

COP4710 Summer 2024 – Final Report - Group 17

Authors:

Lianne Lamorena Beltran

Ernesto Ferrer Gomez

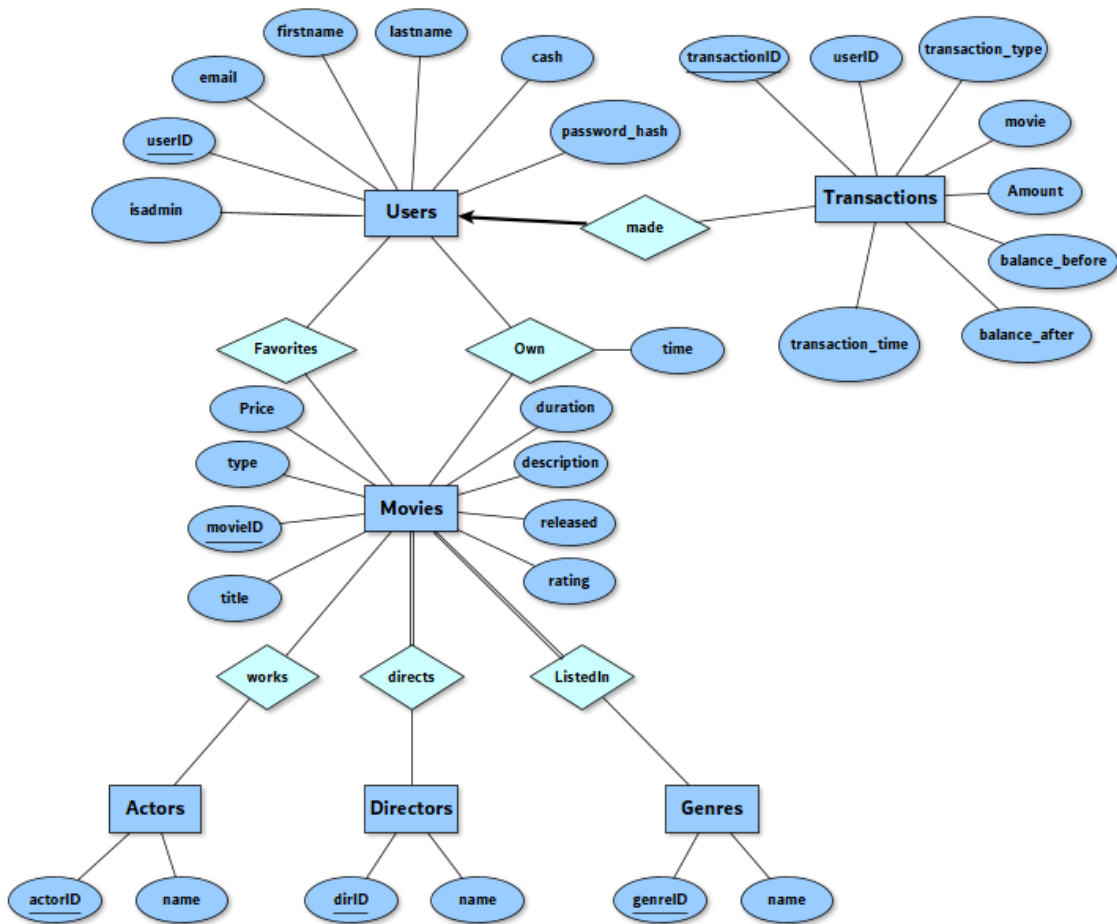
Mario A Landa Pulido

Introduction

As we have mentioned in the previous reports, our project develops an online movie platform site. Regular users can register to the application and login. They should be able to search for movies, add movies to their Favorites, buy movies, check the balance on their accounts, and add credit to their balance. On the other hand, the admin users can add and remove movies from the application and update different information about the movies. The frontend of our web application runs on HTML, CSS, and JavaScript while using Bootstrap to make the interface uniform. The backend of our application is developed in Flask, a Python framework based on route management. For the database side, we use the relational DBMS PostgreSQL. To allow interaction with the database server and query execution from our web application we use the python connector psycopg2 establishing communication between the application and the database. We also use JQuery and AJAX to process some routing and HTTP requests, enhancing the user experience in our Web Application.

Tables

The final EER shown in the picture below is pretty similar to the one presented in milestone 1. The only change is that we replaced the relationship “**watch**” with “**own**” since the first one was not feasible to develop.



PICTURE 1. FINAL EER

There is a total of 11 tables in the database: 6 tables for the entities (**Users**, **Transactions**, **Movies**, **Actors**, **Directors**, and **Genres**) and 5 for the relationships (**works**, **directs**, **listedin**, **favorites** and **owns**). The Schema and attribute domains is included in the Appendix 1

The **Users** table contains the users that register/sign up to the application. By default, users are set up with the attribute **isadmin** set to false and **cash** set to \$100 which is a cash bonus for them to use to buy movies. A hashed version of the password is stored in the database for security purposes. The admin account (userid = 1) was created along with the database and has different accesses than the account for a regular user.

userid [PK] integer	firstname character vary	lastname character var	email character varying (100)	password_hash character varying (255)	cash numeric (9,2)	isadmin boolean
1	Movie	Admin	movie_admin@filmfusion.com	pbkdf2:sha256:600000\$9p...	0.00	true
2	Test	Test	test@test.com	pbkdf2:sha256:600000\$Cm...	105.00	false
3	Tom	Gmail	tom@gmail.com	pbkdf2:sha256:600000\$9W...	100.00	false

PICTURE 2. USERS TABLE

The **Movies** table contains 8806 movies with the attributes shown below.

	movieid [PK] integer	price numeric (5,2)	title character varying (255)	type character var	duration character var	released smallint	rating character var	score numeric (3,1)	description text
1	1	5.00	Dick Johnson Is Dead	Movie	90 min	2020	PG-13	7.3	As her father nears ...
2	2	10.00	Blood & Water	TV Show	2 Seasons	2021	TV-MA	1.1	After crossing path...
3	3	10.00	Ganglands	TV Show	1 Season	2021	TV-MA	7.5	To protect his famil...
4	4	10.00	Jailbirds New Orleans	TV Show	1 Season	2021	TV-MA	8.2	Feuds, flirtations a...
5	5	10.00	Kota Factory	TV Show	2 Seasons	2021	TV-MA	5.0	In a city of coachin...
Total rows: 8806 of 8806		Query complete 00:00:00.445		Rows selected: 8806					

PICTURE 3. MOVIES TABLE

The tables **Directors**, **Actors**, **Genres** store their respective id (SERIAL attributes) and their names.

	dirid [PK] integer	name character varying (100)
1	1	Kirsten Johnson
2	2	Julien Leclercq
3	3	Mike Flanagan
4	4	Robert Cullen
5	5	José Luis Ucha
Total rows: 4997 of 4997		Query complete 00:00:00

PICTURE 4. DIRECTORS TABLE

	actorid [PK] integer	name character varying (100)
1	1	Ama Qamata
2	2	Khosi Ngema
3	3	Gail Mabalane
4	4	Thabang Molaba
5	5	Dillon Windvogel
6	6	Natasha Thahane
7	7	Arno Greeff

PICTURE 5. ACTORS TABLE

	genreid [PK] integer	name character varying (100)
1	1	Documentaries
2	2	International TV Shows
3	3	TV Dramas
4	4	TV Mysteries
5	5	Crime TV Shows
Total rows: 51 of 51		Query complete 00:00:

PICTURE 6. GENRES TABLE

The **Transactions** table contains logs of all the operations made by the users that involve a balance change like movie purchases or cash addition.

	transactionid [PK] integer	userid integer	transactiontype character varying (50)	movie character varying (255)	amount numeric (9,2)	balancebefore numeric (9,2)	balanceafter numeric (9,2)	transaction_time timestamp without tim
1	1	2	Purchase	Parker	5.00	100.00	95.00	2024-07-14 10:44:25
2	2	2	Purchase	An Unfinished Life	5.00	95.00	90.00	2024-07-14 10:44:32
3	3	2	Purchase	Rain Man	5.00	90.00	85.00	2024-07-14 10:44:56
4	4	2	Add cash	[null]	20.00	85.00	105.00	2024-07-14 10:45:51
5	5	10	Bonus cash	[null]	100.00	0.00	100.00	2024-07-14 11:43:22
Total rows: 10 of 10		Query complete 00:00:01.050		Rows selected: 10				

PICTURE 7. GENRES TABLE

On the user's side, we show the two tables corresponding to relationships with movies: **Favorites** and **Owens**. Users should be able to add/remove movies from their favorites and also buy movies.

	userid [PK] integer	movieid [PK] integer	time timestamp without time zone
1	2	130	2024-07-14 10:44:00
2	2	1255	2024-07-14 10:45:34
3	12	8149	2024-07-18 22:26:59
4	12	4512	2024-07-18 22:27:01
5	12	6266	2024-07-18 22:27:04
Total rows: 5 of 5			Query complete 00:00:00.208
			Rows select

PICTURE 8. FAVORITES

	userid [PK] integer	movieid [PK] integer	time timestamp without time zone
1	2	1240	2024-07-14 10:44:25
2	2	130	2024-07-14 10:44:32
3	2	1255	2024-07-14 10:44:56
4	11	1240	2024-07-14 11:45:30
5	11	7501	2024-07-14 11:45:32
6	11	7304	2024-07-14 11:45:35
Total rows: 6 of 6			Query complete 00:00:00.227
			Rows select

PICTURE 9. FAVORITES

The last 3 tables correspond to the relationships between Movies with actors, directors and genres. The joins of table Movies with actors and directors will be used for the search feature implementation that will be described later in the report.

	movieid [PK] integer	dirid [PK] integer
1	1	1
2	3	2
3	6	3
4	7	4
5	7	5
Total rows: 6977 of 6977		Query c

PICTURE 10. DIRECTS RELATION TABLE

	movieid [PK] integer	genreid [PK] integer
1	1	1
2	2	2
3	2	3
4	2	4
5	3	5
Total rows: 19322 of 19322		Query

PICTURE 11. LISTEDIN RELATION TABLE

	movieid [PK] integer	actorid [PK] integer
1	2	1
2	2	2
3	2	3
4	2	4
5	2	5
Total rows: 64138 of 64138		Quer

PICTURE 12. WORKS RELATION TABLE

Application and Queries

SIGN IN & SIGN UP

The homepage of the website is the login screen. It includes two options: one for **Sign In** for the users that have registered already and want to access their account page and the other option **Sign Up** for the users that do not have an account yet and want to register.

127.0.0.1:5000/login

FiLM FuSHioN

Sign in

FiLM FuSHioN
MOVIE PLATFORM

Email Address

Enter email

Password

Enter password

Sign In

Don't have an account?

Sign Up

PICTURE 13. APPLICATION HOMEPAGE

When a user enters an email address and password to login on this screen, the following query is executed to select the user that with a match on the email address. If there is no match on the email address, an error message is shown. If there is a match on the email address, then the passwords on the query result and the password entered are compared. If the passwords are the same, the user is successfully logged in. If not, an error message is shown.

```
cur.execute('SELECT * FROM users WHERE email = %s', (email,))  
account = cur.fetchone()
```

This query is also used on the **Sign-Up** page shown below, which contains a form for a user to fill in. The email address should be unique. The passwords must match and have between 8-20 characters, with at least an uppercase, a lowercase, a number and a special character. The query is used to confirm if the email already exists in the database.

Sign Up

Email Address

Enter email

First Name

Enter first name

Last Name

Enter last name

Password

Enter password

Password (Confirm)

Confirm password

Sign Up

PICTURE 14. SIGN-UP PAGE

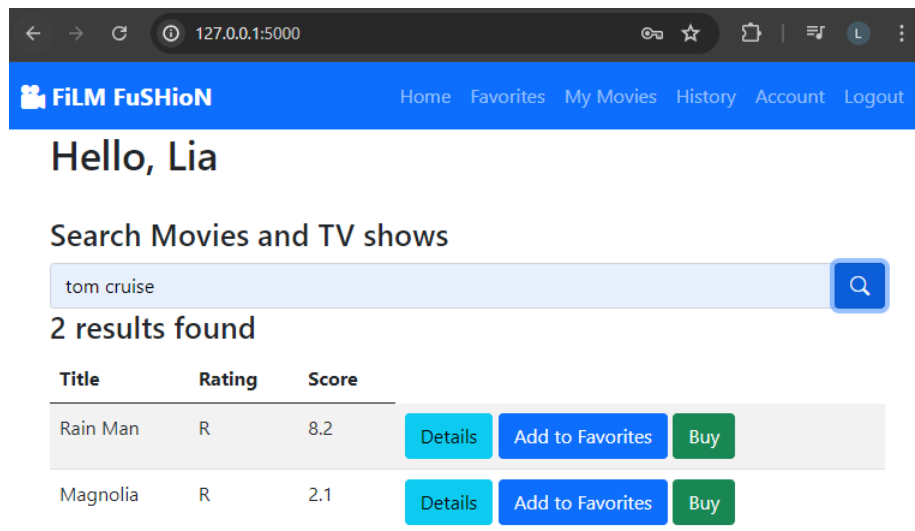
Once all the information is filled in correctly and meeting the requirements, the following queries are executed to add the user to the **Users** table with all the information provided and also to add the cash bonus record to the **Transactions** table.

```
cur.execute('''INSERT INTO users (email, firstname, lastname, password_hash)
VALUES (%s, %s, %s, %s)''', (email, firstName, lastName, hashed_password))

cur.execute("""INSERT INTO Transactions (userID, transactionType,
amount, balanceBefore, balanceAfter) VALUES (%s, %s, %s, %s, %s)""",
(user_id, "Cash Bonus", 100, 0, 100,))
```

SEARCH

The home page for a user logged in is shown in the picture below. On the search bar, the user can type part of the name of a movie, actor or director and the search will return all the results with a match.



