Ahmedabad University

IMDB - Movie DataBase

Semester: Winter 2021 (4)

Course Name: CSE250 DBMS

Submitted to: Prof. Shefali Naik

Roll No	Name	Programme
AU1940137	Tirth Patel	CSE B-Tech
AU1940157	Rohan Parikh	CSE B-Tech
AU1940265	Hirmay Sandesara	CSE B-Tech

Ahmedabad University

Table of Contents

Description

We have made a database using which a user can view details of any given movie in the database. When the user opens our website, he/she will first see a *login or signup* page (if he/she is not logged in already). If the user does not have an account, they can register on our platform easily. We store sensitive information such as passwords in a hash in our database. We have also added @login_required and @admin_only to avoid any infiltration of some unknown person. Certain functionalities and parameters were also added to avoid any sort of SQL Injection Query attack.

On the homepage of the website, the user will see the top-rated movies present in our database. If they click on any movie title, they will be redirected to a page that displays all the details of that particular movie. Next, users can search for movies that they want to see. We have checked whether the user searching for movies is a kid or not. If he/she is a kid, we don't display movies that are rated 'R'.

No.	Name	Rating
1	The Dark Knight	9.0
2	The Godfather: Part II	9.0
3	Schindler's List	8.9
4	The Lord of the Rings: The Return of the King	8.9
5	Inception	8.8
6	The Lord of the Rings: The Fellowship of the Ring	8.8
7	Fight Club	8.8
8	The Lord of the Rings: The Two Towers	8.7

The search will display the movie titles which consist of the string the user has inputted in the search bar. Upon clicking any of those movie titles, the user will be redirected to a page where they can see the movie's details. To further optimize and customize their search, user's can filter out movies on the basis of their release dates, ratings, genres and which OTT platforms they are available on.

A user can also search for *celebrities* to see their details. After clicking on a celebrity's name, the user will be able to see his details and the movies he/she has worked in. Even here, we make sure that if the user is a minor and the movie the actor has worked on is 'R' rated, we do not display that movie.

On the page where movie details are displayed, there is a *like* button. If a user clicks it, the movie gets a like and if he clicks it again, he will remove the like from that movie.

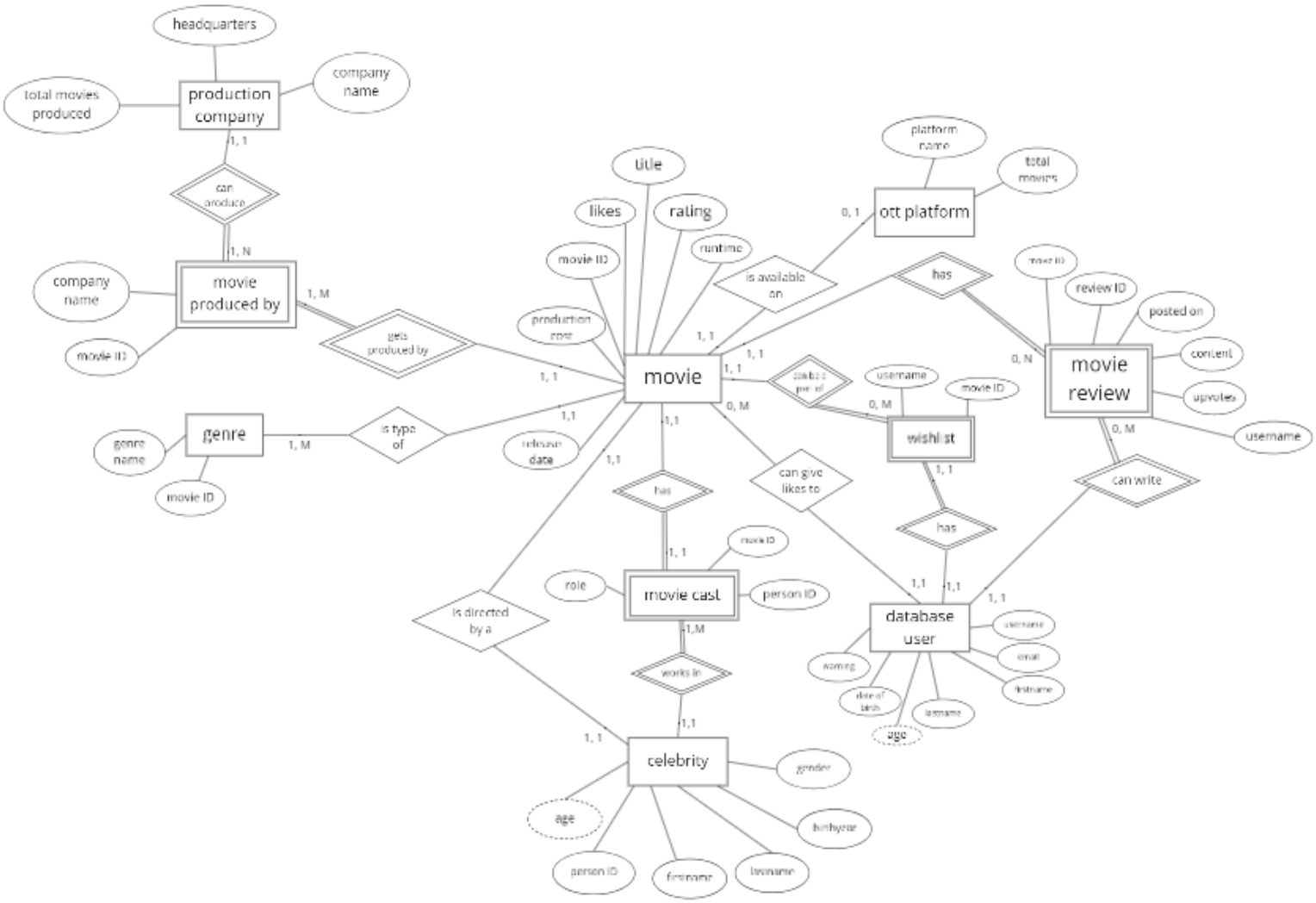
Each user has his or her own watchlist. They can add movies to that watchlist by clicking on the "*add to watchlist*" button and similarly, they can remove that movie from their watchlist.

On the movie display page, the user even has an option to post their review and give upvotes or remove previously given upvotes to the reviews given by other users. The reviews are displayed in a decreasing order. Here, we have added a functionality that if the review has an abusive word, that review will not get posted and the user will receive a warning. If the same user receives *3 warnings*, we block their account from the website and now, they will not be able to log in or sign up using the previously used email ID.

We also give the latest statistics for the movies present in our database - which OTT platform has the most number of movies, average rating of movies on each OTT platform, average rating of movies genre-wise etc.

The website has a separate admin page. An admin can add, delete and update records of all tables like movie, reviews, genres, users, production houses etc.

