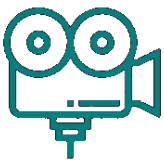
University of Pisa

Department of Information Engineering

Large Scale and Multi-Structured Databases



MOVIELAND

INDEX

INTRODUCTION

MOVIELAND is a comprehensive platform designed for movie and TV show enthusiasts, providing an intuitive and engaging experience for exploring, discovering, and sharing cinematic content. With advanced features and a user-friendly interface, users can access reviews, rate titles, and uncover new content effortlessly.

The platform allows for a highly personalized experience by enabling users to create profiles, curate watchlists, and select favorite genres, actors, and films. Every rating and review contribute to a global conversation, enhancing the collective knowledge and recommendations within the community.

MOVIELAND incorporates a sophisticated recommendation system that suggests movies, actors, and like-minded users based on individual preferences and ratings. Additionally, the advanced search functionality, equipped with detailed filters, ensures quick and accurate content discovery.

Beyond content exploration, MOVIELAND fosters interaction among users through discussions, comments, and shared insights, creating a dynamic and engaged community. It is more than just a content catalog; it is an evolving ecosystem that enriches the passion for cinema and television.

MOVIELAND offers an immersive and tailored experience, making it the ideal destination for anyone who appreciates the world of film and television.

The code is available on GitHub: github.com/Leo-Cecchini/Movieland

1. APPLICATION MANUAL

MOVIELAND is a comprehensive platform developed for film enthusiasts, offering a space where users can discover and enjoy their favorite movies and TV series. It also enables users to share their opinions and connect with a community of like-minded cinephiles. A standout feature of MOVIELAND is its personalized recommendation system, which suggests new movies and series based on user preferences.

1.1 APPLICATION HIGHLIGHTS

MOVIELAND combines the features of a social network with a focus on movies and TV series, emphasizing two main functionalities:

- **Search**: Users can easily search for movies, TV series, and actors, and they can access dedicated pages with detailed information, including cast lists, and user-generated reviews.
- **Discover**: The application is designed to help users explore new films and series, showcasing trendy titles and personalized recommendations across the platform.

Additionally, MOVIELAND includes specialized administrative tools for content moderation, statistical analysis, and potential monetization strategies.

1.2 USER MANUAL

Upon accessing the platform, users are greeted with options to log in or register, along with a search bar and links to trending movies and TV series.

Unregistered Users

Unregistered users can explore the site but have limited interaction. They can:

- Search for movies and celebrities.
- View movies and celebrities' pages.
- Browse trending content.

However, they cannot interact by liking movies, writing reviews, or managing a watchlist. These features become available upon registration.

Registered Users

Once logged in, registered users gain full access to the platform's features. They can:

- Navigate to their personal profile page.
- Browse the **Most Popular** section, showcasing top-rated films and TV series.
- Find movies, TV series, actors, or other users.
- Discover new films through the **Recommendations** page.
- Post or like reviews.
- Post or reply to a comment.

Personal Profile

The personal profile serves as a hub for the user's activities. It displays:

- Account details and settings.
- Watchlists and favorite movies.
- The most recent review.
- Followed celebrities.
- Number of followers.

Searching for Movies

Using the search bar, users can enter a movie title, actor name, or TV series. The system provides up to ten results per category, leading to detailed pages where users can:

- View cast lists.
- Read and write reviews.
- Like and add movies to their watchlist.
- Movie and Actor Pages

Each movie page contains:

  - **Movie details**, such as release year, revenues, budget, ecc.
  - **A like button** to add or remove the movie from the liked movies.
  - **A watchlist button** to add or remove the movie from the watchlist.
  - **A Directors list** showing the people that directed the movie.
  - **An Actors list** showing the main actors that participated in the movie.

- **User Reviews**, where registered users can contribute with their opinions.
- **User posts,** where registered users can start or participate in a discussion.

The celebrity page includes:

- The celebrity's basic information.
- A filmography section displaying all the movies and TV shows featuring the celebrity, along with the specific roles they played.

