



# Movie Recommendation Application

Ben Hackett-Delaney

N00211885

Supervisor: Cyril Connolly

Second Reader: Naoise Collins

Year 4 2024/25

DL836 BSc (Hons) in Creative Computing

## Abstract

This project aims to develop a movie recommendation application that provides personalized movie suggestions based on the preferences of the user. The objective is to enhance the user's experience by using machine learning to give the users recommendations. This system will have a Flask backend, which will act as the API for retrieving and ranking movies using TensorFlow models. React.js is used for the front-end for user interaction.

The development for this project includes processing data, model training, API integration, and front-end implementation.

## Acknowledgements

**The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.**

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

**WARNING:** Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed, and/or copied by others if left unattended.

Plagiarism is an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

**The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below**

*Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.*

**DECLARATION:**

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student: Ben Hackett-Delaney

Signed: Ben Hackett-Delaney

Failure to complete and submit this form may lead to an investigation into your work.

## Table of Contents

1	Introduction .....	1
2	Research.....	2
3	Requirements.....	2
3.1	Introduction .....	2
3.2	Requirements gathering .....	2
3.2.1	Similar applications.....	2
3.2.2	Interviews .....	2
3.2.3	Survey .....	2
3.3	Requirements modelling .....	2
3.3.1	Personas.....	2
3.3.2	Functional requirements .....	3
3.3.3	Non-functional requirements .....	3
3.3.4	Use Case Diagrams.....	3
3.4	Feasibility .....	3
3.5	Conclusion.....	3
4	Design .....	4
4.1	Introduction .....	4
4.2	Program Design.....	4
4.2.1	Technologies .....	4
4.2.2	Structure of Laravel/Unity/Android (2 pages).....	4
4.2.3	Design Patterns.....	4
4.2.4	Application architecture (1 page) .....	5
4.2.5	Database design.....	5
4.2.6	Process design.....	5
4.3	User interface design .....	5
4.3.1	Wireframe .....	5
4.3.2	User Flow Diagram.....	5
4.3.3	Style guide.....	5
4.3.4	Storyboard .....	6
4.3.5	Level Design .....	6
4.3.6	Environment .....	6
4.4	Conclusion.....	6
5	Implementation .....	7
5.1	Introduction .....	7
5.2	Scrum Methodology .....	7

5.3	Development environment .....	7
5.4	Sprint 1.....	8
5.4.1	Goal.....	8
5.4.2	Item 1.....	8
5.4.3	Item 2.....	8
5.5	Sprint 2.....	8
5.5.1	Goal.....	8
5.5.2	Item 1.....	8
5.5.3	Item 2.....	9
5.6	Sprint 3.....	9
5.7	Sprint 4.....	9
5.8	Sprint 5.....	9
5.9	Sprint 6.....	9
5.10	Sprint 7.....	9
5.11	Sprint 8.....	9
5.12	Sprint 9.....	9
5.13	Conclusion.....	9
6	Testing.....	10
6.1	Introduction .....	10
6.2	Functional Testing.....	10
6.2.1	Navigation .....	10
6.2.2	Calculation .....	11
6.2.3	CRUD .....	11
6.2.4	Discussion of Functional Testing Results .....	11
6.3	User Testing .....	11
6.4	Conclusion.....	11
7	Project Management .....	13
7.1	Introduction .....	13
7.2	Project Phases.....	13
7.2.1	Proposal .....	13
7.2.2	Requirements.....	13
7.2.3	Design .....	13
7.2.4	Implementation .....	13
7.2.5	Testing.....	13
7.3	SCRUM Methodology .....	13
7.4	Project Management Tools.....	14

7.4.1	Trello .....	14
7.4.2	GitHub .....	14
7.4.3	Journal.....	14
7.5	Reflection .....	14
7.5.1	Your views on the project.....	14
7.5.2	Completing a large software development project.....	14
7.5.3	Working with a supervisor .....	15
7.5.4	Technical skills .....	15
7.5.5	Further competencies and skills .....	15
7.6	Conclusion.....	15
8	Conclusion.....	16
	References .....	17

## Introduction

The aim of the project is to develop a movie recommendation application that will provide the users with personalized suggestions based on their preferences. It uses machine learning to rank and retrieve relevant movies.

### Application area

The project focuses on recommender systems and machine learning, specifically on movie recommendations. Such systems are widely used in streaming platforms to help users discover content based on past interactions.

### Technologies

- Flask, used to handle the backend, acting as an API process, and serving movie recommendations.
- React JS, Provides interaction for the recommendations system.
- TensorFlow, used for the implementation of machine learning models for basic retrieval and ranking of movies.
- SQLite, For data storage.

### Project management

Trello, for task management and process tracking.

GitHub, for version control and to maintain project repositories.

Figma, to design wireframes for the application.

Miro, for organization and to keep track of the sprints.

### Requirements

### Design

### Implementation

### Testing

# 1 Research

## (Concept 1) -- Recommender Systems

### Introduction

A Recommender System is a widely used technology that provides the users of these applications with personalized suggestions of items they might be interested in. In this section the recommender system will be discussed in all it is from depending on the system that uses the technologies, such as for streaming web applications such as Netflix or Prime video using collaborative filtering, content-based filtering or a hybrid model using both. Pretrained models have become more popular over the years, models using deep learning can handle larger datasets and learn complex patterns for the items features and user interactions which play a part in the predictions of recommender systems. These methods for models provide an effect way to recommend user items.

### What is a Recommender System (RS)?

A Recommender System (RS) is software application designed to give users personalized recommendations on items, or services based on their interactions, and past interests. Analyzing user data, the (RS) can identify trends that help the predictions for what the user might like (Shah et al., 2017). The (RS) plays a curtail role in helping the users find items that are aligned with their interests across numerous sectors, these systems are used in industries such as the movie industry where users would be suggested movies based on their viewing habits. By looking and analyzing the data more of the user's preferences are tailored to the recommender system that then offers the personalized items (Shah et al., 2017).

The recommender makes predictions on how the user rates and or interacts with items they have not seen before, this is based on the user's behavior history and the histories of similar users (Prasad and Kumari, 2012). Effective (RS) provide personalized recommendations, tailoring suggestions to each individual user's tastes and preferences (Prasad and Kumari, 2012).

### How Does it Work?

The (RS) functions by collecting the data and then analyzing the user's personal suggestions. Starting off it gathers the data to then be processed using various algorithms to identify patterns, based on this the systems analysis it predicts what items would be interesting to that user, the system continuously learns from new data being analyzed to refine its suggestions (Prasad and Kumari, 2012).

(RS) leverages the user's past interactions and interests to give them a personalized suggestion that aligns with their own personal taste and wants. The systems go and analyze the user's behaviors, like their search patterns, ratings, and more to recommend items to the users that are relevant and appealing to them (Shah et al., 2017).



## TensorFlow

### 1.1.1.1 Definition of TensorFlow

TensorFlow is a framework that is used in the development of machine learning and machine learning models using deep learning techniques. Languages such as Python are commonly used and are suited for housing models from TensorFlow, it allows for the use of libraries such as TensorFlow and Keras, which allows for effective building, training, and deployment of the model with less coding (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022).

### 1.1.1.2 TensorFlow Pretrained Models

TensorFlow has a wide range of pretrained models at its disposal for developers to use in applications (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022). The deep learning models are pretrained for large datasets for a specific task to complete, the models can be accessed through repositories, in this case the repository would be TensorFlow which can be used to extract features from data content to complete its task. Google colab is the dev environment that is used for the model. The Python tools are necessary for the libraries TensorFlow and Keras for the training and exporting of the model (Puja Singh, Dr Harsh Mathur, 2017).

### 1.1.1.3 Google Colab

Google Colab is a dev environment that allows for building machine learning models, using google colab, it can support python which can gives it access to the pre trained model libraries, and if the process takes more time the developer can use the accelerator for GPUs to fast track the process, for example when doing the epochs it can take time depending on the model and the data it is processing. When a model is pretrained it can be exported and converted to TensorFlow lite. Problems can arise when the model is exported and put into a backend folder depending on if the framework is compatible with the model and Python (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022), (Puja Singh, Dr Harsh Mathur, 2017).

### 1.1.1.4 Basic Ranking Model

A ranking model from TensorFlow usually worked using the score-and-sort method. The method works using the neural network that assigns the scores using a function to all the items in a list using the items features to predict the ranking. The model's training can involve training on one item, a pair of items, or the whole list. It normally sorts the rankings in descending order after the predictions have been made for the items (G. Adomavicius, A. Tizhilin, 2005, R. Burke, 2002), (Qian Zhang, Jie Lu, Yaochu Jin, 2020).

## Types of Recommender Systems

### 1.1.2 Collaborative Filtering

Collaborative Filtering (CF) is a recommender method that uses the predictions of other users to predict what a user might like, by using the preferences and behaviors of other users. The system works by analyzing the user-item interactions the user has made and seeing the patterns or

similarities between the users (Shah et al., 2017).

### 1.1.3 Principle of Collaborative Filtering

The (CF) systems work on the basis that users that have shown similar preferences for items in the past are likely to share similar preferences in the future; this works on the assumption that patterns of user behaviors are consistent over time. (Goldberg et al. 1992).

### 1.1.4 Types of Collaborative Filtering

There are two main types of (CF) that can be categorized into memory-based and model-based. Memory-based methods can also be called neighborhood-based or heuristic-based (Goldberg et al. 1992).

Memory-based (CF) uses the user's stored ratings to predict or the preferences for new items, there are two approaches to consider, the User-Based (CF) is used to predict the user's interest in an item by going through ratings from other users who have similar preferences (Kunal Shah, Akshaykumar Salunke, Saurabh Dongare, Kisandas Antala. 2017)

Model-based (CF) uses machine learning and data mining to build models to analyze the users' rating data. The models find patterns in the data and then make predictions based on the user's preferences to give them effective recommendations (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

### 1.1.5 Content-Based Filtering

Content-Based Filtering (CBF) is a recommendation approach that creates recommendations for the users based on the content of the item and the user's profile. Differing from the methods that use the user's opinion the (CBF) gives recommendations that are comparable items the user has liked in the past (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007)

### 1.1.6 Characteristics of Content-Based Filtering

(CBF) is done by matching items to the user's preference through a series of steps.

User Profile Creation is a system that builds on the data that the user provides. This data can be implicit data, which is the items the user clicks or views, it can also be explicit data such as ratings or reviews. The user profile reflects the users interests and the user's behaviors (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

Item Representation is a system where the items are represented as a set of characteristics, these can be categories, keywords that describe the content of an item (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

Recommendation Generation is when the users and items representation are made. It then compares the content of the user's profile and the items; the system will then recommend items

that are like the items the user has shown an interest in. (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

### 1.1.7 Key aspects of Content-Based Filtering

Item-to-Item correlation focuses on generating recommendations by comparing items to other items, the system looks for similarities between the content of items and the suggested items the user is already shown interest in (Prasad and Kumari, 2012).

Feature Extraction is used to extract the relevant features from the item, like the keywords, or categories, to build a detailed representation of the items, this makes the matching process to the user easier (Prasad and Kumari, 2012).

Term Parsing involves selecting the individual words to represent the content, the system can identify the key terms to understand the main idea of the content to then be compared to different items (Kunal Shah, Akshaykumar Salunke, Saurabh Dongare, Kisandas Antala. 2017).

### 1.1.8 Hybrid Approach

The Hybrid Recommender System (HRS) combines the different recommendation methods to improve the performance and to address the limitations of the individual model's methods. The common hybridization strategy combines (CF) and (CBF) (Kunal Shah, Akshaykumar Salunke, Saurabh Dongare, Kisandas Antala. 2017).

### 1.1.9 Hybridized Strategies

Weighted hybridization is a method that combines recommendations from different methods by getting the averaging scores individually, each of the methods are given a weight based on their reliability, the more reliable methods have a greater impact on the final recommendation. This helps to get improved accuracy by balancing multiple approaches (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

Mixed Hybrid involves multiple recommender systems, those being the (CF), and the (CBF), creating their own lists of recommendations, these lists are then combined into a single list of recommendations. This offers a diverse and balanced list of suggestions (Kunal Shah, Akshaykumar Salunke, Saurabh Dongare, Kisandas Antala. 2017).

Cascade Hybridization organizes the recommender methods in a specific order of priority, the higher priority methods create an initial list of the recommendations, and the lower priority refine the recommendations or fixes problems between them. As an example, the (CF) could make the initial list and then the (CBF) would refine it (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

Switching Hybridization would switch between recommender methods based on the criteria, the type of user or item. For a new user with limited data the (CF) would be used, while for users with more data the (CBF) would be used. This allows the system to adapt to the availability of data (Kunal Shah, Akshaykumar Salunke, Saurabh Dongare, Kisandas Antala. 2017).

#### 1.1.10 Collaborative Filtering Classifiers

Collaborative Filtering (CF) is a recommender system that predicts and recommends items by using the users' preferences and their interactions (Meenu Gupta, Aditya Thakkar, 2021). Unlike Content-Based Filtering (CBF), (CF) relies on the collaborative behaviors of users rather than the attributes of the items (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

(CF) is good in scenarios where there is not enough user interaction data, this makes it suitable for platforms that use larger user engagement, it identifies patterns in the user's behaviors and then the (CF) can recommend items to other users with similar preferences have liked, even if the content is different (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

(CF) can uncover hidden relationships between the item and the user, this leads to a diverse and at times an unexpected recommendation (Shah et al., 2017).

#### 1.1.11 Classification

The system will predict whether a user will like the item by analyzing the patterns in the user's interactions such as the ratings they give other items (Fahin Mansur, Vibha Patel, Mihir Patel. 2017), (CF) uses this to recommend items based on the preferences of others with similar behaviors (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

(CF) does not focus on the items' features, it identifies the relationships between the users and the items by creating a matrix of user-item interactions (Shah et al., 2017). This allows the application to recommend items by recognizing similarities between user preferences and suggesting items that are liked by similar users (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

#### 1.1.12 Overview of KNearest-Neighbors (k-NN)

KNearest-Neighbors (k-NN) is a classifier method that is used in recommender systems, the algorithms are used in Content-based Filtering and Collaborative Filtering (Meenu Gupta, Aditya Thakkar, 2021).

#### 1.1.13 Classification

(k-NN) determines the class of an unlabeled data point by going through the classes of its nearest neighbors (Nabil Belacel, Guangze Wei, Yassine Bouslimani, 2020). It is done through a majority review, for example if most of the book's neighbors are tagged as action the algorithm will classify it as part of the action genre (Nabil Belacel, Guangze Wei, Yassine Bouslimani, 2020).

#### 1.1.14 Nearest Neighbors

The k in (k-NN) determines how many neighbors are considered for classification (Meenu Gupta, Aditya Thakkar, 2021). The smaller the k value is the more specific the predictions get but it can

cause incorrect recommendations due to irrelevant or misleading data points (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007). With a larger k value it averages over a broader range which in turn creates a better prediction but potentially loses the finer details (Meenu Gupta, Aditya Thakkar, 2021).

#### 1.1.15 Memory-Based

Using a memory-based approach the (k-NN) utilizes the user-item rating or have vectors creating explicit models (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007). This makes it flexible and quick to implement, however it also requires storing and accessing the whole dataset, this can make the model need for resources intensive when working with a large-scale application (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

### KNearest-Neighbors (k-NN) in Collaborative Filtering (CF)

#### 1.1.16 Item-Based Collaborative Filtering (CF)

Using item-based (CF) the (k-NN) recommends the items like the ones the user has liked or interacted with in the past, the system computes the similarity between items based on the user's ratings (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

#### User-Based Collaborative Filtering (CF)

Using User-based (CF) the (k-NN) finds the users with similar preferences and then recommends them items that the users who are similar have liked (Meenu Gupta, Aditya Thakkar, 2021).

#### 1.1.17 AI Evolution in Recommender Systems

Recommender systems (RS) have evolved due to an increased volume of online data and users' expectations (Qian Zhang, Jie Lu, Yaochu Jin, 2020). Early recommender systems relied solely on simple algorithms using (CF) and (CBF) to get basic recommendations (Qian Zhang, Jie Lu, Yaochu Jin, 2020). As the internet has expanded and the data gets more complex these systems can encounter challenges.

##### 1.1.17.1 AI Recommender Systems (RS)

AI-based techniques have risen over the past few years and as such it has transformed recommendation systems into a dynamic and personalized model. The systems now use a more advanced algorithm such as machine Learning to improve the recommendations accuracy and to handle copious quantities of user data (Shah et al., 2017). Hybrid systems have become common as they use the strengths of (CF) and (CBF) methods for more accurate suggestions (Qian Zhang, Jie Lu, Yaochu Jin, 2020).

#### 1.1.17.2 Advanced AI in Recommender Systems (RS)

(RS) using AI has continuously gotten better and used more complex methods such as deep learning, to provide more accurate recommendations (Qian Zhang, Jie Lu, Yaochu Jin, 2020). The systems use neural networks to get complex relationships between the users and the items. Recurrent Neural Networks (RNNs) enable session-based recommendations that respond to the user's interactions in real time.

#### 1.1.17.3 Context Aware and Personalization

AI recommender systems are context-aware and personalized, the systems adapt to the user's interactions and personal preferences and gives them real-time suggestions based on the user's historical data (Meenu Gupta, Aditya Thakkar, 2021). Through continuous analysis of user interactions and the context, the recommender system personalizes the recommendations based on the areas where users interacted the most (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

#### 1.1.17.4 Emerging Trends and the Future

New interactive systems and voice feedback systems have been developed as voice assistants such as Siri, and Alexa. Systems that are based on Natural language processing (NPL), voice feedback is a niche area for recommender systems (Qian Zhang, Jie Lu, Yaochu Jin, 2020).

### 1.1.18 AI and Collaborative Filtering

#### 1.1.18.1 Improved Accuracy

AI uses machine learning and deep learning, this has improved (CF) recommendation accuracy (Qian Zhang, Jie Lu, Yaochu Jin, 2020), (CF) can struggle to understand more complex patterns unlike the AI that can identify intricate relationships between the users and the items, which will result in better predictions

#### 1.1.18.2 Improved Personalization

AI improves personalization by using the user's user-specific factors like the demographic, and or historical data (Fahin Mansur, Vibha Patel, Mihir Patel. 2017). The deep learning models process the datapoints and then identify the complex user preferences, doing this provides the users with increased personalized recommendations (Qian Zhang, Jie Lu, Yaochu Jin, 2020). Neutral Collaborative Filtering (NCF) refines the learning non-linear relationship between the users and the items giving personalized recommendations to the users improving their quality on the application (Qian Zhang, Jie Lu, Yaochu Jin, 2020).

### 1.1.19 Critical Analysis

(RS) personalize suggestions to users based on their preferences and behaviors, this helps the users discover new items they may not have previously seen across many domains (Prasad and Kumari, 2012). Although they are effective in personalizing for the users, recommender systems still face challenges, these challenges can include affecting the performance of the model, the data the model looks at, and sparsity in data (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007). There can be tradeoffs in the design of the recommender system as optimizing one aspect can lead to one

part of the model doing well while the other sections fail, this can affect the whole result (Shah et al., 2017).

#### 1.1.20 Limitations of Recommender Systems

##### 1.1.20.1 Data Sparsity

(RS) and especially (CF) can often struggle with data sparsity, when a small subset of the items has been rated there is a lack of data that is needed to make accurate predictions making it a challenge to get the users preferences (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

##### 1.1.20.2 Cold-Start Problem

Inexperienced users and items do not have historical data which makes it difficult for the (RS) to give recommendations (Prasad and Kumari, 2012). This is called a Cold-Start and is particularly challenging for a (CF).

#### 1.1.21 Trade Offs with Recommenders

##### 1.1.21.1 -5.2.1 Accuracy vs. Diversity

Accuracy ensures that the (RS) is recommending similarly matched preferences from the user, this can lead to overspecialization, meaning the system will only suggest items that are remarkably similar, the users may have already interacted with the item (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

#### 1.1.22 Conclusion

Recommender systems are important in the role of giving users items they previously may have not looked at before. By using methods such as collaborative filtering, content-based filtering, or a hybrid approach, or the use of a pretrained model can suggest relevant items to users based on the users features, preferences, and interactions. Pretrained models based on deep learning have become more popular making it easier to handle larger datasets while improving accuracy through fine tuning. With these methods a strong recommendation system can be made.

#### 1.1.23 (Concept 2) -- Data Used in Recommender Systems

##### 1.1.24 Introduction

Data is extremely important regarding a (RS), it gives them the ability to provide personalized recommendations (Meenu Gupta, Aditya Thakkar, 2021). The results depend on the quality, diversity and how the data is structured (Oren Sar Shalom, Shlomo Berkovsky, Royi Ronen, Elad Ziklik, Amihood Amir, 2015). Understanding what types of data is the best to use, and how processing and optimizing them is important for making a good model.

#### 1.1.25 Data in Recommender Systems

#### 1.1.26 Data Sources

##### 1.1.26.1 Public Datasets

Kaggle and TensorFlow are websites that host datasets which can include the users rating and interaction data (Fahin Mansur, Vibha Patel, Mihir Patel. 2017). These datasets are useful for training and testing the recommender system as it will show if the algorithm is working through evaluation during development (Meenu Gupta, Aditya Thakkar, 2021).

#### 1.1.27 Types of Data Used

This data type gets the user's data and the interaction history of the user, this helps the system understand user preferences to then give better recommendations (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007). (RS) can use more than one of the data types with some relying of the ratings, while other systems look at the content-based data (Qian Zhang, Jie Lu, Yaochu Jin, 2020).

##### 1.1.27.1 User data

User data includes the demographic of the user being their age, and gender, it also uses the behavior of the user to see their past preferences and what their needs are (Prasad and Kumari, 2012). The way this data is collected is through explicit means using the user's inputs such as the ratings they have given or what items they have liked or disliked, this then gives the preferences of the users. The data can also be obtained through implicit means which go through the user's interaction history, for example how many times a user visits the same page (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

##### Item data

Item data is the data about the items that could be recommended, this can include keywords like a genre or a description or any of the item's attributes (Prasad and Kumari, 2012). The item data can also include the user's ratings on the items, the reviews they have left on the items (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

#### 1.1.28 Data Storage

(RS) need data storage for the types of information, this data includes the user's data, the items-rating data, and the item data. The type of storage that is used depends on the data and what the systems configured for (Goran Antolic, Ljiljana Brkic, 2017).

#### 1.1.29 Databases

(RS) use diverse types of databases to store and manage the data needed to create recommendations to users. The choice of database types is relational databases and NoSQL databases, The performance of these two databases depends on the type of data that is being used,



if the volume of the data is large or small, and the requirements for the system (Goran Antolic, Ljiljana Brkic, 2017).

#### 1.1.29.1 NoSQL

NoSQL databases are used to store the user's profiles and the recommendations, it does this because it can handle substantial amounts of users and flexible schemas (Meenu Gupta, Aditya Thakkar, 2021). NoSQL databases can store data about the users like their relationships (Goran Antolic, Ljiljana Brkic, 2017).

#### 1.1.29.2 SQLite

SQLite is an open-source, light, database, it runs within the application which makes it efficient, it stores data in tables and organizes the database into a file, it is not the best database for larger datasets or large-scale operations for that matter, but it is great for smaller applications (Grant Allen, Mike Owens, 2010).

#### 1.1.30 Preprocessing Data

Data Processing is a crucial step in creating a (RS), real world data can be incomplete and or inconsistent (Oren Sar Shalom, Sholmo Berkovsky, Royi Ronen, Elad Zikik, Amihoud Amir).

#### 1.1.31 Data Cleaning

Cleaning the data will ensure the dataset is consistent, accurate, and does not cause any errors to occur, is the foundations for building a reliable (RS) and missing data can affect the results if not handled properly, this would involve filling the missing values replacing them with zero or removing the empty value entirely (Bernard Magara Maake, Sunday O. Ojo, Tranos Zuva, 2019). Another issue that can occur is noisy data which are the outliers and errors that need to be smoothed over to remove any inconsistencies in the dataset. Outliers would then be removed to avoid a skew in the analysis.

#### 1.1.32 Data Transforming

Data transforming changes the raw data's format and makes it suitable for use in machine learning models for analysis. Normalization and the aggregation of data are used in the analysis and the building development of a model, it scales numerical data to a get the range, which can improve the performance of the algorithm (Xavier Amatriain, Alejandro Jaimes, Nuria Oliver, Josep M. Pujol). Text transformation converts the text into a consistent format so that it can be reviewed, this can be changing the uppercase to lowercase or removing special characters, and keywords.

#### 1.1.33 Conclusion

The data is an important part of the application in terms of the recommendation system being a success, storing the data for it to be reused in the application. Going over the databases required to

store the data is important for managing it. This will provide the model with ample data for it to then recommend items to users more accurately.

#### 1.1.34 (Concept 3) -- Architecture of Model Integration

##### 1.1.35 Introduction

The integration of machine learning models into front-end and back-end applications can be challenging when considering between front-end and back-end deployment, both come with advantages and disadvantages that can hinder the performance of the application (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022). The Front-end allows for reduced latency and real-time interaction, but this requires the model to be optimized for devices with limited resources (Dimas, 2024). In the back end it can access powerful server resources and security measures, the problems with the back end are it can cause latency issues and can make the interaction with an API complex. A hybrid model could help solve these issues.

##### How Back-End Integrates Models

The back-end integration requires hosting the model on a server or a cloud platform, these are accessed through APIs (Dimas, 2024). TensorFlow would serve and another application like Google cloud could deployment of the model (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022). The models can use GPUs or TPUs for more strenuous tasks during development and training, this happens on the server.

Alternatively, the model can be exported into a zip file from google colab, it can then be downloaded and implemented into the backend to then be queried, from this the ranked model data can be called through the use of an API (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022), (Sebastian Proksch, Veronika Bauer, Gail C. Murphy, 2014).

##### 1.1.36 Front-End Technologies for Integration

##### 1.1.37 Frameworks and Libraries

Front-end web development is crucial in creating the user interface. This would focus on the layout, design of the website, and the pages of the website, and a smooth interactive experience. JavaScript can create an interactive and dynamic front-end allowing for many features for design like form validation and asynchronous data loading (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007). Other frameworks like React.JS can build scalable, interactive applications, using tools like Bootstrap or Tailwind which offer pre-built components that can be used in the application can help save time under constraints (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022). CSS processors like SASS can help maintain the design over the whole application, then using GitHub to track the changes and to collaborate the back end, the model can be exported to the backend and serves as an API to be used in the front-end to get the data to be displayed.

#### 1.1.38 API Integration

APIs are used in building the (RS) and the web app by getting data using user data and can use item databases. This also provides access to pre-trained models which can make deployment easier (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022). APIs give access to the presentation of recommendations in the existing user interface. This works through auto-complete which is a UI element that suggests matches such as when a user uses the search bar and suggestions are ready to pop up for them. The API can also make customized UI elements which can help the users understand the recommendations improving the users experience (Sebastian Proksch, Veronika Bauer, Gail C. Murphy, 2014).

A problem that can occur when using APIs is where the developer of a ready-made API is the vocabulary, the API doesn't know the terminology or the structure of the API which can cause time loss when finding the information for the recommender (Sebastian Proksch, Veronika Bauer, Gail C. Murphy, 2014). Another problem that could occur is outdated APIs or lack of necessary components regarding attributes in the data.

Alternatively, the models can be extracted by querying the model in the backend. After this the model can be used as an API allowing for the ranked data to be displayed for the user in the front end (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022), (Sebastian Proksch, Veronika Bauer, Gail C. Murphy, 2014).

#### 1.1.39 Integration of a Data

Using TensorFlow, it is a JavaScript library that allows for machine learning models to be integrated into the front-end directly, this allows for real-time predictions in the browser or on a server like Node.JS. It can support pre-trained models, AI images, sentiment analysis, and importantly, recommender systems (Qian Zhang, Jie Lu, Yaochu Jin, 2020). TensorFlow can use a library called WebGL which can speed up machine learning tasks. It does this by offloading calculations to the GPU, allowing the browser to render 2D and 3D graphics. This improves TensorFlow by making the running of models more efficient across all platforms (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022).

Alternatively, the use of google colab allows you to export models from google colab directly making it a zip file that can be downloaded. Once the model is downloaded it can be put into a folder. This folder contains the zip file and the opened file where it is then loaded to be queried involving changing vectors and converting numeric data to string form. From this the data can be extracted from the model and displayed (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022), (Sebastian Proksch, Veronika Bauer, Gail C. Murphy, 2014).

#### 1.1.40 Back-End Technologies for Integration

Back-end integration is important when integrating models into a web app, which would enhance the performance improving the processing speed, how the applications deploy, and the scalability of the model (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022). It provides data management and infrastructure for scalability.

## Frameworks and Integration

Back-end technologies like Python, PHP, Flask, JavaScript, and C++ frameworks are essential for machine learning model integration into web applications and (RS) these frameworks are used in the model's development, processing the data, and creating the back-end logic for the application. TensorFlow, Node.JS and Django would be used to deploy the models, using Docker it can deploy and help with scaling the application, for a serverless option AWS Lambda can be used, it automatically scales based on the number of requests. Using databases like MongoDB, MySQL with external sources like XAMPP can store data for (Goran Antolic, Ljiljana Brkic, 2017). MySQL and MongoDB are suitable for maintaining unstructured data such as user profiles, the user's interactions and their preferences, MySQL is a relational database which can be used for structured data also such as the user's ratings and the descriptions for the items in the (RS).

Additional technologies that can be used are Mahout, a JavaScript library for the recommender system. The strengths of this are the ability to handle larger datasets making it useful for analyzing considerable amounts of data. It also scales and handles the increasing number of users and the items (Goran Antolic, Ljiljana Brkic, 2017).

### 1.1.41 Conclusion

Integrating the front and backend is critical for the recommender system, getting the data from the backend can involve different methods such as hosting the model, or downloading it and implementing in a folder in the backend and then extracting the data from the model through querying seems like the best option as the model and backend will be in one place improving response times when getting the data for the front end. Having the model in the backend also improves the scalability of the application as new data comes in and makes the system more maintainable overtime.

## 2 Requirements

### 2.1 Introduction

This project is on creating a movie recommendation application that will suggest movies based on the users' interactions to get their preferences. Streaming platforms commonly use these systems to help their users find content they might enjoy. Using machine learning the aim for the system is to provide personal recommendations to users. The application will be made using a Flask backend for processing recommendations and use as an API, a React js front-end for the users to interaction, and the use of a TensorFlow model, the basic ranking. SQLite is the database which will store all the data.

### 2.2 Requirements gathering

Front-end

- User interface
  - The front-end will have a straightforward design to make it easy for users to interact with the system. It will have options for users to enter their preferences through the genres, and or ratings of the movies.
- Movies display
  - The front-end will provide pages for the movie to be displayed, deleted, edited, created but only by admin users.
- User input
  - The front-end will allow the users to enter their preferences, through filtering, and refining movie suggestions based on the user's selection
- Interactive components
  - Interactive components such as buttons, will let the user rate the movie, which will refine their preferences in the system.

## Backend

- Movie recommendation system
  - The backend will use TensorFlow to create a recommendation system that provides personalized movie suggestions based on user preference. ranking models to offer the relevant movies.
- API
  - A Flask API will handle requests from the front-end, processing user input which would return personalized movie recommendations.
- Database
  - The application will use SQLite as the database to store user data, movie data.
- User authentication
  - User authentication provides preference and rating saving to improve the recommendations over time.
- Role based authentication
  - Allows for authenticated DELETE, EDIT, and CREATE in the application.

### 2.2.1 Similar applications

Fig (2.0) Netflix



## Netflix

Netflix is a huge streaming platform, it has movies, TV shows, and documentaries. It uses an AI algorithm to recommend movies, shows, documentaries based on the user's preference, interactions, and view history.

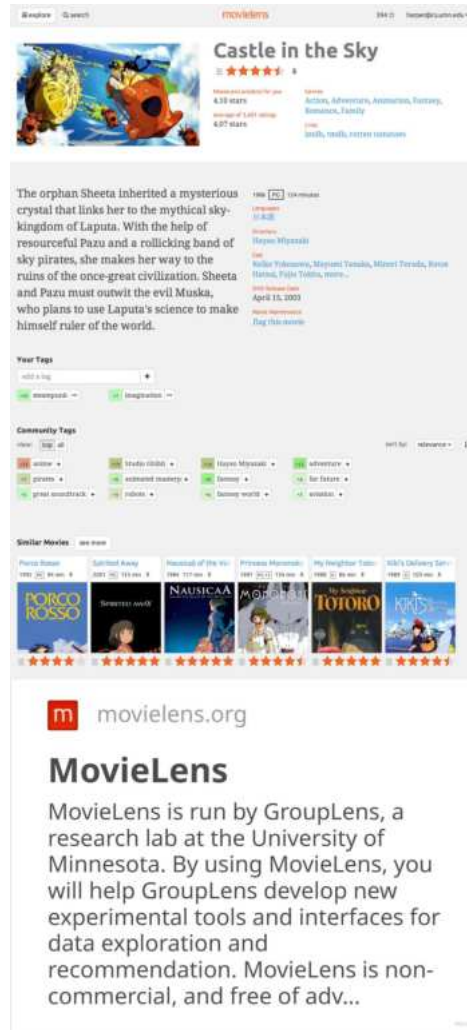
### Advantages

- Netflix has a massive library of content.
- It uses a powerful recommendation system.
- Netflix is supported on multiple devices.

### Disadvantages

- The content in Netflix depends on what region you live in.
- The price of the subscription is remarkably high.

Fig (2.1) MovieLens



## MovieLens

MovieLens is a movie recommendation website that allows users to rate movies and receive suggestions based on the ratings the users have given. It recommends movies based on collaborative filtering.

### Advantages

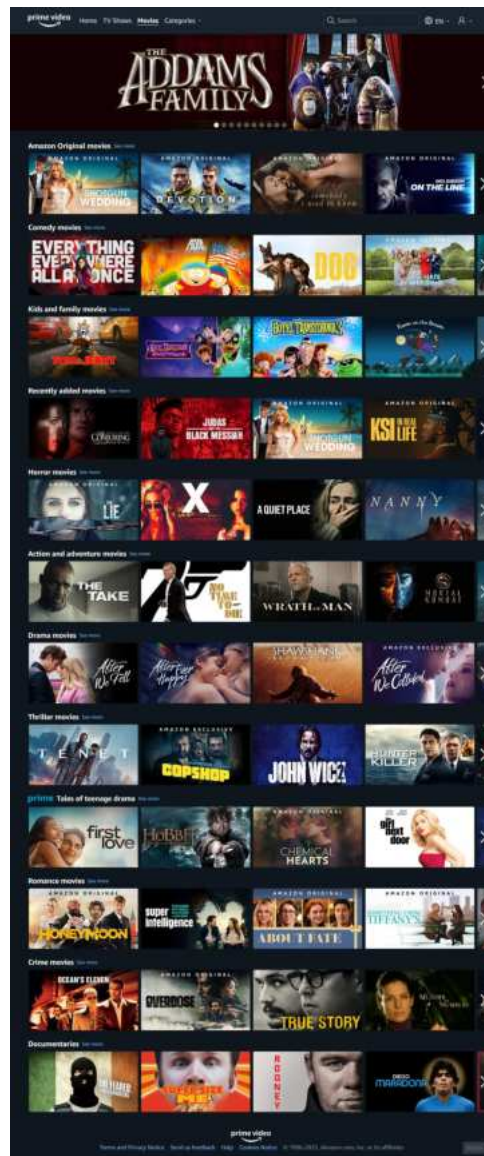
- Provides accurate recommendations.
- No subscription needed to use this service.

### Disadvantages

- The application is limited to the website.
- MovieLens provides a smaller selection of movies.

Fig (2.2) Amazon Prime





## Amazon Prime

Amazon Prime is a streaming service that offers movies, and TV shows. It provides recommendations based on the user's viewing history and their preferences.

### Advantages

- Prime has a large variety of content to view including movies, and TV shows.
- There are no ads when using Prime.
- It is convenient for users who already have a subscription to Amazon.

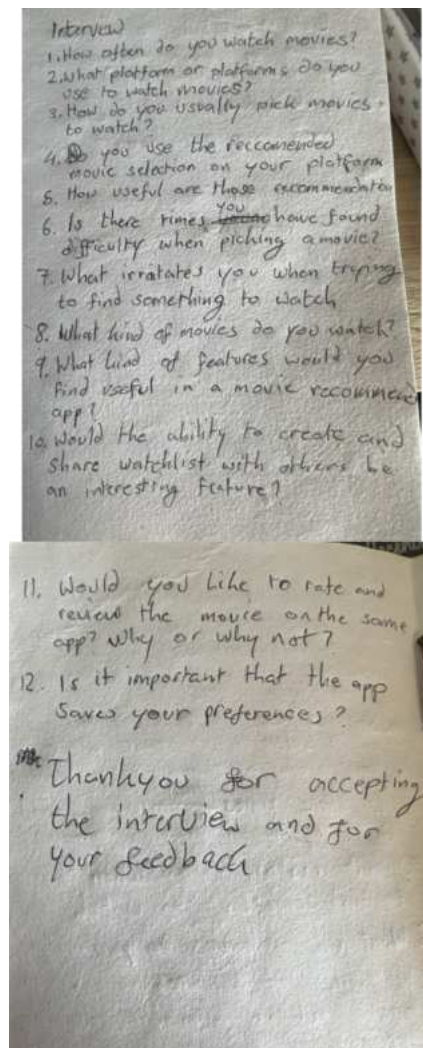
### Disadvantages

- The interface is slow and cluttered.
- Some of the content is free but some require further purchases.
- Prime is region blocked.
- Prime is supported on multiple devices.



### 2.2.2 Interviews

Fig (2.3) Interview questions



The interviews were conducted with three users that were chosen at random, this was done to remove bias in the users' answers. In each of the interviews the focus was to find key features, the problems they face and what solutions may help, suggestions on improvement from other similar applications. The insights that were gathered from the interviews helped to Invision the features that would be necessary to build the application.

### 2.2.3 Survey

Fig (2.4) Survey

The survey was conducted to a group of potential users to collect the feedback on their habits when watching movies, what kind of preferences would they want to have, what they expect from the recommendations for the movies, and key features that will be needed for the application. With the responses it gave a clear picture of features that would be needed and the type of data that would need to be gathered for the recommendation system to give accurate suggestions for movies.

## 2.3 Requirements modelling

### 2.3.1 Personas

Fig (2.5) Persona 1 & 2



Samantha and James are two personas that represent typical users of the movie recommendation application.

Samantha is a twenty-nine-year-old who works in marketing. She watches three movies a week but often feels frustrated with generic recommendations that do not match her interests or repeats movies she has already seen. She wants a personalized, time saving solution for finding new movies.

James is a twenty-one-year-old student who watches movies with his friends and family. With a limited budget he struggles with managing multiple streaming services and spends a considerable amount of time trying to find a movie that he is interested in.

### 2.3.2 Functional requirements

1. The application needs to provide personalized movie recommendations using machine learning models to give movies suggestions.
2. Implementation of the basic ranking methods from TensorFlow to sort and present movie recommendations based on the user's input.
3. Allow users to filter through genres to get movies based on that genre.
4. Allow users to search for specific movies in the search bar.
5. It needs to display a list of movies and give the admin the ability to delete, create, and edit.
6. Admin can interact with actors giving them the ability to assign actors to movies and remove them from movies. The admins can use CRUD functionality on the actors.
7. Users can leave a review on a movie, they can delete and edit their own reviews
8. Needs to allow users the ability to rate movies to improve recommendations. They can also edit and remove their ratings
9. Users need to be able to create, edit, and delete, and add their own watchlist.
10. Users can add their watchlists to public making it display in the public watchlists. Users can also remove their watchlists from public and make it private.

### 2.3.3 Non-functional requirements

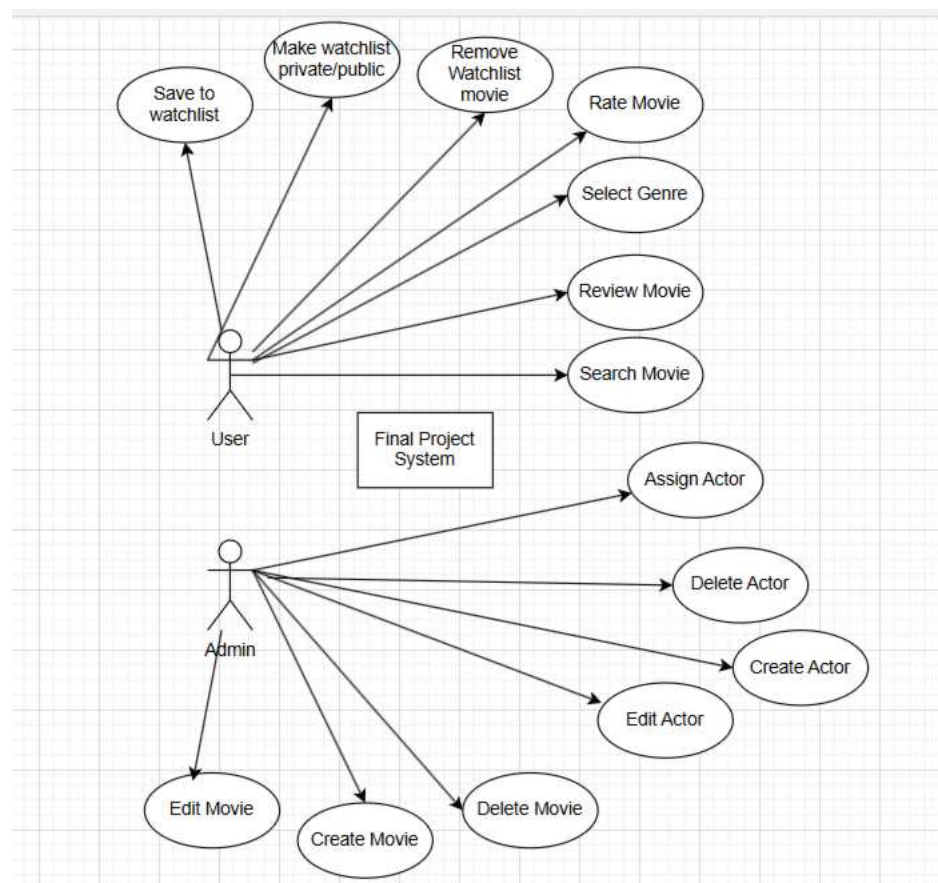
1. The applications usability should be user-friendly, intuitive, and allows users to navigate through the pages, movie recommendations, input preferences, and interactions with the system without problems.
2. The system should be able to load the recommendations quickly.
3. User data should be stored securely, using encryption for the password and other sensitive data that might need to be hidden.
4. The code should be well-organized, and easy to maintain allowing for updates

### 2.3.4 Use Case Diagrams

- A User:
- Can filter movies by genre.
- Can rate a movie.
- Can save a movie to their watchlist
- Can make their watchlist public or private.
- Can review movies.
- Can view actor data.
- Interactions

- Can filter movies based on a selected genre, and the system returns movies with just that genre.
  - Can select a movie and save it to one of their watchlists.
  - Can remove a movie from their watchlist.
  - Can make their watchlist public or private.
  - Can edit and delete their own reviews.
  - Can view other users reviews on movies.
  - Can edit and delete their own ratings.
- 
- An Admin:
  - Can manage movie CRUD.
  - Can manage actor CRUD.
  - Can assign and remove actors from movies.
- 
- Can create, edit, and delete movies and actors

Fig (2.6) Use Case Diagram



## 2.4 Feasibility

The project will use these technologies.

- React.js: This is used for the front-end, which will allow for a dynamic and responsive user interface where users can input ratings or view movie recommendations.
- Flask: Flask will act as the backend and API, handling requests from the frontend, executing machine learning logic, and interfacing with the database.
- SQLite: SQLite is used for data storage, managing movie datasets and user information.
- TensorFlow: TensorFlow has pretrained models which can power the machine learning recommendations system to predict users ranking of a movie

These technologies were chosen for their compatibility, widespread use, and they were suitable for a machine learning web application. React.js and Flask integrate well through RESTful APIs; SQLite provides a simple for an application. And TensorFlow offers tools for deploying the basic ranking model.

The challenges that need to be considered.

