# Portfolio Project 1

Matthias Kaas-Mason
Isaac Parker Kelsall
Sebastian Rix
Neha Sharma

October 7, 2024

**Visit our GitHub Repository**

## Contents

# A Application design

The design of the movie application aims to provide an intuitive and effective user interface for searching, browsing, rating and managing title (movies, TV shows, etc.) and actor information. The solution supports multi-user functionality and provides features such as search history tracking, bookmarking and user ratings. The design has been influenced and inspired by existing websites making an approachable user experience.

The wireframes (Figures 1, 2, 3) represent the core pages of the application. These are designed to reflect our vision for the intended user experience for browsing and searching, as well as for viewing detailed information about specific movies, actors and directors.

We decided upon a common top-bar, which is the applications main search bar. This navigation and search bar is accessible from all pages, allowing users to search for titles and actors. This creates consistency across all web pages, and giving simple ability to navigate between the pages of our application. When the user clicks on this search bar, we are going to display their recent search history to them here as a scrollable drop down menu.

## A.1 Wireframe for display page for a movie/episode

Once a title is selected, users are directed to the Movie Display Page (Figure 1). This page shows basic details about the selected title such as the Title, Year, Runtime, Genre, Director, Writers, and Actors. The plot description is displayed to the right of the poster image. There is also options for users to rate the movie, add it to bookmarks and write a review. Reviews posted by other users are visible in a designated section.

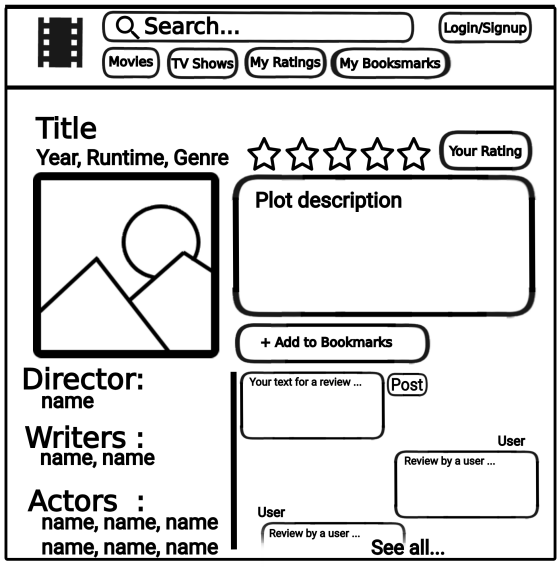


Figure 1: Wireframe for the display page showing a movie

## A.2 Wireframe for display page for a person

When viewing actors or directors, users are directed to the Person Display Page (Figure 2), which presents the actor's or director's name, roles, and known For titles in a listed text field. Users can bookmark the person and the application displays a list of co-actors who have frequently worked with the individual.

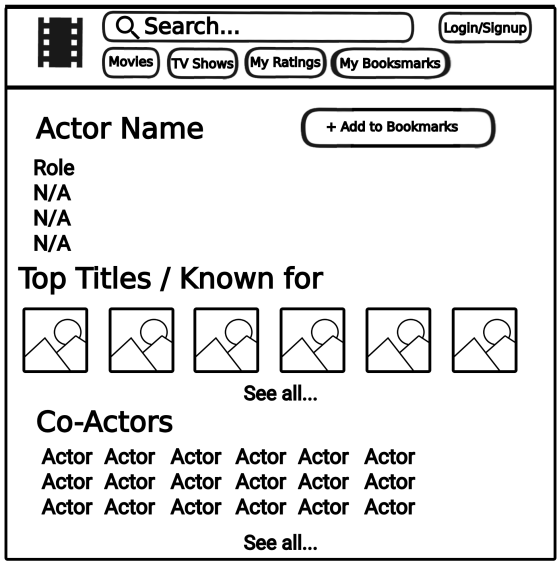


Figure 2: Wireframe for the display page showing a person

## A.3 Wireframe for search results page

When a user have entered a search query they are directed to the Search Results Page (Figure 3) where a set of filter options on the left allows users to narrow their searches by category, such as Movie, TV Show, Actor, or additional criteria. These options provide precision in the search by the users. Search results are listed and categorised into Titles and People sections, each providing a summary of relevant information.

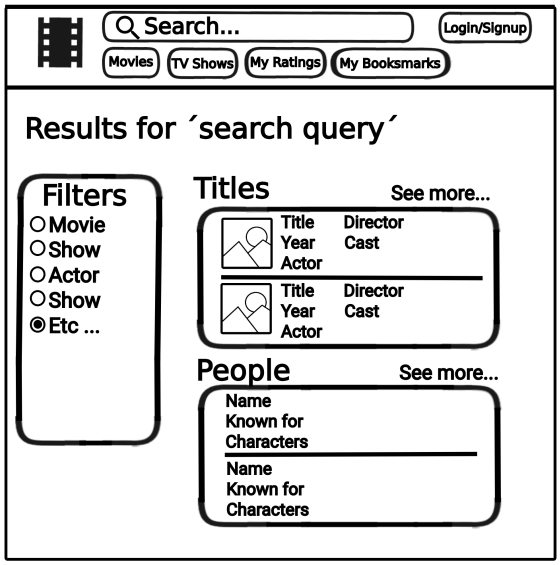


Figure 3: Wireframe showing the search page

These wireframes are made based on our initial mockups we made earlier in the project. Our initial mockups can be seen in the appendix at the end of this report.

# B The Movie Data Model

To begin with our new database, the initial observation we made was that the IMDb dataset contained a number of comma-separated values (such as **genres**, **languages**, and **known for titles**) which violated First Normal Form. To ensure normalization of this data, we created separate tables for each of these instances, making new tables such as **titleGenre** and **titleLanguage**. This is evident throughout our script for these tables, where we utilized firstly the `TRIM()` function to remove the whitespace that would now appear at values and secondly, the `UNNEST()` function using ',' as the delimiter to split the values up by commas. There was also one instance of **titleCharacters** where we had to remove '[' characters using the `REPLACE()` function so these characters would become empty strings. `NULLIF` was also used throughout our script to ensure any values that were originally an empty string, or 'N/A' would be streamlined to be NULL throughout our new dataset.

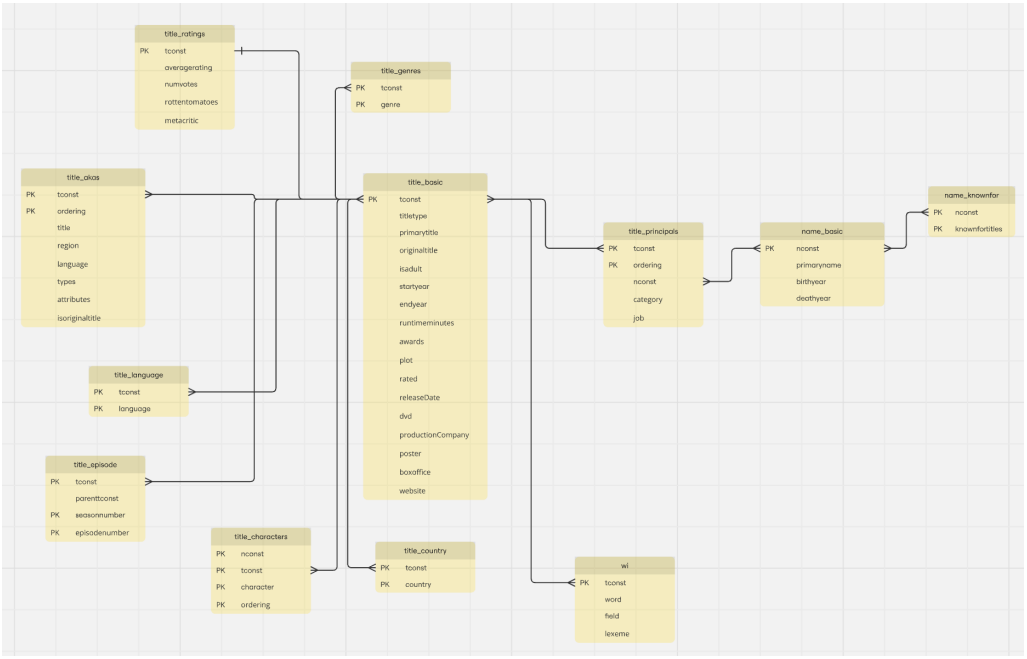


Figure 4: Final ER Diagram

Our next step was to see if there was any data that could potentially be removed as we believed it would be unnecessary moving forward. We considered removing the **titleAkas** table entirely since our website will likely only support English. However, we decided to retain it to support multilingual search queries. Keeping this table ensures that users who search for a title in different languages can still find the correct movie or TV show, improving the inclusivity and accessibility of our final application.

Initially, we considered removing the **job** column from the **titlePrincipals** table because much of its content overlaps with the **category** column (e.g., actor, director, etc.). However, upon further analysis, we realized that **job** can provide more descriptive information, such as specifying if someone is a stunt double, costume designer, or production assistant. Therefore, we decided to retain the **job** column for its potential to add richer detail to our dataset.

We noticed that the **title_crew** table that was originally in the IMDb dataset contained information about directors and writers, which was largely duplicated in the **title_principals** table under the **category** and **job** columns. To streamline our database and reduce redundancy, we decided to remove the **title_crew** table and rely solely on **titlePrincipals** to capture all relevant crew information.

We noticed a large amount of overlapping data between the IMDb **title_principals** table and the **primaryprofession** column in the IMDb **name_basic** table. At first, we split the **primaryprofession** column into a new table called **nameProfession** to eliminate the comma-separated values that made up this column. We then realized much of this information was redundant and had already been stored in our new **titlePrincipals** table, and ultimately made the decision to remove our newly made **nameProfession** table along with the **primaryprofession** column from

the original IMDb database.

We originally didn't include ordering in our **titleCharacters** table, but in the end, we had to include it due to primary key constraints and duplicates in the data if it wasn't included. This was helpful for us as it gave us a greater understanding of why the **ordering** column was functional and included in the IMDb dataset.

Now to merge the **omdb_data** table into our newly created models. We found that most of the information from **omdb** was a further expansion onto our **titlebasic** table, and this can also be seen in our new model. A few of the columns will have overlapping data, such as **title** and **genre**, and for these instances where **tconst** was equal in the IMDb and OMDB datasets, we decided to prioritize the IMDb data to maintain integrity. This meant that if the data field was already filled by the IMDb data, the OMDB data wouldn't be included unless the field was empty or null in the IMDb data. Along with this, we didn't want to add any new titles from the OMDB dataset, so if the **tconst** didn't already exist in the IMDb data, we chose not to include this new title. Again, this was to maintain consistency in our data and not create any gaps in our larger set.

Some data, such as **totalseasons** and **response**, were removed from the OMDB data entirely as we believed them to be redundant. To find the total seasons, we can just take the **max(seasonnumber)** from our already existing **titleEpisode** table, and the response is unnecessary in our static dataset. We also had a discussion relating to the **plot** and **posters** columns in the OMDB dataset, and whether the entire **tconst** should be removed if these values were null. The argument to remove was due to the webpage looking empty and unfinished if a user were to click on one of these titles. In the end, we decided this would result in losing too much data (approximately 50% of the dataset). Removing them could result in losing valuable titles, even if they lack these attributes. We also saw that missing data could potentially be added in the future as more information becomes available.

We had to go through a complex process to extract the **Metacritic** and **Rotten Tomatoes** scores from the **omdb_data** table. In the `WITH` clause, the **RatingsExtract** subquery parses the ratings JSON array, looking for values associated with the sources 'Rotten Tomatoes' and 'Metacritic'. It trims percentages from **Rotten Tomatoes** and takes the first value of the **Metacritic** score. The `UPDATE` statement then uses the extracted values from **RatingsExtract** to update the **rottenTomatoes** and **metaCritic** columns in the **titleRatings** table, ensuring that existing values are only updated if a new valid rating is found (`COALESCE()`).

