Executive Summary

Below is a summary of the material changes to the app since day one based on feedback and independent learning, with supporting commentary on why we took these actions. It's broken down into a few major themes – scope, code optimization, and user experience.

As you can see in the project outline below, our original scope included building a full-stack user-facing web app. However, this was too optimistic for a 10-week introductory database course. As a result, we elected to limit this work to phase one: a backend database which supports the frontend and leverages CRUD (Create Read Update Delete) actions. Then, after this class we could expand on this work (if desired) with phase two: a frontend user-facing web app, and phase three: a mobile app. Even then feedback suggested the phase one scope was too optimistic since there were many tables and interactions. As a result, we removed the streaming services tables from the project scope and instead added it to phase two.

Most of the updates and new features since day one was to the app's code. Since we were still relatively new to full-stack design, we began very early with a sample Flask app provided in class to learn how the html/j2 pages interacted with the app file running the routes and the SQL queries. We stripped away items that weren't necessary, added many new features, and in the end only about 30% of the original coding framework remains. The first of these changes was to update the table column labels to use common names (as feedback suggested). We elected to do this in the html/j2 files since we only wanted to change the presented label, not the table name used elsewhere. This allowed us to better understand how to embed python syntax in the html using jinja.

The biggest change, initially inspired by a simple suggestion from a reviewer, was an overhaul to our html/j2 files so they could use templating for the navigation link and table rendering. By building upon the sample app, we had a large amount of repeated html code. Implementing these changes was a "eureka" moment because they were easy to implement and streamlined our code significantly. Now we can update column labels and navigation links in one place (and have wide impact), which is easier and has much higher quality. In reflection, first producing inefficient code was important because only then could we critically analyze if, how, and why we should templatize the code. It seems the perfect segway into phase two design.

The rest of the major changes were sought to improve user experience. The first of these enhancements was to add a movie search filter that allowed the user to search a substring within a string. Next, we added a movie summary page which allows the user to select a movie and see all the relevant data for it. Several reviewers suggested this would help tie the data in the app together. Additionally, it's the bridge to the front-end app build in phase two. We also added error handling for all SQL edit and delete queries. Now, all errors will point to a shared error page which presents users a custom message based on the error. This is helpful for instances where the user tries to add a value (or combination of values) that already exists in the table, or engages a restricted cascade delete (several exist to ensure data integrity). While first seeming to be a "nice to have" feature, it now seems a key piece of the app that helps the user have better clarity to made effective decisions when using it. Lastly, we updated director as a nullable in the movies table. This allowed use to explore the impact of inserting and updating a record that has a null foreign key. However, all movies should have a director so in phase two, we'd seek to add more fields that aren't as critical to the app and could potentially be nullable. Overall, we're satisfied with the final product and the learnings gained help us see potential for new ideas in later phases and other projects.

Project Outline

In this project, we built a backend app for adding, removing, and editing records in a database that supports a frontend app called *Movie Night*. *Movie Night* simulates a movie review platform where users browse and add reviews for movies. The app will allow users to filter on genre, director, actor, minimum rating value and rating count, then see the movies that fit these criteria. While the prototype is just sample of movies, a production version would store in a database all movies available at some point (approximately 613k currently [1]) in theatres, on television, available in DVD format, or available for streaming online. This database will not include videos made available only for platforms such as YouTube or those less than 75 minutes in length.

After this class, we could build a front-end app (phase 2) and eventually a mobile app (phase 3) if desired. Mobile would allow users to quickly open their phone when choosing what to do tonight and possibly make it a *Movie Night* (a possible slogan). Future phases may also include displaying where the movie can be streamed online. IMDB has about 83 million users [2]. If this app filled a particular niche in the market, it's possible this would be the reasonable upper bound of users that would use it. Given *Movie Night* is a new app, however, achieving 5 million users over the first few years would be a great accomplishment. Further research would be needed to evaluate how many user reviews might be generated on a database of millions of users. This would be the maximum size of the database needed to accommodate the app.

The product of this project serves as a database tool that implements full CRUD (Create Read Update Delete) functionality on multiple tables. The entities include movies, actors, genres, directors, and users, with intersection tables to demonstrate the relationships between movies and their actors, movies and their genres, and users and their reviews for a movie. From the homepage, users can pull a page that summarizes all the data for an individual movie, including the related actors, genres, and its ratings. There are separate pages for the movies, actors, directors, and genres for all the movies entered in the database, users, the users that have entered a review into the database, and the intersection tables. All these tables can be accessed via the navigation bar at the top of the website. The app communicates with a database that stores the data. For this class, the database was hosted on a server at Oregon State University (flip3). After the class ends, the database will likely be hosted in MongoDB.

Database Outline

- **movies:** This is the list of movies. The combination of movies plus year is unique (since some movies are remade under the same name).
    - movie_id: int, auto increment, not NULL, PK
    - movie_name: varchar(250), unique, not NULL
    - movie_year_released: year(4), not NULL
    - movie_language: varchar(250), not NULL
    - director_id: int, NULL, FK

    Relationships:
    - A 1:M required relationship between movies and movies_genres with movie_id as a FK within movies_genres.
    - A 1:M required relationship between movies and movies_actors with movie_id as a FK within movies_actors.
    - A 1:M optional relationship between movies and users_reviews with movie_id as a FK within users_reviews.
    - A M:1 optional relationship between movies and directors with director_id as a FK within movies.

- **actors:** This the list of actors associated with the movies in the database.
    - actor_id: int, auto increment, not NULL, PK
    - actor_name: varchar(250), unique, not NULL

    Relationship: A 1:M required relationship between actors and movies_actors with actor_id as a FK within movies_actors.

- **directors:** This the list of directors associated with the movies in the database.
    - director_id: int, auto increment, not NULL, PK
    - director_name: varchar(250), unique, not NULL

    Relationship: A 1:M required relationship between directors and movies with director_id as a FK within movies.

- **genres:** This the list of all major movie genres.
    - genre_id: int, auto increment, not NULL, PK
    - genre_name: varchar(250), unique, not NULL

    Relationship: A 1:M required relationship between genres and movies_genres with genre_id as a FK within movies_genres.

- **users:** This the list of all users. Someone can be a user without providing reviews.
    - user_id: int, auto increment, not NULL, PK
    - user_first_name: varchar(250), not NULL
    - user_last_name: varchar(250), not NULL
    - user_email: varchar(250), unique, not NULL
    - The combination of user_first_name and user_last_name is unique.

Relationships: A 1:M optional relationship between users and users_reviews with user_id as a FK within users_reviews.

- **movies_actors:** This is an intersection table to relate movies and actors.
    - movie_actor_id: int, auto increment, not NULL, PK
    - movie_id: int, not NULL, FK
    - actor_id: int, not NULL, FK
    - The combination of movie_id and actor_id is unique.

    Relationships: A M:M required relationship between movies and actors with movie_id and actor_id as FKs.

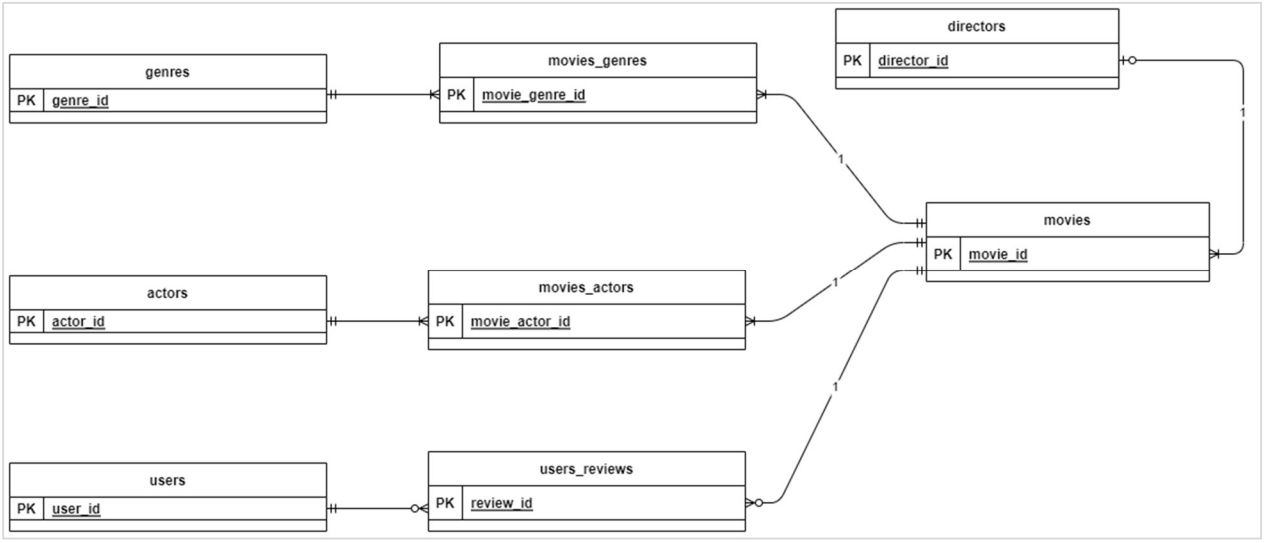
- **movies_genres:** This is an intersection table to relate movies and genres.
    - movie_genre_id: int, auto increment, not NULL, PK
    - movie_id: int, not NULL, FK
    - genre_id: int, not NULL, FK
    - The combination of movie_id and genre_id is unique.