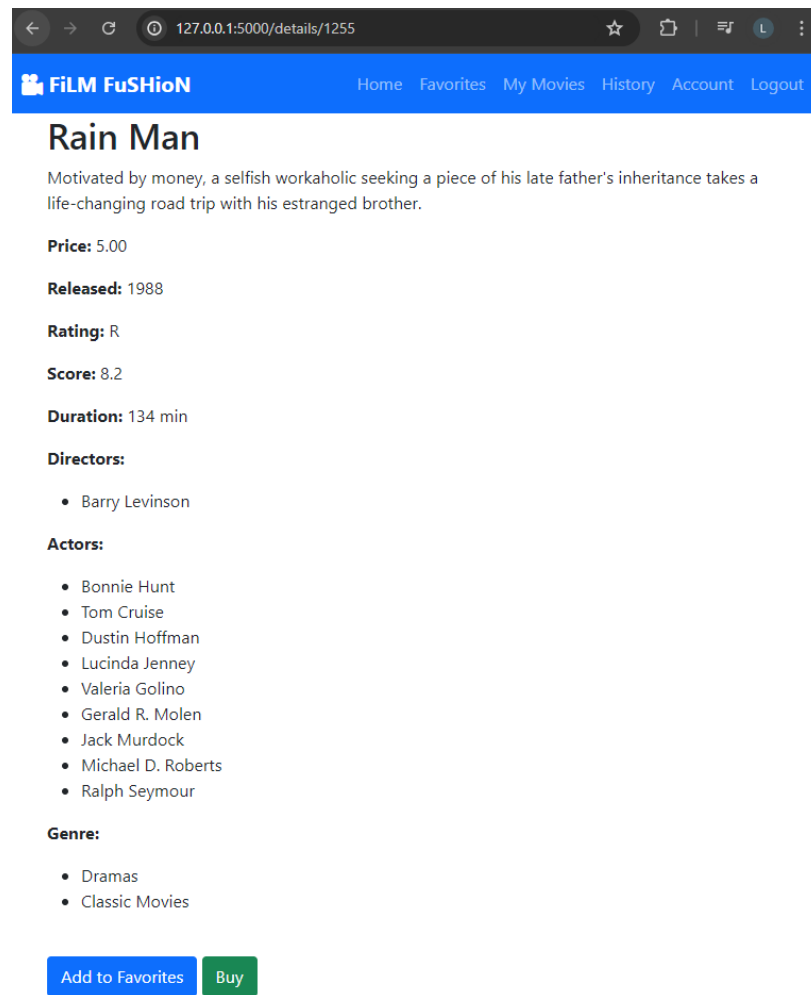
PICTURE 15. SIGN-UP PAGE

The query executed for the search is shown below. It does a selection on the table **Movies** by any title match with the text entered on the search bar, a join (cartesian with selection on **actorID** or **dirID**) on the **Works** and **Actors** tables or **Directs** and **Directors** tables by the name of the actors or directors.

```
cur.execute("""
    SELECT m.* FROM Movies m
    WHERE m.title ILIKE %s
    OR m.movieID IN
    (SELECT movieID FROM Works w, Actors a
    WHERE w.actorID = a.actorID AND a.name ILIKE %s)
    OR m.movieID IN
    (SELECT movieID FROM Directs d, Directors di
    WHERE d.dirID = di.dirID AND di.name ILIKE %s)
    ORDER BY m.score DESC""",
    ('%' + search_word + '%', '%' + search_word + '%', '%' + search_word + '%',))
```

SEE DETAILS

Once a movie is returned as the result of the search, the user can click on Details to see more information about the movie. It will show the following page.



← → ↻ 127.0.0.1:5000/details/1255 ☆ | 📄 🗑️ ⌵ ⋮

FILM FuSHioN Home Favorites My Movies History Account Logout

Rain Man

Motivated by money, a selfish workaholic seeking a piece of his late father's inheritance takes a life-changing road trip with his estranged brother.

Price: 5.00

Released: 1988

Rating: R

Score: 8.2

Duration: 134 min

Directors:

- Barry Levinson

Actors:

- Bonnie Hunt
- Tom Cruise
- Dustin Hoffman
- Lucinda Jenney
- Valeria Golino
- Gerald R. Molen
- Jack Murdock
- Michael D. Roberts
- Ralph Seymour

Genre:

- Dramas
- Classic Movies

[Add to Favorites](#) [Buy](#)

PICTURE 16. MOVIE DETAILS

To show this information, the following queries are executed. They select all the information from the Movies table for the matching **movieID** and select the names of actors, directors and genres of the movie using again a cartesian product with a selection on the respective table's IDs.

```

cur.execute('SELECT * FROM Movies WHERE movieID = %s', (movieID,))
movie = cur.fetchone()

cur.execute("""SELECT directors.name FROM Directors, Directs
              WHERE directors.dirid = directs.dirid
              AND movieID = %s""", (movieID,))
directors = cur.fetchall()

cur.execute("""SELECT actors.name FROM Actors, Works
              WHERE actors.actorid = works.actorid
              AND movieID = %s""", (movieID,))
actors = cur.fetchall()

cur.execute("""SELECT Genres.name FROM Genres, ListedIn
              WHERE Genres.genreID = ListedIn.genreID
              AND movieID = %s""", (movieID,))
genres = cur.fetchall()

```

FAVORITES

From the home page or the details page, the user has the option to add movies to their Favorites list using the Add to Favorites button. The following queries are executed to accomplish this. The first query checks if the user has already added the movie to their Favorites selecting records with the same **userID** and **movieID** in the **Transactions** table. If it does exist, an error message is shown. If not, the tuple with **userID** and **movieID** is added to the table.

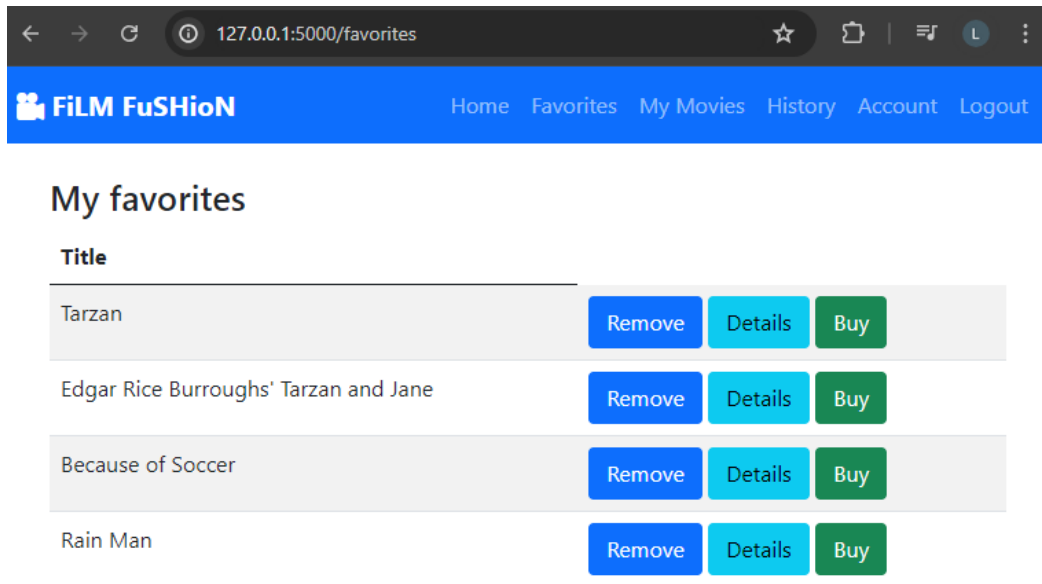
```

cur.execute("SELECT * FROM Favorites WHERE userID = %s AND movieID = %s", (session['userid'], movieId))
if cur.rowcount > 0:
    message = "You have already added this movie to Favorites"
    cur.close()
    conn.close()
    return jsonify({'message': message, 'status': 'error'})

else:
    cur.execute("INSERT INTO favorites (userid, movieid) VALUES (%s, %s)", (userId, movieId))
    conn.commit()
    message = 'Movie added to Favorites!'