- The Flask API may have delays when rendering in the data from the UI component.
- Database performance as the data scales in the database can be a problem when the numbers of movies or ratings, or even watchlists increase.

## 2.5 Conclusion

In the requirement phase, it goes over the requirements needed for the application. This section focuses on the technologies that were used, goes over the TensorFlow model for the front and backend, and goes over user feedback with interviews and questionnaires. This chapter highlights the functional and non-functional requirements for the application.

## 3 Design

### 3.1 Introduction

The design of an application is usually divided into:

1. Program Design:  
Technologies
  - Flask:
  - React:
  - Python Libraries:
2. User Interface Design.

The application for this project is a movie recommendation web application, it uses machine learning to provide personalized movie recommendations to users, it does this by predicting the ranking a user might give a movie through their interactions on the application, it gives that movie a rating that's assigned to the user for that movie. It is built with a React.js front-end, in this user can filter movies by genre, search for specific movies, rate movies, review movies, view actors, and save selected movies to their own watchlists. There are administrative abilities admins can utilized such as the CRUD functionality for movies and actor assignment to movies and the CRUD functionalities with actors. The Flask backend acts as the API and manages the machine learning logic, processing user inputs, user authentication, and coordinating with the recommender model. Both the components interact together through RESTful APIs. For data storage, SQLite is used to store the data.

The recommendations are from a TensorFlow model, the basic ranking model, which analyses users' behaviour, such as when they rate a movie it saves the movies attributes such as the genre and predicts what the user might rate a movie that is like.

Additionally, the application will allow certain users to use CRUD functionality on movies and watchlists so that users and admins can keep up to date.

### 3.2 Program Design

The program design is the structure and the components that are needed to make coding the application more straightforward. It goes over the sections of the applications and outlines the responsibilities those areas are undertaking such as detailing the front-end, backend, database, and machine learning model and how they interact with each other.

#### Flask (Backend application)

Flask will be used as the web framework for the backend of the application. It is ideal for creating APIs. In this project, Flask handles the HTTP requests, routes them to the controller functions, and serves the machine learning model to the front-end.

### Responsibilities

- Handles authentication and authorization.
- Interacts with the SQLite database to store and retrieve movie and user data.
- Provides the API endpoints for the user's inputs.
- Processes the incoming data and it interacts with the machine learning model

### Extensions

- Flask-RESTful, uses for creating the RESTful APIs.
- Flask-SQLAlchemy for database interactions.

### React.js (front-end application)

React.js will be used for the front-end for the application, this will allow for an interactive interface for the users to use with the system. React provides component-based structures which allows users to reuse the same components making the process of coding easier

#### Responsibilities

- Renders the UI components.
- Allows users to rate movies and add them to watchlists.
- Displays the recommended movies for the users.
- Allows users to search and filter through movies.
- Makes the asynchronous requests to the Flask backend to retrieve data.

#### State management

- Using React context for managing global state.
- Using Axios for making the actual API call to the backend.

### Python libraries for Machine Learning (Backend)

This backend application will include a machine learning model that is prebuilt from TensorFlow. The model will generate movie recommendations based on the how the user interacts with the application.

#### TensorFlow

- TensorFlow is used to create and run the basic ranking model that predicts what a user might rate a movie based on past interactions.

#### Pandas and NumPy

### SQLite (Database)

SQLite will be used for the database to store the movie data, user data, ratings, and watchlists. It also holds the relationships between these tables.

#### Responsibilities

- The SQLite database stores movie data such as the title, genres, description, and other attributes.
- The SQLite database will keep track of users, their ratings, and their watchlists, their reviews.

#### TensorFlow's model for Recommendations

- The movie recommendation system will work using the pretrained basic ranking model built using TensorFlow. This model will analyse the user's inputs, and it will generate recommendations based on the movie's attributes and user ratings.

#### Responsibilities

- The model will process the users' inputs and predict the ranking the user would give a movie.
- In the route for the ranking model there will be queries to extract the titles from the models' predictions and then the top five highest predicted rankings will be displayed for each user.
- This will be displayed in the home page in the front-end

#### 3.2.1 Technologies

The technologies being used to create this application are:

- Flask
- React.js
- TensorFlow
- SQLite

These technologies were chosen because they offer an efficient and scalable way to build the movie recommender system. Flask is flexible and make it easy to develop and integrate machine learning models.

React.js provides a fast, has a responsive user interface, and makes the coding process more convenient with components.

TensorFlow is ideal for getting models to use in applications, using the basic ranking model the application will be able to predict ranks the users might have on movies.

And SQLite is simple and provides an effective way to store data for this application.

Other technologies which could have been used were Django for the backend and PostgreSQL for the database. but Flask was preferred for its simplicity. PostgreSQL is a more powerful database but was not needed for this application, which is why SQLite will be used.

#### 3.2.2 Structure of Flask/React (2 pages)

The Flask and React application are structured to keep the front-end and back end separate. Which will make it easier to develop, maintain, and scale. The Flask backend will manage the API routes, business logic, and the database. The React front end will communicate with the backend through HTTP requests, with React sending messages to the Flask backend and then Flask returning the data.



This allows the front-end to focus on the interaction while the backend manages the data and the machine learning process.

The config folder holds the configurations files for the application

- Config.py stores configuration settings like the database URI and other environment configurations.

The env folder contains the environment-related files like the environmental variables or virtual environments.

The instance folder holds instance-specific files such as the database

- SQLite database file to store application data.

The migrations folder holds the script for the database schema. It also helps manage changes to the databases structure if it changes over time.

The ml\_models folder contains the machine learning model that is used in the application.

- Ranking\_model is the directory for the movie ranking model to use the recommendations.
- Ranking\_model\_tar\_gz is the zipped that can be loaded for making predictions.

The models contain the python classes that are representing the applications data model.

- The movie.py file defines the movies model.
- The user.py file defines the user model.
- The rating.py file defines the rating model; this will link the user and movie tables.
- The watchlist.py defines the watchlist model, where the users can save movies.
- The actor.py defines the actor model, this table is a many-to-many with movies.

The routes folder stores all the routes for the application; this is where the logic for processing requests and returning the responses.

- The movie\_route.py handles the movie related API endpoints.
- The auth.py handles the authentication routes.
- The ranking\_route.py manages the routes for the ranking model and the movie recommendations.
- The rating\_route.py handles the routes for rating movies.
- The watchlist route handles the routes for the watchlist functionality.
- The actor route handles the routes for the actor functionality.

The seeders folder contains the scripts that are used to seed the database with user and movie data.

- The movie\_seeder.py seeds the database with movie data and assigns actors to movies.
- The actor\_seeder.py seeds the database with the actor data.
- The user\_seeder.py seeds the database with user, watchlist, rating, and review data.
- The seed\_all.py file that sends the data to the database and can drop the tables to reset the database.

The gitignore file is used to ignore large download files, these files should not be sent to GitHub.

The app.py file is the main point of the application; it initializes and runs the Flask application.

The extensions.py file will initialize third-party extensions such as Flask-SQLAlchemy and Flask-Migrate.

Requirements.txt is a list of python dependencies needed for the project to work; this can be installed using pip.

The application is in the folder my-react-app.

The node\_modules contain all the dependencies for the application and can be installed using npm install.

The public folder contains the public files that will serve, this includes static files.

The src folder holds all the source code for the application, this includes the pages, components, and the styles.

- The components folder is in the src folder, this folder holds components that can be reused depending on how it works.
  - The Login.js component is used for user login functions.
  - The Register.js component is used for registration functions.
  - The Navbar component allows users to navigate through the application.
  - The MovieCard displays the movie information.
  - The WatchlistCard displays the watchlist.
  - The rating card will display the movies users have rated.
  - The publicWatchlistCard displays the PublicWatchlist data.
  - SignOut.js is the component used for signing out users.

The pages folder will also be in the src folder. This folder contains different pages in the application.

- The movies folder contains pages related to movies.
  - The movie create allows admins to create movies.
  - The movie edit allows admins to edit movies.
  - The movie single page allows admins to delete movie, allows users to rate and add movie to watchlist.
  - The movie all page allows users to see all the movies.
- The watchlists folder contains pages that are related to the watchlists.
  - The watchlist create allows users to make a new watchlist.
  - The watchlist edit allows users to edit existing watchlists.
  - The watchlist single allows users to view all the movies in that watchlist.
  - The watchlist all page shows all watchlists for that specific user.
  - The publicWatchlist all page shows all the public watchlists in the application.
  - The singlePublicWatchlist displays the information of the public watchlists.
- The actor folder contains pages that are related to actors.
  - The actor create allows admins to create actors.
  - The actor edit allows admins to edit actors.
  - The actor single page is accessible for all users displaying actor data.
  - The actor all page is only visible for admins and allows them to assign users to movies.

The App.js is the root file for the application. It handles routing, and renders different components or pages based on the URL.

Index.css adds daisyUI to style the application.

The gitignore file is used to ignore sending large files to GitHub such as the node\_modules.

Tailwind.config.js is the configuration file for tailwind.

### 3.2.3 Design Patterns

The Model View Controller (MVC) design pattern applies to the structure of the back and front-end of the application. It split the application into three different components, each of the components have their own responsibilities, making the application maintainable.

#### Model

In the Model View Controller, the Model refers to the data and the logic to manipulate that data. The Model component is implemented in the backend.

- The Movie Model defines the data structure of movies, this includes their attributes such as the movie title, movie genre, and other related attributes.
- The User Model defines the data structure of the users; it contains specific data such as the users email and password.
- The Rating manages the user ratings for the movies. It defines a relationship between the user and movie.
- The watchlist model represents the watchlist where a user can add movies.
- Actors model defines the actor model and its attributes such as name, description, and other attributes.
- Review model defines the review model and its attributes.

The models interact with the database to store data and retrieve data.

#### View

The view displays the data to the user. It represents the user interface and how the data will be presented.

- The MovieCard component will display the movie data.
- The WatchlistCard component displays the watchlist where a user can choose which watchlist they want to see.
- The PublicWatchlistCard will display the public watchlists to users where they can see other users watchlists and view their contents.
- The Navbar displays on all pages excluding the login and registration pages, in the movie all page it will display a search bar and filter, on other pages these components will not be visible.
- The RankingCard component displays the predicted ratings a user might give a movie and lists the top five
- The RatingCard component displays the movies the users have rated.

The user can interact with the application, viewing and editing the data if allowed.

## Controller

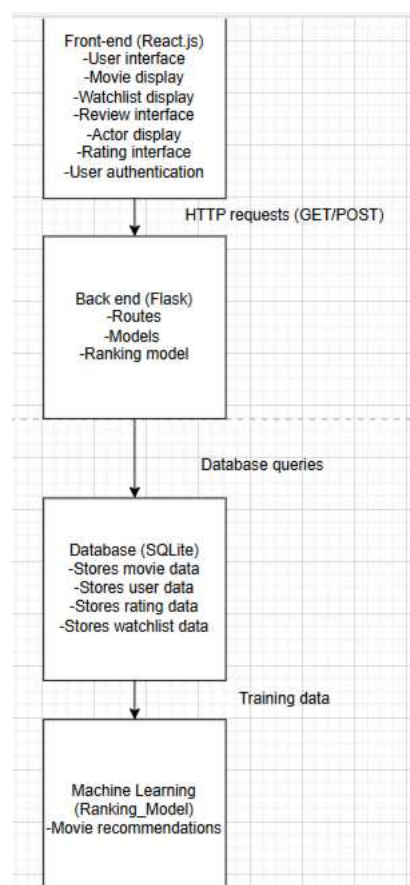
The controller acts the middle of the process handling user inputs, processing the data through the model, and updating the view. In Flask, the internal logic is made in the routes.

- The Movie Routes handle the requests to get, create, edit, and delete for the movies data.
- The Watchlist Routes handle the requests to get, create, edit, and delete for watchlist data, it also allows users to add movie to the watchlist and remove.
- The Actor Routes handle the requests to get, create, edit, and delete for actor data, it also allows the admins to assign actors to movies through a PUT request.
- The Review Routes handle the requests to get, create, edit, and delete for the review data, in the users can view all the reviews but can only edit and delete their own reviews through their user id.
- The Rating Routes defines the routes that manage the user's ratings for movies.
- The Ranking Routes manage the ranking\_model and handles the machine learning model used to generate movie recommendations based on the users' inputs.
- The Authentication Routes handle the routes used to login and register.

These routes make sure that the correct data is sent from the backend to be used in the front-end.

### 3.2.4 Application architecture (1 page)

Fig (3.0) Application architecture



Front-end (React.js)

The React frontend interacts with the Flask backend through HTTP requests to fetch data and display it on the front-end.

### Backend (Flask)

The Flask backend acts as an API that manages data flow between the front-end and the database. It also integrates the ranking model to generate recommendations.

### Database (SQLite)

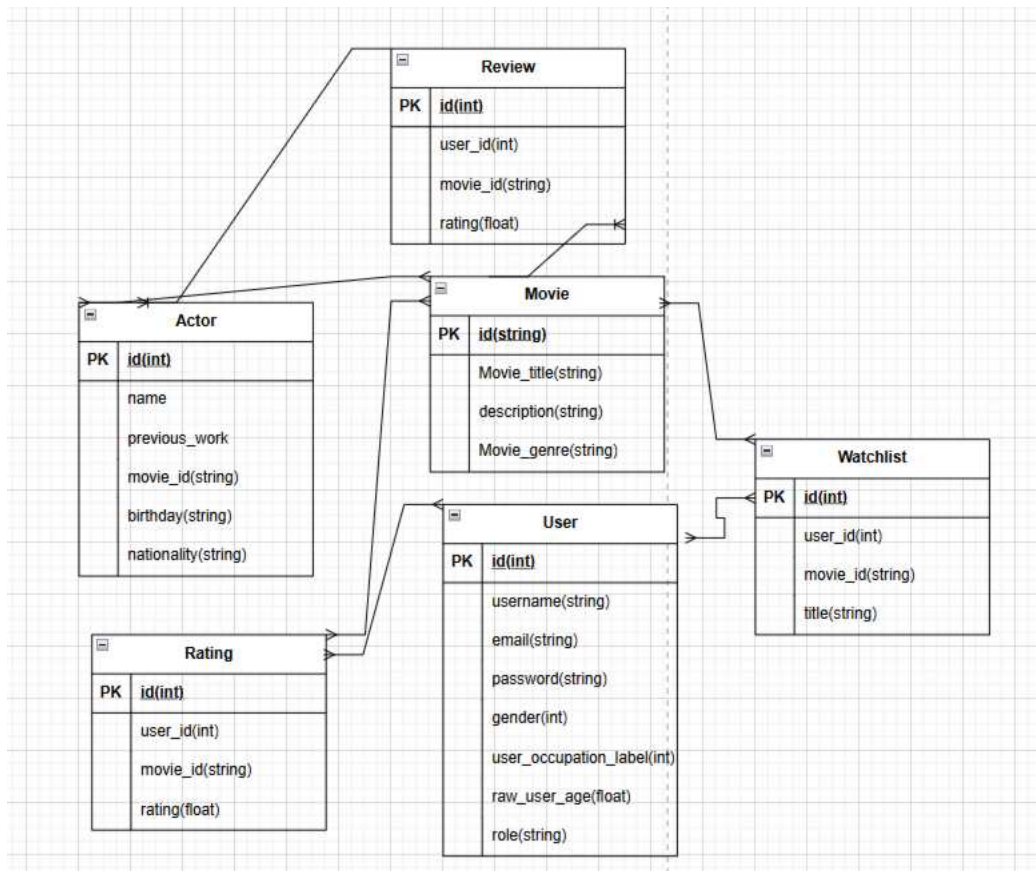
The SQL database stores the data for the movies, ratings, rankings, users, reviews, actors, and watchlists. The backed queries the database to get or modify the data.

### Machine Learning (TensorFlow)

The ranking model processes the user's data and the movie attributes to give a suggested movie recommendation based on how the model prediction on how a user will rate a movie.

## 3.2.5 Database design

Fig (3.1) Entity Relationship Diagram



User

- A user can have many ratings; a user can rate any movie.
- A user can have many watchlists.
- A user has a many-to-many relationship with movies through the watchlist table.

#### Movie

- A movie can appear in multiple watchlist, all users can add all movies to watchlist.
- A movie can have many ratings from users
- The rating table links the movie and user tables through storing the movie\_id and user\_id in ratings.
- Movie has a many-to-many relationship with actors.

#### Ratings

- A rating is associated with a user and a movie.  
Rating table holds both the user and movie ids.

#### Watchlist

- A watchlist stores a movie or multiple that a user has selected to be in their watchlist.
- Watchlist has a many-to-many relationship between user and movie.

#### Actor

- Actors can appear in multiple movies
- Has a many-to-many relationship with movies.

#### Review

- Reviews are associated with one user and one movie.

### 3.2.6 Process design

#### Class diagrams

A class diagram is used to visually represent the structure of the system by showing its classes, attributes, methods, and their relationships. A class diagram helps to understand the main entities, for this project it would be the User, Movie, Rating, and Watchlist.

#### User

- Attributes
  - Id
  - username
  - Email
  - Password
  - User\_Gender

- User\_Occupation\_Label
- User\_ratings
- Role
- Methods
  - Is an admin, is a user.
- Relationships
  - Can have many ratings and many watchlists.

## Movie

- Attributes
  - Id
  - Movie\_title
  - Description
  - Movie\_genres
- Methods
- Relationships
  - Can be in many watchlists and can have many ratings.

## Rating

- Attributes
  - Id
  - User\_id
  - Movie\_id
  - Rating
- Methods
- Relationships
  - Is associated with a user and movie through a one-to-many.

## Watchlist

- Attributes
  - Id
  - User\_id
  - Movie\_id
  - Title
- Methods
- Relationships
  - Is associated with user and movie through many-to-many.

## Actor

- Attributes
  - Id
  - Name
  - Description
  - Previous\_work
  - Birthday
  - Nationality
  - Movie\_id

- Methods
- Relationship
  - Has a many-to-many relationship with movies.

#### Review

- Attributes
  - Id
  - User\_id
  - Movie\_id
  - Content
- Methods
- Relationship
  - Is associated with user and movie in a one-to-many

#### Relationship Types

##### Movies and users

- One-to-Many:
  - Can have many ratings.
- Many-to-Many:
  - Can be in many watchlists, and a user can have many movies in their watchlists.
  - Can have many actors, and an actor can be in multiple movies.
  - Can rate multiple movies, and a movie can be rated by many users. (“Er Diagram for Movie Recommendation System | Restackio”)

### 3.3 User interface design

The user interface design for this application was done in Figma, the design focuses on a simple layout works smoothly, the design needed to be consistent throughout the pages and navigation was in mind as the pages were developed.

The designs process started with wireframes to outline the structure of the pages; they acted as a blueprint for the app displaying where I need to add in components such as where the movie card would display in the application.

#### Features in UI design

##### Home page

The home page will display the recommended movies from the model as well as showing rated movies that the users has rated. In this page you can navigate to the watchlist and public watchlists link where a user can view all watchlists and their own. This page will also allow users to navigate to the movie section where users can rate movies. In the home page users can view their recommended movies from the machine learning model where the user can click on them to add



them to their watchlist and rate it. The rated movies also appear in the home page and as a user rate more movies, more ratings will display in home, from this user can edit or delete their review on the movie.

#### Movie page

The movie pages will involve the movie all page where the users can view all the movies on the website. From this page an admin can create a new movie, users can search and filter through movies to find the specific movie they want to view. From this page the user and admin can click on a movie and then be taken to that specific movies details where an admin can edit the movie and delete it. A user from this single movie page can then add a movie to one of their watchlists and they can rate the movie they're viewing, and they can leave a review on the movie that other users can see, users can also see the actors in the movies where they can view their descriptions if they click into them.

Using the create and edit buttons takes the admin to a different form page where they can input the data to either create or edit a movie.

#### Watchlist page

Watchlist page will display all the watchlists a user has. The user can delete the watchlist they have, and they can edit the name of their watchlist. A user can click into one of their watchlists to see all the movies in their watchlist. From here they can remove movies from the list.

The users can also navigate to the public watchlist pages where all the public watchlists are displayed, from here the user can click into the public watchlist to view it and see what movies are in the watchlist that a different user has made.

#### Actor page

The actor pages the regular users can view is the actor single page where they will be able to view the data for the specific actor, for the actor all, edit, and delete pages they are locked by authentication and can only be accessed by the admin. In the actor all page the admin can delete, edit, and assign actors to movies, from this page the admin can click to view the actor single page.

#### PageNotFound page

This page is for if a user goes to a page that does not exist.

#### Profile page

In this page the user can view their data and sign-out.

#### Login and Register pages

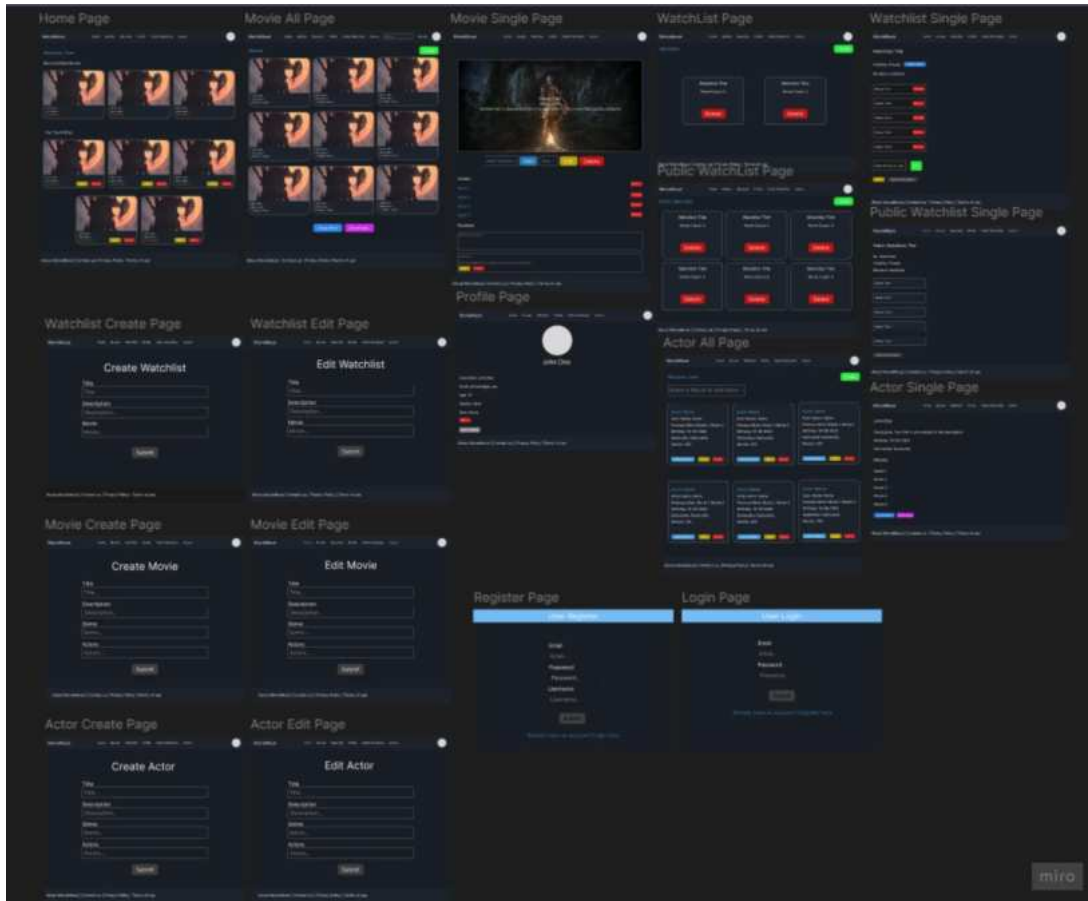
The login and register pages feature a form the users will fill in to gain authentication, the design will be simple for ease of use.

Figma is great to base your design off, it allows for direction when designing the front-end and it can help visualize what component will work in certain pages and where they will not make sense to be in the application.

### 3.3.1 Wireframe

Figma: <https://www.figma.com/design/PNu4ugz8GTH2mdozz9pOy6/Figma-for-Major-Project?node-id=0-1&t=3HBIWdz0rInqBXOd-1>

Fig (3.2) Wireframe



#### Register page

In this page the new user needs to register an account, the users will input their email field, password, gender, occupational\_label, and age. This will register a new user to the database and give them authentication to use the applications features.

#### Login Page

The Login page works similarly to the register apart from the fields that need to be inputted, the login will only require a user or admin to enter their email and passwords, this works because that specific user is already registered in the database and does not need to input extra fields.

#### Home Page

The home page will display the recommended movies from the ranking model as well as the rated movies the users have rated. From this page a user can navigate to the movies all page or the watchlists all page or their user profile.

#### User profile

This page is where a user can sign out of their account bringing them back to the login page.

### Movie pages

The movie consist of a movie all page where users and admins can view all the movies. From this page an admin can create a new movie. From this page the user and admin can click and view a single movie. In the single movie page, the user and admin and view the movie data and add the movie to one of their watchlists, they can also rate the movie from this page. As an admin in the single movie page, you can edit or delete the movie. Users can add ratings and reviews in this page along with adding the movie to their watchlist. Users can also view the actor single page by click on an actor in this movie single page.

### Watchlist pages

The watchlist pages consist of the watchlist all which displays all the watchlists a user has. A user can delete or create a new watchlist from this page. If a user selects one of the watchlists they can view all the movies that are in the watchlists, they can remove movies from the watch list from this page. The public watchlist pages will display as a link in the navbar that the user can navigate to, in this page the user can view watchlists from other users.

### Actor pages

The actor all page displays all the actors; admins can only view this page and here they can create actors, edit actors, and assign the actors to movies they are not already in. From this page you can click into the actor and see the description for the actor.

In the create and edit pages for actors the admin will have to fill out a form and them the actor will be created or edited.

### 3.3.2 User Flow Diagram

Fig (3.3) Use Case Diagram



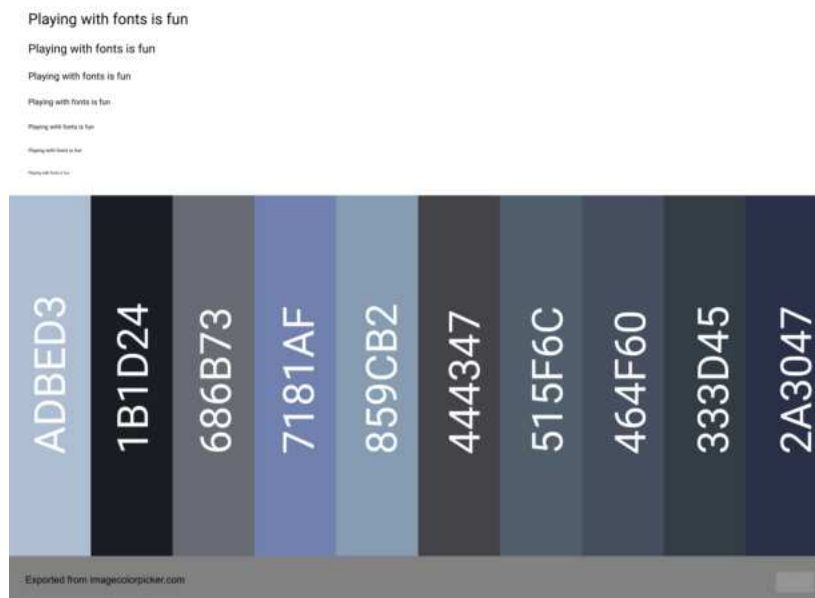
This use case diagram provides the visual representation of how different users interact with the system. ("How to Create a Use Case Diagram for a Chat ... - Diagram Central") It shows the core functionalities that are available to the user and the admin.

The diagram helps to understand the relationship between the user actions whether that is logging in, registering, viewing the movies, or managing their watchlists and the systems processes involving authentication, and managing the data.

### 3.3.3 Style guide

This shows the colours, and the font that was used in the project, the same style is implemented in all the pages. A colour pallet was made to show the colours used for the backgrounds, cards, buttons, and other components.

Fig (3.4) Style guide – Colours – Fonts



#### Colour Scheme

The colour scheme consists of cooler toned colours, primarily those being blues and greys to create a modern look for the application. The palette includes a range of colours:

Blue shades:

#ADBED3

#7181AF

#859CB2

#2A3047

Grey Shades:

#1B1D24

#686B73

#444347

#515F6C

#464F60

#333D45

Typography

Layouts, Grids

The application follows the grid system using 12 columns, this was done to ensure the alignment of the components where correct and that the wireframe would have consistent spacing, the margins and padding were added to avoid clutter and improve usability.

### 3.4 Conclusion

The design phase goes over the structure for the front and backend while also going into the designs of the elements for both. It goes over key elements such as the movie recommendations, admin capabilities. The designs for the front-end are developed showing a wireframe, style guide, and the structure the folder would have. It ensures that the designs are user-friendly with no over complication. This is the foundation of the application.

## 4 Implementation

### 4.1 Introduction

The application is a movie recommendation application; the system will give the users a recommendation suggestion based on what the model predicts the user rating will be. Users can register, log in, rate movies, add them to their watchlists, reviews the movies, and give the users recommendations. The recommendation is done through the basic ranking model from TensorFlow. The application consists of the following technologies.

- Flask
- React.js
- TensorFlow
- SQLite

Flask is used for the backend development. Flask allows for easy integration of TensorFlow models and will act as the API for the application handling requests and using queries to interact with the model extracting data from it to display in the front end for the user. The relationships between the tables are defined in the backend along with the logic of those attributes such as how movies are only able to be deleted by an admin user, if a movie is deleted it is deleted from any watchlist that movie appears in, and more.

React.js was used for the front-end of the application, this side is where the user will be able to interact with the components such as rating movies, reviewing movies, adding movies to their watchlist, and more. Through the API calls the data will come in through the backend and will display on the specific pages they are needed in. The application will allow users to use CRUD functionalities depending on the data and whether that user has admin privileges to access certain abilities not obtainable by normal users.

TensorFlow was used to get the pretrained model, basic ranking model predicts a user's ranking of a movie based on their interacts with the application and gives the user a recommendation based on those interacts and movie details.

SQLite is the database used in this application to store all the data for the application.

Here we outline the technologies that were used to develop this application, in the backend using Flask and in the front-end using React with a machine learning model that is queried to get the results for the users. This data is then stored in the SQLite database so it can then be displayed for the users when they are interacting with the application.

### 4.2 Scrum Methodology

In this project's development, the implementation phase involved seven sprints that lasted two weeks for each, the requirements for the project were gotten and documented in the projects backlog, this is a list of tasks that needed to be completed before moving on to the next sprint, these tasks were broken down to the simplest form to make the process of completing the tasks easier and organized which formed into the full sprint.

Managing the sprints involved meeting with the assigned mentor to discuss what tasks would need to be completed for the two-week window during development. This was done to keep the project organized and kept my mentor in the loop of what was being completed on the two-week basis. Miro was used to keep track of the sprints and to display the work that was completed to achieve the end results of the application.

At the end of each sprint tests were done to ensure that the projects requirements were met and worked as expected. For each sprint, the task was written for those two weeks and then the next so that when the time came to move on to the next sprints the tasks would already be known.

Through the sprint method it allowed for the application to be improved such as when challenges appeared when creating the application, such as the problem concerning the basic ranking model, improper relationship correlation in the models, version of Numpy used in the application, Cors errors, and more that have been documented. Adjustments to the project were made and tasks that were previously needed to be completed had to be changed to fit the new way the system would work for the users.

### 4.3 Development environment

#### Visual Studio Code

For both the front and backend of the application in this project, visual studio code was used to as the primary development environment. This was used for its easy GitHub integration, and it was the development environment that was most familiar. Visual studio code comes with benefits such as auto-completion for code, integrated terminal, and it manages extensions for Python, React, and Prettier making the process of coding the applications easier and improving the quality of the code and its format.

#### GitHub

GitHub was used for the version control and to store and manage the code for the application. Both the front and backend were initialized with GitHub to track the changes in the code; the code was then committed with messages describing what was changed and what was added with each commit.

GitHub was useful for keeping versions of the code that could be reverted to at any time, this was great when trying to implement more strenuous components when issue arose, and the problem was too difficult to solve at that time the project was reverted to a previous state so that other work could be completed, it was mostly utilized for the backend with logic routes not working as

intending. This helped the process of building up the application slowly with correct relationships and routes.

## 4.4 Sprint 1

### 4.4.1 Goal

For sprint 1 we needed to gather the items to then get the requirements that were going to be needed for this application. This began with a questionnaire to see how interested people would be in the application so we could gauge the type of user who would use an app like this and then the feedback was gathered to see the final opinions of the users.

With the feedback that was gathered by the questionnaire, interviews were conducted by a random assortment of people from many age ranges. The individuals were asked what pains they had to endure when using similar applications, they were also asked what type of implementation they would want to see in the application. The questionnaire and interviews helped to gather requirements that the application would need to have.

Similar web applications were then examined to see what functionality they used and what options the users had while interacting with the web applications. From this more requirements for the project were gathered.

The personas were then created to simulate a typical user. This was done to show the individuals that were likely to use the recommender application, the personas go into the users' personal goals, interests, behaviours, pains and gains, and their experiences using other applications. This help narrow down an age range for the app and allows others to visually see what kind of users the app is developed for.

A use case diagram was then constructed to show the system and how it depicts how a user would use the application and goes through the sections and what users can do these tasks. In the diagram made for this project it goes over how the user and admin can navigate through the application from when they login or register. It shows how each user can interact with different elements in the app and how to get to those areas.

After finishing the use case diagram, the functional requirements were completed which outlined the feature of the project and the tasks the user must complete to achieve their goals while using the app.

To get the non-functional requirements a prototype was made along with doing competitor analysis on similar applications such as Netflix, Prime, and MovieLens. These applications were examined to identify the how they handled load times, performance, and what components they had that users could interact with. Through the finding we can see that Prime performed the worst of the three websites, MovieLens is not the same application, but it has feature that can advance the functionality of the project, and Netflix performed the best. Through this the non-functional requirements were gathered to ensure the application runs smoothly.

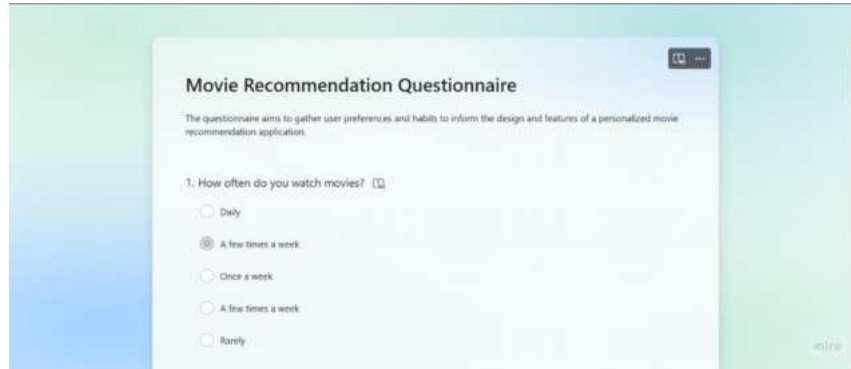
The prototype for the application was made to visually show a representation of the users experience when using the app, it is a low fidelity wireframe with mapped out the layout of important pages that would be needed for the application, this would include the home page, movie



all and movie single pages, login and register, and more. Making this prototype, users could give feedback allowing for refinement in the design moving forward.

#### 4.4.2 Item 1

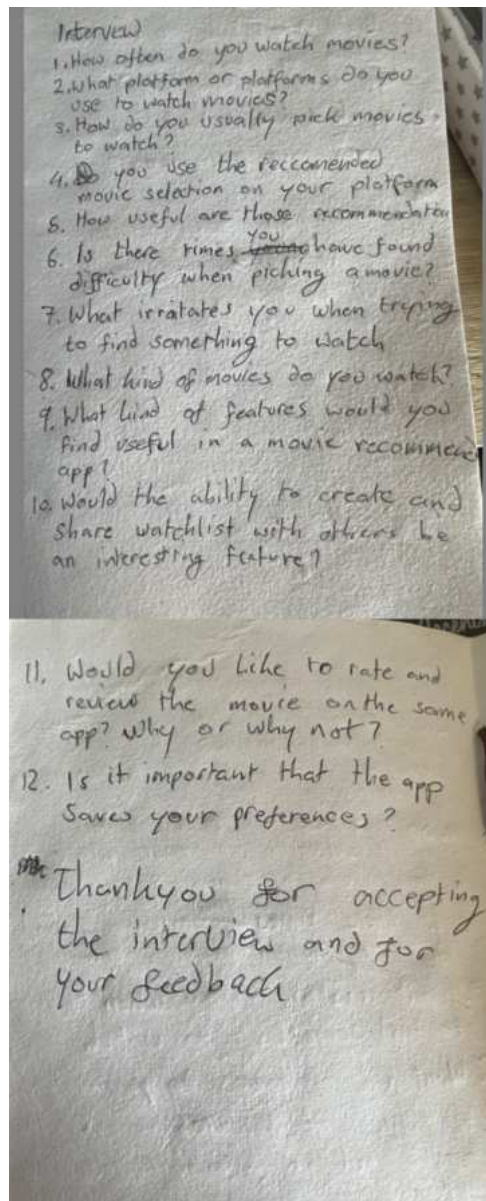
- Questionnaire  
Fig (4.0) Questionnaire

A screenshot of a web-based questionnaire titled "Movie Recommendation Questionnaire". The form has a light blue header with the title and a subtitle: "The questionnaire aims to gather user preferences and habits to inform the design and features of a personalized movie recommendation application." Below this, the first question is "1. How often do you watch movies?". It features five radio button options: "Daily", "A few times a week" (which is selected), "Once a week", "A few times a week" (repeated), and "Rarely". The form is displayed on a device screen with a green and blue gradient background.

- The questionnaire involved creating a list of questions that a potential user would complete, from this we could see the users' opinions on the initial idea for the project. The users could express their interests and from this review the feedback to conclude that people were interested in the application.
- This survey helped compile the necessary requirements that would be needed in the application.

#### 4.4.3 Item 2

- Interview  
Fig (4.1) Interview

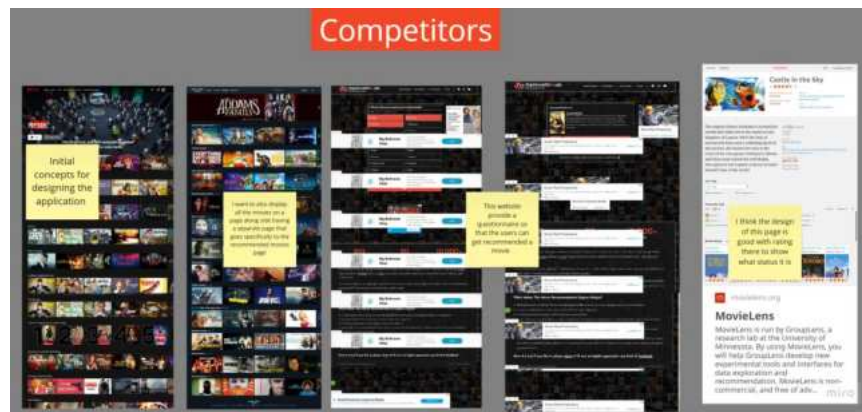


- Conducting the interviews allowed for a face-to-face conversation with many people from diverse backgrounds and at various stages of their life. In the interview the questions that were asked outlined any pains they felt while using streaming services. From this we were able to make more requirements the application needed for the application.
- The interviews were conducted one-on-one, and their answers were recorded to then be analysed, from the interviews gathered the feedback and compared them to the feedback from the questionnaire to determine what people thought about the project overall.

#### 4.4.4 Item 3

- Existing applications & Competitor analysis

Fig (4.2) Existing applications & Competitor analysis



- The existing applications were gathered to be compared and from this a list of requirements produced itself from the analysis of these three websites, this involved looking at the components they had implemented, the performance of the application, and what they have not done in their apps. From the analysis the consensus was that Netflix was the superior streaming service compared to Prime, and reviewing MovieLens as a movie review website gave inspiration for the project and multiple components would stem from the examination of that app.

#### 4.4.5 Item 4

- Persona

Fig (4.3) Personas

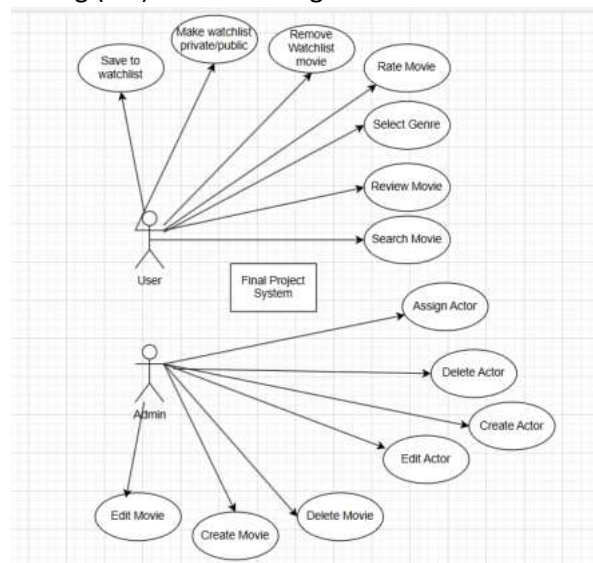


- A persona was created that would act as a normal user for the application, this was done to determine the age range of the users and their habits to see what the target audience would be. Using the interview and the questionnaire we could see what people were interested in the application and who were not. From that feedback the two personas were made which represent a typical user showing their pains, gains, and personality characteristics and what reason they would have for using the app.

#### 4.4.6 Item 4

- Use Case Diagram

Fig (4.4) Use Case Diagram



- The use case diagram represents the user's interactions with the system of the application that will be implemented later through the projects duration. It shows how the application will work in relation to the navigation and how the website will be accessed.

#### 4.4.7 Item 5

- Functional Requirements

Fig (4.5) Functional Requirements

# Functional Requirements

1. The application needs to provide personalized movie recommendations using machine learning models to give movies suggestions.
2. Implementation of the basic ranking methods from TensorFlow to sort and present movie recommendations based on the user's input.
3. Allow users to filter through genres to get movies based on that genre.
4. Allow users to search for specific movies in the search bar.
5. It needs to display a list of movies and give the admin the ability to delete, create, and edit.
6. Admin can interact with actors giving them the ability to assign actors to movies and remove them from movies. The admins can use CRUD functionality on the actors.
7. Users can leave a review on a movie, they can delete and edit their own reviews
8. Needs to allow users the ability to rate movies to improve recommendations. They can also edit and remove their ratings
9. Users need to be able to create, edit, and delete, and add their own watchlist.
10. Users can add their watchlists to public making it display in the public watchlists. Users can also remove their watchlists from public and make it private.

- The functional requirements outline what requirements would be needed for the users to achieve their goals when using the application. From conducting the interview, questionnaires, and showing the use case to potential users, the list of functional requirements was made. From this the prototype for the application could be made.

## 4.4.8 Item 6

- Non-Functional Requirements

Fig (4.6) Non-Functional Requirements

# Non-Functional Requirements

1. The applications usability should be user-friendly, intuitive, and allows users to navigate through the pages, movie recommendations, input preferences, and interactions with the system without problems.
2. The system should be able to load the recommendations quickly.
3. User data should be stored securely, using encryption for the password and other sensitive data that might need to be hidden.
4. The code should be well-organized, and easy to maintain allowing for updates
- 5.