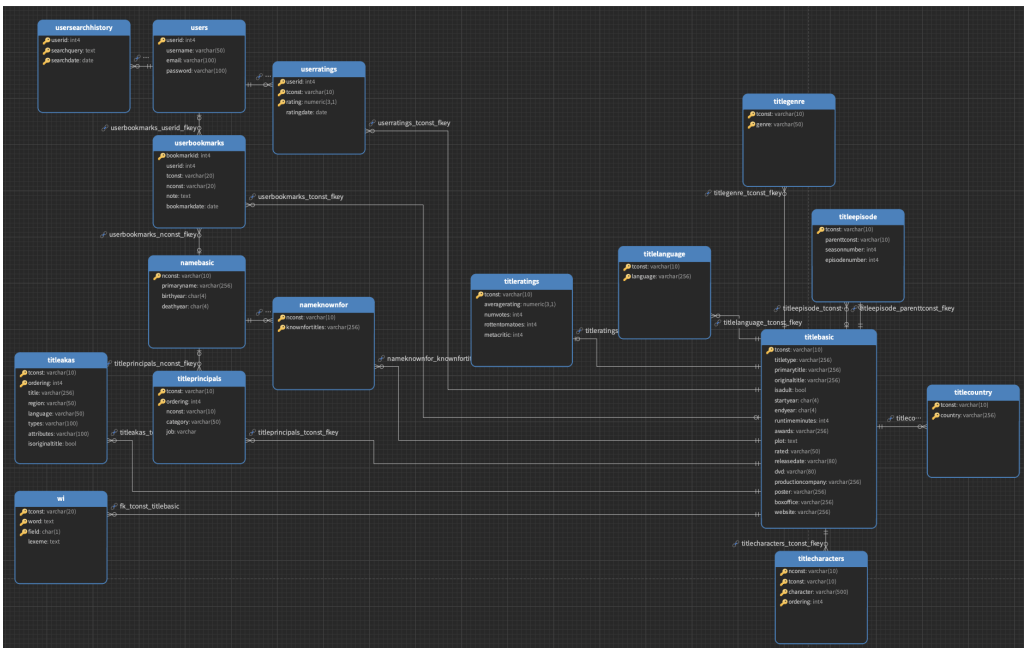


Figure 5: Reverse Engineered ER Diagram

The Reverse Engineered ER diagram shows several one-to-many relationships between tables (see Figure 5). At the center, **titleBasic** is connected to many tables such as **titleRatings**, **titleAkas**, **titleLanguage**, **titleGenre**, **titleEpisode**, **titleCountry**, and **titlePrincipals**, indicating that

each title can have multiple ratings, alternate titles, languages, genres, episodes, countries, and associated people.

**nameBasic** has a one-to-many relationship with **titlePrincipals** and **nameKnownFor**, showing that one person can be associated with multiple titles in different roles (such as actor or director) and can be known for multiple titles.

The **users** table connects with **userRatings**, **userSearchHistory**, and **userBookmarks** through one-to-many relationships, reflecting that a user can rate, search, and bookmark multiple titles. These relationships are facilitated by foreign keys, which ensure that data is consistently connected between entities.

# C The Framework Model

The User Framework Model incorporates all user functionalities including logging the search history, rating titles, and bookmarking. The framework helps efficiently store and retrieve data for each user interaction, and allows for a smooth connection into our larger movie database model. The key entities include Users, Search History, User Ratings, and Bookmarks. These entities work together to track user actions like searching, rating and bookmarking titles and names throughout our application.

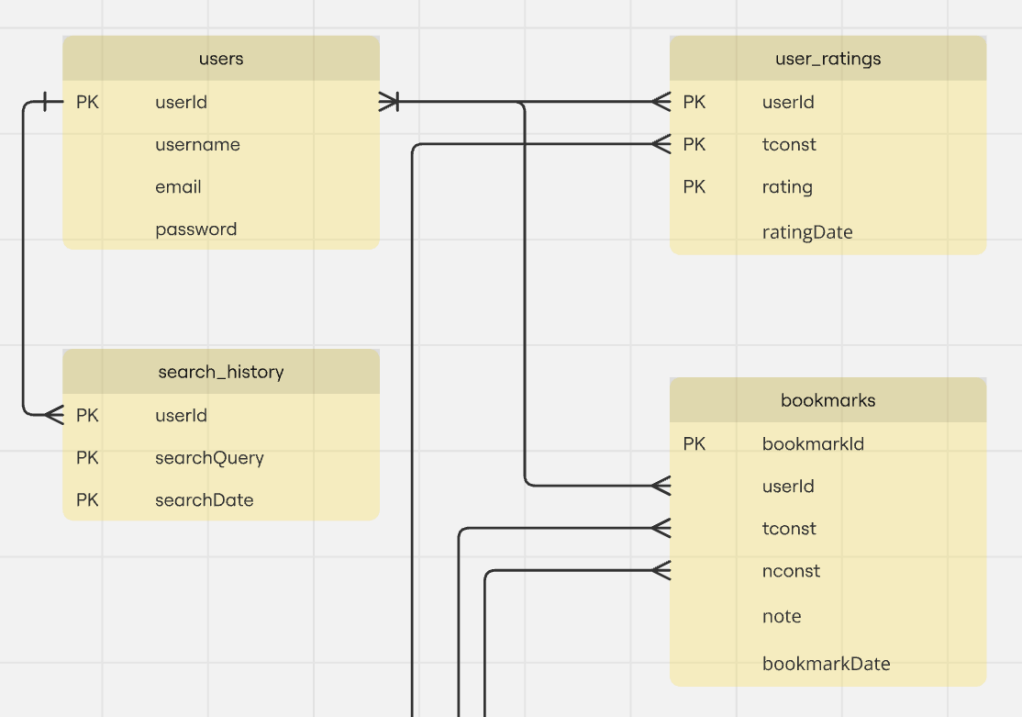


Figure 6: User Framework ER Diagram

Below is a breakdown of the entities involved, their relationships, and how they interact within the framework.

1. **Users Table**: This table stores essential account information for each user.
   - **userID** (Primary Key): Unique identifier for each user.
   - **username**: The display name chosen by the user.
   - **email**: The user's email address.
   - **password**: The user's password for authentication, securely stored as a hash.
2. **Search History Table**: Storing the user's search history. The three attributes together form a composite Primary Key.
   - **userID** (Foreign Key): Refers to the `userID` in the **Users** table.
   - **searchQuery**: The content or keyword searched by the user.

- **searchDate**: The date and time of the search.

3. **User Ratings Table**: Storing the user's ratings for either titles or names. The combination of `userID`, `tconst`, and `rating` form a composite Primary Key.
   - **userID** (Foreign Key): Refers to the `userID` in the **Users** table.
   - **tconst**: The unique title identifier, referencing the **titleBasic** table.
   - **rating**: The user's rating for the content, between 1 and 10.
   - **ratingDate**: The date when the rating was provided.
4. **Bookmarks Table**: This table tracks the titles and names users have bookmarked.
   - **bookmarkID** (Primary Key): Unique identifier for each bookmark entry.
   - **userID** (Foreign Key): Refers to the `userID` in the **Users** table.
   - **tconst**: References the **titleBasic** table.
   - **nconst**: References the **nameBasic** table.
   - **note**: An optional note that the user can add to the bookmark.
   - **bookmarkDate**: The date when the bookmark was created.

## Relationships Between Tables

Users have a one-to-many relationship with **Search History**, **User Ratings**, and **Bookmarks**. Each `userID` in the **Users** table can be linked to multiple records in the other tables, representing various actions by the same user.

The **User Ratings** table has a reference to our **titleBasic** table through the `tconst`. The **Bookmarks** table has an optional reference to either the **nameBasic** or **titleBasic** table via `nconst` or `tconst` respectively, depending on whether the user is bookmarking a name or title. This structure allows us to track user actions across the system.

Please refer to Figure 5 for the Reverse Engineered ER Diagram of our entire model, and for a visual representation of how the **User Framework Model** interacts with the movie database.

# D Functionality and Testing

## D.1 Basic framework functionality

The basic framework functionality supports user management, bookmarking and storing search history. The **add_user** function, we choose to use a straightforward design that inserts a new user with a username and email into the users table. An alternative approach could have involved more sophisticated user validation, such as checking for existing emails to avoid duplicates or adding more user information such as timestamps for account creation.

The **add_user** function allows insertion of a new user into the users table. The function takes two input parameters: **p_username** (the username of the new user) and e_email (the email address of the new user). It executes an **INSERT INTO** statement that adds the new user into the **users** table

This function is essential for user registration within the system and ensures that every user has both a username and an email associated with their profile.

Listing 1: PostgreSQL Function for User Management

```
CREATE OR REPLACE FUNCTION "public"."add_user"("p_username" varchar, "p_email" varchar)
  RETURNS "pg_catalog"."void" AS $BODY$
BEGIN
    INSERT INTO users (username, email)
    VALUES (p_username, p_email);
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

For the bookmarking functionality we created **bookmark_movie** and **bookmark_name**. These allow the user to bookmark titles and names respectively. We chose this separation for clarity and future flexibility, because it allows for more precise control over how bookmarks for movies and names are handled. An alternative could have been a single generic bookmark function that distinguishes between names and titles internally in the function, but separating them made the code easier to maintain and expand. This choice also maintain a clear overview and prioritise simplicity and functionality over complexity.

The **bookmark_movie** function or **bookmark_name** function bookmarks either a specific movie or person by inserting the **tconst** (the unique identifier for the movie or series) or **nconst** (the unique identifier for a person) into the **userbookmarks** table. The functions takes two input parameters: **p_userid** (the ID of the user bookmarking) and **p_tconst**/**p_nconst** and inserts a new row into the userbookmarks table. This associates the user with a new bookmark. The script checks if the user already bookmarked a given name or title, to make sure there is no duplicate bookmarks. It is noteworthy to mention that the function wont work if there are no users created in the database. Check **add_user** function for adding users to the database.

These functions are essential for tracking which movies or persons a user have bookmarked for future reference.

Listing 2: PostgreSQL Function for Bookmarking Movies

```
CREATE OR REPLACE FUNCTION "public"."bookmark_movie"("p_userid" int4, "p_tconst"
    varchar)
  RETURNS "pg_catalog"."void" AS
$BODY$
BEGIN
    -- Check if the bookmark already exists
    IF NOT EXISTS (
        SELECT 1
        FROM userbookmarks
        WHERE userid = p_userid AND tconst = p_tconst
    ) THEN
        -- If not exists, insert the new bookmark
        INSERT INTO userbookmarks (userid, tconst)
        VALUES (p_userid, p_tconst);
    END IF;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

Listing 3: PostgreSQL Function for Bookmarking Names

```
CREATE OR REPLACE FUNCTION "public"."bookmark_name"("p_userid" int4, "p_nconst" varchar)
  RETURNS "pg_catalog"."void" AS
$BODY$
BEGIN
    -- Check if the bookmark for the name already exists
    IF NOT EXISTS (
        SELECT 1
        FROM userbookmarks
        WHERE userid = p_userid AND nconst = p_nconst
    ) THEN
        -- If not exists, insert the new bookmark
        INSERT INTO userbookmarks (userid, nconst)
        VALUES (p_userid, p_nconst);
    END IF;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

The **get_bookmarks** function allows a user to retrieve their saved bookmarks. This includes both movies/series (**tconst**) and actors/directors (**nconst**). The function takes the user ID (**p_userid**) as an input parameter and returns a list of bookmarks. The return values includes the type of bookmark (either a "title" for movies/series or "name" for actors/directors), the corresponding ID (**tconst** or **nconst**) and the title or name associated with that bookmark.

The function uses a **UNION ALL** to combine results from both types of bookmarks (titles and names) and makes sure that the results are cast correctly to **VARCHAR**.

The function **get_bookmarks** provides an important feature within the framework. It's enables users to keep tracking of content they are interested in.

Listing 4: PostgreSQL Function for Retrieving User Bookmarks

```
CREATE OR REPLACE FUNCTION "public"."get_bookmarks"("p_userid" int4)
RETURNS TABLE(bookmark_type VARCHAR, id VARCHAR, name_or_title VARCHAR) AS $$
BEGIN
    RETURN QUERY
    -- Fetch bookmarked movies
    SELECT
        'title'::VARCHAR AS bookmark_type,  -- Indicate this is a title bookmark, cast to VARCHAR
        b.tconst::VARCHAR AS id,            -- Return the tconst (movie/series ID), cast to VARCHAR
        tb.primarytitle::VARCHAR AS name_or_title  -- Return the movie/series title, cast to VARCHAR
    FROM
        userbookmarks b
    JOIN
        titlebasic tb ON b.tconst = tb.tconst
    WHERE
        b.userid = p_userid

