(Email)
);
```

Manage hall table:



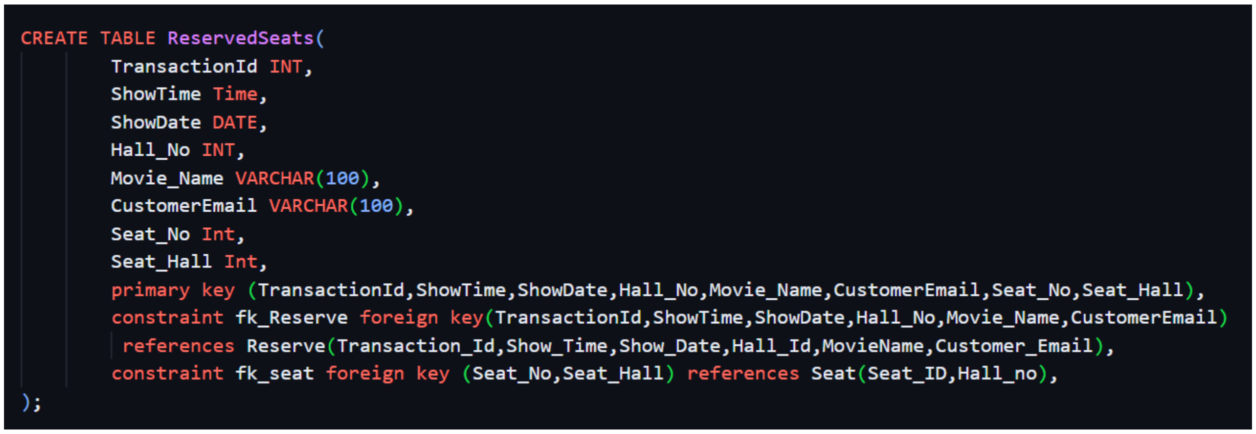
```sql
CREATE TABLE Manage_Halls (
    Hall_Number INT,
    Manager_Id INT,
    PRIMARY KEY (Manager_Id, Hall_Number),
    FOREIGN KEY (Manager_Id) REFERENCES Employee(Emp_Id),
    FOREIGN KEY (Hall_Number) REFERENCES Hall(Hall_Num)
);
```

Reserve table:



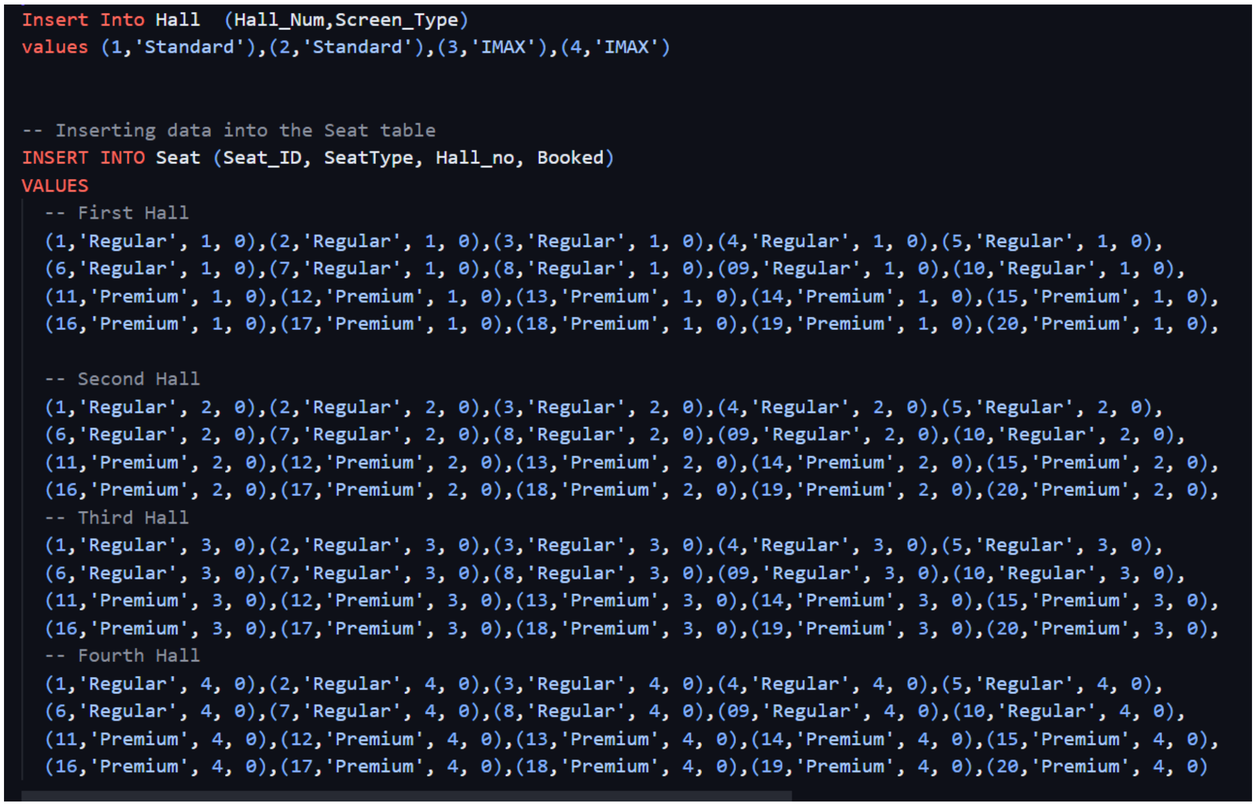
```sql
CREATE TABLE Reserve (
        Transaction_Id INT,
        Show_Time Time,
        Show_Date DATE,
        Hall_Id INT,
        MovieName VARCHAR(100),
        Customer_Email VARCHAR(100),
        price FLOAT,
        type VARCHAR(50),
        constraint pk_reserve primary key (Transaction_Id,Show_Time,Show_Date,Hall_Id,MovieName,Customer_Email),
        constraint fk_transactionid foreign key (Transaction_Id) references [Transaction](TransactionID),
        constraint fk_showtime foreign key (Show_Time,Show_Date,MovieName,Hall_Id) references ShowTime(Time,Date,Movi
        constraint fk_email foreign key (Customer_Email) references Customer(Email),

);
```

Reserved seats table:



```sql
CREATE TABLE ReservedSeats(
        TransactionId INT,
        ShowTime Time,
        ShowDate DATE,
        Hall_No INT,
        Movie_Name VARCHAR(100),
        CustomerEmail VARCHAR(100),
        Seat_No Int,
        Seat_Hall Int,
        primary key (TransactionId,ShowTime,ShowDate,Hall_No,Movie_Name,CustomerEmail,Seat_No,Seat_Hall),
        constraint fk_Reserve foreign key(TransactionId,ShowTime,ShowDate,Hall_No,Movie_Name,CustomerEmail)
         references Reserve(Transaction_Id,Show_Time,Show_Date,Hall_Id,MovieName,Customer_Email),
        constraint fk_seat foreign key (Seat_No,Seat_Hall) references Seat(Seat_ID,Hall_no),
);
```

Table insertions:



```sql
Insert Into Hall  (Hall_Num,Screen_Type)
values (1,'Standard'),(2,'Standard'),(3,'IMAX'),(4,'IMAX')


-- Inserting data into the Seat table
INSERT INTO Seat (Seat_ID, SeatType, Hall_no, Booked)
VALUES
  -- First Hall
  (1,'Regular', 1, 0),(2,'Regular', 1, 0),(3,'Regular', 1, 0),(4,'Regular', 1, 0),(5,'Regular', 1, 0),
  (6,'Regular', 1, 0),(7,'Regular', 1, 0),(8,'Regular', 1, 0),(09,'Regular', 1, 0),(10,'Regular', 1, 0),
  (11,'Premium', 1, 0),(12,'Premium', 1, 0),(13,'Premium', 1, 0),(14,'Premium', 1, 0),(15,'Premium', 1, 0),
  (16,'Premium', 1, 0),(17,'Premium', 1, 0),(18,'Premium', 1, 0),(19,'Premium', 1, 0),(20,'Premium', 1, 0),

  -- Second Hall
  (1,'Regular', 2, 0),(2,'Regular', 2, 0),(3,'Regular', 2, 0),(4,'Regular', 2, 0),(5,'Regular', 2, 0),
  (6,'Regular', 2, 0),(7,'Regular', 2, 0),(8,'Regular', 2, 0),(09,'Regular', 2, 0),(10,'Regular', 2, 0),
  (11,'Premium', 2, 0),(12,'Premium', 2, 0),(13,'Premium', 2, 0),(14,'Premium', 2, 0),(15,'Premium', 2, 0),
  (16,'Premium', 2, 0),(17,'Premium', 2, 0),(18,'Premium', 2, 0),(19,'Premium', 2, 0),(20,'Premium', 2, 0),
  -- Third Hall
  (1,'Regular', 3, 0),(2,'Regular', 3, 0),(3,'Regular', 3, 0),(4,'Regular', 3, 0),(5,'Regular', 3, 0),
  (6,'Regular', 3, 0),(7,'Regular', 3, 0),(8,'Regular', 3, 0),(09,'Regular', 3, 0),(10,'Regular', 3, 0),
  (11,'Premium', 3, 0),(12,'Premium', 3, 0),(13,'Premium', 3, 0),(14,'Premium', 3, 0),(15,'Premium', 3, 0),
  (16,'Premium', 3, 0),(17,'Premium', 3, 0),(18,'Premium', 3, 0),(19,'Premium', 3, 0),(20,'Premium', 3, 0),
  -- Fourth Hall
  (1,'Regular', 4, 0),(2,'Regular', 4, 0),(3,'Regular', 4, 0),(4,'Regular', 4, 0),(5,'Regular', 4, 0),
  (6,'Regular', 4, 0),(7,'Regular', 4, 0),(8,'Regular', 4, 0),(09,'Regular', 4, 0),(10,'Regular', 4, 0),
  (11,'Premium', 4, 0),(12,'Premium', 4, 0),(13,'Premium', 4, 0),(14,'Premium', 4, 0),(15,'Premium', 4, 0),
  (16,'Premium', 4, 0),(17,'Premium', 4, 0),(18,'Premium', 4, 0),(19,'Premium', 4, 0),(20,'Premium', 4, 0)
```

Procedures Explanation:

CreateCustomerAccount:

```
CREATE PROCEDURE CreateCustomerAccount
    @Email VARCHAR(100),
    @FirstName VARCHAR(50),
    @LastName VARCHAR(50),
    @Age INT,
    @Gender VARCHAR(6),
    @PhoneNumber VARCHAR(11),
    @Password VARCHAR(100)
AS
BEGIN
    -- Check if the customer already exists
    IF EXISTS (SELECT 1 FROM Customer WHERE Email = @Email)
    BEGIN
        -- Customer with the same email already exists, raise an error
        RAISERROR('Customer with the same email already exists', 16, 1);
        RETURN;
    END;
```

1. The stored procedure expects certain information about a customer, such as their email, first name, last name, age, gender, phone number, and password.

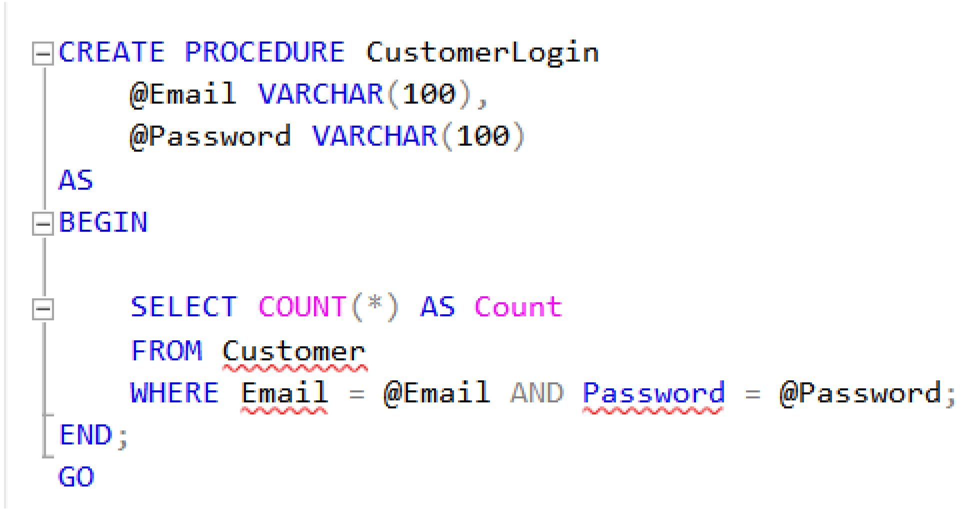
2. It begins with a check to see if a customer with the same email already exists in the "Customer" table. This is done to avoid creating duplicate customer records.

3. If a customer with the same email is found, it raises an error to let you know that a customer with that email already exists. This ensures that each customer has a unique email address.

4. If no customer with the same email is found, the stored procedure continues to the next step.

5. It inserts a new customer record into the "Customer" table, using the provided information for email, first name, last name, age, gender, phone number, and password.

CustomerLogin:


```
CREATE PROCEDURE CustomerLogin
    @Email VARCHAR(100),
    @Password VARCHAR(100)
AS
BEGIN