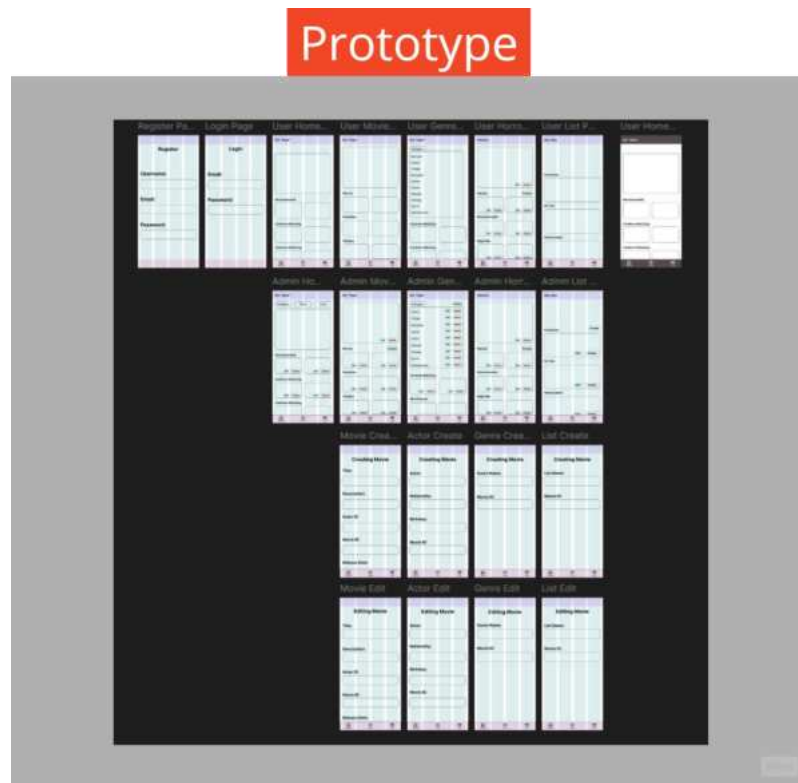
- Gathering the non-functional requirements the feedback from the users was examined to determine what expectations they have on what functions are needed for the overall application to be successful in terms of the performance security, and how usable the app would be.

- These requirements helped set a goal to achieve when developing the app making sure of the usability and how components render times might affect the users over experience when it comes to the performance, speed, and fidelity of the app.

#### 4.4.9 Item 7

- Prototype

Fig (4.7) Prototype



- A prototype was made to display how the application would look and function at its simplest form. This was done on Figma where the wireframe showed the login and register pages, the home page, the navigation, and other sections that were necessary for the application to have. The prototype was made and shown to potential users to get their feedback and thoughts on the design to see if any improvements needed to be made or feature that needed to be added.

## 4.5 Sprint 2

### 4.5.1 Goal

In sprint 2 the development on the design portion of the application was underway with designing the front end and the backend implementations that would be crucial for the application.

This sprint was split into four tasks to complete to achieve the vision for the application, these tasks included creating the ERD for the database schema, making the tables and columns, and identifying the relationships between these tables.

With the database the design was first drawn up to see what table would be needed initially starting with movie, user, and watchlist. Later more tables were implemented such as rating, reviews, and actors for more functionality. These tables all represent the data for the system to use to display in the front end to then be interacting with by the user as the change remove and add data to the database.

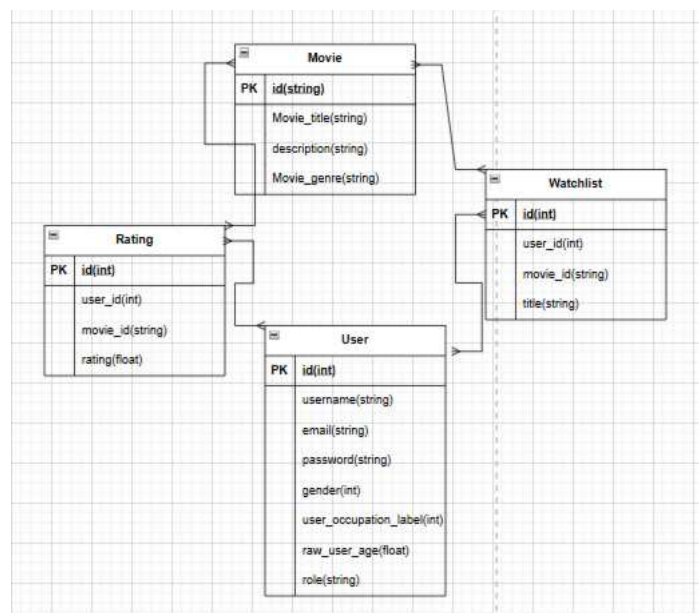
A wireframe was constructed to show a visual representation of what the application might look like outlining the pages and components on each of the pages. This involved adding features suggested by the users in the interview and questionnaire and implementing the components for the users to achieve their goals.

A user flow diagram was then constructed to show how the user would navigate through the various pages and what abilities would they have on each of those pages for the user to complete their tasks whilst using the application.

And finally, a style guide was created to show the colour scheme that was used in the application, the font style used. And why these colours and fonts were chosen.

#### 4.5.2 Item 1

- Database  
Fig (4.8) Erd Schema



- The database designs the tables were defined with their accompanying attributes, the tables were designed with the requirements of the application in mind meaning some tables were crucial to the project's success such as having a watchlist for users to store movies they would want to watch in the future, the movies in which the users could

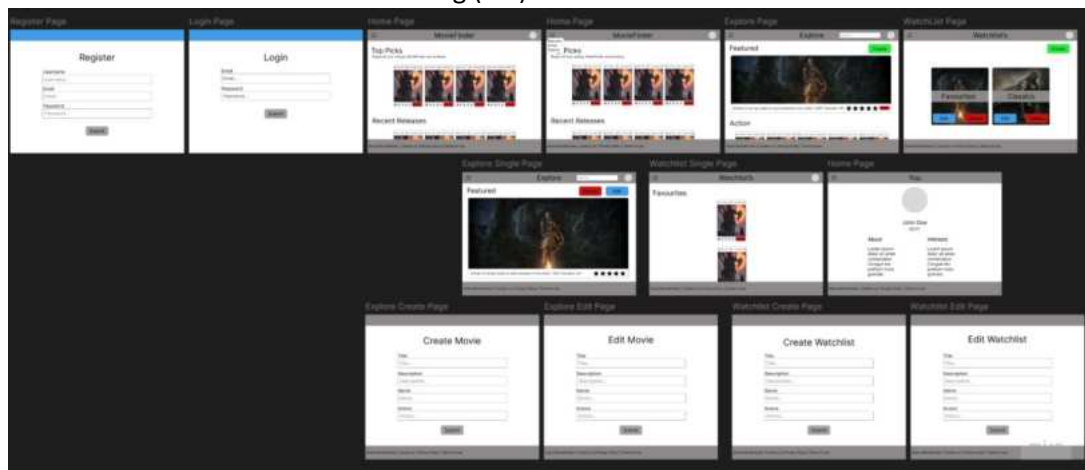


have those movies to view, rate, and add to their watchlist. These implementations were necessary to create the Erd for the database, from this defining the relationships between the tables were complete and the scheme was created.

#### 4.5.3 Item 2

- Wireframe

Fig (4.9) Wireframe



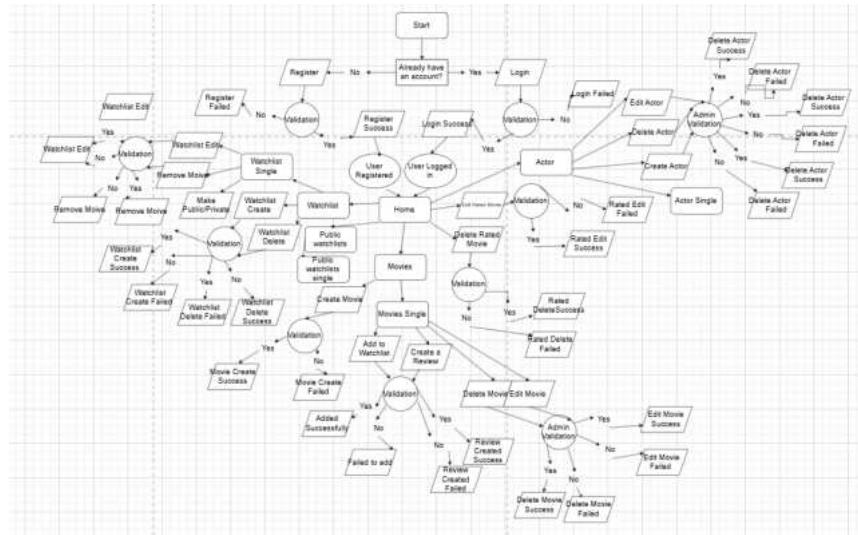
- The wireframe was created on Figma and structured it to look like a few websites that were previously examined in a competitor analysis to make the designs for the application. Elements of Netflix were used for the base pages such as the movie all page and the movie single page. Some of the inspiration for the components shown in the wireframe came from the MovieLens website which involved letting users' rate, review, and add movies to their watchlist where the user can choose to make their watchlist public or private. Other websites were looked at for their similarity to Netflix and movie Lens such as Disney plus and Prime for the pages on movies, and MyAnimeList for its similar functionality to MovieLens. From all these websites the basis for the wireframe was formed.

#### 4.5.4 Item 3

- User Flow Diagram

Fig (4.10) User Flow Diagram





- The user flow diagram was made to illustrate the steps that are involved for the user to navigate through the application and to identify the key actions the users can take in the app. The goal was to show the structure of the app and its navigation.

#### 4.5.5 Item 4

- Style Guide  
Fig (4.11) Style Guide



- The style guide was made to show the colour scheme and font used for the application that was used in the wireframe.

## 4.6 Sprint 3

In sprint 3 the tasks that were needed to complete this goal was to finally integrate the basic ranking model into the Flask backend, this involved setting up the Flask application and downloading the TensorFlow packages such as downloading the TensorFlow libraries and dataset libraries. From this it downloads NumPy for the data from the model to be processed.

Flask was set up as the backend framework because of its ease of use regarding integrating models. It was also chosen for its flexibility and simplicity when creating the logic for the routes that would be used to query the model and load it and define and give logic to the other routes used in the application. Initializing the backend involved setting up app.py to and then setting up the integration of GitHub, SQLite, and downloading the necessary libraries such as SQLAlchemy for the database for migrations and libraries.

Integrating the model from tensorflow involved using Google Colab to train and test the model and once it was functional it was then exported into a zip file with all necessary components and then it was transferred into the ml\_model folder in the Flask backend. Alongside the model's zip file, the folder contained the unzipped version of the model as well housing its variables and asset which was needed to load and run the model more efficiently. This structure allowed the model to be properly accessed and used when processing.

The next step after getting the model into the application was then to load the model to extract the data from it to be used in the front end to display recommended movies to users.

Once the model was in place the create of the models was underway, this included defining the models for the movies, users, watchlists, and ratings in the backend. The models store essential data for the user interactions and share attributes with the model for integration without any concerns for incompatibility. With the models in place the next step was to create the endpoints for the APIs to implements CRUD functionality and to define how that would work in the application.

The routes were then made to make the endpoints for the APIs, the routes in Flask act as the controllers as well so defining the logic of the endpoints also resides in these files and will serve the data to the database.

The next task was to create the seeders to serve data to the database. This populates the database with initial data, the data populated for the movies and users contain the data the model also uses for its predictions, making it use the same data for each.

### 4.6.1 Item 1

- Initializing Flask Application  
Fig (4.12) App.py in Flask

```

1 # app.py
2 from flask import Flask
3 from flask_cors import CORS
4 from flask_jwt_extended import JWTManager
5 from routes.movie_routes import movie_bp
6 from routes.ranking_routes import ranking_bp
7 from routes.watchlist_routes import watchlist_bp
8 from routes.rating_routes import rating_bp
9 from routes.auth import auth_bp
10 from routes.reviews_routes import review_bp
11 from routes.actor_routes import actor_bp
12 from extensions import db, migrate
13 from config.config import Config
14
15 def create_app(config_class=Config):
16     app = Flask(__name__)
17     app.config.from_object(config_class)
18     app.secret_key = 'your-secret-key-here'
19     app.config['JWT_SECRET_KEY'] = 'your-jwt-secret-key-here'
20     app.config['JWT_TOKEN_LOCATION'] = ['cookies', 'headers']
21     app.config['JWT_COOKIE_CSRF_PROTECT'] = True
22     app.config['JWT_ACCESS_TOKEN_EXPIRES'] = 3600
23
24     jwt = JWTManager(app)
25     CORS(app, supports_credentials=True, origins=["http://localhost:3000"])
26     print("CORS configured with origins: http://localhost:3000")
27     app.logger.debug("Starting Flask app")
28
29     db.init_app(app)
30     migrate.init_app(app, db)
31
32     app.register_blueprint(movie_bp, url_prefix='/movies')
33     app.register_blueprint(ranking_bp, url_prefix='/ranking')
34     app.register_blueprint(watchlist_bp, url_prefix='/watchlists')
35     app.register_blueprint(rating_bp, url_prefix='/ratings')
36     app.register_blueprint(auth_bp, url_prefix='/auth')
37     app.register_blueprint(review_bp, url_prefix='/reviews')
38     app.register_blueprint(actor_bp, url_prefix='/actors')
39
40     @app.route('/', methods=['GET'])
41     def hello_world():
42         return "Hello World"
43
44     return app
45
46 if __name__ == '__main__':
47     app = create_app()
48     with app.app_context():
49         db.create_all()
50     app.run(port=5000, debug=True)

```

- Flask was chosen as the backend, app.py was setup alongside downloading libraries like SQLAlchemy for database management and migrations. GitHub and SQLite were also initialized.

#### 4.6.2 Item 2

- Integrating the basic ranking model into Flask backend

Fig (4.13) imports

```

import os
os.environ['TF_USE_LEGACY_KERAS'] = '1'

```

- Importing os, this allows interaction with the operating system, which also includes setting the environment variables. The os.environ sets the TF\_USE\_LEGACY\_KERAS environment variable to 1.  
The reason for this is to tell TensorFlow to use an older version of Keras API and then setting the environment variables makes it compatible with the older TensorFlow Keras models.
- Initially this code was not in the pretrained model and cause it to fail when run because the Keras version was more updated so to get the model to run the version had to be put to an older one.

Fig (4.14) imports

```
!pip install -q tensorflow-recommenders
!pip install -q --upgrade tensorflow-datasets
```

- These two commands were installed and upgraded the essential libraries for building the recommendation system using the TensorFlow libraries.
- The tensorflow-recommenders helps to build the ranking model, and the tensorflow-datasets provide datasets, which make it easier to load the data for the training of the model.

Fig (4.15) imports

```
import os
import pprint
import tempfile
from google.colab import files
from typing import Dict, Text

import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds

import tensorflow_recommenders as tfrs
```

- Pprint provides a better more readable way to print the structured data, which is useful for debugging the model.
- Tempfile create a file directory that is used to store the results of the model.
- The google.colab.files import give utilities for uploading and downloading files in Google Colab which is use when exporting the model.
- Numpy uses for numerical computing, it is useful for handling array data.
- TensorFlow was downloaded as it is the core library for using machine learning and doing tasks involving machine learning.
- This sets up the environment with the ranking model.

Fig (4.16) imports

```
ratings = tfds.load("movielens/100k-ratings", split="train")

ratings = ratings.map(lambda x: {
    "movie_title": x["movie_title"],
    "user_id": x["user_id"],
    "user_rating": x["user_rating"]
})
```

}

- This code is loading the MovieLens 100K dataset using the TensorFlow dataset. This selects only the training split as well as mapping the dataset to extract the key features being the movie\_title, user\_id, and user\_rating to prepare the data for training the ranking model.

Fig (4.17) imports

```
tf.random.set_seed(42)
shuffled = ratings.shuffle(100_000, seed=42, reshuffle_each_iteration=False)

train = shuffled.take(80_000)
test = shuffled.skip(80_000).take(20_000)
```

- This code is setting a random seed, so it is suitable for reproducibility, it shuffles the dataset and splits it into testing and training sets.
- The `tf.random.set_seed` sets the random seed, initially it is 42 to make sure that it is consistent for different runs.
- `shuffled = ratings.shuffle(100_000, seed=42)` shuffles the dataset with a buffer size of one hundred thousand using the same 42 seed.
- The `reshuffle_each_iteration=False` makes sure it is only shuffled once to keep consistency across epochs.
- `Train = shuffled.take(80_000)` takes the first eighty thousand samples from the dataset to be trained in the training set.
- `Test = shuffled.skip(80_000).take(20_000)` skips the first eighty thousand and takes the next twenty thousand into the testing set.

Fig (4.18) imports

```
movie_titles = ratings.batch(1_000_000).map(lambda x: x["movie_title"])
user_ids = ratings.batch(1_000_000).map(lambda x: x["user_id"])

unique_movie_titles = np.unique(np.concatenate(list(movie_titles)))
unique_user_ids = np.unique(np.concatenate(list(user_ids)))
```

- This code is processing the MovieLens dataset to extract and identify the unique `movie_titles` and `user_ids`.
- `movie_titles = ratings.batch(1_000_000).map(lambda x: x["movie_title"])` batches the dataset with one million elements at one time to process the movie titles in the dataset, it uses `map()` to extract the data from each element in the dataset.
- `user_ids = ratings.batch(1_000_000).map(lambda x: x["user_id"])` does the same, extracting the user ids from the dataset one million elements at one time.
- `unique_movie_titles = np.unique(np.concatenate(list(movie_titles)))` puts all the movie titles into an array
- `unique_user_ids = np.unique(np.concatenate(list(user_ids)))` puts all the user ids into an array.

Fig (4.19) imports

```
class RankingModel(tf.keras.Model):
    def __init__(self):
        super().__init__()
        embedding_dimension = 32

        # Compute embeddings for users.
        self.user_embeddings = tf.keras.Sequential([
            tf.keras.layers.StringLookup(
                vocabulary=unique_user_ids, mask_token=None),
            tf.keras.layers.Embedding(len(unique_user_ids) + 1, embedding_dimension)
        ])

        # Compute embeddings for movies.
        self.movie_embeddings = tf.keras.Sequential([
            tf.keras.layers.StringLookup(
                vocabulary=unique_movie_titles, mask_token=None),
            tf.keras.layers.Embedding(len(unique_movie_titles) + 1, embedding_dimension)
        ])

        # Compute predictions.
        self.ratings = tf.keras.Sequential([
            # Learn multiple dense layers.
            tf.keras.layers.Dense(256, activation="relu"),
            tf.keras.layers.Dense(64, activation="relu"),
            # Make rating predictions in the final layer.
            tf.keras.layers.Dense(1)
        ])

    def call(self, inputs):
        user_id, movie_title = inputs

        user_embedding = self.user_embeddings(user_id)
        movie_embedding = self.movie_embeddings(movie_title)

        return self.ratings(tf.concat([user_embedding, movie_embedding], axis=1))
```

- This code defines the RankingModel, is defines it as a neural network-based ranking model that will predict the movie ratings using the TensorFlow Keras API. The code initializes the user and movie embedding with the StringLookUp layer and the embedding layer.
- A neural network with dense layers is used then to process the embedding and then predicts the rating. The call method retrieves the movie and user embedding, it passes through the network to then generate the ratings.

Fig (4.20) imports

```
RankingModel()([["42"], ["One Flew Over the Cuckoo's Nest (1975)"]])
```

- This code is testing the RankingModel by predicting a rating for the user and the movie.

Fig (4.21) imports

```
task = tf.keras.tasks.Ranking(
    loss = tf.keras.losses.MeanSquaredError(),
    metrics=[tf.keras.metrics.RootMeanSquaredError()]
)
```

- This is defining a task for training the model using the TensorFlow recommenders.
- The loss = tf.keras.losses.MeanSquaredError() uses Mean Squared Error loss function, it measures how far the predictions are from the rating.
- metrics=[tf.keras.metrics.RootMeanSquaredError()] is tracking the Root Mean Square Error to assess the model's performance.

Fig (4.22) imports

```

class MovieLensModel(tfrs.models.Model):
    def __init__(self):
        super().__init__()
        self.ranking_model: tf.keras.Model = RankingModel()
        self.task: tf.keras.layers.Layer = tfrs.tasks.Ranking(
            loss = tf.keras.losses.MeanSquaredError(),
            metrics=[tf.keras.metrics.RootMeanSquaredError()]
        )

    def call(self, features: Dict[str, tf.Tensor]) -> tf.Tensor:
        return self.ranking_model(
            features["user_id"], features["movie_title"])

    def compute_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor:
        labels = features.pop("user_rating")

        rating_predictions = self(features)

        # The task computes the loss and the metrics.
        return self.task(labels=labels, predictions=rating_predictions)

```

- This is how the model defines the MovieLensModel, it initializes the RankingModel and the task previously stated, it calculates the loss and tracks it as a metric. the call method the user and movie title feature and passes them through the RankingModel to generate the predictions. The compute\_loss method extracts the user rating label then makes the prediction then computes the loss using the task.

Fig (4.23) imports

```

model = MovieLensModel()
model.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.1))

```

- This is initializing and compiling the MovieLens model for training.
- model = MovieLensModel() is creating an instance of the MovieLensModel and configures it for training.
- model.compile(optimizer=tf.keras.optimizers.Adagrad(learning\_rate=0.1)) uses the Adagrad Optimizer with a learning rate of 0.1 which is done to improve the convergence and adapts the learning rate for each parameter.

Fig (4.24) imports

```

cached_train = train.shuffle(100_000).batch(8192).cache()
cached_test = test.batch(4096).cache()

```

- This is preparing the training and testing datasets for processing, the training data is then shuffled with a buffer of one hundred thousand and then batched into groups of eight thousand one hundred and ninety-two. It then cached to speed by training keeping redundant data out. The test data is batched into a group of four thousand and ninety-six and then it is cached to optimize the evaluation performance.

Fig (4.25) training the model with three epochs

```

model.fit(cached_train, epochs=3)

```

- This code is training the model on the cached training dataset for three epoch.

Fig (4.26) Evaluating the model

```

model.evaluate(cached_test, return_dict=True)

```



- This code evaluates the performance of the trained machine learning model on the test dataset and then it returns the evaluation.

Fig (4.27) Predicting the rating of one user

```
test_ratings = {}
test_movie_titles = ["M*A*S*H (1970)", "Dances with Wolves (1990)", "Speed (1994)"]
for movie_title in test_movie_titles:
    test_ratings[movie_title] = model({
        "user_id": np.array(["42"]),
        "movie_title": np.array([movie_title])
    })
print("Ratings:")
for title, score in sorted(test_ratings.items(), key=lambda x: x[1], reverse=True):
    print(f"{title}: {score}")
```

- This uses the trained model to predict the movie ratings for one user and does this across a set of movies, it then displays the movie and the predicting rating and ranks them from highest to lowest.
- Test\_ratings is an empty directory stores the predicted ratings for each of the movies.
- Test movie titles is a list of movie titles where the ratings will be predicted.
- The code then iterates through each of the movie titles in the test movie titles list, it then goes through each movie calling the model with a directory containing the users id, from this the model performs the prediction and returns the rating. It is then stored in the test\_rating array.

Fig (4.28) Saving the model

```
tf.saved_model.save(model, "ranking_model")
```

- This saves the model which can later be loaded and used.

Fig (4.29) Loading the model

```
loaded = tf.saved_model.load("ranking_model")
loaded({"user_id": np.array(["42"]), "movie_title": ["Speed (1994)"]}).numpy()
```

- This code is loading the previous saved model and then uses it to make a prediction for a single specific movie and user; it then is converting the results to a numerical form.

Fig (4.30) Converting the model to TFLite

```
converter = tf.lite.TFLiteConverter.from_saved_model("ranking_model")
tflite_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_model)
```

- This is converting the saved model into a smaller TFLite model, it then saves it to a file name. This file can then be deployed.

Fig (4.31) How to execute the TFLite model in python



```

interpreter = tf.lite.Interpreter(model_path="converted_model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Test the model.
if input_details[0]["name"] == "serving_default_movie_title:0":
    interpreter.set_tensor(input_details[0]["index"], np.array(["Speed (1994)"]))
    interpreter.set_tensor(input_details[1]["index"], np.array(["42"]))
else:
    interpreter.set_tensor(input_details[0]["index"], np.array(["42"]))
    interpreter.set_tensor(input_details[1]["index"], np.array(["Speed (1994)"]))

interpreter.invoke()

rating = interpreter.get_tensor(output_details[0]["index"])
print(rating)

```

- This code is loading a converted TFLite model, it prepares the model for inference, then sets the inputs tensors with the user and movie data. It then runs the inference and prints the predicted ratings. This is a demonstration on how to execute the TFLite model in python.

Fig (4.32) Compressing content into a zip

```
!tar -czvf ranking_model.tar.gz ranking_model/
```

- This is the ranking\_model that was previously saved and compressing its contents into a zip file where it was then downloaded and put into the ml\_models folder.
- Difficulties arose when trying to export the model into a zip file, the difficulty stemmed from not having the correct code or miss typing the name on the model, all issue were solved, and the model was able to be put into the Flask backend.

## 5.6.2 item 3

- Querying the Model
  - The Flask API endpoints load the ranking model, it then retrieves the movie data from the database, predicts the ranking for a specific user, and then returns the top five recommended movies as a JSON response.

Fig (4.33) Imports

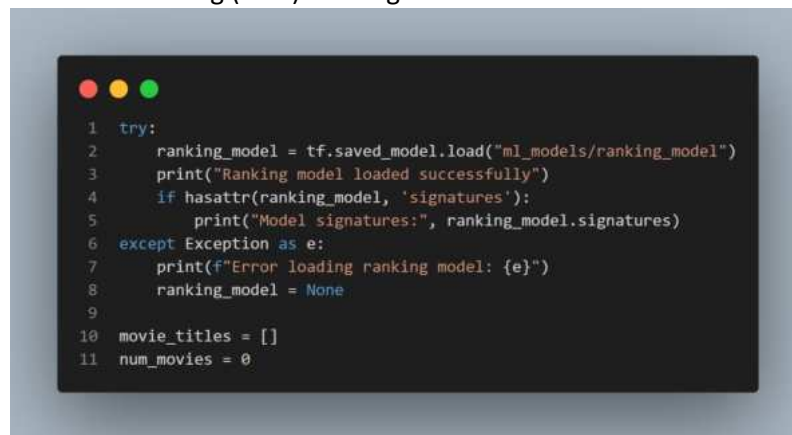
```

1 from flask import Blueprint, request, jsonify
2 import tensorflow as tf
3 import numpy as np
4 from extensions import db
5 from models.movie import Movie
6

```

- Importing the necessary libraries such as the Flask components, TensorFlow, NumPy, and the database models. It then creates a Flask Blueprint named ranking\_bp which will be called in the app.py where the route can be loaded.

Fig (4.34) Loading the model



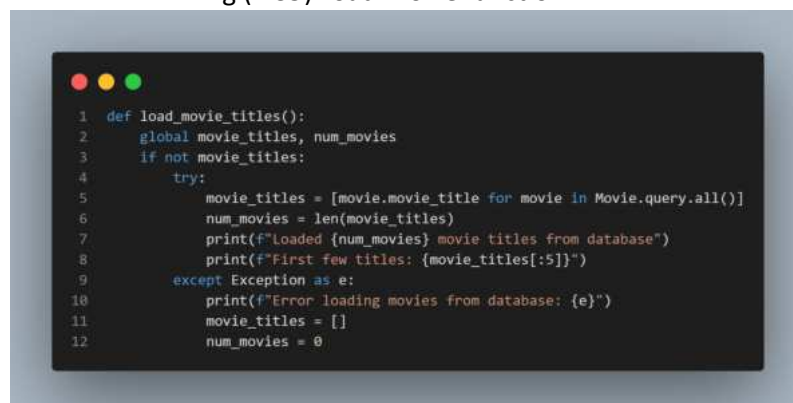
```

1  try:
2      ranking_model = tf.saved_model.load("ml_models/ranking_model")
3      print("Ranking model loaded successfully")
4      if hasattr(ranking_model, 'signatures'):
5          print("Model signatures:", ranking_model.signatures)
6  except Exception as e:
7      print(f"Error loading ranking model: {e}")
8      ranking_model = None
9
10 movie_titles = []
11 num_movies = 0

```

- It then loads the TensorFlow model printing a success message when loaded properly and an error message if not. It initializes the movie\_titles as an empty list.

Fig (4.35) Load Movie function



```

1  def load_movie_titles():
2      global movie_titles, num_movies
3      if not movie_titles:
4          try:
5              movie_titles = [movie.movie_title for movie in Movie.query.all()]
6              num_movies = len(movie_titles)
7              print(f"Loaded {num_movies} movie titles from database")
8              print(f"First few titles: {movie_titles[:5]}")
9          except Exception as e:
10             print(f"Error loading movies from database: {e}")
11             movie_titles = []
12             num_movies = 0

```

- The load movie titles function fetches all the movie titles from the database using the Movie.query.all function, it then stores the movie\_titles in a list and sets the num\_movies to be the number of movie titles fetched.

Fig (4.36) Get Top Ranked Movies function

```
1 @ranking_bp.route('/', methods=['GET'])
2 def get_top_ranked_movies():
3     load_movie_titles()
4
5     print(f"Request method: {request.method}")
6     print(f"Request headers: {request.headers}")
7     print(f"Request body: {request.get_data(as_text=True)}")
8
9     json_data = request.get_json(silent=True)
10    if json_data and 'user_id' in json_data:
11        user_id = json_data['user_id']
12    else:
13        user_id = request.args.get('user_id')
14        if not user_id:
15            return jsonify({'error': 'user_id is required in body or query'}), 400
16    0
```

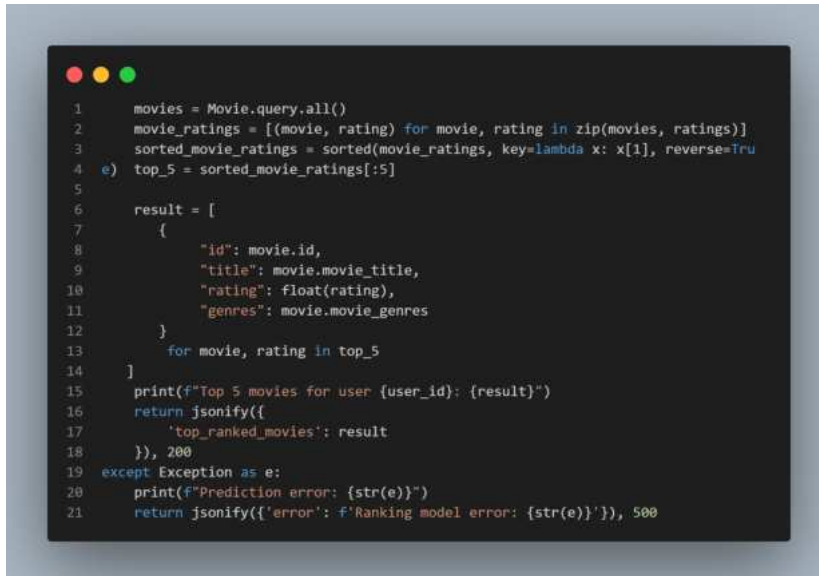
- The `get_top_ranked_movies` function handles the GET request to the ranking route Blueprint. It then calls the `load_movie_titles` function to make sure the movie titles are loaded; it then retrieves the users' ids from the request from the JSON body and performs an input validation to ensure the user id comes in as a number. Error messages occur if ranking model or the movie titles are not loading properly.

Fig (4.37) Extracting data

```
1 json_data = request.get_json(silent=True)
2 if json_data and 'user_id' in json_data:
3     user_id = json_data['user_id']
4 else:
5     user_id = request.args.get('user_id')
6     if not user_id:
7         return jsonify({'error': 'user_id is required in body or query'}), 400
8 0
9 try:
10    user_id = str(int(user_id))
11 except ValueError:
12    return jsonify({'error': 'user_id must be an integer'}), 400
13
14 if ranking_model is None:
15    return jsonify({'error': 'Ranking model not available'}), 500
16 if not movie_titles:
17    return jsonify({'error': 'Movie titles not available'}), 500
18
19 user_ids = tf.constant([user_id] * num_movies, dtype=tf.string)
20 movie_titles_tensor = tf.constant(movie_titles, dtype=tf.string)
21 input_data = {
22     "user_id": user_ids,
23     "movie_title": movie_titles_tensor
24 }
25
26 try:
27     predictions = ranking_model(input_data)
28     ratings = predictions.numpy().flatten()
```

- It creates TensorFlow tensors for the user ids and the movie titles which replicates the user's id and checks them against all the movies. The ranking model is then called with the input data to get the predictions; it then extracts all the ratings from the prediction and combines the ratings with the movie object that was retrieved from the database.

Fig (4.38) Top 5 ranking



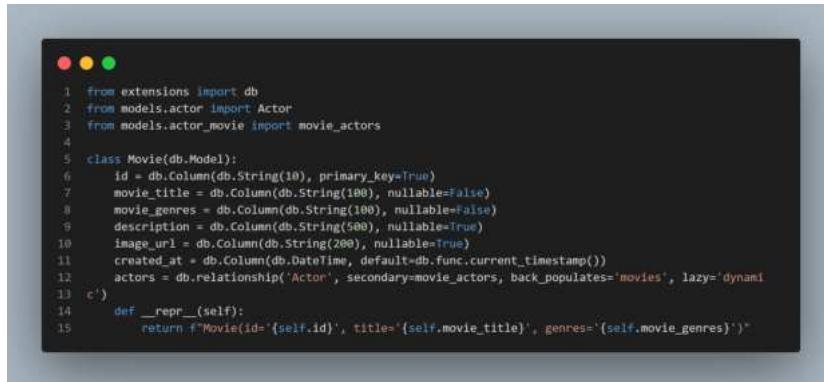
```
1  movies = Movie.query.all()
2  movie_ratings = [(movie, rating) for movie, rating in zip(movies, ratings)]
3  sorted_movie_ratings = sorted(movie_ratings, key=lambda x: x[1], reverse=True)
4  e) top_5 = sorted_movie_ratings[:5]
5
6  result = [
7      {
8          "id": movie.id,
9          "title": movie.movie_title,
10         "rating": float(rating),
11         "genres": movie.movie_genres
12     }
13     for movie, rating in top_5
14 ]
15 print(f"Top 5 movies for user {user_id}: {result}")
16 return jsonify({
17     'top_ranked_movies': result
18 }, 200)
19 except Exception as e:
20     print(f"Prediction error: {str(e)}")
21     return jsonify({'error': f'Ranking model error: {str(e)}'}), 500
```

- It then sorts the movie and rating pairs in descending order to then list the top five highest rated movies; it then makes a JSON response with the top five movies. This can then be shown to existing users to show them their recommended movies.
- It returns a 200 response if all successful which includes error handling if problems occurred.
- The difficulties getting this model to extract data and get predictions was a difficult venture as trying to get all the querying right posed a challenge when having to convert values from integers to strings and vice versa when the model is trying to read the data in a certain way. Other major problems occurred referring to the second model that also resides in the ml\_model folder, the two models were successfully extracting, those being the basic ranking model, and the basic retrieval model. Both models together form to make the full recommendation model retrieve both ranked data predictions and basic correlated recommendations based on the user's profile and other interactions. The problem with the basic retrieval model was that the version of numpy used in the application was not compatible with the model. Efforts were made to lower the version of numpy with little success, even when switched to the specific version of numpy it needed.
- The second problem with the models, specifically the ranking model was that when it is trained it only uses the dataset that is in the model and not my data from the database. The model cannot take in new data and make a prediction based on that it already has all the predictions previously made and cannot add more to it. So, the problem is the recommendations can display and is linked to the users in the actual SQLite database, but it is not accurate predictions for their account choices, preferences, or interactions.

### 5.6.3 item 4

- Models

Fig (4.39) Movie Model

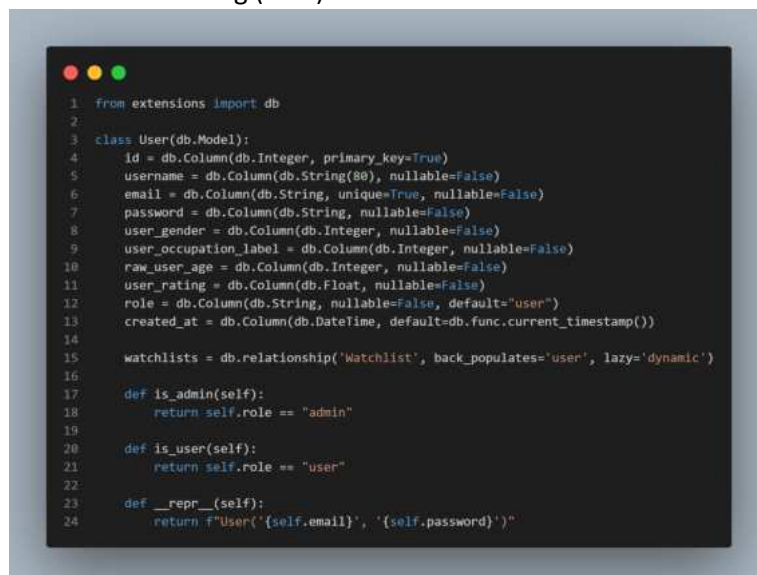


```

1 from extensions import db
2 from models.actor import Actor
3 from models.actor_movie import movie_actors
4
5 class Movie(db.Model):
6     id = db.Column(db.String(10), primary_key=True)
7     movie_title = db.Column(db.String(100), nullable=False)
8     movie_genres = db.Column(db.String(100), nullable=False)
9     description = db.Column(db.String(500), nullable=True)
10    image_url = db.Column(db.String(200), nullable=True)
11    created_at = db.Column(db.DateTime, default=db.func.current_timestamp())
12    actors = db.relationship('Actor', secondary=movie_actors, back_populates='movies', lazy='dynamic')
13
14    def __repr__(self):
15        return f'Movie(id='{self.id}', title='{self.movie_title}', genres='{self.movie_genres}')
```

- This is the movie model; it represents the movies in the database.
- The id is a unique identifier for all the movies, it is the primary key and is set as a string so that the model can read the movie ids and so that they can be put into an array for watchlists
- Movie title field for all the movie titles, it is set as a string.
- The movie genres are also set as a string and stores the genres.
- The description stores the description data for the movies. Set as a string.
- Movie has a one-to-many relationship with ratings

Fig (4.40) User model



```

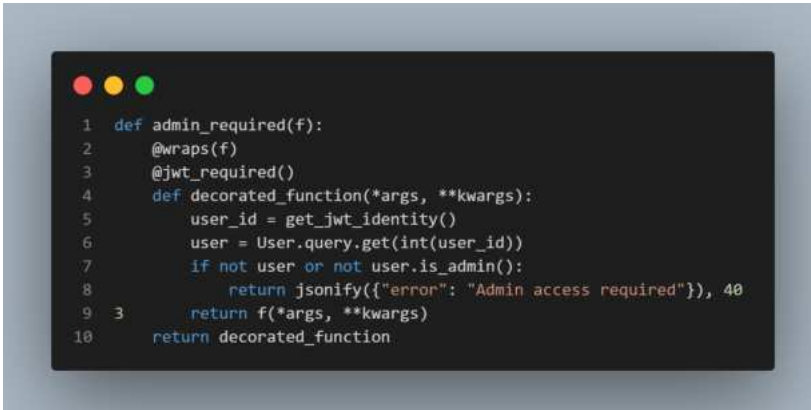
1 from extensions import db
2
3 class User(db.Model):
4     id = db.Column(db.Integer, primary_key=True)
5     username = db.Column(db.String(80), nullable=False)
6     email = db.Column(db.String, unique=True, nullable=False)
7     password = db.Column(db.String, nullable=False)
8     user_gender = db.Column(db.Integer, nullable=False)
9     user_occupation_label = db.Column(db.Integer, nullable=False)
10    raw_user_age = db.Column(db.Integer, nullable=False)
11    user_rating = db.Column(db.Float, nullable=False)
12    role = db.Column(db.String, nullable=False, default="user")
13    created_at = db.Column(db.DateTime, default=db.func.current_timestamp())
14
15    watchlists = db.relationship('Watchlist', back_populates='user', lazy='dynamic')
16
17    def is_admin(self):
18        return self.role == "admin"
19
20    def is_user(self):
21        return self.role == "user"
22
23    def __repr__(self):
24        return f'User('{self.email}', '{self.password}')
```

- This is the user model; it represents the users in the database.
- The id is auto incremented and is the primary key, it is set as an integer.
- The username stores the username data in the database. Set as a string.
- The email is set as unique so other users can use the same emails. Set as a string and is used to login and register accounts.
- The password stores the password data in the database; it is hashed for security and set as a string.
- The user gender allocates whether the user is a man or women. It is set as an integer and would have been a contributing factor in the model's predictions if it worked as intended.

- The user\_occupation\_label was also a factor for the retrieval model; this is set as an integer.
- The raw\_user\_age was a factor for the retrieval model; it is set as an integer and represents the ages of the users.
- User\_rating would have been used with the ranking model to predict rankings on the movies; we can simulate how it was supposed to work. This field is set as a float.
- And finally, role is used to set the user as either an admin or a user, admin have certain privileges they can perform that the normal user cannot.
- Is admin and is user are helper methods in the user model which returns the user true if they are either an admin or user.
- User has a one-to-many relationship with watchlists
- User has a one-to-many relationship with ratings

#### 5.6.4 item 5

- Routes  
Fig (4.41) Auth Route



```

1 def admin_required(f):
2     @wraps(f)
3     @jwt_required()
4     def decorated_function(*args, **kwargs):
5         user_id = get_jwt_identity()
6         user = User.query.get(int(user_id))
7         if not user or not user.is_admin():
8             return jsonify({"error": "Admin access required"}), 403
9         return f(*args, **kwargs)
10    return decorated_function

```

- The admin required function was made to ensure that only admins have access to certain routes in the application it wraps around a function and first requires a JWT token authentication. When the user logs in the id are then extracted from the JWT and if the id corresponds to with the id in the database and if that user is an admin the route will work, if not an error response will appear for the user.

Fig (4.42) Register

```

1 @auth_bp.route('/register', methods=['POST'])
2 def register():
3     data = request.get_json()
4     if not data:
5         return jsonify({"error": "Request body must be JSON"}), 400
6
7     required_fields = ["username", "email", "password", "user_gender", "user_occupation_label", "raw_user_age"]
8     missing_fields = [field for field in required_fields if field not in data or data[field] is None]
9     if missing_fields:
10        return jsonify({"error": f"Missing required fields: {', '.join(missing_fields)}"}), 400
11
12    username = data["username"]
13    email = data["email"]
14    password = data["password"]
15    user_gender = data["user_gender"]
16    user_occupation_label = data["user_occupation_label"]
17    raw_user_age = data["raw_user_age"]
18
19    if len(password) < 8:
20        return jsonify({"error": "Password must be at least 8 characters"}), 400

```

```

1 try:
2     user_gender = int(user_gender)
3     user_occupation_label = int(user_occupation_label)
4     raw_user_age = int(raw_user_age)
5 except ValueError:
6     return jsonify({"error": "user_gender, user_occupation_label, and raw_user_age must be integers"}), 400
7
8 if User.query.filter_by(email=email).first():
9     return jsonify({"error": "Email already registered"}), 400
10
11 hashed_password = bcrypt.generate_password_hash(password).decode('utf-8')
12 new_user = User(
13     username=username,
14     email=email,
15     password=hashed_password,
16     user_gender=user_gender,
17     user_occupation_label=user_occupation_label,
18     raw_user_age=raw_user_age,
19     user_rating=0.0,
20     role="user"
21 )

```

```

1 try:
2     db.session.add(new_user)
3     db.session.commit()
4     access_token = create_access_token(identity=str(new_user.id))
5     response = jsonify({
6         "message": "Registration successful",
7         "user_id": str(new_user.id),
8         "access_token": access_token
9     })
10    set_access_cookies(response, access_token)
11    return response, 201
12 except Exception as e:
13     db.session.rollback()
14    return jsonify({"error": f"Failed to register: {str(e)}"}), 500

```

- The register route handles the registration for a user, the request body check to see if the data is JSON and if the required field have been filled in correctly and are valid. If the fields are not valid the system responds with an error message. The email is unique and give an error if the user tries to have the same email as another user, the password is hashed and must be at least eight characters long if the POST request was successful the user will be given a token set in the cookie response and the user will be registered in the database, if this operation fails an error message will occur.

Fig (4.43) Login

```
1 @auth_bp.route("/login", methods=["POST"])
2 def login():
3     data = request.get_json()
4     if not data:
5         return jsonify({"error": "Request body must be JSON"}), 400
6
7     email = data.get("email")
8     password = data.get("password")
9
10    if not email or not password:
11        return jsonify({"error": "Email and password are required"}), 400
12
13    user = User.query.filter_by(email=email).first()
14    if user and bcrypt.check_password_hash(user.password, password):
15        access_token = create_access_token(identity=str(user.id))
16        print(f"Generated token for user {user.id}: {access_token}")
17        response = jsonify({
18            "message": "Login successful",
19            "user_id": str(user.id),
20            "role": user.role,
21            "access_token": access_token
22        })
23        set_access_cookies(response, access_token)
24        return response, 200
25    return jsonify({"error": "Invalid email or password"}), 401
```

- The login route handles the user's authentication with a POST request, similarly to the register route it checked the validation of the email and password fields, this looks up the user in the database and if they are there it signs in the user. It generates the logged in user a token for authentication.

