Advanced Management of Data Term Paper

Final Project: Film Management System

TECHNISCHE UNIVERSITÄT
IN DER KULTURHAUPTSTADT EUROPAS
CHEMNITZ

Name: Calvin Liusnando
Course of Study: Web Engineering
Matriculation Number: 708955

Table of Contents

Word count: 4012 words.

Utilised Technologies / Frameworks

As per the requirements, PostgreSQL and PL/pgSQL are used for the database section of the project. The extension "**uuid-ossp"** is installed (via PL/pgSQL) into the database to generate UUIDs (version 4, generated randomly) which are used as the primary key *(PK)* for the tables of the designed database model.

ASP.NET Core (v5.0.0) is used for the client-server implementation. It is chosen because it provides a simple, ready-to-use, and nice template for building web applications. It also has a rich environment, thorough documentation, and many tools that greatly help in web development. The template **MVC Web Application** is used due to its simplicity, and also because it facilitates a straighforward conversion from database tables into its respective model classes. **Razor** templating engine is used by default as ASP.NET MVC uses Razor for its HTML rendering engine.

Since the project is developed in .NET platform, **C#** is used as the server-side programming language. **Npgsql (6.0.3)** package is used to access the PostgreSQL database. Npgsql provides a simple and efficient way to connect to and make queries to the database, and it offers full support for most of PostgreSQL types, which are good enough for the purposes of this project.

System.Configuration.ConfigurationManager (v6.0.0) package is used to handles the configuration for the database connection, where user can simply change the values in the App.config file to connect with another database.

For client-side, **jQuery (v3.5.1)** is used as the javascript framework for client-server communication with AJAX. **Bootstrap (v4.3.1)** is used as the CSS framework for styling the pages with minimal effort and to have responsive web design. **DataTables (v1.10.2)** is used to create tables with advanced multi-features such as pagination, filtering and sorting to provide better user experience. Similarly, **Slim Select (v1.27.1)** is used to replace native html <select> tag with enhanced select dropdown for better visual and data manipulation.

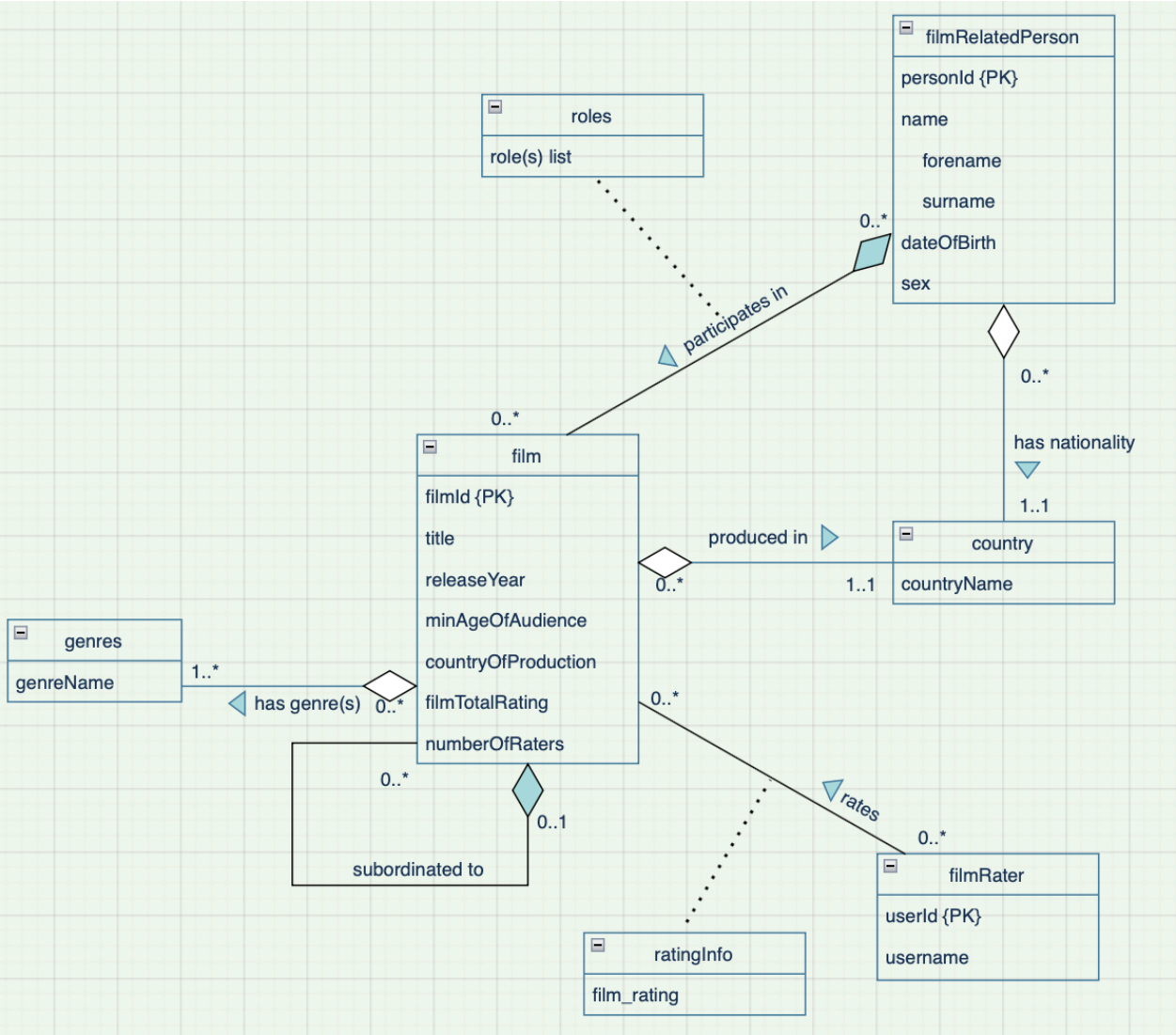
The project is written in **Visual Studio Community 2019** for Mac Community, an IDE platform developed by Microsoft. All of the used technologies are open-source and cross-platform, and each of them has its documentation that can be found in the References part of the term paper.