ER diagram



For better resolution, you can visit: https://miro.com/app/board/o9J_lYdA5_Q=/

OR

https://cdn.discordapp.com/attachments/763788240318234625/832672972375588864/DBMS-Project-Rohan_Updated_11PM_CloseUp.jpg

Table Design (Data Dictionary)

- For table “**celebrity**”

```
                    Table "public.celebrity"
  Column   |         Type          | Collation | Nullable | Default
-----------+-----------------------+-----------+----------+---------
 person_id | character varying(10) |           | not null |
 firstname | character varying(20) |           | not null |
 lastname  | character varying(20) |           | not null |
 birthyear | integer               |           |          |
 age       | integer               |           |          |
 gender    | character varying(10) |           |          |
Indexes:
    "celebrity_pkey" PRIMARY KEY, btree (person_id)
Referenced by:
    TABLE "movie_cast" CONSTRAINT "fk_cast_person_id" FOREIGN KEY (person_id) REFERENCES celebrity(person_id)
    TABLE "show_cast" CONSTRAINT "fk_cast_person_id" FOREIGN KEY (person_id) REFERENCES celebrity(person_id)
    TABLE "movie" CONSTRAINT "fk_movie_director" FOREIGN KEY (director) REFERENCES celebrity(person_id)
```

- For table “**production_company**”

```
                  Table "public.production_company"
     Column     |         Type          | Collation | Nullable | Default
----------------+-----------------------+-----------+----------+---------
 company_name   | character varying(50) |           | not null |
 headquarter    | character varying(40) |           |          |
 total_produced | integer               |           |          |
Indexes:
    "production_company_pkey" PRIMARY KEY, btree (company_name)
Referenced by:
    TABLE "movie_produced_by" CONSTRAINT "fk_movie_company" FOREIGN KEY (company_name) REFERENCES production_company(company_name)
    TABLE "show_produced_by" CONSTRAINT "fk_tv_company" FOREIGN KEY (company_name) REFERENCES production_company(company_name)
```

- For table “**ott_platform**”

```
                     Table "public.ott_platform"
    Column     |         Type          | Collation | Nullable | Default
---------------+-----------------------+-----------+----------+---------
 platform_name | character varying(20) |           | not null |
 total         | integer               |           |          |
Indexes:
    "ott_platform_pkey" PRIMARY KEY, btree (platform_name)
Referenced by:
    TABLE "movie" CONSTRAINT "fk_movie_platform" FOREIGN KEY (platform) REFERENCES ott_platform(platform_name)
    TABLE "tv_show" CONSTRAINT "fk_tv_platform" FOREIGN KEY (platform) REFERENCES ott_platform(platform_name)
```

- For table “**movie**”

```
                          Table "public.movie"
     Column      |         Type          | Collation | Nullable | Default
-----------------+-----------------------+-----------+----------+---------
 movie_id        | character varying(10) |           | not null |
 title           | character varying(60) |           | not null |
 production_cost | double precision      |           |          |
 rating          | double precision      |           |          |
 rated           | character varying(10) |           |          |
 release_date    | date                  |           |          |
 platform        | character varying(20) |           |          |
 likes           | integer               |           |          |
 runtime         | integer               |           |          |
 director        | character varying(10) |           |          |
Indexes:
    "movie_pkey" PRIMARY KEY, btree (movie_id)
Check constraints:
    "movie_rating_check" CHECK (rating > 0::double precision AND rating <= 10::double precision)
Foreign-key constraints:
    "fk_movie_director" FOREIGN KEY (director) REFERENCES celebrity(person_id)
    "fk_movie_platform" FOREIGN KEY (platform) REFERENCES ott_platform(platform_name)
Referenced by:
    TABLE "movie_cast" CONSTRAINT "fk_cast_movie_id" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
    TABLE "movie_produced_by" CONSTRAINT "fk_produced_movie_id" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
    TABLE "movie_review" CONSTRAINT "fk_review_movie_id" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
    TABLE "movie_genre" CONSTRAINT "movie_genre_movie_id_fkey" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
```

- For table “**movie_genre**”

```
                    Table "public.movie_genre"
  Column   |         Type          | Collation | Nullable | Default
----------+-----------------------+-----------+----------+---------
 movie_id | character varying(10) |           | not null |
 genre    | character varying(10) |           | not null |
Indexes:
    "movie_genre_pkey" PRIMARY KEY, btree (movie_id, genre)
Foreign-key constraints:
    "movie_genre_movie_id_fkey" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
```

- For table “**movie_cast**”

```
                    Table "public.movie_cast"
  Column   |         Type          | Collation | Nullable | Default
----------+-----------------------+-----------+----------+---------
 movie_id  | character varying(10) |           | not null |
 person_id | character varying(10) |           | not null |
 role      | character varying(40) |           | not null |
Indexes:
    "movie_cast_pkey" PRIMARY KEY, btree (movie_id, person_id, role)
Foreign-key constraints:
    "fk_cast_movie_id" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
    "fk_cast_person_id" FOREIGN KEY (person_id) REFERENCES celebrity(person_id)
```

- For table “**db_user**”

```
                    Table "public.db_user"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 email         | character varying(50)  |           | not null |
 username      | character varying(20)  |           |          |
 date_of_birth | date                   |           |          |
 firstname     | character varying(20)  |           | not null |
 lastname      | character varying(20)  |           | not null |
 hash          | character varying(100) |           | not null |
 warning       | integer                |           |          |
Indexes:
    "db_user_pkey" PRIMARY KEY, btree (email)
    "db_user_username_key" UNIQUE CONSTRAINT, btree (username)
Referenced by:
    TABLE "movie_review" CONSTRAINT "fk_review_username" FOREIGN KEY (username) REFERENCES db_user(username)
    TABLE "show_review" CONSTRAINT "fk_show_review_username" FOREIGN KEY (username) REFERENCES db_user(username)
Triggers:
    if_user_blocked BEFORE INSERT ON db_user FOR EACH ROW EXECUTE FUNCTION check_if_user_blocked()
    user_delete BEFORE DELETE ON db_user FOR EACH ROW EXECUTE FUNCTION delete_user()
```

- For table “**wishlist**”

```
                    Table "public.wishlist"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 username | character varying(20) |           |          |
 movie_id | character varying(10) |           |          |
```

- For table “**movie_review**”

```
                    Table "public.movie_review"
  Column   |          Type          | Collation | Nullable | Default
-----------+------------------------+-----------+----------+---------
 review_id | character varying(10)  |           | not null |
 posted_on | date                   |           |          |
 content   | character varying(1000) |           |          |
 up_votes  | integer                |           |          |
 movie_id  | character varying(10)  |           |          |
 username  | character varying(20)  |           |          |
Indexes:
    "movie_review_pkey" PRIMARY KEY, btree (review_id)
Foreign-key constraints:
    "fk_review_movie_id" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
    "fk_review_username" FOREIGN KEY (username) REFERENCES db_user(username)
Triggers:
    check_abusive_trigger BEFORE INSERT ON movie_review FOR EACH ROW EXECUTE FUNCTION check_abusive()
```

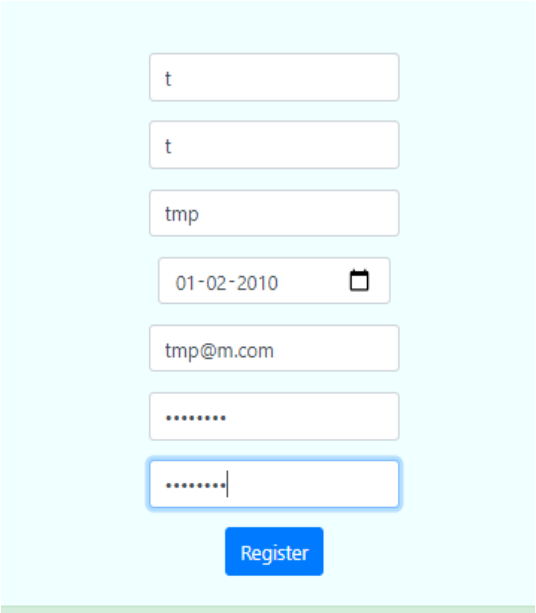
- For table “**movie_produced_by**”