```

To see the movies in the favorites list, the user can go to the **Favorites** tab on the navigation bar. It will show the following screen with all the movies already added to this list.



PICTURE 17. FAVORITES

To show this favorite list the following query is executed, which retrieves all the entries in the Favorites that matches the **userID**.

```
cur.execute("""SELECT m.* FROM Movies m, Favorites f WHERE m.movieID = f.movieID
            AND userID = %s ORDER BY f.time""", (session['userid'],))
favorites = cur.fetchall()
```

To remove a movie from this list, the user can click on the Remove button and the following query will be executed. It finds the entry with the requested **movieID** and **userID** on the **Favorites** table and deletes it.

```
cur.execute("""DELETE FROM favorites WHERE movieid IN
            (SELECT movieid FROM movies m WHERE m.title = %s)
            AND userid = %s""", (movieId, userId,))
conn.commit()
```

BUY MOVIES

The user also has the option to buy movies from the home page, the favorites page and the details page. Once the user submits the request to buy a movie, multiple queries are executed.

The first one below checks if the user has already bought the movie and throws an error message if this is the case. It retrieves records that match the **userID** and the **movieID** in the **Owens** table.

```
# checking if the user has already bought that movie
cur.execute("SELECT * FROM Owens WHERE userID = %s AND movieID = %s", (session['userid'], movieID))
```

If the user has not bought the movie yet, then it executes the following query to check that the balance (cash) on the user's account from the table **Users** is enough to buy the movie. If not, it throws an error message.

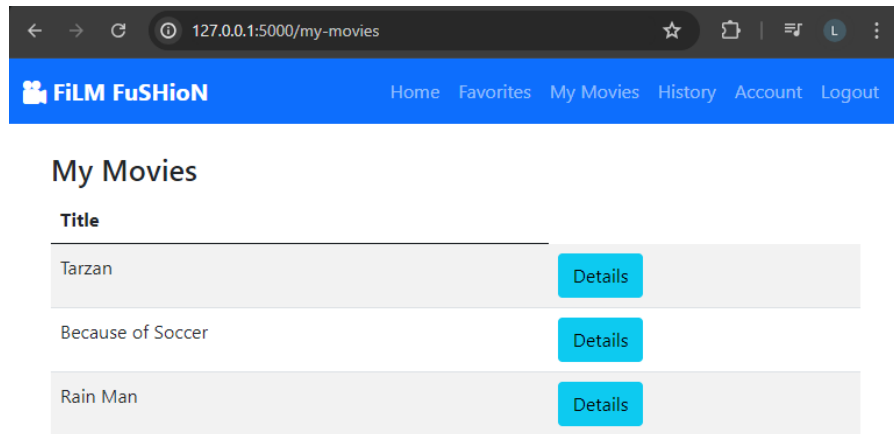
```
cur.execute("SELECT cash FROM Users WHERE userID = %s", (session['userid'],))
cash_result = cur.fetchone()
```

If the balance is enough to buy the movie, then the following queries are executed. The first one updates the cash value on the **Users** table for the user. The second one adds the tuple with **userID** and **movieID** to the **Owens** table (contains movies bought by users). The third one adds this operation to the **Transactions** table with the **transactionType** = "Purchase". In this case, it also adds the title of the movie.

```
cur.execute("UPDATE Users SET cash = %s WHERE userID = %s", (new_cash, session['userid'],))
cur.execute("INSERT INTO Owens (userID, movieID) VALUES (%s, %s)", (session['userid'], movieID,))
cur.execute("""INSERT INTO Transactions
              (userID, transactionType, movie, amount, balanceBefore, balanceAfter)
              VALUES (%s, %s, %s, %s, %s, %s)
              """, (session['userid'], "Purchase", title, price, cash, new_cash,))
conn.commit()
message = "Purchase successfully completed.\n To see your purchased movies go to 'My Movies'"

```

To see the list of the movies already bought, the user can go to **My Movies** tab on the navigation bar.



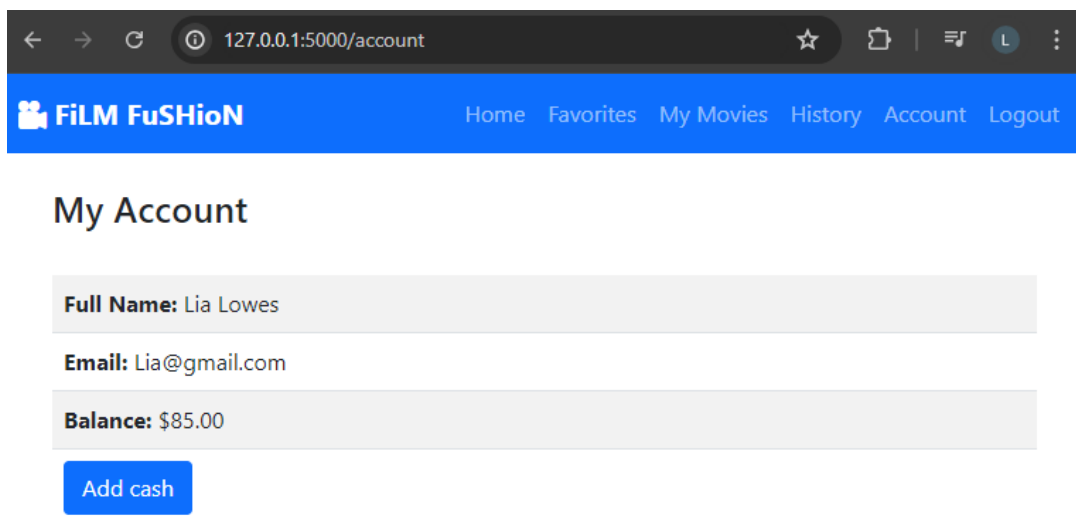
PICTURE 18. MY MOVIES

To show this list of bought movies, the query below is executed. It is similar to the one executed to show favorites, but on the table **Owns**.

```
cur.execute("""SELECT m.* FROM Movies m, Owns o WHERE m.movieID = o.movieID  
AND userID = %s ORDER BY o.time""", (session['userid'],))
```

ACCOUNT AND ADD CASH

The user also has the option to see some information about the account (name, email and balance) shown in the picture below by clicking on the **Account** option on the navigation bar.



PICTURE 19. ACCOUNT

To show this information, the query below is executed. It selects all the information about the user.

```
cur.execute("""SELECT * FROM Users WHERE userID = %s """, (session['userid'],))
```

From this screen, the user has the option to add credit to their account balance by clicking on the **Add cash** button. The user needs to confirm the amount to add to the account and then the following queries are executed. The first one updates the cash value on the Users table for **userID** requested. The second one added this operation to the **Transactions** table with the **transactionType** = “Add cash”.

```
cur.execute("""UPDATE Users SET cash = %s WHERE userID = %s """, (new_balance, session['userid'],))
cur.execute("""INSERT INTO Transactions
              (userID, transactionType, amount, balanceBefore, balanceAfter)
              VALUES (%s, %s, %s, %s, %s)
              """, (session['userid'], "Add cash", amount, cash, new_balance,))
conn.commit()
```

HISTORY

To see all the operations made (purchase, add cash), the user can go to the **History** tab on the navigation bar.