Discovering New Movies:

The Home page personalizes recommendations with three sections:

- **Popular movies: list of movies sorted by imdb ratings.**
- **Popular celebrities: list of celebrities sorted by most frequently in popular movies.**
- **Recommended movies:** list of movies obtained through recommendation queries based on multiple parameters such as followed celebrities and users, liked reviews, ecc.
- **Recommended celebrities:** list of celebrities obtained through recommendation queries based on multiple parameters such as followed actors and followed users, liked movies, ecc.
- **Recommended users:** list of users obtained through recommendation queries based on multiple parameters such as followed actors and followed users, liked movies, ecc.

1.3 ADMIN MANUAL

Upon accessing the Admin Dashboard, administrators are presented with essential tools for managing the platform. The dashboard provides options for controlling content, user interactions, and performance metrics.

Administrators

Administrators, once authenticated, have full access to the advanced management features of the platform, allowing them to manage and modify the platform's content and settings.

Administrators have access to the following specific features:

- Manage Movie and TV Series Content: Administrators can add, modify, or remove content from the platform. They can also organize content by genre, release date, and other categories.
- View Movie and Actor Statistics from Users: Administrators can access statistics on how movies and actors are rated by users, including rankings based on interactions and received ratings. They can monitor user activity, including reviews and comments, to ensure proper content management.
- Check for databases inconsistencies: Administrators have the ability to check the platform's data, confronting the database instances to find inconsistencies between them, allowing him to then resolve these inconsistencies individually.

The **MOVIELAND Admin Dashboard** offers powerful tools for managing content and user engagement. By utilizing the features outlined in this manual, administrators can keep the platform fresh and engaging, while maintaining smooth operations.

2. DESIGN OVERVIEW

This chapter provides a detailed exploration of the key components and specifications that form the foundation of the MOVIELAND system. Understanding the architecture and the interactions between various elements is crucial for the development, implementation, and scalability of the system.

2.1 MAIN ACTORS

Identifying and understanding the main actors interacting with the MOVIELAND system is essential. These actors—whether they are users, external systems, or hardware components—play key roles in defining the system's use cases and ensuring that it meets the needs of all stakeholders. The main actors in the application are:

- **Unregistered Users:** Unregistered users are visitors who explore the platform without logging in or signing up. They cannot interact with the content in terms of liking, commenting, or reviewing.
- **Registered Users:** Registered users are individuals who have signed up for an account and gained access to the platform's full functionality.
- **Admins:** Admins are platform administrators responsible for managing and maintaining the website.

2.2 REQUIREMENTS

2.2.1 Functional Requirements

This section outlines the specific functionalities and features that the MOVIELAND system offers. These requirements are crucial for guiding development and ensuring the system aligns with user needs.

Unregistered Users

The application should allow unregistered users to:

- **Register** within the application.
- **Browse content**: View popular movies, TV series, and actors, including:
  - Movies with high average ratings.
  - Movies and TV shows with the most likes (both all-time and past week).
  - Actors with high user engagement.

- **Access detailed pages** for selected movies, TV series, or actors, including basic information such as plot, cast, director, and user reviews.
- **Search** for movies, TV series, or actors, with access to relevant pages upon selection.

Registered Users

Registered users should be able to:

- **Log in and log out to the platform.**
- **Manage their profile:** View and edit account details, liked movies, reviews, and followed users.

- **Interact with content:**
  - Like movies and TV series.
  - Write and like reviews.
  - Comment on content.

- **Search:** Look for movies, TV series, and actors, with the option to sort and filter by rating, genre, and release date.
- **Access recommendations:** Get personalized movie, celebrity and user suggestions based on preferences.
- Follow celebrities: Find and follow celebrities.
- **Follow other users:** Find and follow other users.

Admins

Admins should be able to:

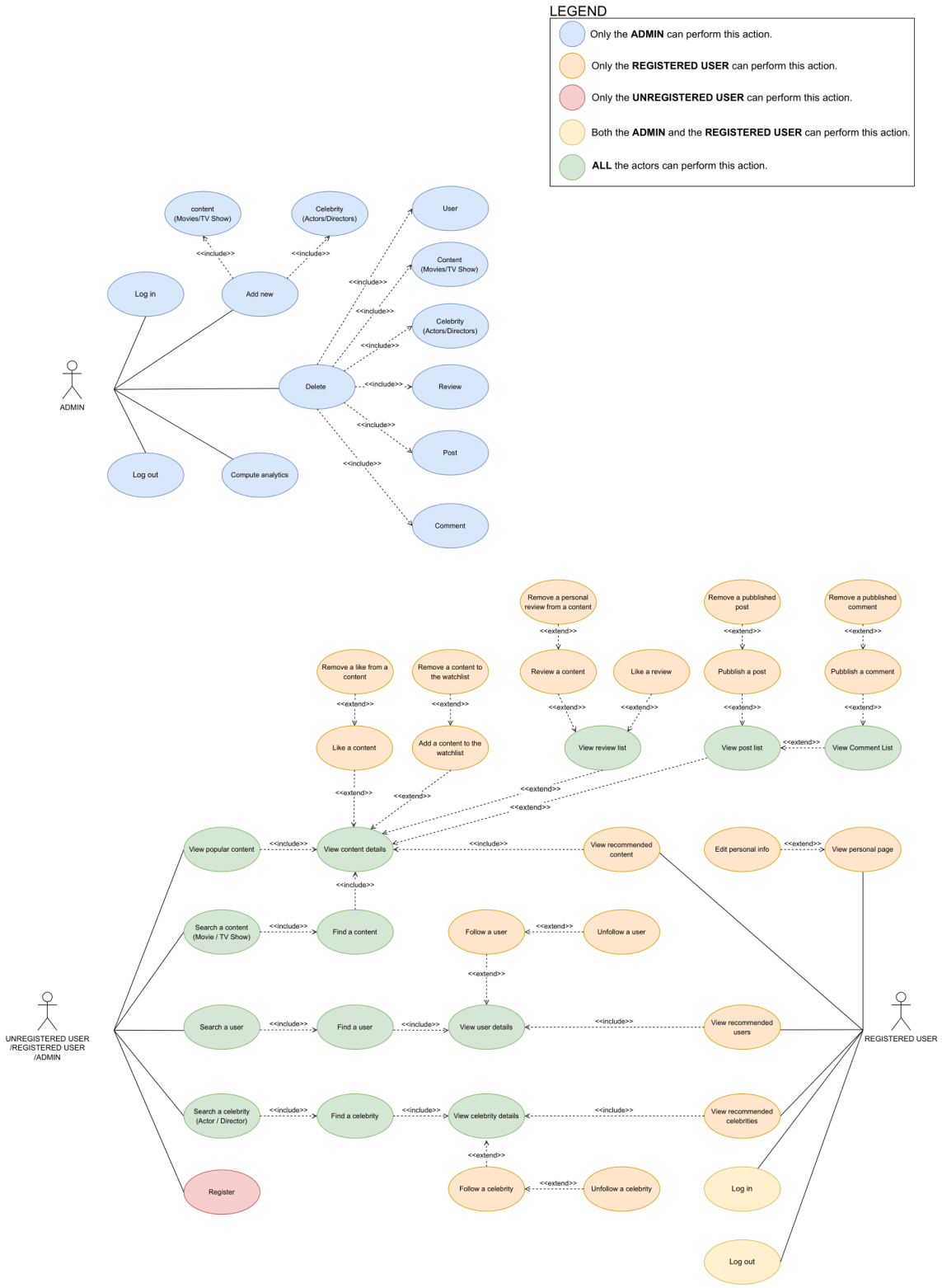
- **Manage content:** Add, update, or remove movies, TV series, and actors.
- Modify movie pages, including descriptions, ratings, and reviews.
- **View statistics:** Access data about movies, actors, and user interactions.
- **Generate reports:** Create reports on platform usage, content performance, and user activity.
- **Database management:** Initiate updates to keep content and user data current.
- **Monitor user interactions:** Review and moderate comments, reviews, and ratings.
- **Browse content:** Navigate the platform like unregistered users but with extra privileges to adjust content.