Database Design

For this section, the design steps written in the lecture slides of Dr. Seifert will be followed closely. The database design is divided into three phases:

Conceptual Database Design

In this phase, the Entity-Relationship Model is constructed based on the entities, attributes, relations described in the project task. Note that the UML diagram below **does not** represent the actual database model used in the final project of the author.



Some notes on the UML diagram

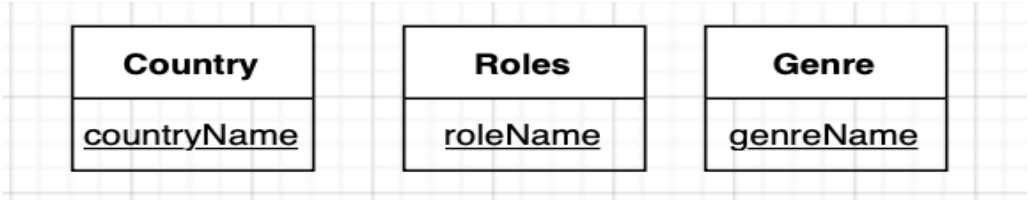
- filmRelatedPerson has a composite relation with film since in the case of deletion of a filmRelatedPerson entity, all related film entities would also get removed too.
- The similar reasoning also applies for film to film relation.

- It is acknowledged that in real world, a film/film-related person might have multiple country of productions/nationalities. However, in the implementation of this project, a film/film-related person can only have single country of production/nationality. Hence, the multiplicity attributes became many-to one instead of many-to-many for those two relations (film with country, filmRelatedPerson with country).
- There is no representation on film suggestions system, as this feature is implemented as a function *(see Functionalities Overview - Film Suggestions System for details)*, and thus it is not going to be displayed in a class diagram.

Logical Database Design

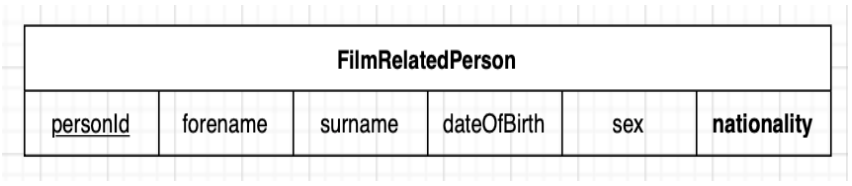
In this phase, the previous conceptual UML diagram is mapped to relational schema which will be used as the base for the actual database model used in the project implementation.

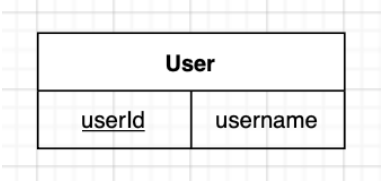
First, three supporting relations (read-only) are obtained:

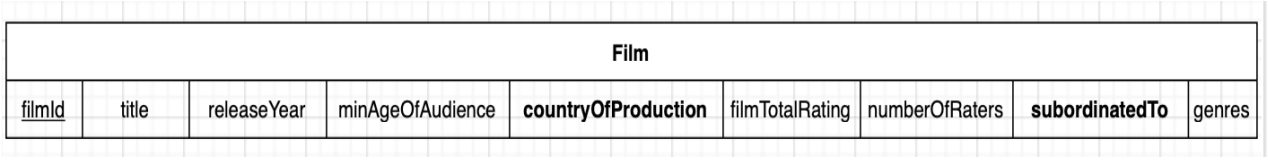


These relations are created so that for this information, users can only choose from pre-defined options instead of inputting their text, which could be non-valid data.

Then, the “main” relations are represented below:



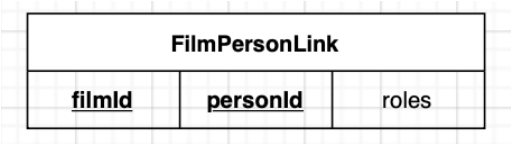




- *nationality* & *countryOfProduction* references *countryName* of *Country.*

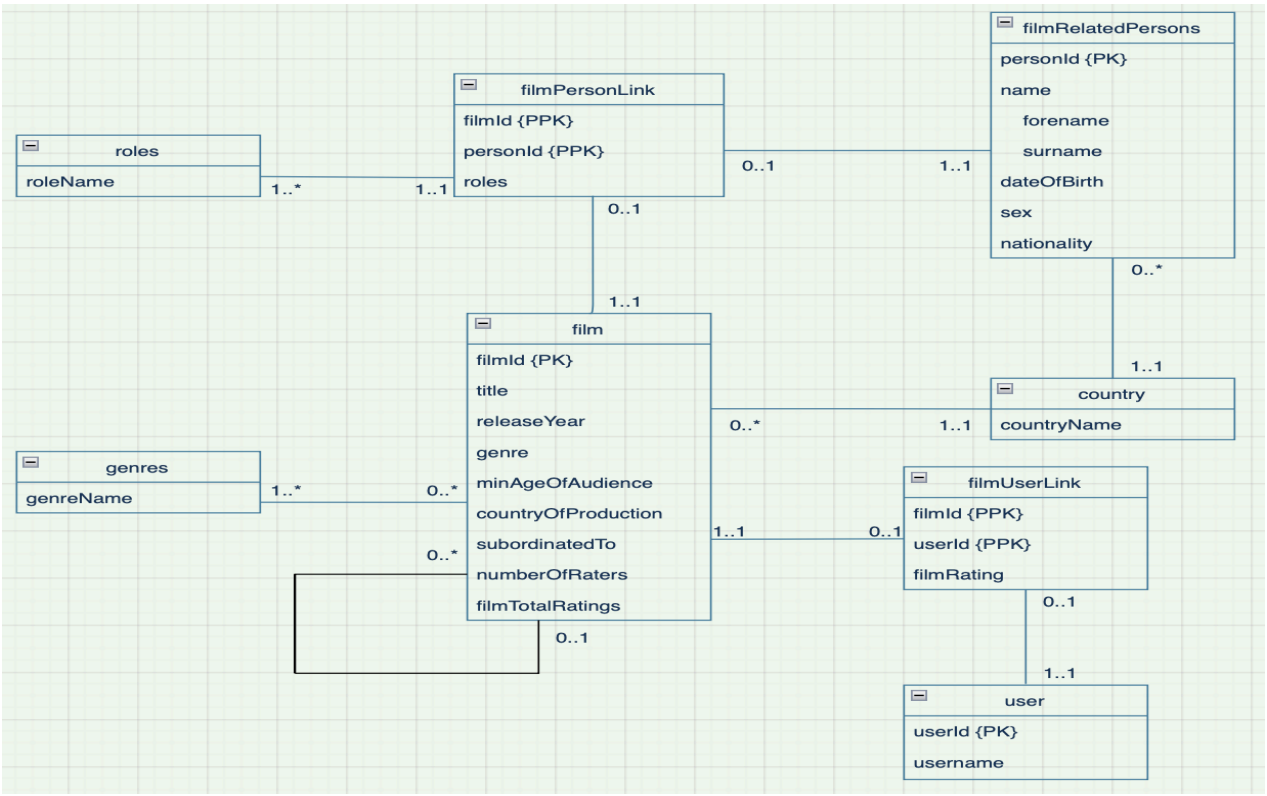
- *subordinatedTo* references *filmId* of *Film.*
- note that *genres* are collection of *genreName,* and is represented with text where each different genre is separated by '&'. Hence, *genres* does not directly reference *genreName.* Validation is instead done in the functions that are called when an user wants to add / update a film.

