TECHNISCHE UNIVERSITÄT CHEMNITZ

ADVANCE MANAGEMENT OF DATA

TERM PAPER ON

DIGITAL GRADING SYSTEM

WEB ENGINEERING

SUBMITTED BY

NAME	**MATRICULATION NO**
MUSTANSIR HUSSAIN	706120
ASHISH KUMAR PATHAK	697211

Work Done by:

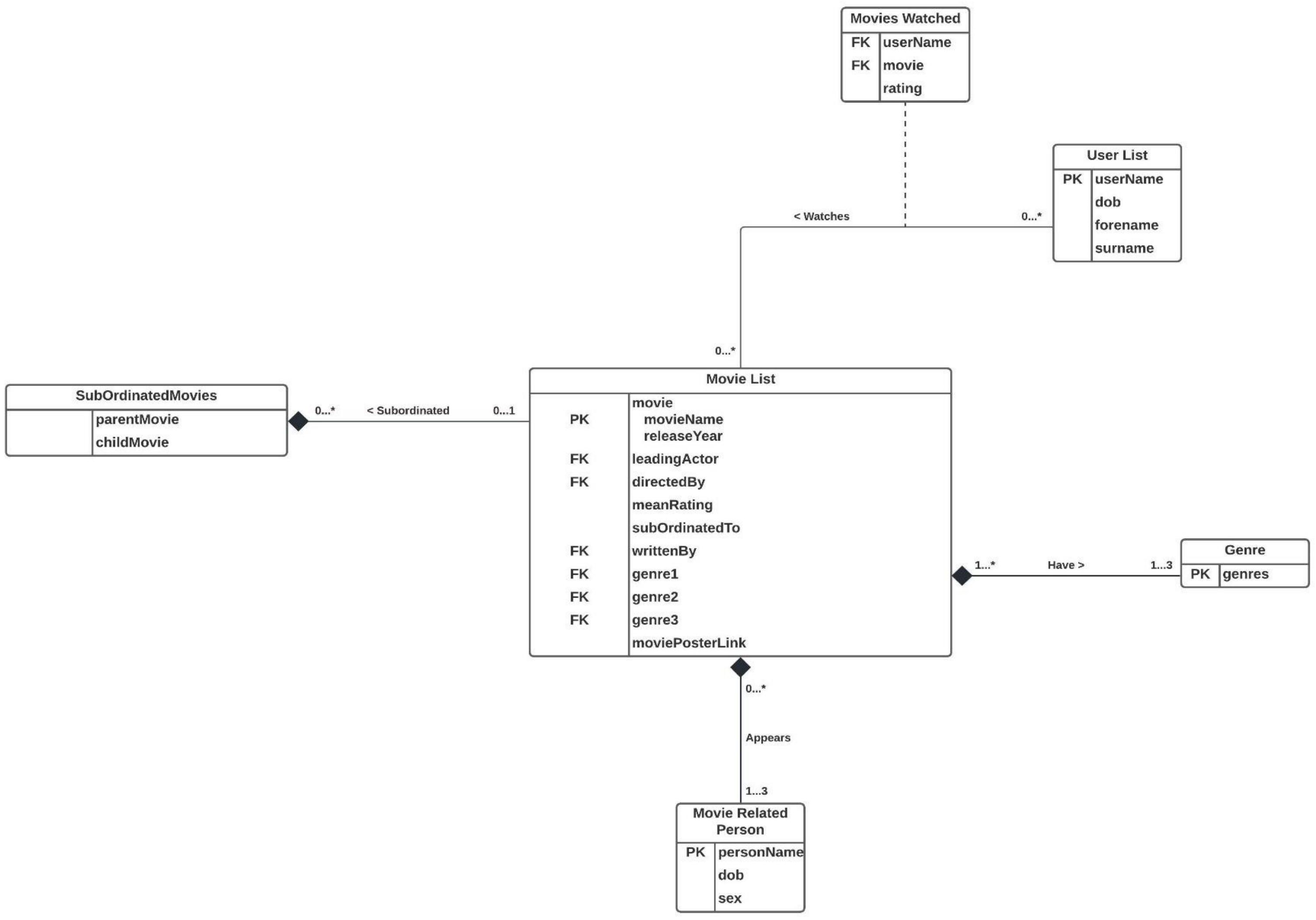
Mustansir Hussain: Backend, Database Design, Suggestions, Triggers.

Ashish Kumar Pathak: Frontend, Backend, Algorithm for managing sub-ordinated movie, Movie Related Person

Both: UML, Schema.

INDEX

UML Diagram



Movie Related Person (personName, dob, sex)

SubOrdinatedMovies (parentMovie, childMovie)

Movies Watched (**userName, movie,** ratings)

User List (userName, forename, surename, dob)

Movie List (movie, **leadingActor, directedBy, writtenBy, genre1, genre2, genre3**, moviePosterlink)

Database Functional Overview:

`movie_list` table

This table has the gist of our database and contain all the elements needed for a functional database as per the requirement.