    Relationship: A M:M required relationship between movies and genres with movie_id and genre_id as FKs.

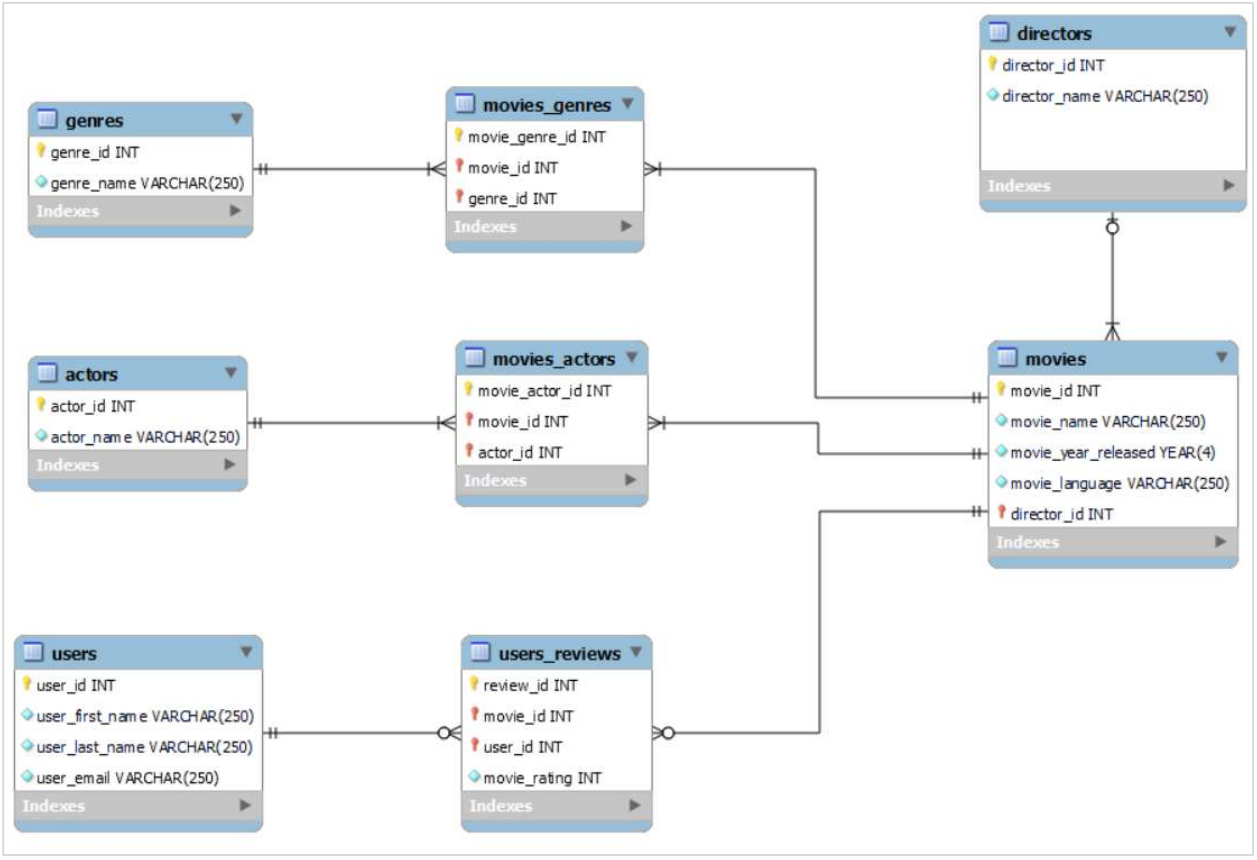
- **users_reviews:** This is an intersection table to relate users and movies, which also includes user reviews. Users will not rate all movies and can only have one active rating per movie (but this rating can be revised at later dates).
    - review_id: int, auto increment, not NULL, PK
    - movie_id: int, not NULL, FK
    - user_id: int, not NULL, FK
    - movie_rating: int, not NULL
    - The combination of movie_id and user_id is unique.

    Relationships: A M:M optional relationship between movies and users with movie_id and user_id as FKs.

Entity-Relationship Diagram (ERD)



Schema



Colors associated with the keys in the ERD diagram above:

- Gold = PK
- Red = FK

Sample Data

movies

movie_ID	movie_name	movie_year_released	movie_language	director_id
1	Life is Beautiful	1997	Italian	1
2	Pan's Labyrinth	2006	Spanish	2
3	Parasite	2019	Korean	3

actors

actor_id	actor_first_name
1	Roberto Benigni
2	Nicoletta Braschi
3	Giorgio Cantarini

directors

director_id	director_name
1	Roberto Benigni
2	Guillermo del Toro
3	Bong Joon Ho

genres

genre_id	genre_name
1	Action
2	Adventure
3	Animated

users

user_id	user_first_name	user_last_name	user_email
1	Frodo	Baggins	frodo_baggins@gmail.com
2	Bilbo	Baggins	bilbo_baggins@gmail.com
3	Pippin	Took	pippin_took@gmail.com

