



**CSE 414 DATABASE
SPRING 2024
TERM PROJECT**

“THE ENCHANTING PRODUCTIONS”

**MERT GÜRŞİMŞİR
1901042646**

CONTENTS

CONTENTS.....	2
INTRODUCTION	3
USER REQUIREMENTS	4
ENTITY-RELATIONSHIP (E-R) DIAGRAM	6
FUNCTIONAL DEPENDENCIES	8
NORMALIZATION.....	9
DATABASE SCHEMA	11
DATABASE IMPLEMENTATION	12
USER INTERFACE	16
MAIN SCREEN	16
ENROLLMENT	17
AUDIENCE PAGE	18
DIRECTOR PAGE	19
MANAGER PAGE	22
QUERY DEVELOPMENT	24
TRIGGERS	27
FIRST TRIGGER	27
SECOND TRIGGER	28
THIRD TRIGGER.....	29
FOURTH TRIGGER.....	30
FIFTH TRIGGER	31
VIEWS	32
FIRST VIEW	32
SECOND VIEW.....	32
THIRD VIEW	33
FOURTH VIEW	33
FIFTH VIEW.....	33
TRANSACTIONS	34
FIRST TRANSACTION.....	34
SECOND TRANSACTION	35
THIRD TRANSACTION.....	36
FOURTH TRANSACTION	37
FIFTH TRANSACTION	38
SIXTH TRANSACTION	39
CONCURRENCY CONTROL	40
INHERITANCE	40
PRIVILEGES AND ROLES	41
SECURITY	41
SQL INJECTION	41
KEEPING PASSWORDS	42

INTRODUCTION

The project's aim is creating the comprehensive database for a Theatre Production Company called "Enchanting Productions". My company aims to bringing magic of theatre to life. As a theatre production company, it specializes in staging plays that leave audiences mesmerized.

Dedicated team of directors, actors, managers, and audience work collaboratively to bring stories to the stage. Comprehensive approach to theatre includes also the logistical and operational aspects. The database structure of the company is managed with every detail, from casting and rehearsals to ticket sales and audience engagement. This advanced database system allows us to coordinate schedules and gather feedback from our audience.

With the comprehensive database, we are aiming enhanced operational efficiency, collaboration and communication, superior audience engagement and management, scalability and growth. By centralizing all the data, our aim is reducing the administrative difficulties. Besides, collecting data from audience feedback helps us improve our performances and meet the expectations of our audience.



The database is needed to keep the followings:

ENTITY	DESCRIPTION
Production Information	To keep information about the play.
Cast and Crew	To keep the members of the plays with their roles.
Venue Information	To keep information about the venues.
Performance Schedule	To keep information about the scheduled individual performances.
Ticket Sales	To keep information about the tickets.
Audience Comments	To keep information about the comment entries.
Manager	To keep information about the managers of venues.
Director	To keep information about the directors of the productions.
Audience	To keep information about the audience of the performances.

USER REQUIREMENTS

Specific requirements of the company are going to be listed in this section. You can see the requirements below:

- Relational database is going to be used. Data is going to be stored in a structured format with tables and relations representing entities that is named at the table above.
- Data normalization is going to be ensured so that minimizing redundancy and maintaining data integrity.
- JDBC is going to be used to connect database to the Java programming language.
- JavaFX is going to be used for user interface.
- PostgreSQL is going to be used as RDBMS.
- IntelliJ is going to be used as IDE.
- Querying: Allow users to search using various filters via generated user interface.
- Real-time access to current data, such as ticket availability and performance schedules.
- Defining user roles with specific permissions to access or modify data.
- Data must be secure and confidential.
- Confidentiality, consistency, and integrity of the data is provided by the triggers.

- Production management:
 - Record and access detailed information about each production
 - Create and manage performance schedules
 - Other than performances, manage also venue, cast and director
- Cast and crew management:
 - Create and manage profiles for each cast member
- Venue management:
 - Record information about venues
 - Maintain contact details about the manager of the venue
 - Manage performances held at the venues
- Performance schedule:
 - Schedule performances and events at different venues
 - Set the desired number of tickets to be available
 - Each performance is a realized version of the production
- Ticket sales:
 - Manage the tickets for the performance
 - Assign tickets to audience with seat number
- Audience comments:
 - Collect comment and ratings for productions
- Person (INHERITANCE)
 - This is the high-level schema for the audience, manager, cast member, and director
 - The four entities are inherited from the Person entity
 - While forming schema for each lower-level entity set, only included the primary key of higher-level entity set and local attributes to eliminate data redundancy

These were the requirements for the database of our company, The Enchanting Productions. While creating the database, the normalization rules (3NF and BCNF) are going to be applied. With this way, I am aiming to eliminate the data redundancy and unrelated product data being gathered together in the same table.

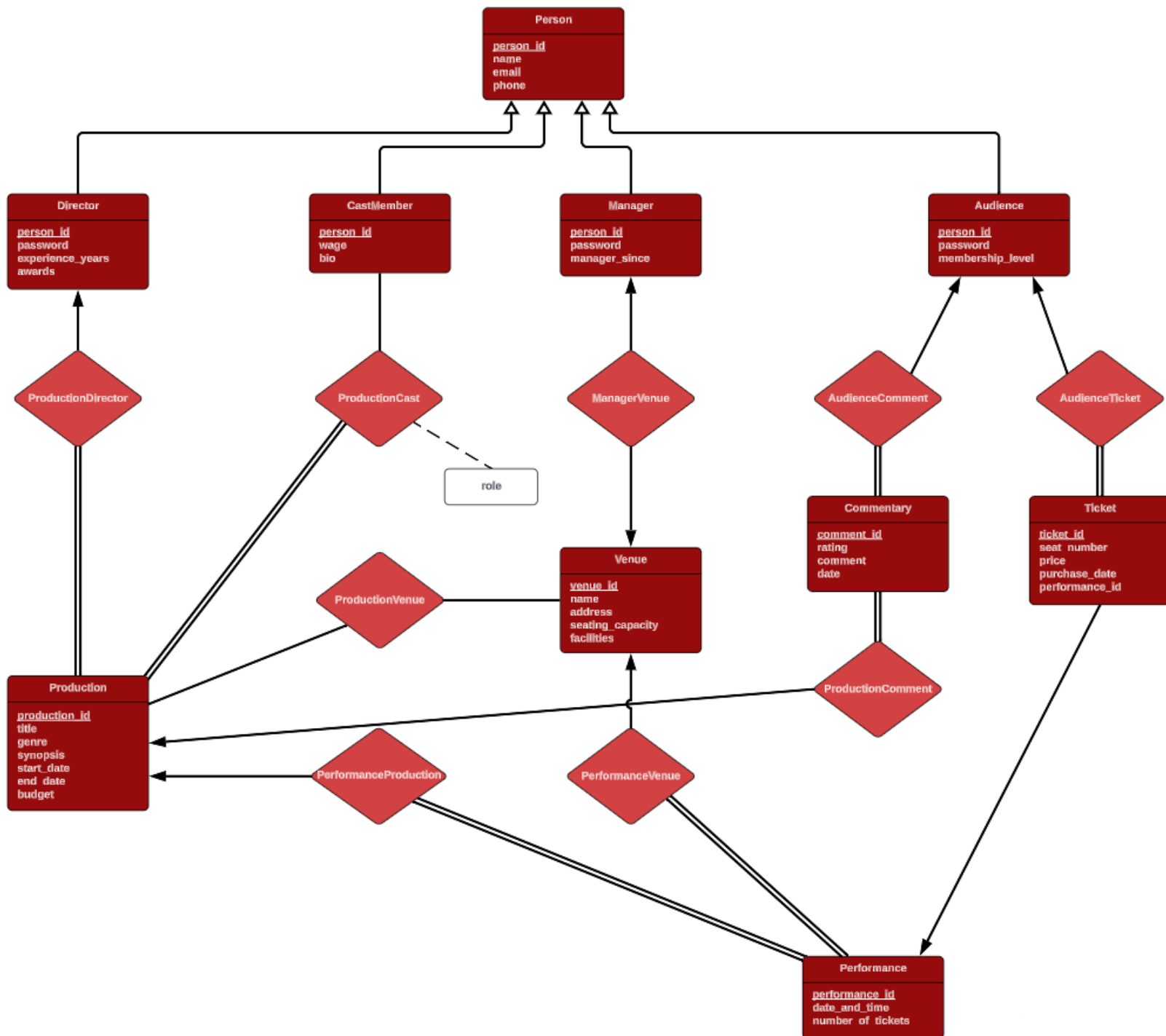
ENTITY-RELATIONSHIP (E-R) DIAGRAM

Before developing the E-R diagram illustrating the relationships between various entities within the company, firstly I am going to list the relationships with cardinalities below:

RELATION	CARDINALITY CONSTRAINT	DESCRIPTION
Production – Performance	one-to-many	One production can have many performances.
Production – CastMember	many-to-many	One production can have many members and vice versa.
Production – Director	many-to-one	One director can direct many productions.
Production – Venue	many-to-many	One production can take place in many venues and vice versa.
Venue – Manager	one-to-one	One venue can have only one manager.
Venue – Performance	one-to-many	One venue can host multiple performances.
Production – Commentary	one-to-many	One production can have many audience comments.
Audience – Ticket	one-to-many	One audience can have many tickets.
Audience – Commentary	one-to-many	One audience can give many comments.