    UNION ALL

    -- Fetch bookmarked actors/directors
    SELECT
        'name'::VARCHAR AS bookmark_type,   -- Indicate this is a name bookmark, cast to VARCHAR
        b.nconst::VARCHAR AS id,            -- Return the nconst (name ID), cast to VARCHAR
        nb.primaryname::VARCHAR AS name_or_title  -- Return the person's name, cast to VARCHAR
    FROM
        userbookmarks b
    JOIN
        namebasic nb ON b.nconst = nb.nconst
    WHERE
        b.userid = p_userid;
END;
$$ LANGUAGE plpgsql VOLATILE
COST 100
ROWS 1000;
```

## D.2 Simple search

The **string_search** function enables users to perform simple text searches across titles. This function, given a search string, returns titles that match the string, and logs the search in the user's history by calling the **update_search_history** function. The choice of using the **ILIKE** operator for case-insensitive matching ensures flexibility for users, allowing partial matches. We also used the **ORDER BY** statement to make sure that the movies with titles that matched the users query would come on top.

Updating search history as a side effect ensures that the solution consistently tracks user behaviour, allowing a search history. We also implemented a **string_search** function that do not call upon **update_search_history** function. This is intended to make searching capabilities for a possible scenario where a user utilise the application without being singed into a profile. Alternatively we could have made these two **string_search** functions into one, but a clear distinction and separation ensures simplicity, easy readability and keeping a clear design for future scaling.

Listing 5: PostgreSQL Function for Simple Search in Titles

```
CREATE OR REPLACE FUNCTION "public"."string_search"("p_search_string" varchar)
  RETURNS TABLE("tconst" varchar, "title" varchar) AS $BODY$
BEGIN
    RETURN QUERY
    SELECT tb.tconst::VARCHAR, tb.primarytitle::VARCHAR
    FROM titlebasic tb
    WHERE tb.primarytitle ILIKE '%' || p_search_string || '%'
       OR tb.tconst IN (
           SELECT tb.tconst
           FROM titlebasic tb
           WHERE tb.plot ILIKE '%' || p_search_string || '%'
       )
    ORDER BY
       -- Exact matches first
       CASE
           WHEN tb.primarytitle ILIKE p_search_string THEN 1
           ELSE 2
       END,
       -- Then sort by closest partial matches
       tb.primarytitle;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000
```

Listing 6: PostgreSQL Function for Simple Search with User Logging

```
CREATE OR REPLACE FUNCTION "public"."string_search"("p_search_string" varchar,
    "p_userid" int4)
  RETURNS TABLE("tconst" varchar, "title" varchar) AS $BODY$
BEGIN
    -- Log the search history for the user
    PERFORM update_search_history(p_userid, p_search_string);

    -- Perform the search with prioritization of exact matches
    RETURN QUERY
    SELECT tb.tconst::VARCHAR, tb.primarytitle::VARCHAR
    FROM titlebasic tb
    WHERE tb.primarytitle ILIKE '%' || p_search_string || '%'
       OR tb.tconst IN (
           SELECT tb.tconst
           FROM titlebasic tb
           WHERE tb.plot ILIKE '%' || p_search_string || '%'
       )
    ORDER BY
       -- Exact matches first
       CASE
           WHEN tb.primarytitle ILIKE p_search_string THEN 1
           ELSE 2
       END,
       -- Then sort by closest partial matches
       tb.primarytitle;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
```

```
  COST 100
  ROWS 1000
```

The **update_search_history** function is resposible for logging a user's search queries into the **userSearchHistory** table. This function maintain a record of the searches performed by the users. The information stored is **p_userid** (the ID of the user performing the search), **p_search_string** (the search query input by the user) and the current date automatically captured with the statement **CURRENT_DATE**. The function inserts these values into the **userSearchHistory** table. Storing the current date are unnecessary, but we thought it could be interesting and possibly what would be done in a real-world scenario. Especially considering that the date stored, could be used later for user analytics, where time and date would matter.

Listing 7: PostgreSQL Function for Updating User Search History

```
CREATE OR REPLACE FUNCTION "public"."update_search_history"("p_userid" int4,
    "p_search_string" varchar)
  RETURNS "pg_catalog"."void" AS
$BODY$
BEGIN
    -- Check if the search query already exists for the user
    IF NOT EXISTS (
        SELECT 1
        FROM userSearchHistory
        WHERE userId = p_userid AND searchQuery = p_search_string
    ) THEN
        -- If not exists, insert the new search query
        INSERT INTO userSearchHistory (userId, searchQuery, searchDate)
        VALUES (p_userid, p_search_string, CURRENT_DATE);
    END IF;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

## D.3 Title rating

One of the very important functionalities for our application to have is the **rate** procedure which allows a user to give a rating to a title. This procedure has a lot it has to do more than just getting that rating. We first take the title, rating and userId as an input and check if the rating is a valid input (integer from 1-10), if this passes then we collect old rating data for that title for future calculations. We want to then check if the user has already places a rating on the title to know if we should update their past rating or create a new one.

If we need to update a rating we first find that row in the table userratings, save the old value and update the rating and rating date. Then we need to calculate a new rating for this title, in this case the number of total votes is not changing since we are just editing a rating. We do this by multiplying the average rating by the number of votes, subtract the users old rating and add the new one, then divide by the number of votes.

If we need to create a new rating we first add this to the userratings table with the current date and then update the titles average rating. This update changes the number of total votes by 1 so we need to update both ratingaverage and numvotes. We do a similar calculation to above but we account for the extra vote when dividing by the numvotes. After these calculations are done we need to also update the name rating of any nconst in that title. We do a calculation almost identical to the one discussed further in the name rating subsection but only on nconsts that were in the rated title instead of all nconsts.

Listing 8: PostgreSQL Procedure for Rating titles

```
CREATE OR
REPLACE PROCEDURE rate (titleId VARCHAR (10),_rating INT4,_userId INT4) LANGUAGE
    plpgsql AS $$ DECLARE oldRating INT; oldnumvotes INT; oldaveragerating NUMERIC
    (3,1); BEGIN IF _rating BETWEEN 1 AND 10 THEN
SELECT numvotes FROM titleratings tr WHERE tr.tconst=titleId INTO oldnumvotes;
SELECT averagerating FROM titleratings tr WHERE tr.tconst=titleId INTO
    oldaveragerating; IF EXISTS (
SELECT 1 FROM userratings ur WHERE ur.tconst=titleId AND ur.userid=_userId)
    THEN--update existing rating
--update userratings (get old value)
```

```
SELECT rating FROM userratings ur WHERE ur.tconst=titleId AND ur.userid=_userId INTO
    oldRating;
UPDATE userratings
SET rating=_rating,ratingdate=CURRENT_DATE WHERE userid=_userId AND
    tconst=titleId;--update titleratings (needs old value)
UPDATE titleratings
SET averagerating=(COALESCE (oldnumvotes,0)*COALESCE
    (oldaveragerating,0)-oldRating+_rating)/COALESCE (oldnumvotes,1) WHERE
    tconst=titleId; ELSE--new rating
--insert into userratings
INSERT INTO userratings (userid,tconst,rating,ratingdate) VALUES
    (_userId,titleId,_rating,CURRENT_DATE);--update titleratings
UPDATE titleratings
SET numvotes=numvotes+1,averagerating=(COALESCE (oldnumvotes,0)*COALESCE
    (oldaveragerating,0)+_rating)/(COALESCE (oldnumvotes)+1) WHERE tconst=titleId; END
    IF;--update nrating in namebasic
UPDATE namebasic
SET nrating=(
SELECT round(SUM (tr.averagerating*tr.numvotes)/SUM (tr.numvotes),1) FROM
    titleprincipals tp INNER JOIN titleratings tr ON tp.tconst=tr.tconst WHERE
    namebasic.nconst=tp.nconst) WHERE namebasic.nconst IN (
SELECT nconst FROM titleprincipals WHERE tconst=titleId); ELSE raise notice 'Input
    value must be between 1 and 10'; END IF; END;
$$;
```

## D.4 Structured string search

The **structured_string_search** function allows for complex search queries across multiple fields. This includes title, plot, characters, and names. This function takes four input parameters: **p_title**, **p_plot**, **p_characters**, and **p_names**. It returns matching results based on any combination of these fields. The function is essential for performing structured searches based on different aspects of a movie's metadata.

The function performs a series of **JOIN**s between tables: **titlebasic**, **titleprincipals**, and **namebasic** to filter and return only the titles that match the provided search criteria. We decided to use the **ILIKE** operator that is case-insensitive to offer users greater flexibility while performing queries.

Listing 9: PostgreSQL Function for Structured String Search

```
CREATE OR REPLACE FUNCTION "public"."structured_string_search"("p_title" varchar,
    "p_plot" varchar, "p_characters" varchar, "p_names" varchar)
  RETURNS TABLE("tconst" text, "title" text) AS $BODY$
BEGIN
    RETURN QUERY
    SELECT tb.tconst::TEXT, tb.primarytitle::TEXT
    FROM titlebasic tb
    JOIN titlecharacters tp ON tb.tconst = tp.tconst
    JOIN namebasic nb ON tp.nconst = nb.nconst
    WHERE tb.primarytitle ILIKE '%' || p_title || '%'
      AND tb.plot ILIKE '%' || p_plot || '%'
      AND tp.character ILIKE '%' || p_characters || '%'
      AND nb.primaryname ILIKE '%' || p_names || '%';
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000
```

## D.5 Finding names

The **search_names_by_text** function will take an actor or actresses name as input from the user and compare this query to primary names stored within our **nameBasic** table. The returned results will be **primaryName**s that are similar to the search value, utilising the `ILIKE` SQL function.

Listing 10: PostgreSQL Function for Finding Names

```
    CREATE OR REPLACE FUNCTION search_names_by_text(search_text VARCHAR)
RETURNS TABLE (
    nconst VARCHAR(20),
```

```
    primaryName VARCHAR(255),
    birthYear CHAR(4),
    deathYear CHAR(4)
)
AS $$
BEGIN
    RETURN QUERY
    SELECT
        nb.nconst,
        nb.primaryName,
        nb.birthYear,
        nb.deathYear
    FROM
        nameBasic nb
    WHERE
        nb.primaryName ILIKE '%' || search_text || '%';  -- Case-insensitive search
END;
$$ LANGUAGE plpgsql;
```

## D.6 Finding co-players

The **coPlayers** function is used to return a list of actors that star in titles with an inputted actor, useful for the persons page on the website to see related actors. This is done by counting how many times any other nconst has been in titles that the tested actor as been in. The resulting list is sorted by the count in descending order to give most frequent co-players. We also perform a filter on the "category" of the resulting nconsts to make sure they are an actor or actress thereby removing any directors, producers or other crew members.

Listing 11: PostgreSQL Function for CoPlayers

```
CREATE OR REPLACE FUNCTION coPlayers (testId VARCHAR (10)) RETURNS TABLE (nconst
    VARCHAR (10),primaryname VARCHAR (256),frequency BIGINT) LANGUAGE plpgsql AS $$
    BEGIN RETURN query
SELECT tp.nconst,nb.primaryname,COUNT (tp.tconst) AS freq FROM titleprincipals tp JOIN
    namebasic nb ON tp.nconst=nb.nconst WHERE tp.tconst IN (SELECT tconst FROM
    titleprincipals WHERE titleprincipals.nconst=testId) AND (tp.category='actor' OR
    tp.category='actress') AND nb.nconst !=testId GROUP BY tp.nconst,nb.primaryname
    ORDER BY freq DESC; END;
$$;
```

## D.7 Name rating

We need to add a name rating and chose to do this on all names instead of just actors, this is calculated by taking each title they have been in and getting a weighted average of their ratings using the number of votes as a weighting. We first create a new column in the namebasic table if it does not already exists that contains a rating value. Indexing is done here to speed up the calculations and will be discussed further in section E. We then set the nRating of each column by calculating the ratings multiplied by the number of votes summed for each of the movies they are in and devide by the sum of the number of votes to get a new average rating.