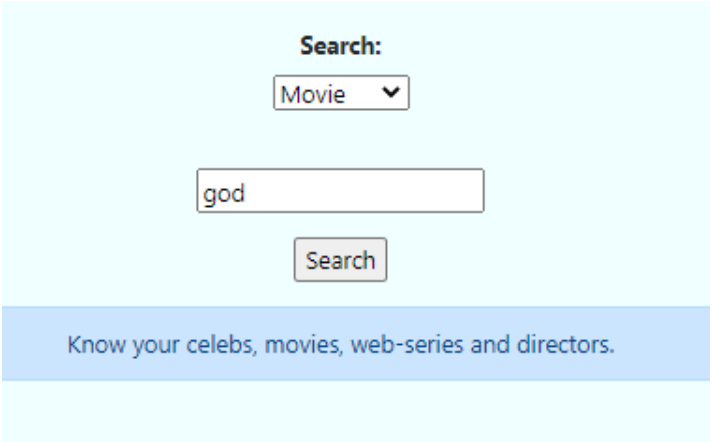
```
                Table "public.movie_produced_by"
    Column     |          Type          | Collation | Nullable | Default
--------------+-----------------------+-----------+----------+---------
 company_name | character varying(50) |           | not null |
 movie_id     | character varying(10) |           | not null |
Indexes:
    "movie_produced_by_pkey" PRIMARY KEY, btree (company_name, movie_id)
Foreign-key constraints:
    "fk_movie_company" FOREIGN KEY (company_name) REFERENCES production_company(company_name)
    "fk_produced_movie_id" FOREIGN KEY (movie_id) REFERENCES movie(movie_id)
```

Procedures and Functions

1) Function to check whether the user is an adult or not

```
CREATE or REPLACE FUNCTION is_adult(dob Date)
returns boolean
language plpgsql
as
$$
DECLARE
BEGIN
    if date_part('year',AGE(CURRENT_DATE,dob))>=18 then
        return TRUE;
    else
        return FALSE;
    END if;
END;
$$;
```





The user is underage so even though The Godfather is a movie present in the database, it will not be displayed as it is an R rated movie.

2) Function to check whether a string is a string is a substring of another string

```
CREATE OR REPLACE FUNCTION contains(parent varchar(100), child
varchar(100))
RETURNS BOOLEAN
```

```
LANGUAGE plpgsql
AS
$$
DECLARE
cnt int;
len int := LENGTH(child);
parentLen int := LENGTH(parent);
begin
cnt:=1;
while cnt + len <= parentLen+1 loop
--raise notice '%,  %', SUBSTR(parent,cnt,len), child;
  if LOWER(SUBSTR(parent,cnt,len)) = LOWER(child) then
return true;
  end if;
  cnt := cnt + 1;
end loop;
--raise notice 'child: %', child;
return false;
end;
$$;
```

This function is used in many of the functions and procedures below.

3) Function to filter movies

```
-----------filter function for movies-----------
CREATE OR REPLACE FUNCTION filtered_movies(
    released_after date default '1000-01-01',
    released_before date default CURRENT_DATE,
    gen varchar(10) default '',
    min_rating float default 0,
    max_rating float default 10,
    plat varchar(20) default ''
)
RETURNS table(
    movie_id varchar
)
language plpgsql
```

```
AS $$
declare
begin
  return query
    select movie_genre.movie_id from movie_genre, movie where(
        movie.release_date >= released_after
        and movie.release_date<= released_before
        and movie_genre.movie_id = movie.movie_id
        and (movie_genre.genre = gen or gen ='')
        and movie.rating >= min_rating
        and movie.rating <= max_rating
        and (contains(movie.platform,plat) or plat = '')
    );
end;
$$;
```

4) Search function for movies such that they are ordered by rating

```
------search function for movies, ordered by rating--------
create or replace function search_movies_by_rating (
  dob Date,
  movie_title varchar(100)
)
    returns table (
        movie_id varchar
    )
    language plpgsql
as $$
begin
    if is_adult(dob) then
        return query
        select movie.movie_id from movie where contains(title,
movie_title) order by rating desc;
    else
    return query
        select movie.movie_id from movie where contains(title,
movie_title)
```

```
        and rated = 'PG-13' order by rating desc;
    end if;
end;
$$;
```

To call the above function:

```
if option == 'Movie':
    cur.execute("SELECT date_of_birth FROM db_user WHERE username=%s", [username])
    datee = cur.fetchall()
    # print(datee)
    # print(search_string)
    query = "SELECT search_movies_by_rating( '"+ str(datee[0][0]) + "' , '" + search_string + "' );"
    cur.execute(query)
    results = cur.fetchall()
```

Movie

No.	Name
1	The Lord of the Rings: The Return of the King
2	The Lord of the Rings: The Fellowship of the Ring
3	The Lord of the Rings: The Two Towers

5) Search function for movies, ordered by likes

```
------search function for movies, ordered by likes--------
create or replace function search_movies_by_likes (
  dob Date,
  movie_title varchar(100)
)
    returns table (
        movie_id varchar
    )
    language plpgsql
as $$
begin
    if is_adult(dob) then
        return query
```

```
        select movie.movie_id from movie where contains(title,
movie_title) order by likes desc;
    else
    return query
        select movie.movie_id from movie where contains(title,
movie_title)
        and rated = 'PG-13' order by likes desc;
    end if;
end;
$$;
```

To call the above function:

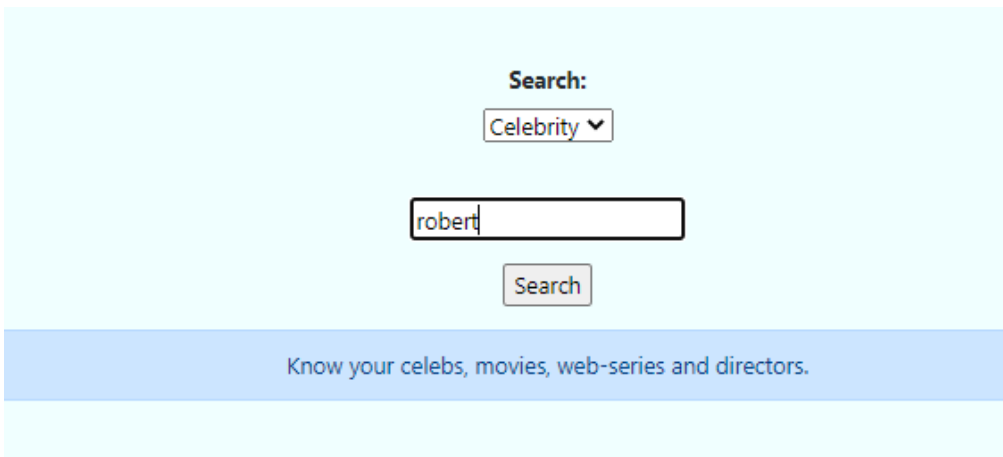
```
if option == 'Movie':
    cur.execute("SELECT date_of_birth FROM db_user WHERE username=%s", [username])
    datee = cur.fetchall()
    # print(datee)
    # print(search_string)
    query = "SELECT search_movies_by_likes( '"+ str(datee[0][0]) + "' , '" + search_string + "'
    cur.execute(query)
    results = cur.fetchall()        Find related code in Internet-Movies-Data-Base
```

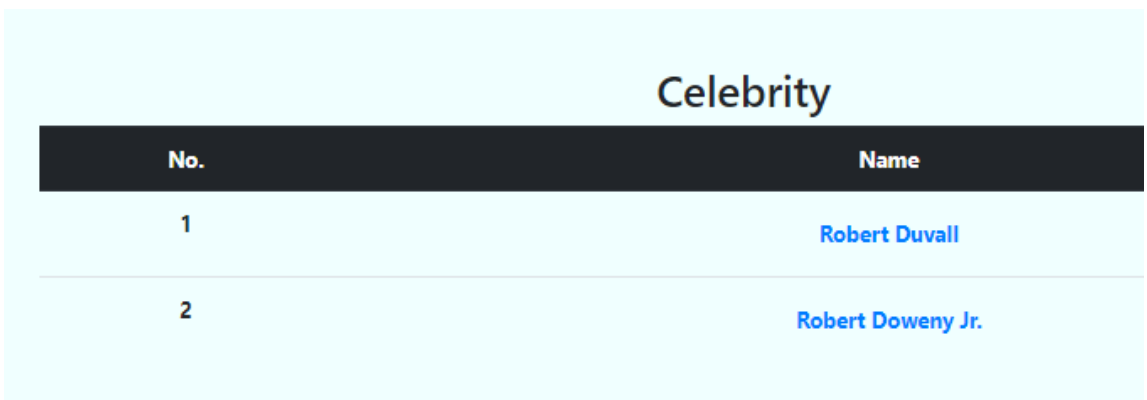
6) Search function for celebrities

```
----------search function for celebs--------
create or replace function search_celebs (
  celeb_name varchar(100)
)
    returns table (
        person_id varchar
    )
    language plpgsql
as $$
begin
    return query
        select celebrity.person_id from celebrity where
        contains(CONCAT(firstname,' ',lastname),celeb_name);
end;
$$;
```

```
elif option == 'Celebrity':
    print(search_string)
    query = "SELECT search_celebs( '" + search_string + "' );"        Find re
# cur.execute("CALL search_movie(%s,%s);",[datee[0], search_string])
# cur.callproc('search_movie',[datee[0], search_string])
# print(query)
    cur.execute(query)
    # print(results)
    results = cur.fetchall()
```