There are total of 10 tables and 9 relations in the database. For each of these relationships, there are relation tables. In the entity relationship diagram, all the relationships between various entities within the company can be seen.

Entity relationship diagram:



Director, CastMember, Manager, and Audience entities are inherited from entity Person. This inheritance is overlapping as it can be seen at the diagram.

FUNCTIONAL DEPENDENCIES

In this section, the functional dependencies that are present within the database, outlining how attributes depend on one another at the tables are going to be shown one by one.

Production

- production_id → title, genre, synopsis, start_date, end_date, budget
- title → genre, synopsis

Person

- person_id → name, email, phone
- email → name

CastMember

- person_id → wage, bio

Venue

- venue_id → name, address, seating_capacity, facilities
- address → name, seating_capacity, facilities

Manager

- person_id → manager_since

Audience

- person_id → membership_level

Director

- person_id → experience_years, awards

Finance

- expense_id → category, amount

Performance

- performance_id → date_and_time, number_of_tickets

Commentary

- comment_id → rating, comment, date

Ticket

- ticket_id → seat_number, price, purchase_date, performance_id

NORMALIZATION

We should ensure that the database is in Boyce-Codd Normal Form, and in Third Normal Form as a result. While structuring the database, I have tried my best to make it efficient and manageable. There should not be data redundancy and database must be consistent. Therefore we need to normalize our databases.

To determine if the database is in BCNF, we need to ensure that for every functional dependency $\alpha \rightarrow \beta$, $\alpha \rightarrow \beta$ is trivial or α is a superkey for R. This means that α must uniquely identify every attributes in the table.

Before going any further, it is good idea to determine primary keys of relations. While determining the primary keys, the following rules are considered:

- **Many-to-Many relationships:** The preceding union of the primary keys is a minimal superkey and is chosen as the primary key.
- **One-to-Many relationships:** The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- **Many-to-one relationships:** The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- **One-to-one relationships:** The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.

According to these rules, following table lists the primary keys for the relations:

RELATION	PRIMARY KEY
ProductionDirector	production_id
ProductionCast	production_id U person_id
ManagerVenue	venue_id OR person_id
AudienceComment	comment_id
AudienceTicket	ticket_id
ProductionVenue	production_id U venue_id
PerformanceProduction	performance_id
PerformanceVenue	performance_id
ProductionComment	comment_id

In the tables other than Production, Person, and Venue we have candidate keys as α and they uniquely determine all other attributes but in these three tables we have additional functional dependencies. Let's see if dependencies of example two of them (Production and Person) cause any normalization form violations and how we can normalize the tables if necessary.

Production Table

Current attributes: production_id, title, genre, synopsis, start_date, end_date, budget

Functional dependencies:

- production_id \rightarrow title, genre, synopsis, start_date, end_date, budget
- title \rightarrow genre, synopsis (Since genre and synopsis is unique for the title of production.)

The table is not in BCNF because the non-trivial functional dependency "title \rightarrow genre, synopsis" holds, but "title" attribute is not a superkey.

Normalization:

- To normalize, we should decompose the table into two tables:
 - Production table (has the original primary key) -----> $(R - (\beta - \alpha))$
 - Attributes: production_id, title, start_date, end_date, budget
 - ProductionDetails table -----> $\alpha \cup \beta$
 - Attributes: title, genre, synopsis

Person Table

Current attributes: person_id, name, email, phone

Functional dependencies:

- person_id \rightarrow name, email, phone
- email \rightarrow name (Since email uniquely identifies person's name.)

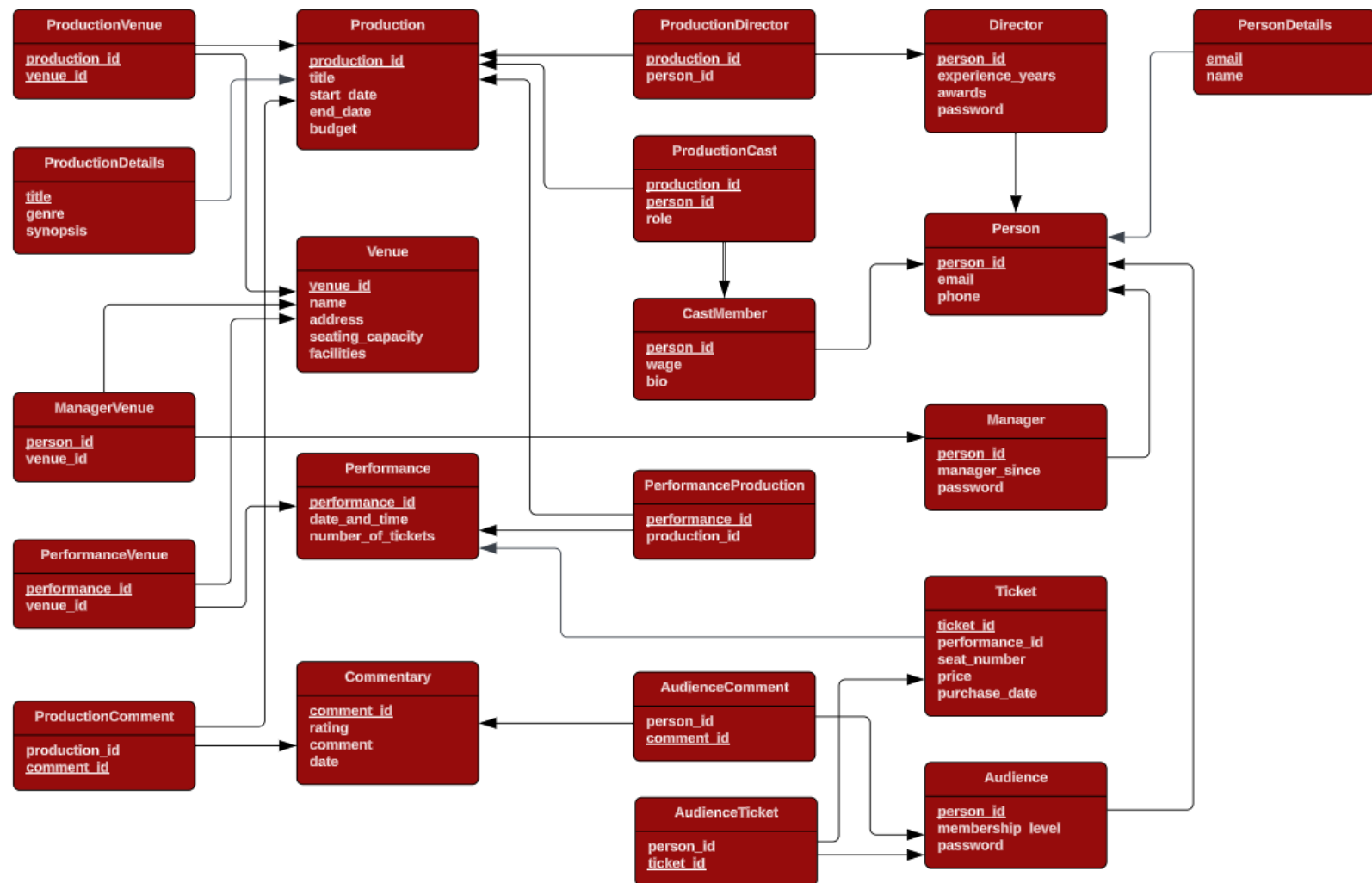
The table is not in BCNF because the non-trivial functional dependency "email \rightarrow name" holds, but "email" is not a superkey.

Normalization:

- To normalize, we should decompose the table into two tables:
 - Person table (has the original primary key) -----> $(R - (\beta - \alpha))$
 - Attributes: person_id, email, phone
 - PersonDetails table -----> $\alpha \cup \beta$
 - Attributes: email, name

DATABASE SCHEMA

In this section, schema of the database, detailing the tables, relationships, and keys are going to be presented with the newly added 2 tables at normalization section. The schema can be seen below:



DATABASE IMPLEMENTATION

All the tables that are shown at the schema are created at the PostgreSQL. I have also used referential integrity constraint, a concept that ensures the consistency and accuracy of relationships between tables. Also, the check constraint is used to specify that the values a column can take must meet a certain condition.