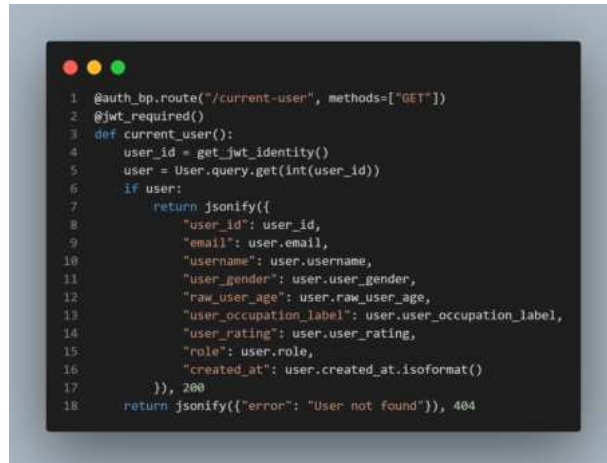
Fig (4.44) Logout

```
1 @auth_bp.route("/logout", methods=["POST"])
2 def logout():
3     response = jsonify({"message": "Successfully logged out"})
4     unset_jwt_cookies(response)
5     return response, 200
```

- The user logout route gives the user the ability to sign out of their accounts removing the token they had and returning them to the login.



Fig (4.45) Current User

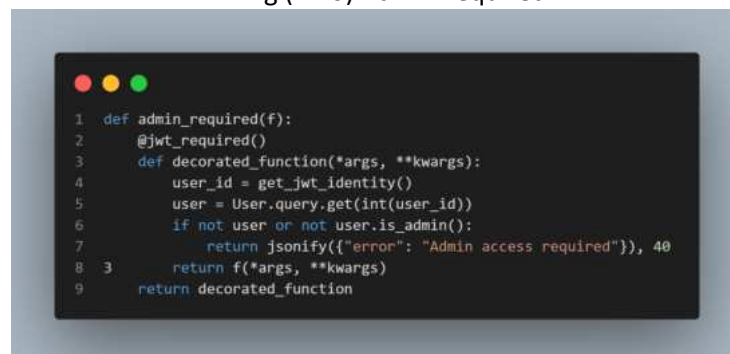


```
1 @auth_bp.route("/current-user", methods=["GET"])
2 @jwt_required()
3 def current_user():
4     user_id = get_jwt_identity()
5     user = User.query.get(int(user_id))
6     if user:
7         return jsonify({
8             "user_id": user_id,
9             "email": user.email,
10            "username": user.username,
11            "user_gender": user.user_gender,
12            "raw_user_age": user.raw_user_age,
13            "user_occupation_label": user.user_occupation_label,
14            "user_rating": user.user_rating,
15            "role": user.role,
16            "created_at": user.created_at.isoformat()
17        }), 200
18     return jsonify({"error": "User not found"}), 404
```

- The current user route fetches the information about the currently authenticated user. It extracts the user's id from the JWT and gets the corresponding user from the database. It displays the user's details outlined in the model. If no users were found it responds with an error message stating no users were found.

#### Movie Route

Fig (4.46) Admin required



```
1 def admin_required(f):
2     @jwt_required()
3     def decorated_function(*args, **kwargs):
4         user_id = get_jwt_identity()
5         user = User.query.get(int(user_id))
6         if not user or not user.is_admin():
7             return jsonify({"error": "Admin access required"}), 40
8     3     return f(*args, **kwargs)
9     return decorated_function
```

- The function from auth admin\_required is loaded into this file to be used on specific routes those being the create, edit, and delete routes to ensure that only admin can use these functions and not normal users.

Fig (4.47) CREATE Movie

```

1 @movie_bp.route('/create', methods=['POST'], endpoint='create_movie')
2 @admin_required
3 def create_movie():
4     logging.debug(f'Create movie request: {request.json}')
5     if not request.is_json:
6         return jsonify({'error': 'Content-Type must be application/json'}), 400
7
8     data = request.json
9     if not data or 'id' not in data or 'movie_title' not in data or 'movie_genres' not in data or 'actor_id' not in data or 'image' not in data:
10         return jsonify({'error': 'Missing id, movie_title, movie_genres, actor_id, or image'}), 400
11
12     if Movie.query.get(data['id']):
13         return jsonify({'error': f'Movie with ID {data["id"]} already exists'}), 400
14
15     actor = Actor.query.get(data['actor_id'])
16     if not actor:
17         return jsonify({'error': f'Actor with ID {data["actor_id"]} not found'}), 404
18
19     image_filename = data['image']
20     if image_filename not in AVAILABLE_IMAGES:
21         return jsonify({'error': f'Image must be one of {', '.join(AVAILABLE_IMAGES)}'}), 400
22     image_path = os.path.join(IMAGE_FOLDER, image_filename)
23     if not os.path.isfile(image_path):
24         return jsonify({'error': f'Image {image_filename} not found in {IMAGE_FOLDER}'}), 404
25     image_url = f'movies/{image_filename}'

```

```

1 try:
2     new_movie = Movie(
3         id=data['id'],
4         movie_title=data['movie_title'],
5         movie_genres=data['movie_genres'],
6         description=data.get('description'),
7         image_url=image_url
8     )
9     new_movie.actors.append(actor)
10    db.session.add(new_movie)
11    db.session.commit()
12    response = {
13        'message': 'Movie created successfully',
14        'id': new_movie.id,
15        'movie_title': new_movie.movie_title,
16        'movie_genres': new_movie.movie_genres,
17        'description': new_movie.description,
18        'image_url': f"/static/{new_movie.image_url}",
19        'created_at': new_movie.created_at.isoformat(),
20        'actors': [{ 'id': actor.id, 'name': actor.name }]
21    }
22    logging.debug(f"Create movie response: {response}")
23    return jsonify

```

- The create movie route allows the admin users to use this function, the request must contain JSON data with the details for the movies such as the id, movie title and description. If the movie provides id matches another movies id the operation will fail and return an error message, when successful the newly created movie will be added to the database and will send a success message.

Fig (4.48) GET Movie

```
1 @movie_bp.route('/', methods=['GET'], endpoint='get_movies')
2 @jwt_required()
3 def get_movies():
4     logging.debug("Fetching all movies")
5     movies = Movie.query.all()
6     response = []
7     for movie in movies:
8         "id": movie.id,
9         "movie_title": movie.movie_title if movie.movie_title else "Unknown Title",
10        "movie_genres": movie.movie_genres if movie.movie_genres else "Unknown",
11        "description": movie.description,
12        "image_url": f"/static/{movie.image_url}" if movie.image_url else "/static/movies/bloodborne.jp",
13        "created_at": movie.created_at.isoformat(),
14        "ratingsCount": Rating.query.filter_by(movie_id=movie.id).count(),
15        "reviewsCount": Review.query.filter_by(movie_id=movie.id).count(),
16        "actors": [{ 'id': actor.id, 'name': actor.name } for actor in movie.actors.all()]
17    } for movie in movies]
18    logging.debug(f"Get movies response: {response[:2]}")
19    return jsonify(response), 200
```

- The GET route gets all the movies from the database, the response includes the movies attributes and the number of ratings the specific movie has received. It allows the users to see all the movies in the database.

Fig (4.49) GET movie by id

```
1 @movie_bp.route('/<id>', methods=['GET'], endpoint='single_movie')
2 @jwt_required()
3 def single_movie(id):
4     logging.debug(f"Fetching movie ID: {id}")
5     movie = Movie.query.get_or_404(id)
6     actors = [{ 'id': actor.id, 'name': actor.name } for actor in movie.actors.all()]
7     response = {
8         "id": movie.id,
9         "movie_title": movie.movie_title if movie.movie_title else "Unknown Title",
10        "movie_genres": movie.movie_genres if movie.movie_genres else "Unknown",
11        "description": movie.description,
12        "image_url": f"/static/{movie.image_url}" if movie.image_url else "/static/movies/bloodborne.jp",
13        "created_at": movie.created_at.isoformat(),
14        "actors": actors
15    }
16    logging.debug(f"Single movie response: {response}")
17    return jsonify(response), 200
```

- The GET by id for the movies allows the users to view the data on one specific movie by its id, it returns the details and if it does not exist an error message will occur.

Fig (4.50) PUT Movie

```

1 @movie_bp.route('/update/<id>', methods=['PUT'], endpoint='update_movie')
2 @admin_required
3 def update_movie(id):
4     logging.debug(f'Update movie ID: {id}, data: {request.json}')
5     if not request.is_json:
6         return jsonify({'error': 'Content-Type must be application/json'}), 400
7
8     data = request.json
9     if not data or 'movie_title' not in data or 'movie_genres' not in data:
10        return jsonify({'error': 'Missing movie_title or movie_genres'}), 400
11
12    movie = Movie.query.get_or_404(id)
13    try:
14        movie.movie_title = data['movie_title']
15        movie.movie_genres = data['movie_genres']
16        movie.description = data.get('description', movie.description)
17        if 'image' in data:
18            image_filename = data['image']
19            if image_filename not in AVAILABLE_IMAGES:
20                return jsonify({'error': f'Image must be one of {', '.join(AVAILABLE_IMAGES)}'}), 400
21            image_path = os.path.join(IMAGE_FOLDER, image_filename)
22            if not os.path.isfile(image_path):
23                return jsonify({'error': f'Image {image_filename} not found in {IMAGE_FOLDER}'}), 404
24            movie.image_url = f'/movies/{image_filename}'
25        if 'actor_id' in data:
26            actor = Actor.query.get(data['actor_id'])
27            if not actor:
28                return jsonify({'error': f'Actor with ID {data["actor_id"]} not found'}), 404
29            movie.actors.append(actor)
30    db.session.commit()
31    response = {
32        'message': 'Movie updated successfully',
33        'id': movie.id,
34        'movie_title': movie.movie_title,
35        'movie_genres': movie.movie_genres,
36        'description': movie.description,
37        'image_url': f'/static/{movie.image_url}' if movie.image_url else '/static/movies/bloodborne1.jpg',
38        'created_at': movie.created_at.isoformat(),
39        'actors': [{ 'id': actor.id, 'name': actor.name } for actor in movie.actors.all()]
40    }
41    logging.debug(f'Update movie response: {response}')
42    return jsonify(response), 200
43 except Exception as e:
44    db.session.rollback()
45    logging.error(f'Update movie error: {str(e)}')
46    return jsonify({'error': f'Failed to update movie: {str(e)}'}), 500

```

- The movie update function allows the admin to edit the movie attribute details in the fields, only the movie title and movie genre can be changed as the id of the movie stays fixed. If the operation is successful, the system will respond with a 200 ok and if it fails an error message will occur.

Fig (4.51) DELETE movie

```

1 @movie_bp.route('/delete/<id>', methods=['DELETE'], endpoint='delete_movie')
2 @admin_required
3 def delete_movie(id):
4     logging.debug(f'Delete movie ID: {id}')
5     movie = Movie.query.get_or_404(id)
6     try:
7         Rating.query.filter_by(movie_id=id).delete()
8         Review.query.filter_by(movie_id=id).delete()
9         watchlists = Watchlist.query.all()
10        for watchlist in watchlists:
11            if id in watchlist.movie_ids:
12                watchlist.movie_ids = [mid for mid in watchlist.movie_ids if mid != id]
13        db.session.delete(movie)
14    db.session.commit()
15    logging.debug(f'Movie ID {id} deleted')
16    return jsonify({'message': 'Movie and associated ratings, reviews, and watchlist entries deleted successfully'}), 200
17 except Exception as e:
18    db.session.rollback()
19    logging.error(f'Delete movie error: {str(e)}')
20    return jsonify({'error': f'Failed to delete movie: {str(e)}'}), 500

```

- The delete method for the movies removes the movie entry from the database entirely. The movie id is provided in the URL, and if that movie exists it will be deleted alongside the ratings the movie had, and the entries will be deleted from all watchlists if it exists in it. If the operation was successful a 200 message will appear, if it does not work an error message appears.

Fig (4.52) Movie Stats

```

1 @movie_bp.route('/<id>/stats', methods=['GET'])
2 def movie_stats(id):
3     logging.debug(f"Fetch stats for movie ID: {id}")
4     movie = Movie.query.get_or_404(id)
5     ratings_count = Rating.query.filter_by(movie_id=movie.id).count()
6     reviews_count = Review.query.filter_by(movie_id=movie.id).count()
7     response = {
8         "ratings_count": ratings_count,
9         "reviews_count": reviews_count,
10        "actors_count": movie.actors.count()
11    }
12    logging.debug(f"Movie stats response: {response}")
13    return jsonify(response), 200

```

- The movie stats route provides the movies with the number of ratings it has from users.
- The other routes those being the ratings and watchlists having similar structures to the movie and auth routes.

#### 5.6.5 item 5

- Seeders

Fig (4.53) Loading the data from a TensorFlow library

```

1 with app.app_context():
2     dataset = list(tfds.load("movielens/100k-movies", split="train"))
3     print("Seeding movies from MovieLens 100k (all movies with 4 actors each)...")
4
5     batch_size = 100
6     counter = 0

```

- The tensorflow movielens 100k movies to seed the movie data with the same data the model uses. Faker is used to generate certain attributes such as the movie's description.

Fig (4.54) Genre map

```

1 genre_map = {
2     0: "Unknown", 1: "Action", 2: "Adventure", 3: "Animation", 4: "Children",
3     5: "Comedy", 6: "Crime", 7: "Documentary", 8: "Drama", 9: "Fantasy",
4     10: "Film-Noir", 11: "Horror", 12: "Musical", 13: "Mystery", 14: "Romance",
5     15: "Sci-Fi", 16: "Thriller", 17: "War", 18: "Western"
6 }
7

```

- We then map the genres by creating a dictionary of genres man mapping the ids to readable genre names such as action.

Fig (4.55) App context and movie extraction

```

1  try:
2      db.session.query(Movie).delete()
3      db.session.commit()
4      print("Cleared existing movies.")
5  except Exception as e:
6      print(f"Error deleting existing movies:
7  {e}")
8      db.session.rollback()
9      return

```

```

1  for movie in dataset:
2      movie_id = movie["movie_id"].numpy().decode('utf-8')
3      movie_title = movie["movie_title"].numpy().decode('utf-8')
4      genre_ids = movie["movie_genres"].numpy().tolist()
5      valid_genre_ids = [gid for gid in genre_ids if gid in genre_map]
6      movie_genres = ", ".join(genre_map[gid] for gid in valid_genre_ids) or "Unknown"
7      n"
8      description = fake.paragraph(nb_sentences=2)
9
10     existing_movie = Movie.query.filter_by(id=movie_id).first()
11     if not existing_movie:
12         selected_actors = random.sample(all_actors, 4)
13         selected_image = available_images[0]
14         image_url = f"movies/{selected_image}"
15
16         new_movie = Movie(
17             id=movie_id,
18             movie_title=movie_title,
19             movie_genres=movie_genres,
20             description=description,
21             image_url=image_url
22         )
23         new_movie.actors.extend(selected_actors)
24         db.session.add(new_movie)

```

- The seed\_movies function establishes the app context and loads the dataset. It then deletes any existing movies to prevent duplication in the database, it then processes the movies by extracting the movies id, movie\_title, and the genre ids converting them into a string. It then generates fake data for the descriptions of the movies and checked they do not already exist before adding the movies.

Fig (4.56) Batching data

```

1  counter += 1
2  if counter % batch_size == 0:
3      try:
4          db.session.commit()
5          print(f"Committed {counter} movies...")
6      except Exception as e:
7          print(f"Error committing batch at {counter} movies: {e}")
8          db.session.rollback()
9          return
10
11  try:
12      db.session.commit()
13      total_movies = Movie.query.count()
14      total_actors = Actor.query.count()
15      print(f"Seeded {total_movies} movies with actors from a pool of {total_actors} actors successfully.")
16      sample_movie = Movie.query.first()
17      if sample_movie:
18          print(f"Sample Movie: ID={sample_movie.id}, Title={sample_movie.movie_title}, Genres={sample_movie.movie_genres}, "
19                f"Description={sample_movie.description}, Image={sample_movie.image_url}, "
20                f"Actors={[{actor.name for actor in sample_movie.actors.all()}]}")
21      except Exception as e:
22          print(f"Final movie commit failed: {e}")
23          db.session.rollback()

```

- The seeder then commits the movies in batches of one hundred to improve the performance of the seeding process.

Fig (4.57) Seeding

```

1
2 if __name__ == "__main__":
3     print("Starting Movie Seeding...")
4     seed_movies()

```

- It then saves the movies and prints out the total movies that have been seeded.

Fig (4.58) User Seeder

```

1
2 with app.app_context():
3     print("Seeding users...")
4     admin_email = "admin@moviemuse.com"
5     admin_password = "adminpassword"
6
7     existing_admin = User.query.filter_by(email=admin_email).first()
8     if not existing_admin:
9         hashed_admin_password = bcrypt.generate_password_hash(admin_password).decode('utf-8')
10    8') admin_user = User(
11        username="admin",
12        email=admin_email,
13        password=hashed_admin_password,
14        user_gender=0,
15        user_occupation_label=0,
16        raw_user_age=30,
17        user_rating=0.0,
18        role="admin"
19    )
20    db.session.add(admin_user)
21    db.session.commit()
22    print("Admin user created.")
23 else:
24    admin_user = existing_admin

```

- The seed user function seeds and admin user if not created already, this user has a fixed email and password. This user will have privileges the other user does not. It then generates normal users of a maximum of 10 are seeded with unique emails and username, if the user already exists it skips that user in the database. The passwords are hashed with Bcrypt and the users are also given their attributes such as their gender, age, and occupation. The users are then added to the database.



Fig (4.59) Ratings seeding

```

1 all_movies = Movie.query.all()
2 if not all_movies:
3     print("No movies found in the database. Please seed movies first.")
4     return
5 movie_ids = [movie.id for movie in all_movies]
6 total_movies = len(movie_ids)
7 print(f"Found {total_movies} movies in the database.")
8
9 max_users = 10
10 user_map = {}
11 new_users = []
12
13 if not existing_admin:
14     user_map["admin"] = admin_user.id
15
16 print(f"Seeding {max_users} users with 1 review and 1 rating per movie, plus private and public watchlists...")
17
18 ratings_dataset = list(tfds.load("movieLens/100k-ratings", split="train"))
19 counter = 0

```

```

1 for rating in ratings_dataset:
2     user_id = rating["user_id"].numpy().decode('utf-8')
3     user_gender = int(rating["user_gender"].numpy())
4     user_age = int(rating["raw_user_age"].numpy())
5     user_occupation_label = int(rating["user_occupation_label"].numpy())
6
7     if user_id not in user_map and len(user_map) < max_users:
8         email = f"user{user_id}@moviemuse.com"
9         max_attempts = 5
10        for attempt in range(max_attempts):
11            username = fake.user_name()
12            if not User.query.filter_by(username=username).first():
13                break
14            if attempt == max_attempts - 1:
15                username = f"{fake.user_name()}{user_id}"

```

```

1 existing_user = User.query.filter_by(email=email).first()
2 if not existing_user:
3     hashed_password = bcrypt.generate_password_hash("password123").decode('utf-8')
4     new_user = User(
5         username=username,
6         email=email,
7         password=hashed_password,
8         user_gender=user_gender,
9         user_occupation_label=user_occupation_label,
10        raw_user_age=user_age,
11        user_rating=0.0,
12        role="user"
13    )
14    db.session.add(new_user)
15    new_users.append(new_user)
16    user_map[user_id] = None

```

```

1 counter += 1
2 if counter >= max_users:
3     break
4
5 if new_users:
6     db.session.commit()
7     for new_user in new_users:
8         user_map[new_user.email.split('@')[0][4:]] = new_user.id
9     print(f"Seeded {len(new_users)} new users.")
10
11 user_ids = [user_id for user_id_str, user_id in user_map.items() if user_id]
12 counter = 0
13 batch_size = 1000

```



```

1 for movie_id in movie_ids:
2     movie = Movie.query.get(movie_id)
3     if movie:
4         reviewer_id = random.choice(user_ids)
5         if not Review.query.filter_by(user_id=reviewer_id, movie_id=movie_id).first():
6             review_content = "{fake_sentence()} {fake_word()} on {fake_word()} and the {fake_word()} really {fake_word()} the experience"
7             new_review = Review(
8                 user_id=reviewer_id,
9                 movie_id=movie_id,
10                content=review_content
11            )
12            db.session.add(new_review)
13            counter += 1
14
15 rater_id = random.choice(user_ids)
16 if not Rating.query.filter_by(user_id=rater_id, movie_id=movie_id).first():
17     rating_value = float(fake_random_int(min=1, max=5))
18     new_rating = Rating(
19         user_id=rater_id,
20         movie_id=movie_id,
21         rating=rating_value
22     )
23     db.session.add(new_rating)
24     counter += 1
25
26 if counter % batch_size == 0:
27     db.session.commit()
28     print(f"Committed {counter} ratings and reviews...")

```

- Each movie is then assigned a random rating from a user who has not rated it yet, these ratings are set to be between one and five and are stored in the ratings tables in the database. It is then set in batch of one thousand.

Fig (4.60) Watchlist seeding

```

1 for user_id in user_ids:
2     if not Watchlist.query.filter_by(user_id=user_id, title="My Private List").first():
3         private_movie_ids = random.sample(movie_ids, min(3, len(movie_ids)))
4         new_private_watchlist = Watchlist(
5             user_id=user_id,
6             title="My Private List",
7             movie_ids=private_movie_ids,
8             is_public=False
9         )
10        db.session.add(new_private_watchlist)
11        counter += 1
12
13 if not Watchlist.query.filter_by(user_id=user_id, title="My Public Favorites").first():
14     public_movie_ids = random.sample(movie_ids, min(4, len(movie_ids)))
15     new_public_watchlist = Watchlist(
16         user_id=user_id,
17         title="My Public Favorites",
18         movie_ids=public_movie_ids,
19         is_public=True
20     )
21     db.session.add(new_public_watchlist)
22     counter += 1

```

- The seeding of the watchlists is then underway seeding each user with a watchlist with a random selection of three movies to populate the watchlist.

Fig (4.61) Seeding users

```

1 db.session.commit()
2
3 total_ratings = Rating.query.count()
4 total_reviews = Review.query.count()
5 total_watchlists = Watchlist.query.count()
6 print(f"Seeded {len(user_map)} users with {total_ratings} ratings, {total_reviews} reviews, and {total_watchlists} watchlists successfully.")
7
8 if __name__ == "__main__":
9     print("Starting User Seeding...")
10    seed_user()

```

- It is them all committed to the database and respond with the successful seeding of users, watchlists, and rating.

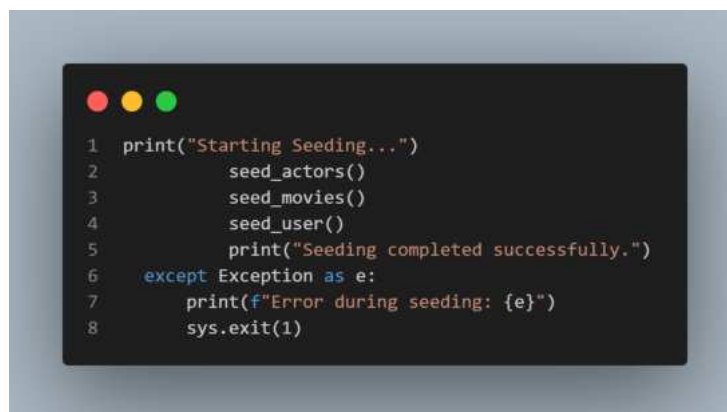
Fig (4.62) Dropping tables and Creating tables

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a Python code snippet for resetting a database.

```
1 try:
2     with app.app_context():
3         print("Dropping all tables...")
4         db.drop_all()
5         print("All tables dropped.")
6
7         print("Creating all tables...")
8         db.create_all()
9         print("Database tables created or verified.")
```

- The seed\_all function resets the database by dropping all the tables to give the database a fresh start. It then remakes all the tables.

Fig (4.63) Running the movie and user seeder

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a Python code snippet for seeding the database.

```
1 print("Starting Seeding...")
2     seed_actors()
3     seed_movies()
4     seed_user()
5     print("Seeding completed successfully.")
6 except Exception as e:
7     print(f"Error during seeding: {e}")
8     sys.exit(1)
```

- Seeding the data runs the movie\_seeder and the user\_seeder, each function populates the table corresponding to the data in the database.
- For error handling if any process of the seeding fails an error message will appear informing the user.

Fig (4.64) Seeding database

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a Python code snippet for the main entry point.

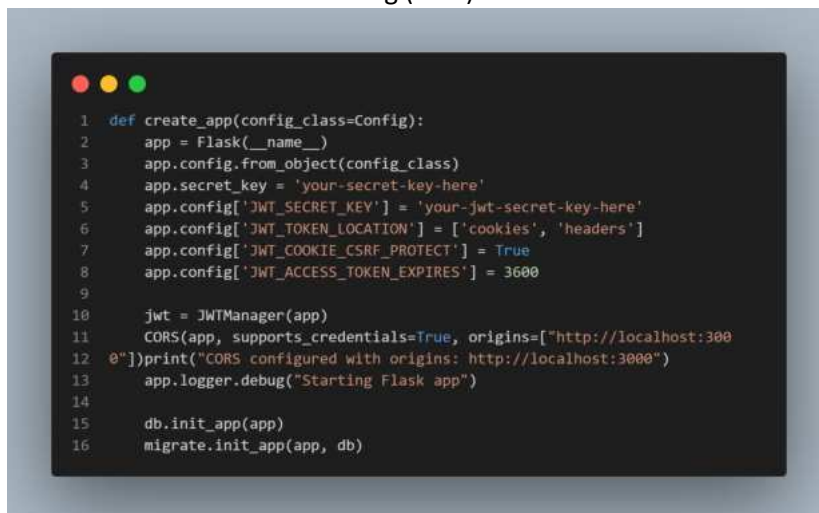
```
1 if __name__ == "__main__":
2     print("Executing seed_all...")
3     seed_all()
```

- It then runs the scripts performing seed\_all to call the other seeding functions to being the seeding process.

#### 4.6.3 Item 6

- App.py
  - Setting up this file by importing Flask, being the main framework. CORS to allow the front end to communicate with the backend without error with the requests.
  - The JWT are imported to manage the authentication and authorization of users.
  - The extension file is important contains database operations along with importing the config file for configurations.
  - The blueprints from the routes are imported to handle the various routes.
  - The apps configuration involves setting the app secret and JWT secret for setting the apps general key and the setting up the key for the JWT for the tokens.

Fig (4.65)



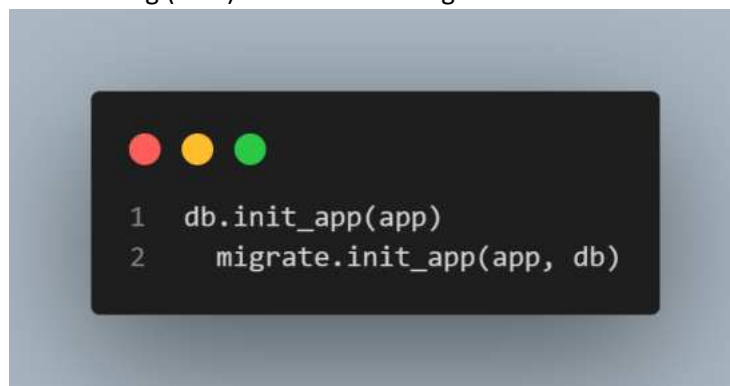
```

1 def create_app(config_class=Config):
2     app = Flask(__name__)
3     app.config.from_object(config_class)
4     app.secret_key = 'your-secret-key-here'
5     app.config['JWT_SECRET_KEY'] = 'your-jwt-secret-key-here'
6     app.config['JWT_TOKEN_LOCATION'] = ['cookies', 'headers']
7     app.config['JWT_COOKIE_CSRF_PROTECT'] = True
8     app.config['JWT_ACCESS_TOKEN_EXPIRES'] = 3600
9
10    jwt = JWTManager(app)
11    CORS(app, supports_credentials=True, origins=["http://localhost:3000"])
12    print("CORS configured with origins: http://localhost:3000")
13    app.logger.debug("Starting Flask app")
14
15    db.init_app(app)
16    migrate.init_app(app, db)

```

- The tokens are sent in cookies and headers, and the token expires in one hour.
- The CORS configuration allows the requests from the front-end being localhost:3000 and enables cookies for the authentication of the users.

Fig (4.66) Database and migration



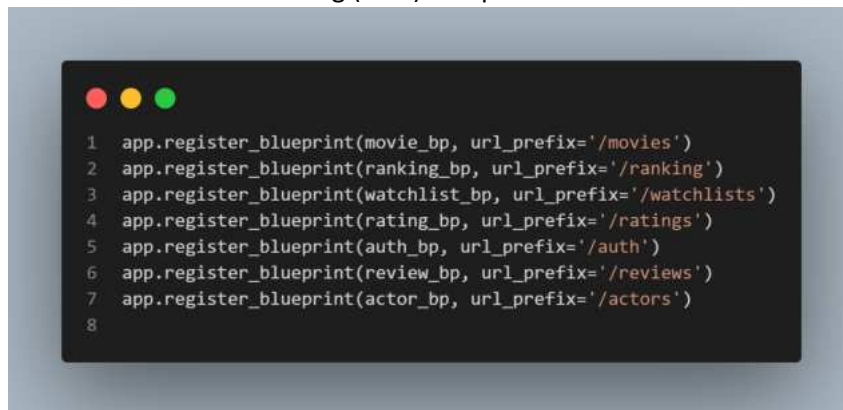
```

1 db.init_app(app)
2 migrate.init_app(app, db)

```

- The app then connects to the database and enables the migrations.

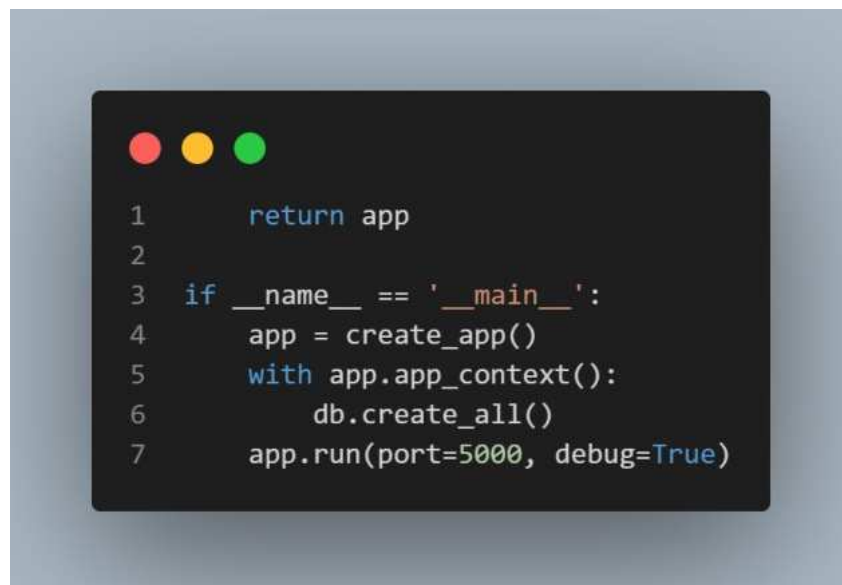
Fig (4.67) Blueprints

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is as follows:

```
1 app.register_blueprint(movie_bp, url_prefix='/movies')
2 app.register_blueprint(ranking_bp, url_prefix='/ranking')
3 app.register_blueprint(watchlist_bp, url_prefix='/watchlists')
4 app.register_blueprint(rating_bp, url_prefix='/ratings')
5 app.register_blueprint(auth_bp, url_prefix='/auth')
6 app.register_blueprint(review_bp, url_prefix='/reviews')
7 app.register_blueprint(actor_bp, url_prefix='/actors')
8
```

- The routes are then registered in this file registering the movie, watchlist, auth, and rating routes.

Fig (4.68) Server

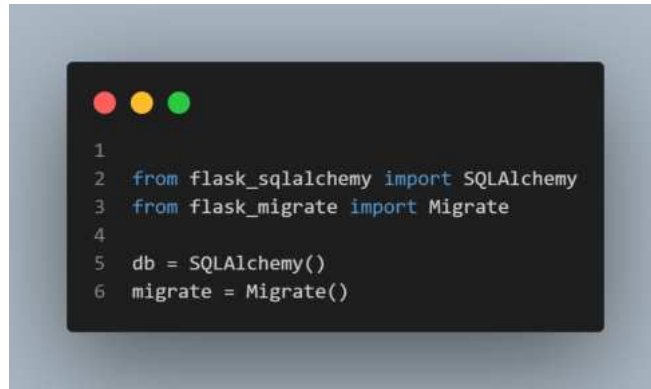
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is as follows:

```
1     return app
2
3 if __name__ == '__main__':
4     app = create_app()
5     with app.app_context():
6         db.create_all()
7     app.run(port=5000, debug=True)
```

- The server can then be served on port 5000.

#### 4.6.4 Item 7

- Extensions.py  
Fig (4.69) Extensions



```

1
2 from flask_sqlalchemy import SQLAlchemy
3 from flask_migrate import Migrate
4
5 db = SQLAlchemy()
6 migrate = Migrate()

```

- In this file extensions.py, Flask-SQLAlchemy is initialized which handles the database operations and the migrations, this handles the schema for the migrations. SQLAlchemy allows for the defining of models, querying the database, and performing the CRUD functionalities on the application.
- Separating the files from app.py to extensions prevents importing issues and makes managing the database organised.

#### 4.6.5 Item 8

- Config.py  
Fig (4.70) Config



```

1 class Config:
2     SQLALCHEMY_DATABASE_URI = 'sqlite:///databasemovie.db'
3     SQLALCHEMY_TRACK_MODIFICATIONS = False

```

- The config file defines the configuration settings for the application. It is setting SQLAlchemy\_DATABASE\_URI to use the SQLite database which specifies the storing location of the database. SQLAlchemy\_TRACK\_MODIFICATIONS is set as false to disable the tracking modification of objects which is done to improve performance.

## 4.7 Sprint 4

In sprint 4 some redesigns were made in the projects wireframe and style guide to update the look of the application. This including adding the pages for the actors, public watchlists, and implementing the review component in the movie single page so users could see how the reviews would look in the page. The style guide shows the colour scheme change to darker colours. This gave the cards more contrast and made the components look better on the application.

The database Erd was also updated to match the new tables that were adding to give more functionality to the web app. The introduction of the actor and review tables were made; the actor table established a many-to-many relationship with the movie table. The review table has a many-to-one relationship with both the movie table and the user table.

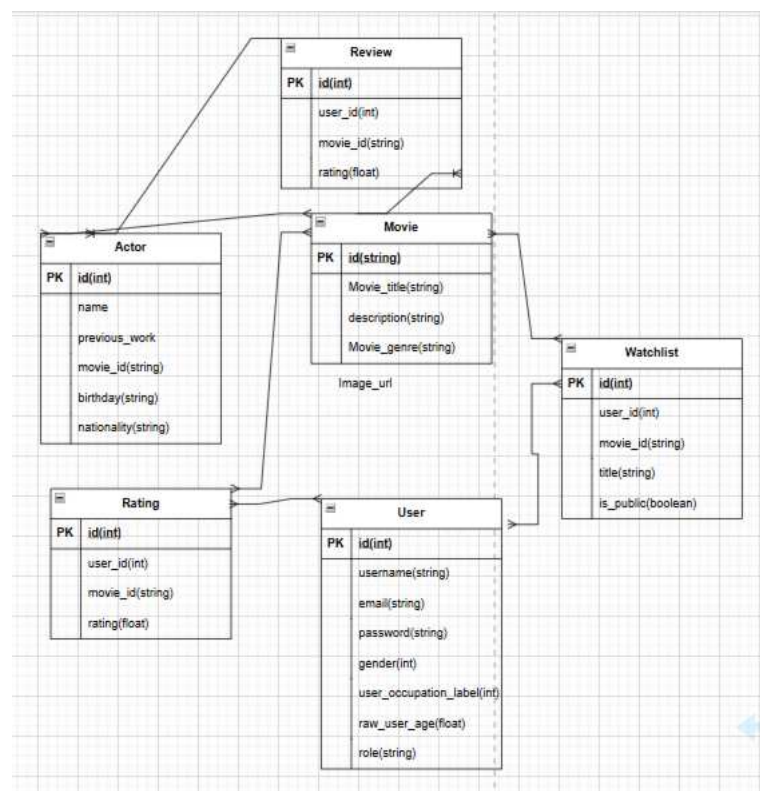
From the additions to the Erd the actor and review model were made. A linking table named actor movie was also made to define the many-to-many relationship with movies. Both models are used to store the users reviews and the actors that would be seen in the movie details in the single page.

The routes were then made for the actors and reviews so that CRUD functionality so that the data could be interacted with by the user and admin. The review routes also allowed for CRUD functionality. In the movie route the delete was altered to delete the actor from the movie if the movie was deleted itself, the same was done with the reviews except when the movie is deleted the reviews for that movie are also removed from the database unlike the actors which stay unless deleted using a different method only the admins can do. The last route that was altered was the watchlist route to add in the public watchlist functions.

Lastly the seeders were updated, the movie seeder was altered to seed the movies with random actors. The user seeder seeds the latest reviews to each of the movies while also make public watchlists for each of the users that were seeded into the database initially. Before the movie and user seeders have been seeded, a new seeder was made to seed the database first so then the movies could seed and use the actors to populate the actor ids in the movies.

#### 4.7.1 Item 1

- Database Erd design update
- Fig (4.71) Updated Erd

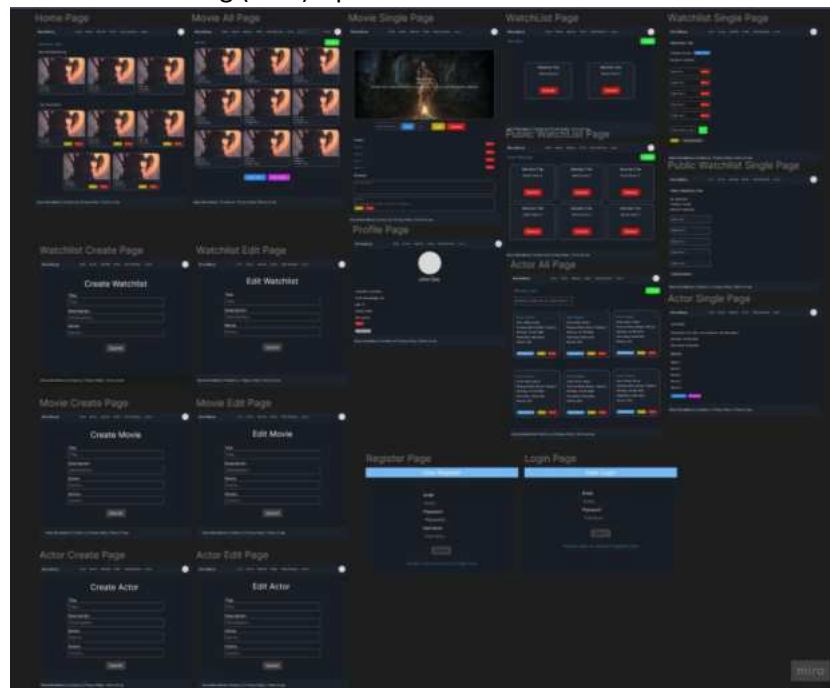


- The database Erd was updated to include the new tables, actors and reviews which displays a visual representation of the new relationships and tables in the application. The actor has a many-to-many relationship with the movies table allowing for multiple actors to be associated with a single movie, and multiple movies to be associated with a single actor. The reviews have a one-to-many relationship with the movie and user table, which allows for the user to make multiple reviews on multiple different movies while keeping a link to the user.

#### 4.7.2 Item 2

- Wireframe design update

Fig (4.72) Updated Wireframe



- The wireframe was redesigned to include new pages for the actors and public watchlists, the actor pages including all, edit, and create are all lock only being accessible to the admin user. The actor single page can be views by all users viewing the actor's details. The review component was also implemented to show how the review component would be seen in the application.

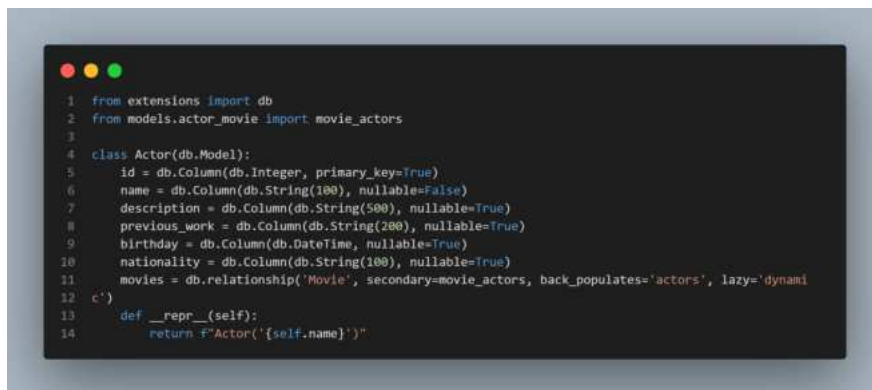
#### 4.7.3 Item 3

- Style Guide Update  
Fig (4.73) Updated style guide

- The style guide was updated to include a darker colour scheme to enhance the design of the application. The darker colour scheme adding contrast to the cards and other components making the application visually pleasing.

#### 4.7.4 Item 4

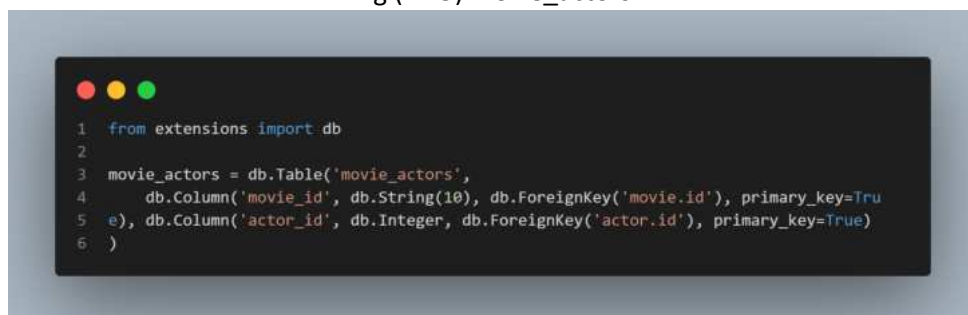
- Addition to models  
Fig (4.74) Actor, Review, Actor\_movies models



```
1 from extensions import db
2 from models.actor_movie import movie_actors
3
4 class Actor(db.Model):
5     id = db.Column(db.Integer, primary_key=True)
6     name = db.Column(db.String(100), nullable=False)
7     description = db.Column(db.String(500), nullable=True)
8     previous_work = db.Column(db.String(200), nullable=True)
9     birthday = db.Column(db.DateTime, nullable=True)
10    nationality = db.Column(db.String(100), nullable=True)
11    movies = db.relationship('Movie', secondary=movie_actors, back_populates='actors', lazy='dynamic')
12
13    def __repr__(self):
14        return f'Actor({self.name})'
```

- the linking table for the actors is the movie\_actors which links actors and movies for the definition of the many-to-many relationship.
- The actor model consists of the id which is used as a unique identifier for each actor and is set as an integer.
- The name field is used to store the name of the actors and is set as a string.
- The actor's description stores more details about the user and is set as a string.
- The actors previous work is used to give depth to the actors giving them previous work they have done. This field is set as a string.
- The actor's birthday stores their date of birth and is set as a string.
- The actor's nationality stores the country the actor is from and is set as a string.
- The relationship with movie is established as a many-to-many and accesses the linking table, this allows the movies to access the actor data.
- The review model has a similar structure with different attributes such as content and the movie\_id.