OUTPUT :





7) Display movies in which a given celebrity has worked

```
-------displays the movie a celeb has worked on---------

create or replace function display_celeb_movies (
  celeb_id varchar(100),
  dob DATE
```

```
)
    returns table (
        movie_id varchar
    )
    language plpgsql
as $$
begin
    return query
        select movie_cast.movie_id from movie_cast where person_id =
celeb_id;
end;
$$;

select display_celeb_movies('14','2000-11-11');
```

To call the above function:

```
username = session.get("username")
cur.execute("SELECT date_of_birth FROM db_user WHERE username=%s", [username])
datee = cur.fetchall()
query = "SELECT display_celeb_movies( '" + rows[0] + "' , '"+ str(datee[0][0]) + "' );"
```

OUTPUT :

Robert Duvall

Movies Worked in:

No.	Name
1	The Godfather
2	The Godfather: Part II

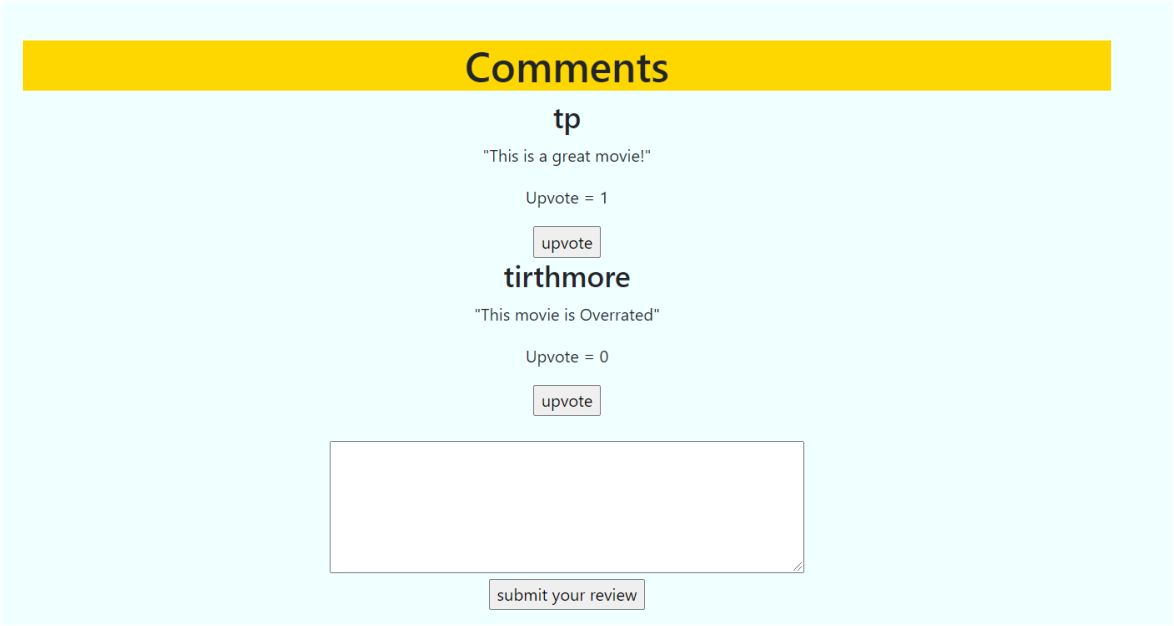
8) Function to display all user reviews on a given movie

```
------------display movie reviews-------------
create or replace function display_movie_reviews (
  mov_ID varchar(10)
)
```

```
    returns table (
        review_Id varchar(10),
        posted_On Date,
        contenT varchar(1000),
        up_Votes int,
        userName varchar(20)
    )
    language plpgsql
as $$
begin
    return query
        select movie_review.review_id, movie_review.posted_on,
movie_review.content, movie_review.up_votes, movie_review.username
        from movie_review where movie_id = mov_ID order by up_votes
desc;
end;
$$
```

To call the above function:

```
cur.execute("SELECT display_movie_reviews(%s);", [movie_id])
reviews = cur.fetchall()
reviews_len = len(reviews)
```



9) Function to display movie details

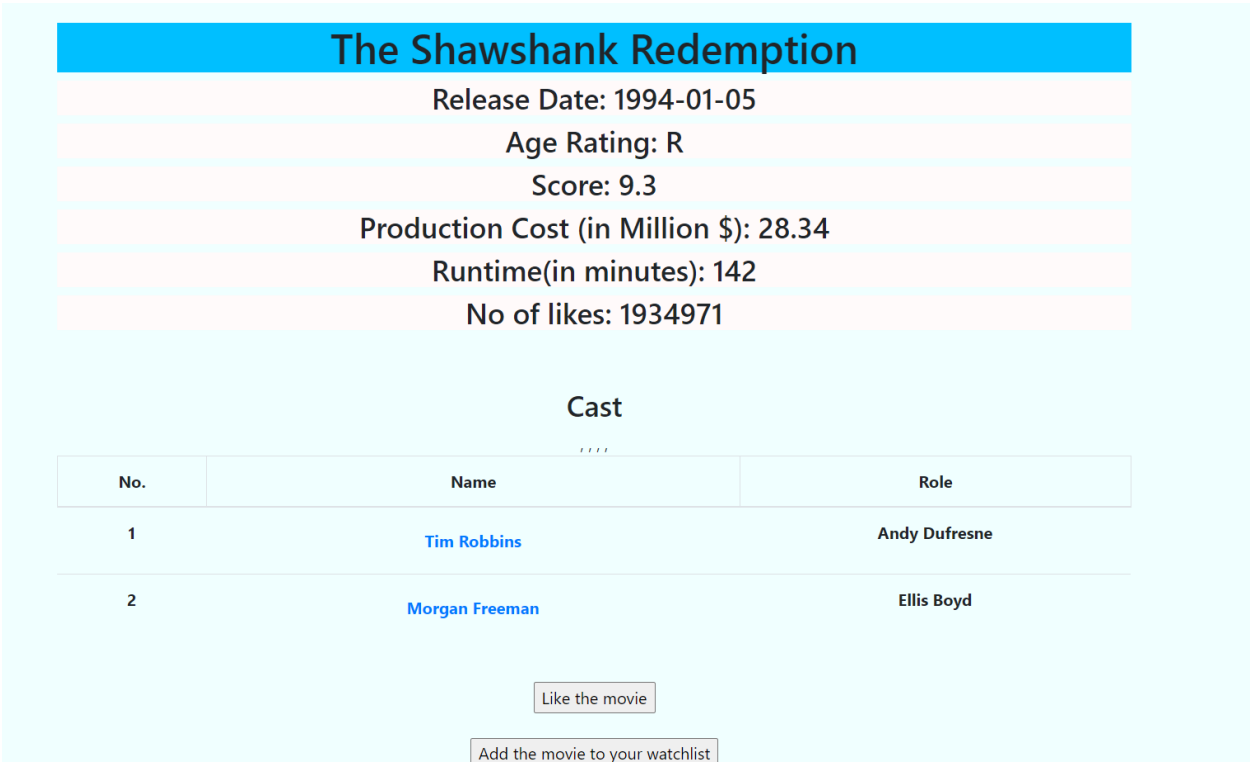
```
-------------display movie details------------
create or replace function display_movies (
  mov_id varchar(100)
)
    returns table (
        movie_id varchar(10),
        title varchar(60),
        production_cost float,
        rating float,
        rated varchar(10),
        release_date Date,
        platform varchar(20),
        likes int,
        runtime int,
        director varchar(10)
    )
    language plpgsql
as $$
begin
    return query
        select movie.movie_id, movie.title, movie.production_cost,
movie.rating,
        movie.rated, movie.release_date, movie.platform, movie.likes,
        movie.runtime, movie.director from movie
        where movie.movie_id = mov_id;
end;
$$;

select display_movies('12');
```