    SELECT COUNT(*) AS Count
    FROM Customer
    WHERE Email = @Email AND Password = @Password;
END;
GO
```

- The stored procedure is designed to handle customer login functionality.
- It takes in the customer's email and password as parameters.
- It executes a SELECT query on the Customer table. The SELECT query counts the number of rows where the provided email and password match those stored in the Customer table.
- The result of the SELECT query is returned as a count of matching rows with the alias Count.
- This stored procedure is intended to be used to verify customer login credentials. If the count returned is 1, it indicates that a customer with the provided email and password combination exists in the database, implying successful login. Otherwise, if the count is 0, it means there's no matching customer record with the provided credentials, suggesting an unsuccessful login attempt.

CreateEmployeeAccount

```sql
CREATE PROCEDURE CreateEmployeeAccount
    @EmpId INT,
    @FirstName VARCHAR(50),
    @LastName VARCHAR(50),
    @Salary FLOAT,
    @Role VARCHAR(50),
    @StreetName VARCHAR(100),
    @BuildingNumber INT,
    @ApartmentNumber INT,
    @Password VARCHAR(100)
AS
BEGIN

        -- Check if the employee already exists
        IF EXISTS (SELECT 1 FROM Employee WHERE Emp_id = @EmpId)
        BEGIN
            RAISERROR('Employee with ID %d already exists.', 16, 1, @EmpId)
            RETURN;
        END

        -- Insert the employee record
        INSERT INTO Employee (Emp_id, firstName, lastName, Salary, Role, 
streetName, buildingNumber, apartmentNumber, Password)
        VALUES (@EmpId, @FirstName, @LastName, @Salary, @Role, @StreetName, 
@BuildingNumber, @ApartmentNumber, @Password);
END;
```

1. The procedure expects specific information about an employee, such as their employee ID, first name, last name, salary, role, street name, building number, apartment number, and password.

2. It begins by checking if an employee with the same employee ID already exists in the "Employee" table. This is important to avoid creating duplicate employee records.

3. If a duplicate employee ID is found, the stored procedure raises an error and lets you know that an employee with that ID already exists. This ensures that each employee has a unique ID.

4. If no duplicate employee ID is found, the procedure moves on to the next step.

5. It then inserts a new employee record into the "Employee" table using the provided information, including the employee ID, first name, last name, salary, role, street name, building number, apartment number, and password.

EmployeeLogin

```sql
CREATE PROCEDURE EmployeeLogin
    @id INT,
    @Password VARCHAR(100)
AS
BEGIN

    SELECT COUNT(*) AS Count
    FROM Employee
    WHERE Emp_Id = @id AND Password = @Password;
END;
```

1. The stored procedure is named "EmployeeLogin" and expects two inputs: the employee ID (`@id`) and the password (`@Password`).

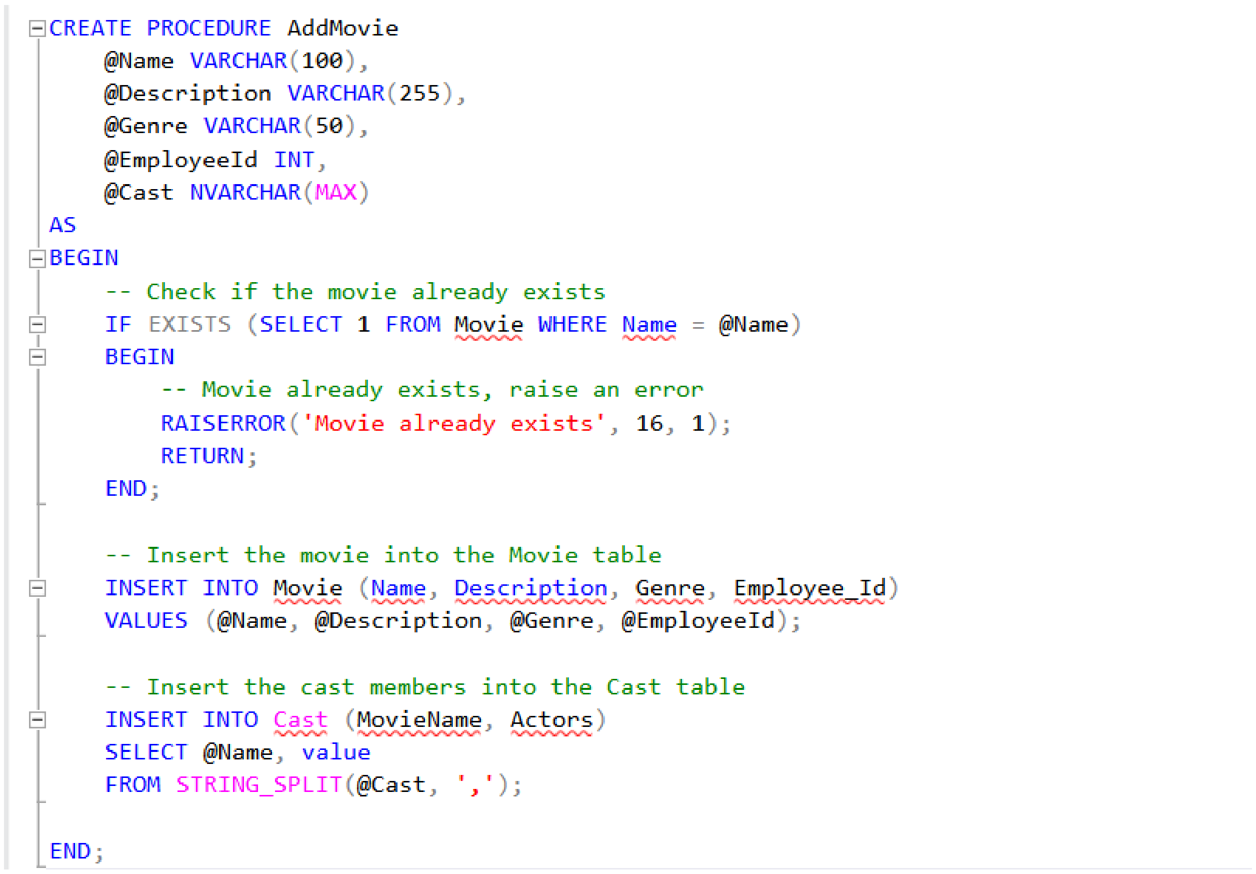
2. When the stored procedure is executed, it performs a search in the "Employee" table to check if there is a matching record where the employee ID matches the provided `@id` and the password matches the provided `@Password`.

3. The stored procedure uses a `SELECT` statement with a `COUNT(*)` function. This function counts the number of rows in the "Employee" table that meet the specified conditions.

4. The result of the `SELECT` statement is a count of matching rows. It represents the number of records in the "Employee" table where both the employee ID and password match the provided values.

5. Finally, the stored procedure returns this count as the result. The count can be used to determine whether the login attempt was successful or not. If the count is greater than zero, it means that a matching record was found, indicating that the login credentials are valid.

AddMovie:



```sql
CREATE PROCEDURE AddMovie
    @Name VARCHAR(100),
    @Description VARCHAR(255),
    @Genre VARCHAR(50),
    @EmployeeId INT,
    @Cast NVARCHAR(MAX)
AS
BEGIN
    -- Check if the movie already exists
    IF EXISTS (SELECT 1 FROM Movie WHERE Name = @Name)
    BEGIN
        -- Movie already exists, raise an error
        RAISERROR('Movie already exists', 16, 1);
        RETURN;
    END;

    -- Insert the movie into the Movie table
    INSERT INTO Movie (Name, Description, Genre, Employee_Id)
    VALUES (@Name, @Description, @Genre, @EmployeeId);

    -- Insert the cast members into the Cast table
    INSERT INTO Cast (MovieName, Actors)
    SELECT @Name, value
    FROM STRING_SPLIT(@Cast, ',');

END;
```

- The stored procedure is created to add a new movie to the database.
- It accepts parameters including the movie name, description, genre, employee ID (presumably the ID of the employee adding the movie),and cast members.
- The procedure begins by checking if a movie with the same name already exists in the database. This is done to prevent the creation of duplicate movie entries.
- If a movie with the same name is found, the procedure raises an error using RAISERROR to notify that the movie already exists, preventing the creation of duplicates. The RETURN statement is then used to exit the procedure.
- If no existing movie with the same name is found, the procedure continues to insert the movie into the Movie table.
- It inserts values for the name, description, genre, and employee ID into the respective columns.
- Additionally, the procedure inserts the cast members into the Cast table. It uses the STRING_SPLIT function to split the comma-separated list of cast members provided as a parameter (@Cast).
- Each cast member is inserted along with the movie name into the Cast table.
- Overall, this stored procedure ensures the addition of a new movie to the database while handling duplicate checks and properly inserting the cast members into the respective tables.

ListMovies

```
CREATE PROCEDURE ListMovies
AS
BEGIN
    SELECT M.Name, M.Description, M.Genre,
        STRING_AGG(C.Actors, ', ') AS Actors
    FROM Movie M
    JOIN Cast C ON M.Name = C.MovieName
    GROUP BY M.Name, M.Description, M.Genre;
END;
```

1. The stored procedure, named "ListMovies," doesn't require any input from the user.

2. When the stored procedure is executed, it retrieves data from two tables: "Movie" and "Cast." These tables store information about movies and the actors involved.

3. The selected columns from the "Movie" table include the movie's name, description, and genre. These details provide a brief overview of each movie.

4. The stored procedure then combines the data from the "Cast" table by joining it with the "Movie" table. This connection is established based on the movie name.

5. The `GROUP BY` clause is used to group the retrieved data based on the movie's name, description, and genre. This ensures that each movie appears only once in the result.

6. To make the information more user-friendly, the stored procedure uses the `STRING_AGG` function. It takes the values from the "Actors" column in the "Cast" table for each movie and concatenates them into a single string. The actors' names are separated by commas, allowing for a neat display of all the actors associated with each movie.

7. Finally, the stored procedure returns the result, which includes the name, description, genre, and a formatted list of actors for each movie.

ListMovieNames

```
CREATE PROCEDURE ListMovieNames
AS
BEGIN
    SELECT Name from Movie
END;
```

1. The stored procedure doesn't require any input from you.

2. When the stored procedure is executed, it performs a search in the database specifically in the "Movie" table.

3. The code block starts with a `SELECT` statement. It fetches the "Name" column from the "Movie" table, which contains the names of the movies.

4. The `SELECT` statement retrieves all the movie names from the "Movie" table.

5. Finally, the stored procedure returns the result, which is a list of all the movie names stored in the database.

ListMovieShowTimes

```
CREATE PROCEDURE ListMovieShowTimes
    @Name VARCHAR(100),
    @HallId INT,
    @Showdate date
AS
BEGIN
    SELECT CONVERT(VARCHAR(5), Time, 108) FROM ShowTime WHERE Movie_Name = @Name
and Hall_Number = @HallId  and Date = @Showdate;
END;
```

1. The stored procedure expects three pieces of information from you: the name of the movie (`@Name`), the hall number (`@HallId`), and the date (`@Showdate`) for which you want to check the show times.

2. When the stored procedure is executed, it performs a search in the database, specifically in the "ShowTime" table.

3. The code block starts with a `SELECT` statement. It retrieves the "Time" column from the "ShowTime" table, which represents the show times.

4. To make the show times more user-friendly, the `CONVERT(VARCHAR(5), Time, 108)` function is used. It converts the time to a string format in the "HH:MM" (hour:minute) representation using the 24-hour clock. For example, it would display 09:30 for 9:30 AM or 14:45 for 2:45 PM.

5. The `SELECT` statement fetches the show times from the "ShowTime" table.

6. The `WHERE` clause is used to filter the show times based on three conditions. It ensures that the movie name (`Movie_Name`) matches the provided `@Name`, the hall number (`Hall_Number`) matches the provided `@HallId`, and the date (`Date`) matches the provided `@Showdate`.

7. Finally, the stored procedure returns the result, which includes the show times of the specified movie in the given hall on the provided date.

ListMovieShowDates