Creation of tables are shown below:

```
CREATE TABLE Production (
    production_id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    start_date DATE,
    end_date DATE,
    budget NUMERIC
);

CREATE TABLE ProductionDetails (
    title VARCHAR(255) PRIMARY KEY,
    genre VARCHAR(50),
    synopsis TEXT
);

CREATE TABLE Person (
    person_id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    phone VARCHAR(20)
);

CREATE TABLE PersonDetails (
    email VARCHAR(255) PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE CastMember (
    person_id INT PRIMARY KEY REFERENCES Person(person_id),
    wage NUMERIC,
    bio TEXT
);

CREATE TABLE Venue (
    venue_id SERIAL PRIMARY KEY,
    name VARCHAR(255),
    address VARCHAR(255) UNIQUE,
    seating_capacity INT,
    facilities TEXT
);
```

```

CREATE TABLE Manager (
    person_id INT PRIMARY KEY REFERENCES Person(person_id),
    manager_since DATE,
    password VARCHAR(255)
);

CREATE TABLE Audience (
    person_id INT PRIMARY KEY REFERENCES Person(person_id),
    membership_level VARCHAR(50),
    password VARCHAR(255)
);

CREATE TABLE Director (
    person_id INT PRIMARY KEY REFERENCES Person(person_id),
    experience_years INT,
    awards TEXT,
    password VARCHAR(255)
);

CREATE TABLE Performance (
    performance_id SERIAL PRIMARY KEY,
    date_and_time TIMESTAMP,
    number_of_tickets INT
);

CREATE TABLE Commentary (
    comment_id SERIAL PRIMARY KEY,
    rating INT CHECK (rating >= 1 AND rating <= 5),
    comment TEXT,
    date DATE
);

CREATE TABLE Ticket {
    ticket_id SERIAL PRIMARY KEY,
    seat_number VARCHAR(10),
    price NUMERIC,
    purchase_date DATE,
    performance_id INT
};

```

```

CREATE TABLE ProductionVenue (
    production_id INT REFERENCES Productions(production_id) ON DELETE CASCADE,
    venue_id INT REFERENCES Venue(venue_id) ON DELETE CASCADE,
    PRIMARY KEY (production_id, venue_id)
);

CREATE TABLE ProductionCast (
    production_id INT REFERENCES Productions(production_id) ON DELETE CASCADE,
    person_id INT REFERENCES CastMember (person_id) ON DELETE CASCADE,
    role VARCHAR(50),
    PRIMARY KEY (production_id, person_id)
);

CREATE TABLE ProductionDirector (
    production_id INT PRIMARY KEY REFERENCES Productions(production_id) ON DELETE CASCADE,
    person_id INT REFERENCES Director(person_id) ON DELETE CASCADE
);

CREATE TABLE PerformanceVenue (
    performance_id INT PRIMARY KEY REFERENCES Performance(performance_id) ON DELETE CASCADE,
    venue_id INT REFERENCES Venue(venue_id) ON DELETE CASCADE
);

CREATE TABLE PerformanceProduction (
    performance_id INT PRIMARY KEY REFERENCES Performance(performance_id) ON DELETE CASCADE,
    production_id INT REFERENCES Productions(production_id) ON DELETE CASCADE
);

CREATE TABLE ManagerVenue (
    person_id INT PRIMARY KEY REFERENCES Manager(person_id) ON DELETE CASCADE,
    venue_id INT REFERENCES Venue(venue_id) ON DELETE CASCADE
);

CREATE TABLE ProductionComment (
    production_id INT REFERENCES Production(production_id) ON DELETE CASCADE,
    comment_id INT PRIMARY KEY references Commentary(comment_id) ON DELETE CASCADE
);

CREATE TABLE AudienceComment (
    person_id INT REFERENCES Audience(person_id) ON DELETE CASCADE,
    comment_id INT PRIMARY KEY references Commentary(comment_id) ON DELETE CASCADE
);

CREATE TABLE AudienceTicket (
    person_id INT REFERENCES Audience(person_id) ON DELETE CASCADE,
    ticket_id INT PRIMARY KEY REFERENCES Ticket(ticket_id) ON DELETE CASCADE
);