Column Types

Most of the column types are standard varchar and double expect for the primary key that we are using names: *movie_and_year*

This is a composite type element which contains movie-name and movie release year. We went with this approach because we thought it would be beneficial for the database having to avoid one extra field to act as a primary key to maintain uniqueness for that row. As we thought that movie name and year do make themselves a great candidate for a composite key. Although we could have made them individual columns and have them as a composite key, but we decided that's not so much fun and went with this approach instead.

Triggers

To maintain our database as per the requirement of this project we are heavily relying on triggers. This table has 5 associated triggers to achieve various functionalities which I will be describing below:

<u>Prevent Cycle</u>

The most challenging part of this project was to have a system which prevents self-referencing movie cycles and we went with the recursive query function which we thought works best for this given scenario. We have a trigger name: *prevent_cycle* which triggers on every insertion and execute function of our recursive query name: *detect_cycle* which recursively runs to find any cycles by storing the newly inserted data into an array and traverse it to find if newly inserted movie and sub-ordinate data has any cycles within them.

<u>Movie To Subs</u>

This trigger maintains another table named *sub_ordinated_movies* table. It basically maintains parent to child relationship which we use later to delete child movies when parent movie gets deleted. More on that later.

<u>Prevent Update Subs</u>

This trigger executes whenever the column value of `*sub_ordinated_to*` is changed. Particularly, we allow null value to be set to some other sub-ordinated movie and change the existing sub-ordinated movie to another sub-ordinated movie. However, we don't allow null values once a value has been set in this field due to architecture of our database as it could cause runaway triggers for deletion.

Clear Movies Watched

This trigger simply removes entries from our movies watched table if any movie gets deleted from the movie_list table.

Clean Subs

This trigger helps maintain sub_ordinated_movies table by removing any deleted movie from the movie_list table.

`sub_ordinated_movies` Table

> In the movie_list table, we have a column name 'sub_ordinated_to' in this, every movie that has a parent gets stored. Meaning movie that has no parent will have that field empty(null) and the movies which have a parent movie will have their parent's name in there.
>
> The relationship we have there is only one way that is only child knows who parent is. But we figure, we needed an additional table to have a parent to child relationship to efficiently delete all the sub-ordinated movies whenever any parent movie gets deleted.

Triggers

> In the 'sub_ordinated_movies`, we have 2 triggers to achieve mostly one very important functionality. That is to delete the child movie if the parent gets deleted and to prevent any inputs in the two columns that doesn't have a presence in the movie_list table. Somewhat making it to follow a foreign key pattern.

Clear Sub

This trigger executes function `delete_child_movies` which deletes up all the child movies of the deleted parent present in the movie_list table and as well as in the `sub_ordinated_to` table as well.

So, it works like this:

Whenever a movie gets deleted from our main table *movie_list*, we trigger a function named *clear_from_sub_ordinated()* which deletes that movie from the *sub-ordinated_movies* table, now when this happens, trigger *Clear Subs* fires and deletes movies from both movie_list table and sub_ordinated_movie table where any movie has the same parent movie which got deleted from the movie_list, and this operation goes on and on until whole chain of movies gets deleted.

In short these are the activities which occurs to delete all the sub-ordinated movies in our database:

1. A Movie gets deleted from *movie_list* table
2. A trigger (clean_subs) fires and deletes the parent_movie from sub_ordinated_movies table
3. When any row gets deleted from sub_ordinated_movies, a trigger(clear_sub) fires and deletes all the child movie of the parent which got deleted from the *movie_list* table
4. These steps get repeated until whole chain of movies gets deleted.

Integrity check subs ordinate

To explore more about triggers, we decided not to have a foreign key relationship with the *sub_ordinated_movies* table so we went ahead and created another trigger which check if the inserted movie exists in the *movie_list* table

`movie_related_person ` Table

This table contains all the persons which can come under the field of *directed_by*, *written_by* and *leading_actor* field in the table *movie_list*

Triggers

Delete from movies by mrp

This table has only one trigger associated it with. This trigger gets execute whenever any *movie_related_person* gets deleted from this table and in turn deletes all those movies in which this person has existed in the fields mentioned above.

`movies_watched ` Table

This table contains all the ratings given by the user for a particular movie.

Triggers

Numbers Insert

This trigger updates movie_list table whenever any rating is given or updated by the user. It uses an avg function to calculate mean rating of a movie.

Edit Movies Watched

This trigger updates the movie_watched table whenever there is a conflict of duplicate values. For eg; If someone tries to insert a new rating for the same username and movie, it will just update the existing movie rating for that field.

`user_list` Table

This table contains the data of all users which can rate movies and in turn get movie suggestions.