2.2.2 Non-functional Requirements

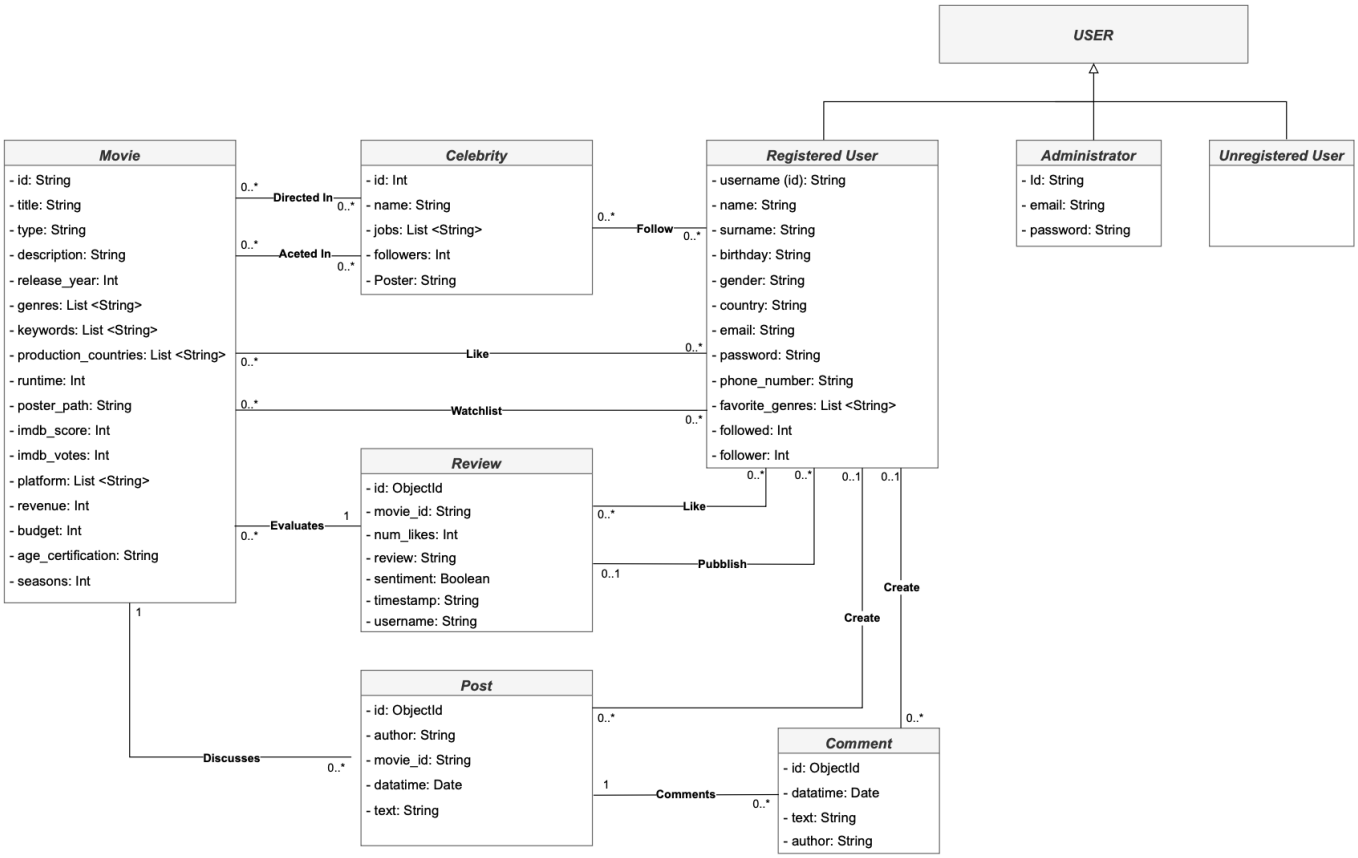
These requirements focus on the system's performance, reliability, scalability, and security, ensuring overall quality:

- **Web Application**: Accessible via modern web browsers.
- **High Availability**: The system should be available 24/7 with minimal downtime.
- **Scalability**: Easily scalable to manage increased traffic.
- **Fault Tolerance**: Minimize single points of failure.
- **Performance**: Ensure low latency and fast content loading.
- **Data Security**: Encrypt user passwords and use secure data exchanges.
- **Reliability**: The system should be resilient to failures, with regular backups.
- **Maintainability**: Use modular, well-documented code for ease of updates.

2.3 *USE CASE DIAGRAM*



2.4 *UML CLASS DIAGRAM*



3. DATA MODELING AND STRUCTURE

This section describes the datasets used and the data structure adopted to manage the main entities of our system. We used various data sources, including public datasets from Kaggle and data generated with AI, to populate the platform with titles, celebrities, users, posts, comments, and reviews.