Fig (4.75) movie\_actors



```
1 from extensions import db
2
3 movie_actors = db.Table('movie_actors',
4     db.Column('movie_id', db.String(10), db.ForeignKey('movie.id'), primary_key=True),
5     db.Column('actor_id', db.Integer, db.ForeignKey('actor.id'), primary_key=True)
6 )
```

- The extensions are imported to use the database object.



- This table create a junction; this table does not have its own model class and is used to establish the many-to-many relationship between actors and movies.
- In the columns the movie\_id is stored and makes sure the movie and actor ids are unique
- The same is done with the actors, storing the id, and making sure the id are unique.

#### 4.7.5 Item 5

- Addition to routes

Fig (4.76) GET Actor

```

1 @actor_bp.route('', methods=['GET'])
2 @jwt_required()
3 def get_all_actors():
4     actors = Actor.query.all()
5     return jsonify([
6         {
7             'id': actor.id,
8             'name': actor.name,
9             'description': actor.description,
10            'previous_work': actor.previous_work,
11            'birthday': actor.birthday.isoformat() if actor.birthday else None,
12            'nationality': actor.nationality,
13            'movie_count': actor.movies.count()
14        } for actor in actors]), 200

```

- The GET for the actors retrieves all the actors in the database, this route requires jwt authentication. It returns a response of the actor's data including the movie count of the actors. If successful a 200 message will be the response, if the operation fails an error message will appear and the actors will not be loaded.

Fig (4.77) CREATE Actor

```

1 @actor_bp.route('', methods=['POST'])
2 @jwt_required()
3 @admin_required
4 def create_actor():
5     if not request.is_json:
6         return jsonify({'error': 'Content-Type must be application/json'}), 400
7
8     data = request.json
9     if not data or 'name' not in data:
10        return jsonify({'error': 'Missing name'}), 400
11
12    try:
13        if len(data['name']) > 100:
14            return jsonify({'error': 'Name must be 100 characters or less'}), 400
15        description = data.get('description')
16        if description and len(description) > 500:
17            return jsonify({'error': 'Description must be 500 characters or less'}), 400
18        previous_work = data.get('previous_work')
19        if previous_work and len(previous_work) > 200:
20            return jsonify({'error': 'Previous work must be 200 characters or less'}), 400
21        nationality = data.get('nationality')
22        if nationality and len(nationality) > 100:
23            return jsonify({'error': 'Nationality must be 100 characters or less'}), 400
24
25        birthday = data.get('birthday')
26        if birthday and isinstance(birthday, str):
27            birthday = datetime.strptime(birthday, '%Y-%m-%d')

```

```

1 new_actor = Actor(
2     name=data['name'],
3     description=description,
4     previous_work=previous_work,
5     birthday=birthday,
6     nationality=nationality
7 )
8 db.session.add(new_actor)
9 db.session.commit()
10 return jsonify({
11     'message': 'Actor created successfully',
12     'id': new_actor.id,
13     'name': new_actor.name,
14     'description': new_actor.description,
15     'previous_work': new_actor.previous_work,
16     'birthday': new_actor.birthday.isoformat() if new_actor.birthday else None,
17     'nationality': new_actor.nationality
18 }), 201
19 except ValueError:
20     return jsonify({'error': 'Invalid birthday format. Use YYYY-MM-DD'}), 400
21 except Exception as e:
22     db.session.rollback()
23     return jsonify({'error': f'Failed to create actor: {str(e)}'}), 500

```

- The create for the actors lets admins create new actors for the website. The inputs are validated ensure a name is used, the other fields are optional and have a required character limit. When successful the system responds with a 200 message and the new actor is created and inputted into the database. If this fails an error message will appear, if it fails due to the user not being an admin the appropriate message is sent to the user explaining the need to be an admin for this function.

Fig (4.78) GET ID Actor

```

1 @actor_bp.route('/<actor_id>', methods=['GET'])
2 @jwt_required()
3 def get_actor(actor_id):
4     actor = Actor.query.get_or_404(actor_id)
5     movies = [{'id': movie.id, 'movie_title': movie.movie_title} for movie in actor.movies.all]
6     return jsonify({
7         'id': actor.id,
8         'name': actor.name,
9         'description': actor.description,
10        'previous_work': actor.previous_work,
11        'birthday': actor.birthday.isoformat() if actor.birthday else None,
12        'nationality': actor.nationality,
13        'movies': movies
14    }), 200

```

- The GET by id for actors allows users to see a specific actor, requiring JWT authentication. It returns the data about the actor along with the number of movies they appear in. If the operation is successful a 200 message will appear along with the actor and movie data. If this fails an error message will appear and the actor and movie data will not appear.

Fig (4.79) PUT Actor

```

1 @actor_bp.route('/<actor_id>', methods=['PUT'])
2 @jwt_required()
3 @admin_required
4 def update_actor(actor_id):
5     if not request.is_json:
6         return jsonify({'error': 'Content-Type must be application/json'}), 400
7
8     data = request.json
9     if not data or 'name' not in data:
10        return jsonify({'error': 'Missing name'}), 400
11
12    actor = Actor.query.get_or_404(actor_id)
13    try:
14        if len(data['name']) > 100:
15            return jsonify({'error': 'Name must be 100 characters or less'}), 400
16        description = data.get('description', actor.description)
17        if description and len(description) > 500:
18            return jsonify({'error': 'Description must be 500 characters or less'}), 400
19        previous_work = data.get('previous_work', actor.previous_work)
20        if previous_work and len(previous_work) > 200:
21            return jsonify({'error': 'Previous work must be 200 characters or less'}), 400
22        nationality = data.get('nationality', actor.nationality)
23        if nationality and len(nationality) > 100:
24            return jsonify({'error': 'Nationality must be 100 characters or less'}), 400
25
26        birthday = data.get('birthday', actor.birthday)
27        if birthday and isinstance(birthday, str):
28            birthday = datetime.strptime(birthday, '%Y-%m-%d')

```

```

1 actor.name = data['name']
2 actor.description = description
3 actor.previous_work = previous_work
4 actor.birthday = birthday
5 actor.nationality = nationality
6
7 db.session.commit()
8 return jsonify({
9     'message': 'Actor updated successfully',
10    'id': actor.id,
11    'name': actor.name,
12    'description': actor.description,
13    'previous_work': actor.previous_work,
14    'birthday': actor.birthday.isoformat() if actor.birthday else None,
15    'nationality': actor.nationality
16}), 200
17 except ValueError:
18    return jsonify({'error': 'Invalid birthday format. Use YYYY-MM-DD'}), 400
19 except Exception as e:
20    db.session.rollback()
21    return jsonify({'error': f'Failed to update actor: {str(e)}'}), 500

```

- The update actors allow admins to edit details about the actors, also long as the details provided match the correct inputs, when committed the actor's details should update and then be changed in the database. If this fails an error message will appear.

Fig (4.80) DELETE Actor

```
1 @actor_bp.route('/<actor_id>', methods=['DELETE'])
2 @jwt_required()
3 @admin_required
4 def delete_actor(actor_id):
5     actor = Actor.query.get_or_404(actor_id)
6     try:
7         db.session.delete(actor)
8         db.session.commit()
9         return jsonify({'message': f'Actor {actor.name} deleted successfully'}), 200
10    except Exception as e:
11        db.session.rollback()
12        return jsonify({'error': f'Failed to delete actor: {str(e)}'}), 500
```

- The delete actors function allows the admins to delete actors from the database. The remove actor from movie allows the admin to disassociate an actor from a movie. It verifies that the actor exists and if they do removes them from the database. If the operation fails an error message appears.
- The review route has a similar structure having a GET, PUT, POST, and DELETE method minus the GET by id as it was not required for that route.

Fig (4.81) All public watchlists

```
1 @watchlist_bp.route('/public', methods=['GET'])
2 def get_public_watchlists():
3     try:
4         public_watchlists = Watchlist.query.filter_by(is_public=True).all()
5         return jsonify([
6             {
7                 "id": item.id,
8                 "user_id": item.user_id,
9                 "title": item.title,
10                "movie_ids": item.movie_ids,
11                "is_public": item.is_public,
12                "username": item.user.username
13            } for item in public_watchlists]), 200
14    except Exception as e:
15        return jsonify({'error': f'Failed to fetch public watchlists: {str(e)}'}), 500
```

- The GET for the public watchlists gets all the watchlists that are made public. The route queries the watchlists table to get all the watchlists where the is\_public field is true, then for the public watchlists its response with the watchlists data that it corresponds with, then the watchlists id, the users id, the associated movies, and the status of the watchlist is displayed and the users can then view other users watchlists.

Fig ( ) Public watchlists added to watchlist route

```

1 @watchlist_bp.route('/public/<int:id>', methods=['GET'])
2 def get_public_watchlist(id):
3     try:
4         watchlist = Watchlist.query.filter_by(id=id, is_public=True).first()
5         if not watchlist:
6             return jsonify({'error': 'Public watchlist not found'}), 404
7
8         movie_details = []
9         for movie_id in watchlist.movie_ids:
10             movie = Movie.query.get(movie_id)
11             if movie:
12                 movie_details.append({
13                     "id": movie.id,
14                     "title": movie.movie_title,
15                     "genres": movie.movie_genres
16                 })
17
18         return jsonify({
19             "id": watchlist.id,
20             "user_id": watchlist.user_id,
21             "title": watchlist.title,
22             "movie_ids": watchlist.movie_ids,
23             "movies": movie_details,
24             "is_public": watchlist.is_public,
25             "username": watchlist.user.username
26         }), 200
27     except Exception as e:
28         return jsonify({'error': f'Failed to fetch public watchlist: {str(e)}'}), 500

```

- The GET id gets a specific public watchlist using its id and makes sure the watchlist exist and is set to public, if the watchlist does exist the data for the watchlist, user, and movies, will display. Additional data is sent showing the movie details. If this operation fails, no data will be displayed, and an error message will appear.

Fig (4.82) GET movie with actors

```

1 @movie_bp.route('/', methods=['GET'], endpoint='get_movies')
2 @jwt_required()
3 def get_movies():
4     logging.debug("Fetching all movies")
5     movies = Movie.query.all()
6     response = []
7     for movie in movies:
8         "id": movie.id,
9         "movie_title": movie.movie_title if movie.movie_title else "Unknown Title",
10        "movie_genres": movie.movie_genres if movie.movie_genres else "Unknown",
11        "description": movie.description,
12        "image_url": f"/static/{movie.image_url}" if movie.image_url else "/static/movies/bloodborne.jpg",
13        "created_at": movie.created_at.isoformat(),
14        "ratingCount": Rating.query.filter_by(movie_id=movie.id).count(),
15        "reviewCount": Review.query.filter_by(movie_id=movie.id).count(),
16        "actors": [{"id": actor.id, "name": actor.name} for actor in movie.actors.all()]
17     logging.debug(f"Get movies response: {response[:2]}")
18     return jsonify(response), 200

```

- The GET method gets all the movies with the corresponding actors and reviews counted. In the response for the GET request, the list of the actors associated with the movies is gotten and their id and name are returned. For the reviews in each movie, using the .count() gets all the reviews that are associated with the movies and displays how many reviews the movie has.

Fig (4.83) GET ID movie with actor

```

1 @movie_bp.route('/<id>', methods=['GET'], endpoint='single_movie')
2 @jwt_required()
3 def single_movie(id):
4     logging.debug(f"Fetching movie ID: {id}")
5     movie = Movie.query.get_or_404(id)
6     actors = [{'id': actor.id, 'name': actor.name} for actor in movie.actors.all()]
7     response = {
8         "id": movie.id,
9         "movie_title": movie.movie_title if movie.movie_title else "Unknown Title",
10        "movie_genres": movie.movie_genres if movie.movie_genres else "Unknown",
11        "description": movie.description,
12        "image_url": f"/static/{movie.image_url}" if movie.image_url else "/static/movies/bloodborne.jp
13        g",
14        "created_at": movie.created_at.isoformat(),
15        "actors": actors
16    }
17    logging.debug(f"Single movie response: {response}")
18    return jsonify(response), 200

```

- The GET by id for the movies, the actors are retrieved if they are associated with the specific movie.

Fig (4.84) CREATE movie with actor

```

1 new_movie.actors.append(actor)
2 db.session.add(new_movie)
3 db.session.commit()
4 response = {
5     'message': 'Movie created successfully',
6     'id': new_movie.id,
7     'movie_title': new_movie.movie_title,
8     'movie_genres': new_movie.movie_genres,
9     'description': new_movie.description,
10    'image_url': f"/static/{new_movie.image_url}",
11    'created_at': new_movie.created_at.isoformat(),
12    'actors': [{'id': actor.id, 'name': actor.name}]
13 }
14 logging.debug(f"Create movie response: {response}")
15 return jsonify(response), 201
16 except Exception as e:
17     db.session.rollback()
18     logging.error(f"Create movie error: {str(e)}")
19     return jsonify({'error': f'Failed to create movie: {str(e)}'}), 500
20

```

- For the movie create, the actor data is also sent with the movie data, the actors id is used and then in the database it checks to see if that actor exists, if the actor does exist it will be added to the newly created movie using the .append(actor) method. If this fails an error message will appear.

Fig (4.85) PUT movie with actor data

```

1 actor = Actor.query.get(data['actor_id'])
2 if not actor:
3     return jsonify({'error': f"Actor with ID {data['actor_id']} not found"}), 404
4 movie.actors.append(actor)
5 db.session.commit()
6 response = {
7     'message': 'Movie updated successfully',
8     'id': movie.id,
9     'movie_title': movie.movie_title,
10    'movie_genres': movie.movie_genres,
11    'description': movie.description,
12    'image_url': f"/static/{movie.image_url}" if movie.image_url else "/static/movies/bloodborne.jp
13    g",
14    'created_at': movie.created_at.isoformat(),
15    'actors': [{'id': actor.id, 'name': actor.name} for actor in movie.actors.all()]
16 }
17 logging.debug(f"Update movie response: {response}")
18 return jsonify(response), 200
19 except Exception as e:
20     db.session.rollback()
21     logging.error(f"Update movie error: {str(e)}")
22     return jsonify({'error': f'Failed to update movie: {str(e)}'}), 500

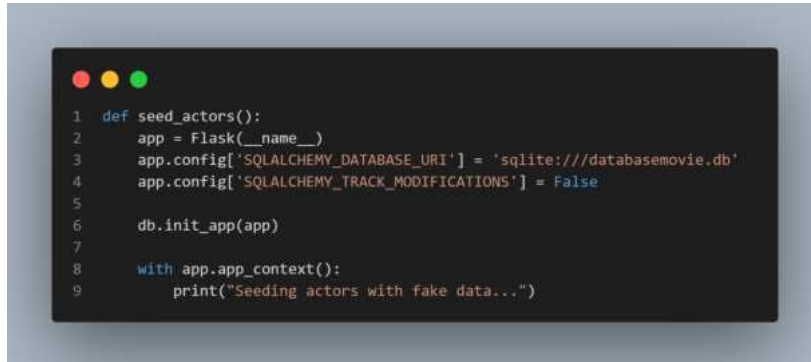
```

- The update movie route, the actor associated with the movie is visible in the response showing the id and name of the actors.

#### 4.7.6 Item 6

- Addition to Seeders

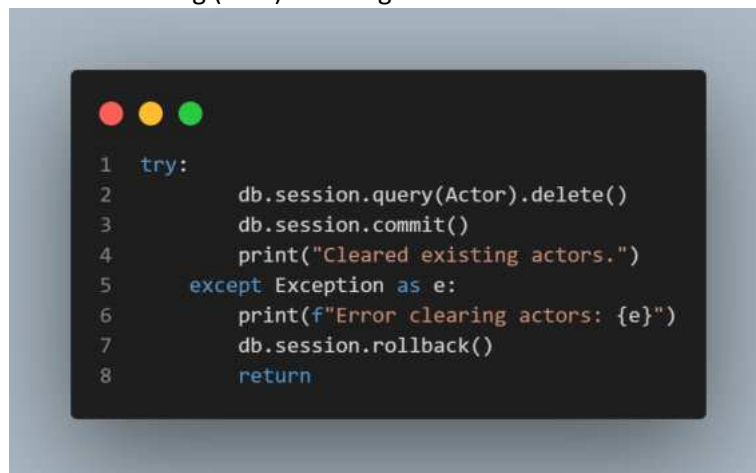
Fig (4.86) Seed Actors function

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and defines a function named `seed_actors()`. The function initializes a Flask application, configures the database URI to `'sqlite:///databasemovie.db'` and sets `'SQLALCHEMY_TRACK_MODIFICATIONS'` to `False`. It then calls `db.init_app(app)`. A `with` block is used to enter the application context, where a print statement displays `"Seeding actors with fake data..."`.

```
1 def seed_actors():
2     app = Flask(__name__)
3     app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///databasemovie.db'
4     app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
5
6     db.init_app(app)
7
8     with app.app_context():
9         print("Seeding actors with fake data...")
```

- The `seed_actors` is the function that generates fake data that is populated in the database. Using app context is set up while the seeding process is underway, it then gives a message displaying that the actors are being seeded.

Fig (4.87) Cleaning database

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and defines a function to clean the database. It uses a `try` block to attempt deleting all records from the `Actor` table using `db.session.query(Actor).delete()`, followed by `db.session.commit()` and a print statement `print("Cleared existing actors.")`. An `except` block catches any `Exception` as `e`, prints an error message `print(f"Error clearing actors: {e}")`, calls `db.session.rollback()`, and then `return` to exit the function early.

```
1 try:
2     db.session.query(Actor).delete()
3     db.session.commit()
4     print("Cleared existing actors.")
5 except Exception as e:
6     print(f"Error clearing actors: {e}")
7     db.session.rollback()
8     return
```

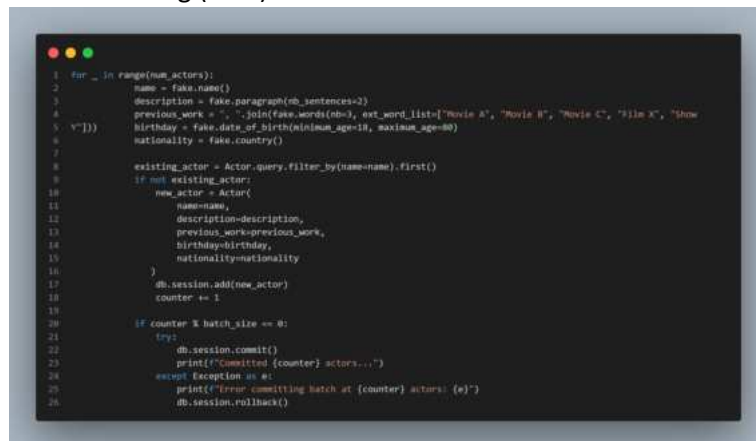
- The database is cleared using the `.delete()` method which deletes all the data from the actor table before the seeding process is done to prevent duplicates. After the deletions are made the `.commit()` method runs to make sure the changes made by the deletion are saved. If a problem occurs while deleting an error message will appear and the function returns early.

Fig (4.88) Seed Actors function



- The function sets the number of actors to be set to thirty actors; it defines the batch size of one hundred for committing the changes to the database in parts which help with performance.

Fig (4.89) Seed Actors function



- A loop is made to go through the thirty actors and in each iteration the fake data is made for each of the actors' attributes using the faker libraries, it fakes the data for the actor's name, description, previous\_work, birthday, and nationality. It checks if the actors name is already used in the database and if it does not exist the process continues and creates the actors. Adding the new actors to the session creates a new instance of Actor with the fake generated data and adds them to the session, if the counter reaches the one hundred batch size the changes are then committed to the database. And gives a message saying the number of actors that are being committed to the database.



Fig (4.90) Seeding data

```
1 try:
2     db.session.commit()
3     total_actors = Actor.query.count()
4     print(f"Seeded {total_actors} actors successfully.")
5     sample_actor = Actor.query.first()
6     if sample_actor:
7         print(f"Sample Actor: ID={sample_actor.id}, Name={sample_actor.name}, Description={sample_actor.description}. "
8               f"Previous Work={sample_actor.previous_work}, Birthday={sample_actor.birthday}, Nationality={sample_actor.nationality}")
9     except Exception as e:
10        print(f"Final actor commit failed: {e}")
11        db.session.rollback()
```

- When the loop is complete any uncommitted actor data is sent to the database. It then shows the total number of actors that were seeded to the database.

Fig (4.91) Seed Actors function

```
1 if __name__ == "__main__":
2     print("Starting Actor Seeding...")
3     seed_actors()
```

- Thirty actors are then seeded to the database.

Fig (4.92) Loading existing actors

```
1 all_actors = Actor.query.all()
2 if len(all_actors) < 4:
3     print("Error: Need at least 4 actors in the database. Run actor_seeder.py first.")
4     return
5
```

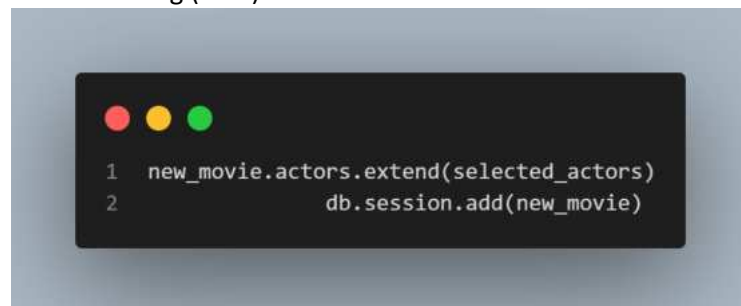
- The first thing was loading the existing actors from the database using the query.all() method, this is possible because the actors are seeded before the movies. It then checks to see there is at least four actors that have been seeded to the database, this is done because the seeding process need for at least four actors to be seeded per movie. If fewer actors were found an error message will appear.

Fig (4.93) Seed Actors function

```
1 f"Actors={[actor.name for actor in sample_movie.actors.all()]}")
```

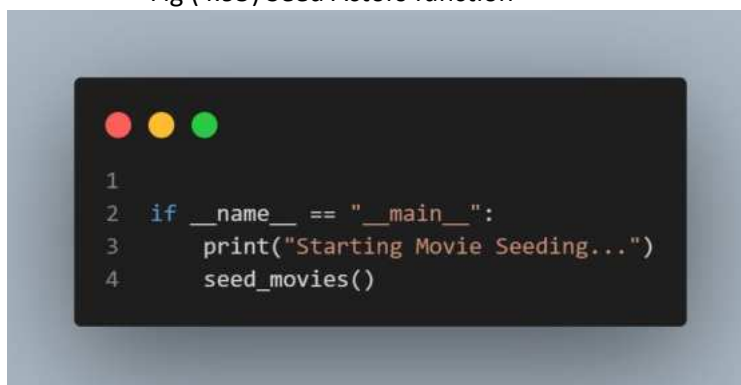
- Using the `select_actors = random.sample(all_actors, 4)` it sets for each movie to be seeded with four actors at random from the list of actors. This makes it so that the movies are seeded with a distinct set of actors each time.

Fig (4.94) Seed Actors function



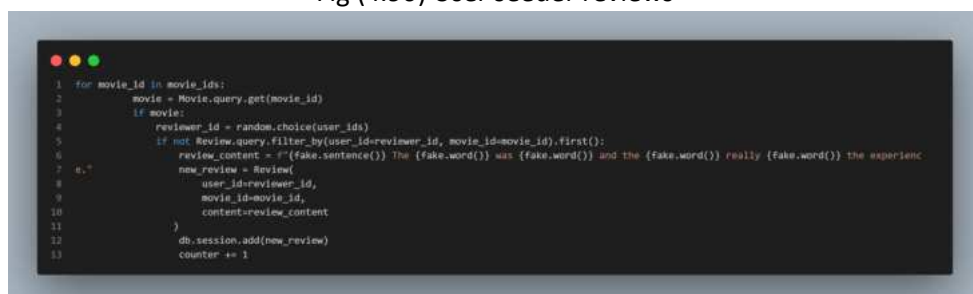
- When the four actors are chosen, they are then added to the movie with the `new_movie.actors.extend(selected_group)` method which links the actors to the movie by extending the actors relationship in the Movie model.
- The movie data is then committed in batches of one hundred and the commit is then underway.

Fig (4.95) Seed Actors function



- It then prints how many movies and actors were seeded displaying a message, if the operation fails an error message will appear.

Fig (4.96) User seeder reviews

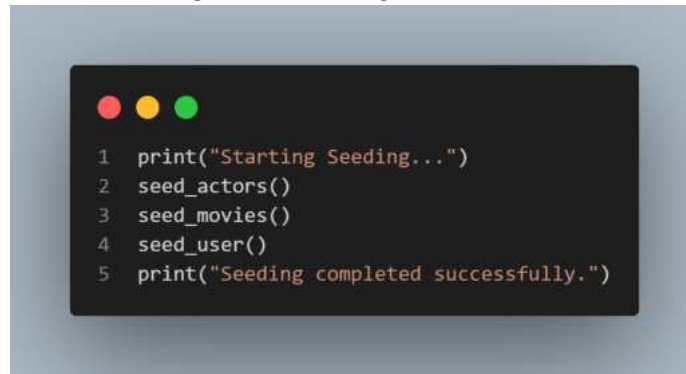


- This is the `user_seeder`, in this file the reviews for the application are seeded. Initially it starts with iterating through all the movie ids that were gotten from the database. A random user id is then selected from the list of existing users for the reviews. It then

checks for existing reviews already in the database. It then makes sure that the user has not written a review for the same movie which prevents duplication in the database, it then generates random content for the reviews using the faker library.

- When the review content is generated, the new review is then added to the database session.

Fig (4.97) Seeding the seeders



- The seed\_all was then updated to include the actor seeder, it seeds the actors first so that when the movie seeder is activated, it can use the actors that were previously seeded to seed the actors for the movies.

## 4.8 Sprint 5

In sprint 5 the development of the front end was underway beginning with initializing the React application. Tailwind and DaisyUI were then implemented for the styling of the application. The app.js file was then modified to manage the routes and render components, app.js also house authentication alongside the search function and the genre selection function.

Once the application was initialized the process of creating the pages began, the movie pages including the index, single, create, and edit were made to show off all the movies in the database, show the details of specific movies, and give the admin the ability to create, edit, and delete movies. The movie single page allows users to add movies to their watchlist, rate movies, and review them in. Similarly to the movie pages the watchlist pages include an index, single, edit, and create page but also houses the public watchlist index and single pages so the watchlists are displayed as well as the public watchlists. Any user can use the CRUD functionality for the watchlists and can make their watchlist public at any time and revert it in needed. The next pages that were made were the actor pages which gave the admin more functionality. The actors have an index, single, create and edit page where the admin can create and edit users, in the index the admin can delete, create, and edit the actors while also being able to assign the actors to any of the movies. In the movie single page, the actors can be viewed by users and removed by admins.

The next pages that were made was the home and profile pages, in the home page the user can view their recommended movies, the data from which is taking from the model in the backend, this recommendation data is displayed in home alongside the ratings the user has made when exploring

the movie section. The users can edit and delete their own ratings in the home page. The profile page displays the current user data of whoever has logged in. From this page the user can sign-out of their accounts which makes them navigate back to the login page.

The components in this application involved developing the login and register pages, the navbar, and the cards that would be used in the pages.

The API calls were separated into a different folder from the files called APIs, the APIs were stored here to limit the amount of code in the pages file.

PageNotFound was implemented so if the users type in the wrong URL this page will appear.

#### 4.8.1 Item 1

- App.js

Fig (4.98) Import & States



```
1 import { useState, useEffect } from "react";
2 import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
3 import "./index.css";
4 import axios from "axios";
5 import Login from "../components/login";
6 import Register from "../components/register";
7 import All from "../pages/movies";
8 import MovieSingle from "../pages/movies/single";
9 import MovieCreate from "../pages/movies/create";
10 import MovieEdit from "../pages/movies/edit";
11 import Watchlist from "../pages/watchlist";
12 import WatchlistSingle from "../pages/watchlist/single";
13 import WatchlistCreate from "../pages/watchlist/create";
14 import WatchlistEdit from "../pages/watchlist/edit";
15 import PageNotFound from "../pages/pageNotFound";
16 import Home from "../pages/home";
17 import Profile from "../pages/userProfile";
18 import Navbar from "../components/navBar";
19 import PublicWatchlists from "../pages/watchlist/public";
20 import PublicWatchlistSingle from "../pages/watchlist/publicSingle";
21 import ActorsAll from "../pages/actors/index";
22 import ActorCreate from "../pages/actors/create";
23 import ActorEdit from "../pages/actors/edit";
24 import ActorSingle from "../pages/actors/single";
25
26 function App() {
27   const [authenticated, setAuthenticated] = useState(false);
28   const [search, setSearch] = useState("");
29   const [selectedGenre, setSelectedGenre] = useState("");
```

- The state management is then initialized for the App function and manages the state variables authenticated, search, and selectGenre. The authentication state is used on all pages and components while the search and selectGenre are used in the navbar where they search for movies and the genre in the movie index page.

Fig (4.99) useEffect



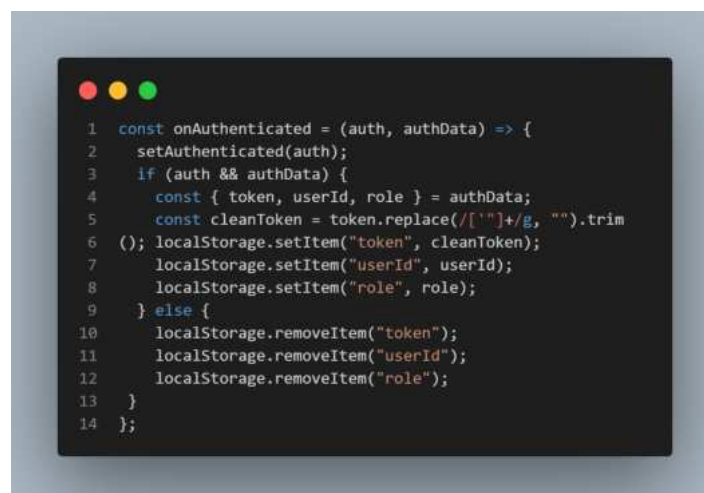
```

1  useEffect(() => {
2    const token = localStorage.getItem("token");
3    const userId = localStorage.getItem("userId");
4    const role = localStorage.getItem("role");
5
6    if (token && userId && role) {
7      axios
8        .get("http://127.0.0.1:5000/auth/current-user", {
9          headers: { Authorization: `Bearer ${token}` },
10         })
11        .then((response) => {
12          setAuthenticated(true);
13          localStorage.setItem("userId", response.data.user_id);
14          localStorage.setItem("role", response.data.role);
15        })
16        .catch((err) => {
17          setAuthenticated(false);
18          localStorage.removeItem("token");
19          localStorage.removeItem("userId");
20          localStorage.removeItem("role");
21        });
22    } else {
23      setAuthenticated(false);
24    }
25  }, []);

```

- The handling of the authentication state uses a useEffect hook which runs when the component mounts to check if the user already has authentication, it then retrieves the token, the users id, and the role they have been assigned and if all of the values exists a call is made to validate the token that was given, when the authentication is successful the users state is set to true and the user id and role are updated in the local storage. If this fails the token, user id, and role are removed from the local storage.

Fig (4.100) OnAuthenticate



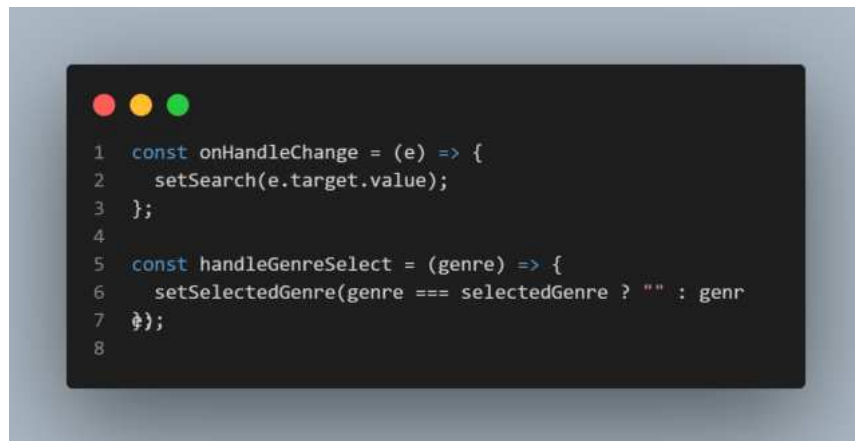
```

1  const onAuthenticated = (auth, authData) => {
2    setAuthenticated(auth);
3    if (auth && authData) {
4      const { token, userId, role } = authData;
5      const cleanToken = token.replace(/['"]+/g, "").trim
6      (); localStorage.setItem("token", cleanToken);
7      localStorage.setItem("userId", userId);
8      localStorage.setItem("role", role);
9    } else {
10     localStorage.removeItem("token");
11     localStorage.removeItem("userId");
12     localStorage.removeItem("role");
13   }
14 };

```

- Handling the authentication updates involved using the onAuthenticated(auth, authData) defines the function that updates the authentication state of the user when they login and logout, when true it stores the users token, id, and role, when a user signs out it removes the token, id, and role from local storage.

Fig (4.101) Search & Genres



- The search and genre functions work similarly, the search updates when the user types into the search bar the title of a movie and it appears along with similarly named movies, this worked similarly with the genre selector which updates the list of movies to include the selected genre a user has chosen. The search uses the `onHandleChange` function to handle the search and the genre selector uses the `handleGenreSelect` function to change the state when the genre is chosen.

Fig (4.102) Routes



- Using react router, the router component wraps around the entire app with the routes defining the different pages that are on the application. These routes include the movies, watchlists, actor, home, and profile pages, navbar.

#### 4.8.2 Item 2

- Components

Login

Fig (4.104) Imports and states

```
1 import axios from "axios";
2 import { useState, useEffect } from "react";
3 import { Link, useNavigate } from "react-router-dom";
4
5 const Login = ({ authenticated, onAuthenticated }) =>
6 { const errStyle = { color: "red" };
7   const navigate = useNavigate();
8
9   const [form, setForm] = useState({
10     email: "",
11     password: "",
12   });
13   const [errMessage, setErrMessage] = useState("");
```

- The state is initialized with two states being the form which contains the data for the email and password for the users and admin and errMessage which stores as a string, when login fails the error message will appear informing the user of the error.

Fig (4.106) Handle Click function

```
1 const handleClick = () => {
2   axios
3     .post(
4       "http://127.0.0.1:5000/auth/login",
5       {
6         email: form.email,
7         password: form.password,
8       },
9       { withCredentials: true }
10    )
11    .then((response) => {
12      console.log("Login response:", response.data);
13      onAuthenticated(true, {
14        token: response.data.access_token,
15        userId: response.data.user_id,
16        role: response.data.role,
17      });
18      navigate("/home");
19    })
20    .catch((err) => {
21      console.error("Error:", err.response?.data || err.message);
22      setErrMessage(err.response?.data?.error || "Login failed");
23    });
24 };
25
```

- The handleClick function sends a POST request to the backend with the email and password that was inputted, if the credentials match the ones in the database the login will be successful and a response that contains the users id, token, and role and they will be redirected to the home page. If not, an error message will appear.

Fig (4.107) Handle Form Function





- The handleForm function changes the form state to the updated version when the user types in the email and password.

Fig (4.108)

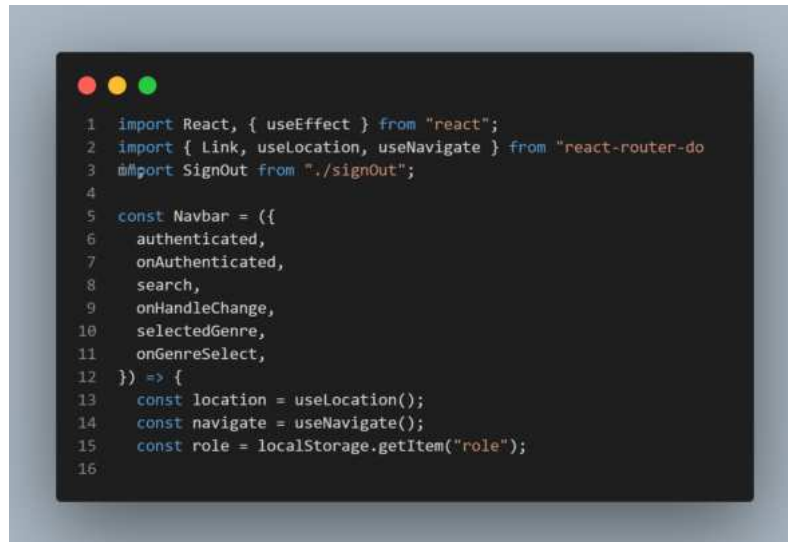


- In the return it displays the login form with the title of login, the two fields email and password, the button the user clicks to submit the form, an error message of the operation fails, and finally a link to the registration page in case a new user wants to register an account.

Navbar

- The Navbar component is the navigation for the application, it provides the user with links to the various pages and other functionalities in the application, some of these functionalities are blocked off depending on the user's role when they have logged in.

Fig (4.110) States



```

1 import React, { useEffect } from "react";
2 import { Link, useLocation, useNavigate } from "react-router-dom";
3 import SignOut from "../signOut";
4
5 const Navbar = ({
6   authenticated,
7   onAuthenticated,
8   search,
9   onChange,
10  selectedGenre,
11  onGenreSelect,
12 }) => {
13   const location = useLocation();
14   const navigate = useNavigate();
15   const role = localStorage.getItem("role");
16

```

- The navbar component takes in many props such as authenticated which shows if the user is logged in or not, the onAuthenticate function which updates the authentication of the user, the search function which houses the queries on the search. The onChange to update the search, the genreSelector for the genre selection and the function that handles the genre selection change onGenreSelect.

Fig (4.111) Not displaying Nav in login or register



```

1 useEffect(() => {
2   console.log("Navbar rendered for path:", location.pathname);
3 }, [location.pathname]);
4
5 if (location.pathname === "/" || location.pathname === "/register") {
6   return null;
7 }

```

- In the useEffect, it logs the path every time the location.pathname changes which was used for debugging the navigation. if the pathname is login or register the navbar will not display.

Fig (4.112) Profile picture and sign out component

```

1 <div className="w-10 rounded-full">
2   
6 </div>
7 </div>
8 <ul>
9   <li>
10     <div className="menu menu-sm dropdown-content bg-base-100 rounded-box z-[1] mt-3 w-52 p-2 shadow"
11       >
12       <div>
13         <Link to="/profile" className="justify-between">
14           Profile <span className="badge" >New</span>
15         </Link>
16       </div>
17       <div>
18         <button onClick={onAuthenticated} />
19       </div>
20     </li>
21 </ul>
22 </div>
23 </div>
24
25 <div className="lg:hidden">
26   <div className="dropdown">
27     <div tabIndex={0} role="button" className="

```

- The user avatar was used from the DaisyUI library it displays a user profile picture along with the option to sign out. The Sign Out is a different component that give the user the ability to log out of their accounts which removes the user id, token, and role removing the authentication. It works as a dropdown and sets the authentication from true to false.

Fig (4.113) Links

```

1 return {
2   <div className="navbar bg-base-100 shadow-sm">
3     <div className="navbar-start">
4       <Link to="/home" className="btn btn-ghost text-xl">
5         MovieMuse
6       </Link>
7     </div>
8
9     <div className="navbar-center hidden lg:flex">
10      <ul className="menu menu-horizontal px-1">
11        <li>
12          <Link to="/home" className="hover:underline">
13            Home
14          </Link>
15        </li>
16        <li>
17          <Link to="/movies" className="hover:underline">
18            Movies
19          </Link>
20        </li>
21        <li>
22          <Link to="/watchlist" className="hover:underline">
23            Watchlist
24          </Link>
25        </li>
26        <li>
27          <Link to="/profile" className="hover:underline">
28            Profile
29          </Link>
30        </li>
31        <li>
32          <Link to="/public-watchlists" className="hover:underline">
33            Public Watchlists
34          </Link>
35        </li>
36        <li>
37          <div>
38            <Link to="/actors" className="hover:underline">
39              Actors
40            </Link>
41          </div>
42        </li>
43      </ul>
44    </div>

```

- The links render the movie all, watchlist all, public watchlist all, profile, and for admins only the actor link.

Fig (4.115) Movie card

```

1  return (
2    <div className="card bg-base-100 w-96 shadow-sm hover:shadow-lg transition-shadow relativ
3    e"> <figure className="relative w-full h-64 overflow-hidden">
4      <Link
5        to={
6          to ||
7          (movie && movie.id && movie.id !== "Unknown"
8            ? "/movies/${movie.id}"
9            : "#")
10       }
11      state={{ imageUrl: displayImageUrl }}
12    >
13      <img
14        src={displayImageUrl}
15        alt={title}
16        className="w-full h-full object-cover rounded-lg shadow-md"
17      />
18    </Link>
19  </figure>
20  <div className="card-body py-6">
21    <Link
22      to={
23        to ||
24        (movie && movie.id && movie.id !== "Unknown"
25          ? "/movies/${movie.id}"
26          : "#")
27      }
28      state={{ imageUrl: displayImageUrl }}
29      className="block"
30    >
31      <h2 className="card-title text-black hover:text-primary">{title}</h2>
32    </Link>
33    <p className="text-gray-600">{genres}</p>
34    {movie && movie.rating && (
35      <p className="text-sm text-yellow-600">
36        Rating: {movie.rating.toFixed(2)}
37      </p>
38    )}
39    {movie &&
40      (movie.ratingsCount !== undefined ||
41        movie.reviewsCount !== undefined) && (
42      <p className="text-sm text-gray-500">
43        {movie.ratingsCount !== undefined
44          ? `${movie.ratingsCount} ratings`
45          : "0 ratings"}
46        {movie.reviewsCount !== undefined
47          ? `, ${movie.reviewsCount} reviews`
48          : ", 0 reviews"}
49      </p>
50    )}

```

```

1  {showActions && (
2    <div className="mt-4 flex justify-end gap-2">
3      {isEditing ? (
4        <div className="flex items-center gap-2">
5          <input
6            type="text"
7            value={newRatingValue}
8            onChange={onRatingChange}
9            placeholder="1.0-5.0"
10           className="input input-bordered w-20"
11         />
12         <button onClick={onSave} className="btn btn-success btn-sm">
13           Save
14         </button>
15         <button onClick={onCancel} className="btn btn-neutral btn-sm">
16           Cancel
17         </button>
18       </div>
19       ) : (
20         <>
21           <button onClick={onEdit} className="btn btn-warning btn-sm">
22             Edit
23           </button>
24           <button onClick={onDelete} className="btn btn-error btn-sm">
25             Delete
26           </button>
27         </>
28       )}
29     </div>
30   )}
31 </div>
32 </div>
33 );
34 };
35
36 export default MovieCard;
37

```

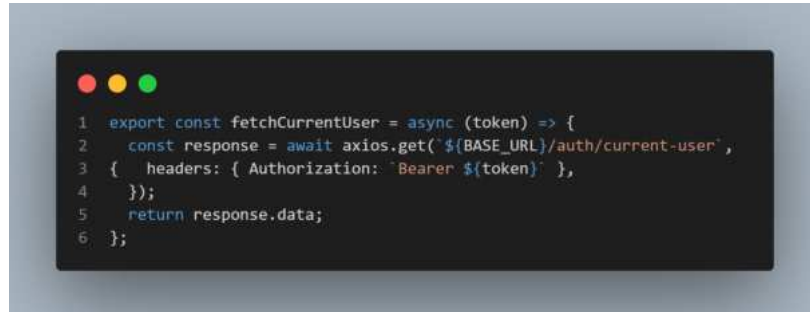
- The movie card is a component that displays the data of the movies in the card, it takes in multiple props which include the movie, containing the movie data, to which is a link path for the navigation, and the showActions which displays the button for delete when deleting a movie.
- This component renders an image taken from daisyUI as a placeholder for when the pictures are taken from the backend from the static folder with the public file that contains images. It then renders the data for the movie as the card with all the data for the movie and ratings, if the movie has a ratings and review count, they are displayed in the card.
- If the showAction is set to true the component shows buttons for editing and deleting for the movie, if the isEdit is set to false, it then shows the edit and delete buttons, if set to true it then renders the edit and delete button for the rating with save and cancel buttons.
- Other components exist being the register, movie single card, watchlist cards, and other functions.