Triggers

<u>Modify Movie Watched Username</u>

This table only has one trigger, which updates the username in the `movies_watched` table should the user changes it.

`genres` Table

This table has only one functionality, this table contains all the genres which a movie can be associated with.

`suggestion_table` Table

This table is a helper table to help provide suggestion tailored for the specific user. More on that later in the suggestions part.

SUGGESTIONS

For suggestions, we created on function named *get_suggestions(string)* which takes an input of username and curates the suggestion for the same. Here is how it works:

As per the requirement of this assignment, only liked movies should be taken into consideration for showing suggestions. So, to create suggestions, we only consider movies that has been watched by the user and has rating of more than 5 on the scale of 10 that we are using.

Suggestions Algorithm:

1. We fetch all the movies watched by the user where rating is greater than 5
2. From the fetched movies, we take out the genres and the people associated with the that movie namely director and leading actor. For our suggestions, we don't take writer into account.
3. We repeat these steps using a for loop to get all the favourable movies.
4. Now, we fetch movies from the table *movie_list* that matches the attributes of the liked movie
5. For suggestions, we have priority. First priority goes to the sub-ordinated movie if exists. The next priority goes to the leading actor or the director and the next one goes to the genres.

In short movies would be recommended in this order

1. If the movie has any sub-ordinates
2. If the movie has the same leading actor or director
3. If the movie has same genres as per user liked movies

These all suggestions then gets stored to the *suggestion table* to help us count and manage the suggestions. Now, if the suggestion count is less 5(arbitrary number set by us) then we run another loop, which shows the movie in addition to the above movies which has mean rating higher than 5.

How Data is Distributed?

Distribute database helps to make project as convenient by logical relation with the nodes in order to make the database more user friendly. In our database it was on a smaller level yet we learned a lot and tried to make data distributed.

Our System is based on the homogenous DDBMS, and we did the transparency like the location and naming transparency, so the user just gets the transparent data.

In our database we have used the "Complete Vertical Fragmentation" at most of the table. Using the data and connecting with other table such as primary key of *movie_list* in our database connected with *movies_watched* table. This helped a lot, to make the connections within the database with the different tables.

Our database is based on scalability providing the flexibility in both ways horizontally and vertically like when the user adds the new movie the movie can be added *movie_list* table easily moreover, multiple users can also be added, we are also creating a different table for the suggestions of the movie in *suggestion_table* which can be easily tailored.

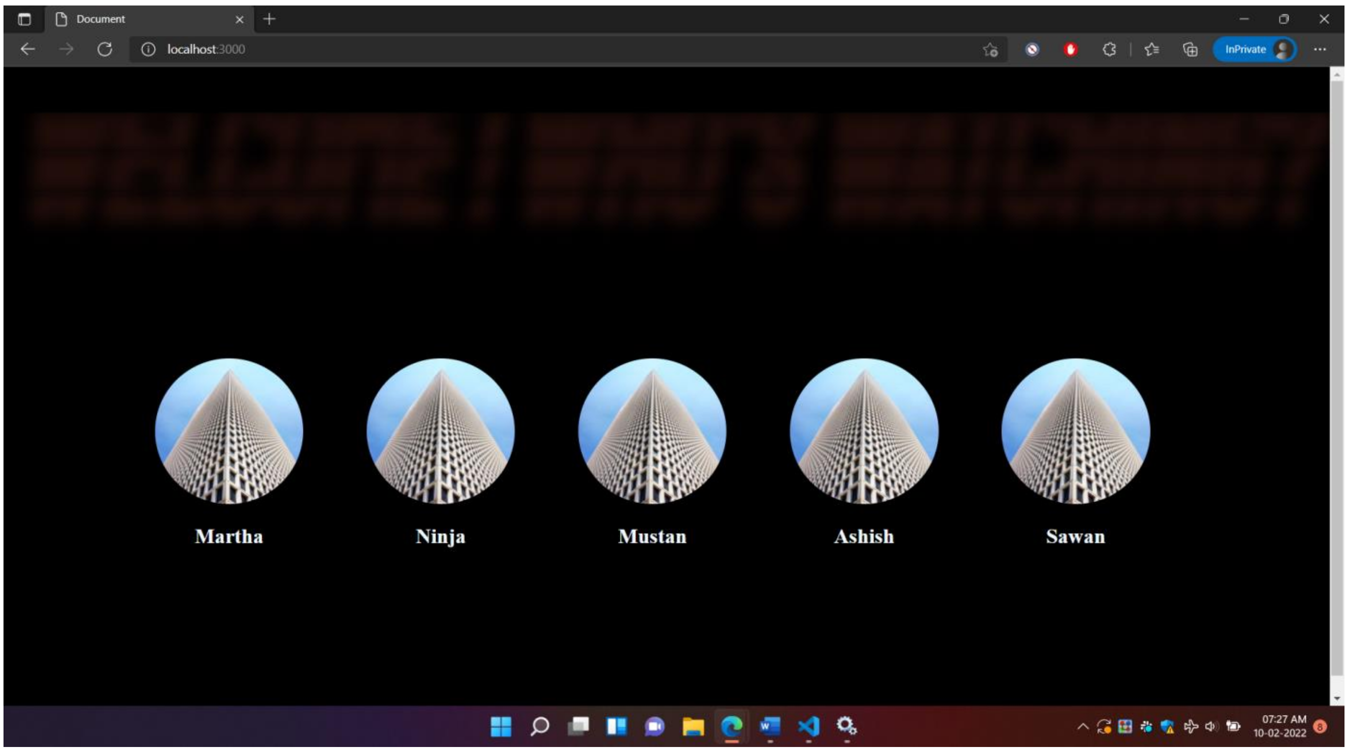
There are possible ways to make our data more distributed