New relations are created to represent the relationships between "main" entities (Film and FilmRelatedPerson, Film and User).



- (*filmId, personId)* forms a composite primary key for *FilmPersonLink,* where *filmId* references *filmId* of *Film and personId* references *personId* of *FilmRelatedPerson.*
- *roles* is implemented similarly like *genres*, and thus it does not have any direct reference.
- (*filmId, userId)* forms a composite primary key for *FilmUserLink,* where *filmId* references *filmId* of *Film and userId* references user*Id* of *FilmUserLink.*

Finally, with all of the relations defined, below is the ER Model that represents the database model used in the project implementation:



Physical Database Design

In this section, the focuses will be on the data types used and constraints existing in the database model.

~ Data types

As mentioned in the *Utilised Technologies / Frameworks* section, UUIDs are used as the PKs. UUIDs are chosen over SERIAL due to various reasons: with UUIDs, all of the PK for entities across all tables (and databases) would be unique (probability of collision is negligible, and even in case it happens, the database would prevent it due to PK constraint). In this way, merging rows from different tables/databases can be done with ease since there would not be a conflicting PK, and this would be useful in the aspect of building scalable applications. In addition, UUIDs do not expose any information regarding the database itself, and with that in mind, the PKs can be used as parameters in URL without need to worry that unwanted information is shared with the general public.

In the previous section, it is mentioned that *genres* of *film* and *roles* in *filmPersonLink* are represented with text, where each genre/role is separated by '&'. This approach is preferred because maintaining text is much more simple than maintaining arrays (of composite type) in table. Furthermore, when needed, converting text to array can be done easily in PL/pgSQL with the array function *string_to_array(),* and thus there is no real benefit in storing an array instead of a text.

dateOfBirth of *filmRelatedPerson* uses the date data type, with ISO 8601 format (YYYY-MM-DD). *sex* uses the text data type, but it can only takes three values: "male", "female", or "N/A". Validations are done in client-side and the functions in the database. Other attributes inside the tables that are not specifically mentioned use common data types of integer and text.

~ Constraints (excluding PK, FK constraints mentioned in previous section):

- *releaseYear* of *film* is an integer within 1888 to 2022 (inclusive),
- *minAgeOfAudience* is an integer within 1 to 21 (inclusive),
- *title, genres* of *film* are non-NULL, non-empty texts,
- filmTotalRatings, numberOfRaters of film are integers >= 0,
- *filmId* and *subordinatedTo* of a *film* entity cannot be equal,
- accepted *dateOfBirth* of *filmRelatedPerson* is between 1900-01-01 to 2020-01-01 (inclusive),

- *name* of *filmRelatedPerson* is a composite type where both *forename* and *surname* are non-NULL and non-empty texts,
- *username* of *user* is a non-NULL, non-empty text, and it has an UNIQUE constraint added. This attribute is an alternate key.
- *roles* in *filmPersonLink* is a non-NULL, non-empty texts,
- *filmRating* in *filmUserLink* is an integer within 1 to 10 (inclusive).
- an additional logical constraint is added: the *releaseYear* of a subordinated film x has to be at least in the same year as its parent (and/or grandparent) film y.
- after an update on a film, there should not be a cycle of subordination among films. This can be done by treating the film table as a directed acyclic graph, and with recursive query, check whether there would be a cycle created due to the update in film's attributes, and then prevent the query if it does.
- in case of deletion of a film, delete all of the subordinated films and also related entries inside *filmPersonLink* and *filmUserLink.* This is done with a BEFORE trigger, and the order in which entries are first deleted is important, otherwise the FK constraints would be violated. First, delete the entries in *filmPersonLink* and *filmUserLink* that are related to the particular deleted film and its subordinated film(s). Then, delete all of the subordinated films. Finally, the film itself can be deleted safely without error.
- in case of deletion of a film-related person, delete all of the film(s) and their subordinated film(s), and the related entries inside *filmPersonLink* and *filmUserLink.* This is also done with BEFORE trigger. First, retrieve all of the soon-to-be deleted films and store it in an array. Then, delete all of the entries of *filmPersonLink* that are directly related to the film-related person that is going to be deleted. Iterate over the film array and delete each film (and then the BEFORE trigger for film would fire to delete the relevant *filmPersonLink* and *filmUserLink).*
- in case of an insertion, update, or deletion in *filmUserLink*, which would happen when an user add/edit/delete a film rating or when user gets deleted, update the attributes *filmTotalRating* and *numberOfRaters* of affected film(s) accordingly.

Functionalities

Overview

All of the functionalities / constraints specified in the project task are implemented. In addition, some extra features are also supported:

- Edit the username of an user;
- Remove an user;
- Retrieve / display the list of film(s) subordinated to a specific film;
- Filmography management for each film-related person: retrieve the list of film(s) and the corresponding roles for each film-related person;
- Crew members management for each film.

For details on how these features are implemented, please refer to the “Overview of Final Product” section.