To call the above function:

```
username = session.get("username")
movie_id = request.args.get('movie_id')        Find related co
cur.execute("SELECT display_movies(%s);", [movie_id])
rows = cur.fetchone()
l = len(rows)
```

OUTPUT :



10) Function to add a given movie to the user's wishlist

```
-----------add to wishlist------------
CREATE OR REPLACE PROCEDURE add_to_wishlist(
        username varchar(20), movie_ID varchar(10)
    )
LANGUAGE plpgsql
AS $$
DECLARE
begin
    insert into wishlist (username, movie_ID);
end $$;
```

To call the above function:

```
try:
    watchlist = request.form["watchlist"]
    flash("Movie has been added to the watchlist")
    cur.execute("CALL add_to_wishlist(%s, %s);", [username, movie_id])
    con.commit()
```

OUTPUT :

Movie has been added to the watchlist

Inception

Release Date: 2010-01-05

Age Rating: PG-13

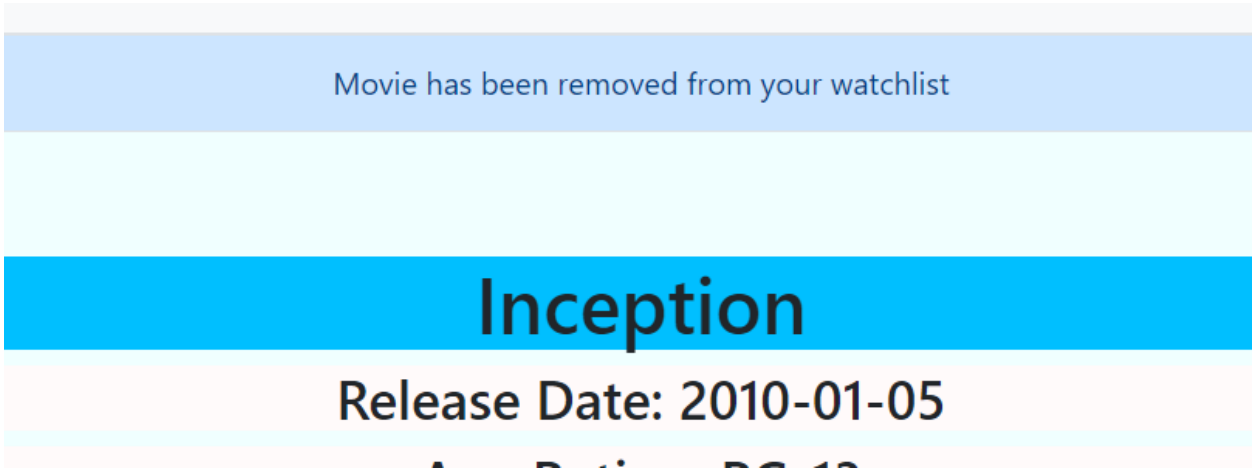
Score: 8.8

11) Function to delete a given movie from the user's wishlist

```
---------------delete from wishlist---------------------
CREATE OR REPLACE PROCEDURE delete_from_wishlist(
        usernam varchar(20), mov_ID varchar(10)
    )
LANGUAGE plpgsql
AS $$
DECLARE
begin
    delete from wishlist where username = usernam and movie_id =
mov_id;
end $$;
```

To call the above function:

```
username = session.get("username")
cur.execute("SELECT delete_from_wishlist(%s, %s);", [username, movie_id])
# row = cur.fetchone()
rows = cur.fetchall()        Find related code in Internet-Movies-Data-Base
```



12) Function to display a user's wishlist

```
-----------------display wishlist ---------------
create or replace function display_wishlist (
        usern varchar(100)
    )
    returns table (
        movie_id varchar
    )
    language plpgsql
as $$
begin
    return query
        select wishlist.movie_id from wishlist where
wishlist.username = usern;
end;
$$;
```

To call the above function:

```
cur = con.cursor()
username = session.get("username")
cur.execute("SELECT display_wishlist(%s);", [username])
# row = cur.fetchone()
rows = cur.fetchall()
```

Watchlist

No.	Name
1	Inception
2	The Godfather
3	The Shawshank Redemption

13) Function to get genre-wise rating for graph

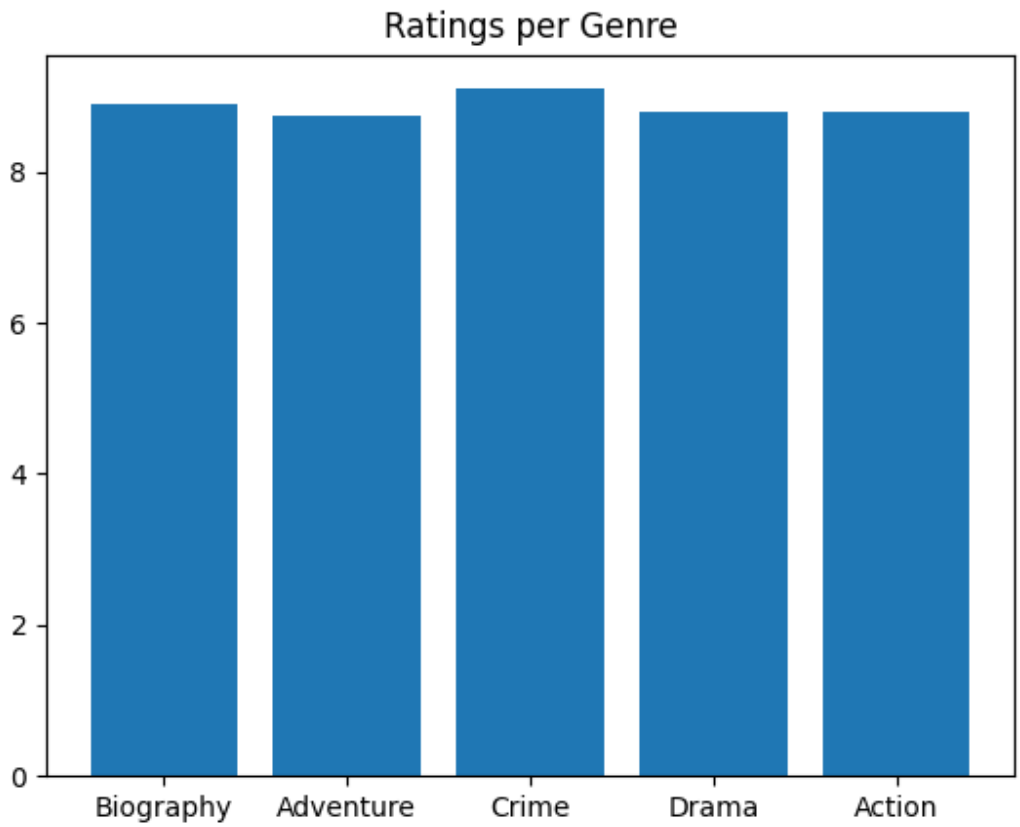
```
----------get genre rating for graph------------
create or replace function get_genre_rating ()
    returns table (
        gen varchar(100),
        avg_rating float
    )
    language plpgsql
as $$
declare
    r_movie_genre record;
    r_movie record;
    r_gens record;
    cnt float:=0;
    total int:=0;
begin
    delete from genre_rating;
    for r_gens in select distinct genre from movie_genre loop
        cnt:=0;
        total:=0;
```

```
      for r_movie_genre in select * from movie_genre where genre =
r_gens.genre loop
        for r_movie in select * from movie
        where movie.movie_id = r_movie_genre.movie_id loop
            cnt:=cnt+r_movie.rating;
            total:=total + 1;
        end loop;
      end loop;
      insert into genre_rating(genre,rating)
values(r_gens.genre,cnt/total);
    end loop;
    return query
        select genre, rating from genre_rating;
end;
$$;
```

To call the above function:

```
def insights():
    results_genre= []
    results_ott = []
    query = "select get_genre_rating();"        Find related
    cur.execute(query)
    results = cur.fetchall()
    for result in results:
```

Graph Obtained:



Ratings per Genre graph is plotted with the help of the sql function to get a better analysis of the Movie Database for the administrator.