Full Replication: - We could had used the full replication. The whole database could had been replicated at every node in the distributed system. Which can actually help in maximizing the data availability.

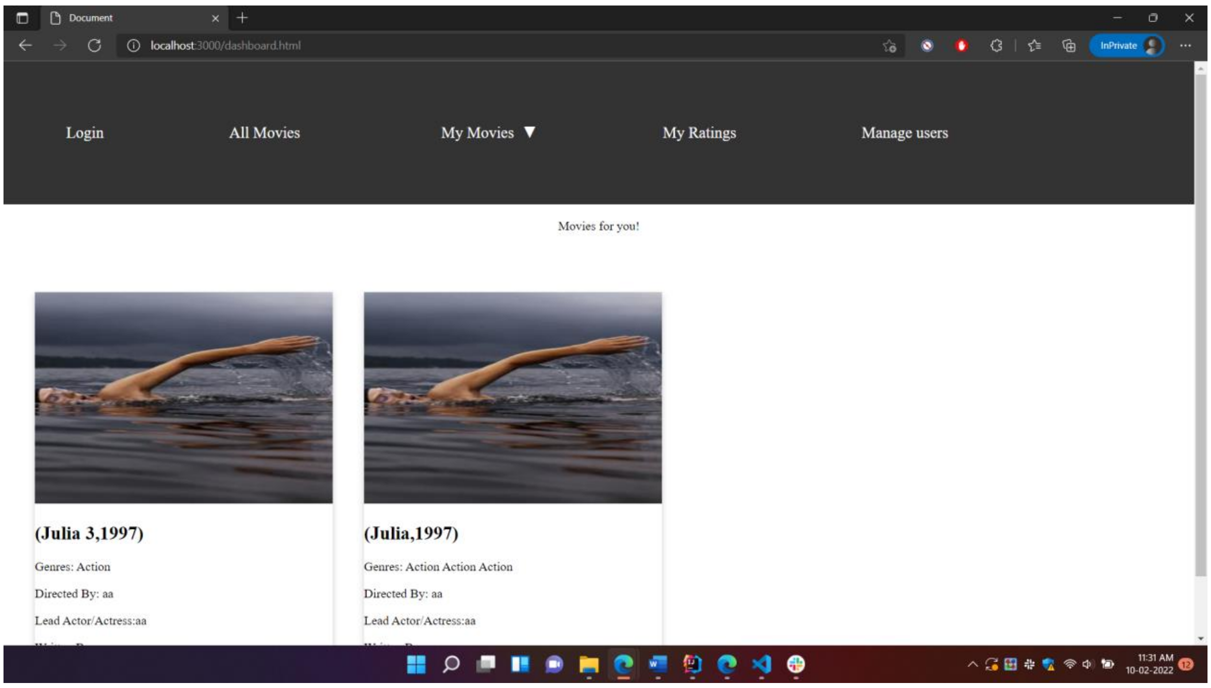
Like in our database most of the node and connection has been made with *movie_list* table, so if the connection somehow fails with from the *movie_list* table, so it can affect all the other tables, and there might be the possibility then the data might not be retrieved.

Frontend Overview

For the frontend we tried to things minimal as possible, after running the local host. The root URL goes to the login page where after clicking we set a cookie to track the user for suggestions and rating purposes.



After clicking any user, you would be on the home page where you will see suggestions with the ability to rate the movie within the suggestions.



Movies for you!