Film Suggestions System

The film suggestions system is based on a content-based filtering strategy and it uses score system, where 10 is the best recommendation score **(rs)** for a film and 1 is the worst possible score (note that for **rs,** decimal values are possible). With exception of special cases, the system would give 3 film suggestions for an user. The system is implemented as a function and the results are not stored in a table. The reasoning is as follows: the results are just too dynamic. For any kind of change in other tables such as *film, filmPersonLink, filmUserLink,* etc, the film suggestions for each user might require changes. In addition, storing the suggestions in a table means randomization would not be possible, and this would make the suggestions system too boring. Hence, the approach of only determining the film suggestions when asked is preferred.

Now, assume that we want to give a list of film suggestions for user *x*. Then, the **rs** of each unrated film is obtained by comparing them with each rated films by *x*. Specifically, these values would determine the output:

(1) the *genres* and crew members (list of film-related persons participating in the film) of all films

(2) parent-subordinated relations among all films

(3) the *filmTotalRatings and numberOfRaters* of all films that are not yet rated by *x*. These values would be used as the baseline for the **rs** for each unrated film.

(4) the *filmRating* of *filmUserLink* for all films rated by *x*.

Now, assume that during the process of finding the film suggestions for *x*, the algorithm is making a comparison between an unrated film *y* and a rated film *z.*

(1) and (2) are the values that are compared between *y* and *z*, in which when there is a match, the recommendation score for *y* would be altered based on (3) and (4). Note that match(es) in different attributes have their own specific weights. For example, a match in genre would affect the recommendation score more than a match in a crew member. The details of the implementation can be found in next pages.

Initial Algorithm

Like before, assume that we want to give a list of film suggestions for user *x*.

There are some special cases that need to be addressed:

- *x* has rated no film yet. In this case, the result would be based solely on each film's overall rating. Films with best overall rating are considered better and the film suggestions list for *x* would consist of 3 films with the highest average ratings.
- *x* has rated all films. In this case, there would be no film suggestion for x.
- *x* has rated all films - n, where n is an integer smaller or equal than 3. In this case, the film suggestions list for *x* would consist of the remaining unrated film(s) by x.

General cases:

1. Get the list of IDs of films rated by x and their film ratings by x, where the information is stored in a separate arrays for easier data access. Retrieval of these information are done via cursors since the arrays need to be updated in a row by row manner (or otherwise two queries are needed instead of one).
2. Get the detailed information of the rated films.
3. Get the detailed information of the unrated films that fulfill these conditions: the overall rating of the film is at least 6 or the number of rater(s) is lower than 5. Films with overall rating below 6 and has high enough number of raters are considered bad films and should not be suggested whenever possible.
4. For each unrated films *u*:
   4.1. Set the initial value for recommendation score of *u* (**rsu**) to *u*'s *filmTotalRatings* multiplied by 2. If *u* is never rated before, set **rsu** = 12.
   4.2. Set a variable divider = 2. This variable is used to give weight (importance) representation for each attributes and to transform the final recommendation score into a score that satisfies the scoring system.

4.2.1. Iterate over the list of rated films to compare them with *u.* For each rated films *r:*

- In case of a matching attribute between *u* and *r,* update the values as follow:
  - in case of a matching genre:
    - **rsu** = prev_rsu + (*filmTotalRatings* of *r)**2; divider = divider + 2
  - ~ note that in case there are *k* times matching of genres between y and z, **rsu** and divider will be updated *k* times, with the same algorithm as above.
  - in case of matching crew member(s):
    - if there is 1 matching crew member, do:
      - **rsu** = prev_rsu + *filmTotalRatings* of *r;* divider = divider + 1
    - 2 or 3 matching crew members:
      - **rsu** = prev_rsu + (*filmTotalRatings* of *r)**2; divider = divider + 2
    - more than 3 matching crew members
      - **rsu** = prev_rsu + (*filmTotalRatings* of *r)**3; divider = divider + 3
  - in case *u* and *r* is connected by the subordination relation (note that grandparent, grandchildren also counts):
    - **rsu** = prev_rsu + (*filmTotalRatings* of *r)**3; divider = divider + 3
- ~ The number of matching genre(s) / crew member(s) can be found as follows:
  - ~ store the information to be compared in separate arrays
  - ~ unnest the arrays and find the intersections
  - ~ use the array constructor syntax to reconvert it back to array, and use cardinality array function to find the total number of elements.

4.3. After iterating over all of the rated films, do the final calculation for **rsu:**

4.3.1. final **rsu = rsu /** divider, and the store the result in an array

5. Return the 3 films with the highest recommendation systems by using ORDER BY and LIMIT clauses. Note that the recommendation scores themselves are not stored / displayed.

Improvement Over the Algorithm

Previous algorithm work fine and gives a logically acceptable list of film suggestion(s). However, there is an issue: the algorithm has poor time complexity as it has to make too many comparisons between rated and unrated films (polynomial time complexity). This is not scalable as the number of films increases. In addition, a fully deterministic suggestion algorithm is not really fun. Therefore, to solve the scalability issue and also introduce some randomization, this following simple idea is implemented: limit the number of comparisons by exiting the iteration when

divider reaches a certain value as it is assumed that the algorithm has enough data to make a judgment for the recommendation score of a particular unrated film. For this project implementation, the threshold is set to 50. To make the comparison random for each time the function is called, there is a simple solution: “shuffle” the array of the rated films. This can be done by these following steps:

1. unnest the rated films array
2. randomize the row orders by using ORDER BY clause with random()
3. aggregrate the result with array_agg() and put it back into the rated films array

Now, it should be noted that theoretically, the worst-case time complexity is still polynomial (in case when each film is disjoint and has no relation with all other films). However, this case can be safely ignored as in real-life implementation, it does not make sense to have a very large set of completely disjoint films, and the time performance by limiting the value of divider should improve significantly in almost all cases when the dataset of films are sufficiently large.

With the above modifications, the film suggestions system should also perform efficiently timewise when dealing with a large dataset. Note that the algorithm is only semi-randomized. When the number of films is sufficiently small, the output of the film suggestions for each user would remain the same as long as the relevant attributes/values are not altered.

Client/Server Implementation

This section will cover only the fundamental aspects of the Client/Server implementation as it is not the main focus of this project.

1. Model

All tables used in the database, except the supporting tables (*country, genres, roles)*, have their equivalent model class. In this way, uniformity between server and database is achieved, which enables simple and orderly data exchange.