We decided to manage our platform using MongoDB and Neo4j for handling all the information related to the main entities of our system and for improving search and recommendation queries respectively.

3.1 DATASET CREATION

This section explains the process of creating the datasets, their origin, data cleaning operations, and the final structure. The data were mainly sourced from Kaggle, except for the user data, which was generated internally.

3.1.1 Titles & Celebrities

The datasets used for titles and celebrities consisted of five sets, one for each platform. Each dataset was composed of two tables:

- **Titles**: Contains all the details about the titles available on the platform.
- **Credits**: Contains the roles of actors in the titles.

We unified these five datasets into one, adding a field for each title that indicates the platforms on which the title is available (since it may be available on more than one platform). We also created two arrays for each title:

- **Actors**: Containing the top 5 actors in the movie.
- **Directors**: Containing the directors of the movie.

For celebrities, we took the main information such as ID, name, and all the roles they have performed.

3.1.2 Users

User data was generated using Python, and passwords were encrypted with cypher.

3.1.3 Posts and Comments

The **Posts** and **Comments** data were obtained from a Kaggle dataset containing movie discussions (1GB), with only the **movie_id** missing. We used **Python** to:

- **Clean** the dataset by removing NaN values and dropping unnecessary columns.
- **Sample** 10% of the posts to reduce the dataset size.
- **Sort** posts by the number of comments to prioritize more discussed topics.
- **Prioritize popular movies** by sorting them based on IMDb votes, ensuring more posts were assigned to higher-rated films.

This streamlined approach helped manage and focus the dataset on the most relevant discussions.

3.1.4 Reviews

For reviews, we used a Kaggle dataset containing IMDb reviews, with the review text and sentiment (positive or negative). We added the **movie_id** to the reviews using the same method applied to posts and comments to assign a higher number of reviews to the most popular movies of all time.

3.2 *VOLUME*

This section provides an overview of the data volume for each entity in our system:

Entity	*Entries*
Titles	11,230
Celebrities	115,708
Users	9,736
Posts	50,550
Comments	372,900
Reviews	50,000

3.3 *MONGODB*

Handling large and continuously growing amounts of data requires a scalable, flexible database solution. MongoDB provides this scalability, flexibility, and high performance for both read and write operations.

MongoDB uses collections to store related data, and in this context, we have different collections representing movies, celebrities, users, posts, comments, and reviews. Below is an organized breakdown of the MongoDB collections used to store data related to these entities:

Movies Collection

The **Movies** collection stores information about individual films. Each document in this collection represents a single movie, including metadata such as title, description, genre, and other related data, like reviews and cast.

Document Structure


```
{
  "_id": "tt0120338",
  "title": "Titanic",
  "description": "Rose tells the story of Titanic's journey and its tragic end.",
  "release_year": 1997,
  "genres": ["drama", "romance"],
  "keywords": ["epic", "shipwreck", "forbidden love", "tragedy", "disaster", "historical fiction"],
  "production_countries": ["US"],
  "runtime": 194,
  "poster_path": "https://image.tmdb.org/t/p/w500/9xjZS2rlVxm8SFx8kPC3aIGCOYQ.jpg",
  "imdb_score": 79,
  "imdb_votes": 1133692,
  "platform": ["AMAZON PRIME", "PARAMOUNT+"],
  "actors": [
    {"id": 616583, "name": "Aaron James Cash", "role": "Dancer"},
    {"id": 117865, "name": "Alexandra Boyd", "role": "First Class Woman (uncredited)"}
  ],
  "reviews": [
    {
      "id": 57794,
      "review": "A weak movie saved only by Goldie Hawn's charm. The plot fails midway.",
      "sentiment": "negative",
      "username": "donald.miller6138",
      "timestamp": "2024-03-31T18:16:53"
    }
  ],
  "revenue": 2264162353,
  "budget": 200000000,
  "age_certification": "PG-13",
  "seasons": null
}
```


In this document, we outline all the details related to the movie, such as the title, description, genres, runtime, keywords, actors, directors, reviews, and more. The arrays in this document represent relationships between the movie and celebrities or users. Specifically, we've chosen to store data about the top 5 actors (if available) associated with the movie. For each actor, we store their ID, name, and the character they portrayed. This allows us to display basic information about the celebrity. When displaying movie reviews, we store only the most recent reviews, including the review ID, review text, sentiment (positive or negative), the username of the reviewer, and the timestamp of the review.