14) Function to get OTT platform wise rating

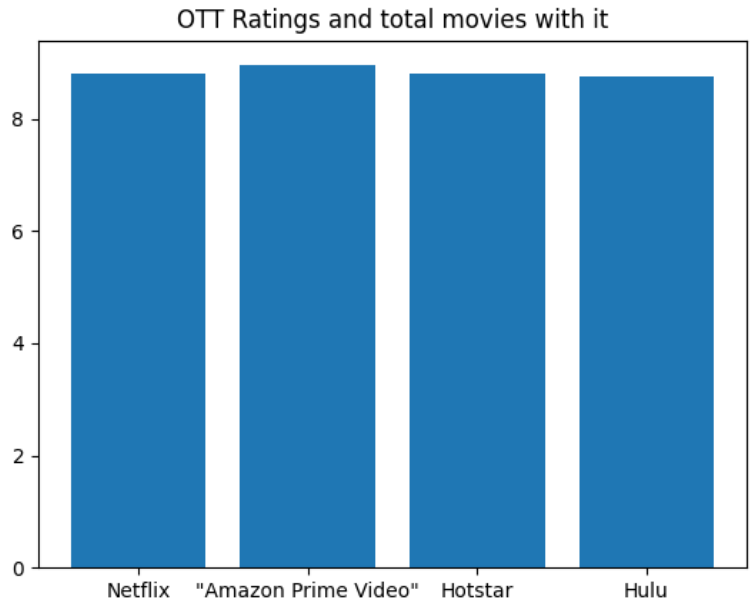
```
------get ott rating and total movies on that platform for graph---
create or replace function get_ott_rating ()
    returns table (
        platform varchar(100),
        avg_rating float,
        total_movies int
    )
    language plpgsql
as $$
declare
    r_ott record;
    r_movie record;
```

```
    cnt float:=0;
    total int:=0;
begin
    delete from ott_rating;
    for r_ott in select * from ott_platform loop
        cnt:= 0;
        total:= 0;
        for r_movie in select * from movie loop
          if r_movie.platform = r_ott.platform_name then
            cnt:= cnt + r_movie.rating;
            total:=total+1;
          end if;
        end loop;
        insert into ott_rating(platform, rating, total)
values(r_ott.platform_name, cnt, total);
    end loop;
    return query
        select ott_rating.platform, rating, ott_rating.total from
ott_rating;
end;
$$;
```

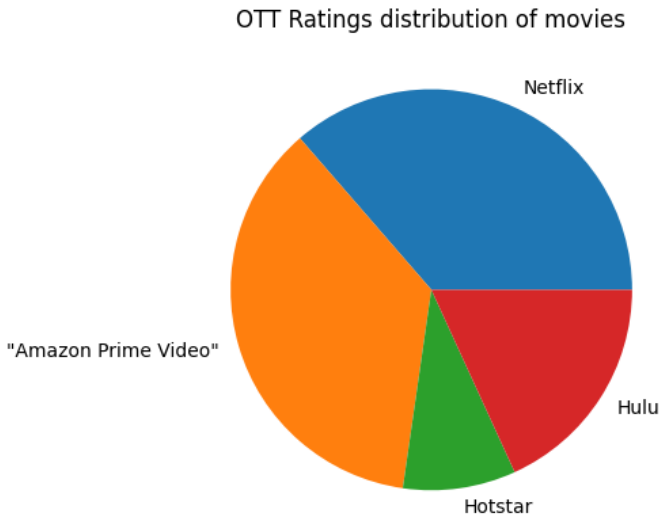
To call the above function:

```
query = "select get_ott_rating();"
cur.execute(query)
results = cur.fetchall()        Find related code in Internet-Mo
for result in results:
    result = result[0][1:len(result[0])-1]
```

Two Graphs Generated:



First OTT Ratings and total movies with graph is plotted with the help of the sql function to get a better analysis of the platforms in Movie Database for the administrator.



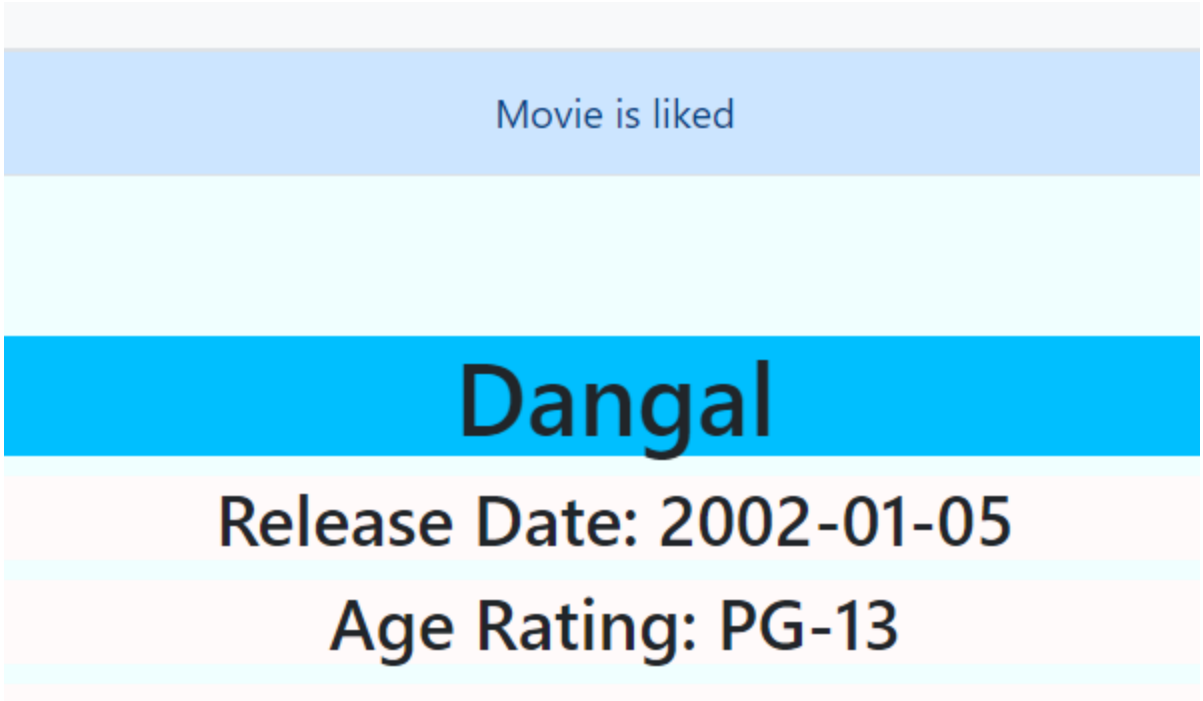
OTT distribution plotted with the help of the sql function to get a better analysis of the platforms in Movie Database for the administrator.