All of the functionalities (e.g. adding a film, retrieving all films, etc) are done by using Npgsql to either make a very simple query command or by calling the stored functions of the database. The return values of these functions then are the query result that are parsed and read by NpgsqlDataReader.

2. View & Controller

There are 7 pages used in total:

Path	Description
/Home	index page. Consists of background information regarding the project.
/About	contains the project's author information (name, course of study, and matriculation number)
/Home/Films	display the list of all root films
/Home/Films?all=true	display the list of all films
/Home/Films?subordinatedTo={id}	display the list of films subordinated to film with id={id}
/Home/Film?filmRec={id}	display the list of film recommendations for user with id={id}
/Home/FilmRelatedPersons	display the list of all film-related persons
/Home/FilmCrews	display all of the film & film-related person relations
/Home/FilmCrews?personId={id}	display the filmography of film-related person with id={id}
/Home/FilmCrews?filmId={id}	display the crew members of a film with id={id}
/Home/Users	display the list of all users
/Home/FilmRatings	display the list of all film ratings from all users
/Home/FilmRatings?userId={id}	display the list of all film ratings by the user with id={id}
/Home/FilmRatings?filmId={id}	display the list of all film ratings for the film with id={id}

The APIs for client-server interaction supports 4 basic operations as defined in CRUD convention. While the APIs are not fully RESTful, there are many concept of REST that are incorporated onto the designed APIs (client-server, stateless, uniform interface based on HTTP protocols, layered system). The web application heavily utilises href for linking pages among each other. A simple client-side data validation is implemented, in which an alert would be displayed in case of invalid form values / server error.

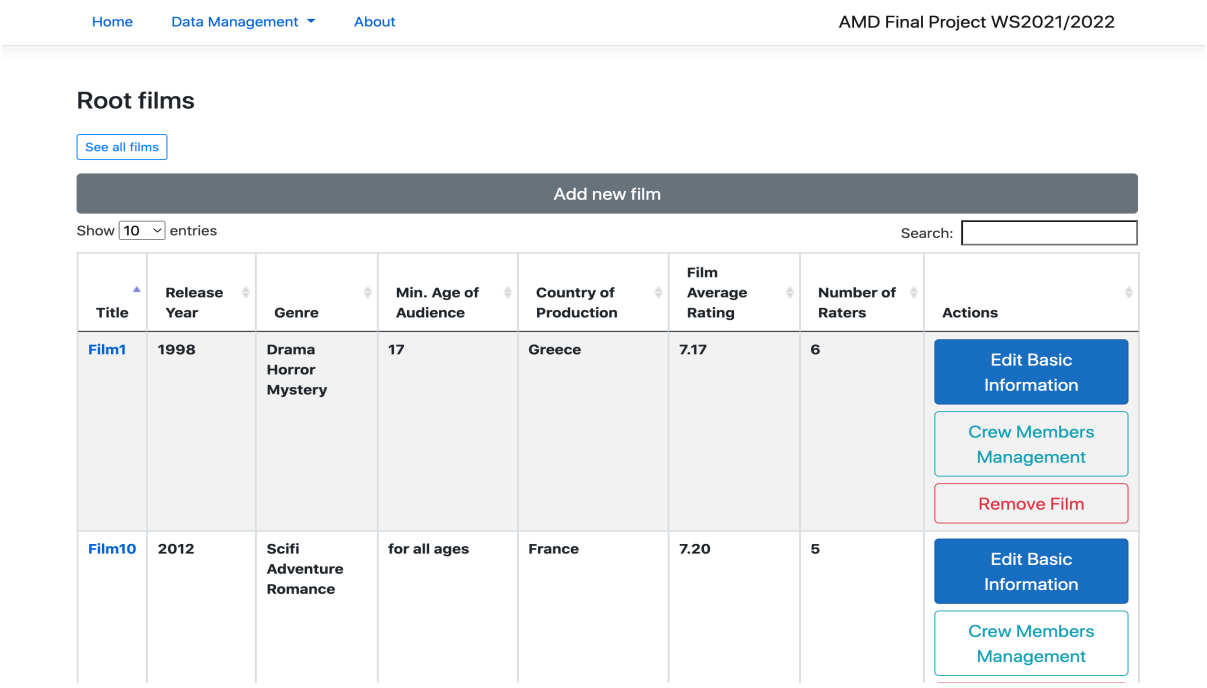
It should be noted that the code is written within a strict time constraint. Therefore, the code is not written with the readability aspect in mind, and the author only focused on making sure that the functionalities are completed without mistakes/bugs and with an intuitive user interface for a good user experience.

Overview of Final Product

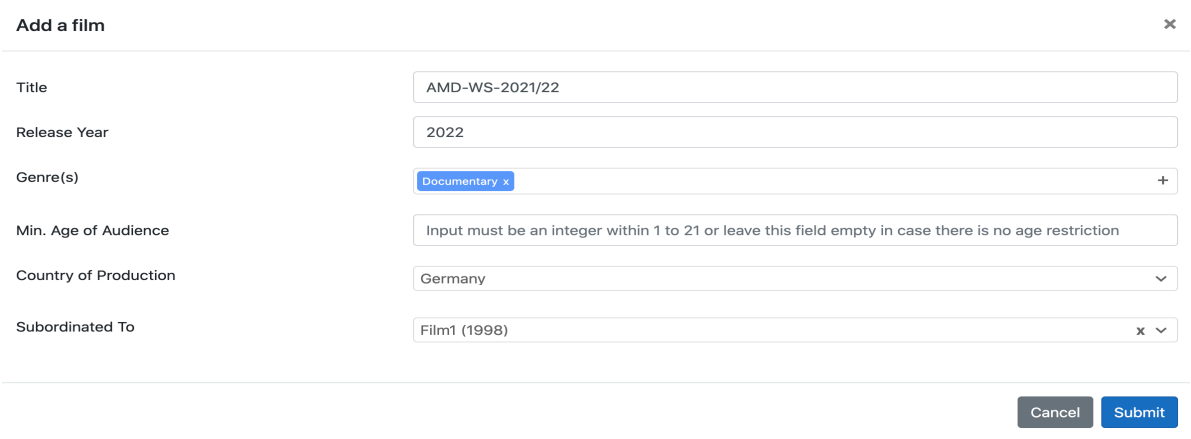
The web application is intended for administrative purposes. The person who has access to the application can do all of the supported operations.The dummy data used for testing is a self-created fictional data that satisfies the requirements as written in the project task.