Celebrities Collection

The **Celebrity** collection is designed to store detailed information about celebrities, such as actors, directors, and other key industry professionals. It includes general data like the celebrity's name and profile picture, as well as a list of their roles throughout their career. This list, stored in the **"jobs"** array, tracks each project the celebrity has worked on, including the type of role, the movie title, character names, and unique job identifiers. This structure allows us to effectively

manage and display a celebrity's career history while maintaining the relationships between the individuals and their projects.

Document Structure


```
{
  "_id": "11",
  "name": "Katherine Kelly",
  "jobs": [
    {
      "role": "ACTOR",
      "movie_id": "tt3093354",
      "movie_title": "The Last Witch",
      "character": "Alice Lister",
      "job_id": "11_job1"
    },
    {
      "role": "ACTOR",
      "movie_id": "tt5431890",
      "movie_title": "Official Secrets",
      "character": "Jacqueline Jones",
      "job_id": "11_job2"
    }
  ],
  "followers": 0,
  "poster": "https://image.tmdb.org/t/p/w500/gh8uNkRvuC6cHpnXKtaT2iKg0Ht.jpg"
}
```


In the **Celebrity** collection, we store general information about the celebrity and their roles throughout their career. The **"jobs"** array represents the relationship between the celebrity and the movies they have worked on. Each element in the **"jobs"** array includes the following fields:

- **role**: The type of role the celebrity has in the movie (e.g., ACTOR or DIRECTOR).
- **movie_id**: The unique identifier for the movie.
- **movie_title**: The title of the movie.
- **character**: The name of the character played by the celebrity (if applicable).
- **job_id**: A unique identifier for each job. This is generated in the format **celebrityId_jobNumber**. For example, if the celebrity has a job with ID 32 and it is their third job, the **job_id** would be **32_job3**.

Users Collection

In the **Users** collection, we store detailed user information such as their account data, contact details, preferences, and their interactions with movies and celebrities. Specifically, we save only a summary of the most recent data for efficient access when displaying a user's personal page. This includes:

- A list of the most recently liked movies.
- A list of movies recently added to the user's watchlist.
- A list of the most recently followed celebrities.
- A record of the most recent reviews posted by the user.

By only storing a brief summary of this information directly in the user document, we avoid overloading the database with large, constantly growing arrays. This ensures that we can quickly retrieve essential data without the need for complex queries or excessive load times, providing an efficient user experience.

Document Structure


```
{
  "_id": "aaron.aguilar5925",
  "email": "aaron.aguilar@hotmail.com",
  "name": "Aaron",
  "surname": "Aguilar",
  "password": "1c8d616dacc6cb8d62c53a2fcecd49415db84928bba46ed97923b28d34c87fef",
  "country": "BR",
  "phone_number": "+55 7230813575",
  "favorite_genres": [
    "war",
    "horror",
    "european",
    "scifi"
  ],
  "gender": "Male",
  "birthday": "1996-06-15",
  "liked_movies": [
    {
      "film_id": "tt0071853",
      "title": "Monty Python and the Holy Grail",
      "poster": "https://image.tmdb.org/t/p/w500/xZ0rC0kuIsjk6RncpNK73kwstEP.jpg"
    }
  ],
  "watchlist": [
    {
      "film_id": "tt0046183",
      "title": "Peter Pan",
      "poster": "https://image.tmdb.org/t/p/w500/cN4Tb2WDhwJMlYLU9vKctunx4rg.jpg"
    }
  ],
  "followed_celebrities": [
    {
      "person_id": 725360,
      "name": "Paula Luiz",
      "poster": "https://image.tmdb.org/t/p/w500/4l7USTRnX02cWYFuIbpUWfw6Yhm.jpg"
    }
  ],
  "recent_review": {
    "review_id": {
      "$oid": "67a72d790acd40960a61f2fa"
    },
    "movie_title": "Open Season",
    "sentiment": "negative",
    "content": "I'd have to agree with ..."
  },
  "follower": 11,
  "followed": 10
}
```


By storing only, the most relevant and recent data directly in the **Users** collection, we ensure that the application performs efficiently and provides a quick overview of the user's activity. The **liked_movies**, **watchlist**, **followed_celebrities**, and **recent_review** arrays enable us to display key information about the user's preferences, interactions, and content engagement, without overwhelming the database with unnecessary large datasets. This approach strikes a balance between

usability and performance, ensuring a smooth user experience when accessing the personal page.