Listing 12: PostgreSQL for creating Name Rating value

```
ALTER TABLE namebasic ADD IF NOT EXISTS nRating NUMERIC (5,1);
CREATE INDEX IF NOT EXISTS index_tp_nconst ON titleprincipals (nconst);
UPDATE namebasic
SET nRating=(
SELECT round(SUM (tr.averagerating*tr.numvotes)/SUM (tr.numvotes),1) FROM
    titleprincipals tp INNER JOIN titleratings tr ON tp.tconst=tr.tconst WHERE
    namebasic.nconst=tp.nconst);
```

## D.8 Popular actors

There are a few popular actor functions and the most useful one is **ratingActors** which is used to list all actors that star in a specific title by their nRating that was calculated above. This is how we plan to display actors on our title page on the website. We first find all the nconsts that have

stared in the title we are interested in and filter these based on if they are an actor or actress, there is another function **ratingCrew** in the full SQL identical to this with filtering out actors/actress. We then simply join this table to namebasic such that we can use the nRating to sort in descending order. We have another popular actor function in the full SQL called **ratingCoPlayers** that is the same as **coPlayers** but returns them sorted by nRating instead of frequency.

Listing 13: PostgreSQL Function for Popular Actors

```
CREATE OR REPLACE FUNCTION ratingActors (testId VARCHAR (10)) RETURNS TABLE (nconst
    VARCHAR (10),nRating NUMERIC (5,1)) LANGUAGE plpgsql AS $$ BEGIN RETURN query
    SELECT nb.nconst,nb.nRating FROM (SELECT tp.nconst FROM titleprincipals tp WHERE
    tp.tconst=testId AND (category='actor' OR category='actress')) NATURAL JOIN
    namebasic nb ORDER BY nb.nRating DESC; END;
$$;
```

## D.9 Similar movies

We have a function **similarMovies** which returns a list of 10 titles that are similar to the inputted titleId. Returning similar titles has a lot of freedom in how this can be done but we chose to use language and genre as our metrics of similarity. We also have not discussed if we want to limit this to only movies or return all types of titles, it could also return only titles that have the same type (tvEpisode, Movie, Short, ...). We currently only check if the language and genre matches at least one of those respectively for the inputted title but an improvement to this function if more time was available would be to count the number of genres and languages it matches and sort initially by the count of genres and then languages. We then finally order these results by the number of ratings the title has as a proxy for popularity as we find that this usually gives better results.

Listing 14: PostgreSQL Function for Similar Movies

```
CREATE OR REPLACE FUNCTION similarMovies (testId VARCHAR (10)) RETURNS TABLE (tconst
    VARCHAR (10),primarytitle VARCHAR (256),numvotes INT4) LANGUAGE plpgsql AS $$ BEGIN
    RETURN query SELECT DISTINCT tb.tconst,tb.primarytitle,tr.numvotes FROM titlebasic
    tb NATURAL JOIN titlegenre tg NATURAL JOIN titlelanguage tl NATURAL JOIN
    titleratings tr WHERE tg.genre IN (SELECT tg.genre FROM titlegenre tg WHERE
    tg.tconst=testId) AND tl.LANGUAGE IN (SELECT LANGUAGE FROM titlelanguage tl WHERE
    tl.tconst=testId) ORDER BY tr.numvotes DESC LIMIT 10; END;
$$;
```

## D.10 Frequent person words

The **person_words** function allows the user to search for an actor or actresses name, and the response will be a list of words that are commonly related to the input name. We decided to also include the category that the word comes from purely for testing (ensure all categories are included) and believing it could be useful in the future, if there was a case we only wanted to filter by character or plot. There was some casting required due to differences in types. Please see comments within the code for explanations of each step. Tests included running simple queries such as:

SELECT * FROM **person_words**('Will Smith');

SELECT * FROM **person_words**('Nicole Kidman', 5);

In the case of Nicole Kidman, I added a 5, which will return a list of only the top 5. In the case of Will Smith, the default of top 10 will be returned.

Listing 15: PostgreSQL Function for Person Words

```
CREATE OR REPLACE FUNCTION person_words(
    p_primaryname VARCHAR,      -- The name of the person we're interested in
    p_limit INTEGER DEFAULT 10 -- Limit on the number of words returned (default 10)
)
RETURNS TABLE (
    word VARCHAR,               -- The word associated with the person
    frequency INTEGER,          -- Frequency of the word in titles the person is
    involved in
    category VARCHAR            -- The field category (t = title, p = plot, c =
    characters)
) AS $$
```

```
BEGIN
    RETURN QUERY
    WITH PersonTitles AS (
        -- Step 1: Retrieve all titles (tconst) the person is associated with from the titlePrincipals table
        SELECT DISTINCT tp.tconst
        FROM titlePrincipals tp
        JOIN namebasic nb ON tp.nconst = nb.nconst
        WHERE LOWER(nb.primaryname) = LOWER(p_primaryname) -- Match the person's name (case-insensitive)
    ),

    WordsFromTitles AS (
        -- Step 2: Retrieve words associated with these titles from the wi table
        SELECT wi.word AS word, wi.field AS category
        FROM wi
        JOIN PersonTitles pt ON wi.tconst = pt.tconst
        WHERE wi.field IN ('t', 'p', 'c') -- Focus on words from primarytitle, plot, and characters
    ),

    WordFrequencies AS (
        -- Step 3: Count the frequency of each word across the titles the person is involved in
        SELECT CAST(wt.word AS VARCHAR), CAST(wt.category AS VARCHAR), CAST(COUNT(*) AS INTEGER) AS frequency
        FROM WordsFromTitles wt -- Using alias 'wt' for WordsFromTitles CTE
        GROUP BY wt.word, wt.category -- Group by both word and category to distinguish between different word sources
        ORDER BY frequency DESC -- Sort by frequency in descending order
        LIMIT p_limit           -- Return only the top words, limited by p_limit
    )

    -- Step 4: Return the words, their frequencies, and their categories
    SELECT wf.word, wf.frequency, wf.category FROM WordFrequencies wf;

END;
$$ LANGUAGE plpgsql;
```

## D.11 Exact-match querying

The **exact_match_query** function implements an exact-match search where a list of keywords must be matched across the indexed data stored in the **wi** table. It returns titles that exactly match all the keywords provided in the input. The function ensures that only titles matching all keywords are returned and it uses **GROUP BY** and **HAVING** operators to ensure the exact match.

Listing 16: PostgreSQL Function for Exact Match Query

```
CREATE OR REPLACE FUNCTION exact_match_query(p_keywords TEXT[])
RETURNS TABLE(tconst VARCHAR, title VARCHAR) AS $$
BEGIN
    RETURN QUERY
    SELECT tb.tconst::VARCHAR, tb.primarytitle::VARCHAR
    FROM titlebasic tb
    JOIN (
        SELECT wi.tconst
        FROM wi
        WHERE wi.word = ANY(p_keywords)
        GROUP BY wi.tconst
        HAVING COUNT(DISTINCT wi.word) = array_length(p_keywords, 1)
    ) matched_titles ON tb.tconst = matched_titles.tconst;
END;
$$ LANGUAGE plpgsql;
```

## D.12 Best-match querying

The **best_match_query** function is a refined version of the **exact match_match_query** function. It ranks titles based on how many of the provided keywords are found. It takes an array of keywords as input and returns the matching titles ranked by the number of keywords matched.

Exact matches are given a higher priority. We did this with the **ORDER BY** operator, sorting the results by the match count in a descending order.

Listing 17: PostgreSQL Function for Best Match Query

```
CREATE OR REPLACE FUNCTION best_match_query(p_keywords TEXT[])
RETURNS TABLE(tconst VARCHAR, title VARCHAR, match_count INT) AS $$
BEGIN
    RETURN QUERY
    SELECT
        tb.tconst::VARCHAR,
        tb.primarytitle::VARCHAR,
        COUNT(DISTINCT wi.word)::INT AS match_count
    FROM
        titlebasic tb
    JOIN
        wi ON tb.tconst = wi.tconst
    WHERE
        wi.word = ANY(p_keywords)
    GROUP BY
        tb.tconst, tb.primarytitle
    ORDER BY
        match_count DESC;
END;
$$ LANGUAGE plpgsql;
```

## D.13 Word-to-word querying

The **word_to_words_query** function returns a ranked list of words that frequently appear relating to other keyword from the **wi** table. This is useful in scenarios where users are exploring a topic and wish to discover relating movies or keywords relating to titles or keywords. This function takes an array of keywords as input, retrieves matching titles that uses those keywords. Using a **COUNT** statement, it counts the frequency of words in those titles. The words are then returned in descending order of frequency. This approach allows users to understand the common themes or words associated with their search, showing which words are most relevant to the search query.

An different approach could have been providing titles directly, as in previous queries. But offering word-based insights adds another dimension to the search experience. Users can discover commonly associated words between movies and find titles from thematic elements of movies.

Listing 18: PostgreSQL Function for Word to Words Query

```
CREATE OR REPLACE FUNCTION word_to_words_query(p_keywords TEXT[])
RETURNS TABLE(word VARCHAR, frequency INT) AS $$
BEGIN
    RETURN QUERY
    WITH matched_titles AS (
        SELECT wi.tconst
        FROM wi
        WHERE wi.word = ANY(p_keywords)
        GROUP BY wi.tconst
    ),
    word_frequencies AS (
        SELECT wi.word::VARCHAR, COUNT(*)::INT AS frequency  -- Cast 'COUNT(*)' to INT
        FROM wi
        JOIN matched_titles mt ON wi.tconst = mt.tconst
        GROUP BY wi.word
    )
    SELECT wf.word, wf.frequency
    FROM word_frequencies wf
    ORDER BY wf.frequency DESC;
END;
$$ LANGUAGE plpgsql;
```

## D.14 Testing

Our full testing script can be found in the appendix and it runs each function we have created giving expected results, we also have the results from the output file with it to verify that each function works as intended in our github repository and handed in with the other code.

# E Indexing

Some indexing is done on our tables but it is mostly preliminary and in some cases it has been hard to see if it has improved results. On unique items or primary keys indexing is done automatically and that handles most of our use cases but an example that made a big difference was the indexes used on titlepricipals. For generating name ratings on almost 500k names this has a lot of searching and joining titlepricipals and these indexes caused the operation to be done in under 30 seconds when before it took multiple minutes. For most other functions we try to index based on what will be searched on but with each operation taking a fraction of a second the inconsistency between runs can sometimes be more than the potential saved from indexing. With more time and a more elaborate testing process we could run commands hundreds of times to get accurate performance results which may be relevant if our web pages are loading slow in later parts of the course.

Listing 19: PostgreSQL Indexing

```
CREATE INDEX IF NOT EXISTS index_tp_nconst ON titleprincipals (nconst);
CREATE INDEX IF NOT EXISTS index_tp_category ON titleprincipals (category);
CREATE INDEX IF NOT EXISTS index_nb_nrating ON namebasic (nrating);
CREATE INDEX IF NOT EXISTS index_tr_numvotes ON titleratings (numvotes);
CREATE INDEX IF NOT EXISTS index_tp_tconst ON titleprincipals (tconst);
CREATE INDEX IF NOT EXISTS index_tb_primarytitle ON titlebasic (primarytitle);
CREATE INDEX IF NOT EXISTS index_tl_languages ON titlelanguage (language);
CREATE INDEX IF NOT EXISTS index_tg_genre ON titlegenre (genre);
```