```
CREATE PROCEDURE ListMovieShowDates
    @Name VARCHAR(100),
    @HallId int
AS
BEGIN
    SELECT Date from ShowTime where Movie_Name = @Name and Hall_Number = 
@HallId
END;
```

1. **Naming and Input**: Our procedure is designed to help you find out when a specific movie is playing in a particular hall. It needs two pieces of info:
    - The movie's name (**@Name**).
    - The hall's ID (**@HallId**).
2. **Search Process**: When you use our procedure, it gets to work by searching through the database, focusing on the "ShowTime" table.

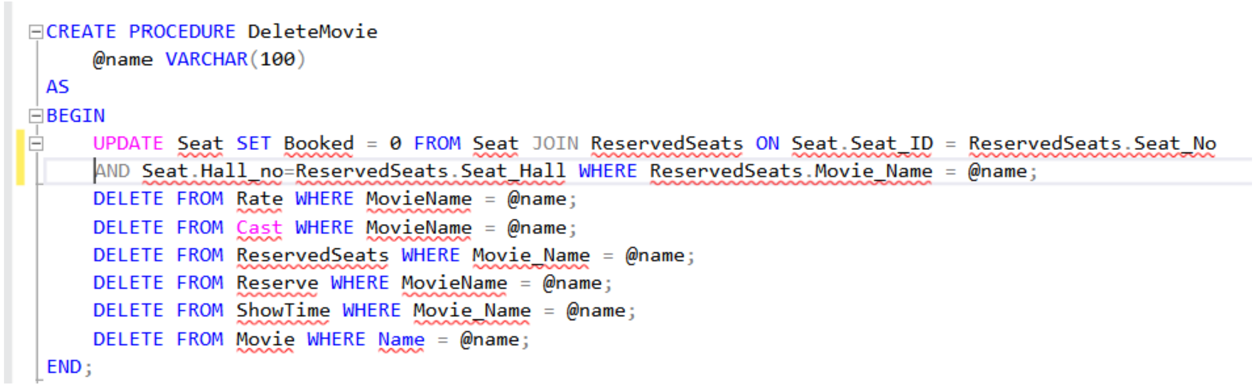
3. **Selecting Data**: Inside our procedure, there's a **SELECT** statement. It's like a spotlight that shines on the "Date" column in the "ShowTime" table, which holds the dates of movie showings.
4. **Formatting for Clarity**: To make the dates easier to read, we convert them into a more user-friendly format. It's like putting them into a readable "MM/DD/YYYY" style.
5. **Filtering the Results**: Our procedure doesn't just give you all the dates; it filters them based on three things:
    - The movie name (**@Name**) you provided.
    - The hall number (**@HallId**) you specified.
    - The specific date (**Date**) you're interested in.
6. **Providing the Result**: Once everything is sorted out, our procedure hands you back the list of dates when your chosen movie is showing in the selected hall. It's like having a schedule tailor-made just for you.

ListMovieShowHalls

```
CREATE PROCEDURE ListMovieShowHalls
    @Name VARCHAR(100)
AS
BEGIN
    SELECT Hall_Number from ShowTime where Movie_Name = @Name
END;
```

1. **Naming and Input**: This procedure only needs one piece of information:
    - The name of the movie you're interested in (**@Name**).
2. **Scanning the Database**: When you call upon this procedure, it scans through the "ShowTime" table in the database.
3. **Picking Relevant Data**: Inside the procedure, there's a **SELECT** statement. It searches the "Hall_Number" column in the "ShowTime" table, which holds the numbers of halls where the movie is playing.
4. **Filtering for the Movie**: This procedure isn't interested in all the halls; it's interested in the ones showing your chosen movie. So, it filters based on the movie name you provided (**@Name**).
5. **Handing Over the Information**: Once the filtering is done, the procedure hands you back the list of hall numbers where your selected movie is currently playing.

DeleteMovie:



```sql
CREATE PROCEDURE DeleteMovie
    @name VARCHAR(100)
AS
BEGIN
    UPDATE Seat SET Booked = 0 FROM Seat JOIN ReservedSeats ON Seat.Seat_ID = ReservedSeats.Seat_No
    AND Seat.Hall_no=ReservedSeats.Seat_Hall WHERE ReservedSeats.Movie_Name = @name;
    DELETE FROM Rate WHERE MovieName = @name;
    DELETE FROM Cast WHERE MovieName = @name;
    DELETE FROM ReservedSeats WHERE Movie_Name = @name;
    DELETE FROM Reserve WHERE MovieName = @name;
    DELETE FROM ShowTime WHERE Movie_Name = @name;
    DELETE FROM Movie WHERE Name = @name;
END;
```

The stored procedure is designed to delete a movie from the database along with related information such as seat reservations, ratings, cast members, showtimes, and reservations.

It first updates the Seat table to mark booked seats for the movie as available (Booked = 0). This is done by joining the Seat table with the ReservedSeats table to identify seats reserved for the movie and updating their status.

It then deletes records from the Rate table where the movie name matches the input parameter (@name), removing any associated ratings.

Next, it deletes records from the Cast table where the movie name matches the input parameter, removing cast information for the movie.

It also deletes records from the ReservedSeats table where the movie name matches the input parameter, removing seat reservations for the movie.

Similarly, it deletes records from the Reserve table where the movie name matches the input parameter, removing additional reservation information.

It deletes records from the ShowTime table where the movie name matches the input parameter, removing associated showtime information.

Finally, it deletes the movie record from the Movie table where the movie name matches the input parameter, effectively removing the movie from the database.

SelectMovie

```sql
CREATE PROCEDURE selectMovie
```

```sql
    @name VARCHAR(100)
AS
BEGIN
    SELECT * FROM Movie WHERE Name = @name;
END;
```

This procedure simply selects all columns in the movie and returns them based on the given movie.

AddShowTime

```sql
CREATE PROCEDURE AddShowTime
  @Time TIME,
  @Date DATE,
  @MovieName VARCHAR(100),
  @HallNumber INT
AS
BEGIN
  -- Check if the ShowTime already exists
  IF EXISTS (
    SELECT 1
    FROM ShowTime
    WHERE Time = @Time
      AND Date = @Date
      AND Movie_Name = @MovieName
      AND Hall_Number = @HallNumber
  )
  BEGIN
    -- ShowTime already exists, raise an error
    RAISERROR('ShowTime already exists', 16, 1);
    RETURN;
  END;

  -- Insert the ShowTime into the ShowTime table
  INSERT INTO ShowTime (Time, Date, Movie_Name, Hall_Number)
  VALUES (@Time, @Date, @MovieName, @HallNumber);
END;
```

1. **Purpose**: You schedule new movie showings.
2. **Input Parameters**:
    - **@Time**: You tell the procedure what time the movie will start.
    - **@Date**: You provide the date when the movie will be shown.
    - **@MovieName**: You specify the name of the movie you're scheduling.

- **@HallNumber**: You indicate which hall will host the movie.

3. **Checking for Existing Showtime**:
    - Before adding a new showing, the procedure checks if there's already a showing scheduled for the same time, date, movie, and hall.
    - If it finds such a duplicate, it stops right there and lets you know that you're trying to schedule a showing that already exists. No double bookings allowed!
4. **Adding the Showtime**:
    - Assuming there's no duplicate, the procedure proceeds to add the new showing to the schedule.
    - It puts all the details you provided—time, date, movie name, and hall number—into the "ShowTime" table.
    - Think of it as officially locking in that time slot for the movie in that particular hall on that specific date.
5. **Error Handling**:
    - If by chance, the procedure discovers a duplicate showtime, it doesn't just silently fail. Instead, it raises a flag, alerting you with a specific error message.
    - This way, you're made aware of the issue and can take corrective action, ensuring your schedule stays accurate .

DeleteShowTime

```
CREATE PROCEDURE DeleteShowTime
  @Time TIME,
  @Date DATE,
  @MovieName VARCHAR(100),
  @HallNumber INT
AS
BEGIN
    UPDATE Seat SET Booked = 0 FROM Seat JOIN ReservedSeats ON Seat.Seat_ID =
ReservedSeats.Seat_No AND Seat.Hall_no=ReservedSeats.Seat_Hall WHERE
ReservedSeats.ShowTime= @Time AND ReservedSeats.ShowDate=@Date AND
ReservedSeats.Hall_No=@HallNumber AND ReservedSeats.Movie_Name=@MovieName;
    DELETE FROM ReservedSeats WHERE ShowTime= @Time AND ShowDate=@Date AND
Hall_No=@HallNumber AND Movie_Name=@MovieName;
    DELETE FROM Reserve WHERE Show_Time = @Time AND Show_Date= @Date AND
MovieName=@MovieName AND Hall_Id= @HallNumber;
    DELETE FROM ShowTime WHERE Time = @Time AND Date = @Date AND Movie_Name =
@MovieName AND Hall_Number = @HallNumber;
END;
```

1. **Purpose**: It tidies up all the related info, like seat reservations, ratings, cast details, and showtimes and deletes showtime.
2. **Seat Status Update**:
    - First off, it makes sure to free up any seats that were booked for the movie. It's like resetting the seats back to "available" status so they can be booked for other movies.
    - This part works by checking which seats were reserved for the movie and then updating their status to show that they're no longer booked.
3. **Removing Ratings**:
    - Next up, it gets rid of any ratings or reviews that movie might have accumulated. It's like erasing the scorecards after the game's over, so the next movie starts with a clean slate.
4. **Eliminating Cast Information**:
    - Then, it wipes out all the details about the cast involved in the movie..
5. **Seat Reservation Cleanup**:
    - Additionally, it clears out all the seat reservations made for the movie. It's like canceling all the tickets for that movie so they can be sold for a different show.
6. **Reservation Data Removal**:
    - It also takes care of any other reservation-related data.
7. **Showtime Information Removal**:
    - Then, it sweeps away all the showtime entries associated with the movie. It's like taking down the posters and schedules once the movie's run is over.
8. **Final Movie Deletion**:
    - Lastly, it removes the movie's record from the database entirely.

ListShowTimes

```
CREATE PROCEDURE ListShowTimes
AS
BEGIN
    SELECT CONVERT(VARCHAR(5), Time, 108), Date, Movie_Name, Hall_Number FROM
ShowTime
```

```
END;
```

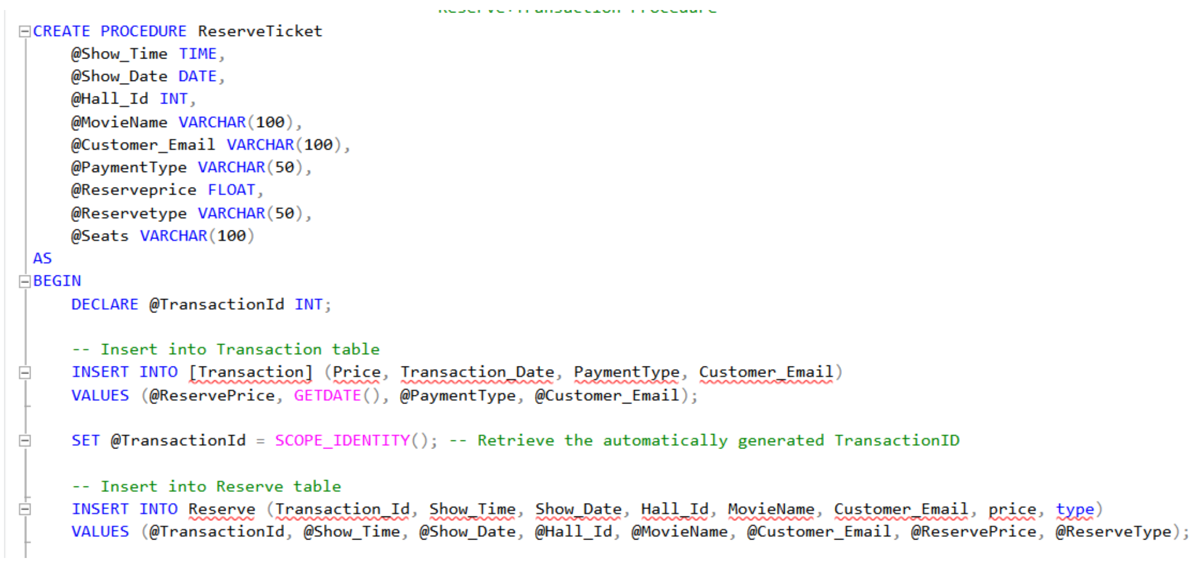
1. **Purpose**: It's there to give you a list of all the movies playing at your cinema, complete with showtimes and hall numbers.
2. **Querying ShowTimes**: Inside this procedure, there's a smart query that goes through your database's "ShowTime" records.
3. **Selecting Data**: The procedure selects four key details for each showtime:
   - The time when each movie starts, but we presented in a way that's easy to read, like "2:30 PM" instead of a technical time format.
   - The date of the show, so you know which day the movie is playing.
   - The name of the movie, because you definitely want to know what you're watching.
   - The hall number, so you know exactly where to go in the cinema.
4. **Formatting for Readability**:
   - To make things even friendlier, the time is converted into a format that's clear and familiar, like how you'd see it on a regular clock.
5. **Presenting the Results**:
   - Once everything's sorted out, the procedure hands you back the schedule. It's like a well-organized list that you can easily check to plan your movie night.

addRating

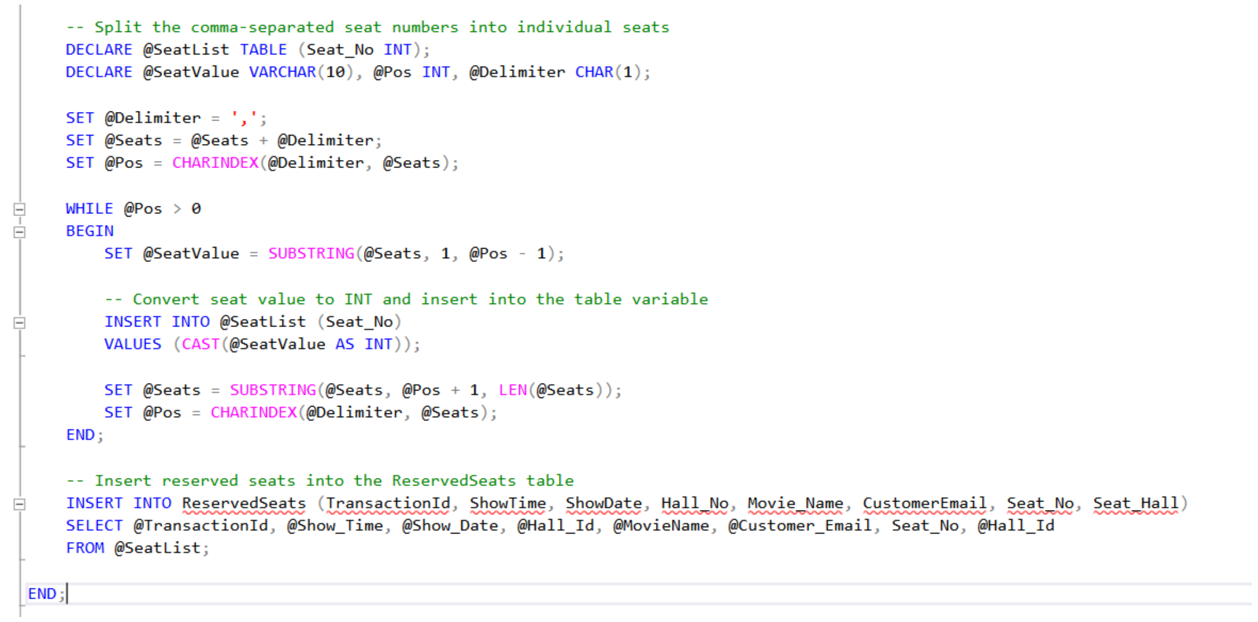
```
CREATE PROCEDURE addRating
    @MovieName VARCHAR(100),
    @CustomerEmail VARCHAR(100),
    @Rating INT ,
    @Comment VARCHAR(250)
AS
BEGIN
    -- Check if the rating already exists
    IF NOT EXISTS (
        SELECT 1 FROM Rate WHERE MovieName = @MovieName AND CustomerEmail = 
@CustomerEmail
    )
    BEGIN
        -- Insert the rating into the Rate table
        INSERT INTO Rate (MovieName, CustomerEmail, Rating, Comment)
        VALUES (@MovieName, @CustomerEmail, @Rating, @Comment);
    END
    ELSE
    BEGIN
        PRINT 'Rating already exists';
    END
END;
```

1. **Purpose**: This procedure is like a comment card system for your movies. It lets customers rate and leave comments about the movies they've watched.
2. **Input Parameters**:
   - **@MovieName**: Specifies the name of the movie the rating is for.
   - **@CustomerEmail**: Identifies the email of the customer who's giving the rating.
   - **@Rating**: Represents the numerical rating given by the customer.
   - **@Comment**: Allows the customer to add a comment along with their rating.
3. **Checking for Existing Rating**:
   - Before adding a new rating, the procedure first checks if the customer has already rated the movie.
   - It does this by searching the "Rate" table to see if there's already a record with the same movie name and customer email.
4. **Adding the Rating**:
   - If the customer hasn't already rated the movie, the procedure proceeds to add their rating and comment to the "Rate" table.
   - It's like dropping their comment card into the box for the first time.
5. **Error Handling**:
   - If the procedure finds that the customer has already rated the movie, it doesn't allow them to submit another rating.
   - Instead, it prints a message saying "Rating already exists," indicating that their rating has already been recorded.

ReserveTicket:



```sql
CREATE PROCEDURE ReserveTicket
    @Show_Time TIME,
    @Show_Date DATE,
    @Hall_Id INT,
    @MovieName VARCHAR(100),
    @Customer_Email VARCHAR(100),
    @PaymentType VARCHAR(50),
    @Reserveprice FLOAT,
    @Reservetype VARCHAR(50),
    @Seats VARCHAR(100)
AS
BEGIN
    DECLARE @TransactionId INT;

    -- Insert into Transaction table
    INSERT INTO [Transaction] (Price, Transaction_Date, PaymentType, Customer_Email)
    VALUES (@ReservePrice, GETDATE(), @PaymentType, @Customer_Email);

    SET @TransactionId = SCOPE_IDENTITY(); -- Retrieve the automatically generated TransactionID

    -- Insert into Reserve table
    INSERT INTO Reserve (Transaction_Id, Show_Time, Show_Date, Hall_Id, MovieName, Customer_Email, price, type)
    VALUES (@TransactionId, @Show_Time, @Show_Date, @Hall_Id, @MovieName, @Customer_Email, @ReservePrice, @ReserveType);
```



```sql
    -- Split the comma-separated seat numbers into individual seats
    DECLARE @SeatList TABLE (Seat_No INT);
    DECLARE @SeatValue VARCHAR(10), @Pos INT, @Delimiter CHAR(1);

    SET @Delimiter = ',';
    SET @Seats = @Seats + @Delimiter;
    SET @Pos = CHARINDEX(@Delimiter, @Seats);

    WHILE @Pos > 0
    BEGIN
        SET @SeatValue = SUBSTRING(@Seats, 1, @Pos - 1);

        -- Convert seat value to INT and insert into the table variable
        INSERT INTO @SeatList (Seat_No)
        VALUES (CAST(@SeatValue AS INT));

        SET @Seats = SUBSTRING(@Seats, @Pos + 1, LEN(@Seats));
        SET @Pos = CHARINDEX(@Delimiter, @Seats);
    END;

    -- Insert reserved seats into the ReservedSeats table
    INSERT INTO ReservedSeats (TransactionId, ShowTime, ShowDate, Hall_No, Movie_Name, CustomerEmail, Seat_No, Seat_Hall)
    SELECT @TransactionId, @Show_Time, @Show_Date, @Hall_Id, @MovieName, @Customer_Email, Seat_No, @Hall_Id
    FROM @SeatList;

END;
```

The procedure begins by inserting a transaction record into the Transaction table with details such as the reservation price, current date, payment type, and the customer's email.

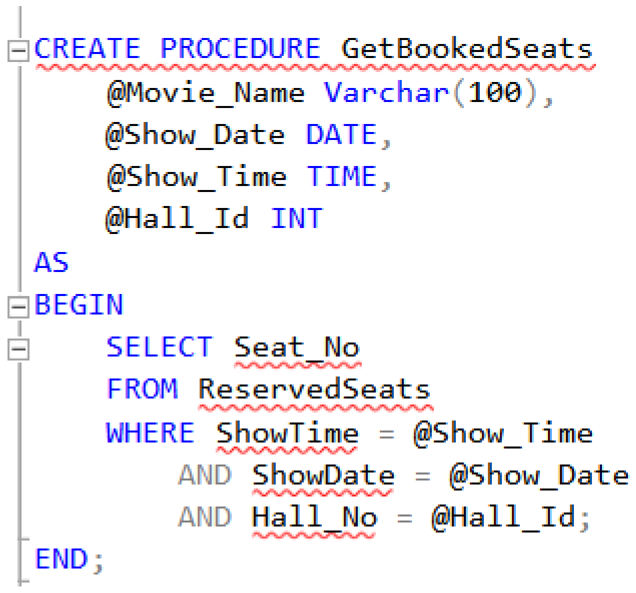
It retrieves the Transaction ID generated by the database after the transaction record is inserted. This ID is crucial for linking all subsequent actions to this financial transaction.

Next, it logs detailed reservation information in the Reserve table using the retrieved Transaction ID. This includes the show time, date, cinema hall ID, movie name, customer email, ticket price, and type of reservation.

The procedure handles seat assignments by processing a comma-separated string of seat numbers provided in the input. It converts each seat number from text to integer and stores them in a temporary table.

Finally, for each seat processed, the procedure inserts a reservation record into the ReservedSeats table, linking each seat with the corresponding show details, Transaction ID, and hall information.

GetBookedSeats:



```
CREATE PROCEDURE GetBookedSeats
    @Movie_Name Varchar(100),
    @Show_Date DATE,
    @Show_Time TIME,
    @Hall_Id INT
AS
BEGIN
    SELECT Seat_No
    FROM ReservedSeats
    WHERE ShowTime = @Show_Time
        AND ShowDate = @Show_Date
        AND Hall_No = @Hall_Id;
END;
```

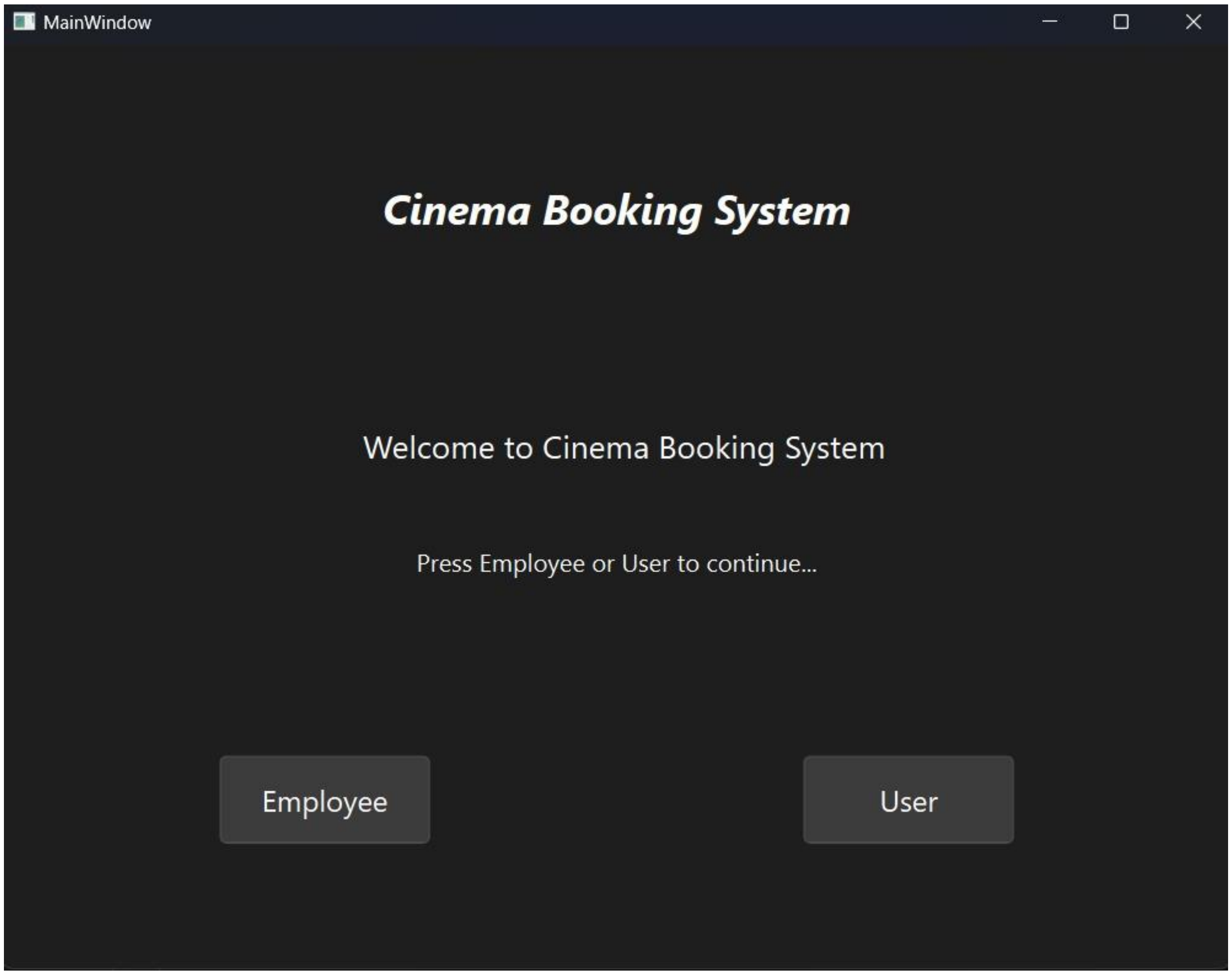
The procedure expects inputs such as the movie name, show date, show time, and hall ID to identify the specific movie show for which seat information is requested.

It queries the ReservedSeats table to fetch the seat numbers that have been reserved for the provided movie show.

It filters the records based on the show time, show date, and hall ID provided as input parameters.