movies_actors

movie_actor_id	movie_id	actor_id
1	1	1
2	1	2
3	1	3

movies_genres

movie_genre_id	movie_ID	genre_id
1	1	4
2	1	5
3	1	10

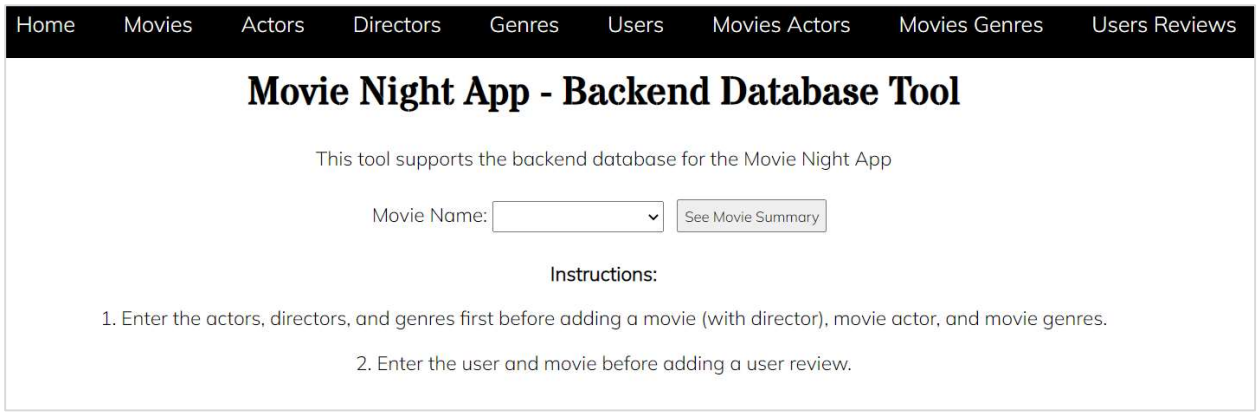
users_reviews

review_id	movie_id	user_id	movie_rating
1	7	1	4
2	6	1	5
3	1	2	5

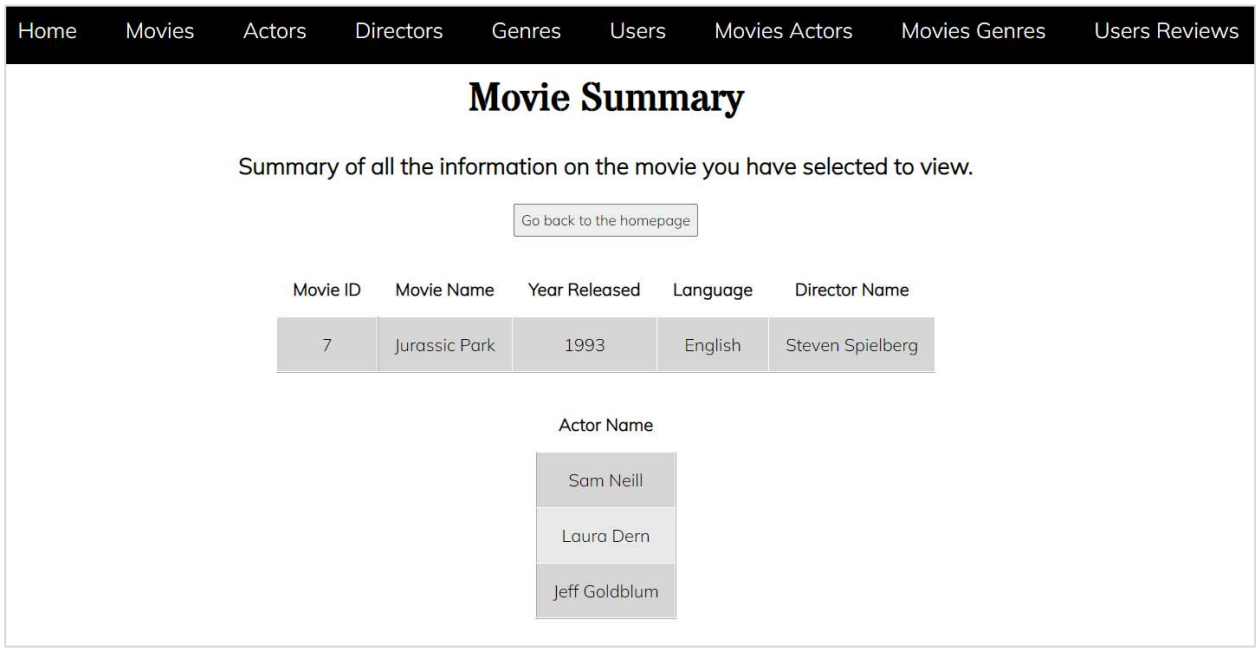
UI Screen Shots

Note: a slight outside grey border was added around the images in this section to help create clean separation, but the borders are not part of the UI.

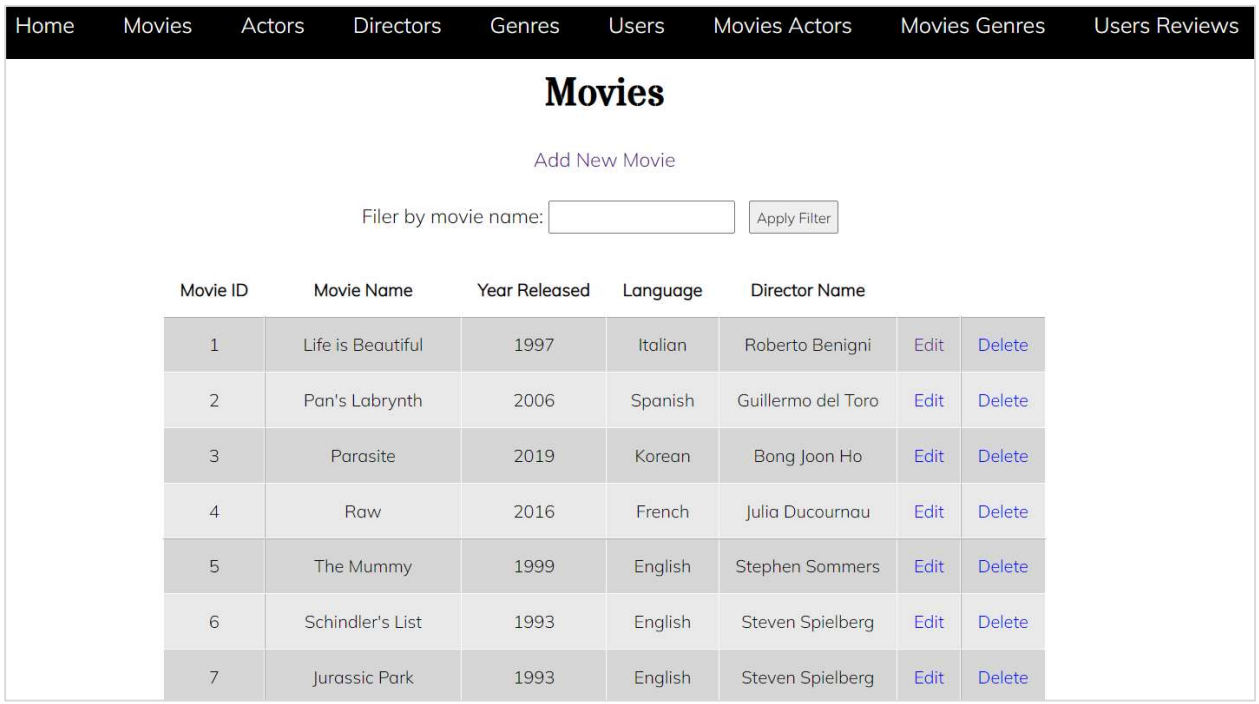
HOME Page:



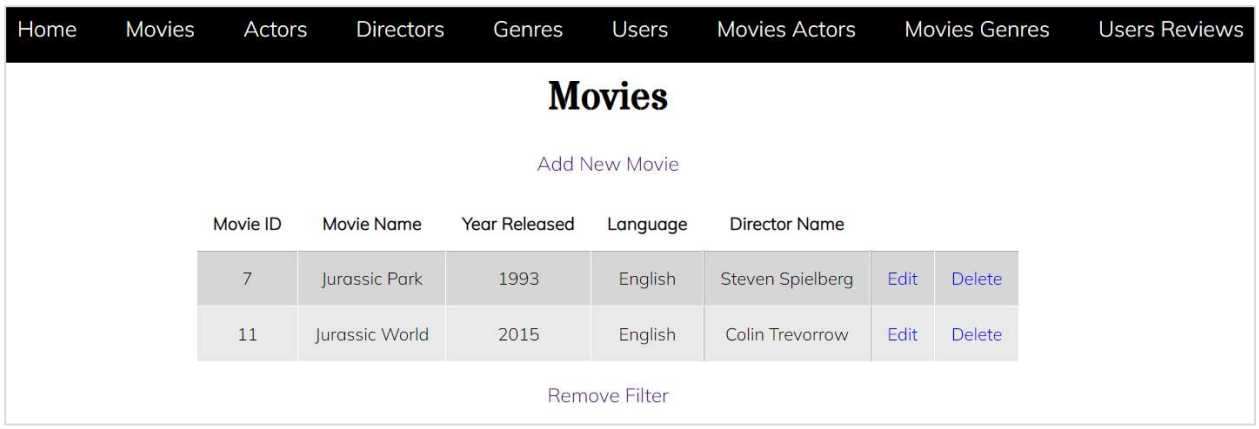
READ Movie Summary Page:



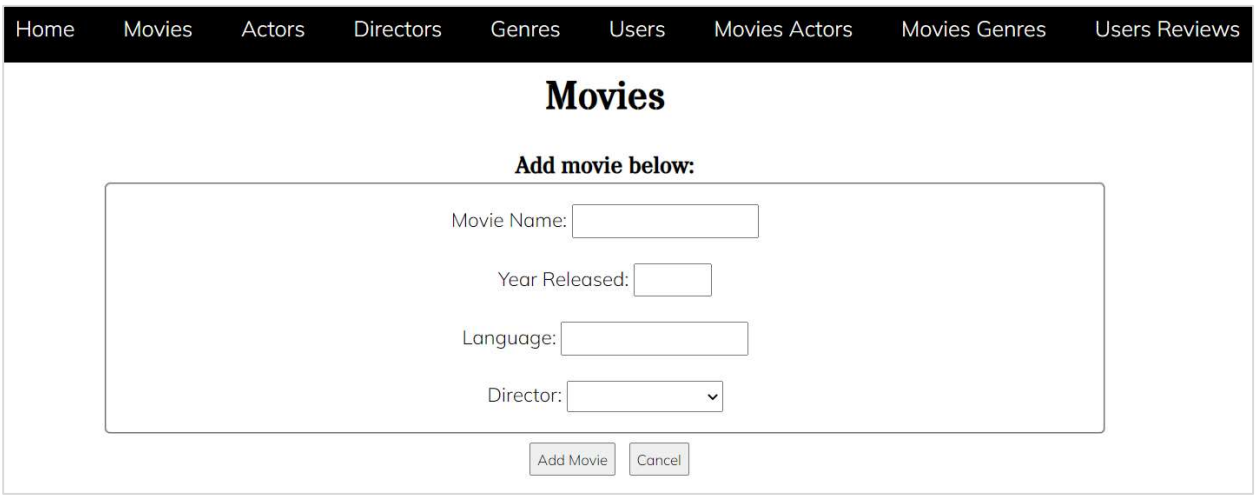
READ, DELETE Movies page:



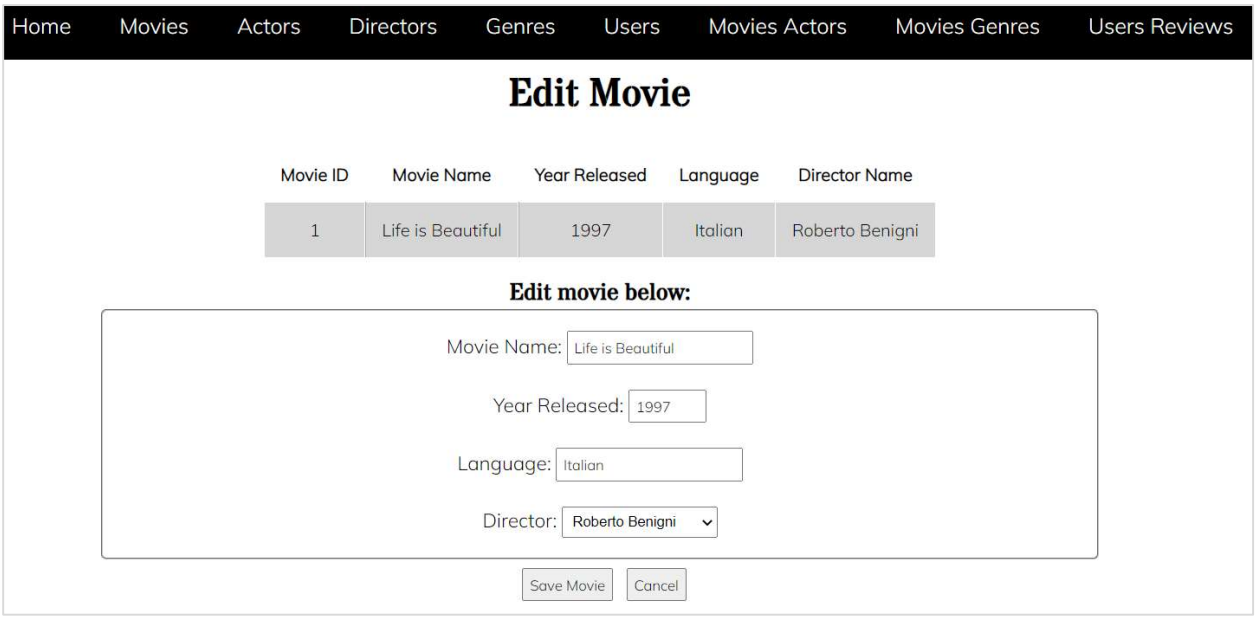
READ Filtered Movies page:



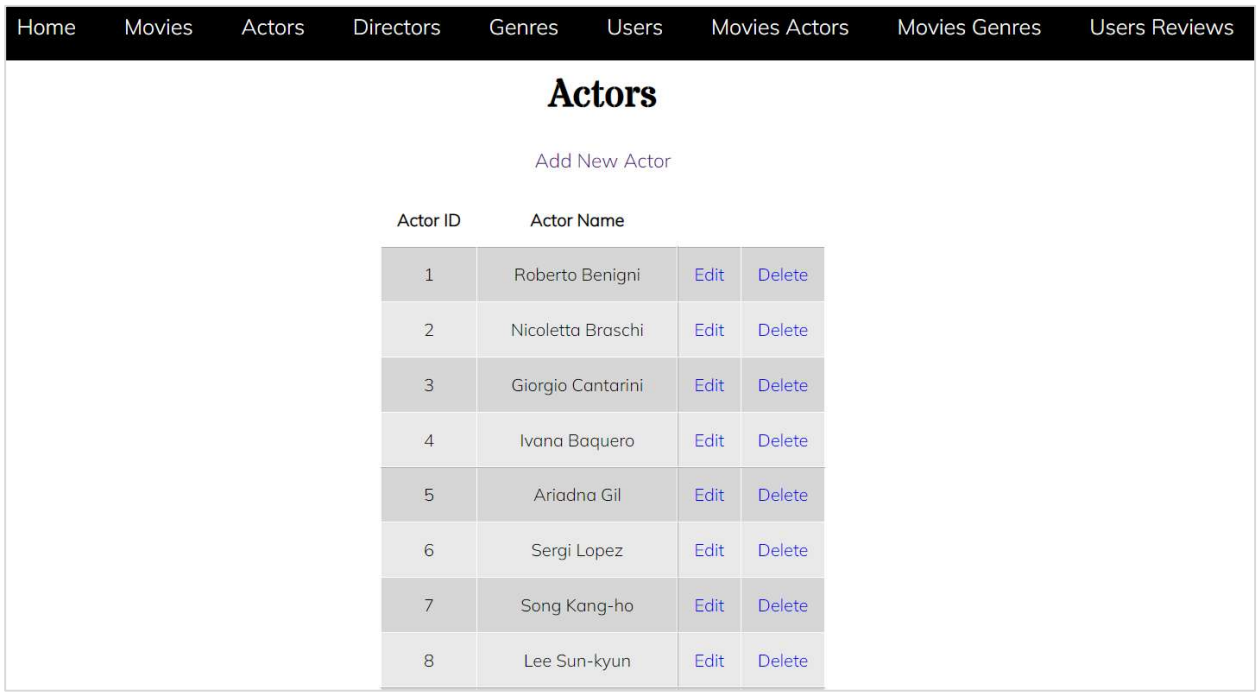
CREATE New Movie page:



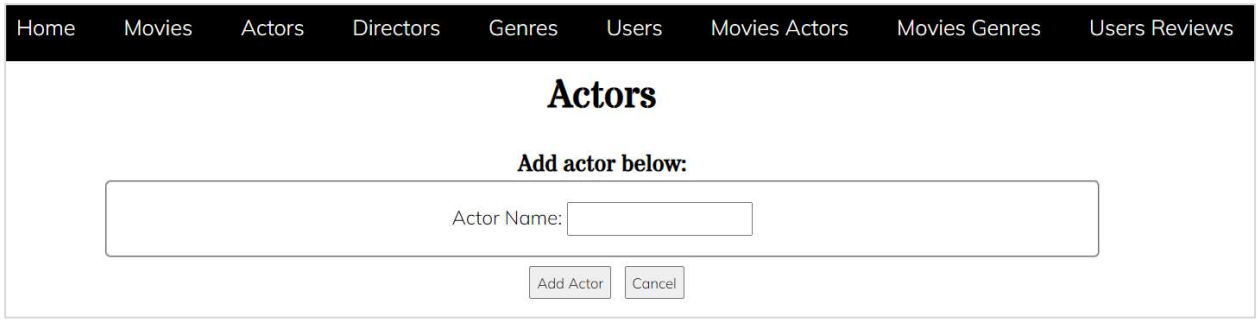
UPDATE Movie page:



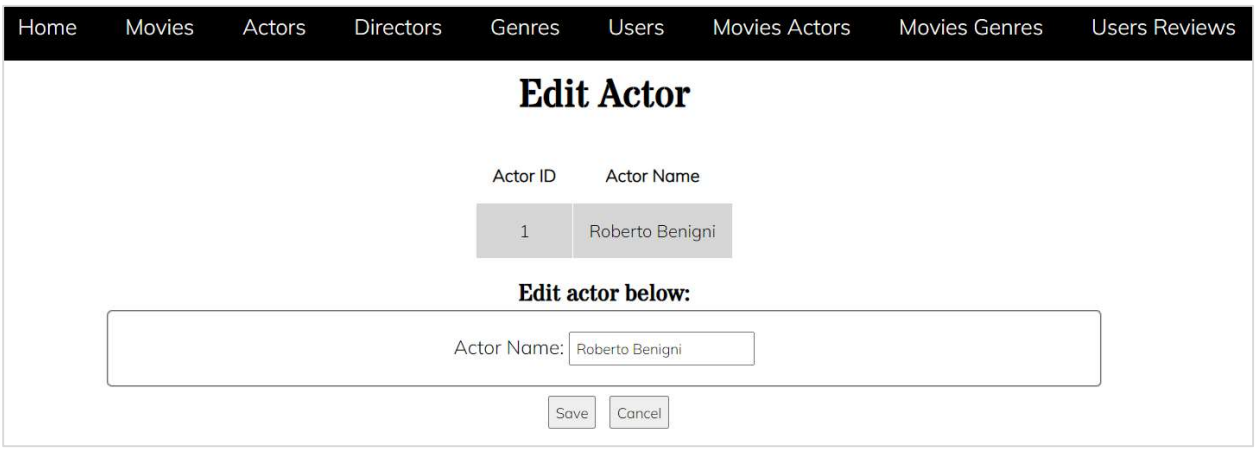
READ, DELETE Actors page:



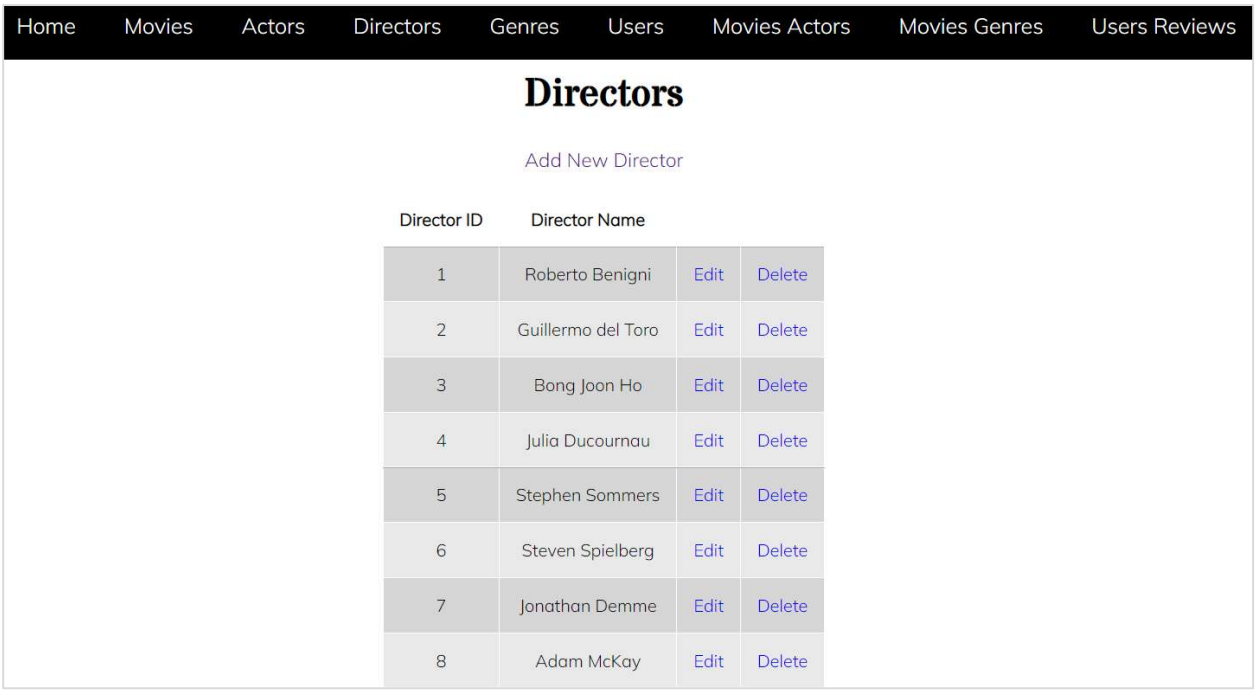
CREATE New Actor page:



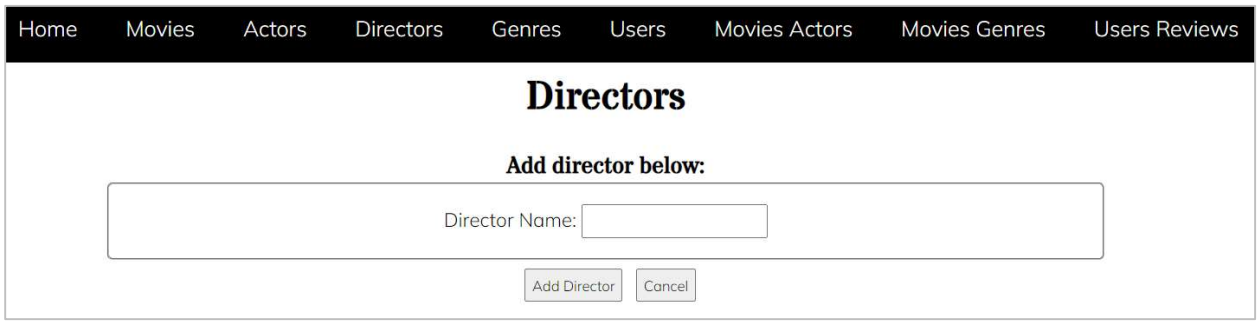
UPDATE Actor page:



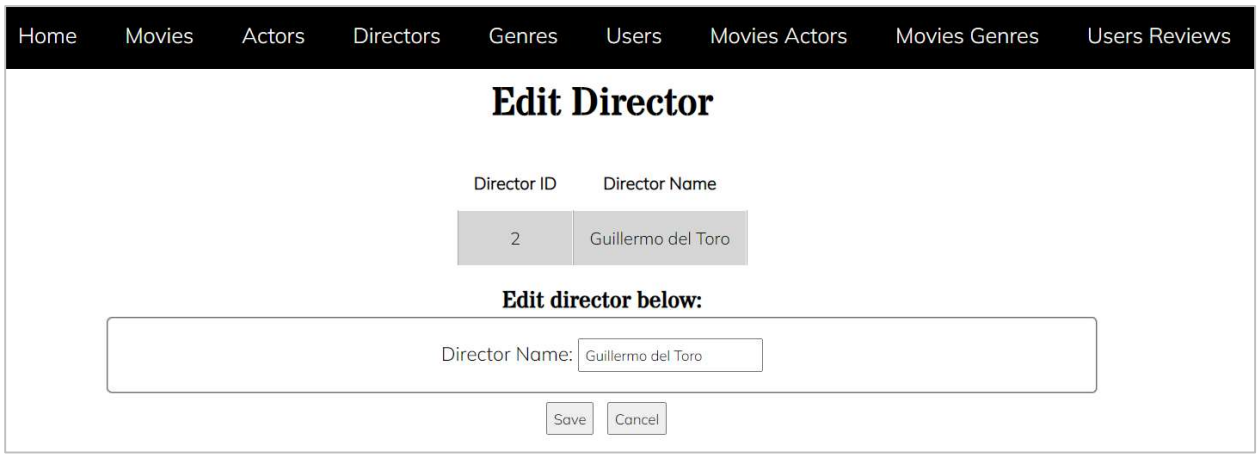
READ, DELETE Directors page



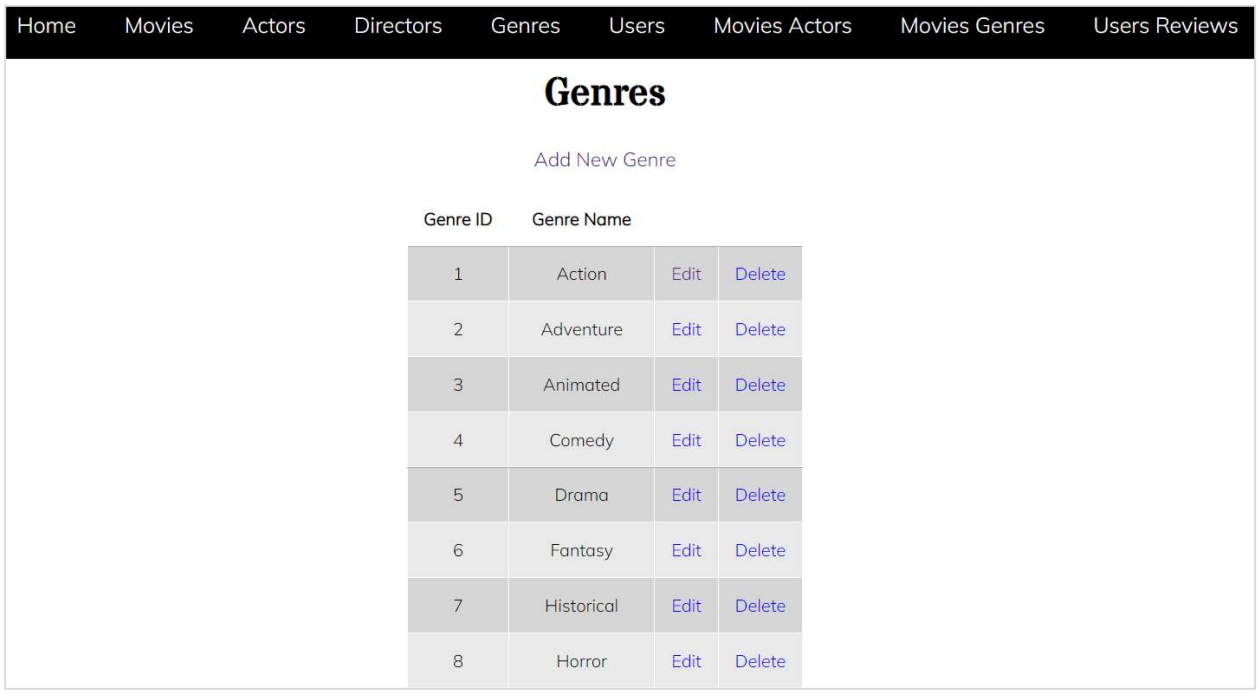
CREATE New Director page:



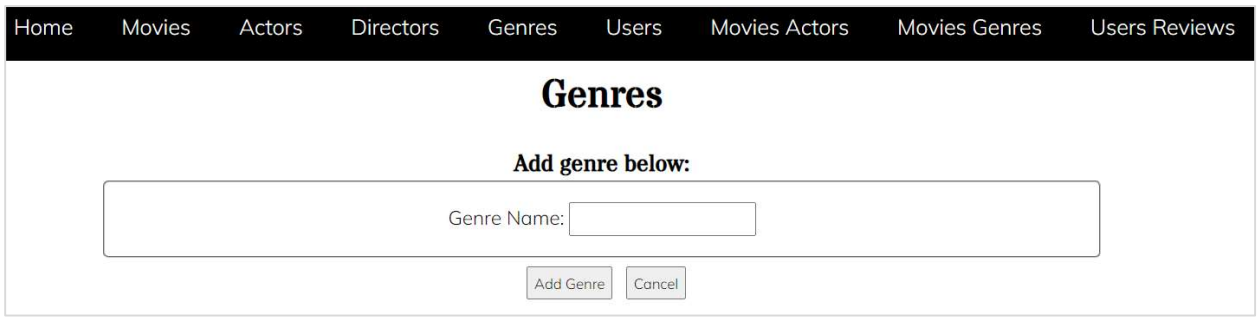
UPDATE Director page:



READ, DELETE Genres page:

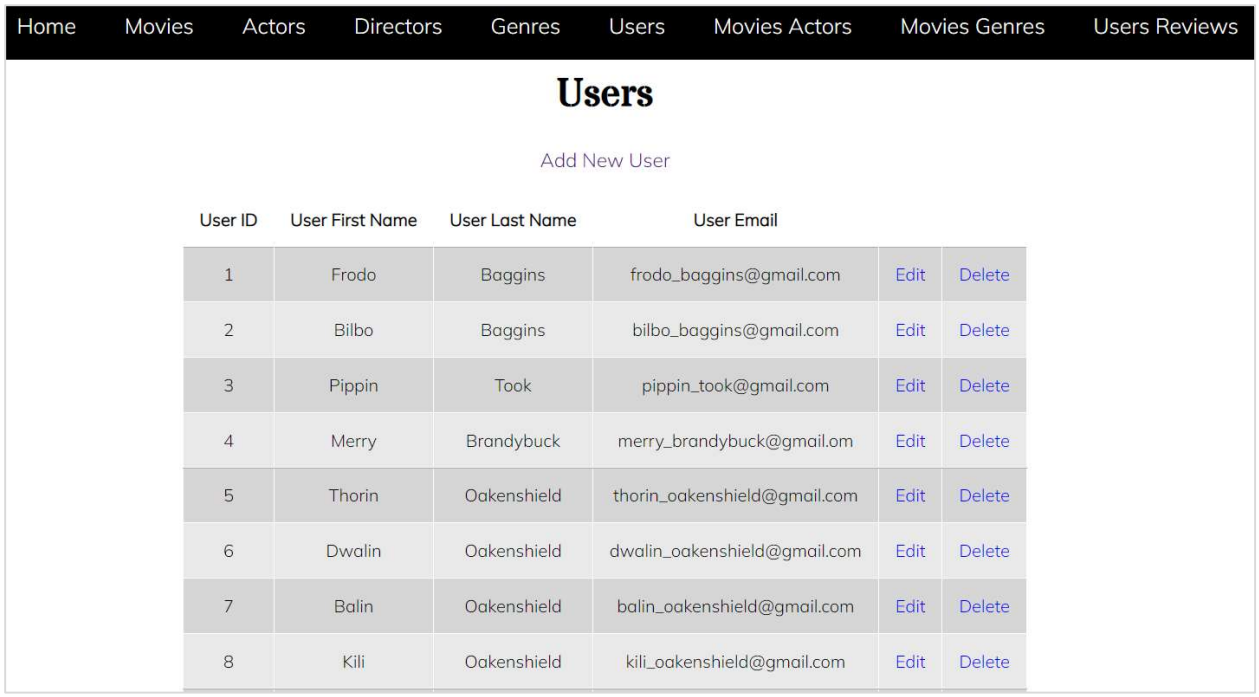


CREATE New Genre page:

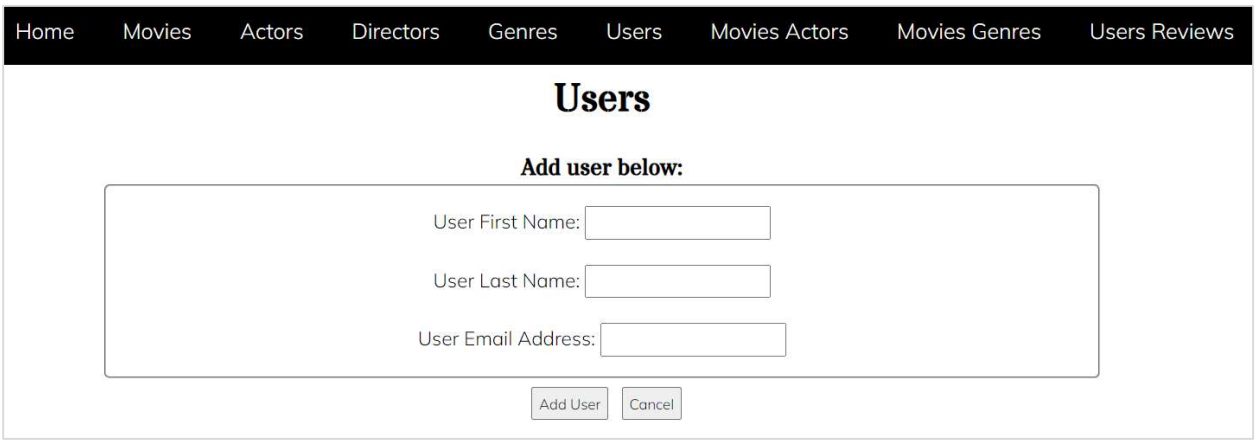


UPDATE Genre page:

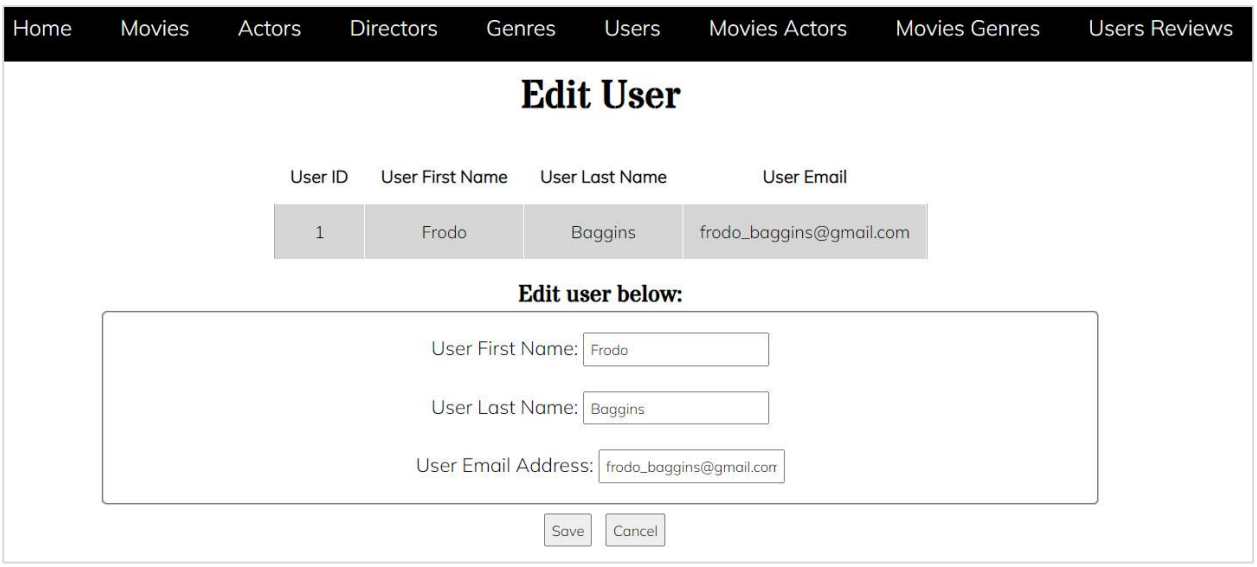
READ, DELETE Users page:



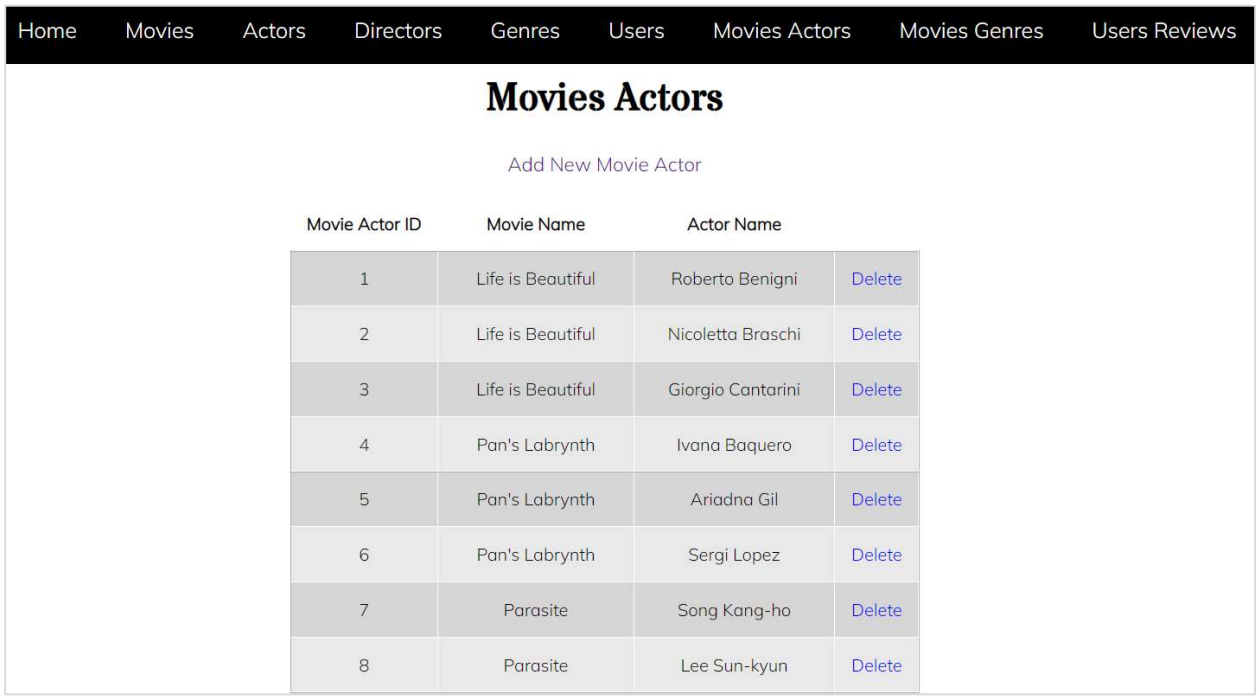
CREATE New User page:



UPDATE User page:

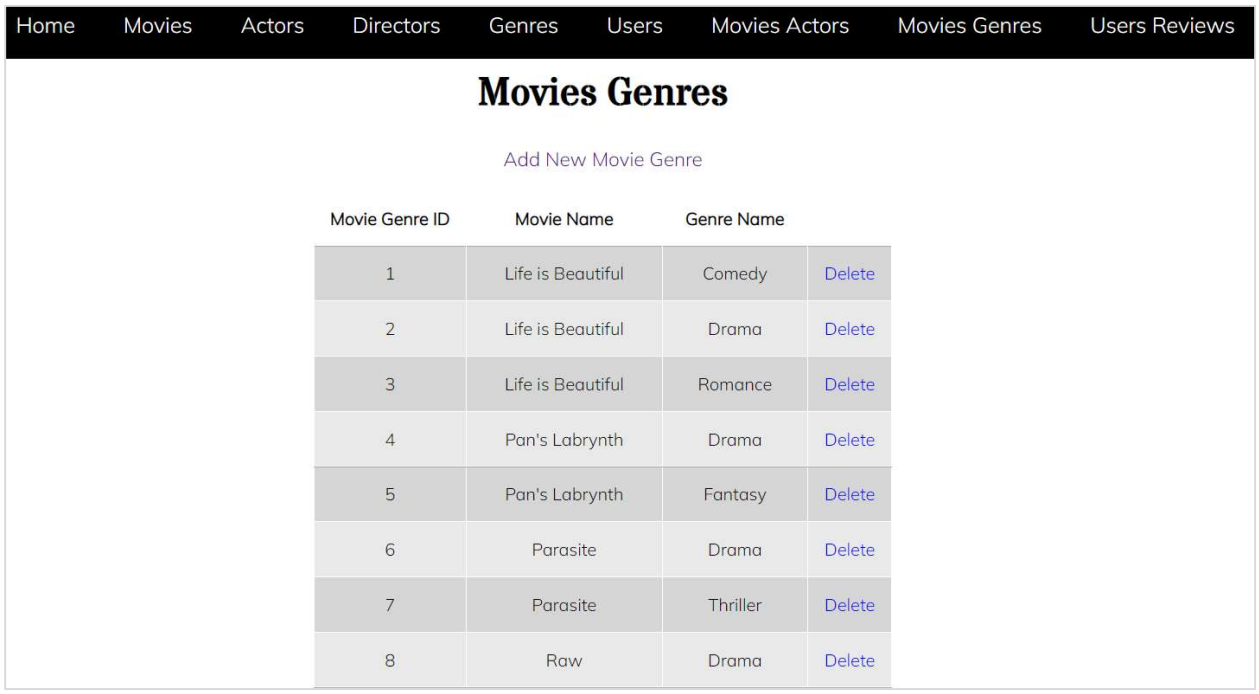


READ, DELETE Movies Actors page:

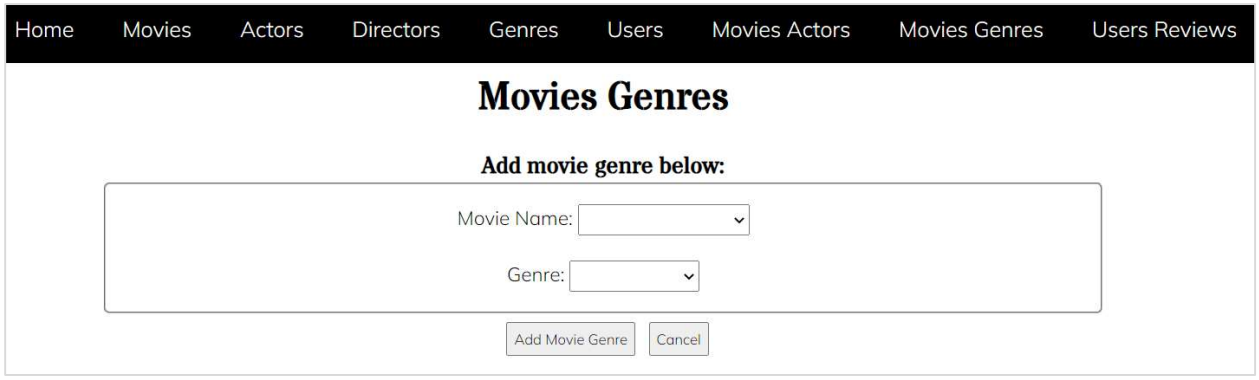


CREATE New Movie Actor page:

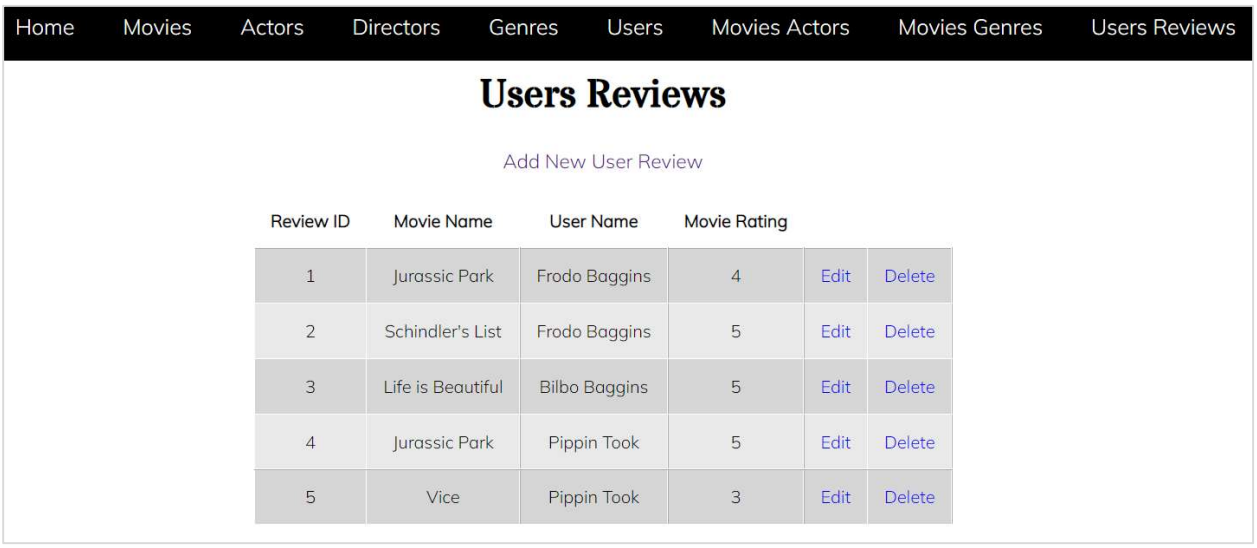
READ, DELETE Movies Genres page:



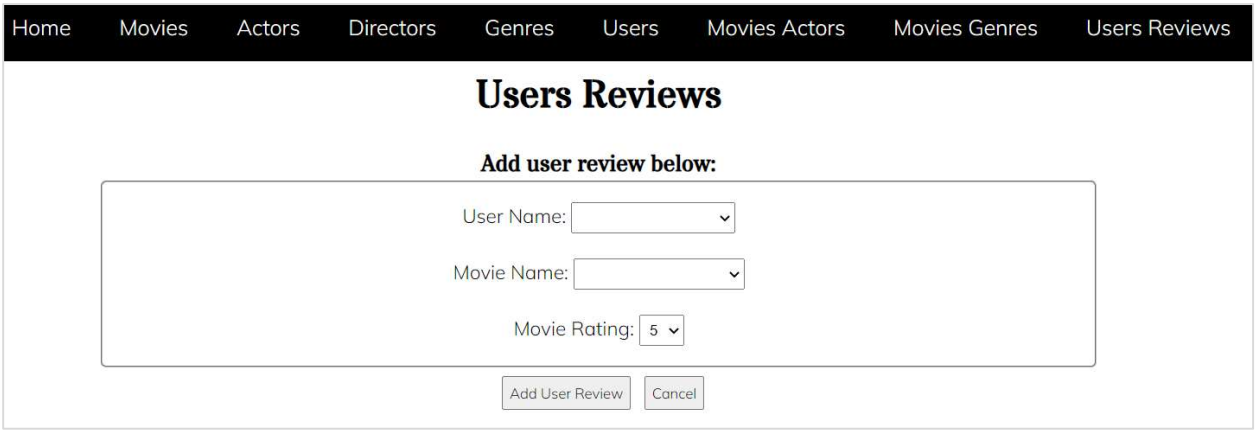
CREATE New Movie Genre page:



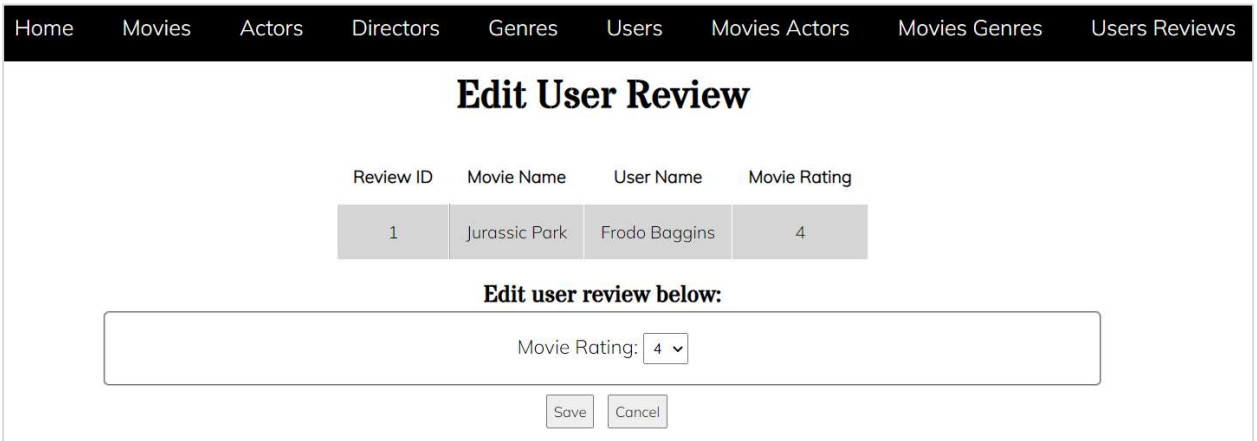
READ, DELETE Users Reviews page:



CREATE New User Review page:



UPDATE User Review page:



Example of error handling page presented when the user tries to add a value already in the data:

Example of error handling page presented when the user tries to delete a value and a foreign key constraint fails:

Citations

[1] “Press Room, IMDb Statistics.” IMDb.com Inc., https://www.imdb.com/pressroom/stats/. Accessed 2 Oct 2022.
[2] “IMDb” Wikipedia.com, https://en.wikipedia.org/wiki/IMDb . Accessed 8 Oct 2022.