```

```

CREATE TABLE ProductionLog (
    log_id SERIAL PRIMARY KEY,
    production_id INT,
    director_id INT,
    title VARCHAR(255),
    genre VARCHAR(100),
    synopsis TEXT,
    start_date DATE,
    end_date DATE,
    budget DECIMAL(10, 2),
    log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

The database tables can be seen below including relation tables:

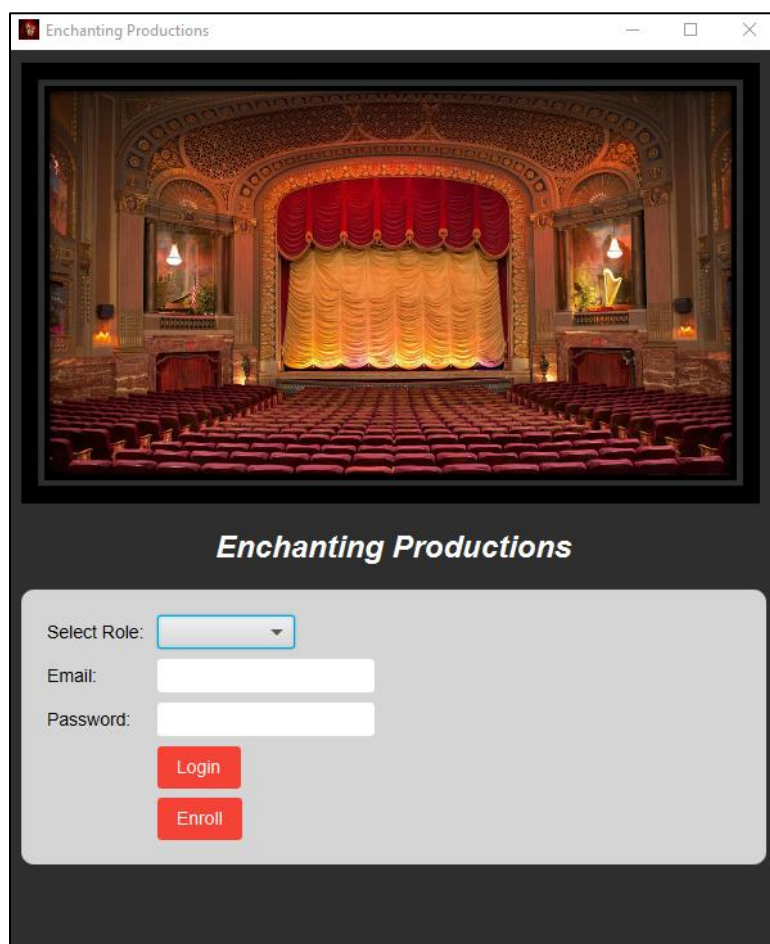
Tables (22)
> audience
> audiencecomment
> audienceticket
> castmember
> commentary
> director
> manager
> managervenue
> performance
> performanceproduction
> performancevenue
> person
> persondetails
> production
> productioncast
> productioncomment
> productiondetails
> productiondirector
> productionlog
> productionvenue
> ticket
> venue

USER INTERFACE

User interface has been developed to interact with the database and fulfill the requirements outlined in the project. The modules are going to be explained one by one under different headers below.

Main Screen

This is the main screen of the interface when it is first executed:

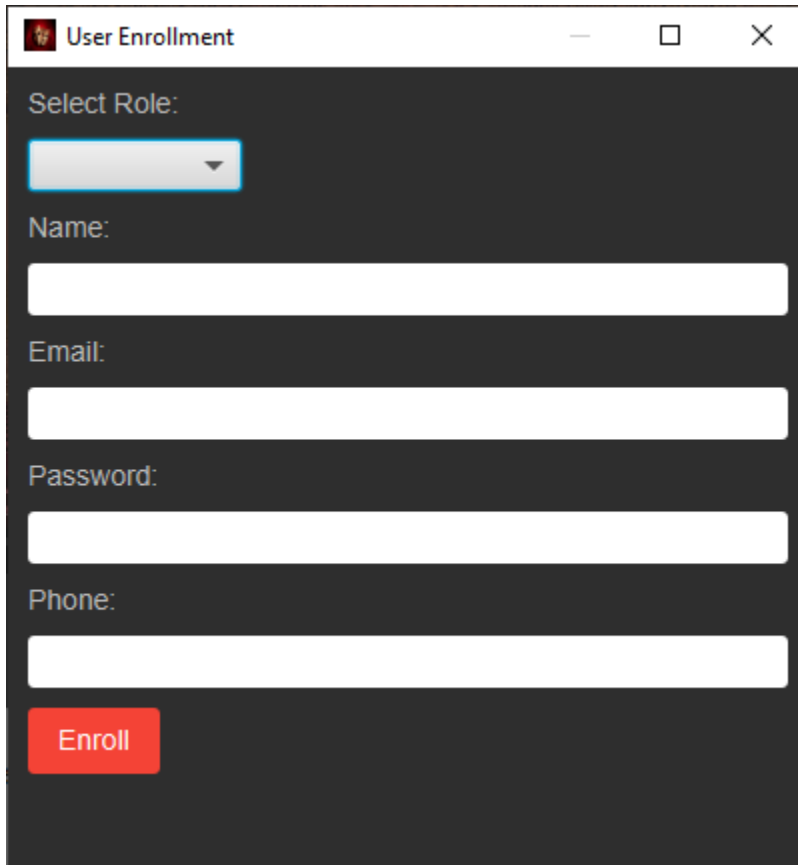


Main Screen of the Interface

Here user can select a role first between 3 roles: audience, director, manager. Then login to the proper dashboard according to role selection. If user is not enrolled before, he/she can enroll to the system which is another page.

Enrollment

With this module, user can enroll to the system by selecting a role between director and audience.

A screenshot of a web application window titled "User Enrollment". The window has a dark gray background and a white title bar with standard minimize, maximize, and close buttons. The form contains the following elements: a "Select Role:" label above a dropdown menu; a "Name:" label above a text input field; an "Email:" label above a text input field; a "Password:" label above a text input field; a "Phone:" label above a text input field; and a red "Enroll" button at the bottom left.

User Enrollment

Select Role:

Name:

Email:

Password:

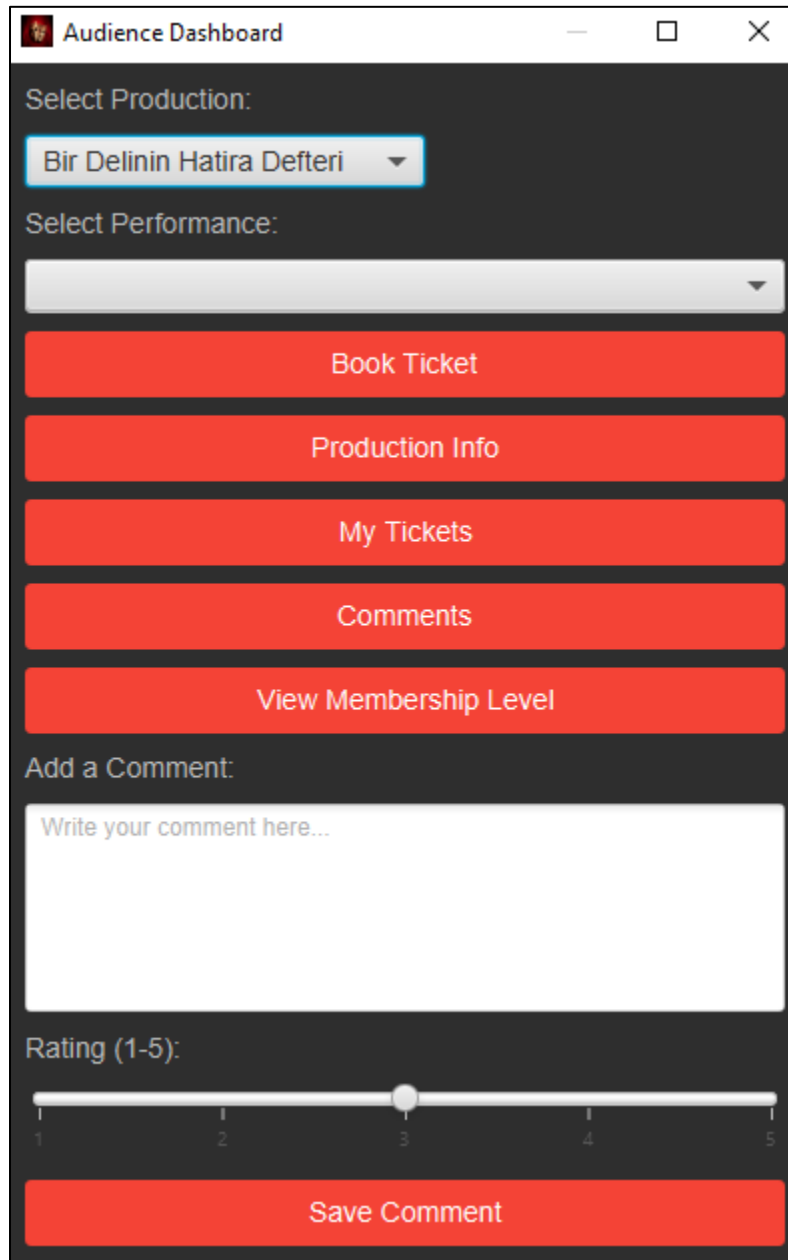
Phone:

Enroll

Enrollment Page

Audience Page

This is the page that is accessed when logged in as an audience:



The screenshot shows a web application window titled "Audience Dashboard". The interface is dark-themed with red buttons. It features two dropdown menus for selecting a production and a performance. Below these are five red buttons: "Book Ticket", "Production Info", "My Tickets", "Comments", and "View Membership Level". There is a text input field for adding a comment, followed by a rating slider (1-5) and a "Save Comment" button.

Audience Dashboard

Select Production:

Bir Delinin Hatira Defteri

Select Performance:

Book Ticket

Production Info

My Tickets

Comments

View Membership Level

Add a Comment:

Write your comment here...

Rating (1-5):

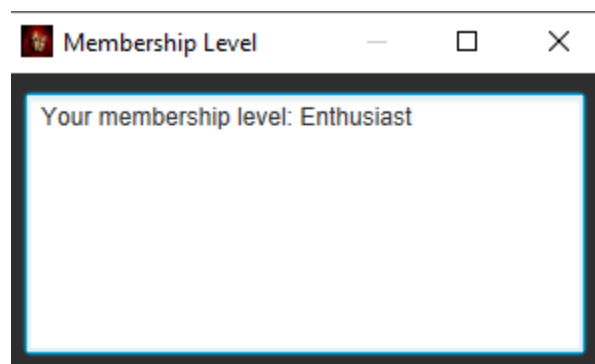
1 2 3 4 5

Save Comment

Audience Page

Audience can do many things at this page. Firstly, he/she can select a production with the top list box item. After selecting the production, the information about the selected production can be seen by clicking the “Production Info” button. User can also select a performance from the second list box after selecting the production. Here, venues are listed with date and time. Then user can book the ticket. The tickets that are booked by the user can be seen by clicking the “My Tickets” button. User can also enter a comment for the productions and give rating. The comments about the selected production can be seen by clicking the “Comments” button.

There are also one more thing. Each audience has a membership level. If audience booked less than 3 tickets, than his/her level is “Basic” , if between 3 and 6 his/her level is “Enthusiast”, if between 6 and 9 his/her level is “Informed”, if between 9 and 12 his/her level is “Connoisseur”, if more than 12 then role is “Professional”. By clicking the “View Membership Level” button, user can see his/her membership level. For example:



Membership Level

Director Page


With this module, directors can add productions to be performed later. Director enters the title, genre, synopsis, budget, start and end dates of the production. Also director needs to add cast members to his/her production. It can be done by clicking the “Add Cast Member” button.

When director clicks that button, the “Add Cast Member” screen is going to be opened. Here director can do 2 things:

- Add cast members that is formerly played in a production.
- OR
- Add new cast member.

After adding at least one cast member, director can insert this production to database. Then managers schedule this production at their venues if they want.

Director Dashboard




Director Dashboard

Insert New Production:


Genre

Synopsis

Start Date



End Date




Budget

Added Cast Members:

Add Cast Member

Insert Production

Director Page

 Add Cast Member

Select Existing Cast Member:

Role

Add Existing Cast Member

Add New Cast Member:

Name

Email

Phone

Wage

Bio

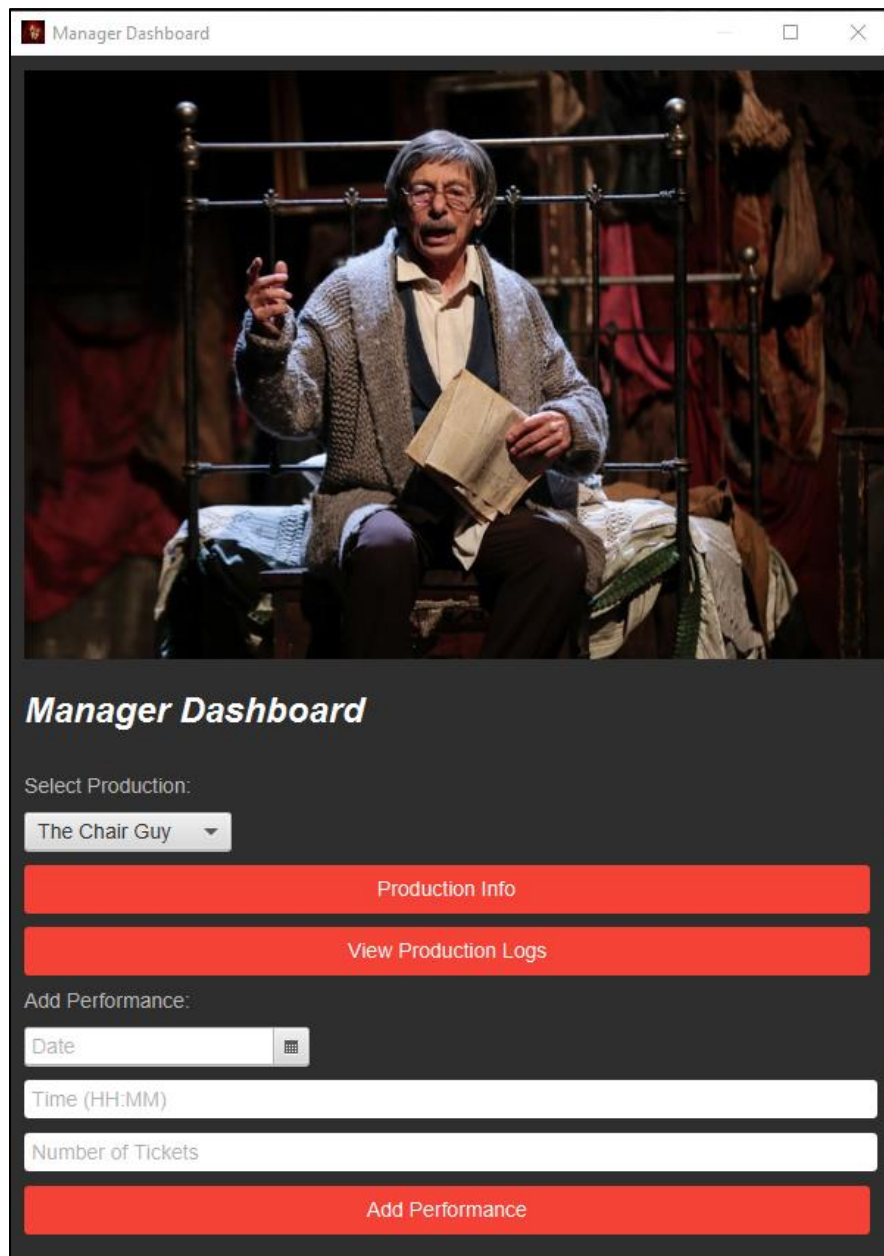
Role

Add New Cast Member

Add Cast Member Page

Manager Page

This module helps managers to select a production, see the production info (by “Production Info” button), view production logs (by “View Production Logs” button), and then add this performance to his/her venue by giving date, time and number of tickets.

The image shows a web application window titled "Manager Dashboard". At the top is a large image of a man sitting on a chair, holding a book. Below the image, the title "Manager Dashboard" is displayed in a bold, italicized font. Underneath the title, there is a "Select Production:" label followed by a dropdown menu currently showing "The Chair Guy". Below the dropdown are two red buttons: "Production Info" and "View Production Logs". Further down, there is an "Add Performance:" label followed by three input fields: "Date" (with a calendar icon), "Time (HH:MM)", and "Number of Tickets". At the bottom is a red button labeled "Add Performance".

Manager Dashboard

Manager Dashboard


Select Production:

The Chair Guy ▼

Production Info

View Production Logs

Add Performance:

Date 

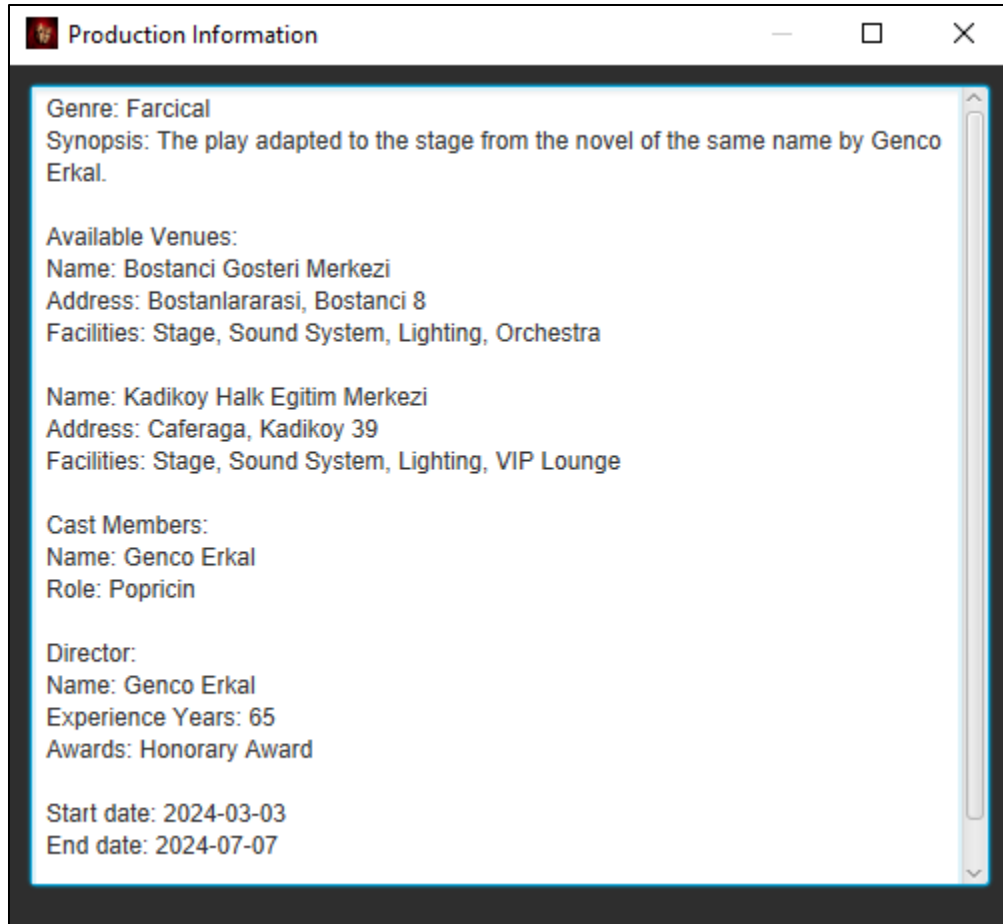
Time (HH:MM)

Number of Tickets

Add Performance

Manager Page

Example production information screen:



The screenshot shows a window titled "Production Information" with a standard Windows-style title bar (minimize, maximize, close buttons). The window contains a text area with the following information:

Genre: Farcical
Synopsis: The play adapted to the stage from the novel of the same name by Genco Erkal.

Available Venues:
Name: Bostanci Gosteri Merkezi
Address: Bostanlararasi, Bostanci 8
Facilities: Stage, Sound System, Lighting, Orchestra

Name: Kadikoy Halk Egitim Merkezi
Address: Caferaga, Kadikoy 39
Facilities: Stage, Sound System, Lighting, VIP Lounge

Cast Members:
Name: Genco Erkal
Role: Popricin

Director:
Name: Genco Erkal
Experience Years: 65
Awards: Honorary Award

Start date: 2024-03-03
End date: 2024-07-07

Information About Production

QUERY DEVELOPMENT

Various SQL queries are executed within the user interface. Examples of these SQL queries are going to be listed below with their explanations:

```
if (role.equalsIgnoreCase( anotherString: "audience")) {  
    sql = "SELECT a.password FROM Audience a " +  
          "JOIN Person p ON a.person_id = p.person_id " +  
          "WHERE p.email = ?";  
} else if (role.equalsIgnoreCase( anotherString: "manager")) {  
    sql = "SELECT m.password FROM Manager m " +  
          "JOIN Person p ON m.person_id = p.person_id " +  
          "WHERE p.email = ?";  
} else if (role.equalsIgnoreCase( anotherString: "director")) {  
    sql = "SELECT d.password FROM Director d " +  
          "JOIN Person p ON d.person_id = p.person_id " +  
          "WHERE p.email = ?";  
} else {  
    showErrorPopup("Invalid role selected.");  
    return;  
}
```

These queries are for assigning proper sql query to login according to selected role.

```
PreparedStatement checkEmailStmt = connection.prepareStatement(  
    sql: "SELECT COUNT(*) FROM Director d JOIN Person p ON d.person_id = p.person_id WHERE p.email = ?");  
checkEmailStmt.setString( parameterIndex: 1, email);  
ResultSet rs = checkEmailStmt.executeQuery();
```

Here I am detecting if any director exists with the same email while we are enrolling as a director.

```
"INSERT INTO Person (email, phone) VALUES (?, ?) RETURNING person_id";  
"INSERT INTO PersonDetails (email, name) VALUES (?, ?)";  
"INSERT INTO Director (person_id, password) VALUES (?, ?)";
```

These are for inserting into proper tables while we are enrolling as a director.

Same two process goes similar to these when we are enrolling as an audience.


```

"INSERT INTO Production (title, start_date, end_date, budget) VALUES (?, ?, ?, ?) RETURNING production_id";
"INSERT INTO ProductionDetails (title, genre, synopsis) VALUES (?, ?, ?)";
"SELECT d.person_id FROM Director d JOIN Person p ON d.person_id = p.person_id WHERE p.email = ?";
"INSERT INTO ProductionDirector (production_id, person_id) VALUES (?, ?)";
"SELECT person_id FROM Person WHERE email = ?";
"INSERT INTO ProductionCast (production_id, person_id, role) VALUES (?, ?, ?)"

```

These are the list of SQL queries that is used for inserting new production.

```

"SELECT COUNT(*) FROM Production WHERE title = ?";

```

This is the query that I have used to check if title is duplicated.

```

"RESET ROLE"

```

I am using this query to reset the role when we go back to main screen.

```

"SELECT mv.venue_id FROM ManagerVenue mv JOIN Person p ON mv.person_id = p.person_id WHERE p.email = ?";

```

Here I am using this query to get the venue id with the manager's email.