127.0.0.1:5000/history

FiLM FuSHioN

Home Favorites My Movies History Account Logout

Transactions

Transaction Type	Movie	Amount	Final Balance (\$)	Transation Time (\$)
Purchase	Spirit Riding Free: Pony Tales	10.00	120.00	2024-07-18 23:59:17
Purchase	Barbie & Her Sisters in a Pony Tale	5.00	130.00	2024-07-18 23:59:13
Add cash		50.00	135.00	2024-07-18 23:59:01
Purchase	Rain Man	5.00	85.00	2024-07-18 23:45:21
Purchase	Because of Soccer	5.00	90.00	2024-07-18 23:45:18
Purchase	Tarzan	5.00	95.00	2024-07-18 23:45:14
Cash Bonus		100.00	100.00	2024-07-18 22:26:36

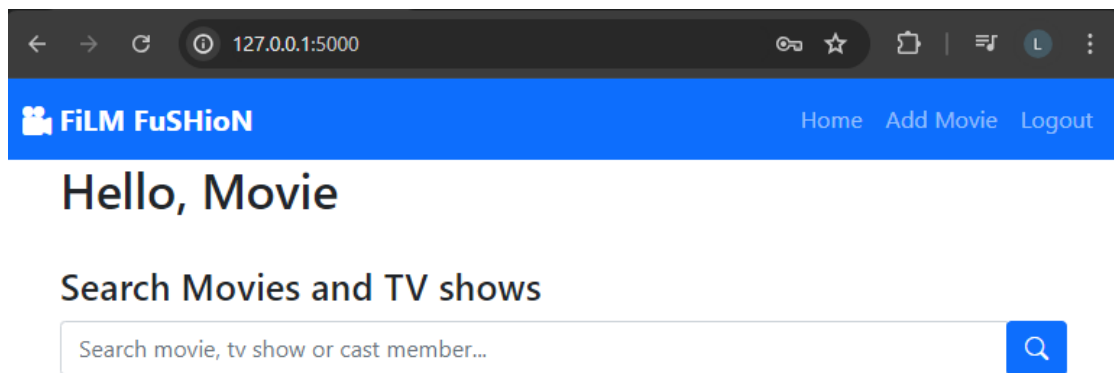
PICTURE 20. HISTORY

To show this information, the following query is executed that retrieves all the transactions that with the matching **userID**.

```
cur.execute("""SELECT * FROM Transactions WHERE userID = %s
              ORDER BY transaction_time DESC""", (session['userid'],))
transactions = cur.fetchall()
```

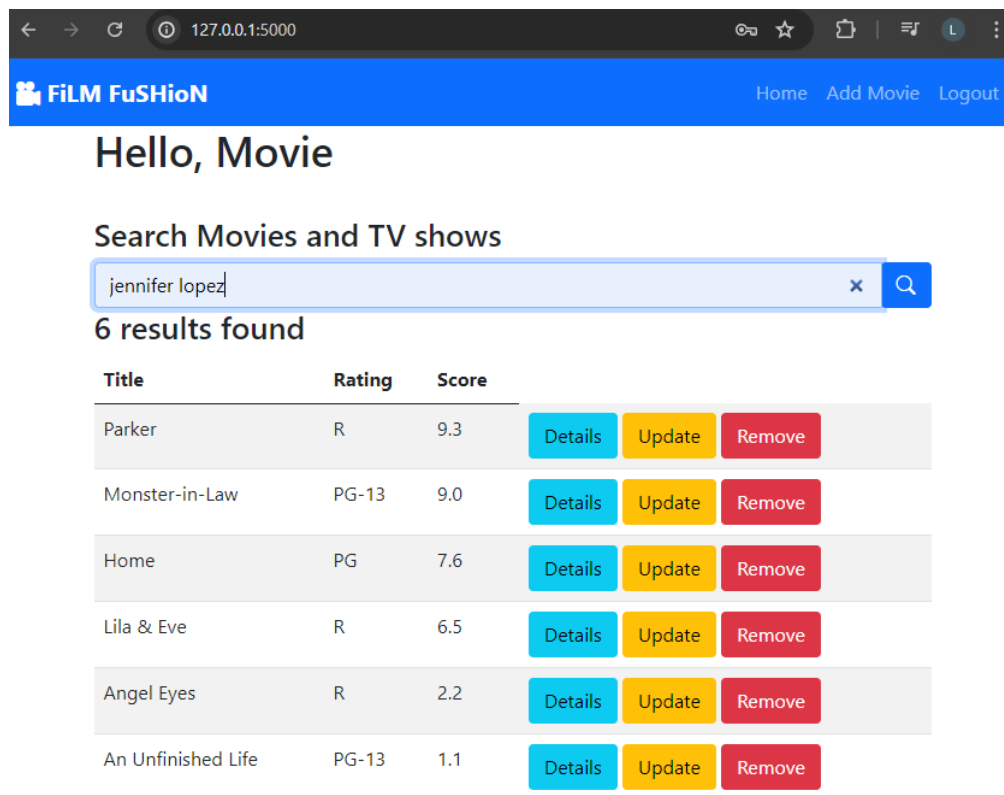
ADMIN USER

We created an admin account for the application. Once this admin logs into the application, the home page has only the search bar and the **Add Movie** option on the navigation bar. This user will have the access to add, update and remove movies from the application.



PICTURE 21. HOME PAGE FOR ADMIN

Once the admin searches for a movie, the app shows the results of the search (it works like the search for a regular user), but with different options per movie. Besides Details (already discussed), it shows the **Update** and **Remove** options.



PICTURE 22. SEARCH FOR ADMIN

ADD MOVIE

Once the admin clicks on **Add Movie** on the navigation bar, the app redirects to the following page with a form that the user needs to fill in, so the movie can be added to the application.

The screenshot shows a web browser at the address 127.0.0.1:5000/add-movie. The page has a blue header with the 'FiLM FuSHion' logo and navigation links for Home, Add Movie, and Logout. The main heading is 'Add a movie'.

The form contains the following fields and controls:

- Movie Title:** A text input field with the placeholder 'Enter movie title'.
- Release year:** A text input field with the placeholder 'Enter release year'.
- Select a type:** Two radio buttons: 'Movie' (selected) and 'TV Show'.
- Price:** A text input field with the placeholder 'Enter price i'.
- Rating:** A dropdown menu.
- Duration:** A text input field with the placeholder 'Enter duration'. Below it, a note reads: 'Enter duration in minutes if it is a movie or enter it as seasons if it is a TV Show'.
- Score:** A text input field with the placeholder 'Enter score'.
- Genres:** A text input field with the placeholder 'Enter genre' and a button 'Add another genre'.
- Actors:** A text input field with the placeholder 'Enter actor' and a button 'Add another actor'.
- Directors:** A text input field with the placeholder 'Enter director' and a button 'Add another director'.
- Description/Sinopsis:** A large text area for the movie's description.
- Submit:** A blue button at the bottom of the form.