(Julia 3,1997)

Genres: Action

Directed By: aa

Lead Actor/Actress:aa

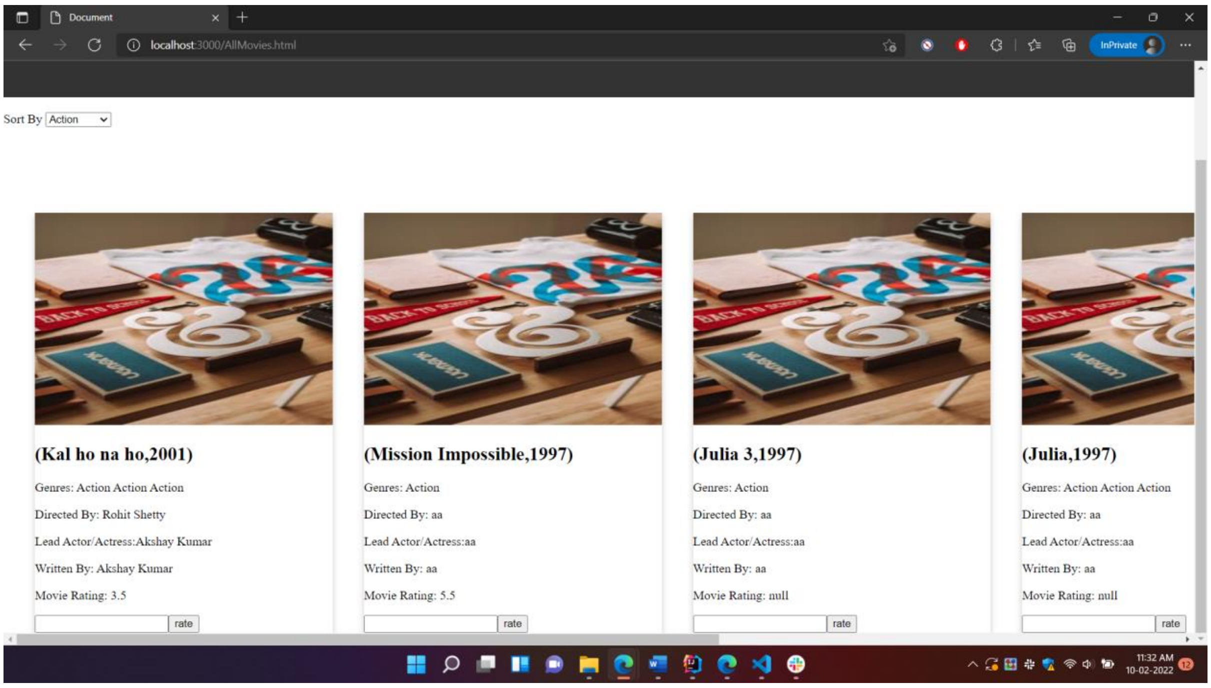
(Julia,1997)

Genres: Action Action Action

Directed By: aa

Lead Actor/Actress:aa

Clicking All Movies would go to the page where you can see all movies present in the database, and sort them by genre.



(Kal ho na ho,2001)

Genres: Action Action Action

Directed By: Rohit Shetty

Lead Actor/Actress:Akshay Kumar

Written By: Akshay Kumar

Movie Rating: 3.5

(Mission Impossible,1997)

Genres: Action

Directed By: aa

Lead Actor/Actress:aa

Written By: aa

Movie Rating: 5.5

(Julia 3,1997)

Genres: Action

Directed By: aa

Lead Actor/Actress:aa

Written By: aa

Movie Rating: null

(Julia,1997)