# Appendix

**B2_build_movie_db.sql** Script

```
CREATE TABLE titleBasic (
    tconst VARCHAR(10) PRIMARY KEY,
    titleType VARCHAR(256),
    primaryTitle VARCHAR(256),
    originalTitle VARCHAR(256),
    isAdult BOOLEAN,
    startYear CHAR(4),
    endYear CHAR(4),
    runtimeMinutes INT,
    awards VARCHAR(256),
    plot TEXT,
    rated VARCHAR(50),
    releaseDate VARCHAR(80),
    dvd VARCHAR(80),
    productionCompany VARCHAR(256),
    poster VARCHAR(256),
    boxOffice VARCHAR(256),
    website VARCHAR(256)
);

CREATE TABLE nameBasic (
    nconst VARCHAR(10) PRIMARY KEY,
    primaryName VARCHAR(256),
    birthYear CHAR(4),
    deathYear CHAR(4)
);

CREATE TABLE titleRatings (
    tconst VARCHAR(10) PRIMARY KEY,
    averageRating DECIMAL(3, 1),
    numVotes INT,
    rottenTomatoes INT,
    metaCritic INT,
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titleAkas (
    tconst VARCHAR(10),
    ordering INT,
    title VARCHAR(256),
    region VARCHAR(50),
    language VARCHAR(50),
    types VARCHAR(100),
    attributes VARCHAR(100),
```

```
    isOriginalTitle BOOLEAN,
    PRIMARY KEY (tconst, ordering),
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titleLanguage (
    tconst VARCHAR(10),
    language VARCHAR(256),
    PRIMARY KEY (tconst, language),
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titleEpisode (
    tconst VARCHAR(10),
    parenttconst VARCHAR(10),
    seasonNumber INT,
    episodeNumber INT,
    PRIMARY KEY (tconst),
    FOREIGN KEY (parenttconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE,
                FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titleCharacters (
    nconst VARCHAR(10),
    tconst VARCHAR(10),
    character VARCHAR(500),
                ordering INT,
    PRIMARY KEY (nconst, tconst, character, ordering),
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titleCountry (
    tconst VARCHAR(10),
    country VARCHAR(256),
    PRIMARY KEY (tconst, country),
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titleGenre (
    tconst VARCHAR(10),
    genre VARCHAR(50),
    PRIMARY KEY (tconst, genre),
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE titlePrincipals (
    tconst VARCHAR(10),
    ordering INT,
    nconst VARCHAR(10),
    category VARCHAR(50),
    job VARCHAR,
    PRIMARY KEY (tconst, ordering),
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE,
    FOREIGN KEY (nconst) REFERENCES nameBasic(nconst) ON DELETE CASCADE
);

CREATE TABLE nameKnownFor (
    nconst VARCHAR(10),
    knownForTitles VARCHAR(256),
    PRIMARY KEY (nconst, knownForTitles),
    FOREIGN KEY (nconst) REFERENCES nameBasic(nconst) ON DELETE CASCADE,
    FOREIGN KEY (knownForTitles) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

-- Insert data into titleBasic from IMDb's title_basics
INSERT INTO titleBasic (tconst, titleType, primaryTitle, originalTitle, isAdult,
    startYear, endYear, runtimeMinutes)
SELECT
    tconst,
    NULLIF(NULLIF(titleType, 'N/A'), ''),
    NULLIF(NULLIF(primaryTitle, 'N/A'), ''),
    NULLIF(NULLIF(originalTitle, 'N/A'), ''),
    isAdult,
    NULLIF(NULLIF(startYear, 'N/A'), ''),
    NULLIF(NULLIF(endYear, 'N/A'), ''),
    runtimeMinutes
```

```
FROM title_basics;

-- Insert data into titleBasic from omdb_data table
INSERT INTO titleBasic (
    tconst, awards, plot, rated, releaseDate, dvd,
    productioncompany, poster, boxOffice, website
)
SELECT
    tconst,
    NULLIF(NULLIF(awards, 'N/A'), ''),
    NULLIF(NULLIF(plot, 'N/A'), ''),
    NULLIF(NULLIF(rated, 'N/A'), ''),
    NULLIF(NULLIF(released, 'N/A'), ''),
    NULLIF(NULLIF(dvd, 'N/A'), ''),
    NULLIF(NULLIF(production, 'N/A'), ''),
    NULLIF(NULLIF(poster, 'N/A'), ''),
    NULLIF(NULLIF(boxOffice, 'N/A'), ''),
    NULLIF(NULLIF(website, 'N/A'), '')
FROM omdb_data
ON CONFLICT (tconst)
DO UPDATE
SET
    awards = COALESCE(NULLIF(NULLIF(EXCLUDED.awards, 'N/A'), ''), titleBasic.awards),
    plot = COALESCE(NULLIF(NULLIF(EXCLUDED.plot, 'N/A'), ''), titleBasic.plot),
    rated = COALESCE(NULLIF(NULLIF(EXCLUDED.rated, 'N/A'), ''), titleBasic.rated),
    releaseDate = COALESCE(NULLIF(NULLIF(EXCLUDED.releaseDate, 'N/A'), ''),
    titleBasic.releaseDate),
    dvd = COALESCE(NULLIF(NULLIF(EXCLUDED.dvd, 'N/A'), ''), titleBasic.dvd),
    productionCompany = COALESCE(NULLIF(NULLIF(EXCLUDED.productionCompany, 'N/A'), ''),
    titleBasic.productionCompany),
    poster = COALESCE(NULLIF(NULLIF(EXCLUDED.poster, 'N/A'), ''), titleBasic.poster),
    boxOffice = COALESCE(NULLIF(NULLIF(EXCLUDED.boxOffice, 'N/A'), ''),
    titleBasic.boxOffice),
    website = COALESCE(NULLIF(NULLIF(EXCLUDED.website, 'N/A'), ''), titleBasic.website);

-- Insert data into nameBasic from IMDb's name_basics
INSERT INTO nameBasic (nconst, primaryName, birthYear, deathYear)
SELECT
    nconst,
    NULLIF(NULLIF(primaryName, 'N/A'), ''),
    NULLIF(NULLIF(birthYear, 'N/A'), ''),
    NULLIF(NULLIF(deathYear, 'N/A'), '')
FROM name_basics;

-- Insert data into titleRatings from IMDb's title_ratings
INSERT INTO titleRatings (tconst, averageRating, numVotes)
SELECT
    tconst,
    averageRating,
    numVotes
FROM title_ratings;

-- Insert data into titleRatings from omdb_data table with Rotten Tomatoes and
    Metacritic ratings
WITH RatingsExtract AS (
    SELECT
        od.tconst,
        -- Extract Rotten Tomatoes rating
        CAST(
            TRIM(BOTH '%' FROM (
                SELECT rating ->> 'Value'
                FROM jsonb_array_elements(od.ratings::jsonb) AS rating
                WHERE rating ->> 'Source' = 'Rotten Tomatoes'
                LIMIT 1
            )) AS INT
        ) AS rottenTomatoes,
        -- Extract Metacritic rating
        CAST(
            SPLIT_PART(
                (
                    SELECT rating ->> 'Value'
                    FROM jsonb_array_elements(od.ratings::jsonb) AS rating
                    WHERE rating ->> 'Source' = 'Metacritic'
                    LIMIT 1
                ), '/', 1
            ) AS INT
```

```
        ) AS metaCritic
    FROM omdb_data od
)
-- Update titleRatings table with the extracted ratings from omdb_data
UPDATE titleRatings tr
SET
    rottenTomatoes = COALESCE(re.rottenTomatoes, tr.rottenTomatoes), -- Update only if new value is found
    metaCritic = COALESCE(re.metaCritic, tr.metaCritic)              -- Update only if new value is found
FROM RatingsExtract re
WHERE tr.tconst = re.tconst;

-- Insert data into titleAkas from IMDb's title_akas
INSERT INTO titleAkas (tconst, ordering, title, region, language, types, attributes, isOriginalTitle)
SELECT
    titleId AS tconst,
    ordering,
    NULLIF(NULLIF(title, 'N/A'), ''),
    NULLIF(NULLIF(region, 'N/A'), ''),
    NULLIF(NULLIF(language, 'N/A'), ''),
    NULLIF(NULLIF(types, 'N/A'), ''),
    NULLIF(NULLIF(attributes, 'N/A'), ''),
    isOriginalTitle
FROM title_akas;

-- Insert data into titleGenre by splitting genres from IMDb's title_basics
INSERT INTO titleGenre (tconst, genre)
SELECT
    tconst,
    TRIM(NULLIF(NULLIF(UNNEST(STRING_TO_ARRAY(genres, ',')), 'N/A'), ''))
FROM title_basics;

-- Insert data into titlePrincipals from IMDb's title_principals
INSERT INTO titlePrincipals (tconst, ordering, nconst, category, job)
SELECT
    tconst,
    ordering,
    nconst,
    NULLIF(NULLIF(category, 'N/A'), ''),
    NULLIF(NULLIF(job, 'N/A'), '')
FROM title_principals;

-- Insert data into titleCharacters from IMDb's title_principals
INSERT INTO titleCharacters (nconst, tconst, character, ordering)
SELECT
    nconst,
    tconst,
    REPLACE(REPLACE(REPLACE(NULLIF(NULLIF(characters, 'N/A'), ''), '[', ''), ']', ''), '""', '') AS cleaned_characters,
    ordering
FROM title_principals
WHERE characters IS NOT NULL
AND characters != ''
AND category = 'actor';

-- Insert data into nameKnownFor from IMDb's name_basics, ensuring only valid titles that are included in our database
INSERT INTO nameKnownFor (nconst, knownForTitles)
SELECT
    nb.nconst,
    nb.knownForTitles
FROM (
    SELECT
        nconst,
        TRIM(NULLIF(NULLIF(UNNEST(STRING_TO_ARRAY(NULLIF(knownForTitles, 'N/A'), ',')), ''), '')) AS knownForTitles
    FROM name_basics
) AS nb
JOIN titleBasic tb ON nb.knownForTitles = tb.tconst;

-- Insert data into titleEpisode from IMDb's title_episode
INSERT INTO titleEpisode(tconst, parenttconst, seasonnumber, episodenumber)
SELECT
    tconst,
```

```
    parenttconst,
    seasonnumber,
    episodenumber
FROM title_episode;

-- Insert data into titleCountry from omdb_data table
INSERT INTO titleCountry(tconst, country)
SELECT
    tconst,
    TRIM(NULLIF(NULLIF(UNNEST(STRING_TO_ARRAY(NULLIF(country, 'N/A'), ',')), ''), ''))
FROM omdb_data;

-- Insert data into titleLanguage from omdb_data table, ignoring duplicates
INSERT INTO titleLanguage(tconst, language)
SELECT
    tconst,
    TRIM(NULLIF(NULLIF(UNNEST(STRING_TO_ARRAY(NULLIF(language, 'N/A'), ',')), ''), ''))
FROM omdb_data
ON CONFLICT (tconst, language) DO NOTHING;

-- Alter wi table to ensure foreign key constraints and same type as tconst in
    titleBasic
ALTER TABLE wi
ALTER COLUMN tconst TYPE VARCHAR(20);
-- Add the foreign key constraint
ALTER TABLE wi
ADD CONSTRAINT fk_tconst_titleBasic
FOREIGN KEY (tconst) REFERENCES titleBasic(tconst)
ON DELETE CASCADE;

--Drop original tables
-- DROP TABLE
-- title_basics, name_basics, title_akas, title_crew, title_episode, title_principals,
    title_ratings;
```

**C2_build_framework_db.sql** Script

```
CREATE TABLE users (
   userId INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100),
    password VARCHAR(100)
);

CREATE TABLE userRatings (
    userId INT,
    tconst VARCHAR(10),
    rating DECIMAL(3, 1),
    ratingDate DATE,
    PRIMARY KEY (userId, tconst, rating),
    FOREIGN KEY (userId) REFERENCES users(userId) ON DELETE CASCADE,
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE
);

CREATE TABLE userSearchHistory (
    userId INT,
    searchQuery TEXT,
    searchDate DATE,
                PRIMARY KEY (userId, searchQuery, searchDate),
    FOREIGN KEY (userId) REFERENCES users(userId) ON DELETE CASCADE
);

CREATE TABLE userBookmarks (
    bookmarkId INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    userId INT,
    tconst VARCHAR(20),
    nconst VARCHAR(20),
    note TEXT,
    bookmarkDate DATE,
    FOREIGN KEY (userId) REFERENCES users(userId) ON DELETE CASCADE,
    FOREIGN KEY (tconst) REFERENCES titleBasic(tconst) ON DELETE CASCADE,
    FOREIGN KEY (nconst) REFERENCES nameBasic(nconst) ON DELETE CASCADE
);
```