PICTURE 23. ADD MOVIE BY ADMIN

Multiple queries are executed once the user submits the form. The first one checks if the movie already exists in the database using the title, type, and release year attributes from the **Movies** table. If it exists, it throws an error message.

```
# checking if the movie already exists in the DB
cur.execute("""SELECT movieID FROM Movies WHERE title = %s
AND type = %s AND released = %s""", (title, type, release_year))
```

If the movie does not exist in the **Movies** table already, the tuple with all the information is added to it.

```
# Insert data into Movies table
cur.execute("""INSERT INTO Movies (title, price, type, duration, released, score, rating, description)
VALUES (%s, %s, %s, %s, %s, %s, %s, %s) RETURNING movieID;
""", (title, price, movie_type, duration, release_year, score, rating, description,))
```

After this, the application also needs to update the **ListedIn** table inserting the tuple (**movieID**, **genreID**). Also, if the genre did not exist in the Genres table, it is added as a new genre to the **Genres** table. To do this, the following code and queries were executed.

```
# Inserting into Genres and ListedIn Tables
for genre in genres:
    # Adding the Genre if the Genre is not in Listed in Genres Table
    cur.execute("""INSERT INTO Genres (name)
VALUES (%s) ON CONFLICT ON CONSTRAINT name_g DO NOTHING
RETURNING genreID;""", (genre,))

    genre_result = cur.fetchone()
    print(genre_result)

    # getting the genre_id
    # if the genre comes from the previous query
    if genre_result:
        genre_id = genre_result[0]
    else:
        # if the genre was ON CONFLICT
        cur.execute("SELECT genreID FROM Genres WHERE name = %s", (genre,))
        genre_result = cur.fetchone()
        print("QUERY SELECT RESULT:" + str(genre_result))
        if genre_result:
            genre_id = genre_result[0]

    # Inserting the genre_id with its respective movie_id if the combination does not exist in the ListedIn table
    if genre_id and movie_id:
        cur.execute("""INSERT INTO ListedIn (movieID, genreID) VALUES (%s, %s)
ON CONFLICT ON CONSTRAINT listedInPK DO NOTHING;""", (movie_id, genre_id))
```

To update the **Actors** and **Works** tables along with the **Directors** and **Directs** tables, similar code, and queries below are executed.

```

# Insert data into Actors and Works tables
for actor in actors:

    # Adding the Actor if the Actor is not in Listed in Actors Table
    cur.execute("""INSERT INTO Actors (name) VALUES (%s)
        ON CONFLICT ON CONSTRAINT name_a DO NOTHING
        RETURNING actorID; """, (actor,))

    actor_result = cur.fetchone()
    print(actor_result)

    # getting the actor_ID
    # if the actor comes from the previous query
    if actor_result:
        actor_id = actor_result[0]
    # if the actor was ON CONFLICT
    else:
        cur.execute("SELECT actorID FROM Actors WHERE name = %s", (actor,))
        actor_result = cur.fetchone()
        print("QUERY SELECT RESULT:" + str(actor_result))
        if actor_result:
            actor_id = actor_result[0]

    # Inserting the actor_id with its respective movie_id if the combination does not exist in the Works table
    if actor_id and movie_id:
        cur.execute("""
            INSERT INTO Works (movieID, actorID)
            VALUES (%s, %s)
            ON CONFLICT ON CONSTRAINT worksPK DO NOTHING;
            """, (movie_id, actor_id))

```

```

# Insert data into Directors and Directs tables
for director in directors:

    # Adding the Director if the Director is not in Listed in Directors Table
    cur.execute("""
        INSERT INTO Directors (name)
        VALUES (%s)
        ON CONFLICT ON CONSTRAINT name_d DO NOTHING
        RETURNING dirID;
        """, (director,))

    dir_result = cur.fetchone()
    print(dir_result)

    # getting the dir_id
    # if the director comes from the previous query
    if dir_result:
        dir_id = dir_result[0]
    else:
        # if the director was ON CONFLICT
        cur.execute("SELECT dirID FROM Directors WHERE name = %s", (director,))
        dir_result = cur.fetchone()
        print("QUERY SELECT RESULT:" + str(dir_result))
        if dir_result:
            dir_id = dir_result[0]

    # Inserting the dir_id with its respective movie_id if the combination does not exist in the Directs table
    if dir_id and movie_id:
        cur.execute("""
            INSERT INTO Directs (movieID, dirID)
            VALUES (%s, %s)
            ON CONFLICT ON CONSTRAINT directsPK DO NOTHING;
            """, (movie_id, dir_id))

```

UPDATE MOVIE

The admin also has the option to update information about a movie by clicking on the **Update** button next to it. Once this happens, the following screen shows up with a form so the user can fill in the information that needs to be changed.

FILM FuSHioN

HomeAdd MovieLogout

Update Movie

TV Show ID: 3407

Title: Spirit Riding Free: Pony Tales

Movie Title

Spirit Riding Free: Pony Tales

Release Year

2019

Select a type: ☐ Movie ☒ TV Show

Price

10.00

Rating

TV-Y7

Duration

Enter duration

Score

10.0

Enter duration in minutes if it is a movie or seasons if it is a TV Show

Genres

Kids' TV

Add another genre

Remove

Actors

Sydney Park

Amber Frank

Bailey Gambertoglio

Darcy Rose Byrnes

Duncan Joiner

Nolan North

Gabriella Graves

Add another actor

Remove

Remove

Remove

Remove

Remove

Remove

Directors

Add another director

Description/Synopsis

Find the fun and adventure of "Spirit Riding Free" in this mix of music videos and short episodes featuring Lucky and all of her friends!

Update

PICTURE 24. UPDATE MOVIE BY ADMIN

To update the information for a movie, multiple queries are executed. The first one below updates the attribute values for the specified movie.

```
# Update movie details
cur.execute("""UPDATE Movies
SET title = %s, type = %s, price = %s, duration = %s, released = %s,
rating = %s, score = %s, description = %s WHERE movieID = %s""",
(title, movie_type, price, duration, release_year, rating, score, description, movieID))
```

After that, the following queries to remove the entries with a matching **movieID** on the table **Genres**, **Actors**, and **Directors** are executed.

```
# Remove existing genres, actors, and directors from ListedIn, Works and Directs Tables
if genres:
    cur.execute("DELETE FROM ListedIn WHERE movieID = %s", (movieID,))
if actors:
    cur.execute("DELETE FROM Works WHERE movieID = %s", (movieID,))
if directors:
    cur.execute("DELETE FROM Directs WHERE movieID = %s", (movieID,))
```

Also, if the genre entered by the user does not exist in the table **Genres**, the application will add the new genre to **Genres** and **ListedIn (movieID, genreID)** with the following code and queries.

```
# Insert new genres
if genres:
    for genre in genres:
        if genre == '':
            continue
        # check if genre already exists in Genres table
        cur.execute("SELECT genreID FROM Genres WHERE name = %s", (genre, ))
        genre_result = cur.fetchone()

        # if the Genre is not listed in the Genres Tables we then insert it in the Genres table
        if not genre_result:
            cur.execute("INSERT INTO Genres (name) VALUES (%s) RETURNING genreID", (genre, ))
            genre_result = cur.fetchone()

        # finally we add an entry to the ListedIn table
        if genre_result:
            genre_id = genre_result[0]
            cur.execute("INSERT INTO ListedIn (movieID, genreID) VALUES (%s, %s)", (movieID, genre_id))
```