Genres: Action Action Action

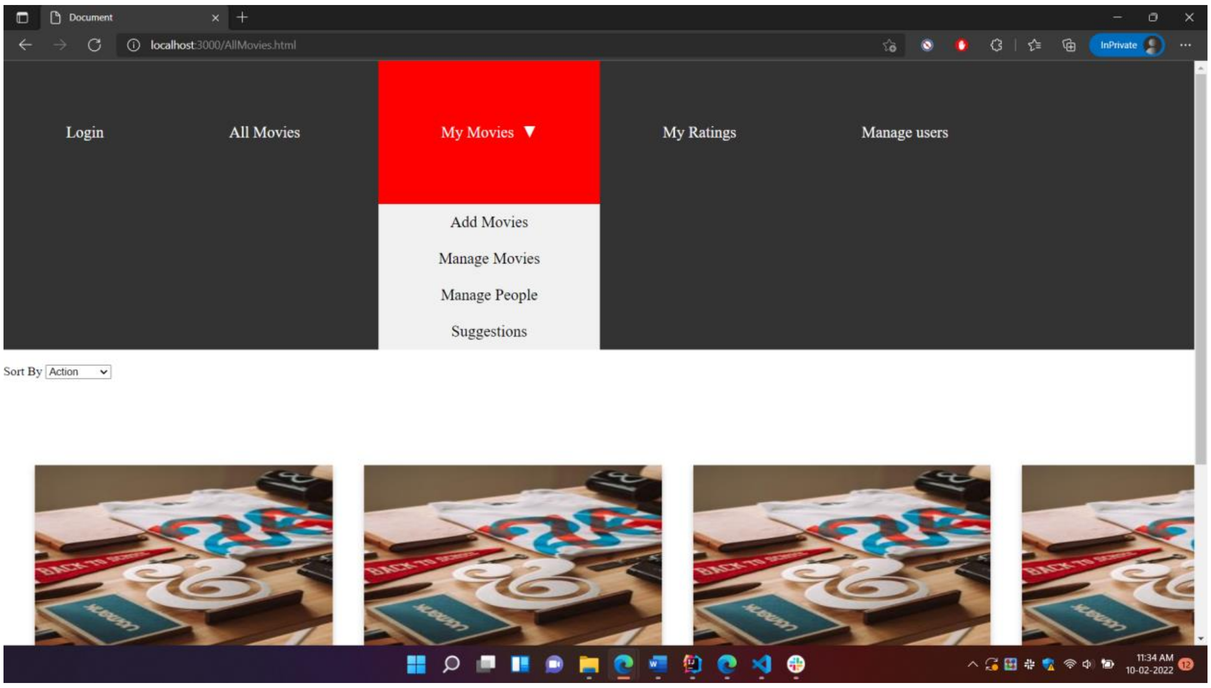
Directed By: aa

Lead Actor/Actress:aa

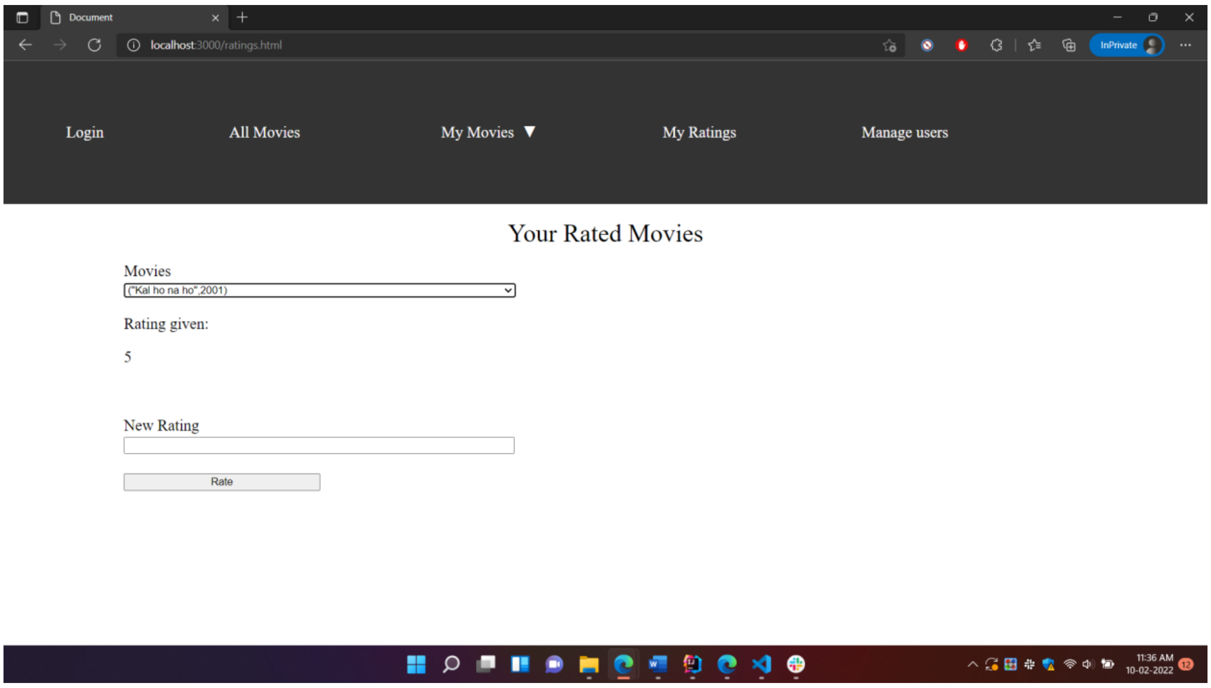
Written By: aa

Movie Rating: null

Hovering on My Movies on header will present more options such as ability to add movies, get to suggestions manage people and more



Subsequently, Clicking on My Rating will open a page where you would be able to see all the rated movie by the current logged in user.