15) Function to add like to a movie or remove like from that movie if it is already liked

```
--add like to movie or remove like if the movie is already liked--
create or replace procedure add_like(mov_id varchar(10), usernam
varchar(20))
language plpgsql
as $$
declare
    r_liked_movies record;
    flag int:=0;
begin
    for r_liked_movies in select * from liked_movies loop
      if r_liked_movies.movie_id = mov_id and r_liked_movies.username
= usernam then
        flag:=1;
        update movie set likes = likes - 1 where movie_id = mov_id;
        delete from liked_movies where movie_id = mov_id and username
= usernam;
      end if;
    end loop;
    if flag = 0 then
        update movie set likes = likes + 1 where movie_id = mov_id;
        insert into liked_movies(movie_id, username) values(mov_id,
usernam);
    end if;
end;
$$;
```

To call the above function:

```
try:
    like =  request.form["like"]
    flash("Movie is liked")
    # Insert into queries remaining
    cur.execute("CALL add_like(%s, %s);", [movie_id, username])
    con.commit()
```



16) Procedure to add upvote to a review or remove upvote from that review if it is already upvoted.

```
----add upvote to review or remove upvote if already upvoted -----
create or replace procedure add_upvote(rev_id varchar(10), usernam
varchar(20))
language plpgsql
as $$
declare
    r_upvoted_reviews record;
    flag int:=0;
begin
  for r_upvoted_reviews in select * from upvoted_reviews loop
      if r_upvoted_reviews.review_id = rev_id and
r_upvoted_reviews.username = usernam then
        flag:=1;
        update movie_review set up_votes = up_votes - 1 where
review_id = rev_id;
        delete from upvoted_reviews where review_id = rev_id and
username = usernam;
      end if;
    end loop;
```

```
    if flag = 0 then
        update movie_review set up_votes = up_votes + 1 where
review_id = rev_id;
        insert into upvoted_reviews(review_id, username)
values(rev_id, usernam);
    end if;
end;
$$;
```

To call the above function:

```
  try:
      upvote = request.form["upvote"]        Find related code in Internet-Movi
      review_id = request.form["review_id"]
      print(review_id)
      flash("Upvoted the Review")
      cur.execute("CALL add_upvote(%s, %s);", [review_id, username])
      con.commit()
```

Upvoted the Review

The Shawshank Redemption

Release Date: 1994-01-05

Age Rating: R

Score: 9.3

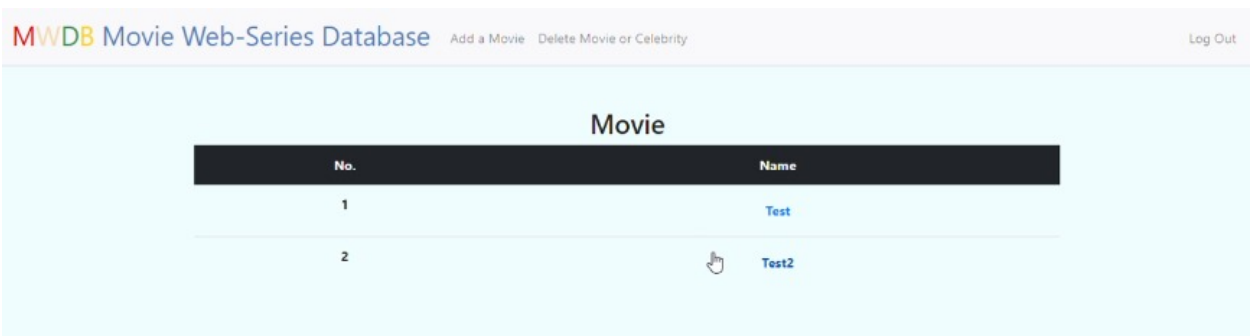
Triggers

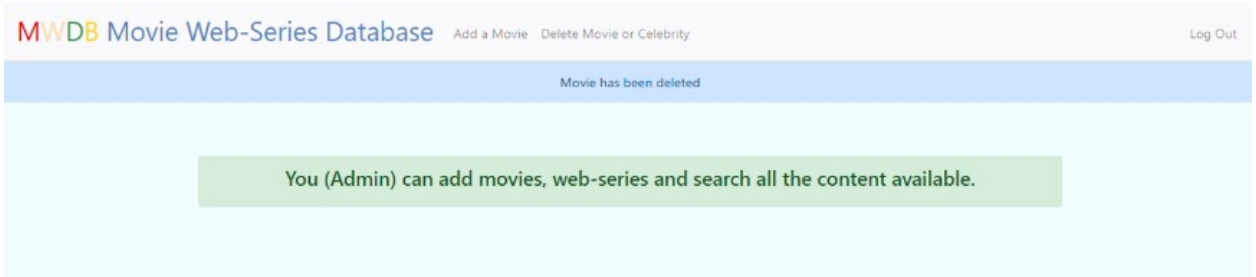
1) Trigger to delete movies (only admin can do this)

```
------------------trigger to delete movies------------------
CREATE OR REPLACE FUNCTION delete_movies() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
begin
```

```
  delete from movie_genre where movie_genre.movie_id = OLD.movie_id;
  delete from movie_produced_by where movie_produced_by.movie_id =
OLD.movie_id;
  delete from movie_cast where movie_cast.movie_id = OLD.movie_id;
  delete from movie_review where movie_review.movie_id =
OLD.movie_id;
  update ott_platform set total = total -1 where platform_name =
OLD.platform;
  RETURN OLD;
end;
$$;

CREATE TRIGGER movie_delete
BEFORE DELETE ON movie
FOR EACH ROW
EXECUTE PROCEDURE delete_movies();
```





2) Trigger to delete celebrities (only admin can do this)

```
------trigger to delete celebs-------
CREATE OR REPLACE FUNCTION delete_celeb() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
begin
  delete from movie_cast where movie_cast.person_id = OLD.person_id;
  --delete from show_cast where show_cast.person_id = OLD.person_id;
  update movie set director = NULL where director = OLD.person_id;
  RETURN OLD;
end;
$$;

CREATE TRIGGER celeb_delete
BEFORE DELETE ON celebrity
FOR EACH ROW
EXECUTE PROCEDURE delete_celeb();
```

Since this is a backend trigger, there is no output available for it.

3) Trigger to check abusive words in a given movie review given by use

```
------if review contains an abusive word trigger will fire----
CREATE OR REPLACE FUNCTION check_abusive() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    r_abusive record;
    r_user record;
```

```
begin
    for r_abusive in select words from abusive_words loop
      if r_abusive.words in (select
unnest(string_to_array(NEW.content,' '))) then
        UPDATE db_user SET warning = warning + 1 where username =
NEW.username;
        for r_user in select * from db_user loop
          if r_user.username = NEW.username then
            r_user.warning:=r_user.warning + 1;
          end if;
        end loop;
        raise exception using message = 'Abusive language detected';
        RETURN OLD;
      end if;
    end loop;
    RETURN NEW;
end;
$$;

CREATE TRIGGER check_abusive_trigger
BEFORE INSERT ON movie_review
FOR EACH ROW
EXECUTE PROCEDURE check_abusive();
```

Use of abusive language detected. You are given a warining. If your warnings exceed 3 then your account will be banned!

User will get the above displayed warning if he tries to enter an abusive word in the review.