```

"SELECT DISTINCT pr.title FROM Production pr LEFT JOIN ProductionVenue pv ON pr.production_id = pv.production_id AND pv.venue_id = ? WHERE pv.venue_id IS NULL";

```

Here I have used different join type. With this query, I am getting the production that is not performed at logged in manager's venue.

```

"SELECT EXISTS (SELECT 1 FROM Performance WHERE performance_id = ?)";

```

This is the query I have used to check if performance with given id exists at the table performance.

I am skipping other insertions since the examples are shown with other tables.

```
"SELECT date, rating, comment, commenter_name, membership_level " +
"FROM CommentaryDetails " +
"WHERE production_title = ?");
```

SQL query is splitted into different lines for demonstration. Here I have used a view. Its details are going to be shown in later sections. Here I am simply getting the comments of a production by giving its title.

Other View examples are below:

```
"SELECT ticket_id, seat_number, price, purchase_date, date_and_time, venue_name, production_title " +
"FROM TicketDetails " +
"WHERE email = ?");
```

```
"SELECT performance_id, venue_name, date_and_time " +
"FROM PerformanceDetails " +
"WHERE title = ?");
```

```
"SELECT v.name, v.address, v.facilities " +
"FROM ProductionVenue pv " +
"JOIN Production pr ON pv.production_id = pr.production_id " +
"JOIN Venue v ON pv.venue_id = v.venue_id " +
"WHERE pr.title = ?");
```

This query uses 2 consecutive SQL queries to get the venue name, address and facilities with giving production title.

```
sql: "SET ROLE " + role);
```

This query is used when switching to another role.

These are the example SQL queries that are used in the project. There are many more SQL queries within the code for further analysis.

TRIGGERS

There are 5 different triggers within the database, each serving a distinct purpose. They are going to be listed below one by one.

First Trigger

```
CREATE OR REPLACE FUNCTION update_seat_number_and_tickets()
RETURNS TRIGGER AS $$
BEGIN
    NEW.seat_number := (SELECT number_of_tickets FROM Performance WHERE performance_id = NEW.performance_id);

    UPDATE Performance
    SET number_of_tickets = number_of_tickets - 1
    WHERE performance_id = NEW.performance_id;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_insert_ticket
BEFORE INSERT ON Ticket
FOR EACH ROW
EXECUTE FUNCTION update_seat_number_and_tickets();
```

This trigger executes before the new ticket is inserted and it decreases the number of tickets of the performance by one and set the seat number value to the number_of_tickets before the decrease.

Second Trigger

```
CREATE OR REPLACE FUNCTION update_membership_level()
RETURNS TRIGGER AS $$
DECLARE
    ticket_count INT;
BEGIN
    SELECT COUNT(*) INTO ticket_count
    FROM Ticket t
    JOIN AudienceTicket at ON t.ticket_id = at.ticket_id
    WHERE at.person_id = NEW.person_id;

    IF ticket_count = 3 THEN
        UPDATE Audience
        SET membership_level = 'Enthusiast'
        WHERE person_id = NEW.person_id;
    ELSIF ticket_count = 6 THEN
        UPDATE Audience
        SET membership_level = 'Informed'
        WHERE person_id = NEW.person_id;
    ELSIF ticket_count = 9 THEN
        UPDATE Audience
        SET membership_level = 'Connoisseur'
        WHERE person_id = NEW.person_id;
    ELSIF ticket_count = 12 THEN
        UPDATE Audience
        SET membership_level = 'Professional'
        WHERE person_id = NEW.person_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_update_membership_level
AFTER INSERT ON AudienceTicket
FOR EACH ROW
EXECUTE FUNCTION update_membership_level();
```

This trigger executes after the new audiencticket relation is inserted. It checks the number of tickets that the audience has and then change the membership level of the user accordingly. Level increases as user attend more performances.

Third Trigger

```
CREATE OR REPLACE FUNCTION handle_zero_tickets()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.number_of_tickets = 0 THEN
        RAISE EXCEPTION 'The tickets are sold out.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_handle_zero_tickets
BEFORE UPDATE ON Performance
FOR EACH ROW
EXECUTE FUNCTION handle_zero_tickets();
```

This trigger executes before the update on performance table and ensures that when number of tickets becomes 0, an exception is raised to indicate that tickets are sold out.

Fourth Trigger

```
CREATE OR REPLACE FUNCTION check_performance_conflict_interval()
RETURNS TRIGGER AS $$
DECLARE
    v_new_start_time TIMESTAMP;
    v_new_end_time TIMESTAMP;
BEGIN
    SELECT date_and_time INTO v_new_start_time FROM Performance WHERE performance_id = NEW.performance_id;
    v_new_start_time := v_new_start_time - INTERVAL '2 hours';
    v_new_end_time := v_new_start_time + INTERVAL '4 hours';

    IF EXISTS (
        SELECT 1
        FROM Performance p
        JOIN PerformanceVenue pv ON p.performance_id = pv.performance_id
        WHERE pv.venue_id = NEW.venue_id
        AND p.date_and_time BETWEEN v_new_start_time AND v_new_end_time
        AND p.performance_id <> NEW.performance_id
    ) THEN
        DELETE FROM Performance WHERE performance_id = NEW.performance_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_check_performance_conflict_interval
AFTER INSERT ON PerformanceVenue
FOR EACH ROW
EXECUTE FUNCTION check_performance_conflict_interval();
```

With the trigger named “trigger_check_performance_conflict_interval” I am controlling is there any other performance at the same venue that 2 hours earlier or 2 hours later than the performance that is going to be added. If so, the performance is deleted from the performance table.

Fifth Trigger

```
CREATE OR REPLACE FUNCTION log_production_creation()
RETURNS TRIGGER AS $$
DECLARE
    v_title VARCHAR(255);
    v_genre VARCHAR(100);
    v_synopsis TEXT;
    v_start_date DATE;
    v_end_date DATE;
    v_budget DECIMAL(10, 2);
BEGIN
    SELECT p.title, p.start_date, p.end_date, p.budget
    INTO v_title, v_start_date, v_end_date, v_budget
    FROM Production p
    WHERE p.production_id = NEW.production_id;
    SELECT pd.genre, pd.synopsis
    INTO v_genre, v_synopsis
    FROM ProductionDetails pd
    WHERE pd.title = v_title;

    INSERT INTO ProductionLog (
        production_id, director_id, title, genre, synopsis, start_date, end_date, budget, log_time
    )
    VALUES (
        NEW.production_id,
        NEW.person_id, -- Director ID
        v_title,
        v_genre,
        v_synopsis,
        v_start_date,
        v_end_date,
        v_budget,
        CURRENT_TIMESTAMP
    );

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_log_production_creation
AFTER INSERT ON ProductionDirector
FOR EACH ROW
EXECUTE FUNCTION log_production_creation();
```

With the trigger_log_production_creation, I am saving the logs about the details into ProductionLog table. This informations are useful for the managers of the venues so when a director inserts new productions, it is good to know about the details of it.

VIEWS

For this part, I have created 5 diverse views within the database. I have mentioned about those views earlier at this report and now I am going to give details about them.

First View

```
CREATE VIEW PerformanceDetails AS
SELECT p.performance_id, v.name AS venue_name, p.date_and_time, pr.title
FROM Performance p
JOIN PerformanceProduction pp ON p.performance_id = pp.performance_id
JOIN Production pr ON pp.production_id = pr.production_id
JOIN PerformanceVenue pv ON p.performance_id = pv.performance_id
JOIN Venue v ON pv.venue_id = v.venue_id;
```

This view uses consecutive joins one after another and looks complex. So I have created a view for this query and use it in my user interface to get the details about the performance.

Second View

```
CREATE VIEW TicketDetails AS
SELECT t.ticket_id, t.seat_number, t.price, t.purchase_date, pe.date_and_time, v.name AS venue_name, pr.title AS production_title, p.email
FROM Ticket t
JOIN AudienceTicket at ON t.ticket_id = at.ticket_id
JOIN Audience a ON at.person_id = a.person_id
JOIN Person p ON a.person_id = p.person_id
JOIN Performance pe ON t.performance_id = pe.performance_id
JOIN PerformanceVenue pv ON pe.performance_id = pv.performance_id
JOIN Venue v ON pv.venue_id = v.venue_id
JOIN PerformanceProduction pp ON pe.performance_id = pp.performance_id
JOIN Production pr ON pp.production_id = pr.production_id;
```

Since there are many relations, it is not easy to get details about tickets with just a single query. To make code look less complex, I have used this view. I have used this view to get details of tickets directly.

Third View

```
CREATE VIEW CommentaryDetails AS
SELECT c.date, c.rating, c.comment, pd.name AS commenter_name, a.membership_level, pr.title AS production_title
FROM Commentary c
JOIN ProductionComment pc ON c.comment_id = pc.comment_id
JOIN Production pr ON pc.production_id = pr.production_id
JOIN AudienceComment ac ON c.comment_id = ac.comment_id
JOIN Audience a ON ac.person_id = a.person_id
JOIN Person p ON a.person_id = p.person_id
JOIN PersonDetails pd ON p.email = pd.email;
```

Another view for getting details about commentary.

Fourth View