Your Rated Movies

Movies
("Kal ho na ho",2001)

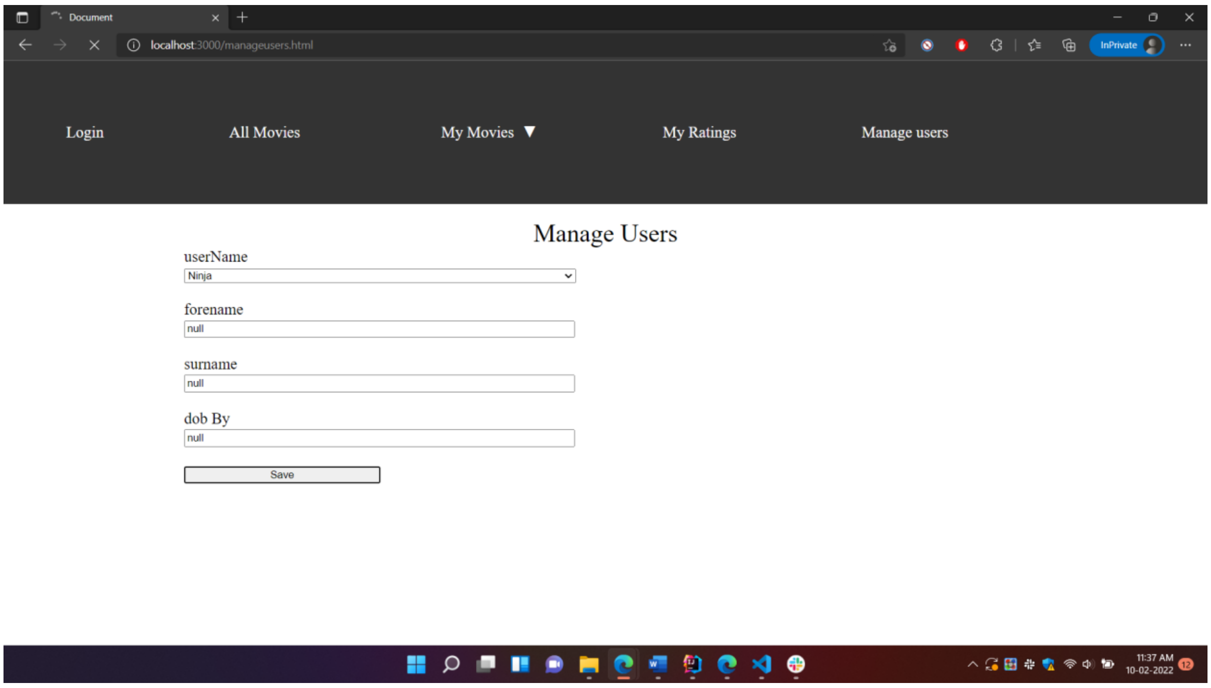
Rating given:

5

New Rating

Rate

And Clicking on Manage User will allow to change user name



Manage Users

userName
Ninja

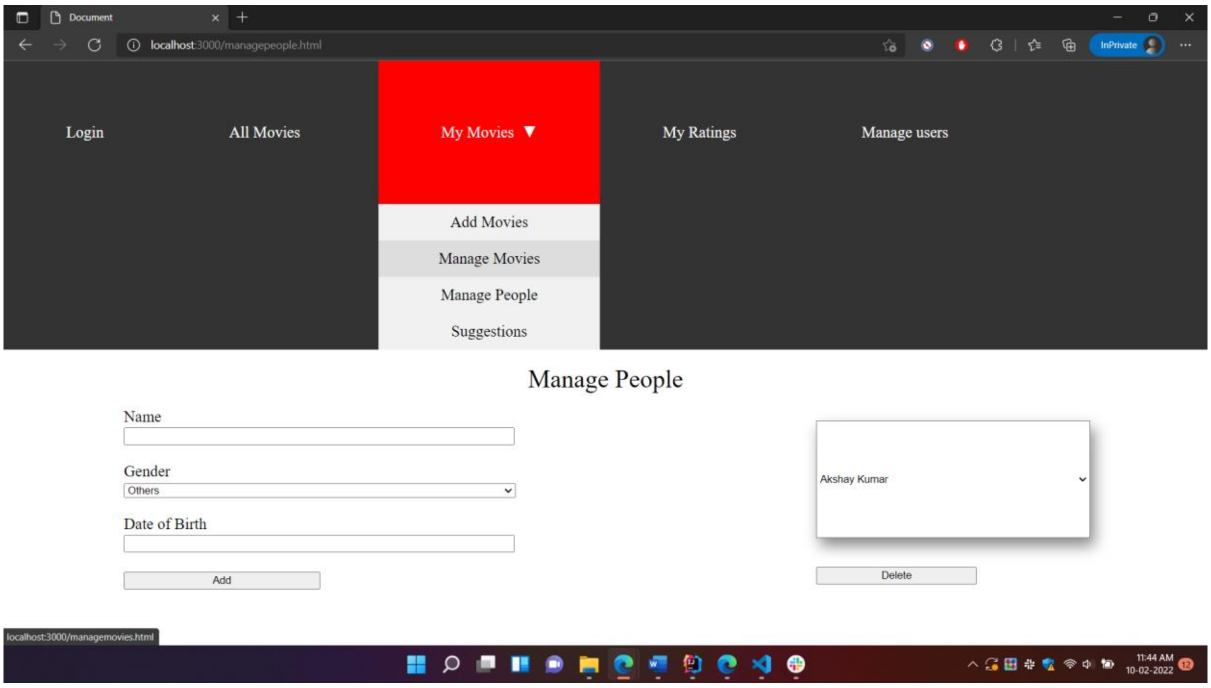
forename
null

surname
null

dob By
null

Save

Manage people allows you to add persons which can be included as director, writer and actor in a movie.