4) Trigger to add 1 to number of total movies of OTT platform when a movie is inserted

```
-------add ott platform if it doesn't exist else update count----
CREATE OR REPLACE FUNCTION add_ott() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
    r_ott record;
```

```
begin
  if NEW.platform is NULL then
    RETURN NEW;
  end if;
  for r_ott in select * from ott_platform loop
    if r_ott.platform_name = NEW.platform then
        update ott_platform set total = total + 1 where platform_name
= NEW.platform;
        RETURN NEW;
    end if;
  end loop;
  insert into ott_platform(platform_name, total) values(NEW.platform,
1);
  RETURN NEW;
end;
$$;

CREATE TRIGGER ott_add
BEFORE INSERT ON movie
FOR EACH ROW
EXECUTE PROCEDURE add_ott();
```

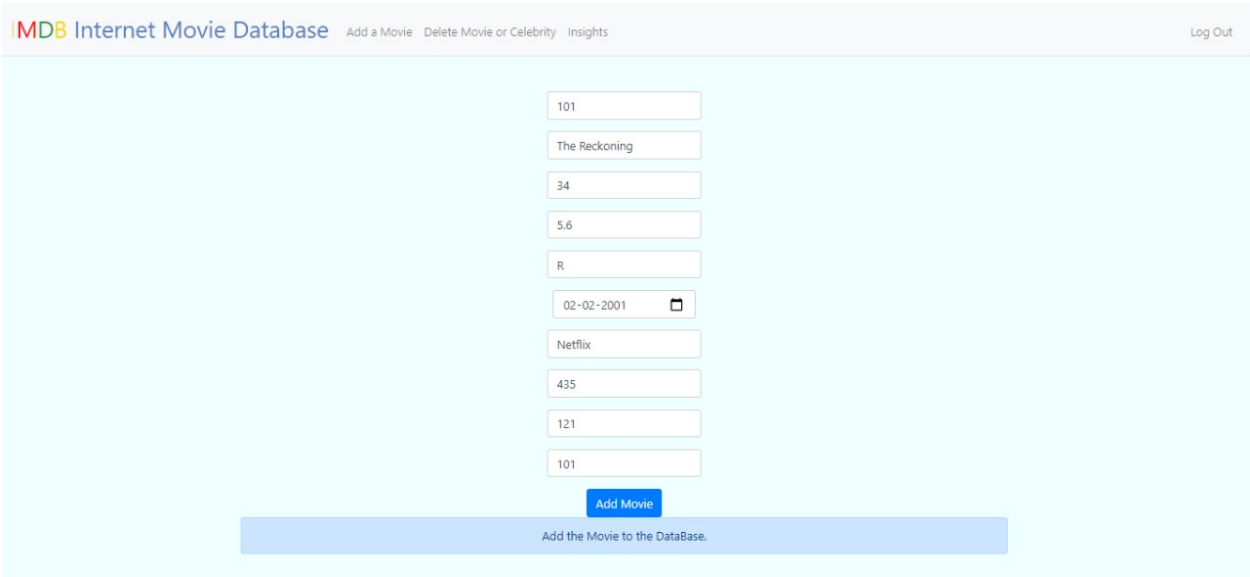
5) Trigger to add 1 to number of total movies of Production company when a movie is inserted

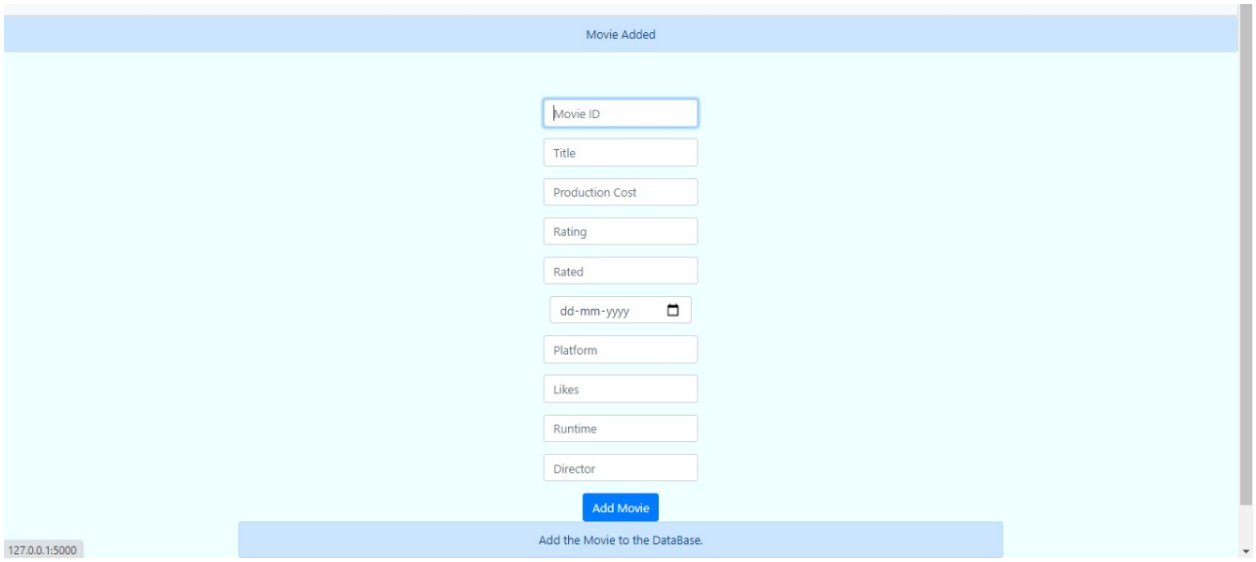
```
----add production house if it doesn't exist else update count-----
CREATE OR REPLACE FUNCTION add_prod() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
    r_prod record;
begin
  for r_prod in select * from Production_company loop
    if r_prod.company_name = NEW.company_name then
        update Production_company set total_produced = total_produced
+ 1
          where company_name = NEW.company_name;
        RETURN NEW;
    end if;
```

```
  end loop;
  insert into Production_company(company_name, total_produced)
  values(NEW.company_name, 1);
  RETURN NEW;
end;
$$;

CREATE TRIGGER prod_add
BEFORE INSERT ON movie_produced_by
FOR EACH ROW
EXECUTE PROCEDURE add_prod();
```

Since this is a backend trigger, there is no output available for it.





This is the display of us testing the trigger

6) Trigger to delete from movie_produced_by

```
CREATE OR REPLACE FUNCTION delete_mov_prod() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
    r_prod record;
begin
  for r_prod in select * from Production_company loop
    if r_prod.company_name = OLD.company_name then
        update Production_company set total_produced = total_produced
- 1
          where company_name = OLD.company_name;
        RETURN OLD;
    end if;
  end loop;
end;
$$;

CREATE TRIGGER mov_prod_delete
BEFORE DELETE ON movie_produced_by
FOR EACH ROW
EXECUTE PROCEDURE delete_mov_prod();
```

Since this is a backend trigger, there is no output available for it.

7) Trigger to delete users

```
-------trigger to delete user ---------
CREATE OR REPLACE FUNCTION delete_user() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
begin
  delete from movie_review where movie_review.username =
OLD.username;
  RETURN OLD;
end;
$$;

CREATE TRIGGER user_delete
BEFORE DELETE ON db_user
FOR EACH ROW
EXECUTE PROCEDURE delete_user();
```

Since this is a backend trigger, there is no output available for it.

8) Trigger to check whether user is blocked before letting him sign up

```
-----------trigger to check if user is blocked-----------
CREATE OR REPLACE FUNCTION check_if_user_blocked() RETURNS TRIGGER
LANGUAGE plpgsql
as $$
DECLARE
    r_blocked_user record;
begin
  for r_blocked_user in select * from blocked_user loop
      if r_blocked_user.email = NEW.email then
        raise exception using message = 'This email has been blocked.
Contact the admin if you think this is a mistake.';
        return OLD;
      end if;
  end loop;
  RETURN NEW;
```

```
end;
$$;

CREATE TRIGGER if_user_blocked
BEFORE INSERT ON db_user
FOR EACH ROW
EXECUTE PROCEDURE check_if_user_blocked();
```

This email has been blocked. Contact the admin if you think this is a mistake.

In the above screenshot you can see the error message that the used email is blocked.

Conclusion

MDB is a web application developed as a handy one-stop destination for a user who is interested to see the ratings, trailers, show times and streaming information of movies. It is successful in meeting the application requirements and there is a lot of scope for future development. However there are few challenges faced and drawbacks in this application, which would be taken care of during the future work.