The procedure returns a list of seat numbers that have been booked for the specified movie show.

GUI:

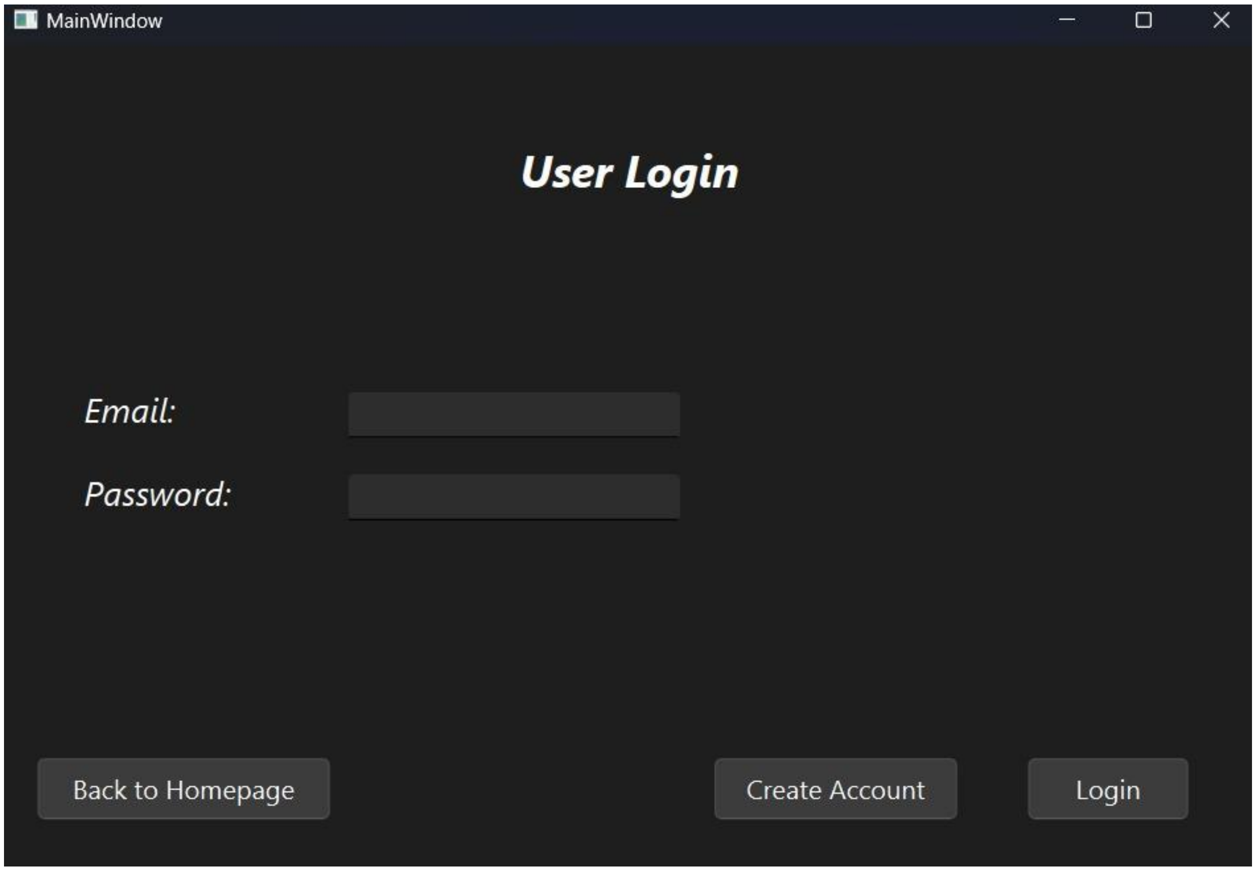


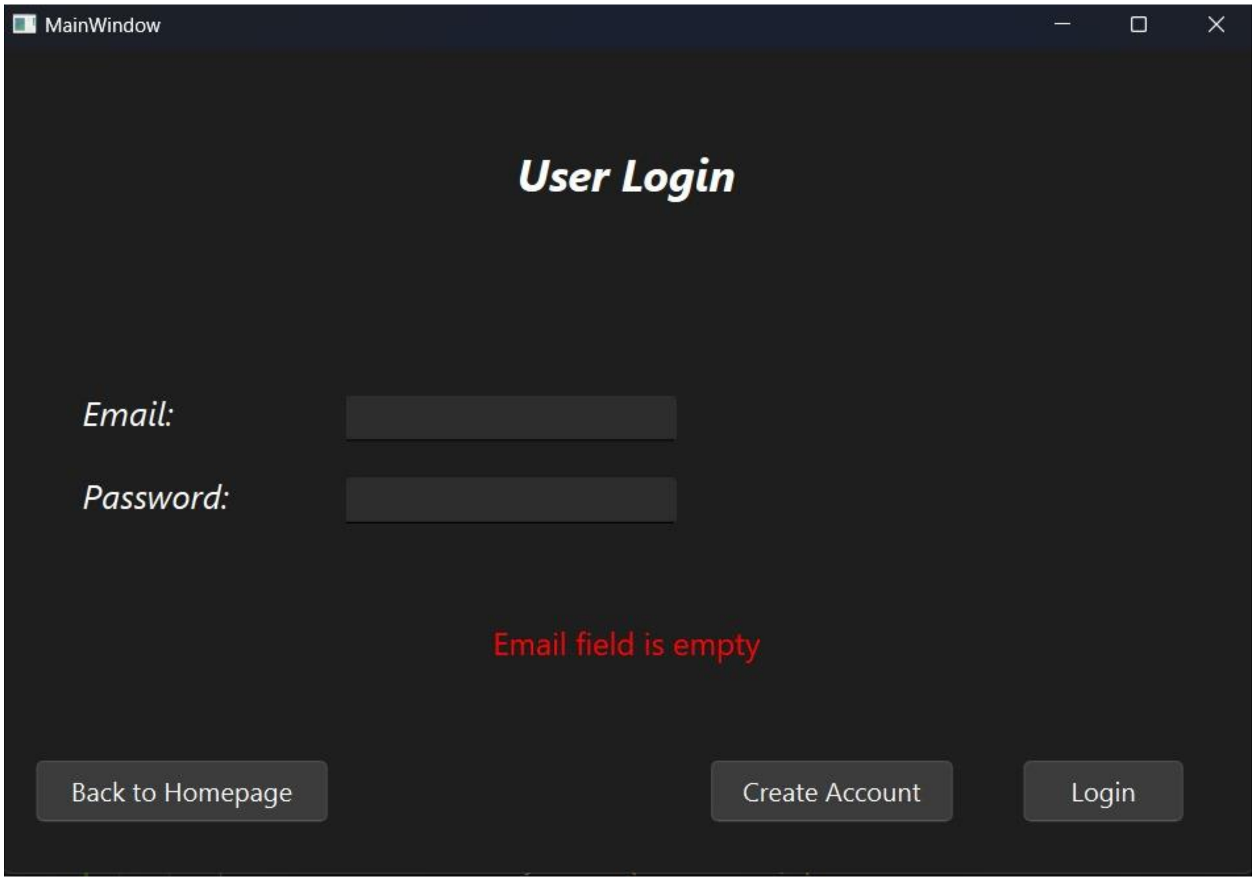
Cinema Booking System

Welcome to Cinema Booking System

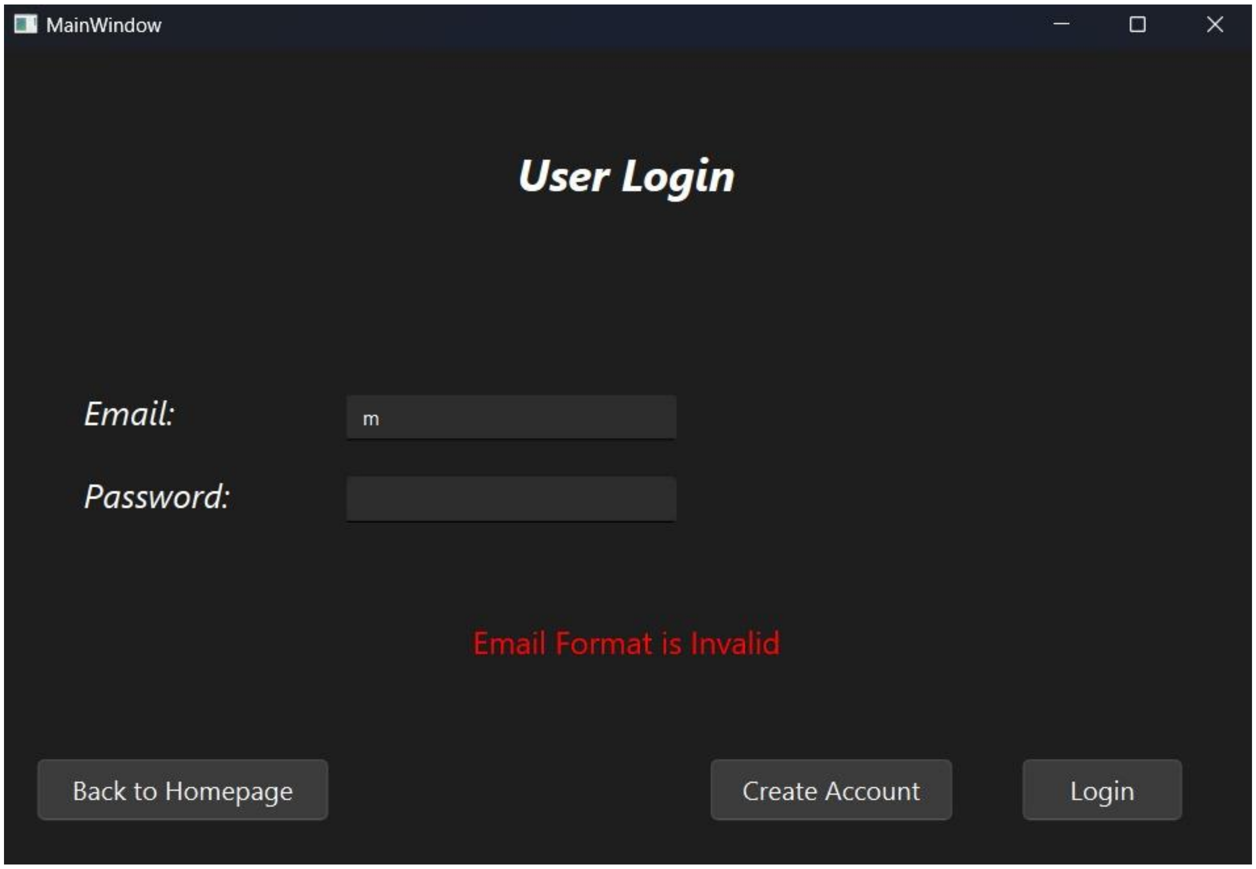
Press Employee or User to continue...

User Interface:

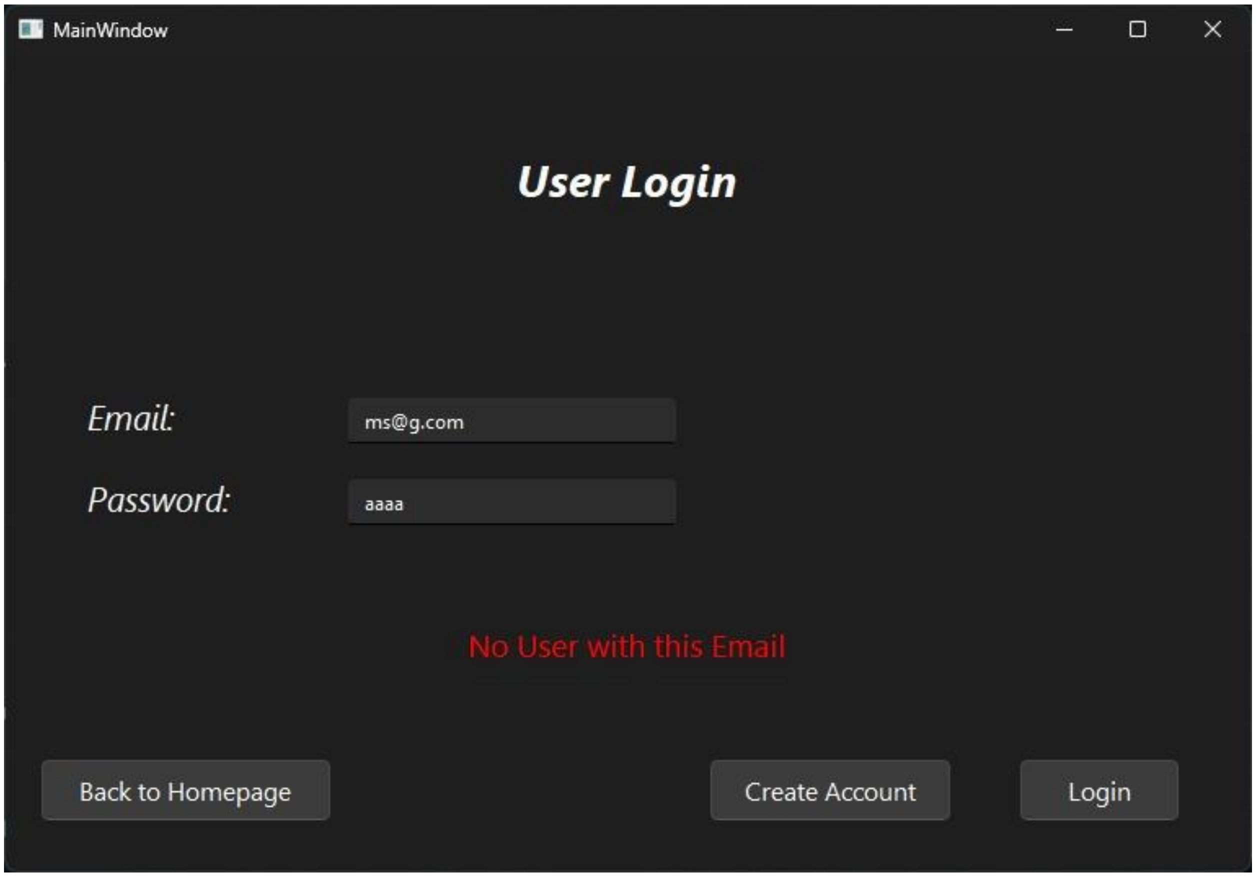




Email field is empty



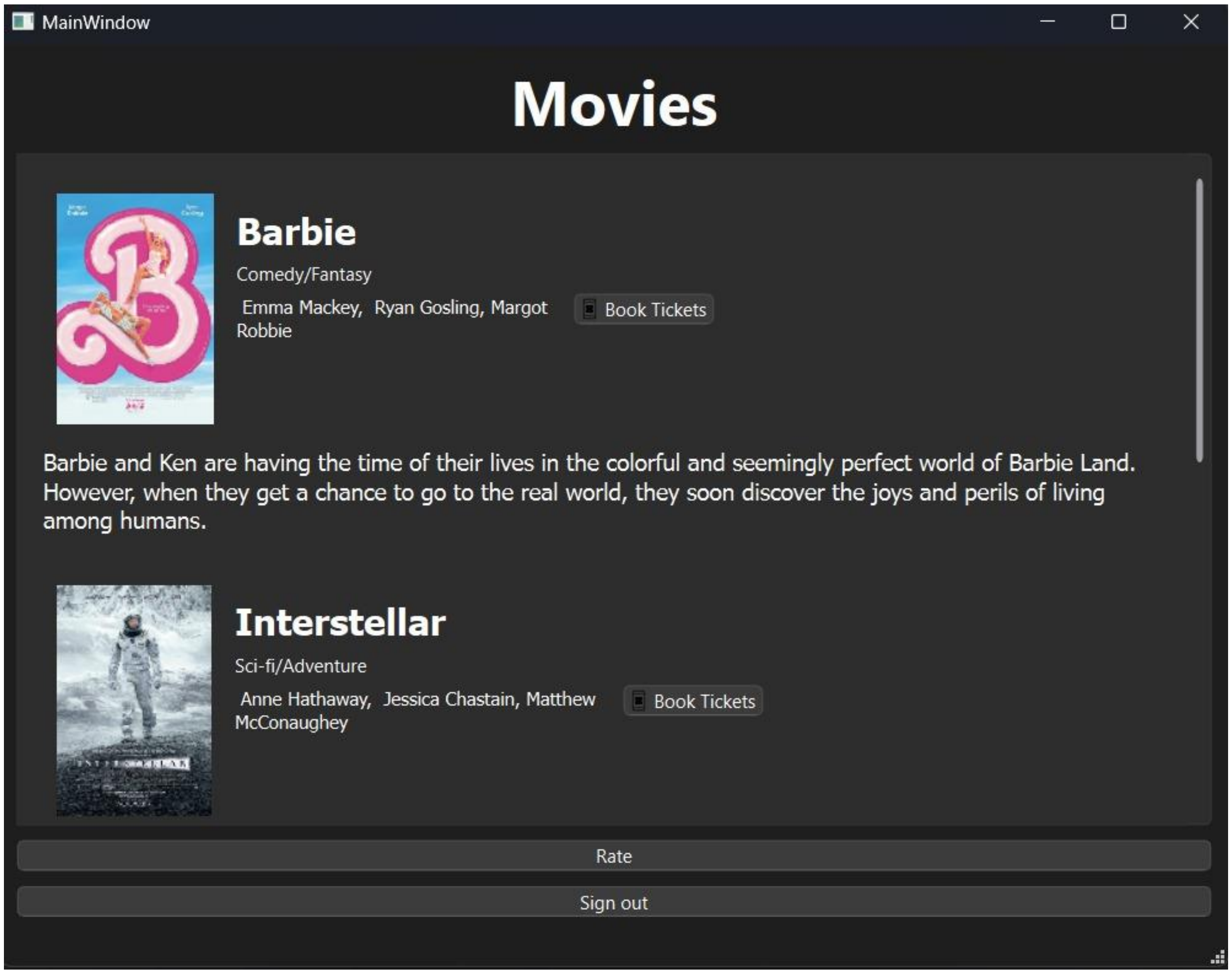
Email Format is Invalid



ms@g.com

aaaa

No User with this Email



Movies

Barbie

Comedy/Fantasy

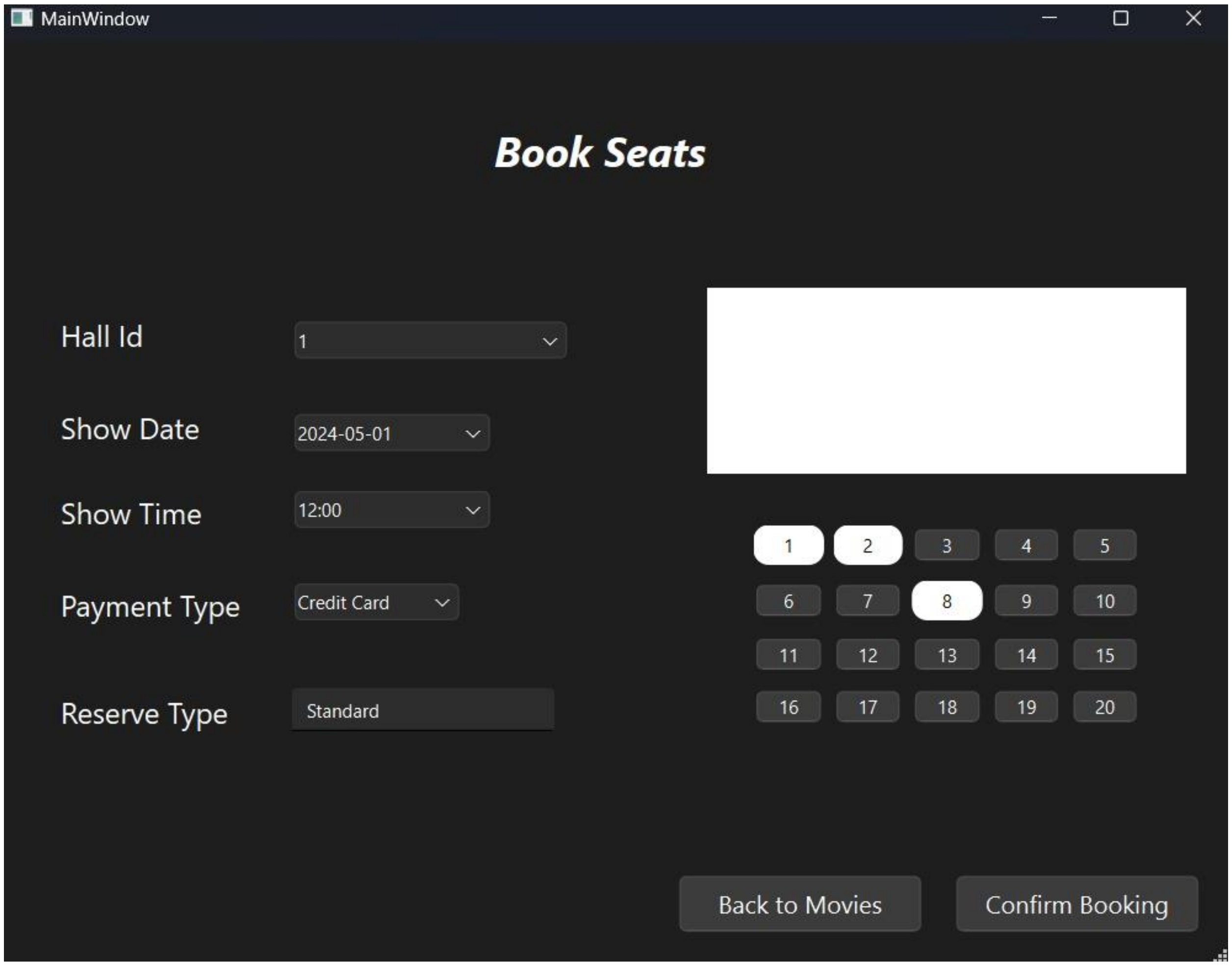
Emma Mackey, Ryan Gosling, Margot Robbie

Barbie and Ken are having the time of their lives in the colorful and seemingly perfect world of Barbie Land. However, when they get a chance to go to the real world, they soon discover the joys and perils of living among humans.