The application does the same with **Actors/Works** and **Directors/Directs** with the code and queries below.

```

# Insert new actors
if actors:
    for actor in actors:
        if actor == '':
            continue
        # check if actor already exists in actors table
        cur.execute("SELECT actorID FROM Actors WHERE name = %s", (actor, ))
        actor_result = cur.fetchone()

        # if the Actor is not listed in the Actors Tables we then insert it in the Actors table
        if not actor_result:
            cur.execute("INSERT INTO Actors (name) VALUES (%s) RETURNING actorID", (actor, ))
            actor_result = cur.fetchone()

        # finally we add an entry to the Works table
        if actor_result:
            actorID = actor_result[0]
            cur.execute("INSERT INTO Works (movieID, actorID) VALUES (%s, %s)", (movieID, actorID))

```

```

# Insert new directors
for director in directors:
    if director == '':
        continue

    # check if director already exists in Directors table
    cur.execute("SELECT dirID FROM Directors WHERE name = %s", (director, ))
    director_result = cur.fetchone()

    # if the director is not listed in the Directors Tables we then insert it in the Directors table
    if not director_result:
        cur.execute("INSERT INTO Directors (name) VALUES (%s) RETURNING dirID", (director, ))
        director_result = cur.fetchone()

    # finally we add an entry to the Directs table
    if director_result:
        directorID = director_result[0]
        cur.execute("INSERT INTO Directs (movieID, dirID) VALUES (%s, %s)", (movieID, directorID))

```

REMOVE MOVIE

The admin also has the option to remove a movie by clicking on the **Remove** button. Once the admin confirms that the movie must be deleted, the following query is executed. Since the option ON DELETE CASCADE was added to the **movieID** on all the tables related to it, it will delete the movie from all the tables.

```

# delete movie
cur.execute("DELETE FROM movies WHERE movieid = %s", (movieID,))
conn.commit()

```

Features

CHECK CONSTRAINT

We added CHECK Constraints to ensure that the attributes price in the Movies table and cash in the Users table is always greater or equal and zero.

```
# Creating tables if they do not exist
create_tables_commands = """

CREATE TABLE IF NOT EXISTS Users
(
    userID SERIAL PRIMARY KEY,
    firstname VARCHAR(100) NOT NULL,
    lastname VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR (255) NOT NULL,
    cash NUMERIC (9,2) DEFAULT 100 NOT NULL,
    isAdmin BOOLEAN DEFAULT FALSE NOT NULL
    CONSTRAINT cash_nn CHECK (cash >= 0)
);

CREATE TABLE IF NOT EXISTS Movies
(
    movieID SERIAL PRIMARY KEY,
    price NUMERIC (5,2) NOT NULL DEFAULT 0,
    title VARCHAR (255) NOT NULL,
    type VARCHAR (50) NOT NULL,
    duration VARCHAR (20),
    released SMALLINT,
    rating VARCHAR (10),
    score NUMERIC(3,1),
    description TEXT,
    CONSTRAINT price_nn CHECK (price >= 0)
);

```

DB USERS WITH DIFFERENT PRIVILEGES

To restrict the security of our database we created two database users with different levels of security according to the type of transaction on level access user that users in the Web application would have.

For all transactions related to regular web users we created Database user: appuser


```
# if it does not exist then we create it
if not app_user_exists:
    user_creation_command = """
        CREATE USER appuser WITH PASSWORD 'Lamarca@2024';
        GRANT CONNECT ON DATABASE "Movies_DB" TO appuser;
        GRANT SELECT, INSERT, UPDATE ON TABLE users, owns, transactions TO appuser;
        GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE favorites TO appuser;
        GRANT SELECT ON TABLE works, directs, movies, listedin, actors, directors, genres TO appuser;
        GRANT USAGE, SELECT ON SEQUENCE users_userid_seq, transactions_transactionid_seq TO appuser;
        """
    cur.execute(user_creation_command)
```

As shown in the picture this app user was granted access to connect to the Movies_DB database. Within the database, we gave this user access full access to the favorites table, SELECT, INSERT, and UPDATE on table users, owns, transactions, and only SELECT permissions for movies, actors, directors, genres, works, directs, and listed in tables as regular users are not supposed to change any values on these tables.

For all transactions related to admin web users we created Database user: appuser

```
# if it does not exist then we create it
if not app_admin_exists:
    admin_creation_command = """
        CREATE USER appadmin WITH PASSWORD 'Unalocura@2024';
        GRANT CONNECT ON DATABASE "Movies_DB" TO appadmin;
        GRANT ALL PRIVILEGES ON DATABASE "Movies_DB" TO appadmin;
        GRANT ALL PRIVILEGES ON TABLE users, owns, transactions, favorites, works, directs, movies, listedin, actors, directors, genres TO appadmin;
        GRANT ALL PRIVILEGES ON SEQUENCE users_userid_seq, transactions_transactionid_seq, movies_movieID_seq, actors_actorID_seq, directors_dirID_seq, genres_genreID_seq TO appadmin;
        """
    cur.execute(admin_creation_command)
```

As shown above, the appadmin user has all privileges over all tables in the Movies_DB

PARAMETRIZED QUERIES

As protection for SQL-Injection attacks we considered using psycopg2 (SQL-Python connector) and its capabilities to process queries with parameters. Using parametrized queries allowed us to take input from the user and execute the queries recognizing the user input as plain text and not SQL statements.

To cite an example, when we check whether the email that the input user inserts in the log in form exists we do it by passing the email as a parameter to the query.

```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    conn = db_conn_user()
    cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)

    if request.method == 'POST' and 'email' in request.form and 'password' in request.form:
        email = request.form['email']
        password1 = request.form['password']

        cur.execute('SELECT * FROM users WHERE email = %s', (email,))
        account = cur.fetchone()
```

Through the code for the project, we use this notation to ensure no SQL Injections.

TRANSACTION PROCESSING

For transaction processing we foresaw that errors while processing queries could appear. That is why in most of the queries that inserted/updated data into tables we include a ROLLBACK statement to ensure the ATOMICITY principle of Transactions. In case there is an error during the transaction processing, the system simply does not make any changes in the database and leaves it in its original state.

To cite an example of where this feature is used, we included it while updating or adding new movies as administrator on the site.

```
def update_movie_in_db(movieID, title, movie_type, price, duration, release_year, rating, score, genres, actors, directors, description):  
    if actor == '':  
        continue  
    # check if actor already exists in actors table  
    cur.execute("SELECT actorID FROM Actors WHERE name = %s", (actor, ))  
    actor_result = cur.fetchone()  
  
    # if the Actor is not listed in the Actors Tables we then insert it in the Actors table  
    if not actor_result:  
        cur.execute("INSERT INTO Actors (name) VALUES (%s) RETURNING actorID", (actor, ))  
        actor_result = cur.fetchone()  
  
    # finally we add an entry to the Works table  
    if actor_result:  
        actorID = actor_result[0]  
        cur.execute("INSERT INTO Works (movieID, actorID) VALUES (%s, %s)", (movieID, actorID))  
  
    for director in directors:  
        if director == '':  
            continue  
  
        # check if director already exists in Directors table  
        cur.execute("SELECT dirID FROM Directors WHERE name = %s", (director, ))  
        director_result = cur.fetchone()  
  
        # if the director is not listed in the Directors Tables we then insert it in the Directors table  
        if not director_result:  
            cur.execute("INSERT INTO Directors (name) VALUES (%s) RETURNING dirID", (director, ))  
            director_result = cur.fetchone()  
  
        # finally we add an entry to the Directs table  
        if director_result:  
            directorID = director_result[0]  
            cur.execute("INSERT INTO Directs (movieID, dirID) VALUES (%s, %s)", (movieID, directorID))  
  
    conn.commit()  
    print("Movie Updated Successfully")  
    value = True  
  
    except Exception as e:  
        conn.rollback()  
        message = str(e)  
        print(message)  
        value = False  
  
    finally:  
        cur.close()  
        conn.close()  
        return value
```

JAVASCRIPT

To enhance the user experience, we used JavaScript.