```
CREATE VIEW DirectorDetails AS
SELECT pd.name AS director_name, d.experience_years, d.awards, pr.start_date, pr.end_date, pr.title AS production_title
FROM ProductionDirector pdirector
JOIN Production pr ON pdirector.production_id = pr.production_id
JOIN Director d ON pdirector.person_id = d.person_id
JOIN Person p ON d.person_id = p.person_id
JOIN PersonDetails pd ON p.email = pd.email;
```

This view helps me to get details of a director with more simple query by using the DirectorDetails view.

Fifth View

```
CREATE VIEW CastDetails AS
SELECT pd.name AS cast_name, pc.role, pr.title AS production_title
FROM ProductionCast pc
JOIN Production pr ON pc.production_id = pr.production_id
JOIN Person p ON pc.person_id = p.person_id
JOIN PersonDetails pd ON p.email = pd.email;
```

Another view with different task. This view helps to get details about cast members.

TRANSACTIONS

In the project, transactions are designed and executed within the code ensuring data integrity and consistency. The transactions that I have used are listed below:

First Transaction

```
connection.setAutoCommit(false);

PreparedStatement personStmt = connection.prepareStatement(
    sql: "INSERT INTO Person (email, phone) VALUES (?, ?) RETURNING person_id");
personStmt.setString( parameterIndex: 1, email);
personStmt.setString( parameterIndex: 2, x: "");
ResultSet personRs = personStmt.executeQuery();
int personId = 0;
if (personRs.next()) {
    personId = personRs.getInt( columnLabel: "person_id");
}

PreparedStatement detailsStmt = connection.prepareStatement(
    sql: "INSERT INTO PersonDetails (email, name) VALUES (?, ?)");
detailsStmt.setString( parameterIndex: 1, email);
detailsStmt.setString( parameterIndex: 2, name);
detailsStmt.executeUpdate();

String hashedPassword = hashPassword(password);

PreparedStatement directorStmt = connection.prepareStatement(
    sql: "INSERT INTO Director (person_id, password) VALUES (?, ?)");
directorStmt.setInt( parameterIndex: 1, personId);
directorStmt.setString( parameterIndex: 2, hashedPassword);
directorStmt.executeUpdate();

connection.commit();
```

When enrolling to the system as director, we want everything to be done at all or none. So we must insert into person, persondetails, and director tables while enrolling to system. These insertions must all be done correctly or must all fail together. Inserting into 1 or 2 tables are not good so I have used transaction here.

Second Transaction

```
connection.setAutoCommit(false);

PreparedStatement personStmt = connection.prepareStatement(
    sql: "INSERT INTO Person (email, phone) VALUES (?, ?) RETURNING person_id");
personStmt.setString( parameterIndex: 1, email);
personStmt.setString( parameterIndex: 2, phone);
ResultSet personRs = personStmt.executeQuery();
int personId = 0;
if (personRs.next()) {
    personId = personRs.getInt( columnLabel: "person_id");
}

PreparedStatement detailsStmt = connection.prepareStatement(
    sql: "INSERT INTO PersonDetails (email, name) VALUES (?, ?)");
detailsStmt.setString( parameterIndex: 1, email);
detailsStmt.setString( parameterIndex: 2, name);
detailsStmt.executeUpdate();

String hashedPassword = hashPassword(password);

PreparedStatement audienceStmt = connection.prepareStatement(
    sql: "INSERT INTO Audience (person_id, password, membership_level) VALUES (?, ?, ?)");
audienceStmt.setInt( parameterIndex: 1, personId);
audienceStmt.setString( parameterIndex: 2, hashedPassword);
audienceStmt.setString( parameterIndex: 3, x: "Basic");
audienceStmt.executeUpdate();

connection.commit();
```

Another transaction has been used for enrolling as audience with the same reason as the first transaction.

Third Transaction

```
connection.setAutoCommit(false);

PreparedStatement insertProductionStmt = connection.prepareStatement(
    sql: "INSERT INTO Production (title, start_date, end_date, budget) VALUES (?, ?, ?, ?) RETURNING production_id");
insertProductionStmt.setString( parameterIndex: 1, title);
insertProductionStmt.setDate( parameterIndex: 2, java.sql.Date.valueOf(startDate));
insertProductionStmt.setDate( parameterIndex: 3, java.sql.Date.valueOf(endDate));
insertProductionStmt.setBigDecimal( parameterIndex: 4, new BigDecimal(budget));
ResultSet productionRs = insertProductionStmt.executeQuery();
int productionId = 0;
if (productionRs.next()) {
    productionId = productionRs.getInt( columnLabel: "production_id");
}

PreparedStatement insertProductionDetailsStmt = connection.prepareStatement(
    sql: "INSERT INTO ProductionDetails (title, genre, synopsis) VALUES (?, ?, ?)");
insertProductionDetailsStmt.setString( parameterIndex: 1, title);
insertProductionDetailsStmt.setString( parameterIndex: 2, genre);
insertProductionDetailsStmt.setString( parameterIndex: 3, synopsis);
insertProductionDetailsStmt.executeUpdate();

PreparedStatement getDirectorStmt = connection.prepareStatement(
    sql: "SELECT d.person_id FROM Director d JOIN Person p ON d.person_id = p.person_id WHERE p.email = ?");
getDirectorStmt.setString( parameterIndex: 1, email);
ResultSet directorRs = getDirectorStmt.executeQuery();
int directorId = 0;
if (directorRs.next()) {
    directorId = directorRs.getInt( columnLabel: "person_id");
}

PreparedStatement insertProductionDirectorStmt = connection.prepareStatement(
    sql: "INSERT INTO ProductionDirector (production_id, person_id) VALUES (?, ?)");
insertProductionDirectorStmt.setInt( parameterIndex: 1, productionId);
insertProductionDirectorStmt.setInt( parameterIndex: 2, directorId);
insertProductionDirectorStmt.executeUpdate();

for (String castMember : castMembersList) {
    String[] parts = castMember.split( regex: " as ");
    String castEmail = parts[0];
    String role = parts[1];

    PreparedStatement getCastStmt = connection.prepareStatement( sql: "SELECT person_id FROM Person WHERE email = ?");
    getCastStmt.setString( parameterIndex: 1, castEmail);
    ResultSet castRs = getCastStmt.executeQuery();
    int personId = 0;
    if (castRs.next()) {
        personId = castRs.getInt( columnLabel: "person_id");
    }

    PreparedStatement insertProductionCastStmt = connection.prepareStatement( sql: "INSERT INTO ProductionCast (production_id, person_id, role) VALUES (?, ?, ?)");
    insertProductionCastStmt.setInt( parameterIndex: 1, productionId);
    insertProductionCastStmt.setInt( parameterIndex: 2, personId);
    insertProductionCastStmt.setString( parameterIndex: 3, role);
    insertProductionCastStmt.executeUpdate();
}

connection.commit();
```

Since insertions are critical, we must insert into production, productiondetails, and productiondirector tables all at once when we are inserting a new production. Insertions should be completed all, or failed all.

Fourth Transaction

```
connection.setAutoCommit(false);

PreparedStatement insertPerformanceStmt = connection.prepareStatement(
    sql: "INSERT INTO Performance (date_and_time, number_of_tickets) VALUES (?, ?) RETURNING performance_id");
insertPerformanceStmt.setTimestamp( parameterIndex: 1, Timestamp.valueOf(dateTime));
insertPerformanceStmt.setInt( parameterIndex: 2, Integer.parseInt(numTickets));
ResultSet performanceRs = insertPerformanceStmt.executeQuery();
int performanceId = 0;
if (performanceRs.next()) {
    performanceId = performanceRs.getInt( columnLabel: "performance_id");
}

PreparedStatement insertPerformanceVenueStmt = connection.prepareStatement(
    sql: "INSERT INTO PerformanceVenue (performance_id, venue_id) VALUES (?, ?)");
insertPerformanceVenueStmt.setInt( parameterIndex: 1, performanceId);
insertPerformanceVenueStmt.setInt( parameterIndex: 2, venueId);
insertPerformanceVenueStmt.executeUpdate();

PreparedStatement checkPerformanceStmt = connection.prepareStatement(
    sql: "SELECT EXISTS (SELECT 1 FROM Performance WHERE performance_id = ?)");
checkPerformanceStmt.setInt( parameterIndex: 1, performanceId);
ResultSet rs = checkPerformanceStmt.executeQuery();
if (!(rs.next() && rs.getBoolean( columnIndex: 1))) {
    connection.rollback();
    showErrorPopup("There is already a performance at that time.");
    return;
}

PreparedStatement insertProductionVenueStmt = connection.prepareStatement(
    sql: "INSERT INTO ProductionVenue (production_id, venue_id) VALUES (?, ?)");
insertProductionVenueStmt.setInt( parameterIndex: 1, productionId);
insertProductionVenueStmt.setInt( parameterIndex: 2, venueId);
insertProductionVenueStmt.executeUpdate();