Manage People

Name

Gender
Others

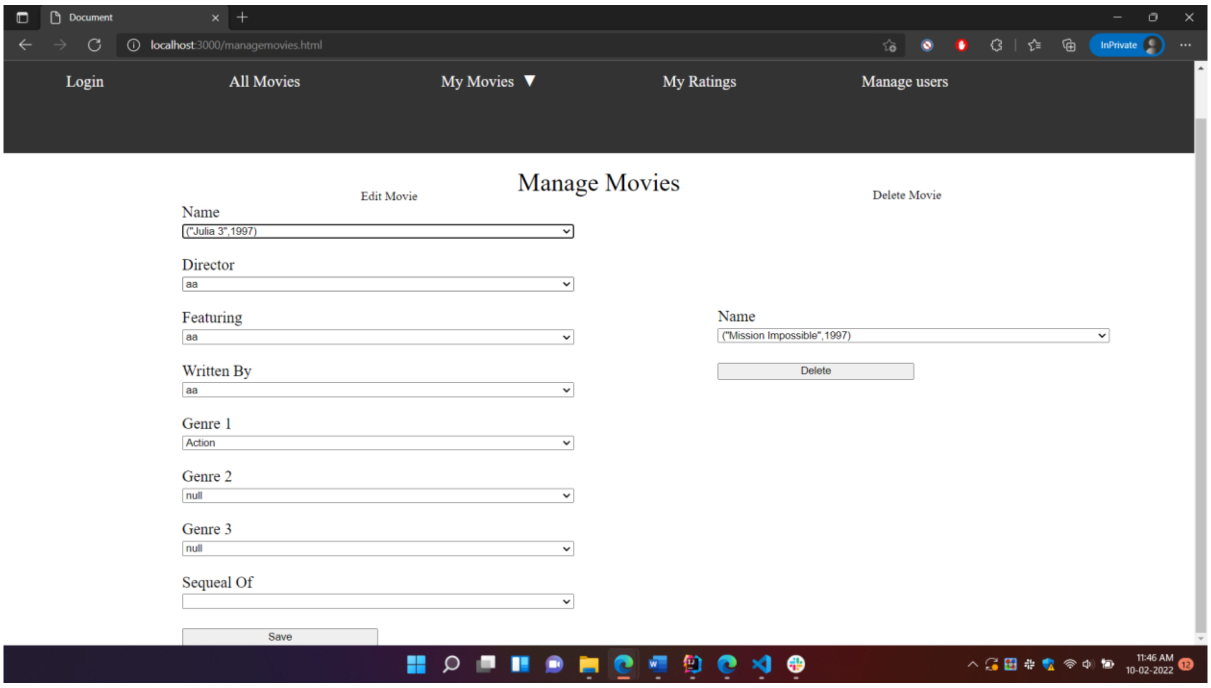
Date of Birth

Add

Akshay Kumar

Delete

In Manage Movie, you can edit attributes of a movie, such as genres, actors etc.



Manage Movies

Edit Movie

Name
("Julia 3",1997)

Director
aa

Featuring
aa

Written By
aa

Genre 1
Action

Genre 2
null

Genre 3
null

Sequeal Of

Save

Delete Movie

Name
("Mission Impossible",1997)

Delete

Technology Used Overview:

Backend:

- Nodejs

  For backend, we used Node.js to manage connection with the postgre db and to maintain connection with our frontend. We choose this framework due to not having any previous experience in this so it was a well learning experience for both of us.

- JavaScript

  To write actual code, and to have harmony with the front end we used js as our go to language for writing both frontend and backend

- AJAX

  For actual communication of data between front and backend we use ajax. Ajax allowed us to learn more about the current curriculum we have in our Web Engineering course and as well as it was fun and easy to work with.

Frontend:

- JavaScript
  To create elements dynamically inside our HTML body and to send and receive data from backend to front end we used JavaScript. Choosing JS was a very simple decision due to technologies involved in the backend.

- HTML
  Actual frontend is entirely written in vanilla HTML code without the usage of any framework whatsoever. We decided this because we thought it would be best to dip toes in HTML first rather than using any framework built on top of HTML for better visual.

- CSS
  For Styling our HTML content somewhat we used CSS as we wanted to learn more about CSS without having to involve any additional framework.

Bibliography

- **Referenced from the slides provided by the professor.**
- **www.postgrestutorial.com**
- W3Schools Online Web Tutorials
- **www.youtube.com**
- **www.quora.com**
- **www.stackoverflow.com**
- **www.tutorialspoint.com**