#### 4.8.3 Item 3

- APIs
  - This file holds all the API calls for the movies, it uses axios to perform the operations which include other API calls from the auth route in the backend, the watchlist, and

ratings and reviews, for this file we'll discuss only the movie calls and discuss the others in a separate file excluding the user calls which will be discussed.

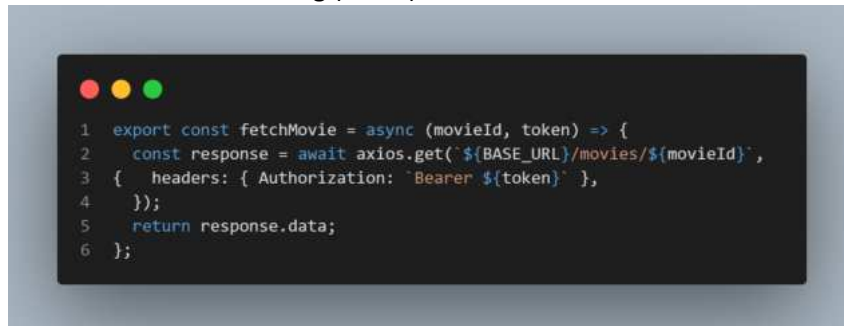
Fig (4.116) Fetch Current User

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript function named `fetchCurrentUser` which is an asynchronous function taking a `token` parameter. The function uses `axios.get` to make a GET request to ``${BASE_URL}/auth/current-user`` with headers including `Authorization: Bearer ${token}`. It returns the `response.data`.

```
1 export const fetchCurrentUser = async (token) => {
2   const response = await axios.get(`${BASE_URL}/auth/current-user`,
3   { headers: { Authorization: `Bearer ${token}` },
4   });
5   return response.data;
6 };
```

- The `fetchCurrentUser` call sends a GET request that gets the current user that is authenticated.

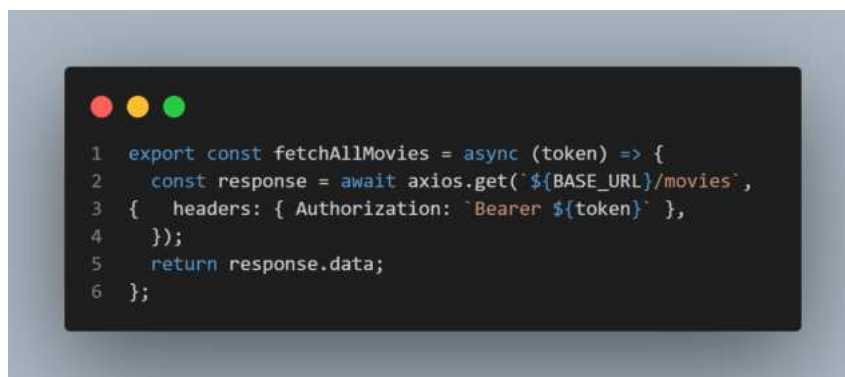
Fig (4.117) Fetch Movie

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript function named `fetchMovie` which is an asynchronous function taking `movieId` and `token` parameters. The function uses `axios.get` to make a GET request to ``${BASE_URL}/movies/${movieId}`` with headers including `Authorization: Bearer ${token}`. It returns the `response.data`.

```
1 export const fetchMovie = async (movieId, token) => {
2   const response = await axios.get(`${BASE_URL}/movies/${movieId}`,
3   { headers: { Authorization: `Bearer ${token}` },
4   });
5   return response.data;
6 };
```

- The `fetchMovies` is a GET request that gets the data of the specific movie also using the token for authentication.

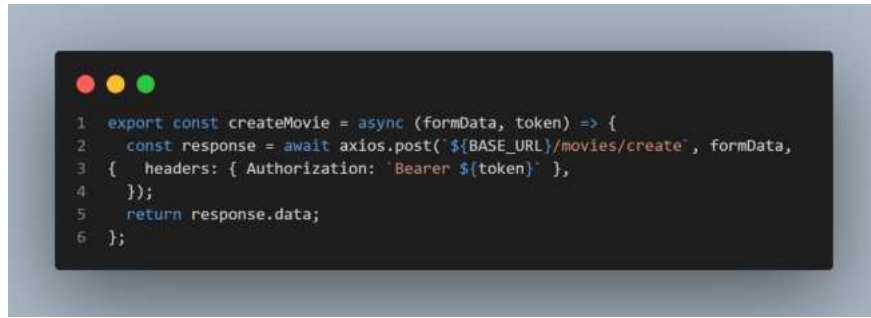
Fig (4.118) Fetch All Movies API

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript function named `fetchAllMovies` which is an asynchronous function taking a `token` parameter. The function uses `axios.get` to make a GET request to ``${BASE_URL}/movies`` with headers including `Authorization: Bearer ${token}`. It returns the `response.data`.

```
1 export const fetchAllMovies = async (token) => {
2   const response = await axios.get(`${BASE_URL}/movies`,
3   { headers: { Authorization: `Bearer ${token}` },
4   });
5   return response.data;
6 };
```

- The `fetchAllMovie` call is a GET request that gets all the movie data from the backend, it also includes the token for authentication.

Fig (4.119) Create Movies API



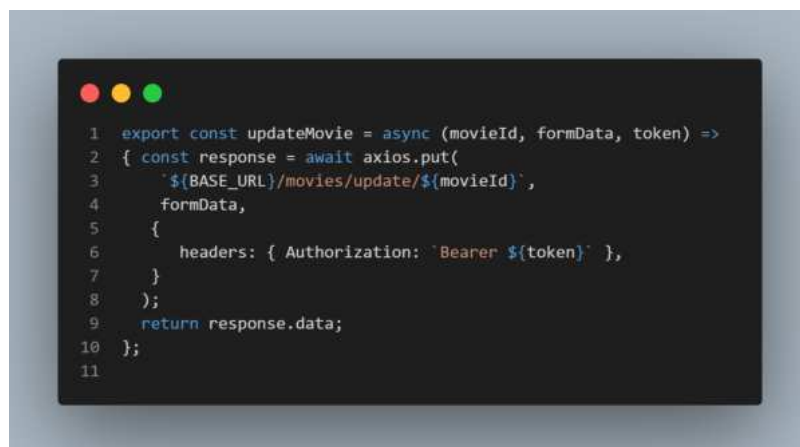
```

1 export const createMovie = async (formData, token) => {
2   const response = await axios.post(`${BASE_URL}/movies/create`, formData,
3   { headers: { Authorization: `Bearer ${token}` },
4   });
5   return response.data;
6 };

```

- The createMovie call sends a POST request to the backend, it sends the request data from the formData and the token for authentication.

Fig (4.120) Update Movie API



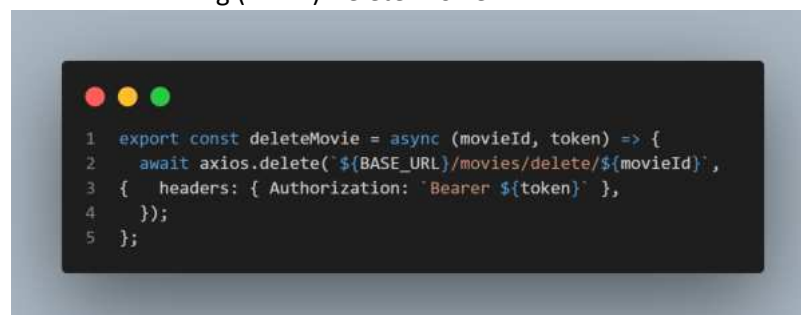
```

1 export const updateMovie = async (movieId, formData, token) =>
2 { const response = await axios.put(
3   `${BASE_URL}/movies/update/${movieId}`,
4   formData,
5   {
6     headers: { Authorization: `Bearer ${token}` },
7   }
8 );
9 return response.data;
10 };
11

```

- The updateMovie call is a PUT request to edit the details of the movie data, it uses the movie id and the formData and sends them to the backend. It also sends the token for the authentication of the user.

Fig (4.121) Delete Movie API



```

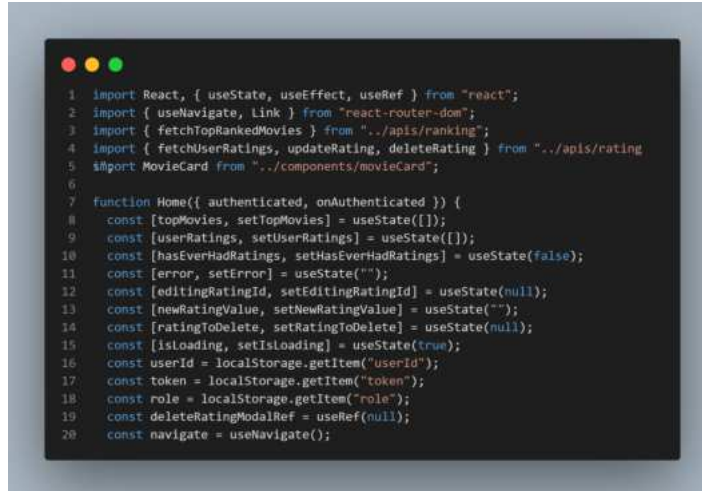
1 export const deleteMovie = async (movieId, token) => {
2   await axios.delete(`${BASE_URL}/movies/delete/${movieId}`,
3   { headers: { Authorization: `Bearer ${token}` },
4   });
5 };

```

- The delete Movie call sends a DELETE request which removes the movie from the database entirely. It sends the request with the movie id and the token to authenticate the call.
- The watchlist, ranking, rating, and actors also have an API file that stores the API calls so that it can be reused in multiple pages if needed.

#### 4.8.4 Item 3

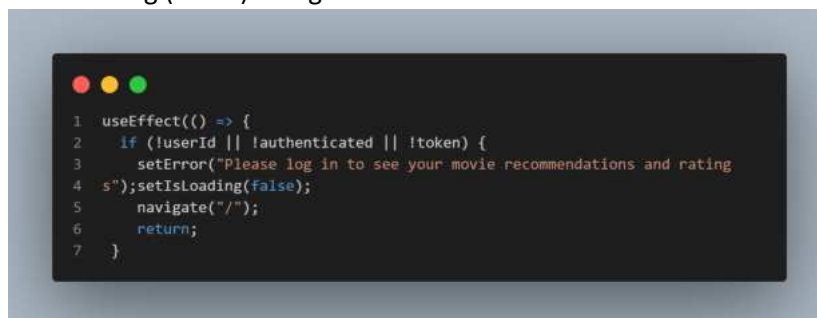
- Pages
- Fig (4.122) Home



```
1 import React, { useState, useEffect, useRef } from "react";
2 import { useNavigate, Link } from "react-router-dom";
3 import { fetchTopRankedMovies } from "../apis/ranking";
4 import { fetchUserRatings, updateRating, deleteRating } from "../apis/rating";
5 import MovieCard from "../components/movieCard";
6
7 function Home({ authenticated, onAuthenticated }) {
8   const [topMovies, setTopMovies] = useState([]);
9   const [userRatings, setUserRatings] = useState([]);
10  const [hasEverHadRatings, setHasEverHadRatings] = useState(false);
11  const [error, setError] = useState("");
12  const [editingRatingId, setEditingRatingId] = useState(null);
13  const [newRatingValue, setNewRatingValue] = useState("");
14  const [ratingToDelete, setRatingToDelete] = useState(null);
15  const [isLoading, setIsLoading] = useState(true);
16  const userId = localStorage.getItem("userId");
17  const token = localStorage.getItem("token");
18  const role = localStorage.getItem("role");
19  const deleteRatingModalRef = useRef(null);
20  const navigate = useNavigate();
```

- The Home function takes in two props, authenticated for the user login and onAuthenticated. Many states are initialized including the top movies that store the ranked recommended movies, the userRatings which is the list of movies that were rated by the user, hasEverHadRatings to track of the user has made a rating before, error for storing the error messages that will display if operations fail. editingRating and newRatingValue are used if the user decided to change their rating, and ratingDelete which tracks the ratings that have been removed. It also goes through the users id, token, and role from local storage which initializes the useRef for the model when removing a rating.

Fig (4.123) Navigational useEffect



```
1 useEffect(() => {
2   if (!userId || !authenticated || !token) {
3     setError("Please log in to see your movie recommendations and rating");
4     setIsLoading(false);
5     navigate("/");
6     return;
7   }
8 }
```

- Getting the user ratings is a useEffect and runs when components are loaded or if the user's id, authenticated, or the token changes checking if the user is authenticated, if they are it redirects to home and an error occurs, if the authentication is true the API call is run and it gets the ratings, if the ratings made by the user do exist it stores them in the hasEverHadRating and it is set to true, if the user is unauthenticated a message will appear stating the unauthorized access.



```

1  try {
2    const data = await fetchUserRatings(userId, token);
3    console.log("User ratings response:", data.rated_movies);
4    const validRatings = data.rated_movies
5      .filter((rating) => rating.movie_id || rating.id)
6      .map((rating) => ({
7        ...rating,
8        movie_title: rating.movie_title || rating.title || "Untitled",
9        image_url: rating.image_url || "/static/movies/bloodborne1.jpg",
10      }));
11    setUserRatings(validRatings);
12    if (validRatings.length > 0) {
13      setHasEverHadRatings(true);
14    }
15  } catch (err) {
16    setError(err.error || "Failed to fetch user ratings");
17    if (err.response?.status === 401) {
18      setError("Unauthorized: Please log in again.");
19    }
20  } finally {
21    setTimeout(() => setIsLoading(false), 1000);
22  }
23 };
24
25 fetchRatings();
26 }, [userId, navigate, authenticated, token]);
27

```

```

1  useEffect(() => {
2    if (userRatings.length > 0 || hasEverHadRatings) {
3      const fetchTopMovies = async () => {
4        try {
5          const data = await fetchTopRankedMovies(userId, token);
6          console.log("Top movies response:", data.top_ranked_movies);
7          const normalizedMovies = data.top_ranked_movies.map((movie) =>
8            ({
9              ...movie,
10             movie_title: movie.title || movie.movie_title || "Untitled",
11             image_url: movie.image_url || "/static/movies/bloodborne1.jpg",
12           }));
13          setTopMovies(normalizedMovies);
14        } catch (err) {
15          setError(err.error || "Failed to fetch top movies");
16          if (err.response?.status === 401) {
17            setError("Unauthorized: Please log in again.");
18          }
19        }
20      };
21      fetchTopMovies();
22    } else {
23      setTopMovies([]);
24    }
25  }, [
26    userRatings,
27    hasEverHadRatings,
28    userId,
29    token,
30  ]);
31

```

- Getting the top ranked movies is also a useEffect that runs when the userRating, user id, token, or hasEverHadRating is changed. If the user has rated a movie in the past or rated one when they have registered it will then get the top ranked movies that are the recommendation of the application using an API call to the ranked movies, it stores ranked movies in topMovies. If the user does not have any ratings the ranked movies will not appear until the user goes and rates a movie.

Fig (4.124) Handle Edit Rating & Handle Saved Rating

```
1  const handleEditRating = (rating) => {
2    setEditingRatingId(rating.id);
3    setNewRatingValue(rating.rating.toString());
4  };
5
6  const handleSaveRating = async (ratingId) => {
7    try {
8      const ratingValue = parseFloat(newRatingValue);
9      if (isNaN(ratingValue) || ratingValue < 1.0 || ratingValue > 5.0) {
10        setError("Rating must be a number between 1.0 and 5.0");
11        return;
12      }
13      await updateRating(ratingId, ratingValue, token);
14      setUserRatings(
15        userRatings.map((rating) =>
16          rating.id === ratingId ? { ...rating, rating: ratingValue } : rating
17        )
18      );
19      setEditingRatingId(null);
20      setNewRatingValue("");
21      setError("");
22    } catch (err) {
23      setError(err.error || "Failed to update rating");
24      if (err.response?.status === 401) {
25        setError("Unauthorized: Please log in again.");
26      }
27    }
28  };
29  };
```

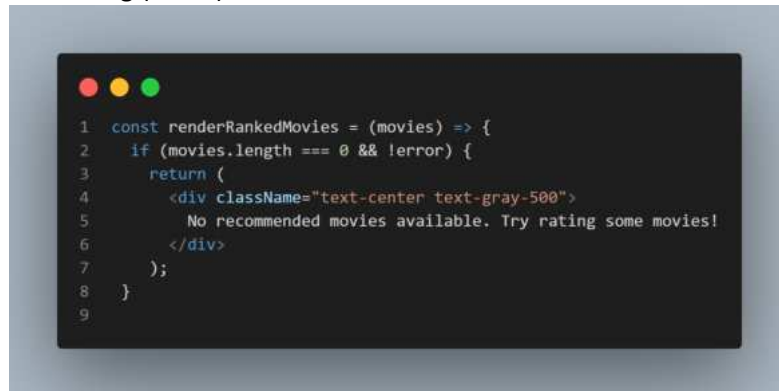
- The handleEditRating is a function that sets the id of the rating that is being updated and stores the new value in the state. When the updated rating is submitted the function handleSavedRating then validates the changed input and updates the rating through the API and the state. If the operation is successful the rating will be updated, if not an error message will appear.

Fig (4.125) Handle Delete Rating & Confirm Delete Rating

```
1  const handleDeleteRating = (ratingId) => {
2    setRatingToDelete(ratingId);
3    deleteRatingModalRef.current.showModal();
4  };
5
6  const confirmDeleteRating = async () => {
7    if (!ratingToDelete) return;
8    try {
9      await deleteRating(ratingToDelete, token);
10     setUserRatings(
11       userRatings.filter((rating) => rating.id !== ratingToDelete
12     );
13     setRatingToDelete(null);
14     deleteRatingModalRef.current.close();
15     setError("");
16   } catch (err) {
17     setError(err.error || "Failed to delete rating");
18     if (err.response?.status === 401) {
19       setError("Unauthorized: Please log in again.");
20     }
21   }
22  };
23  };
```

- The delete function `handleDeleteRating` is used in the `ratingDelete` that shows the modal making sure that the user wants to delete their rating, if they do it then runs the `confirmDeleteRating` making the DELETE API call removing the rating.

Fig (4.126) Render Ranked Movies



- It renders the ranked movies using `renderRankedMovies` function which loads the `movieCards` with the ranked data doing this by mapping through the `topMovies` array.

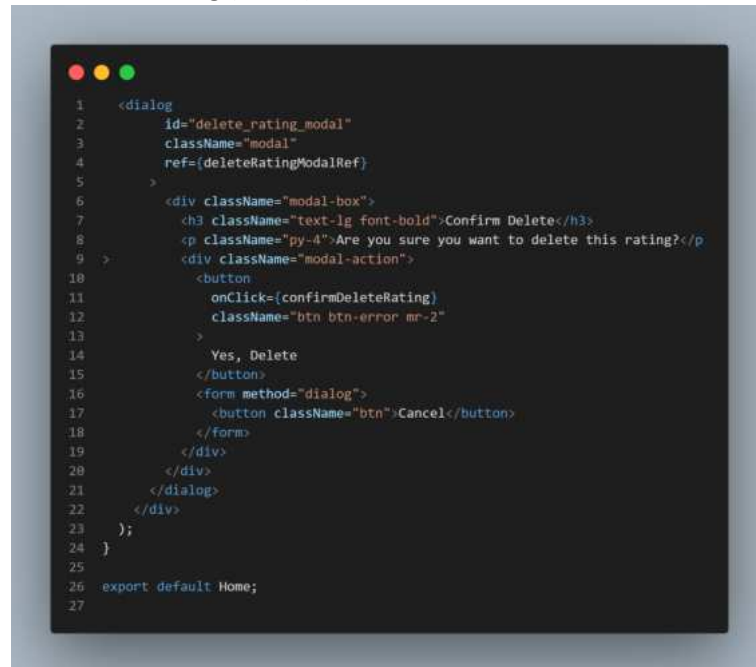
Fig (4.127) Return Statement



- In the return it shows a welcome message to the user or admin that logs in, if the user has not made a rating yet a message appears telling the user to explore the movie

section, if they do have ratings, it displays them along with the ranked movies recommended to the user with the rated movies and the edit and delete buttons.

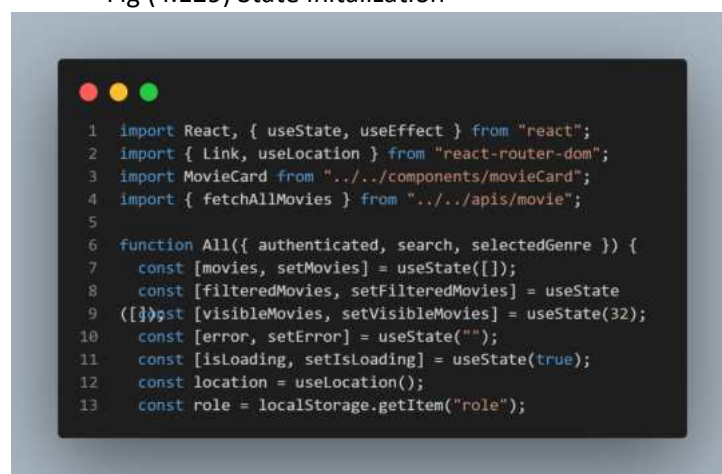
Fig (4.128) Modal



- The delete modal for the ratings renders showing dialog with two buttons, the delete confirms the removal of the rating and cancel closes the modal.

Movie all

Fig (4.129) State initialization



- The All initializes the states used for this file, the movie stores the movie data, the filtered movie state are filters used for the search and selectGenre functions, the visible movies state stores the movies that can be displayed at one time, and the error handles any errors involving the authentication of a user or the GET request failing.

Fig (4.130) useEffect for movie data



```
1  useEffect(() => {  
2    fetchMovies();  
3  }, [location.pathname]);
```

- The first useEffect gets the movie data from the server, being the Flask backend.

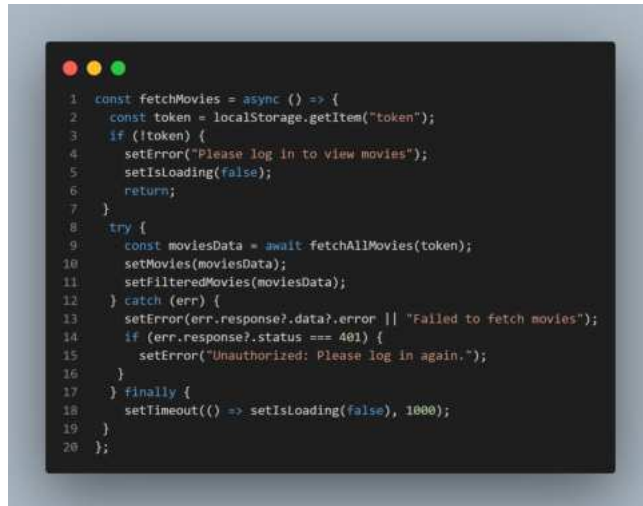
Fig (4.131) Search useEffect



```
1  
2  useEffect(() => {  
3    let filtered = movies;  
4  
5    if (search && search.length > 1) {  
6      filtered = filtered.filter((movie) =>  
7        (movie.movie_title || "").toLowerCase().includes(search.toLowerCase())  
8      );  
9    }  
10  
11    if (selectedGenre) {  
12      filtered = filtered.filter((movie) =>  
13        (movie.movie_genres || "")  
14          .toLowerCase()  
15          .includes(selectedGenre.toLowerCase())  
16        );  
17    }  
18  
19    setFilteredMovies(filtered);  
20    setVisibleMovies(32);  
21  }, [movies, search, selectedGenre]);
```

- The second useEffect is used when the movie, search data, or the selectGenre data is changed. When a user types in the search bar it filters through the movie list to display the movies correlating with what the user typed, similarly the selectGenre changes what movies are displaying based on what the user has selected. It edits the list, and the visible count of movies is counted to thirty-two.

Fig (4.132) Fetch Movies

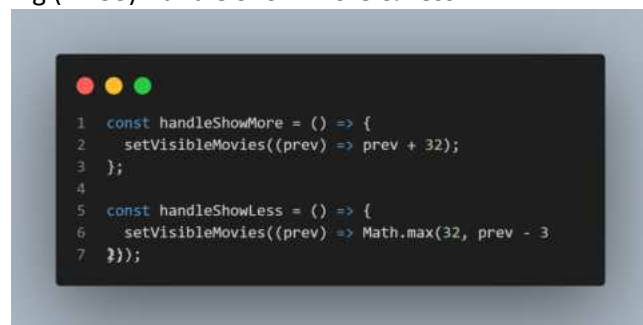


```
1  const fetchMovies = async () => {
2    const token = localStorage.getItem("token");
3    if (!token) {
4      setError("Please log in to view movies");
5      setIsLoading(false);
6      return;
7    }
8    try {
9      const moviesData = await fetchAllMovies(token);
10     setMovies(moviesData);
11     setFilteredMovies(moviesData);
12   } catch (err) {
13     setError(err.response?.data?.error || "Failed to fetch movies");
14     if (err.response?.status === 401) {
15       setError("Unauthorized: Please log in again.");
16     }
17   } finally {
18     setTimeout(() => setIsLoading(false), 1000);
19   }
20   };

```

- The function fetchMovies gets the user's token from local storage, it exists the operation proceeds, if not an error will occur. When successful it calls the API function set in the movie file in the apis folder, it updates the movies and filteredMovies states.

Fig (4.133) Handle Show More & Less



```
1  const handleShowMore = () => {
2    setVisibleMovies((prev) => prev + 32);
3  };
4
5  const handleShowLess = () => {
6    setVisibleMovies((prev) => Math.max(32, prev - 32));
7  };

```

- The handleShowMore and handleShowLess functions are used to load movies to be displayed. It loads an additional thirty-two movies or takes away thirty-two movies from being displayed and updates the visibleMovies state when used, it cannot go below thirty-two.

Fig (4.134) Return Statement

```

1  return (
2    <div className="container mx-auto p-8">
3      <div className="flex justify-between items-center mb-8">
4        <h2 className="text-3xl font-bold text-primary">Movies</h2>
5        {authenticated && role === "admin" && (
6          <Link to="/movies/create" className="btn btn-success">
7            Create New Movie
8          </Link>
9        )}
10     </div>
11     {error && <div className="alert alert-error mb-8">{error}</div>}
12     <div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-
13 8">
14       {filteredMovies.slice(0, visibleMovies).map((movie) => (
15         <MovieCard key={movie.id} movie={movie} />
16       ))}
17     </div>
18     <div className="text-center mt-8 flex justify-center gap-4">
19       {visibleMovies < filteredMovies.length && (
20         <button onClick={handleShowMore} className="btn btn-primary">
21           Show More
22         </button>
23       )}
24       {visibleMovies > 32 && (
25         <button onClick={handleShowLess} className="btn btn-secondary">
26           Show Less
27         </button>
28       )}
29     </div>
30   );
31 }
32
33 export default All;
34

```

- In the return it renders the create button the admins can use to create movies which is linked to the movie create page, if an error occur it will display, the filteredMovies are the rendered which shows a count of thirty-two and it also renders the movie card with the movie data. Below is the show more and show less buttons to either gain an additional list of movies or take away the movies from the view.
- The movie all pages displays all the movie data, similar pages exist in the application such as the public watchlist all page, the watchlist all page, and the actor all page. These pages follow a similar structure but contain contrasting functions that operate similarly to the other all pages.

## Movie Single



Fig (4.135) Imports, State variables, and useRef

```

1  import React, { useState, useEffect, useRef } from "react";
2  import { useParams, useNavigate, Link } from "react-router-dom";
3  import { createRating } from "../../apis/ratings";
4  import { updateWatchlist } from "../../apis/watchlist";
5  import { fetchMovie, deleteMovie } from "../../apis/movie";
6  import { fetchWatchlists } from "../../apis/watchlist";
7  import {
8    fetchReviews,
9    createReview,
10   updateReview,
11   deleteReview,
12 } from "../../apis/reviews";
13 import { removeActorFromMovie } from "../../apis/actor";
14 import MovieSingleCard from "../../components/movieSingleCard";
15
16 function MovieSingle({ authenticated }) {
17   const { movieId } = useParams();
18   const [movie, setMovie] = useState(null);
19   const [watchlists, setWatchlists] = useState([]);
20   const [reviews, setReviews] = useState([]);
21   const [selectedWatchlistId, setSelectedWatchlistId] = useState(
22     null
23   );
24   const [reviewContent, setReviewContent] = useState("");
25   const [editingReviewId, setEditingReviewId] = useState(null);
26   const [actorToRemove, setActorToRemove] = useState(null);
27   const userId = localStorage.getItem("userId");
28   const token = localStorage.getItem("token");
29   const role = localStorage.getItem("role");
30   const [error, setError] = useState("");
31   const [reviewToDelete, setReviewToDelete] = useState(null);
32   const deleteMovieModalRef = useRef(null);
33   const deleteReviewModalRef = useRef(null);
34   const deleteActorModalRef = useRef(null);
35   const navigate = useNavigate();

```

- In the single page for the movies hooks are imported including useState, useEffect, and useRef, the API calls from the apis folder to manage the movie, watchlist, actor, ratings, and review data, useParams is used so that the movies id can be used from the URL, useNavigate is also imported for navigation.
- In this file there are many state variables, the movie stores the movie data, watchlists store watchlist data, the review stores the review data, selectWatchlistId tracks the watchlist the user or admin has selected, this then gives the user the option to add the movie to the watchlist. The review content state stores the review content the user has written. The editReviewId state stores the review id that the user is editing, the actorToRemove state stores the id of the actor being removed from the movie, the error state houses the error messages, and the reviewDelete stores the reviews id that is being deleted.
- The useRef is hooking the deleteMovieRef, the deleteReviewRef, and the deleteActorRef, this is used for the modal when the user and admin want to delete a review, and the delete actor and movie is for when the admin wants to remove the movie or the actors in the movie.



Fig (4.136) useEffect

```

1  useEffect(() => {
2    if (!movieId) return;
3
4    if (!token || !authenticated) {
5      setError("Please log in to view this page");
6      navigate("/");
7      return;
8    }
9
10   const fetchMovieData = async () => {
11     console.log("Token used for fetching movie:", token);
12     try {
13       const response = await fetchMovie(movieId, token);
14       console.log("Fetched movie:", response);
15       setMovie(response);
16     } catch (err) {
17       setError(err.response?.data?.error || "Failed to fetch movie");
18       if (err.response?.status === 401)
19         setError("Unauthorized: Please log in again.");
20     }
21   };
22
23   const fetchWatchlistData = async () => {
24     console.log("Token used for fetching watchlist:", token);
25     try {
26       const response = await fetchWatchlists(userId, token);
27       setWatchlists(response);
28       //
29       const mylist = response.find(item => item.title === "My list");
30       if (mylist) setSelectedWatchlistId(mylist.id);
31     } catch (err) {
32       console.error("Fetch watchlists error:", err);
33       setError(err.response?.data?.error || "Failed to fetch watchlists");
34       if (err.response?.status === 401)
35         setError("Unauthorized: Please log in again.");
36     }
37   };
38
39   const fetchReviewData = async () => {
40     console.log("Token used for fetching reviews:", token);
41     try {
42       const response = await fetchReviews(movieId, token);
43       setReviews(response);
44     } catch (err) {
45       console.error("Fetch reviews error:", err);
46       setError(err.response?.data?.error || "Failed to fetch reviews");
47       if (err.response?.status === 401)
48         setError("Unauthorized: Please log in again.");
49     }
50   };
51
52   fetchMovieData();
53   if (userId && authenticated) {
54     fetchWatchlistData();
55     fetchReviewData();
56   }, [movieId, userId, navigate, authenticated, token]);

```

- The useEffect runs when mounted or when the user id, movie id, or token is changed. It checks the user to see if they are authenticated if they are directing them to the home page if not an error will appear. the fetchMovieData, fetchWatchlistData, and fetchReviewData are then defined to get the movie data, watchlist data, and review data. Each have error handling of the operations fails.

Fig (4.137) Handle Adding to Watchlist

```

1  const handleAddToWatchlist = async () => {
2    if (!userId || !authenticated) {
3      setError("Please log in to add to watchlist");
4      navigate("/");
5      return;
6    }
7    if (!selectedWatchlistId) {
8      setError("Please select a watchlist");
9      return;
10   }
11   console.log("Token used for adding to watchlist:", token);
12   try {
13     await updateWatchlist(
14       selectedWatchlistId,
15       { user_id: userId, movie_id: movieId },
16       token
17     );
18     const watchlistResponse = await fetchWatchlists(userId, token);
19     setWatchlists(watchlistResponse);
20   } catch (err) {
21     setError(err.response?.data?.error || "Failed to add to watchlist");
22     if (err.response?.status === 401)
23       setError("Unauthorized: Please log in again.");
24   }
25 };

```

- The handleAddingToWatchlist function allows the users and admin to add the movie they are currently viewing into one of their watchlists, the function checks if the user is

authenticated and if the watchlists selected exist and if it does make the API call to POST the movie to the watchlist. If the operation was successful the movie will be added, if not or if the movie is already in the watchlist an error will occur.

Fig (4.138) Handle Rate

```
1  const handleRate = async (rating) => {
2    if (!userId || !authenticated) {
3      setError("Please log in to rate movies");
4      navigate("/");
5      return;
6    }
7    console.log("Token used for rating:", token);
8    try {
9      await createRating(userId, movieId, rating, token);
10   } catch (err) {
11     setError(err.response?.data?.error || "Failed to rate movie");
12     if (err.response?.status === 401) {
13       setError("Unauthorized: Please log in again.");
14     }
15   }
16 }
```

- The handleRate function allows for the current movie to be rated by the user and admin. It checks the validation of the user before allowing for the rating to be posted. The rating if the process was a success will be sent to the database, if not an error will appear.

Fig (4.139) Handle Delete

```
1  const handleDelete = () => {
2    if (!userId || !authenticated) {
3      setError("Please log in to delete movies");
4      navigate("/");
5      return;
6    }
7    deleteMovieModalRef.current.showModal();
8  };
9
10 const confirmDeleteMovie = async () => {
11   console.log("Token used for deleting movie:", token);
12   try {
13     await deleteMovie(movieId, token);
14     deleteMovieModalRef.current.close();
15     navigate("/movies");
16   } catch (err) {
17     setError(err.response?.data?.error || "Failed to delete movie");
18     if (err.response?.status === 401) {
19       setError("Unauthorized: Please log in again.");
20     }
21   }
22 }
```

- The handleDelete is the function for the modal that displays before the movie is deleted giving the admin an option to cancel the delete. The confirmDelete function delete the movie and removes it from the database, it then redirects the user back to the movie All page.

Fig (4.140) Handle Review

```

1 const handleAddReview = async () => {
2   if (!userId || !authenticated) {
3     setError("Please log in to add a review");
4     navigate("/");
5     return;
6   }
7   console.log("Token used for adding review:", token);
8   try {
9     const response = await createReview(movieId, reviewContent, token);
10    setReviewId(response._id);
11    setReviewContent("");
12  } catch (err) {
13    setError(err.response?.data?.error || "Failed to add review");
14    if (err.response?.status === 401) {
15      setError("Unauthorized: Please log in again.");
16    }
17  }
18 }
19
20 const handleEditReview = async (reviewId) => {
21   if (!userId || !authenticated) {
22     setError("Please log in to edit review");
23     return;
24   }
25   console.log("Token used for editing review:", token);
26   try {
27     const response = await updateReview(reviewId, reviewContent, token);
28     setReviewId(response._id);
29     setReviewContent("");
30   } catch (err) {
31     setError(err.response?.data?.error || "Failed to edit review");
32     if (err.response?.status === 401) {
33       setError("Unauthorized: Please log in again.");
34     }
35   }
36 }
37
38 const handleDeleteReview = (reviewId) => {
39   if (!userId || !authenticated) {
40     setError("Please log in to delete review");
41     return;
42   }
43   setReviewIdDelete(reviewId);
44   deleteReviewModalRef.current.showModal();
45 }
46
47 const confirmDeleteReview = async () => {
48   if (!reviewIdDelete) return;
49   console.log("Token used for deleting review:", token);
50   try {
51     await deleteReview(reviewIdDelete, token);
52     setReviews(reviews.filter(r => r._id !== reviewIdDelete));
53     setReviewIdDelete(null);
54     deleteReviewModalRef.current.close();
55   } catch (err) {
56     setError(err.response?.data?.error || "Failed to delete review");
57     if (err.response?.status === 401) {
58       setError("Unauthorized: Please log in again.");
59     }
60   }
61 }

```

- The handleAddReview function which give the user and admin the ability to add a review to the movie they are currently on if the user if authenticated, it is a POST request and is the API call from the review file in the apis folder. The next function is the handleEditReview which allows the user and admin to edit the review they made. In it defined in the backend that only the user that has the corresponding user id with the review can only edit that review removing the option of a different user being able to edit another user's reviews. The handleDeleteReview function gives the user and admin the ability to delete their reviews which removes them from the database and the movie.

Fig (4.141) Handle Actor

```

1  const handleRemoveActor = (actorId) => {
2    if (!userId || !authenticated || role !== "admin") {
3      setError("Only admins can remove actors from movies");
4      return;
5    }
6    setActorToRemove(actorId);
7    deleteActorModalRef.current.showModal();
8  };
9
10 const confirmRemoveActor = async () => {
11   if (!actorToRemove) return;
12   console.log("Token used for removing actor:", token);
13   try {
14     await removeActorFromMovie(movieId, actorToRemove, token);
15     setMovie({
16       ...movie,
17       actors: movie.actors.filter((actor) => actor.id !== actorToRemove
18     e),
19   });
20   setActorToRemove(null);
21   deleteActorModalRef.current.close();
22 } catch (err) {
23   setError(
24     err.response?.data?.error || "Failed to remove actor from movie"
25   );
26   if (err.response?.status === 401)
27     setError("Unauthorized: Please log in again.");
28 }
29 };

```

- The handleActorDelete is the function that lets the admin remove actors from the current movie. Before they can remove the actor, it checks the user's role to confirm they are the admin, then the confirmActorDelete deletes the actors instance in this movie and updates the state when the actor is removed, this delete is a soft delete keeping the actor in the database but removing its id from the movie.

Fig (4.142) Return Statement

```

1  return (
2    <div className="container mx-auto mt-10 px-4">
3      {error && <div className="alert alert-error mb-8">{error}</div>}
4      <MovieSingleCard
5        movie={movie}
6        watchlists={watchlists}
7        selectedWatchlistId={selectedWatchlistId}
8        setSelectedWatchlistId={setSelectedWatchlistId}
9        handleAddToWatchlist={handleAddToWatchlist}
10       handleRate={handleRate}
11       handleDelete={handleDelete}
12       userId={userId}
13       authenticated={authenticated}
14       role={role}
15       movieId={movieId}
16     />
17
18     <renderActors() />
19
20     <div className="mt-8">
21       <h3 className="text-xl mb-4">Reviews</h3>
22       <textarea
23         value={reviewContent}
24         onChange={e => setReviewContent(e.target.value)}
25         placeholder="Write your review..."
26         className="p-2 border rounded w-full mb-2"
27         rows="3"
28       />
29       <button
30         onClick={
31           editingReviewId
32             ? () => handleEditReview(editingReviewId)
33             : handleAddReview
34         }
35         className="bg-green-500 text-white p-2 rounded"
36       >
37         {editingReviewId ? "Update Review" : "Add Review"}
38       </button>
39       {reviews.length === 0 ? (
40         <p>No reviews yet.</p>
41       ) : (
42         <div className="mt-4 space-y-4">
43           {reviews.map((review) => (
44             <div
45               key={review.id}
46               className="p-4 bg-gray-100 rounded shadow flex flex-col"
47             >

```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
<div className="flex justify-between items-center">
  <p className="font-semibold">{review.username}</p>
  <p className="text-sm text-gray-500">
    {new Date(review.created_at).toLocaleDateString()}
  </p>
</div>
<p className="mt-2">{review.content}</p>
{review.user_id === parseInt(userId) && (
  <div className="mt-2 flex space-x-2">
    <button
      onClick={() => {
        setEditingReviewId(review.id);
        setReviewContent(review.content);
      }}
      className="bg-yellow-500 text-white p-1 rounded"
    >
      Edit
    </button>
    <button
      onClick={() => handleDeleteReview(review.id)}
      className="bg-red-500 text-white p-1 rounded"
    >
      Delete
    </button>
  </div>
)}
</div>
)}}
</div>
))
</div>
))
</div>

```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
```

- The watchlists, public watchlists, and actors have their own single page where the id is extracted to get the correct data for the id. These pages are all like the movie single page apart from using different APIs and some divergent functions.

## Movies create

Fig (4.143) Movie Create Function & States

```

1  import React, { useState, useEffect } from "react";
2  import { useNavigate, Link } from "react-router-dom";
3  import { fetchCurrentUser, createMovie } from "../apis/movie";
4  import { fetchAllActors } from "../apis/actor";
5
6  function MovieCreate({ authenticated }) {
7    const [formData, setFormData] = useState({
8      id: "",
9      movie_title: "",
10     movie_genres: "",
11     description: "",
12     actor_id: "",
13     image: "bloodborne1.jpg",
14   });
15   const [actors, setActors] = useState([]);
16   const [error, setError] = useState("");
17   const [isAdmin, setIsAdmin] = useState(false);
18   const [loading, setLoading] = useState(true);
19   const navigate = useNavigate();
20   const token = localStorage.getItem("token");

```

- The states are then initialized including formData that stores the input values for the movie to be created which have the movie\_title, description, movie\_genres, and the id, actors is an array that stores the actors from the API. The error state stores error messages when operation fail, isAdmin is a Boolean and it tracks to see if the current user is set as an admin, the loading state is used to display a loading message if the data is still loading. Navigate is defined for the navigation and the token is gotten from local storage.

Fig (4.144) useEffect for user authentication

```

1  useEffect(() => {
2    if (!token || !authenticated) {
3      setError("Please log in to continue");
4      setLoading(false);
5      navigate("/");
6      return;
7    }
8
9    const checkUserRoleAndFetchActors = async () => {
10     try {
11       const userData = await fetchCurrentUser(token);
12       setIsAdmin(userData.role === "admin");
13
14       const actorData = await fetchAllActors(token);
15       setActors(actorData);
16     } catch (err) {
17       console.error("Failed to fetch data:", err.response?.data);
18       setError("Please log in to continue");
19       setIsAdmin(false);
20     } finally {
21       setLoading(false);
22     }
23   };
24   checkUserRoleAndFetchActors();
25 }, [navigate, authenticated, token]);

```

- The useEffect gets the data, it is checking to see if the user is authenticated and if they are not an error occurs, and they are redirected to home. If the user does have a token, it gets

the user's role to see if they are an admin and then gets all the actors from the API call in movie and actor. if the operation fails an error occurs and isAdmin is set to false.

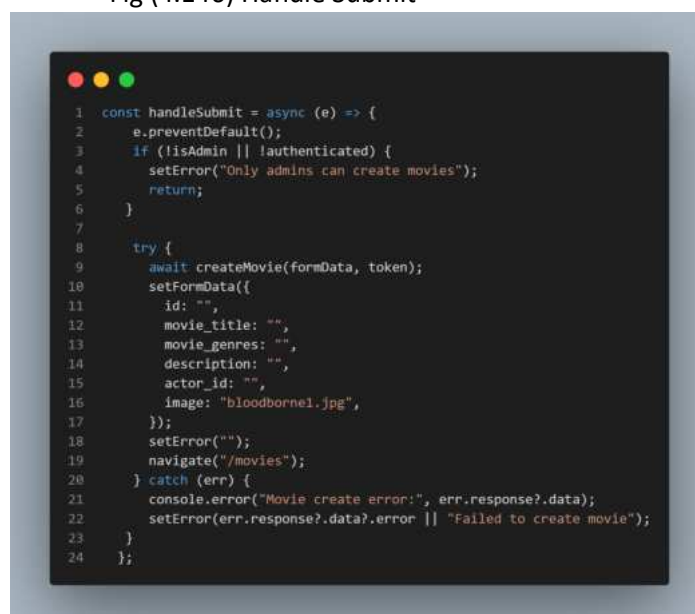
Fig (4.145) Handle Change

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a JavaScript function named handleChange. The code is as follows:

```
1 const handleChange = (e) => {  
2   setFormData((prev) => ({  
3     ...prev,  
4     [e.target.name]: e.target.value,  
5   }));  
6 };
```

- The handleChange function handles the changes in the form inputs, when an admin is inputting the fields for the movies it updates the value in the formData updating the state.