PreparedStatement insertProductionPerformanceStmt = connection.prepareStatement(
    sql: "INSERT INTO PerformanceProduction (performance_id, production_id) VALUES (?, ?)");
insertProductionPerformanceStmt.setInt( parameterIndex: 1, performanceId);
insertProductionPerformanceStmt.setInt( parameterIndex: 2, productionId);
insertProductionPerformanceStmt.executeUpdate();

connection.commit();
```

Here, I am adding a new performance for production. While adding a performance, we must insert into performance, performancevenue, productionvenue, and performanceproduction tables so these insertions must be all completed or all failed.

Fifth Transaction

```
connection.setAutoCommit(false);

PreparedStatement personStmt = connection.prepareStatement( sql: "INSERT INTO Person (email, phone) VALUES (?, ?) RETURNING person_id");
personStmt.setString( parameterIndex: 1, email);
personStmt.setString( parameterIndex: 2, phone);
ResultSet personRs = personStmt.executeQuery();
int personId = 0;
if (personRs.next()) {
    personId = personRs.getInt( columnLabel: "person_id");
}

PreparedStatement detailsStmt = connection.prepareStatement( sql: "INSERT INTO PersonDetails (email, name) VALUES (?, ?)");
detailsStmt.setString( parameterIndex: 1, email);
detailsStmt.setString( parameterIndex: 2, name);
detailsStmt.executeUpdate();

PreparedStatement castStmt = connection.prepareStatement( sql: "INSERT INTO CastMember (person_id, wage, bio) VALUES (?, ?, ?)");
castStmt.setInt( parameterIndex: 1, personId);
castStmt.setBigDecimal( parameterIndex: 2, new BigDecimal(wage));
castStmt.setString( parameterIndex: 3, bio);
castStmt.executeUpdate();

connection.commit();
```

Here, I am adding a new cast member to production. While adding a new cast member, as it is done at audience and director enrollments, new insertions should be done also for person and persondetails tables. For these 3 insertions to be done completely or failed completely, I have used a transaction here.

Sixth Transaction

```
connection.setAutoCommit(false);

PreparedStatement commentStmt = connection.prepareStatement(
    sql: "INSERT INTO Commentary (rating, comment, date) VALUES (?, ?, ?) RETURNING comment_id");
commentStmt.setInt( parameterIndex: 1, rating);
commentStmt.setString( parameterIndex: 2, commentText);
commentStmt.setDate( parameterIndex: 3, new java.sql.Date(System.currentTimeMillis()));
ResultSet commentRs = commentStmt.executeQuery();
int commentId = 0;
if (commentRs.next()) {
    commentId = commentRs.getInt( columnLabel: "comment_id");
}

PreparedStatement audienceStmt = connection.prepareStatement(
    sql: "SELECT a.person_id FROM Audience a JOIN Person p ON a.person_id = p.person_id WHERE p.email = ?");
audienceStmt.setString( parameterIndex: 1, email);
ResultSet audienceRs = audienceStmt.executeQuery();
int personId = 0;
if (audienceRs.next()) {
    personId = audienceRs.getInt( columnLabel: "person_id");
}

PreparedStatement productionStmt = connection.prepareStatement(
    sql: "SELECT production_id FROM production WHERE title = ?");
productionStmt.setString( parameterIndex: 1, productionTitle);
ResultSet productionRs = productionStmt.executeQuery();
int productionId = 0;
if (productionRs.next()) {
    productionId = productionRs.getInt( columnLabel: "production_id");
}

PreparedStatement productionCommentStmt = connection.prepareStatement(
    sql: "INSERT INTO ProductionComment (production_id, comment_id) VALUES (?, ?)");
productionCommentStmt.setInt( parameterIndex: 1, productionId);
productionCommentStmt.setInt( parameterIndex: 2, commentId);
productionCommentStmt.executeUpdate();

PreparedStatement audienceCommentStmt = connection.prepareStatement(
    sql: "INSERT INTO AudienceComment (person_id, comment_id) VALUES (?, ?)");
audienceCommentStmt.setInt( parameterIndex: 1, personId);
audienceCommentStmt.setInt( parameterIndex: 2, commentId);
audienceCommentStmt.executeUpdate();

connection.commit();
```

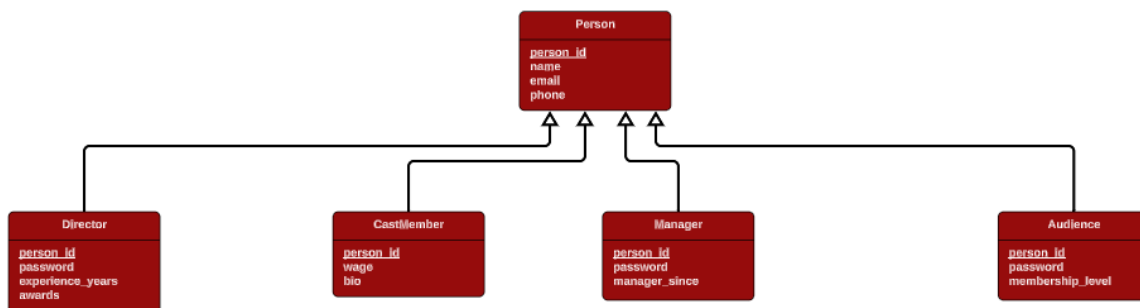
Here I am saving the comment that user has entered. While inserting comment into commentary table, we should also do insertions into productioncomment and audiencecomment to indicate corresponding audience and production for the comment. For doing these transactions all at once, I am using transaction here.

Concurrency Control

As I have mentioned for the transactions, while inserting a new audience, comment, etc., there are also other insertions that need to be done for relation tables. For this reason, I have used transactions to properly handle simultaneous database operations. If I did not use any transaction, there might be concurrency issues. For example, assume I have not used the transaction while I am inserting a new comment. Comment may be inserted into the commentary table, which keeps the comments, but not into the audiencecomment table. In that case, there is a comment but it is not associated with any audience. This is not the case that I want because this creates concurrency issues. Therefore, I have used transactions for the operations that I wanted all of the queries success or fail together.

INHERITANCE

I have utilized inheritance concept to establish relationships between tables, enhancing data organization and management. I have created Person as high-level schema. CastMember, Audience, Director, and Manager inherit from the Person because they have common properties like email, name, and phone. This inheritance is overlapping because a director may be a cast member at the same time for different performances or even for same performances. Also an audience can also be a cast member, etc. Therefore the inheritance I have used here is overlapping.



PRIVILEGES AND ROLES

I have granted privileges and created roles within the database, assigning appropriate access levels to different users. There are 3 different roles in my database. I have created them as follows:

```
CREATE ROLE audience;  
CREATE ROLE manager;  
CREATE ROLE director;
```

Then the proper privileges are granted to them for making them do their tasks:

```
GRANT SELECT, INSERT ON AudienceTicket, Ticket, Commentary, ProductionComment, AudienceComment TO audience;  
GRANT SELECT ON CastDetails, DirectorDetails, CommentaryDetails, TicketDetails, PerformanceDetails, ProductionDetails,  
Director, PersonDetails, ProductionCast, ProductionDirector, ProductionVenue, Person, Audience, Venue, PerformanceProduction,  
PerformanceVenue, Production, Performance TO audience;  
GRANT UPDATE ON Performance, Audience TO audience;  
GRANT USAGE, SELECT ON SEQUENCE ticket_ticket_id_seq, commentary_comment_id_seq TO audience;  
  
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO manager;  
GRANT USAGE, SELECT ON SEQUENCE performance_performance_id_seq TO manager;  
  
GRANT SELECT, INSERT ON ProductionLog, Production, ProductionCast, ProductionDetails, CastMember, Person, Director, ProductionDirector, PersonDetails TO director;  
GRANT USAGE, SELECT ON SEQUENCE production_production_id_seq, person_person_id_seq, productionlog_log_id_seq TO director;
```

Users choose their role while logging in to system and then proper role is set for them. For example, if a user logs in as audience, then “SET ROLE audience” query is executed and user’s privileges are determined. If user exits from his/her dashboard, then the “RESET ROLE” query is executed. So that role is reset.

SECURITY

SQL Injection

In the project, while executing the SQL queries using the inputs that users send, I have not used any string concatenation because this can cause SQL injection. User can enter his/her own SQL query to run on my database and this is not good in terms of security. Therefore I have used prepared statements everywhere that I have executed an SQL query. With this way, user’s inputs are not executed and directly put into my SQL query as an input.

Keeping Passwords

Keeping passwords as plaintext in the database is not a secure approach. If there is any database leak later, then attackers can get users' passwords easily. For this reason, I have used SHA256 hashing algorithm to keep passwords hashed in the database. When user enters his/her password to login, I take the hash of the input and then compare it with the hash that is in the database. Users' passwords look like this at the tables:

password character varying (255)
19d75c03b1e93f857a20e1ae7d687f51ecfbc139b6f2c8ffc7e78997c19009eb
39f8eddc6859281639462ea79dfd9e91c9f536437b98a6499149fd405206f009
9181162c51ddef9fa59cc7b99396f7f859f30cae1c21ffb7636260866e600afa

With this way users' passwords are kept in the database in a secure way.