Interstellar

Sci-fi/Adventure

Anne Hathaway, Jessica Chastain, Matthew McConaughey



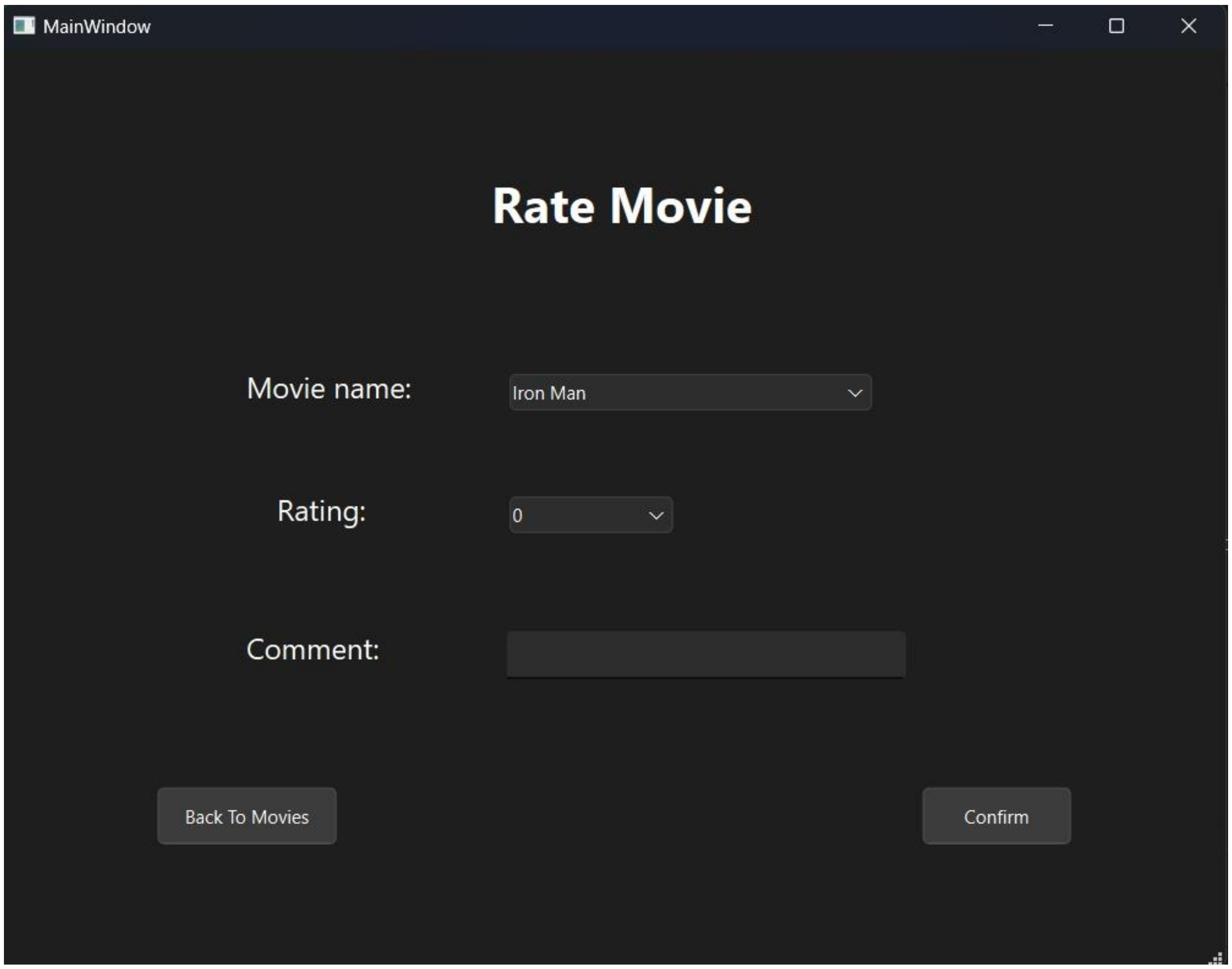
Hall Id: 1

Show Date: 2024-05-01

Show Time: 12:00

Payment Type: Credit Card

Reserve Type: Standard



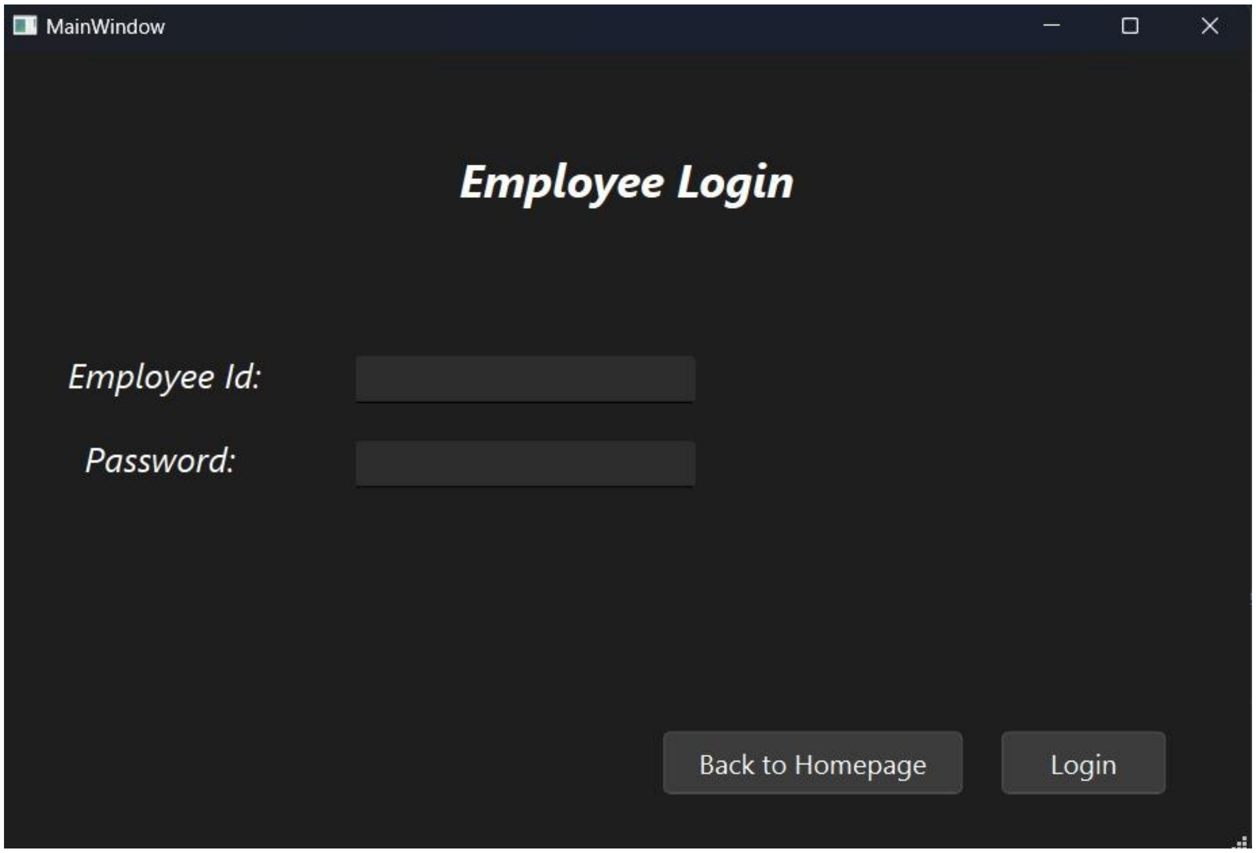
Rate Movie

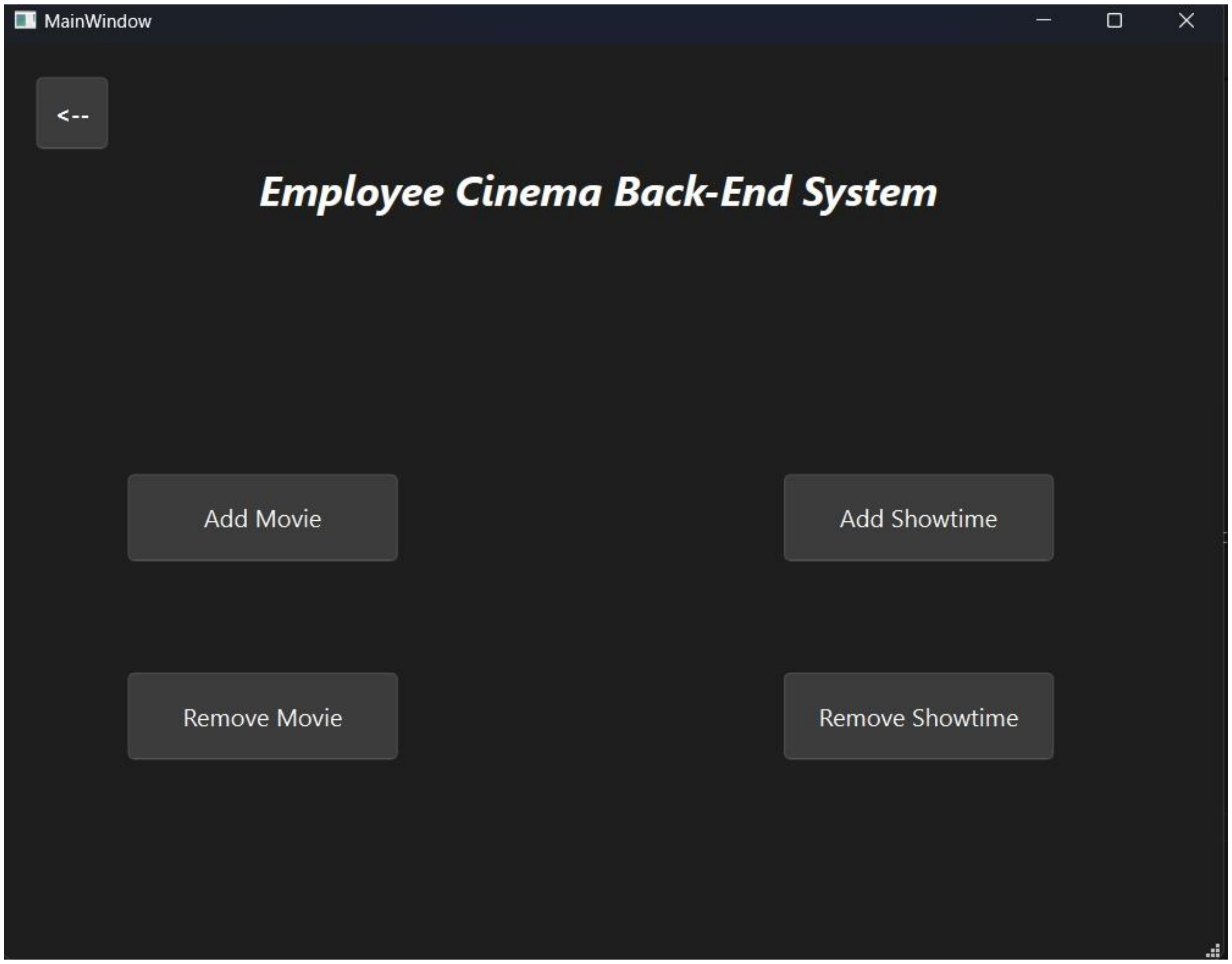
Movie name: Iron Man

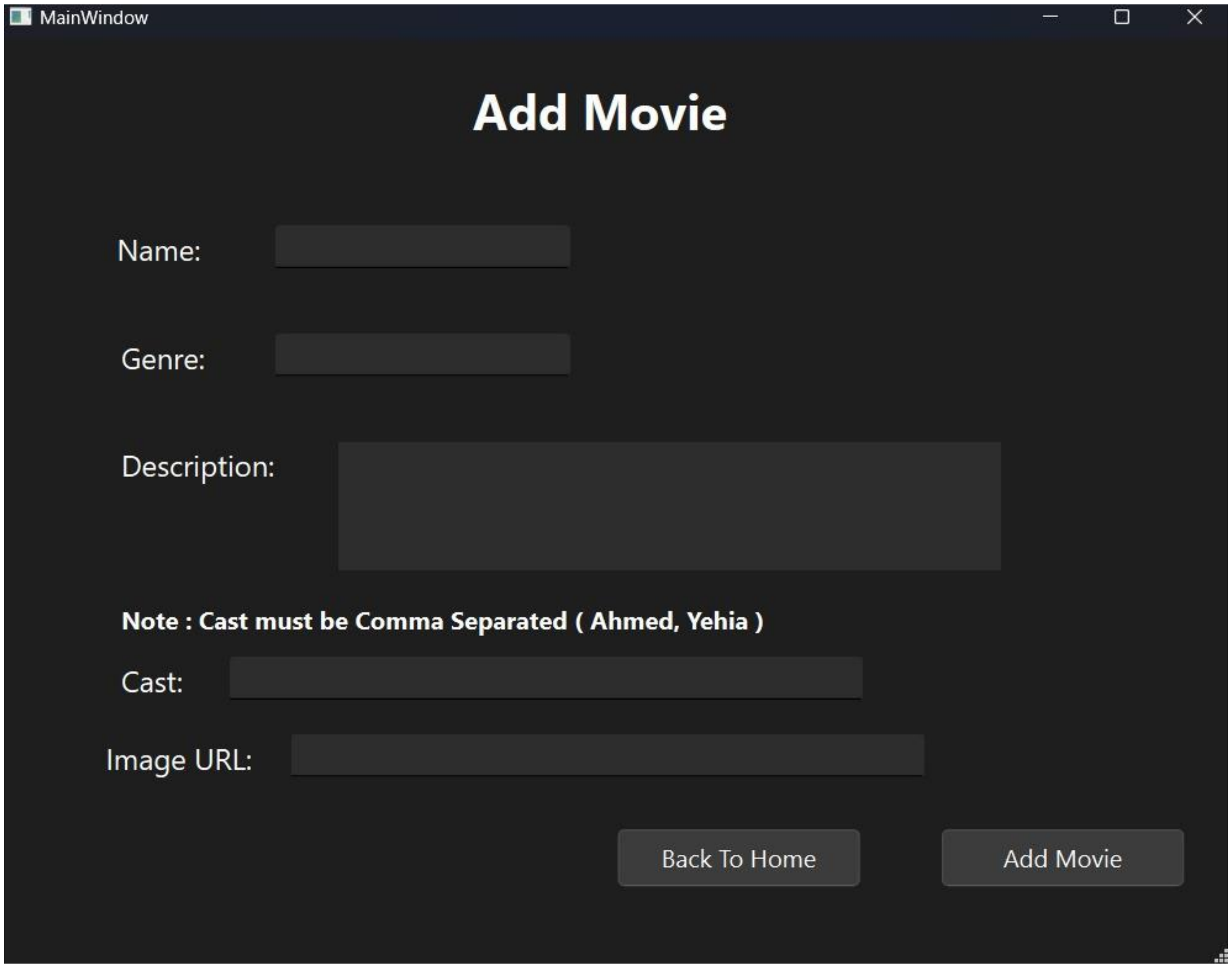
Rating: 0

Comment:

Employee Interface:




MainWindow
<--
Employee Cinema Back-End System
Add Movie
Add Showtime
Remove Movie
Remove Showtime



Add Movie

Name:

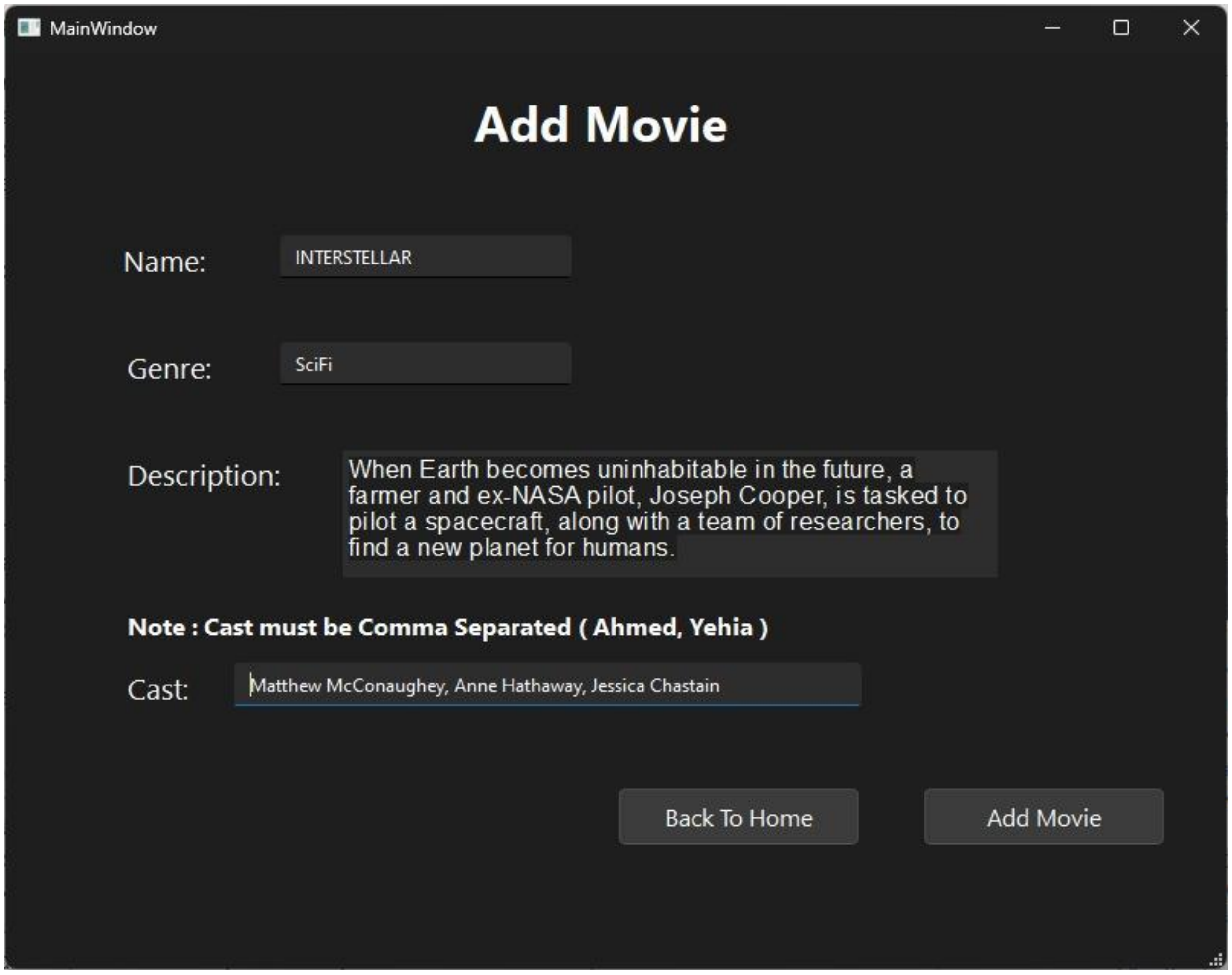
Genre:

Description:

Note : Cast must be Comma Separated (Ahmed, Yehia)

Cast:

Image URL:



Add Movie

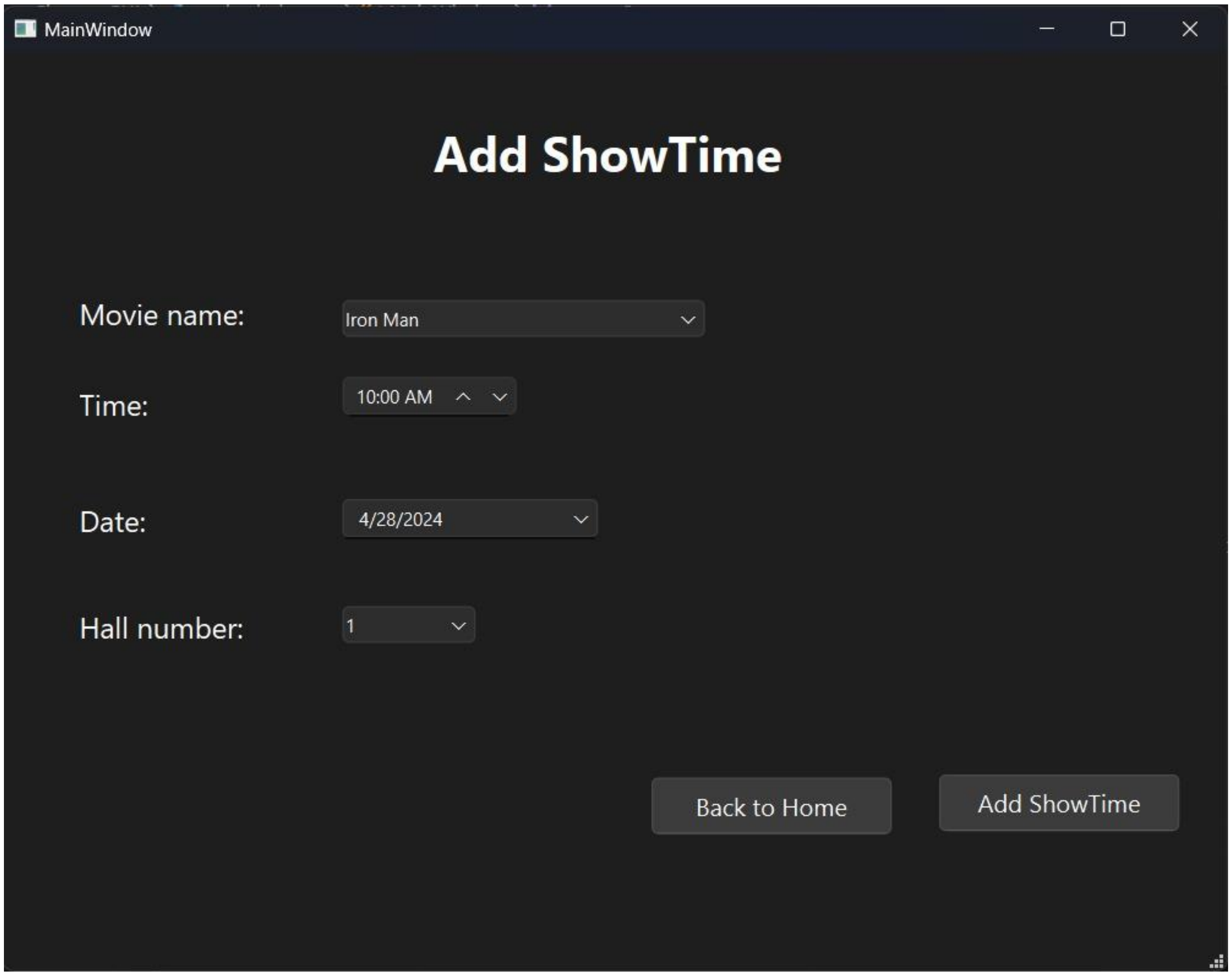
Name: INTERSTELLAR

Genre: SciFi

Description: When Earth becomes uninhabitable in the future, a farmer and ex-NASA pilot, Joseph Cooper, is tasked to pilot a spacecraft, along with a team of researchers, to find a new planet for humans.

Note : Cast must be Comma Separated (Ahmed, Yehia)

Cast: Matthew McConaughey, Anne Hathaway, Jessica Chastain



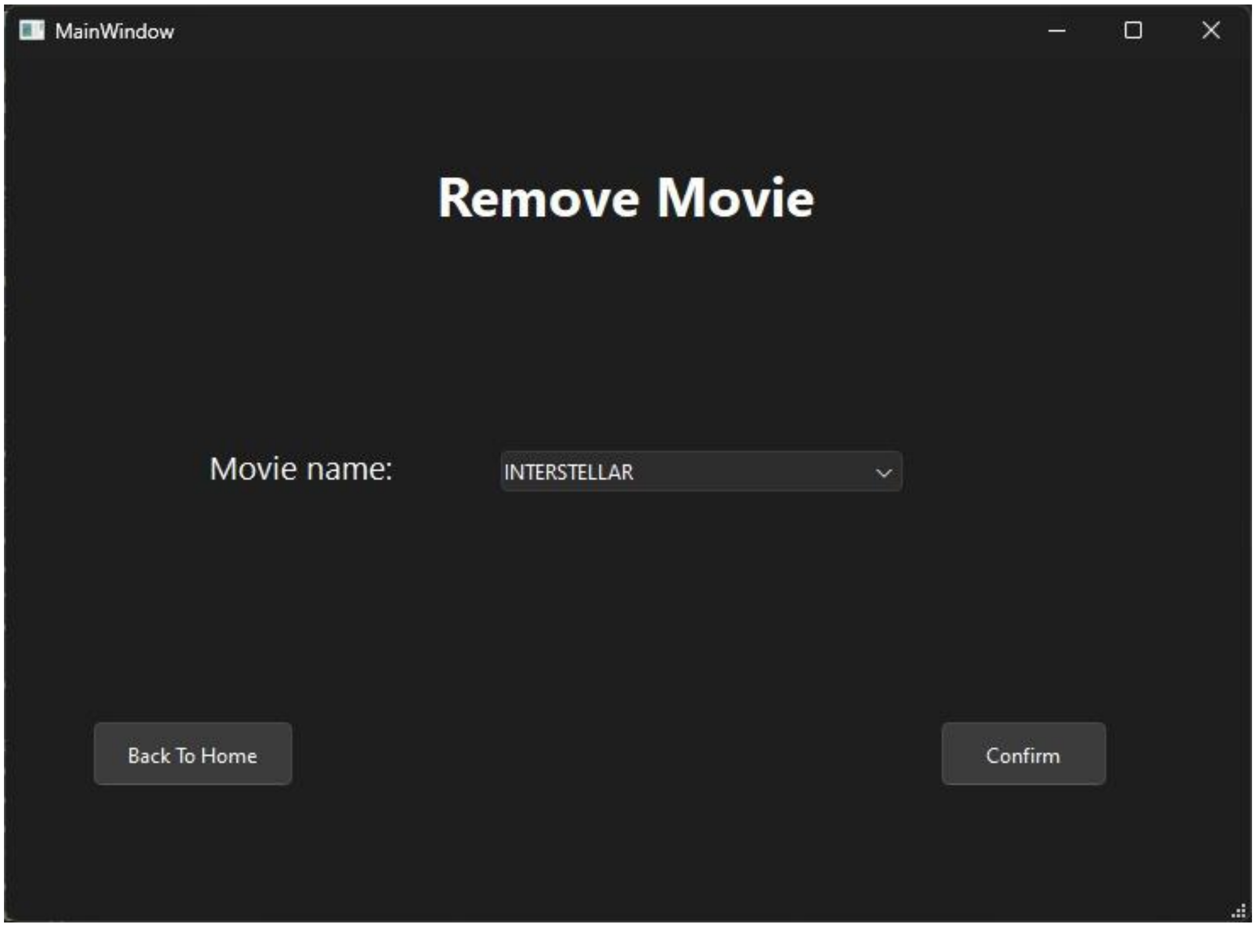
Add ShowTime

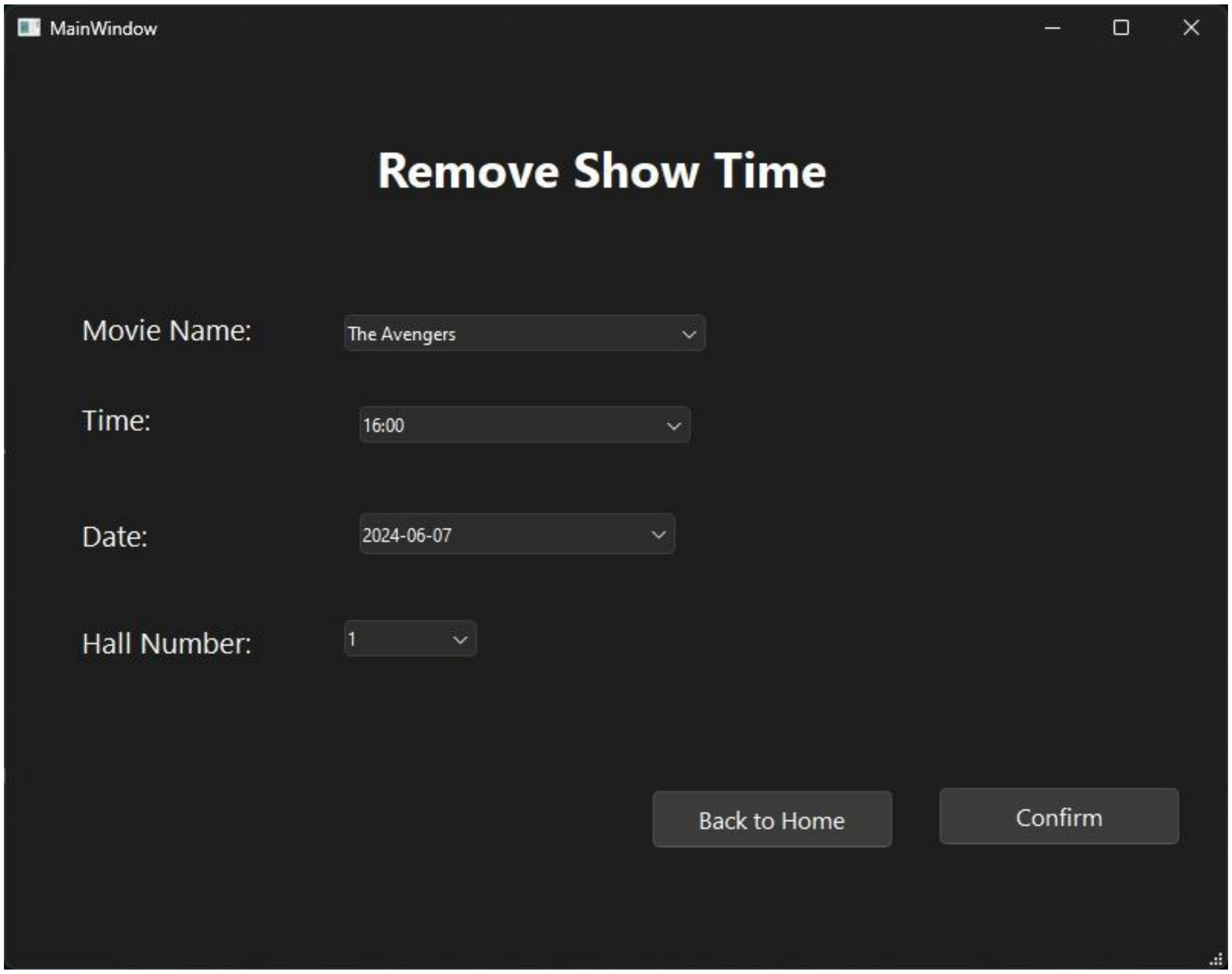
Movie name: Iron Man

Time: 10:00 AM

Date: 4/28/2024

Hall number: 1


MainWindow
Remove Movie
Movie name:
INTERSTELLAR
Back To Home
Confirm



Remove Show Time

Movie Name: The Avengers

Time: 16:00

Date: 2024-06-07

Hall Number: 1