Fig (4.146) Handle Submit

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a JavaScript function named handleSubmit. The code is as follows:

```
1 const handleSubmit = async (e) => {  
2   e.preventDefault();  
3   if (!isAdmin || !authenticated) {  
4     setError("Only admins can create movies");  
5     return;  
6   }  
7  
8   try {  
9     await createMovie(formData, token);  
10    setFormData({  
11      id: "",  
12      movie_title: "",  
13      movie_genres: "",  
14      description: "",  
15      actor_id: "",  
16      image: "bloodborne1.jpg",  
17    });  
18    setError("");  
19    navigate("/movies");  
20  } catch (err) {  
21    console.error("Movie create error:", err.response?.data);  
22    setError(err.response?.data?.error || "Failed to create movie");  
23  }  
24 };
```

- The handleSubmit function is run when the admin tries to submit the create form, it checks to see if the user is an admin, if not an error will occur, if successful the movie can be made, it calls the API call from the apis folder using the movie file which stores the POST request for the movies, when the movie has been made the form is reset and the admin is navigated to the movies all page. If the create fails an error will appear.

Fig (4.147) Return Statement

```

1  return (
2    <div className="container mx-auto mt-10 flex flex-col items-center space-y-1
3  2"> <h2 className="text-4xl">Create New Movie</h2>
4    <error && (
5      <p style={errStyle} className="text-center">
6        {error}
7      </p>
8    )
9    {!isAdmin || !authenticated ? (
10     <p style={errStyle} className="text-center max-w-md">
11       You must be an admin to create movies. Please log in with an admin
12       account.
13     </p>
14   ) : (
15     <form
16       onSubmit={handleSubmit}
17       className="flex flex-col items-center space-y-6"
18     >
19       <input
20         type="text"
21         name="id"
22         placeholder="Movie ID (e.g., tt1234567)"
23         value={formData.id}
24         onChange={handleChange}
25         className="input input-bordered w-full max-w-xs"
26         required
27       />
28       <input
29         type="text"
30         name="movie_title"
31         placeholder="Movie Title"
32         value={formData.movie_title}
33         onChange={handleChange}
34         className="input input-bordered w-full max-w-xs"
35         required
36       />
37       <input
38         type="text"
39         name="movie_genres"
40         placeholder="Genres (comma-separated)"
41         value={formData.movie_genres}
42         onChange={handleChange}
43         className="input input-bordered w-full max-w-xs"
44         required
45       />

```

```

1  <textarea
2    name="description"
3    placeholder="Description (optional)"
4    value={formData.description}
5    onChange={handleChange}
6    className="input input-bordered w-full max-w-xs h-24"
7  />
8  <select
9    name="actor_id"
10   value={formData.actor_id}
11   onChange={handleChange}
12   className="select select-bordered w-full max-w-xs"
13   required
14 >
15   <option value="">Select an Actor</option>
16   {actors.map((actor) => (
17     <option key={actor.id} value={actor.id}>
18       {actor.name}
19     </option>
20   ))}
21 </select>
22 <input type="hidden" name="image" value={formData.image} /
23 > <button type="submit" className="btn btn-active">
24   Create Movie
25 </button>
26 </form>
27 ))
28 <Link to="/movies" className="underline text-blue-500">
29   Back to Movies
30 </Link>
31 </div>
32 );
33 }
34
35 export default MovieCreate;
36

```

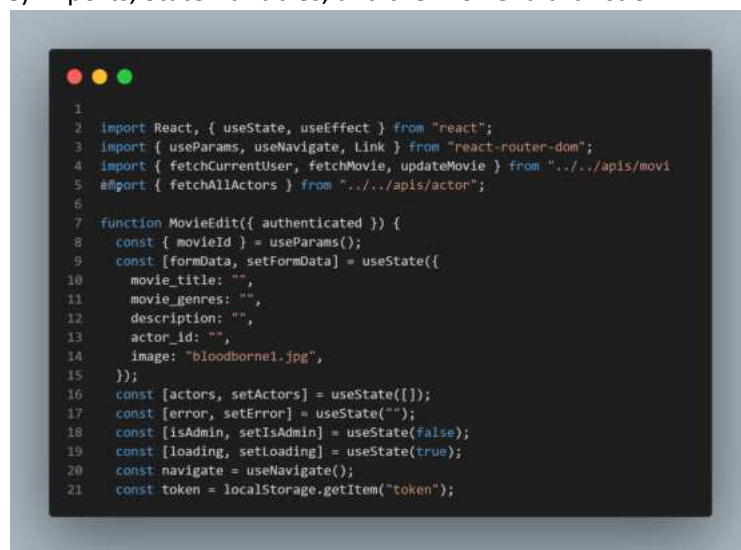


- In the return it checks if the user is an admin, if they are not an error message displays state the user is unauthorized, if they are an admin, it then renders the form for the movie creation displaying the input fields, movie\_id, movie\_title, description, actor\_id as a dropdown. If there are no actors, the dropdown appears empty. The submit button is rendered and links back to the movie all page when submitted calling the handleSubmit function to create the movie.
- The watchlist and actors also have these pages to create a new watchlist or actor. Similarly, the ratings and review also have a POST but that works alongside the single page for the movies, and the logic lies in that file similarly to the edit in the home page for the ratings.

## Movie edit

- The edit movie page allows the admin to edit movies, the hooks, useParams and useNavigate, movie and actor APIs are imported for use in this page.

Fig (4.148) Imports, State Variables, and the MovieEdit function



- The MovieEdit function is then declared which takes in the authenticated as a prop, and the state variables are defined, these state variables include the useParams extracting the movie\_id from the URL to get the specific movie that needs to be edited, the formData is used to store the movie data that the admin wants to update, those input fields being the movie\_title, movie\_genres, description, and actor\_id. The actor state stores the actor data, the error stores all the error messages, the isAdmin tracks if the current user is an admin or not. Loading is set to true when the data is still rendering, is set to false when data comes in, navigate is initialized for navigation and the token from the current user is gotten from local storage.

Fig (4.149) useEffect for authenticated admins

```
1
2  useEffect(() => {
3    if (!token || !authenticated) {
4      setError("Please log in to continue");
5      setLoading(false);
6      navigate("/");
7      return;
8    }
9
10   if (!movieId) {
11     setError("No movie ID provided");
12     setLoading(false);
13     navigate("/movies");
14     return;
15   }
16
17   const checkUserRoleAndFetchData = async () => {
18     try {
19       const userData = await fetchCurrentUser(token);
20       setIsAdmin(userData.role === "admin");
21
22       const movieData = await fetchMovie(movieId, token);
23       setFormData({
24         movie_title: movieData.movie_title,
25         movie_genres: movieData.movie_genres,
26         description: movieData.description || "",
27         actor_id: movieData.actors[0]?.id || "",
28         image: movieData.image_url
29           ? movieData.image_url.split("/").pop()
30           : "bloodborne1.jpg",
31       });
32
33       const actorData = await fetchAllActors(token);
34       setActors(actorData);
35     } catch (err) {
36       console.error("Error:", err.response?.data);
37       setError(err.response?.data?.error || "Failed to load movie data");
38       setIsAdmin(false);
39     } finally {
40       setLoading(false);
41     }
42   };
43
44   checkUserRoleAndFetchData();
45 }, [movieId, navigate, authenticated, token]);
```

- The useEffect checks if the admin is authenticated, if not an error will occur, if they are it see if the movie id exists for the movie that needs to be updated if it does not the user is redirected to movie all. If successful it gets the admins data and checks if they are an admin, if they are it gets the movie data for the form and displays it, if this fails an error occurs and loading is set to false.


Fig (4.150) Handle Change

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript function named `handleChange` defined as a constant. The function takes an event `e` as an argument and calls `setFormData` with a new object. This object spreads the previous `formData` state and updates the value for the field specified by `e.target.name` to `e.target.value`.

```
1  const handleChange = (e) => {  
2    setFormData((prev) => ({  
3      ...prev,  
4      [e.target.name]: e.target.value,  
5    }));  
6  };
```

- The function `handleChange` which updates the `formData` state when one of the fields have been modified, it edits the value based on the field that was changed.

Fig (4.151) Handle Submit

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains an asynchronous JavaScript function named `handleSubmit`. The function first calls `e.preventDefault()` to prevent the default form submission. It then checks if the user is an admin; if not, it sets an error and returns. If the user is an admin, it constructs a `data` object from the `formData` state, including fields like `movie_title`, `movie_genres`, `description`, `image`, and `actor_id`. It then attempts to call `updateMovie` with the movie ID, the constructed data, and a token. If the update is successful, it clears the error and navigates to the movies page. If it fails, it logs the error to the console and sets a specific error message.

```
1  const handleSubmit = async (e) => {  
2    e.preventDefault();  
3    if (!isAdmin || !authenticated) {  
4      setError("Only admins can edit movies");  
5      return;  
6    }  
7  
8    const data = {  
9      movie_title: formData.movie_title,  
10     movie_genres: formData.movie_genres,  
11     description: formData.description,  
12     image: formData.image,  
13   };  
14   if (formData.actor_id) {  
15     data.actor_id = formData.actor_id;  
16   }  
17  
18   try {  
19     await updateMovie(movieId, data, token);  
20     setError("");  
21     navigate("/movies");  
22   } catch (err) {  
23     console.error("Update error:", err.response?.data);  
24     setError(err.response?.data?.error || "Failed to update movie");  
25   }  
26 };
```

- The function `handleSubmit` checks to see if the user is an admin if they are it updates the movie using the function `updateMovie`, when the operation succeeds it updates the movie, if it fails an error will occur.

Fig (4.152) Return Statement

```

1  return (
2    <div className="container mx-auto mt-10 flex flex-col items-center space-y-1
3    2"> <h2 className="text-4xl">Edit Movie</h2>
4    (error && {
5      <p style={errStyle} className="text-center">
6        (error)
7      </p>
8    })
9    {isAdmin || !authenticated} ? {
10     <p style={errStyle} className="text-center max-w-md">
11       You must be an admin to edit movies. Please log in with an admin
12       account.
13     </p>
14   } : {
15     <form
16       onSubmit={handleSubmit}
17       className="flex flex-col items-center space-y-8"
18     >
19       <input
20         type="text"
21         name="movie_title"
22         placeholder="Movie Title"
23         value={formData.movie_title}
24         onChange={handleChange}
25         className="input input-bordered w-full max-w-xs"
26         required
27       />
28       <input
29         type="text"
30         name="movie_genres"
31         placeholder="Genres (comma-separated)"
32         value={formData.movie_genres}
33         onChange={handleChange}
34         className="input input-bordered w-full max-w-xs"
35         required
36       />
37       <textarea
38         name="description"
39         placeholder="Description (optional)"
40         value={formData.description}
41         onChange={handleChange}
42         className="input input-bordered w-full max-w-xs h-24"
43       />
44       <select
45         name="actor_id"
46         value={formData.actor_id}
47         onChange={handleChange}
48         className="select select-bordered w-full max-w-xs"
49       >
50         <option value="">Select an Actor (optional)</option>
51         {actors.map(actor) => (
52           <option key={actor.id} value={actor.id}>
53             {actor.name}
54           </option>
55         )}
56       </select>

```

```

1  <input type="hidden" name="image" value={formData.image} />
2    <button type="submit" className="btn btn-active">
3      Update Movie
4    </button>
5  </form>
6  )}
7  <Link to="/movies" className="underline text-blue-50
8  0"> Back to Movies
9  </Link>
10 </div>
11 );
12 }
13
14 export default MovieEdit;
15

```

- In the render it checks if any errors are present, if so it displays an error message, if not and everything is loaded, it checks to see if the admin is set as an admin and if so it renders the form with the fields movie\_title, movie\_genres, description, and actor\_id display.
- The watchlists, reviews, and ratings all have a POST method, the watchlists itself has its own create page while the reviews and ratings POST are performed in other files such as home and movie single page.

Fig (4.153) PageNotFound



```
1
2 const PageNotFound = () => {
3   return (
4     <>
5     <h1 className="text-lg ml-3 mt-
6 3">
7       <b>404, Page not found</b>
8     </h1>
9   </>
10  );
11 };
12 export default PageNotFound;
13
```

- The page not found page is for if the user types in a URL that does not exist.
- The PageNotFound takes in no props.
- It renders a message in the return telling the user this page is not found with a 404 error.
- It is then exported to be rendered in App.js.

## 4.9 Sprint 6

Testing the CRUD functionalities of the application involved downloading a library called Pytest, Pytest is a framework for Python that allows for tests to be run on the different modules in the application.

Functional and user testing were done to ensure that the CRUD functions involving GET, PUT, POST, and DELETE worked as expected. The user testing involved setting tasks that a user would have to complete without assistance from the developer to see how the experience went while using the application. The user's feedback was then collected which outlined the pains the user experienced, this helped to find and fix issues from a user's perspective.

During this phase improvements to the project were made such as adding images from the backend to the movies so that they could be displayed in the front-end. The MovieSingleCard was created and moved the watchlist, rating, edit, and delete to this file. The same process was done with the actor all page and the new ActorCard.

### 4.9.1 Item 1

- Functional Testing

Fig (4.154) Test Config

```
1 class TestConfig:
2     TESTING = True
3     SQLALCHEMY_DATABASE_URI = 'sqlite:///memory:'
4     SQLALCHEMY_TRACK_MODIFICATIONS = False
5     JWT_SECRET_KEY = 'test-secret-key'
6     JWT_TOKEN_LOCATION = ['headers']
7     JWT_COOKIE_CSRF_PROTECT = False
8     JWT_ACCESS_TOKEN_EXPIRES = 3600
9
10 @pytest.fixture
11 def app():
12     app = create_app(TestConfig)
13     with app.app_context():
14         db.create_all()
15         yield app
16         db.drop_all()
```

The TestConfig class is the configuration file that allows for tests to be run, with in-memory database it can run these tests without the resulting data going into the actual database. Here it sets up the JWT settings for authentication.

The TestConfig over all is creating an app and opens the app context so the temporary database can be initialized. It then creates all the tables for the data and then the app yields so the tests can be conducted. After the tests, the tables are dropped making it so that when future tests are run there will not be any persisting data.

Fig (4.155) User & Admin

```

1  @pytest.fixture
2  def init_database(app):
3      with app.app_context():
4          user = User(
5              id=1,
6              username="testuser",
7              email="testuser@example.com",
8              password="testpass",
9              user_gender=1,
10             user_occupation_label=1,
11             raw_user_age=30,
12             user_rating=4.0,
13             role="user"
14         )

```

```

1  admin = User(
2      id=2,
3      username="adminuser",
4      email="adminuser@example.co
5  m", password="adminpass",
6      user_gender=1,
7      user_occupation_label=1,
8      raw_user_age=40,
9      user_rating=4.5,
10     role="admin"
11 )
12 db.session.add_all([user, admin])

```

This is the setup for the two users being created for the tests, the `@pytest.fixture` `def init_database(app)` is setting up the testing database with hardcoded data before a test can be conducted. Inside there are two users the normal user and the admin with their roles assigned, this is important for authentication-based routes. The users are then added to the session.

This now give the temporary database with user data that will be used in other tests. The other setups for the movies, actors, ratings, rankings, reviews, and watchlists follow the same structure for their set ups.

Fig (4.156)

```
1 db.session.commit()
2 yield db
3
4 @pytest.fixture
5 def user_token(app):
6     with app.app_context():
7         return create_access_token(identity="1")
8
9 @pytest.fixture
10 def admin_token(app):
11     with app.app_context():
12         return create_access_token(identity="2")
13
14 @pytest.fixture
15 def mock_ranking_model():
16     mock_model = MagicMock()
17     mock_model.return_value = MagicMock(numpy=lambda: np.array([0.9, 0.8, 0.7, 0.6, 0.5]))
18     return mock_model
```

The DB yield runs after the data for all the modules has been added, the yield then allows test to be done on the temporary database and when this is completed it can then drop the tables after this.

The @pytest.fixture def user\_token(app) makes a token for the user with the id of 1. this was done to test authentication.

@pytest.fixture def admin\_token(app) is like the user\_token except this authentication is for admin users as they have different abilities on the application.

@pytest.fixture def mock\_ranking\_model(app) this creates a mock model. It does this using the library from Python, MagicMock, this was done to simulate the models process of giving the rankings.

Fig (4.157) GET all movies



```

1  @pytest.mark.usefixtures("init_database")
2  def test_get_movies_success(client, user_token):
3      headers = {"Authorization": f"Bearer {user_token}"}
4      response = client.get("/movies", headers=headers)
5      data = json.loads(response.data)
6
7      assert response.status_code == 200
8      assert isinstance(data, list)
9      assert len(data) == 1
10     assert data[0]["id"] == "tt0000001"
11     assert data[0]["movie_title"] == "Test Movie"
12     assert data[0]["movie_genres"] == "Action"
13     assert data[0]["actors"][0]["id"] == 1
14
15     @pytest.mark.usefixtures("init_database")
16     def test_get_movies_unauthorized(client):
17         response = client.get("/movies")
18         data = json.loads(response.data)
19
20         assert response.status_code == 401
21         assert "msg" in data

```

- Testing for all movies involved two tests, `test_get_movies_success` checks if the authentication and then gets the list of movies with the testing data. The `test_get_movie_unauthorized` tests if the user can still get movies with a token that is not authorized.

Fig (4.158) GET ID Movie Test


```

1  @pytest.mark.usefixtures("init_database")
2  def test_get_single_movie_success(client, user_token):
3      headers = {"Authorization": f"Bearer {user_token}"}
4      response = client.get("/movies/tt0000001", headers=headers)
5      data = json.loads(response.data)
6
7      assert response.status_code == 200
8      assert data["id"] == "tt0000001"
9      assert data["movie_title"] == "Test Movie"
10     assert data["actors"][0]["id"] == 1
11
12     @pytest.mark.usefixtures("init_database")
13     def test_get_single_movie_not_found(client, user_token):
14         headers = {"Authorization": f"Bearer {user_token}"}
15         response = client.get("/movies/tt9999999", headers=headers)
16
17         assert response.status_code == 404

```

- The tests for getting a single movie with a specific id. The test\_get\_single\_movies\_success gets the specific movie with an authenticated id. The test\_get\_single\_movie\_unauthorized tests if the user can get the movie if not authenticated.

Fig (4.159) CREATE Movie Test



```
1 @pytest.mark.usefixtures("init_database")
2 def test_create_movie_success(client, admin_token):
3     headers = {"Authorization": f"Bearer {admin_token}"}
4     data = {
5         "id": "tt0000002",
6         "movie_title": "New Movie",
7         "movie_genres": "Comedy",
8         "description": "A funny movie",
9         "actor_id": 2,
10        "image": "bloodborne1.jpg"
11    }
12    with patch("os.path.isfile", return_value=True):
13        response = client.post("/movies/create", json=data, headers=headers)
14        data = json.loads(response.data)
15
16    assert response.status_code == 201
17    assert data["movie_title"] == "New Movie"
18    assert data["message"] == "Movie created successfully"
19    assert data["actors"][0]["id"] == 2
20
21 @pytest.mark.usefixtures("init_database")
22 def test_create_movie_unauthorized(client, user_token):
23     headers = {"Authorization": f"Bearer {user_token}"}
24     data = {
25         "id": "tt0000002",
26         "movie_title": "New Movie",
27         "movie_genres": "Comedy",
28         "actor_id": 2,
29         "image": "bloodborne1.jpg"
30     }
31     response = client.post("/movies/create", json=data, headers=headers)
32     data = json.loads(response.data)
33
34     assert response.status_code == 403
35     assert data["error"] == "Admin access required"
```

- This is the test for the movie create and tests if the user is an admin and this user can successfully create a movie.

Fig (4.160) PUT Movie Test

```

1 @pytest.mark.usefixtures("init_database")
2 def test_update_movie_success(client, admin_token):
3     headers = {"Authorization": f"Bearer {admin_token}"}
4     data = {
5         "movie_title": "Updated Movie",
6         "movie_genres": "Drama",
7         "description": "An updated movie",
8         "image": "bloodborne1.jpg"
9     }
10    with patch("os.path.isfile", return_value=True):
11        response = client.put("/movies/update/tt0000001", json=data, headers=headers)
12        data = json.loads(response.data)
13
14    assert response.status_code == 200
15    assert data["movie_title"] == "Updated Movie"
16    assert data["message"] == "Movie updated successfully"
17

```

- This is the test for the movie edit and tests if the user is an admin and this user can successfully edit the movie with the specific id.

Fig (4.161) DELETE Movie Test

```

1 @pytest.mark.usefixtures("init_database")
2 def test_delete_movie_success(client, admin_token):
3     headers = {"Authorization": f"Bearer {admin_token}"}
4     response = client.delete("/movies/delete/tt0000001", headers=headers)
5     data = json.loads(response.data)
6
7     assert response.status_code == 200
8     assert "message" in data
9     assert "Movie and associated ratings, reviews, and watchlist entries deleted successfully" in data["message"]

```

- This test is for the movie DELETE that checks if the user is an admin, if they are the movie is successfully delete using the movies id to get the specific movie.

Fig (4.162) Add Actor

```

1 @pytest.mark.usefixtures("init_database")
2 def test_add_actor_success(client, admin_token):
3     headers = {"Authorization": f"Bearer {admin_token}"}
4     data = {"actor_id": 2}
5     response = client.post("/movies/tt0000001/actors", json=data, headers=headers)
6     data = json.loads(response.data)
7
8     assert response.status_code == 200
9     assert "message" in data
10    assert "Second Actor added to movie Test Movie" in data["message"]

```

- This test is testing the to see if an actor can be added to a movie, it checks to see if the user is an admin and the actor is then successfully added to the movie.

Fig (4.163) Remove actor

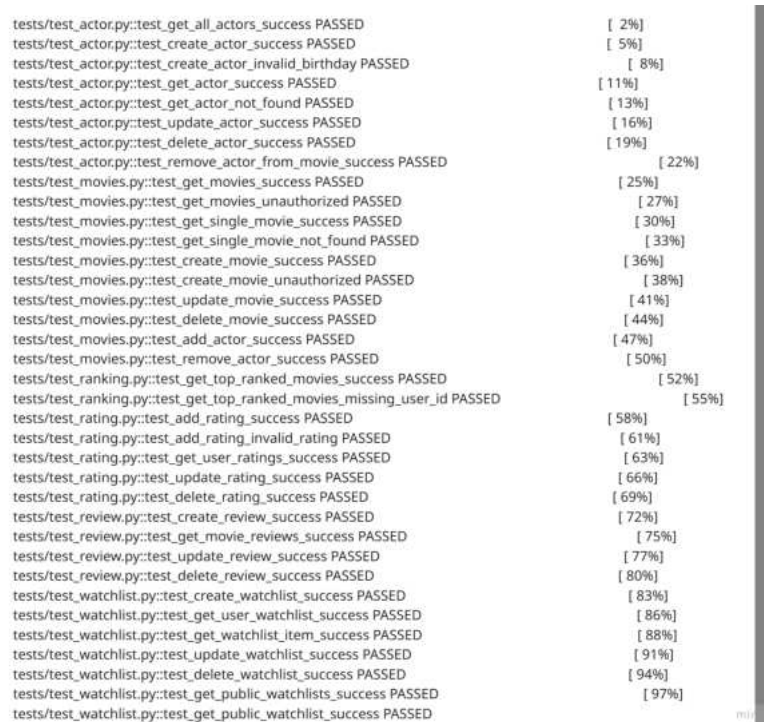
```

1 @pytest.mark.usefixtures("init_database")
2 def test_remove_actor_success(client, admin_token):
3     headers = {"Authorization": f"Bearer {admin_token}"}
4     response = client.delete("/movies/tt0000001/actors/1", headers=headers)
5     data = json.loads(response.data)
6
7     assert response.status_code == 200
8     assert "message" in data
9     assert "Test Actor deleted from database" in data["message"]

```

- This test is testing the removal of the actor for the movies. It checks the user to see if they are an admin and then it successfully removes the actor from the movie.

Fig (4.164)



Tests were done on all the modules in the application including for Actors, Watchlists, Ratings, Reviews, and Rankings. The results show all functions passing.

#### 4.9.2 Item 2

- User Testing  
Fig (4.165) Tasks

1. Register an account on the application.
2. Once on the home page go to the watchlists page.
3. Create a watchlist.
4. navigate to the movie page and select a movie.
5. From the selected movie choose your watchlist and add the movie to it.
6. Add a rating to the movie.
7. Leave a review on the movie.
8. Navigate back to the watchlists and select your watchlist.
9. From your selected watchlist make it public, and edit the title of the watchlist.
10. navigate to the public watchlist page and find your watchlist.
11. View your public watchlist.
12. navigate to the home page and see the movie that have been recommended.
13. From home edit the rated movie and then remove it.
14. Navigate to the profile page and log out.
15. Using these credentials, email: admin@moviemuze.com, password: adminpassword.
16. Once logged in as the admin, navigate back to the movie page.
17. Create a movie.
18. Use the search bar to find your movie.
19. Select the movie and then edit the title and actor.
20. Find your movie again and delete it.
21. Try and find it again using search.
22. Navigate to the actor page.
23. Assign an actor to a the first movie.
24. Edit the actors name.
25. Go to the movie that the actor was added to and remove it.
26. Delete the actor.
27. Navigate back to the profile page and sign out.

- The tasks were done by the users to tests the navigation and the usability of the application; The users would have to complete all the tasks and give feedback while interacting with the app.
- The tasks that the users had to do included the following, the tasks for users were:
- Registering an account then going to the watchlist page to make a watchlist.
- The user would then navigate to the movie page where the user will select a movie and add it to their watchlist. They will also leave a rating and review on the movie.
- The user will then make their way back to the watchlist page where they will edit the watchlists title and make the watchlist public. They will then navigate to the home page to view their recommended movies. They will edit a rating they have given and then delete that rating. The user will then log out of the account and log into the admin user.
- From the log in the admin will go and create a movie, they will then use the search to find the movie, edit it, and then delete it.
- The admin then navigates to the actor page where the admin will edit an actor, delete an actor, and assign one to a movie.
- The admin will then go to that movie and remove the actor.
- The user will then log out of the account.

#### 4.9.3 Item 3

- Code Additions

Fig (4.166) Images in movie model

```
1 from extensions import db
2 from models.actor import Actor
3 from models.actor_movie import movie_actors
4
5 class Movie(db.Model):
6     id = db.Column(db.String(10), primary_key=True)
7     movie_title = db.Column(db.String(100), nullable=False)
8     movie_genres = db.Column(db.String(100), nullable=False)
9     description = db.Column(db.String(500), nullable=True)
10    image_url = db.Column(db.String(200), nullable=True)
11    created_at = db.Column(db.DateTime, default=db.func.current_timestamp())
12    actors = db.relationship('Actor', secondary=movie_actors, back_populates='movies', lazy='dynamic')
13
14    def __repr__(self):
15        return f"Movie(id='{self.id}', title='{self.movie_title}', genres='{self.movie_genres}')
```

- The images involved making a new attribute in the model to hold the images for the movies.

Fig (4.167) Image\_url in movie route

```
1 @movie_bp.route('/create', methods=['POST'], endpoint='create_movie')
2 @admin_required
3 def create_movie():
4     logging.debug(f"Create movie request: {request.json}")
5     if not request.is_json:
6         return jsonify({'error': 'Content-Type must be application/json'}), 400
7
8     data = request.json
9     if not data or 'id' not in data or 'movie_title' not in data or 'movie_genres' not in data or 'actor_id' not in data or 'image' not in data:
10         return jsonify({'error': 'Missing id, movie_title, movie_genres, actor_id, or image'}), 400
11
12     if Movie.query.get(data['id']):
13         return jsonify({'error': f'Movie with ID {data["id"]} already exists'}), 409
14
15     actor = Actor.query.get(data['actor_id'])
16     if not actor:
17         return jsonify({'error': f'Actor with ID {data["actor_id"]} not found'}), 404
18
19     image_filename = data['image']
20     if image_filename not in AVAILABLE_IMAGES:
21         return jsonify({'error': f'Image must be one of {AVAILABLE_IMAGES}'}), 400
22     image_path = os.path.join(IMAGE_FOLDER, image_filename)
23     if not os.path.isfile(image_path):
24         return jsonify({'error': f'Image {image_filename} not found in {IMAGE_FOLDER}'}), 404
25     image_url = f"movies/{image_filename}"
26
27     try:
28         new_movie = Movie(
29             id=data['id'],
30             movie_title=data['movie_title'],
31             movie_genres=data['movie_genres'],
32             description=data.get('description'),
33             image_url=image_url
34         )
```



```

1 @movie_bp.route('/update/<id>', methods=['PUT'], endpoint='update_movie')
2 @admin_required
3 def update_movie(id):
4     logging.debug(f"Update movie ID: {id}, data: {request.json}")
5     if not request.is_json:
6         return jsonify({'error': 'Content-type must be application/json'}), 400
7
8     data = request.json
9     if not data or 'movie_title' not in data or 'movie_genres' not in data:
10         return jsonify({'error': 'Missing movie_title or movie_genres'}), 400
11
12     movie = Movie.query.get_or_404(id)
13     try:
14         movie.movie_title = data['movie_title']
15         movie.movie_genres = data['movie_genres']
16         movie.description = data.get('description', movie.description)
17         if 'image' in data:
18             image_filename = data['image']
19             if image_filename not in AVAILABLE_IMAGES:
20                 return jsonify({'error': f"Image must be one of {'', '.join(AVAILABLE_IMAGES)"}}), 400
21             image_path = os.path.join(IMAGE_FOLDER, image_filename)
22             if not os.path.isfile(image_path):
23                 return jsonify({'error': f"Image {image_filename} not found in {IMAGE_FOLDER}"}), 404
24             movie.image_url = f"/movies/{image_filename}"
25         if 'actor_id' in data:
26             actor = Actor.query.get(data['actor_id'])
27             if not actor:
28                 return jsonify({'error': f"Actor with ID {data['actor_id']} not found"}), 404
29             movie.actors.append(actor)
30     db.session.commit()
31     response = {
32         'message': 'Movie updated successfully',
33         'id': movie.id,
34         'movie_title': movie.movie_title,
35         'movie_genres': movie.movie_genres,
36         'description': movie.description,
37         'image_url': f"/static/{movie.image_url}" if movie.image_url else "/static/movies/bloodborne1.jp
38 g",
39         'created_at': movie.created_at.isoformat(),
40         'actors': [{'id': actor.id, 'name': actor.name} for actor in movie.actors.all()]
41     }
42     logging.debug(f"Update movie response: {response}")
43     return jsonify(response), 200
44 except Exception as e:
45     db.session.rollback()
46     logging.error(f"Update movie error: {str(e)}")
47     return jsonify({'error': f"Failed to update movie: {str(e)}"}), 500

```

- The image was then put into the create and edit for the movie route to create it. In the GET the image is also defined for the movies.

Fig (4.168) Images being seeded

```

1
2 image_folder = "static/movies"
3 os.makedirs(image_folder, exist_ok=True)
4 available_images = ['bloodborne1.jpg']
5 for img in available_images:
6     full_path = os.path.join(image_folder, img)
7     if not os.path.isfile(full_path):
8         print(f"Error: Image {img} not found in {image_folder}. Please add it before seeding.")
9     return
10 if not available_images:
11     print("Error: No images available in static/movies/. Please add images before seeding.")
12     return
13
14 for movie in dataset:
15     movie_id = movie["movie_id"].numpy().decode('utf-8')
16     movie_title = movie["movie_title"].numpy().decode('utf-8')
17     genre_ids = movie["movie_genres"].numpy().tolist()
18     valid_genre_ids = [gid for gid in genre_ids if gid in genre_map]
19     movie_genres = ", ".join(genre_map[gid] for gid in valid_genre_ids) or "Unknown"
20     description = fake.paragraph(nb_sentences=2)
21
22     existing_movie = Movie.query.filter_by(id=movie_id).first()
23     if not existing_movie:
24         selected_actors = random.sample(all_actors, 4)
25         selected_image = available_images[0]
26         image_url = f"movies/{selected_image}"
27
28         new_movie = Movie(
29             id=movie_id,
30             movie_title=movie_title,
31             movie_genres=movie_genres,
32             description=description,
33             image_url=image_url
34         )
35         new_movie.actors.extend(selected_actors)
36         db.session.add(new_movie)

```

- In the movie\_seeder, it defines the folder the image is stored in, that being the static/movies folder and creates it if it does not exist. The code looks for the bloodborne1.png image and assigns it to the movies that are being seeded and storing them in the image\_url attribute.

## MoveSingleCard

Fig (4.169) Imports and states

```

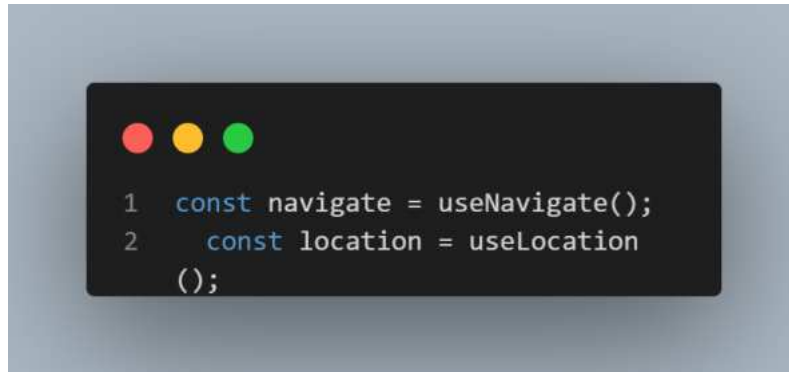
1 import React from "react";
2 import { useNavigate, useLocation } from "react-router-dom";
3
4 function MovieSingleCard({
5     movie,
6     watchlists,
7     selectedWatchlistId,
8     setSelectedWatchlistId,
9     handleAddToWatchlist,
10    handleRate,
11    handleDelete,
12    userId,
13    authenticated,
14    role,
15    movieId,
16 }) {
17     const navigate = useNavigate();
18     const location = useLocation();

```



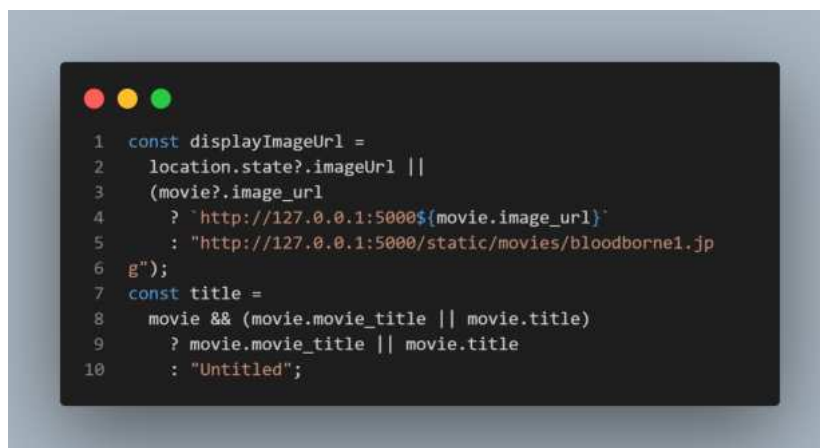
- In the MovieSingleCard the function takes in many props, this includes the movie details and functions for the adding of ratings, adding watchlists, and deleting and editing movies.

Fig (4.170) Hooks



- Using useNavigate and useLocation for navigation and to give access to the local state.

Fig (4.171) Images



- This displays the movie image and has a fallback to a default image if the movie image\_url cannot be accessed.

Fig (4.172) Return Statement

```

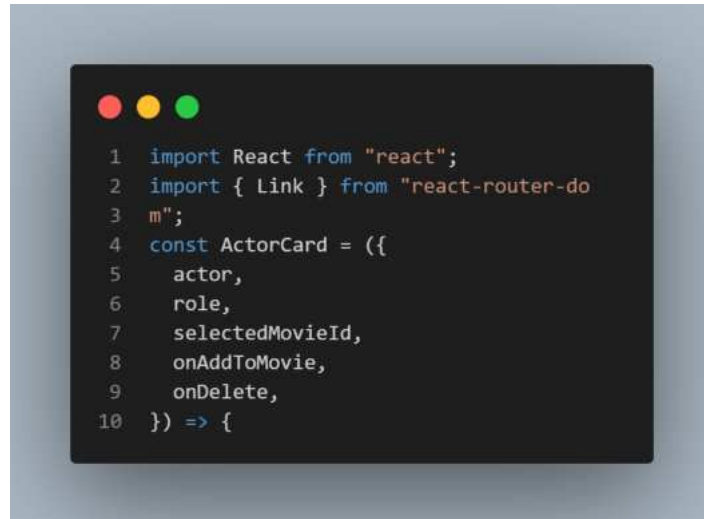
1  return (
2    <div className="container mx-auto p-6">
3      <div
4        className="hero min-h-[400px] shadow-xl mx-auto rounded-lg"
5        style={{
6          backgroundImage: `url(${displayImageUrl})`,
7          backgroundSize: 'cover',
8          backgroundPosition: 'center',
9        }}
10     >
11       <div className="hero-overlay bg-opacity-60"></div>
12       <div className="hero-content text-neutral-content text-center">
13         <div className="max-w-md">
14           <h2 className="mb-2 text-4xl font-bold">{title}</h2>
15           <p className="mb-2 text-sm">{movie.movie_genres || "No genres"}</p>
16           <p className="mb-4">{movie.description || "No description available"}</p>
17         </div>
18       </div>
19     </div>
20     <div className="max-w-lg mx-auto mt-4 flex justify-evenly gap-2">
21       <select
22         value={selectedWatchlistId}
23         onChange={(e) => setSelectedWatchlistId(e.target.value)}
24         className="select select-bordered max-w-xs text-base-content"
25       >
26         <option value="">Select Watchlist</option>
27         {watchlists.map(watchlist) => (
28           <option key={watchlist.id} value={watchlist.id}>
29             {watchlist.title}
30           </option>
31         )}
32       </select>
33       <button
34         onClick={handleAddToWatchlist}
35         className="btn btn-primary flex-1"
36       >
37         Add to Watchlist
38       </button>
39       <select
40         onChange={(e) => handleRate(parseFloat(e.target.value))}
41         className="select select-bordered btn btn-outline btn-primary flex-1 text-base-content"
42         defaultValue=""
43       >
44         <option value="" disabled>
45           Rate
46         </option>
47         {[1, 2, 3, 4, 5].map((r) => (
48           <option key={r} value={r}>
49             {r}
50           </option>
51         )}
52       </select>
53       {authenticated && role === "admin" && (
54         <button
55           onClick={() => navigate(`/movies/${movieId}/edit`)}
56           className="btn btn-warning flex-1"
57         >
58           Edit
59         </button>
60         <button onClick={handleDelete} className="btn btn-error flex-1">
61           Delete
62         </button>
63       )}
64     </div>
65   )}
66   {!(authenticated && role === "admin") && (
67     <div className="flex-1"></div>
68   )}
69 </div>
70 </div>
71 )}
72 }
73
74 export default MovieSingleCard;

```

- In the return it defines the select for the watchlist button, the rate movies button, and if the user is authenticated as an admin the edit and delete buttons will be visible. The functions defined in the movie single page such as handleRate, handleDelete, and handleAddToWatchlist are called using the imported functions.

ActorCard

Fig (4.173) Imports and states



- The ActorCard function has props that includes the actor data, the role of the user for authentication, movie id so that the actors can be assigned to a movie, the function to add an actor to a movie, and the delete function to delete an actor.

Fig (4.174) Return statement

```

1  return (
2    <div className="card bg-base-100 shadow-xl">
3      <div className="card-body">
4        <h2 className="card-title">
5          <Link to={` /actors/${actor.id}`} className="text-blue-500 underline">
6            {actor.name || "Unknown Actor"}
7          </Link>
8        </h2>
9        <p>
10         <strong>Description:</strong> {actor.description || "N/A"}
11       </p>
12       <p>
13         <strong>Previous Work:</strong> {actor.previous_work || "N/A"}
14       </p>
15       <p>
16         <strong>Birthday:</strong>{" "}
17         {actor.birthday
18           ? new Date(actor.birthday).toLocaleDateString()
19           : "N/A"}
20       </p>
21       <p>
22         <strong>Nationality:</strong> {actor.nationality || "N/A"}
23       </p>
24       <p>
25         <strong>Movies:</strong> {actor.movie_count || 0}
26       </p>
27       {role === "admin" && (
28         <div className="card-actions justify-end mt-2">
29           <button
30             onClick={() => onAddtoMovie(actor.id)}
31             className="btn btn-primary"
32             disabled={!selectedMovieId}>
33             Add to Movie
34           </button>
35           <Link to={` /actors/${actor.id}/edit`} className="btn btn-warning">
36             Edit
37           </Link>
38           <button
39             onClick={() => onDelete(actor.id)}
40             className="btn btn-error">
41             Delete
42           </button>
43         </div>
44       )}
45     </div>
46   );
47 </div>
48 </div>
49 );
50 };
51
52 export default ActorCard;
53

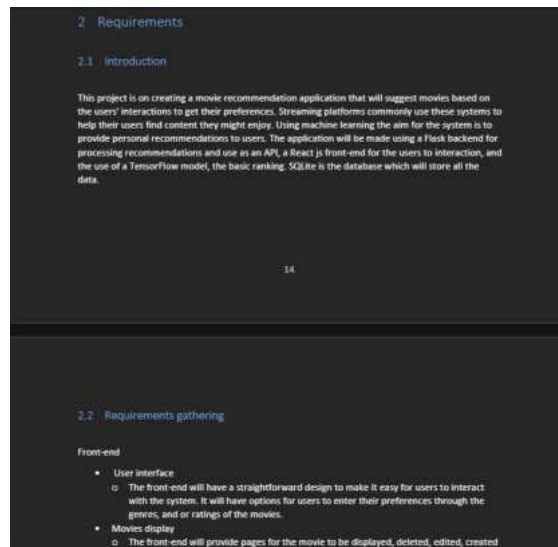
```

- In the return the actors name is linked to the actor single page to view the description of the actors, normal users have access to this page. The admin users will be able to add actors to movies using the button, they also can edit and delete the actors. These are functions imported from the actor index page those being the onAddMovie and onDelete.

## 4.10 Sprint 7

### 4.10.1 Item 1

- Pages  
Fig (4.175) Requirements



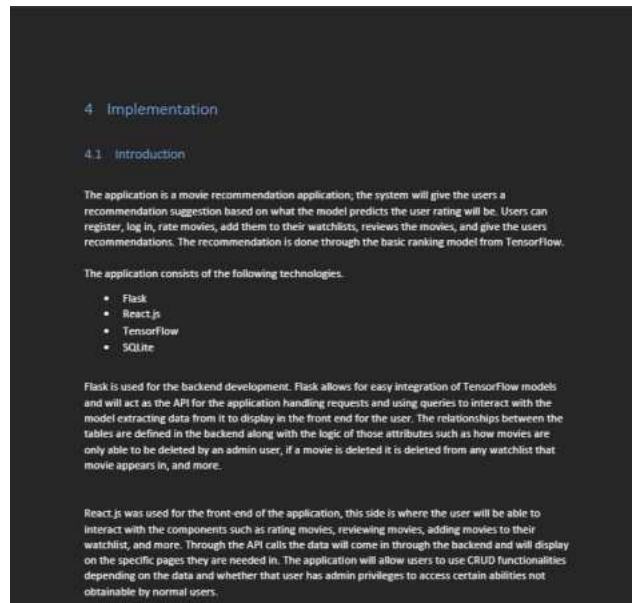
- The requirements chapter includes the talks that were done to get the functional and non-functional requirements for the application and to judge of this would be a feasible idea. This chapter includes a persona to represent a user, use case diagram to show how the app would be used, discussing the functional and non-functional requirements gained from all the completed tasks, and more.