**createFunctionScript.sql**

```
    CREATE OR REPLACE FUNCTION "public"."add_user"("p_username" varchar, "p_email" varchar)
  RETURNS "pg_catalog"."void" AS $BODY$
BEGIN
    INSERT INTO users (username, email)
    VALUES (p_username, p_email);
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;


  CREATE OR REPLACE FUNCTION "public"."bookmark_movie"("p_userid" int4, "p_tconst" varchar)
  RETURNS "pg_catalog"."void" AS
$BODY$
BEGIN
    -- Check if the bookmark already exists
    IF NOT EXISTS (
        SELECT 1
        FROM userbookmarks
        WHERE userid = p_userid AND tconst = p_tconst
    ) THEN
        -- If not exists, insert the new bookmark
        INSERT INTO userbookmarks (userid, tconst)
        VALUES (p_userid, p_tconst);
    END IF;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;

  CREATE OR REPLACE FUNCTION "public"."bookmark_name"("p_userid" int4, "p_nconst" varchar)
  RETURNS "pg_catalog"."void" AS
$BODY$
BEGIN
    -- Check if the bookmark for the name already exists
    IF NOT EXISTS (
        SELECT 1
        FROM userbookmarks
        WHERE userid = p_userid AND nconst = p_nconst
    ) THEN
        -- If not exists, insert the new bookmark
        INSERT INTO userbookmarks (userid, nconst)
        VALUES (p_userid, p_nconst);
    END IF;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;

  CREATE OR REPLACE FUNCTION "public"."get_bookmarks"("p_userid" int4)
RETURNS TABLE(bookmark_type VARCHAR, id VARCHAR, name_or_title VARCHAR) AS $$
BEGIN
    RETURN QUERY
    -- Fetch bookmarked movies
    SELECT
        'title'::VARCHAR AS bookmark_type,  -- Indicate this is a title bookmark, cast to VARCHAR
        b.tconst::VARCHAR AS id,            -- Return the tconst (movie/series ID), cast to VARCHAR
        tb.primarytitle::VARCHAR AS name_or_title  -- Return the movie/series title, cast to VARCHAR
    FROM
        userbookmarks b
    JOIN
        titlebasic tb ON b.tconst = tb.tconst
    WHERE
        b.userid = p_userid

    UNION ALL

    -- Fetch bookmarked actors/directors
    SELECT
        'name'::VARCHAR AS bookmark_type,   -- Indicate this is a name bookmark, cast
```

```
    to VARCHAR
71          b.nconst::VARCHAR AS id,            -- Return the nconst (name ID), cast to
    VARCHAR
72          nb.primaryname::VARCHAR AS name_or_title  -- Return the person's name, cast to
    VARCHAR
73      FROM
74          userbookmarks b
75      JOIN
76          namebasic nb ON b.nconst = nb.nconst
77      WHERE
78          b.userid = p_userid;
79  END;
80  $$ LANGUAGE plpgsql VOLATILE
81  COST 100
82  ROWS 1000;
83
84  CREATE OR REPLACE FUNCTION "public"."string_search"("p_search_string" varchar)
85    RETURNS TABLE("tconst" varchar, "title" varchar) AS $BODY$
86  BEGIN
87      RETURN QUERY
88      SELECT tb.tconst::VARCHAR, tb.primarytitle::VARCHAR
89      FROM titlebasic tb
90      WHERE tb.primarytitle ILIKE '%' || p_search_string || '%'
91         OR tb.tconst IN (
92             SELECT tb.tconst
93             FROM titlebasic tb
94             WHERE tb.plot ILIKE '%' || p_search_string || '%'
95         )
96      ORDER BY
97         -- Exact matches first
98         CASE
99             WHEN tb.primarytitle ILIKE p_search_string THEN 1
100            ELSE 2
101        END,
102        -- Then sort by closest partial matches
103        tb.primarytitle;
104 END;
105 $BODY$
106   LANGUAGE plpgsql VOLATILE
107   COST 100
108   ROWS 1000;
109
110
111   CREATE OR REPLACE FUNCTION "public"."string_search"("p_search_string" varchar,
      "p_userid" int4)
112   RETURNS TABLE("tconst" varchar, "title" varchar) AS $BODY$
113 BEGIN
114     -- Log the search history for the user
115     PERFORM update_search_history(p_userid, p_search_string);
116
117     -- Perform the search with prioritization of exact matches
118     RETURN QUERY
119     SELECT tb.tconst::VARCHAR, tb.primarytitle::VARCHAR
120     FROM titlebasic tb
121     WHERE tb.primarytitle ILIKE '%' || p_search_string || '%'
122        OR tb.tconst IN (
123            SELECT tb.tconst
124            FROM titlebasic tb
125            WHERE tb.plot ILIKE '%' || p_search_string || '%'
126        )
127     ORDER BY
128        -- Exact matches first
129        CASE
130            WHEN tb.primarytitle ILIKE p_search_string THEN 1
131            ELSE 2
132        END,
133        -- Then sort by closest partial matches
134        tb.primarytitle;
135 END;
136 $BODY$
137   LANGUAGE plpgsql VOLATILE
138   COST 100
139   ROWS 1000;
140
141   CREATE OR REPLACE FUNCTION "public"."update_search_history"("p_userid" int4,
      "p_search_string" varchar)
```

```
  RETURNS "pg_catalog"."void" AS
$BODY$
BEGIN
    -- Check if the search query already exists for the user
    IF NOT EXISTS (
        SELECT 1
        FROM userSearchHistory
        WHERE userId = p_userid AND searchQuery = p_search_string
    ) THEN
        -- If not exists, insert the new search query
        INSERT INTO userSearchHistory (userId, searchQuery, searchDate)
        VALUES (p_userid, p_search_string, CURRENT_DATE);
    END IF;
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;

  create or replace PROCEDURE rate(titleId VARCHAR(10), _rating int4, _userId int4)
LANGUAGE plpgsql as $$
declare oldRating int;
oldnumvotes int;
oldaveragerating numeric(3,1);
BEGIN

if _rating between 1 and 10 then
  select numvotes from titleratings tr where tr.tconst = titleId into oldnumvotes;
  select averagerating from titleratings tr where tr.tconst = titleId into
    oldaveragerating;
  if exists (select 1 from userratings ur where ur.tconst = titleId and ur.userid =
    _userId) THEN
    --update existing rating
    --update userratings (get old value)
    select rating from userratings ur where ur.tconst = titleId and ur.userid = _userId
    into oldRating;
    update userratings set rating = _rating, ratingdate = CURRENT_DATE where userid =
    _userId and tconst = titleId;
    --update titleratings (needs old value)
    update titleratings set averagerating =
    (COALESCE(oldnumvotes,0)*COALESCE(oldaveragerating,0)-oldRating +
    _rating)/COALESCE(oldnumvotes,1) where tconst = titleId;
  ELSE
    --new rating
    --insert into userratings
    insert into userratings(userid, tconst, rating, ratingdate)
    values (_userId, titleId, _rating, CURRENT_DATE);
    --update titleratings
    update titleratings set numvotes = numvotes + 1,
    averagerating = (COALESCE(oldnumvotes,0)*COALESCE( oldaveragerating,0) +
    _rating)/(COALESCE(oldnumvotes)+1) where tconst = titleId;
  end if;
    --update nrating in namebasic
    update namebasic set nrating = (select
    round(sum(tr.averagerating*tr.numvotes)/sum(tr.numvotes),1) from titleprincipals tp
    inner join titleratings tr on tp.tconst = tr.tconst where namebasic.nconst =
    tp.nconst) where namebasic.nconst in (select nconst from titleprincipals where
    tconst = titleId);
else
  raise notice 'Input value must be between 1 and 10';
end if;
END;
$$;


CREATE OR REPLACE FUNCTION "public"."structured_string_search"("p_title" varchar,
    "p_plot" varchar, "p_characters" varchar, "p_names" varchar)
  RETURNS TABLE("tconst" text, "title" text) AS $BODY$
BEGIN
    RETURN QUERY
    SELECT tb.tconst::TEXT, tb.primarytitle::TEXT
    FROM titlebasic tb
    JOIN titlecharacters tp ON tb.tconst = tp.tconst
    JOIN namebasic nb ON tp.nconst = nb.nconst
    WHERE tb.primarytitle ILIKE '%' || p_title || '%'
      AND tb.plot ILIKE '%' || p_plot || '%'
      AND tp.character ILIKE '%' || p_characters || '%'
```

```
      AND nb.primaryname ILIKE '%' || p_names || '%';
END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;

CREATE OR REPLACE FUNCTION search_names_by_text(search_text VARCHAR)
RETURNS TABLE (
    nconst VARCHAR(10),
    primaryName VARCHAR(256),
    birthYear CHAR(4),
    deathYear CHAR(4)
)
AS $$
BEGIN
    RETURN QUERY
    SELECT
        nb.nconst,        -- Qualify nconst with the table alias
        nb.primaryName,   -- Qualify primaryName with the table alias
        nb.birthYear,
        nb.deathYear
    FROM
        nameBasic nb      -- Use an alias for the table
    WHERE
        nb.primaryName ILIKE '%' || search_text || '%';  -- Case-insensitive search
END;
$$ LANGUAGE plpgsql;


create or replace function coPlayers(testId varCHAR (10)) RETURNS TABLE (nconst
    varCHAR(10), primaryname VARCHAR(256), frequency BIGINT)
LANGUAGE plpgsql as $$
BEGIN

return query
SELECT tp.nconst, nb.primaryname, count(tp.tconst) as freq from titleprincipals tp JOIN
    namebasic nb on tp.nconst = nb.nconst where tp.tconst in (select tconst from
    titleprincipals where titleprincipals.nconst = testId) and (tp.category = 'actor'
    or tp.category = 'actress') and nb.nconst != testId group by tp.nconst,
    nb.primaryname order by freq desc;

END;
$$;

ALTER TABLE namebasic ADD IF NOT EXISTS nRating NUMERIC (5,1);
CREATE INDEX IF NOT EXISTS index_tp_nconst ON titleprincipals (nconst);
UPDATE namebasic
SET nRating=(
SELECT round(SUM (tr.averagerating*tr.numvotes)/SUM (tr.numvotes),1) FROM
    titleprincipals tp INNER JOIN titleratings tr ON tp.tconst=tr.tconst WHERE
    namebasic.nconst=tp.nconst);
CREATE INDEX IF NOT EXISTS index_tp_category ON titleprincipals (category);
CREATE INDEX IF NOT EXISTS index_nb_nrating ON namebasic (nrating);
CREATE INDEX IF NOT EXISTS index_tr_numvotes ON titleratings (numvotes);
CREATE INDEX IF NOT EXISTS index_tp_tconst ON titleprincipals (tconst);
CREATE INDEX IF NOT EXISTS index_tb_primarytitle ON titlebasic (primarytitle);
CREATE INDEX IF NOT EXISTS index_tl_languages ON titlelanguage (language);
CREATE INDEX IF NOT EXISTS index_tg_genre ON titlegenre (genre);

create or replace function ratingActors(testId VARCHAR(10)) returns table (nconst
    VARCHAR(10), nRating numeric(5,1))
LANGUAGE plpgsql as $$
BEGIN

return query
select nb.nconst, nb.nRating from (select tp.nconst from titleprincipals tp where
    tp.tconst = testId and (category = 'actor' or category = 'actress')) natural join
    namebasic nb order by nb.nRating DESC;

END;
$$;

create or replace function ratingCoPlayers(testId VARCHAR (10)) RETURNS TABLE (nconst
    VARCHAR(10), primaryname VARCHAR(256), nRating numeric(5,1))
LANGUAGE plpgsql as $$
```

```
271 BEGIN
272
273 return query
274 SELECT DISTINCT tp.nconst, nb.primaryname, nb.nRating from titleprincipals tp natural
        JOIN namebasic nb where tp.tconst in (select tconst from titleprincipals where
        titleprincipals.nconst = testId) and (tp.category = 'actor' or tp.category =
        'actress') and nb.nconst != testId order by nrating desc nulls last;
275
276 END;
277 $$;
278
279 create or replace function ratingCrew(testId VARCHAR(10)) returns table (nconst
        VARCHAR(10), nRating numeric(5,1))
280 LANGUAGE plpgsql as $$
281 BEGIN
282
283 return query
284 select nb.nconst, nb.nRating from (select tp.nconst from titleprincipals tp where
        tp.tconst = testId and category != 'actor' and category != 'actress') natural join
        namebasic nb order by nb.nRating DESC;
285
286 END;
287 $$;
288
289 create or replace function similarMovies(testId VARCHAR(10)) returns table (tconst
        VARCHAR(10), primarytitle VARCHAR(256), numvotes int4)
290 LANGUAGE plpgsql as $$
291 BEGIN
292
293 return query
294 select DISTINCT tb.tconst, tb.primarytitle, tr.numvotes from titlebasic tb
295 natural join titlegenre tg
296 natural join titlelanguage tl
297 natural join titleratings tr
298 where tg.genre in (select tg.genre from titlegenre tg where tg.tconst = testId)
299 and tl.language in (select language from titlelanguage tl where tl.tconst = testId)
300 order by tr.numvotes DESC limit 10;
301
302 END;
303 $$;
304
305 DROP FUNCTION IF EXISTS person_words(VARCHAR, INTEGER);
306
307 CREATE OR REPLACE FUNCTION person_words(
308     p_primaryname VARCHAR,      -- The name of the person we're interested in
309     p_limit INTEGER DEFAULT 10 -- Limit on the number of words returned (default 10)
310 )
311 RETURNS TABLE (
312     word VARCHAR,               -- The word associated with the person
313     frequency INTEGER,          -- Frequency of the word in titles the person is
        involved in
314     category VARCHAR            -- The field category (t = title, p = plot, c =
        characters)
315 ) AS $$
316 BEGIN
317     RETURN QUERY
318     WITH PersonTitles AS (
319         -- Step 1: Retrieve all titles (tconst) the person is associated with from the
        titlePrincipals table
320         SELECT DISTINCT tp.tconst
321         FROM titlePrincipals tp
322         JOIN namebasic nb ON tp.nconst = nb.nconst
323         WHERE LOWER(nb.primaryname) = LOWER(p_primaryname) -- Match the person's name
        (case-insensitive)
324     ),
325
326     WordsFromTitles AS (
327         -- Step 2: Retrieve words associated with these titles from the wi table
328         SELECT wi.word AS word, wi.field AS category
329         FROM wi
330         JOIN PersonTitles pt ON wi.tconst = pt.tconst
331         WHERE wi.field IN ('t', 'p', 'c') -- Focus on words from primarytitle, plot,
        and characters
332     ),
333
334     WordFrequencies AS (
```

```
        -- Step 3: Count the frequency of each word across the titles the person is involved in
        SELECT CAST(wt.word AS VARCHAR), CAST(wt.category AS VARCHAR), CAST(COUNT(*) AS INTEGER) AS frequency
        FROM WordsFromTitles wt -- Using alias 'wt' for WordsFromTitles CTE
        GROUP BY wt.word, wt.category -- Group by both word and category to distinguish between different word sources
        ORDER BY frequency DESC -- Sort by frequency in descending order
        LIMIT p_limit           -- Return only the top words, limited by p_limit
    )

    -- Step 4: Return the words, their frequencies, and their categories
    SELECT wf.word, wf.frequency, wf.category FROM WordFrequencies wf;

END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION exact_match_query(p_keywords TEXT[])
RETURNS TABLE(tconst VARCHAR, title VARCHAR) AS $$
BEGIN
    RETURN QUERY
    SELECT tb.tconst::VARCHAR, tb.primarytitle::VARCHAR
    FROM titlebasic tb
    JOIN (
        SELECT wi.tconst
        FROM wi
        WHERE wi.word = ANY(p_keywords)
        GROUP BY wi.tconst
        HAVING COUNT(DISTINCT wi.word) = array_length(p_keywords, 1)
    ) matched_titles ON tb.tconst = matched_titles.tconst;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION best_match_query(p_keywords TEXT[])
RETURNS TABLE(tconst VARCHAR, title VARCHAR, match_count INT) AS $$
BEGIN
    RETURN QUERY
    SELECT
        tb.tconst::VARCHAR,
        tb.primarytitle::VARCHAR,
        COUNT(DISTINCT wi.word)::INT AS match_count
    FROM
        titlebasic tb
    JOIN
        wi ON tb.tconst = wi.tconst
    WHERE
        wi.word = ANY(p_keywords)
    GROUP BY
        tb.tconst, tb.primarytitle
    ORDER BY
        match_count DESC;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION word_to_words_query(p_keywords TEXT[])
RETURNS TABLE(word VARCHAR, frequency INT) AS $$
BEGIN
    RETURN QUERY
    WITH matched_titles AS (
        SELECT wi.tconst
        FROM wi
        WHERE wi.word = ANY(p_keywords)
        GROUP BY wi.tconst
    ),
    word_frequencies AS (
        SELECT wi.word::VARCHAR, COUNT(*)::INT AS frequency  -- Cast 'COUNT(*)' to INT
        FROM wi
        JOIN matched_titles mt ON wi.tconst = mt.tconst
        GROUP BY wi.word
    )
    SELECT wf.word, wf.frequency
    FROM word_frequencies wf
    ORDER BY wf.frequency DESC;
END;
$$ LANGUAGE plpgsql;
```