- We used JavaScript to dynamically create HTML objects based on user event triggers.

For example in the form for adding movies and updating movies, the fields of genres, directors and actors could accept more than one entry so we used buttons and JavaScript functions to dynamically create HTML elements.

Movie Title: Enter movie title

Release year: Enter release year

Select a type: ☒ Movie ☐ TV Show

Price: Enter price in USD

Rating: [Dropdown menu]

Duration: Enter duration

Score: Enter score

Enter duration in minutes if it is a movie or enter it as seasons if it is a TV Show

Genres: Enter genre, Enter genre, Enter genre, Enter genre, Add another genre

Actors: Enter actor, Enter actor, Enter actor, Add another actor

Directors: Enter director, Enter director, Add another director

Example of functions used:

```
// function to add another entry for another actor
window.addActor = function() {
  const container = document.getElementById('actors-container');
  const inputGroup = document.createElement('div');
  inputGroup.className = 'input-group mb-2';
  inputGroup.innerHTML = `
    <input type="text" class="form-control" name="actors" placeholder="Enter actor" required/>
    <button type="button" class="btn btn-danger" onclick="removeInput(this)">Remove</button>
  `;
  container.appendChild(inputGroup);
}
```

- We used Javascript (in this case as JQuery and AJAX) for communicating with the back, processing a route and its functions without having to refresh a page or send a form.

We used this during the Search function as we wanted the user to be able to search without having to constantly refresh the page for new searches.

For this we handled the search functionality using AJAX (Asynchronous JavaScript and XML),

sending the request to process the search in the backend and then returning the result to the front end.

To cite some examples, these were the AJAX functions we included in our project for searching movies.

```
// Functions to perform search

// case the user clicked the search icon
$('#search_button').click(function() {
    performSearch();
});

// case the user pressed enter on the search text area
$('#search_text').keypress(function(event){
    if (event.which == 13) {
        performSearch();
    }
});

// Function to perform search

function performSearch() {
    var search = $('#search_text').val();
    if (search != '') {
        load_data(search);
    } else {
        load_data();
    }
}

// "Load the data" this function passes the "query" to the /search route as a POST request
function load_data(query = '') {
    $.ajax({
        url: "/search",
        method: "POST",
        data: { query: query }, // we will retrieve this 'query' value in the /search route
        success: function(data) {
            $('#result').html(data);
            $('#result').append(data.htmlresponse);
        }
    });
}
```

Conclusions:

Our project successfully developed a comprehensive online movie platform that allows both regular users and admin users to interact with the application in meaningful ways. Through our use of modern web technologies, we created a user-friendly interface.

Overall, the project was a valuable learning experience since by modern technologies and best practices, we successfully created a robust application. We were able to apply the knowledge learned during class to make our database more efficient and more secure, guaranteeing the proper functioning of the application.

Appendix -1:

Database Schema:

```
CREATE TABLE IF NOT EXISTS Users
(
    userID SERIAL PRIMARY KEY,
    firstname VARCHAR(100) NOT NULL,
    lastname VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR (255) NOT NULL,
    cash NUMERIC (9,2) DEFAULT 100 NOT NULL,
    isAdmin BOOLEAN DEFAULT FALSE NOT NULL
    CONSTRAINT cash_nn CHECK (cash >= 0)
);
```

```
CREATE TABLE IF NOT EXISTS Movies
(
    movieID SERIAL PRIMARY KEY,
    price NUMERIC (5,2) NOT NULL DEFAULT 0,
    title VARCHAR (255) NOT NULL,
    type VARCHAR (50) NOT NULL,
    duration VARCHAR (20),
    released SMALLINT,
    rating VARCHAR (10),
    score NUMERIC(3,1),
    description TEXT,
    CONSTRAINT price_nn CHECK (price >= 0)
```

```
);
```

```
CREATE TABLE IF NOT EXISTS Genres
```

```
(
```

```
    genreID SERIAL PRIMARY KEY,
```

```
    name VARCHAR(100) NOT NULL,
```

```
    CONSTRAINT name_g UNIQUE (name)
```

```
);
```

```
CREATE TABLE IF NOT EXISTS Actors
```

```
(
```

```
    actorID SERIAL PRIMARY KEY,
```

```
    name VARCHAR(100) NOT NULL,
```

```
    CONSTRAINT name_a UNIQUE (name)
```

```
);
```

```
CREATE TABLE IF NOT EXISTS Directors
```

```
(
```

```
    dirID SERIAL PRIMARY KEY,
```

```
    name VARCHAR(100) NOT NULL,
```

```
    CONSTRAINT name_d UNIQUE (name)
```

```
);
```

```
CREATE TABLE IF NOT EXISTS Favorites
```

```
(
```

```
    userID INTEGER REFERENCES Users ON DELETE CASCADE,
```

```
    movieID INTEGER REFERENCES Movies ON DELETE CASCADE,
```

```
CONSTRAINT favsPK PRIMARY KEY(userID,movieID),
time TIMESTAMP(0) DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TABLE IF NOT EXISTS Owns
```

```
(
  userID INTEGER REFERENCES Users ON DELETE CASCADE,
  movieID INTEGER REFERENCES Movies ON DELETE CASCADE,
  CONSTRAINT ownsPK PRIMARY KEY(userID,movieID),
  time TIMESTAMP(0) DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TABLE IF NOT EXISTS Works
```

```
(
  movieID INTEGER REFERENCES Movies ON DELETE CASCADE,
  actorID INTEGER REFERENCES Actors ON DELETE CASCADE,
  CONSTRAINT worksPK PRIMARY KEY(movieID, actorID)
);
```

```
CREATE TABLE IF NOT EXISTS Directs
```

```
(
  movieID INTEGER REFERENCES Movies ON DELETE CASCADE,
  dirID INTEGER REFERENCES Directors ON DELETE CASCADE,
  CONSTRAINT directsPK PRIMARY KEY(movieID, dirID)
);
```

```
CREATE TABLE IF NOT EXISTS ListedIn
```



```
(  
    movieID INTEGER REFERENCES Movies ON DELETE CASCADE,  
    genreID INTEGER REFERENCES Genres ON DELETE CASCADE,  
    CONSTRAINT listedInPK PRIMARY KEY(movieID, genreID)  
);
```

CREATE TABLE IF NOT EXISTS Transactions

```
(  
    transactionID SERIAL PRIMARY KEY,  
    userID INTEGER NOT NULL REFERENCES Users ON DELETE  
    CASCADE,  
    transactionType VARCHAR(50) NOT NULL,  
    movie VARCHAR(255),  
    amount NUMERIC (9,2) NOT NULL,  
    balanceBefore NUMERIC (9,2) NOT NULL,  
    balanceAfter NUMERIC (9,2) NOT NULL,  
    transaction_time TIMESTAMP(0) DEFAULT CURRENT_TIMESTAMP  
    NOT NULL  
);
```

NOTE: Database Schema and process to import the Data to the Database can be found in the script named “populate.ipynb” in folder “Populating the Movies_DB” inside the “Movie Platform Project” Folder. T