Fig (4.176) Design



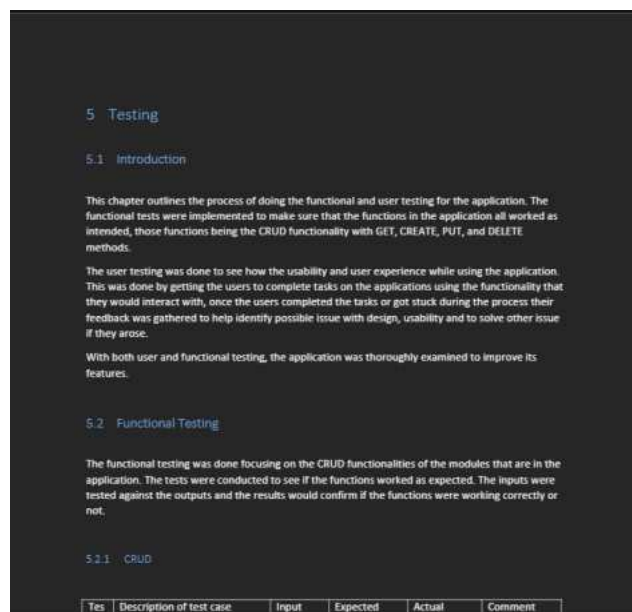
- The design chapter goes over the designs of the application, from the structure of the application to database designs. More specifically it shows an ERD, application architecture diagram, wireframe, style guide, and more aspects about the designs of the front and backends. It discusses how the model would be exported from google colab to be inserted into the Flask backed via zipped file.

Fig (4.177) Implementation



- The implementation chapter were completed in two-week sprints with there being seven in total, these sprints included: requirements, designs, implementations, designs 2, implementation 2, testing, Thesis.
- It went over the steps and the processes of creating the application from the initial requirements gathering, designing the initial designs for the app, implementing those designs, redesigning of necessary, further implementations, testing the application, and finally briefing giving an overview of the chapters.

Fig (4.178) Testing



- The testing chapter was completed before this section of the implementation chapter could be written, it goes over the functional and user testing that was conducted for the

application. It goes over the Unit testing for the CRUD functionality and used real users to test the navigation and usability. Results were mostly positive with improvements suggested by the test users.

#### 4.11 Conclusion

The implementation chapter goes over the initial beginning of creating the project to the definitive version. It shows the process of gathering requirements, outlining the designs, implementing the features from the designs, overhauling the designs and adding to the implantation, and then testing. It goes over the features that were and designs that were implemented. Unfortunately, due to the model not working as intended the application cannot reach it has intended potential but overall, the projects implementations went smoothly.

## 5 Testing

### 5.1 Introduction

This chapter outlines the process of doing the functional and user testing for the application. The functional tests were implemented to make sure that the functions in the application all worked as intended, those functions being the CRUD functionality with GET, CREATE, PUT, and DELETE methods.

The user testing was done to see how the usability and user experience while using the application. This was done by getting the users to complete tasks on the applications using the functionality that they would interact with, once the users completed the tasks or got stuck during the process their feedback was gathered to help identify possible issue with design, usability and to solve other issue if they arose.

With both user and functional testing, the application was thoroughly examined to improve its features.

### 5.2 Functional Testing

The functional testing was done focusing on the CRUD functionalities of the modules that are in the application. The tests were conducted to see if the functions worked as expected. The inputs were tested against the outputs and the results would confirm if the functions were working correctly or not.

#### 5.2.1 CRUD

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Create a movie.	POSTing movie with validation and data.	Movie created message should appear and the movie should be added to the database.	Movie was created and added to the database.	The movie Create passed creating the movie and adding it to the database.
2	Getting all the movies	Using the GET method to fetch	Movies were found displaying the list of	All the movies were fetched along with the movies	The movies were displayed in a



		all the movie data in the database with validation.	movies in the database.	corresponding data.	list passing the test.
3	Getting a single movie.	Using the GET method to get data for a single movie with validation and data from the database.	The specific movie was found displaying that movies data.	The single movies data was found getting the data for that movie.	The single movie route passed and received the data for the specific movie.
4	Editing a movie.	Using PUT method with token and altered data with admin validation.	The specific movie being updated successfully updates the data for the movie and changes the data in the database if they are an admin user.	The movie was edited and updated as expected changing the data for the movie and updating the movie in the database.	The movie was successfully updated altering the data for the specific movie. This was done by the admin user as the normal users do not have authorization.
5	Deleting movie.	Using the DELETE method to remove a movie from the database with admin validation.	The movie is deleted and removed from the database by the admin.	The movie was removed from the database by the admin user.	The movie deleted successfully, removing the data from the database, the operation was done by the admin user as the normal users do not have access.

6	Adding an Actor to a movie	This is a POST that add an actor to a movie using actor data and being an admin user for validation.	The actor was successfully added to the movie by the admin user.	The actor was successfully added to the movie.	Adding the actors to movies was successful, validation was correct as the normal user cannot perform these operations.
7	Removing an actor from a movie.	Using the Delete method the actor is removed from the movie by an admin user.	The actor is removed from the movie but not the database, this is run by an admin user.	The actor was successfully removed from the movie.	The actor being removed from the movie without deleting the actor from the database entirely. This was done by the admin user.

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
8	Getting all actor data.	Using the GET method to get all the actors from the database with validation.	The actor data is fetched gathering all the movies in the database.	All the movies were gotten from the database and displayed as a list of actors.	The GET passed giving back all the actors in the database.
9	Creating an actor.	Using the POST method to make a new actor with admin validation.	The actor should be created and added to the database; the admin user should be allowed to create the actors	The actor was created and added to the database successfully.	The POST was successful adding a new actor to the database with the correct user validation.

			while normal user will not.		
10	Editing an actor.	This uses the PUT method to alter existing actor data, this operation can only be done with admin validation.	The actor data is changed and is then updated in the database; this operation being done by an admin user.	The actor's data was successfully altered and changed in the database.	The operation was successful updating the actor in the database. This being done by the admin user.
11	Getting a single actor.	Using the GET method and specifying the id of the actor to find its data.	The specific actor should display the data that corresponds with it.	The specific actor was found and the data displayed.	The GET method passed giving the actor data with the id that was specified.

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
12	Creating a watchlist.	Using a POST method and sending data to create the watchlist, validation is also sent to authentication	The watchlist should be created with the data inputted.	The watchlist was successfully created.	The operation was successful creating the new watchlist and adding the data to the database.
13	Getting all watchlists.	Using the GET method to get all the data for the watchlists that correspond with the users id.	The user should see all the watchlists that are associated with their id.	The watchlists display and the user can view their own watchlists.	The GET was successful displaying all relevant watchlists to the user.

14	Getting a single watchlist.	Using the GET method and specifying the id to get the correct watchlist data.	The watchlist should display the data for the watchlists including the movie_ids, public being true or false, etc.	The watchlist data was returned successfully displaying its data showing the movie ids and status of the watchlist.	The method was a success displaying the correct data for the watchlist.
15	Editing a watchlist.	Using the PUT method to alter the data for the watchlist.	The data in the watchlists should be altered to match the new updated data.	The watchlists update as expected editing the data present in the database to match the altered data.	The PUT was successful in updating the data for the watchlists.
16	Deleting a watchlist.	Using the DELETE method to remove the watchlist from the database.	The watchlists data should be removed entirely from the database.	The watchlist was removed from the database.	The DELETE method worked as expected removing the data for the watchlist.
17	Getting all the public watchlists.	Using the GET method to get all the watchlists that have the is_public attribute set to public.	It should give a list of watchlists that other users have set to public. These watchlists are then put into a separate list which then allows all	The public watchlists were displayed as expected.	Getting the public watchlists was a success displaying only those watchlists that are set to public.

			the users to view the public watchlists		
18	Getting a single public watchlist.	Using the GET method with the specified id and if it is set to public to get the data for a specific watchlist.	The data for a single public watchlist should display its data.	The data for the single public watchlist displayed data for the correct watchlist.	The output was a success displaying the public watchlist data.

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
19	Creating a review.	Using a POST method to send the review data with validation.	The review should be created with the inputted data.	The review was created successfully and added to the database.	The operation was successful posting the review data to the database.
20	Getting a review	Using the GET method to get all the reviews in the database with its data with validation.	All the reviews should be displaying as a list of reviews.	All the reviews display and can be viewed by any user.	The GET was successful in getting all the data for the reviews.
21	Editing a review	Using the PUT method to alter the data in the review with validation.	The review should be updated with the new inputs the user has chosen.	The reviews were updated and the data altered.	The update for the reviews was successful altering the data for the review and sending it to the database.

22	Deleting a review	Using the DELETE method to remove the review from the database.	The review should be deleted which should remove the review data from the database.	The review data was removed successfully.	The DELETE method worked as expected removing the review from the database.
----	-------------------	---	---	---	---

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
23	Creating a rating.	Using the POST method to send the rating data with validation.	The rating will be created and added to the database.	The rating was created.	The rating data was added to the database signifying the operation was a success.
24	Getting a rating.	Using the GET method to fetch all the ratings and the corresponding data with validation.	All the ratings were displayed if the user id matches the ratings.	The ratings were successful gotten from the database.	It is important that the user cannot interact with other user's ratings, this was a problem now it is solved.
25	Editing a rating.	Using the PUT method to alter the data in the rating with validation.	The data being the content of the rating is changed to a different number from 1-5.	The rating was successfully updated to include the new rating value.	The rating was successfully updated and sent to the database.
26	Deleting a rating.	Using the Delete method to remove the rating data	The rating with the specific id is deleted and removed	The rating was successfully removed from the database.	The DELETE method for the rating was successful.

		from the database.	from the database.		
--	--	--------------------	--------------------	--	--

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
27	Creating a ranking for the user.	Using a POST to make the ranking with the data.	The rankings are made and posted to the database.	The rankings were added to the database.	This data is being extracted from the basic ranking model to be used for the user recommendations.
28	Getting all rankings for the user.	Using the GET method to get all the top 5 highest ranked movies for the users.	5 movies will display in a list showing the highest rated movie down to the lowest.	The rankings were gotten and display as expected.	These recommendations are not accurate for the users in the database and works more to show the concept of how the recommendation system was supposed to work.

### 5.2.2 Discussion of Functional Testing Results

Fig (5.0) Results from Unit Tests

tests/test_actor.py::test_get_all_actors_success PASSED	[ 2%]
tests/test_actor.py::test_create_actor_success PASSED	[ 5%]
tests/test_actor.py::test_create_actor_invalid_birthday PASSED	[ 8%]
tests/test_actor.py::test_get_actor_success PASSED	[ 11%]
tests/test_actor.py::test_get_actor_not_found PASSED	[ 13%]
tests/test_actor.py::test_update_actor_success PASSED	[ 16%]
tests/test_actor.py::test_delete_actor_success PASSED	[ 19%]
tests/test_actor.py::test_remove_actor_from_movie_success PASSED	[ 22%]
tests/test_movies.py::test_get_movies_success PASSED	[ 25%]
tests/test_movies.py::test_get_movies_unauthorized PASSED	[ 27%]
tests/test_movies.py::test_get_single_movie_success PASSED	[ 30%]
tests/test_movies.py::test_get_single_movie_not_found PASSED	[ 33%]
tests/test_movies.py::test_create_movie_success PASSED	[ 36%]
tests/test_movies.py::test_create_movie_unauthorized PASSED	[ 38%]
tests/test_movies.py::test_update_movie_success PASSED	[ 41%]
tests/test_movies.py::test_delete_movie_success PASSED	[ 44%]
tests/test_movies.py::test_add_actor_success PASSED	[ 47%]
tests/test_movies.py::test_remove_actor_success PASSED	[ 50%]
tests/test_ranking.py::test_get_top_ranked_movies_success PASSED	[ 52%]
tests/test_ranking.py::test_get_top_ranked_movies_missing_user_id PASSED	[ 55%]
tests/test_rating.py::test_add_rating_success PASSED	[ 58%]
tests/test_rating.py::test_add_rating_invalid_rating PASSED	[ 61%]
tests/test_rating.py::test_get_user_ratings_success PASSED	[ 63%]
tests/test_rating.py::test_update_rating_success PASSED	[ 66%]
tests/test_rating.py::test_delete_rating_success PASSED	[ 69%]
tests/test_review.py::test_create_review_success PASSED	[ 72%]
tests/test_review.py::test_get_movie_reviews_success PASSED	[ 75%]
tests/test_review.py::test_update_review_success PASSED	[ 77%]
tests/test_review.py::test_delete_review_success PASSED	[ 80%]
tests/test_watchlist.py::test_create_watchlist_success PASSED	[ 83%]
tests/test_watchlist.py::test_get_user_watchlist_success PASSED	[ 86%]
tests/test_watchlist.py::test_get_watchlist_item_success PASSED	[ 88%]
tests/test_watchlist.py::test_update_watchlist_success PASSED	[ 91%]
tests/test_watchlist.py::test_delete_watchlist_success PASSED	[ 94%]
tests/test_watchlist.py::test_get_public_watchlists_success PASSED	[ 97%]
tests/test_watchlist.py::test_get_public_watchlist_success PASSED	

The CRUD functionality was tested, testing the movies, actors, watchlists, ratings, reviews, and rankings in the application. The twenty-eight test that were conducted go over the core functions, the authorization and validation of a user to be able to do the operations, and how the data interacts with the database.

## Movies

The movies core functions all operate as expected including, GET, CREATE, PUT, and DELETE. These functions can only work if the validation requirements are met, these requirements are for the PUT and DELETE methods only allowing admin users to use these functions.

## Actors

The actors core functions were all successfully tested going over the GET, CREATE, PUT, and DELETE methods. The admin user validation is required to do the PUT, DELETE, and CREATE methods while the normal users can view the single actor data. Allowing admins to remove and assign actors to movies is also a function in which only the admin user can do.

## Watchlists

The watchlists operate differently from the movie and actors as the CRUD functions are open to the normal users as they do not need admin validation for the operations those being the CREATE, PUT, DELETE, and GET methods, the user can receive all their watchlists, can edit them, and create new watchlists, and can delete them. The public watchlists are fetched using a GET request and display the watchlists that have been set to public, the users can get all the watchlists from all the users if



set to put like using this method. The public watchlists also allow the user to view the specific watchlists data if for the public watchlists to view other users watchlists.

### Ratings

The ratings allow the users to use the CRUD functionality but only on ratings that they are assigned to as they cannot see other ratings from other users. Through the tests it shows that the ratings use the user's id and prevent other ids from altering the data if unauthorized. This was bugged and was solved.

### Reviews

The reviews work similarly to the ratings allowing users to use the CRUD functionality being GET, PUT, POST, and DELETE if the review belongs to the user's id. The user cannot alter or remove another user review as the user is not authorized.

### Rankings

The ranking, which extracts data from the basic ranking model was tested to see if the ranking data could POST and GET the data.

The tests were a success resulting in the CRUD functionality, validation, and user logic passing the tests. These tests were done to verify the that these functions were correct and ran properly.

## 5.3 User Testing

Fig (5.1) User Testing Tasks

1. Register an account on the application.
2. Once on the home page go to the watchlists page.
3. Create a watchlist.
4. navigate to the movie page and select a movie.
5. From the selected movie choose your watchlist and add the movie to it.
6. Add a rating to the movie.
7. Leave a review on the movie.
8. Navigate back to the watchlists and select your watchlist.
9. From your selected watchlist make it public, and edit the title of the watchlist.
10. navigate to the public watchlist page and find your watchlist.
11. View your public watchlist.
12. navigate to the home page and see the movie that have been recommended.
13. From home edit the rated movie and then remove it.
14. Navigate to the profile page and log out.
15. Using these credentials, email: admin@moviemuze.com, password: adminpassword.
16. Once logged in as the admin, navigate back to the movie page.
17. Create a movie.
18. Use the search bar to find your movie.
19. Select the movie and then edit the title and actor.
20. Find your movie again and delete it.
21. Try and find it again using search.
22. Navigate to the actor page.
23. Assign an actor to a the first movie.
24. Edit the actors name.
25. Go to the movie that the actor was added to and remove it.
26. Delete the actor.
27. Navigate back to the profile page and sign out.

User testing was done to get feedback from a real user about how their experience using the application was. This was done to see the usability of the application from their perspective, this included testing the navigation and interactable components without previous knowledge of the application.

The users were given tasks to complete, these tasks include using the functionality in the front end to either send data, receive data, or delete data.

Throughout the process of the user testing the users were able to speak and give their thoughts while engaged in the process of completing the tasks. From this their feedback was gathered to get a sense of how feasible the layout of the features is.

Once the testing was finished the feedback received from the user indicated that some placements of buttons, clearer messages for when adding a rating, review, and watchlist so the users know they were added and other issues that arose.

The tests overall were successful in that the users gave great feedback on improvements that need to be corrected for the application to be more usability giving the users a better experience.

## 5.4 Conclusion

The testing chapter went over the functional tests and the user testing that was conducted. It went over the CRUD functionality of the modules in the backend, and the users were giving the application to use and navigate through to test navigation, components, and the design of the app. The tests addressed unseen issues that when fixed, elevated the project.

## 6 Project Management

### 6.1 Introduction

This chapter goes over the process of management for the project; it goes over the details of the project through several phases beginning with the initial phase to the final implementations and the testing. It goes through the stages which include the requirement gathering for the development of the designs and the implementation features, the design phase laid down the foundations of the implementation, and testing. Tools such as Trello was used to track the goals that needed to be completed for the sprints, and GitHub was used to track and manage the code for the front and back end of the application.

### 6.2 Project Phases

The project consisted of five phases. The proposal was the initial step outlining the goal of the project, the requirements phase was done to gather the requirements for the application, the design phase was done to gain a base for the project, the implementation phase consists of the implementations that were added to the project during development, and the testing phase went over the tests that were done for the application.

#### 6.2.1 Proposal

The proposal was the initial phase that outlined the goals and the purpose of the project. This phase was about outlining the scope of the project, what technologies would be used to build the application, and what problem does this application so solve.

#### 6.2.2 Requirements

The requirements phase focused on the gathering of the features and functionality of the application. The goal was to add the functionality while also keeping the project feasible so it could be made within the timeline of the project.

#### 6.2.3 Design

The design phase included creating the backend and front-end designs so that when production on the application began there would be an outlined path to follow. This came with its own challenges such as changing technologies and trying to fit the new one into the designs. In this phase the focus was designing the backend going over the projects routes models and other necessary files. The front end involved the same designs for the file structure along with designs for the look of the application.

#### 6.2.4 Implementation

In the implementation phase it involved transferring the designs of the application into a real working application. Many challenges arose from this phase such as the model not working as intended, or certain features on the application not working as intended. Overall, the implemented designs were successful.

#### 6.2.5 Testing

The testing was conducted on the functionalities and usability of the application. This was done through functional tests focusing on the API endpoints for the application and user testing was done to identify issues with navigation, function placement, and finding any bugs or areas of confusion the user had whilst using the app.

### 6.3 SCRUM Methodology

#### Sprints

Overall, the sprints were helpful to stay organized. The sprints provided a structure to the project making it more manageable as there were outlined steps to take for every sprint. It shows the progress the application went through before the last version was created.

How well did the 7 sprints work?

The sprints worked well with some items being backlogged for more time that was needed. The objectives every two weeks were defined, and this was critical for the project development. It allowed for constant improvement and getting feedback from the supervisor.

#### Requirements

The first sprint goes over the requirements that would be needed to for the app, laying down its foundation.

#### Design

The design phase shows the process of designing elements of the backend and front-end such as the ERD for the database, and the wireframe for the Front-end. This gave the project structure and an end goal to reach.

#### Implementation

This sprint was more difficult to complete than the previous two as setting up a new technology was strenuous along with trying to integrate the model from TensorFlow into that backend. This cause a backlog in the project until the model was correctly implemented. More problems arose with model and authentication, and it took time for this sprint to be completed.

## Design 2

In design two this sprint involved altering the designs of the code to add more features and overhauling the designs of the previous design chapter. In this phase the newly added designs were implemented into the backend.

## Implementation 2

The second implementation chapter involved creating the front-end of the application which included making the app.js file, pages, components, and adding in their own functions within these pages with data from the backend.

## Testing

The testing phase went over the functional and user testing to make sure the applications designs, and functionality all worked and was usable by the user.

## Report

This sprint went over the previous phases in the application.

## Project Backlog

The backlog of the project was updated while doing the sprints, some tasks were prioritized more than others for their importance in building up the application such as a backlog on the routes because of the model having issues. This allowed for the project to stay organized.

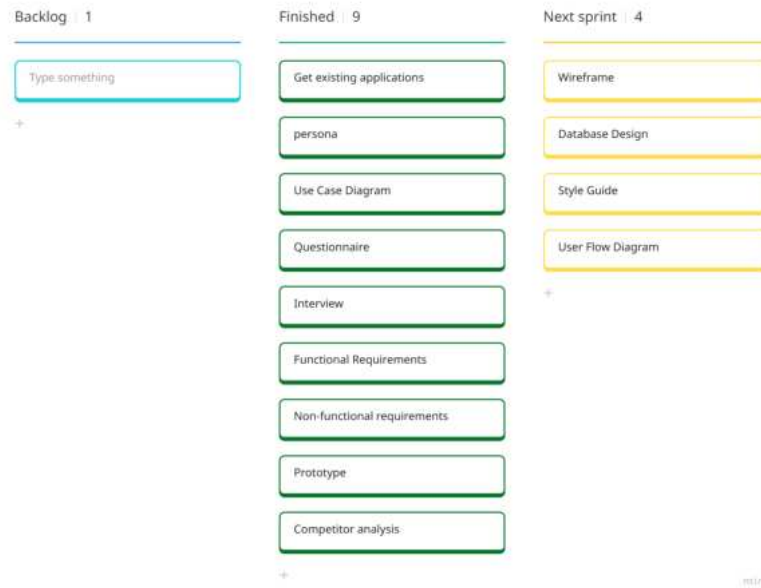
## 6.4 Project Management Tools

### 6.4.1 Trello

#### Description

Trello was used to create boards and tasks for the project to follow. It broke up the tasks into smaller tasks which made it more manageable when going to implement these tasks into the actual project. Each task could be put into a section those being the backlog, completed, and the next set of tasks for the following two weeks.

Fig (6.0) Trello diagram

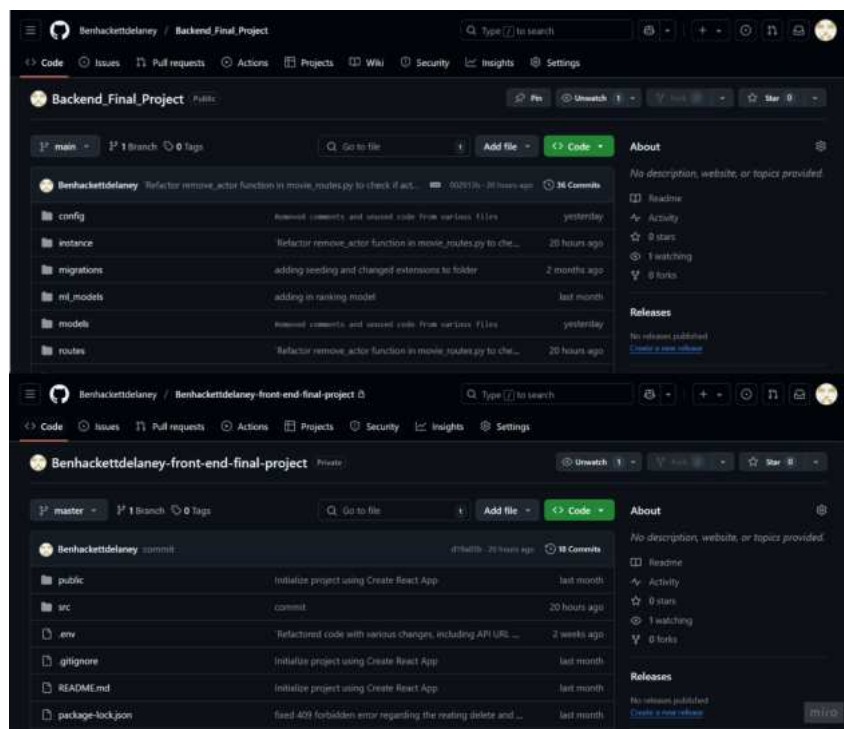


## How it worked in practice

The Trello boards worked well in practice, it helped to keep up with the work and stay organized, even if some sections had backlogged worked for a long time it was easy to go back and make the task as incomplete so it could do later. Overall, the Trello boards kept the project organized throughout the project's creation.

## 6.4.2 GitHub

Fig (6.1) GitHub for Front & Backend



### Description

GitHub was used to manage the code for the backend and the front-end. It allowed for the tracking of the code, and helped manage the code in case problems arose.

### How it is used

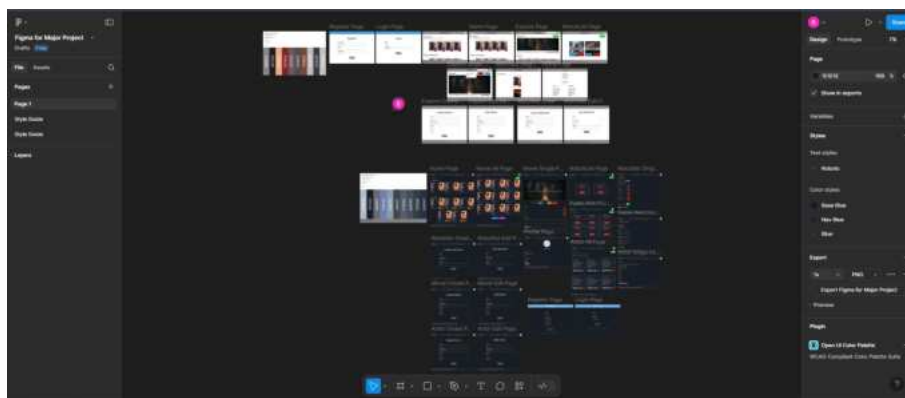
GitHub was initialized in the back and front-end and the code was sent in batches describing what the code was for and why it was implemented.

### How it worked in practice

GitHub worked well and kept the code organized.

## 6.4.3 Figma

Fig (6.2) GitHub for Front & Backend



### Description

Figma is a design tool for creating UI interfaces. It allows the user to create wireframes, prototypes, and components.

### How it is used

Figma was used to create the prototype and wireframe for the applications front-end.

### How it worked in practice

Figma was a useful tool, it allowed for the designs to be altered, the colour scheme changed, with minimal difficulty.



#### 6.4.4 Miro

Fig (6.3) GitHub for Front & Backend



#### Description

Miro is a digital whiteboard platform that give the user a workspace where they can display their projects.

#### How it is used

Miro was used to keep track of sprints and tasks; it allowed for a more organized approach with every task laid out.

#### How it worked in practice

Miro worked well for managing the project making sure it was kept organized.

### 6.5 Reflection

#### 6.5.1 Your views on the project

The project was a great learning experience as the scale of the project grew more as the development went on and the necessary changes were made to create the project. Some area of the application need improvement such as the design but overall, the project is solid and works as intended.

#### 6.5.2 Completing a large software development project

This project showed the importance of keeping the scale down as it can come back to be an issue later if not dealt with properly. The planning of the project is critical for the success of the application as time management was an area where the organization fell off. The skill that would be improved from this would be the scale that was set and the time management.

### 6.5.3 Working with a supervisor

The supervisor in the beginning with all the problems the application was facing due to the model having problems cause some conflict but when that was solved and the rest of the project fell into place. The supervisor gave confidence in the build and gave useful tips to improve the app. It was good to work with a supervisor to keep on top of the project.

### 6.5.4 Technical skills

The knowledge gained from the project would be the modern technologies that were use. Flask was a new using Python to create the backend that would be used for the application. Learning how the machine learning model was getting the prediction was a good skill to gain as this sector of coding is relevant today.

### 6.5.5 Further competencies and skills

The skills that would gain from this application involved improving time management, communication skills, and my own technical skills while coding. Thes skills that were gained will further expand my knowledge of the area.

### 6.5.6 Problems during implementation

The implementation of the model was a problem from the beginning of the application. From trying to export it with failing results to the model not working as it was supposed to. Initially the issue was the model could not be exported from Google Colab, once this was solved the model was implemented into the backend. When querying the model, the realization began that the model was trained on a dataset and goes through all the prediction as intended on google colab, the problem was that the model does not take in new data and only predicts with the data it used for the training. This means that new data made from user on the application would not be used by the model. This causes the application to not recommend movies properly. The query for the route was made for the model to query the model on the id, movie\_title, and movie\_genres. Then the movies from the model are then ranked by the top five highest. The movie and user data are seeded with the same data as the model so the ids of the match the one in the model, it is still an inaccurate recommendation as the model does not interact with the database. This was done to simulate what it was supposed to be, how it would work, and how it would be displayed in the front-end.

## 6.6 Conclusion

The Project Management goes over the phases of the project and how they were managed. It gives an overview of the sprints that were done and the tasks that were involved with each, the tools that were used to keep the project organized, and a short reflection. Despite the challenges that were posed by this application, specifically problems with the model, the project was created.



## 7 Conclusion

The goal was to create a movie recommendation application that would use a machine learning model to recommend movies. The model was not able to be fully functional in this project however the model still give data and is used. Although this aspect of the application was not fully realised the other functionality such as the adding actors to movies or creating watchlists and many other were fully functional and showed how the application was supposed to work in relation to the model.

The application used Flask as the backend as an API. The front-end was created in React.js, and TensorFlow was used for the machine learning basic ranking model. The database was SQLite. These technologies were used in the development of the application. Using Flask for the first time was challenging but once the structure was learned it was manageable. React was more familiar as it was used in previous applications. TensorFlow was a big challenge to integrate, the model had a lot of problems not mentioning its inability to take in new data. Overall, working with these technologies was a great learning experience and improved skills in Python and Machine Learning.

### Research

The research section involved researching about machine learning recommender systems. There were many avenues the look at as different recommender systems were applicable to different needs. The research also goes over the data and how it would be stored. The last section was on how to integrate the model from TensorFlow to the application, and it discusses methods for this.

### Requirements

The requirements chapter goes over the process of retrieving the functional and non-functional requirements for the application. It gave a road map for the features that the application would need to implement.

### Design

The design chapter give a foundation for the requirements that were gathered. It gives a visual process of how the application was going to be built, what it would look like, and the components that would be integrated.

### Implementation

The implementation chapter goes over the implementations that were made using the designs as a base. It goes over the sections of code that were made, the integration of the model, the process of designing and gathering requirements, and it goes over the testing of the application including functional and user testing.

### Testing

The testing chapter goes over the tests that were conducted in the application. It began with functional testing that tests the CRUD functionality using authenticated credentials. It also goes over the user testing and the tasks they were asked to complete, when completed

or during the process the test users would give feedback on improvements that can be made.

#### Overall result

The results for the functional tests were good with all tests for all the modules passing, this included more tests such as if the user had invalid credentials. The user testing was also positive with minor issues appearing. These issues included button placement, confusing error messages, and designs of certain pages.

#### Project management

The project management chapter gave an overview of the phases of the chapter, and a reflection of the overall experience building the application. It goes over tools that were used, what sprints were done, and challenges that were faced during development.

#### What was learnt

Throughout the project it was a learning process for all the steps involving Flask and TensorFlow with Google Colab, i have gained skilled in both technologies that can be used for future projects.

#### How the project could be further developed

To improve the project the model would have to be improved. This could be done by either creating the model itself from scratch, which would be difficult and time consuming, or finding a better model and modifying it to take in data directly, this would also be difficult. The use of a bigger database to store more data would be implemented to handle more data coming in from the model.

## References

### Questionnaire

[https://forms.office.com/Pages/ResponsePage.aspx?id=e5V92hEVQkqy9Xj4R\\_jlekGRf67e9PRDniQ86DzeckhUMUcwTkPCV1EwQzE2MVdESTI1UlK5MTAwMi4u](https://forms.office.com/Pages/ResponsePage.aspx?id=e5V92hEVQkqy9Xj4R_jlekGRf67e9PRDniQ86DzeckhUMUcwTkPCV1EwQzE2MVdESTI1UlK5MTAwMi4u)

### Figma

<https://www.figma.com/design/PNu4ugz8GTH2mdozz9pOy6/Figma-for-Major-Project?node-id=0-1&t=1UBK2AROEaFkdOUB-1>

### Google Colab

<https://colab.google/>

### TensorFlow Model

[https://colab.research.google.com/drive/1bvG-gLGub9\\_2dbIMcMiQuUe6G-H0JoJT?usp=sharing](https://colab.research.google.com/drive/1bvG-gLGub9_2dbIMcMiQuUe6G-H0JoJT?usp=sharing)

### Miro

<https://miro.com/app/board/uXjVLFumj0E=/>

### A Categorical Review of Recommender Systems (Prasad and Kumari, 2012).

[https://d1wqtxts1xzle7.cloudfront.net/37823853/7-libre.pdf?1433412601=&response-content-disposition=inline%3B+filename%3DA\\_CATEGORICAL\\_REVIEW\\_OF\\_RECOMMENDER\\_SYST.pdf&Expires=1735501206&Signature=eN5BOYn06QbC~FdmWbstmezYH~niYJc6mCo-L-XnzSxAC0o790cR32Wy5SJcX6KvEiK8YqokrZ8RoBCar5-KqvW8LLmcCuGrtJoxGU6ezFC0OpI3bWE5EfgbL7k5yFbYQTlyYnI7an8Fg~5TvuFDq33Ja47QbfJPF~d2EZCixaA4MRCtHa9iQexGhuUASLpyUIntHGXFv15uRh-hBiDok1Ur1c5OMxmzLinJVpxfWm5vRHbifVN2U9LPDr20zr4UilKn1JsDSJPFghLeqVn9fNtOU70ZdXsfN8tWbclpEyNW7xF7fzoB5OAa8l~cxNlgDDx4eLvqy6629ciGi20aMg\\_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA](https://d1wqtxts1xzle7.cloudfront.net/37823853/7-libre.pdf?1433412601=&response-content-disposition=inline%3B+filename%3DA_CATEGORICAL_REVIEW_OF_RECOMMENDER_SYST.pdf&Expires=1735501206&Signature=eN5BOYn06QbC~FdmWbstmezYH~niYJc6mCo-L-XnzSxAC0o790cR32Wy5SJcX6KvEiK8YqokrZ8RoBCar5-KqvW8LLmcCuGrtJoxGU6ezFC0OpI3bWE5EfgbL7k5yFbYQTlyYnI7an8Fg~5TvuFDq33Ja47QbfJPF~d2EZCixaA4MRCtHa9iQexGhuUASLpyUIntHGXFv15uRh-hBiDok1Ur1c5OMxmzLinJVpxfWm5vRHbifVN2U9LPDr20zr4UilKn1JsDSJPFghLeqVn9fNtOU70ZdXsfN8tWbclpEyNW7xF7fzoB5OAa8l~cxNlgDDx4eLvqy6629ciGi20aMg_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA)

### A Review on Recommender System (Fahin Mansur, Vibha Patel, Mihir Patel. 2017).

<https://ieeexplore.ieee.org/abstract/document/8276182>

[https://www.researchgate.net/profile/Vibha-Patel-3/publication/322994553\\_A\\_review\\_on\\_recommender\\_systems/links/5c4eb80e299bf12be3e8e9c6/A-review-on-recommender-systems.pdf](https://www.researchgate.net/profile/Vibha-Patel-3/publication/322994553_A_review_on_recommender_systems/links/5c4eb80e299bf12be3e8e9c6/A-review-on-recommender-systems.pdf)

Artificial Intelligence in Recommender Systems (Qian Zhang, Jie Lu, Yaochu Jin, 2020). (“Figure 1 from Personalized Recommendation Algorithm for Electronic ...”)

<https://link.springer.com/article/10.1007/s40747-020-00212-w>

<https://link.springer.com/content/pdf/10.1007/s40747-020-00212-w.pdf>

Built Machine Learning powered apps using Node.JS (Dimas, 2024)

<https://dimasmds.medium.com/built-machine-learning-powered-apps-using-node-js-318ca0413818>

A Survey on Data Mining Techniques in Research Paper Recommender System (Bernard Magara Maake, Sunday O. Ojo, Tranos Zuva, 2019).

<https://www.igi-global.com/chapter/a-survey-on-data-mining-techniques-in-research-paper-recommender-systems/232427>

Collaborative Filtering Recommender System (J. Ben Schafer, Dan Frankowski, Jon Herlocker, Shalid Sen. 2007).

[https://link.springer.com/content/pdf/10.1007/978-3-540-72079-9\\_9?pdf=chapter%20toc](https://link.springer.com/content/pdf/10.1007/978-3-540-72079-9_9?pdf=chapter%20toc)

Data Mining Methods for Recommender Systems (Xavier Amatriain, Alejandro Jaimes, Nuria Oliver, Josep M. Pujol).

[https://link.springer.com/chapter/10.1007/978-0-387-85820-3\\_2](https://link.springer.com/chapter/10.1007/978-0-387-85820-3_2)

<https://amatria.in/pubs/RecsysHandbookChapter.pdf>

Quality Matters in Recommender Systems (Oren Sar Shalom, Shlomo Berkovsky, Royi Ronen, Elad Ziklik, Amihod Amir, 2015).

<https://dl.acm.org/doi/abs/10.1145/2792838.2799670>

[https://www.researchgate.net/profile/Oren-Sar-Shalom/publication/301432329\\_Data\\_Quality\\_Matters\\_in\\_Recommender\\_Systems/links/60c580df4585157774d23daa/Data-Quality-Matters-in-Recommender-Systems.pdf](https://www.researchgate.net/profile/Oren-Sar-Shalom/publication/301432329_Data_Quality_Matters_in_Recommender_Systems/links/60c580df4585157774d23daa/Data-Quality-Matters-in-Recommender-Systems.pdf)

Front-end deep learning web apps development and deployment: a review (Hock-Ann Goh, Chin-Kuan Ho, Fazly Salleh Abas, 2022).

<https://link.springer.com/article/10.1007/s10489-022-04278-6>

<https://link.springer.com/content/pdf/10.1007/s10489-022-04278-6.pdf>

Recommender Systems: An overview of different approaches to recommendations (Goldberg et al. 1992) (Kunal Shah, Akshaykumar Salunke, Saurabh Dongare, Kisandas Antala. ("Product Recommendation System") ("Product Recommendation System") 2017).

<https://ieeexplore.ieee.org/abstract/document/8276172>

[https://www.researchgate.net/profile/Saurabh-Dongare/publication/323000727\\_Recommender\\_systems\\_An\\_overview\\_of\\_different\\_approaches\\_to\\_recommendations/links/60cec798458515dc17951d54/Recommender-systems-An-overview-of-different-approaches-to-recommendations.pdf](https://www.researchgate.net/profile/Saurabh-Dongare/publication/323000727_Recommender_systems_An_overview_of_different_approaches_to_recommendations/links/60cec798458515dc17951d54/Recommender-systems-An-overview-of-different-approaches-to-recommendations.pdf)

How to Build a Recommender System for Software Engineering (Sebastian Proksch, Veronika Bauer, Gail C. Murphy, 2015).

[https://link.springer.com/chapter/10.1007/978-3-319-28406-4\\_1](https://link.springer.com/chapter/10.1007/978-3-319-28406-4_1)

[https://www.researchgate.net/profile/Veronika-Bauer/publication/299398536\\_How\\_to\\_Build\\_a\\_Recommendation\\_System\\_for\\_Software\\_Engineering/links/56f40c7308ae81582bf09dd9/How-to-Build-a-Recommendation-System-for-Software-Engineering.pdf](https://www.researchgate.net/profile/Veronika-Bauer/publication/299398536_How_to_Build_a_Recommendation_System_for_Software_Engineering/links/56f40c7308ae81582bf09dd9/How-to-Build-a-Recommendation-System-for-Software-Engineering.pdf)

Recommender System Based on the Analysis of Publicly Available Data (Goran Antolic, Ljiljana Brkic, 2017).

<https://ieeexplore.ieee.org/abstract/document/7973637>

[https://www.researchgate.net/profile/Ljiljana-Brkic/publication/318690813\\_Recommender\\_system\\_based\\_on\\_the\\_analysis\\_of\\_publicly\\_available\\_data/links/5aff2d1d4585154aeb041351/Recommender-system-based-on-the-analysis-of-publicly-available-data.pdf](https://www.researchgate.net/profile/Ljiljana-Brkic/publication/318690813_Recommender_system_based_on_the_analysis_of_publicly_available_data/links/5aff2d1d4585154aeb041351/Recommender-system-based-on-the-analysis-of-publicly-available-data.pdf)

The k Closest Resemblance Classifiers for Amazon Products Recommender Systems (Nabil Belacel, Guanze Wei, Yassine Bouslimani, 2020).

<https://www.scitepress.org/Papers/2020/91551/91551.pdf>



Movie Recommender System Using Collaborative Filtering (Meenu Gupta, Aditya Thakkar, Aashish, Vishal Gupta, 2020). ("Movies Recommendation System Using Cosine Similarity - Academia.edu") ("Movie Recommender System Using Collaborative Filtering - Semantic Scholar")

[https://www.researchgate.net/profile/Meenu-Gupta-8/publication/348239082\\_Movie\\_Recommender\\_System\\_Using\\_Collaborative\\_Filtering/links/5ff4adbdb299bf1408874ca98/Movie-Recommender-System-Using-Collaborative-Filtering.pdf](https://www.researchgate.net/profile/Meenu-Gupta-8/publication/348239082_Movie_Recommender_System_Using_Collaborative_Filtering/links/5ff4adbdb299bf1408874ca98/Movie-Recommender-System-Using-Collaborative-Filtering.pdf)

A Categorical Review of Recommender Systems (Prasad, Kumari, 2012).

[https://d1wqtxts1xzle7.cloudfront.net/37823853/7-libre.pdf?1433412601=&response-content-disposition=inline%3B+filename%3DA\\_CATEGORICAL\\_REVIEW\\_OF\\_RECOMMENDER\\_SYST.pdf&Expires=1735505584&Signature=fgeWlmxq1CTXC8Jw~mmiD15C3d3crHaHGkYzSus7dhqvVNSZqRHIPfQawabl4H9QlXBEJgTKO0RZHvumRyanS6QAqMF45zutMcV~5y3X9BX-SFOtmb57vKNkequyBvqRZXDnhrcYUggX8V-RpEm6JrqIYrnkPI8qDzidNJ0Z36pVxuPkxpQUTY8E90~VFRvo8jY2RS7OkxY4y0uoYbRggwHAyzmENGtyuVVqYS79lZgF3Z0ftguYVlKTHtbpkeucTKcQHnDmMmvl-eDn15mOrBBDAuV574FMbbb9YG-Inr0WjOcNYIxbLap8qO7yJ~SIN6wzWK7lIVetmz40AMaCg\\_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA](https://d1wqtxts1xzle7.cloudfront.net/37823853/7-libre.pdf?1433412601=&response-content-disposition=inline%3B+filename%3DA_CATEGORICAL_REVIEW_OF_RECOMMENDER_SYST.pdf&Expires=1735505584&Signature=fgeWlmxq1CTXC8Jw~mmiD15C3d3crHaHGkYzSus7dhqvVNSZqRHIPfQawabl4H9QlXBEJgTKO0RZHvumRyanS6QAqMF45zutMcV~5y3X9BX-SFOtmb57vKNkequyBvqRZXDnhrcYUggX8V-RpEm6JrqIYrnkPI8qDzidNJ0Z36pVxuPkxpQUTY8E90~VFRvo8jY2RS7OkxY4y0uoYbRggwHAyzmENGtyuVVqYS79lZgF3Z0ftguYVlKTHtbpkeucTKcQHnDmMmvl-eDn15mOrBBDAuV574FMbbb9YG-Inr0WjOcNYIxbLap8qO7yJ~SIN6wzWK7lIVetmz40AMaCg_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA)

TF-Ranking: Scalable tensorflow library for leaning-to-rank (Heng-Tze Cheng, Zakaria Haque, Lichen Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D Sculley, Jamie Smith, David Soergel, 2017)

<https://dl.acm.org/doi/abs/10.1145/3292500.3330677>

SecureTF: A Secure TensorFlow Framework. (Grant Allen, Mike Owens, 2010).

<https://dl.acm.org/doi/abs/10.1145/3423211.3425687>

The Definitive Guide to SQLite. (Grant Allen, Mike Owens, 2010).

<https://www.programmershouse.ir/Library/Files/SQLite.pdf>