**function_testing.sql** Script

```
    --D1 test
select * from add_user('testUser1', 'testUser@ruc.dk');
select * from add_user('testUser2', 'testUser@ruc.dk');
select * from add_user('testUser3', 'testUser@ruc.dk');
select * from bookmark_movie(1, 'tt10061752');
select * from bookmark_name(1, 'nm5189091');
select * from get_bookmarks(1);
--D2 test
select * from string_search('Inception');
select * from string_search('Inception', 1);
--D3 test
select * from titleratings where tconst = 'tt10061752';
select * from namebasic where nconst = 'nm5189091';
call rate('tt10061752', 1, 1);
call rate('tt10061752', 1, 2);
call rate('tt10061752', 1, 3);
select * from titleratings where tconst = 'tt10061752';
select * from namebasic where nconst = 'nm5189091';
select * from userratings;
call rate('tt10061752', 10, 1);
select * from titleratings where tconst = 'tt10061752';
select * from namebasic where nconst = 'nm5189091';
select * from userratings;
--D4 test
SELECT * FROM structured_string_search('Inception', 'Dream', 'Cobb', 'leonardo');
--D5 test
select * from search_names_by_text('Gosling');
--D6 test
select * from coplayers('nm0331516');
--D7 test
select * from namebasic where nconst = 'nm0331516';
--D8 test
select * from ratingactors('tt3783958');
select * from ratingcoplayers('nm0331516');
select * from ratingcrew('tt3783958');
--D9 test
select * from similarmovies('tt3783958');
--D10 test
SELECT * FROM person_words('Will Smith');
--D11 test
SELECT * FROM exact_match_query(ARRAY['inception', 'dream']);
--D12 test
SELECT * FROM best_match_query(ARRAY['inception', 'dream']) LIMIT 100;
--D13 test
SELECT * FROM word_to_words_query(ARRAY['inception', 'dream']) limit 100;
```