A. Films

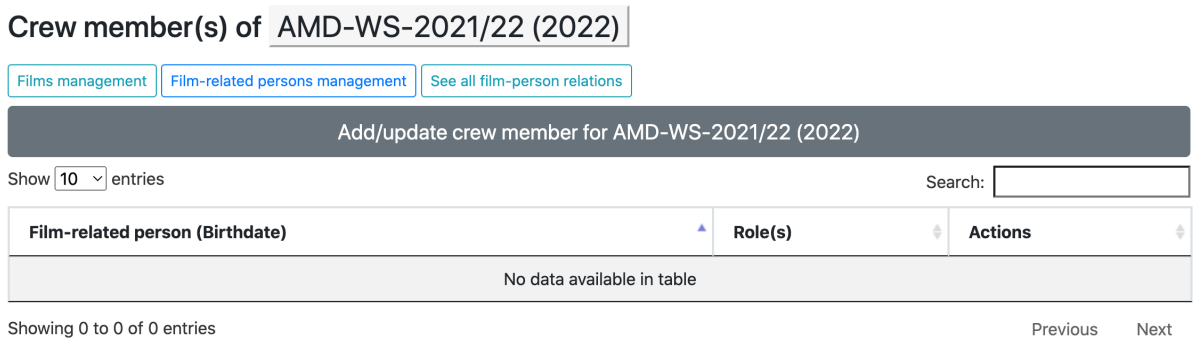
Clicking on Data Management -> Film will retrieve the list of all root films. Clicking on “See all films” button would show all of the films. Clicking on the film’s title hyperlink would show the list of subordinated film(s) to that particular film.



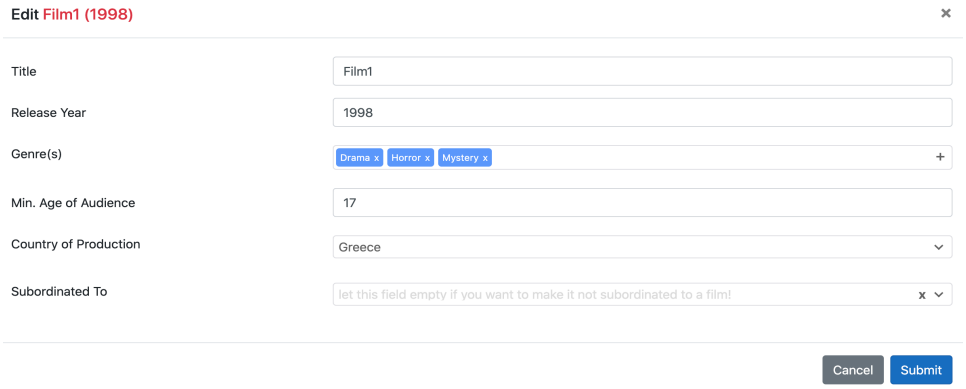
Adding a film.

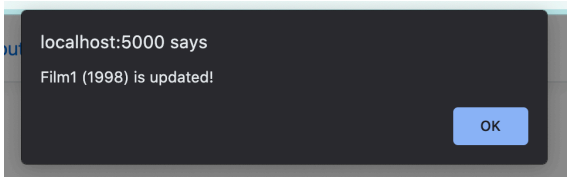


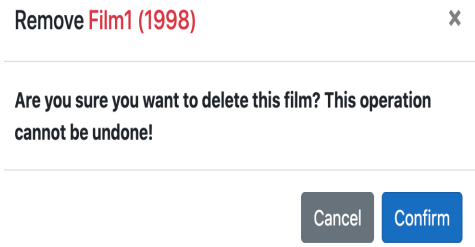
After successful creation of a film, user would get redirected into the crew members management for the newly created film.



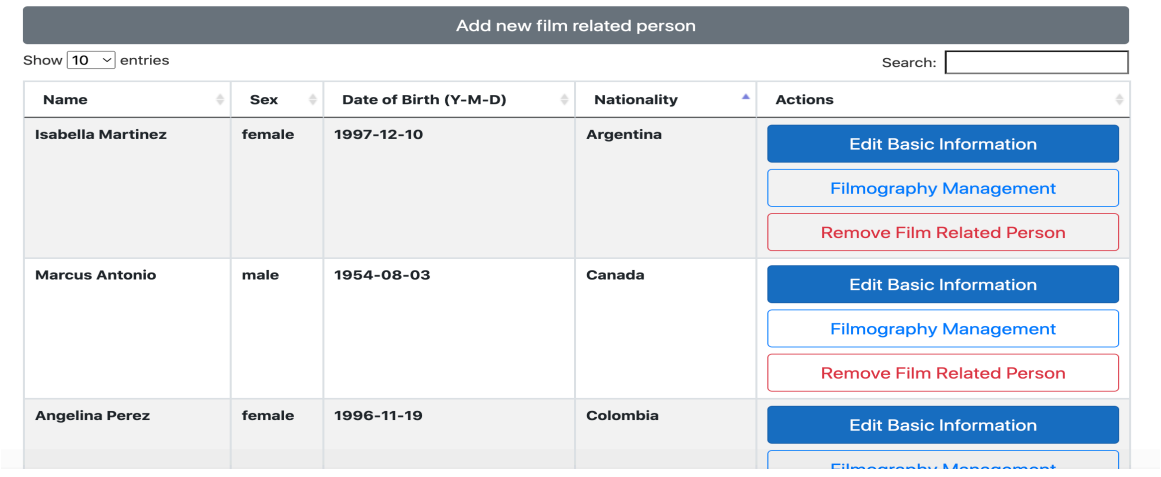
Editing & Removing a film:

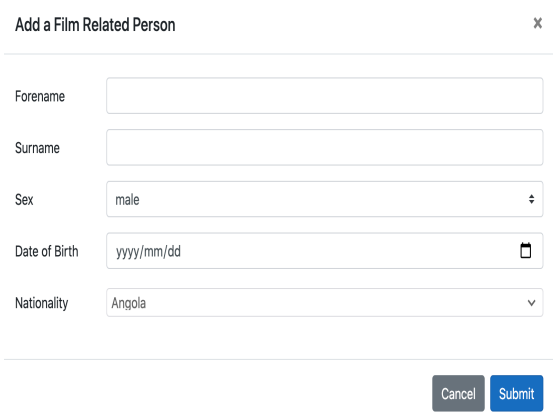


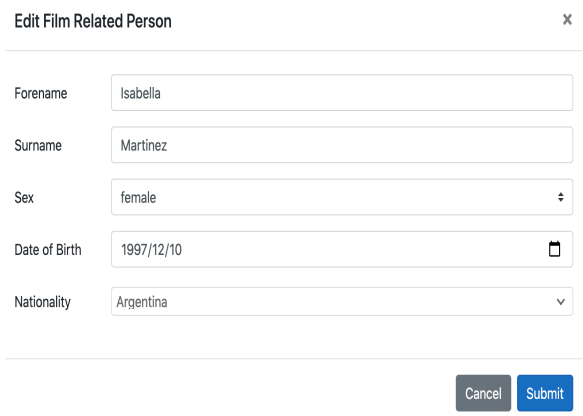




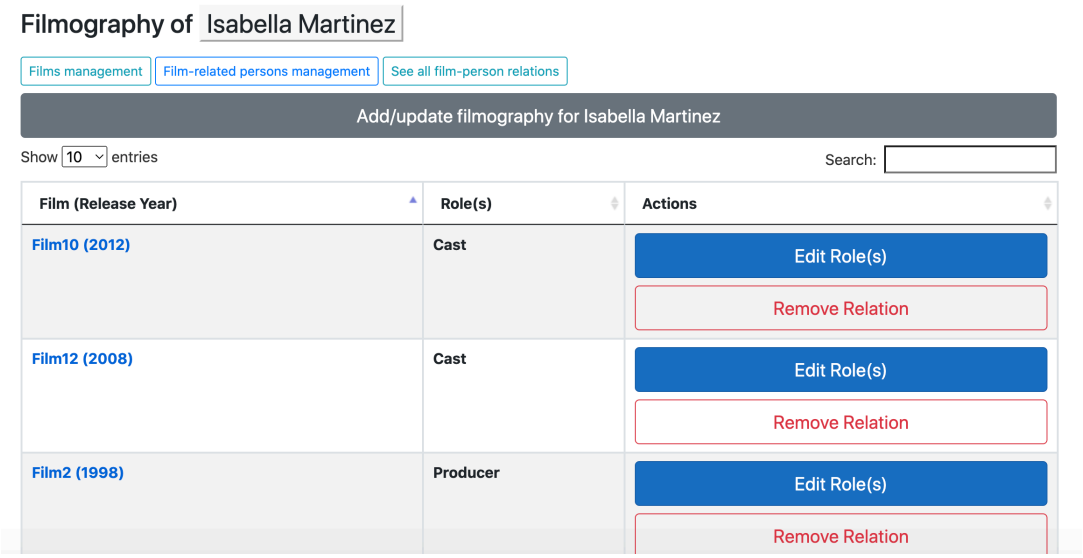
B. Film-related persons

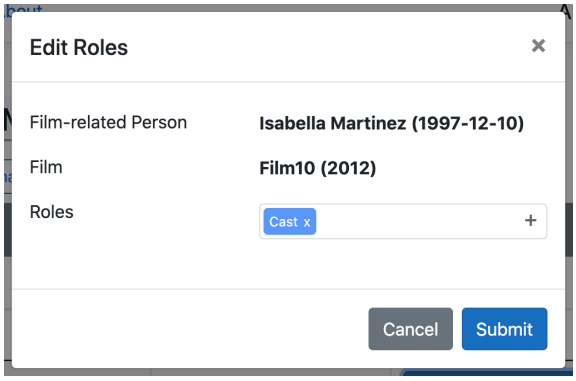


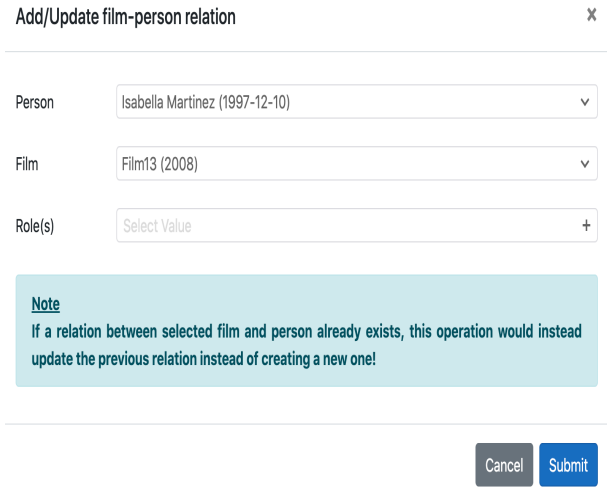




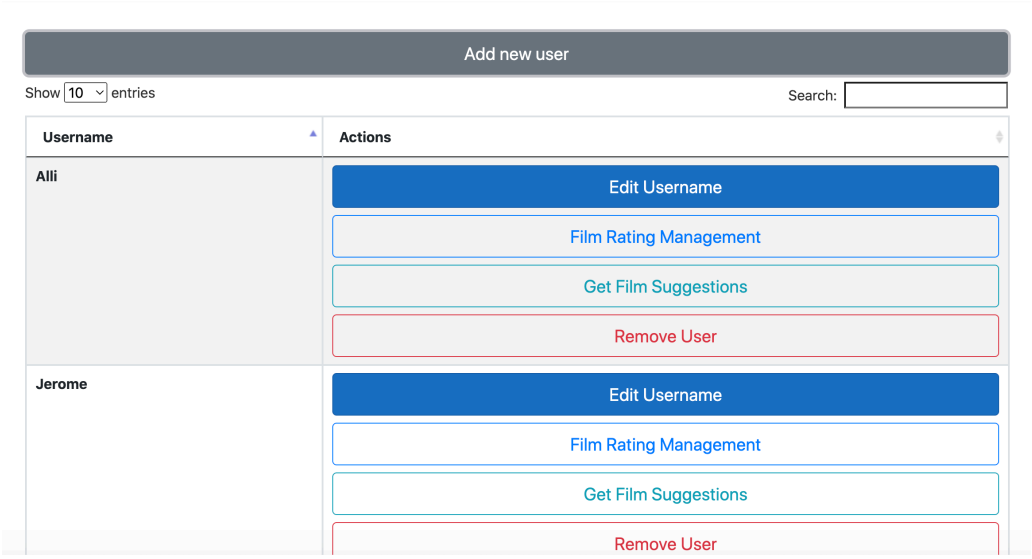
After clicking on “Filmography Management” of a film-related person:



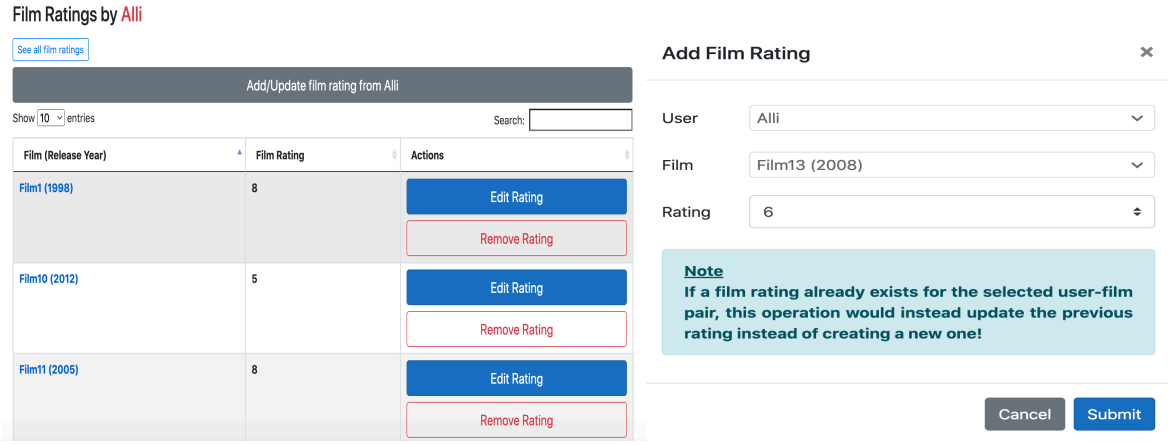




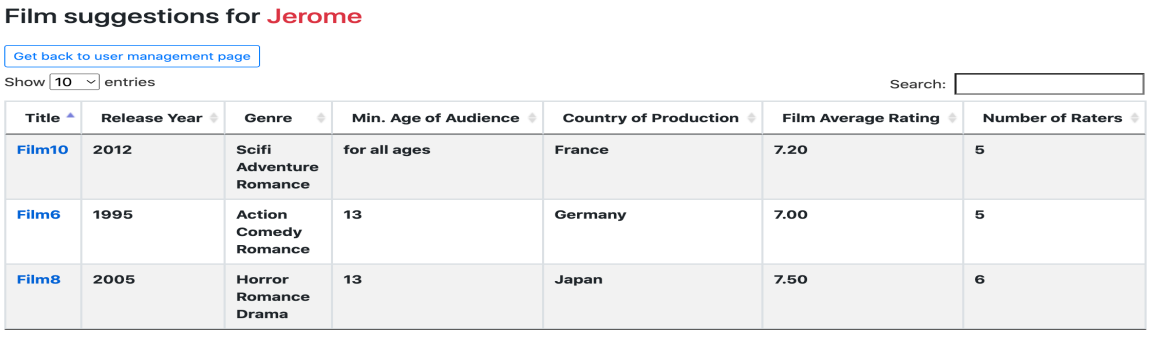
C. Users



Clicking “Film Rating Management” of an user:



Clicking “Get Film Suggestions” of an user:



Details of the implementations & other functionalities will be hopefully explained during the presentation session.

Distributed Databases Scenario

There are several conditions on when it would be beneficial to use distributed databases for the project. In this section, only one scenario will be discussed. In the following scenario, it is assumed that the users do not have admin privileges: they can only rate a film and get a film recommendation.

Scenario

The application is used by large number of users across continents. Users outside of Western Europe are unhappy with the significant network latency that they encountered while using the application.

Solution

Based on above scenario, the database should be replicated across regions (geo-distributed databases). As the geographical distance between users and their designated databases decreases, the propagation delay would also decreases, which would improve the performance of the application.

While the tables concerning films and film-related persons would have to be fully replicated, horizontal fragmentation can be applied to user-related tables to avoid unnecessary data redundancy. Additional attribute *userRegion* have to be added to the tables *user* and *filmUserLink.* Then, sharding for those tables would be based on that attribute. Each database would have different entries for its tables *user* and *filmUserLink,* based on its designated region.

Unfortunately, this implementation has one glaring disadvantage: any update on a film's overall rating (and number of raters) has to be broadcasted to all databases in the system, which can be quite time-consuming and requires significant bandwidth. One way to mitigate the problems is by choosing to do the updates asynchronously with scheduling. For example, updates on Database A (which is logged) would be propagated to all other databases for every 1 minute. Then, based on the data received, all other databases should update the relevant attributes accordingly. We do this async updates propagation with all other databases. Note that we can choose to do the updates asynchronously since the application itself does not deal with highly sensitive data. In addition, the film suggestions system is supposed to be randomized, and therefore it would not be a problem if the database is not fully up-to-date for several minutes / hours.

References

[1]. Documentations for used technologies

PostgreSQL: https://www.postgresql.org/docs/
Visual Studio: https://docs.microsoft.com/en-us/visualstudio
.NET Core: https://docs.microsoft.com/en-us/aspnet/core
Npgsql: https://www.npgsql.org
jQuery: https://jquery.com
Bootstrap: https://getbootstrap.com
DataTables: https://datatables.net
Slim Select: https://slimselectjs.com

[2]. Dr. Seifert's lecture slides on Conceptual & Logical Database Design and Concept of Distributed Databases

[3]. UUID data type: https://www.postgresqltutorial.com/postgresql-uuid/

[4]. Additional information regarding replication in distributed databases: http://www.inf.fu-berlin.de/lehre/SS10/DBS-TA/folien/09-10-TA-Repl-1-1.pdf