## Mockups

Figure 7: Mockup of final ideas for actor/director page

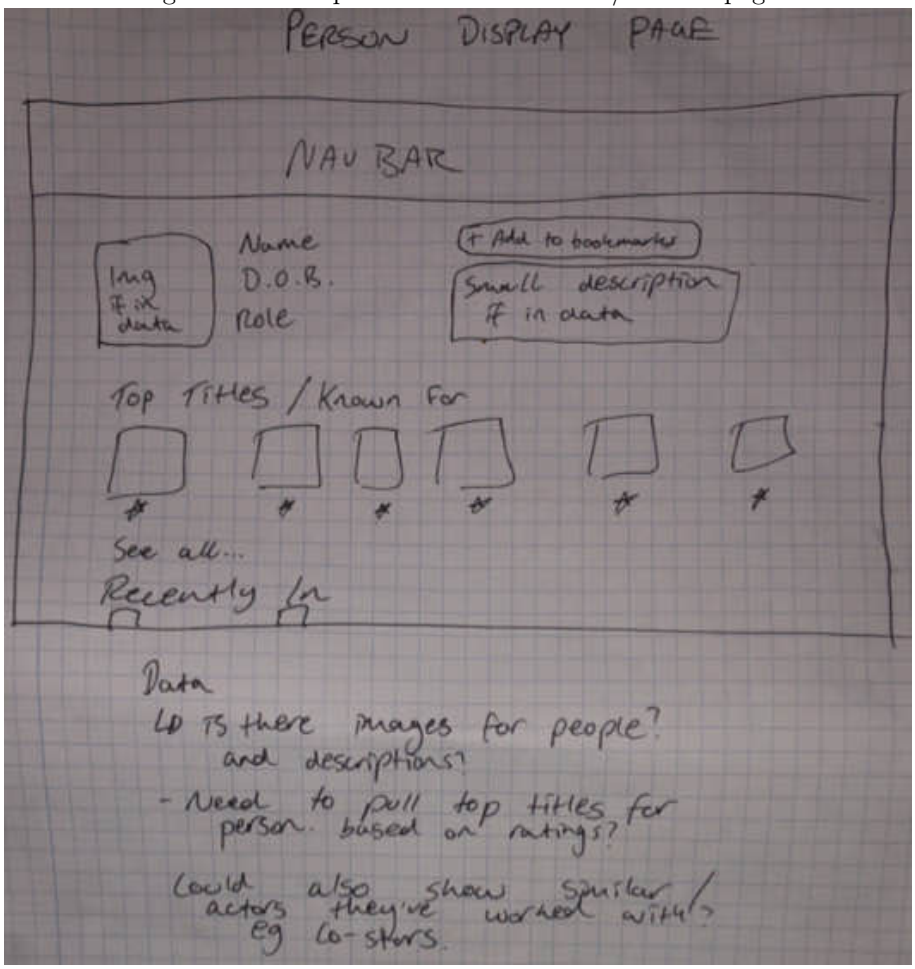

Figure 8: Mockup of final ideas for movie/series page

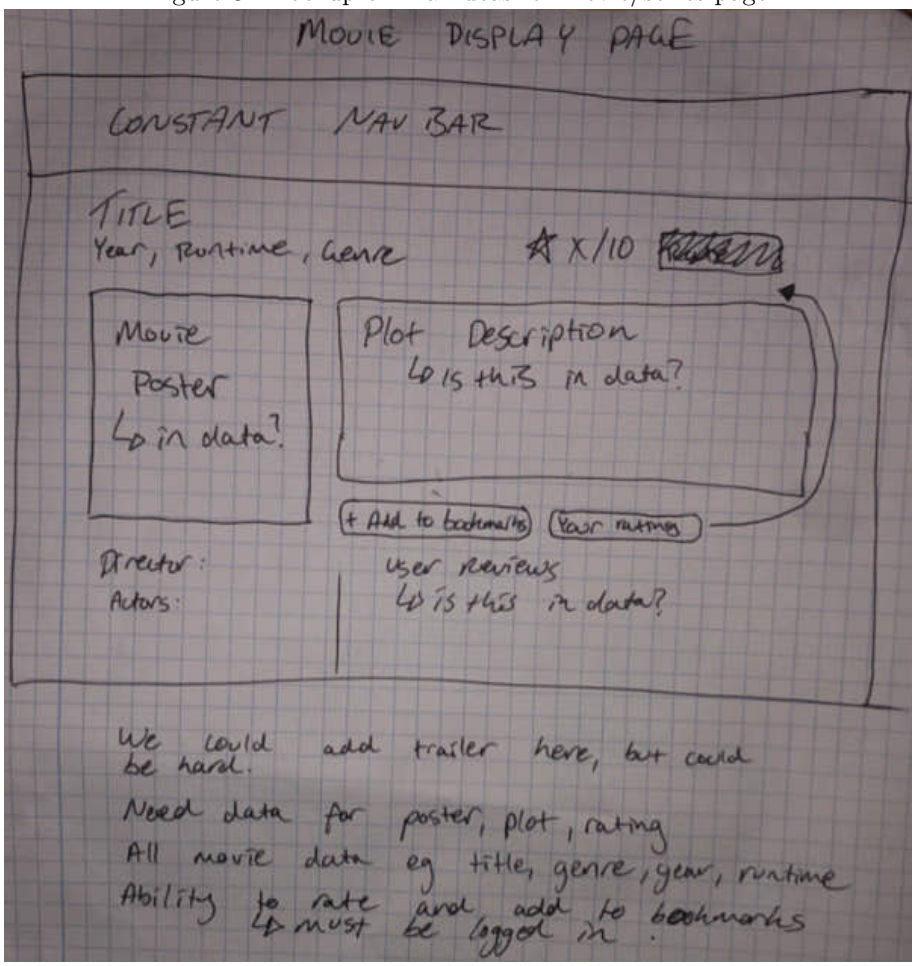

Figure 9: Mockup of final idea for search page

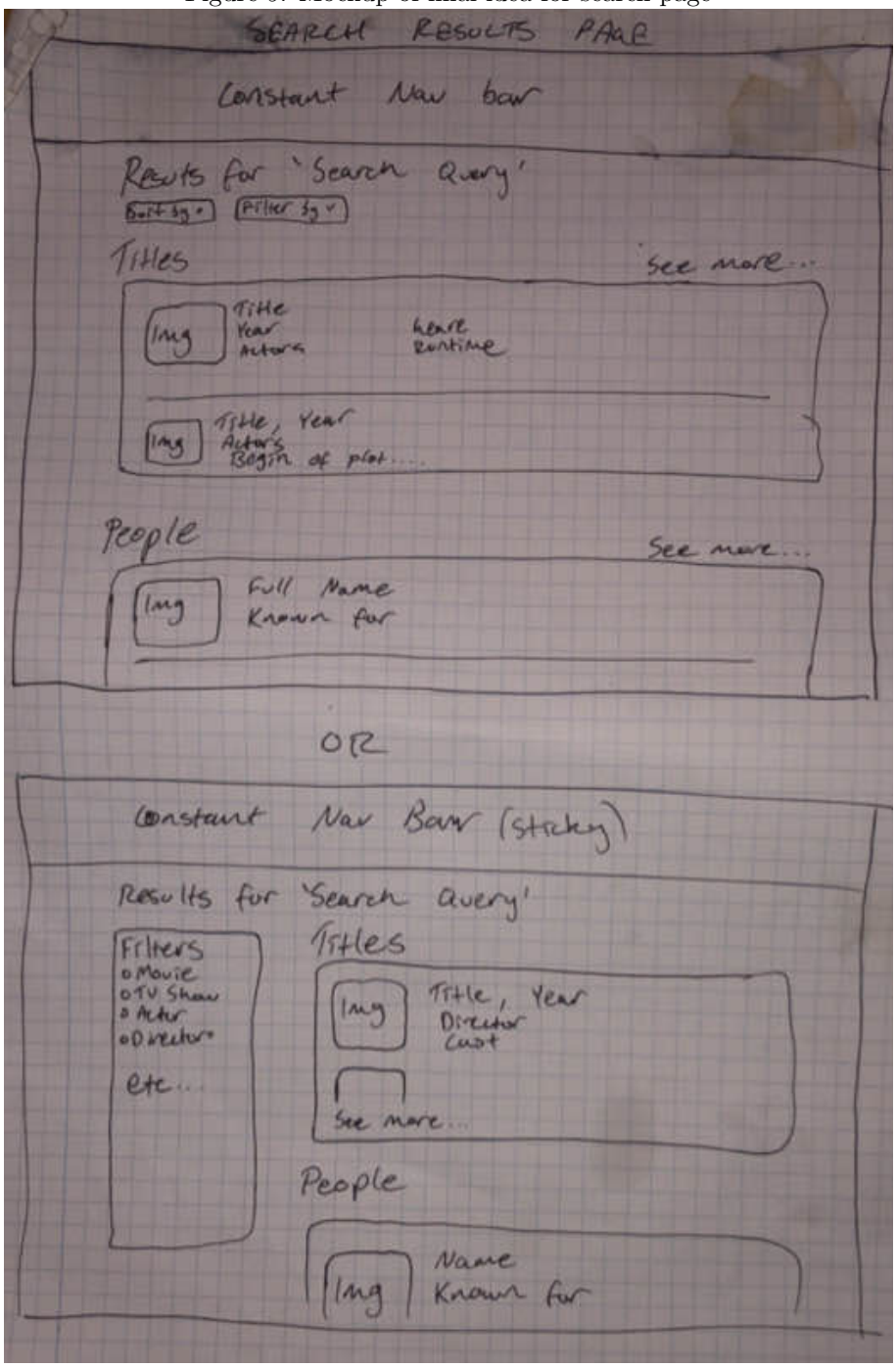

Figure 10: Mockup of log-in/sing-up page

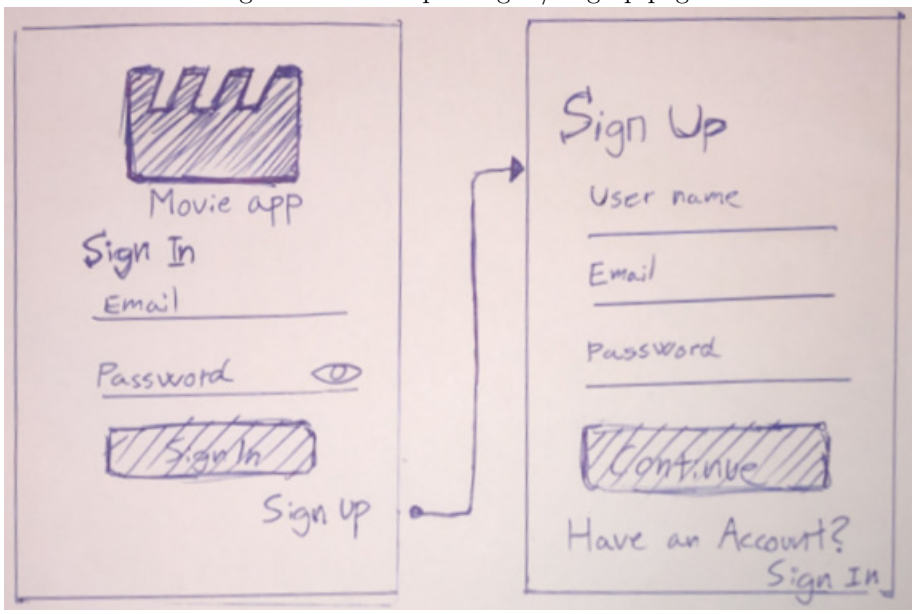

Figure 11: Mockup of early ideas for actor/director page

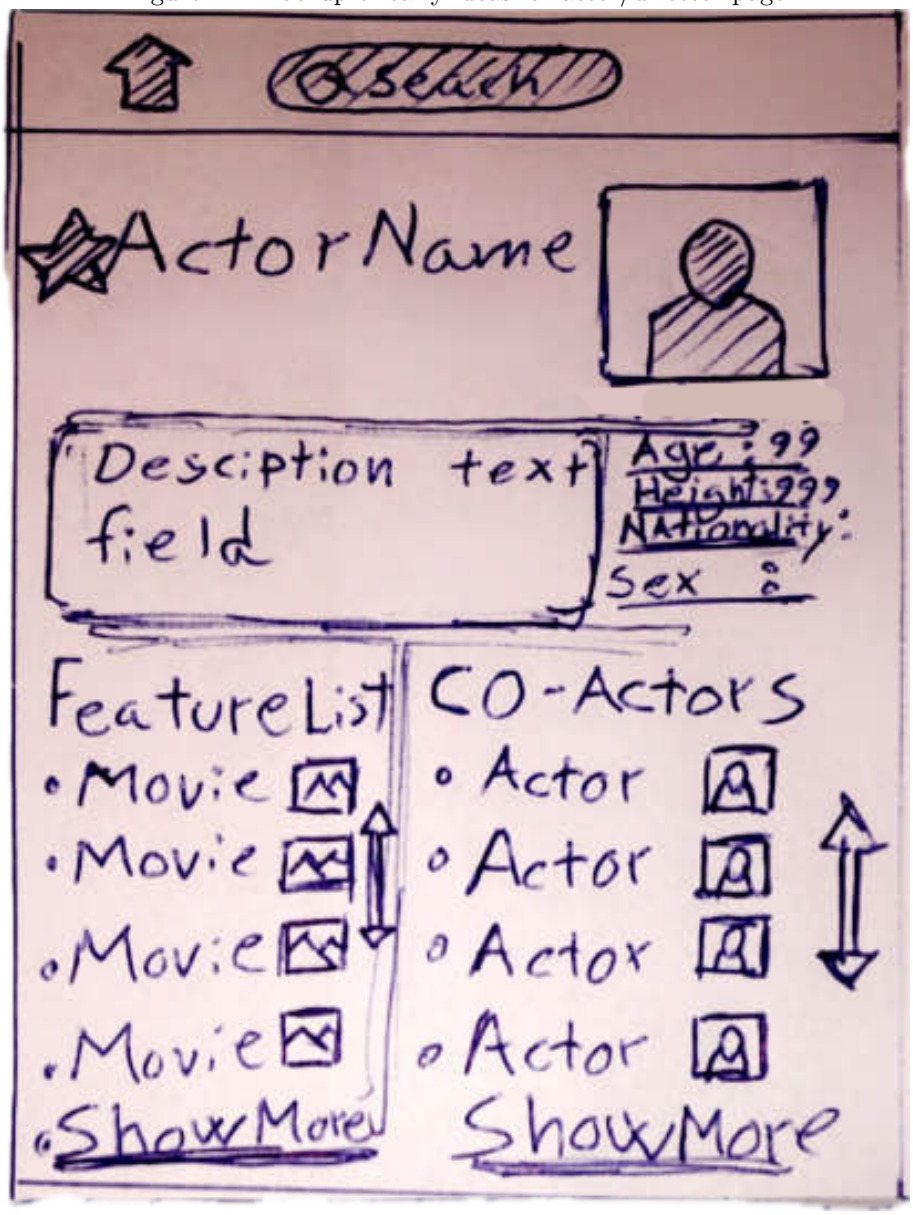

Figure 12: Mockup of early ideas for movie/series page

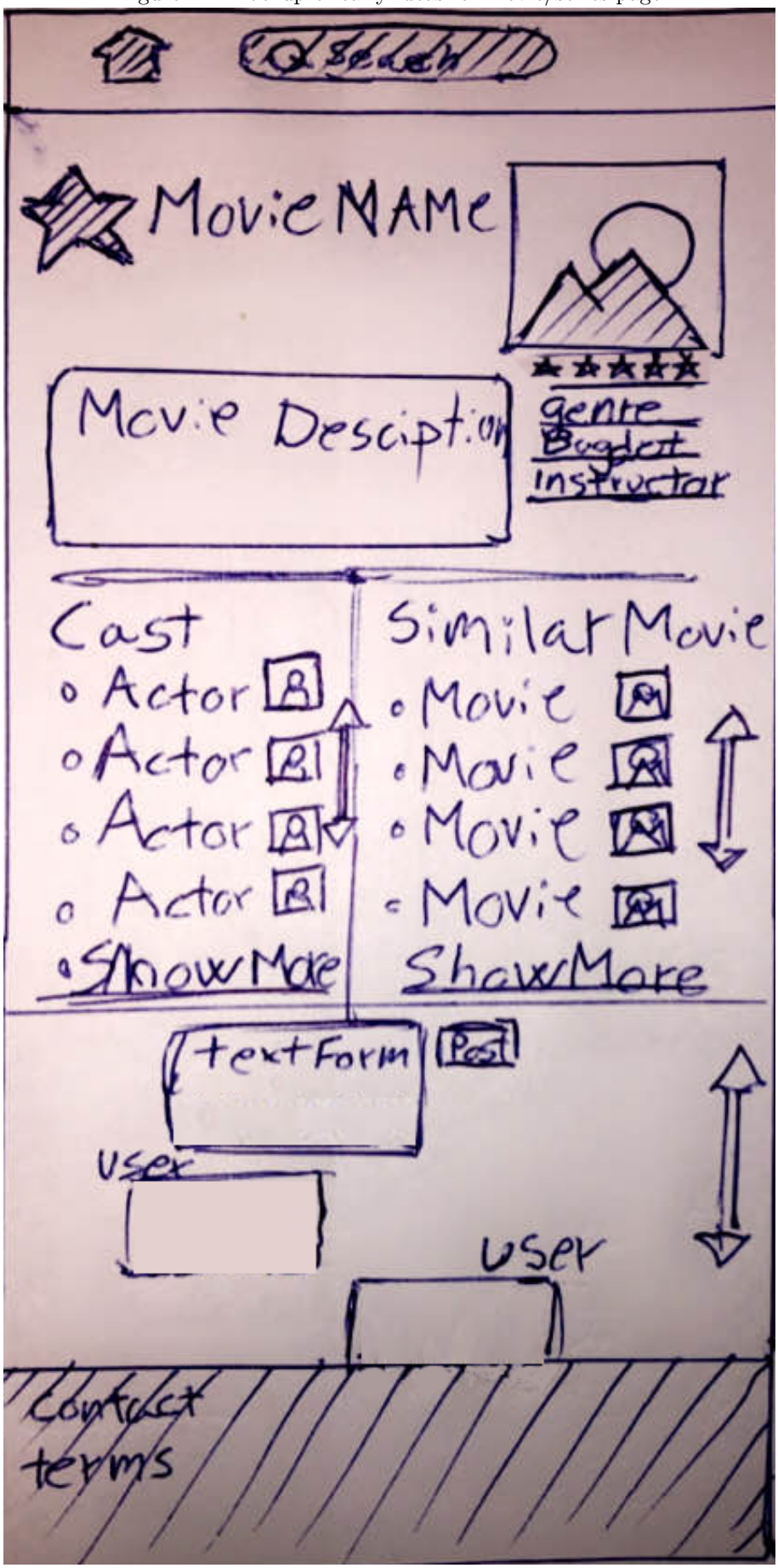

Figure 13: Mockup of scrapped user page idea