Posts Collection

The "Posts" collection is used to store posts created by users within the platform. Each post is associated with a specific movie and contains the text written by the user, along with other metadata such as the author and the date of publication. This structure allows us to track user contributions and maintain the connection between the content and the movies to which they refer.

Document Structure


```
{
  "_id": {
    "$oid": "67adf4ba630e2552c27d0f84"
  },
  "author": "james.schneider9833",
  "datetime": {
    "$date": "2023-06-10T10:39:08.000Z"
  },
  "movie_id": "tt0110413",
  "text": "This old movie theater in Riverdale MD still has posters up"
}
```


In this collection, post data includes basic information about the author, the movie, and the content of the message, enabling efficient management of user interactions. The structure provides easy access to the details related to the post, facilitating the display and management of content generated by the community.

Comments Collection

The "Comments" collection stores user-generated comments related to specific posts. Each comment document includes the author's username, the datetime of the comment, the post it is associated with (via the post_id field), and the text of the comment itself. This approach ensures that comments are stored separately from the posts collection, enabling more efficient retrieval when needed. We store the minimal information required for displaying and managing comments related to each post.

Document Structure


```
{
  "_id": {
    "$oid": "67adf7d1630e2552c287e3a6"
  },
  "author": "kiara.buck267",
  "datetime": {
    "$date": "2023-06-09T23:56:36.000Z"
  },
  "post_id": {
    "$oid": "67adf4ba630e2552c27d0fb6"
  },
  "text": "No cast members harmed, only underlings. The big names are fine."
}
```

In summary, the "Comments" collection is designed to hold the necessary details about each comment without unnecessarily embedding them within the post documents. By using the post_id reference, we can efficiently retrieve the comments for each post when required. This design helps to keep the database well-structured and optimized for performance, particularly when dealing with large amounts of comments across multiple posts.

Reviews Collection

The "Reviews" class manages user feedback for movies, allowing users to share opinions on films they have watched. It includes key details like review text, sentiment (positive or negative), timestamp, and the number of likes. This class helps provide insights into user preferences, movie ratings, and overall reception. By storing sentiment and likes, it enhances user interaction and helps users discover popular movies and track opinions. To optimize performance and avoid scalability issues, the system efficiently handles large review volumes while maintaining access to the full review text.

Document Structure


```
{
  "_id": {
    "$oid": "67a72d780acd40960a619e15"
  },
  "movie_id": "tt3640424",
  "num_likes": 32,
  "review": "Oz is a brutal, violent show set in a prison, offering raw, dark content that chall
  "sentiment": true,
  "timestamp": "2024-03-08T10:50:51",
  "username": "david.curtis5210"
}
```


We chose to store only the number of "likes" rather than the full list of users who liked the review to avoid the potential growth of a massive array over time. This could cause performance and scalability issues, as it could exceed MongoDB's maximum document size limit. To handle this more efficiently, we decided to store the list of likes in Neo4j, while the "num_likes" field is updated each time a user likes the review. This approach helps optimize data management and improves the overall performance of the application.

3.3.1 Indexes

MOVIE SEARCH – simple

Single word (no indexes)	*Single word (no indexes)*

db.Movies.aggregate([
{
$match: {
$and: [
{
$or: [
{ title: { $regex: "love", $options: "i" } },
{ keywords: { $regex: "love", $options: "i" } }
]
},
{ type: "MOVIE" }
] } },
{
$sort: { imdb_score: -1 }
},
{
$project: {
_id: 1,
title: 1,
release_year: 1,
poster_path: 1,
imdb_score: 1
} }
]).explain("executionStats");
	db.Movies.aggregate([
	{
	$match: {
	$and: [
	{
	$or: [
	{ title: { $regex: "star.*wars", $options: "i" } },
	{ keywords: { $regex: "star.*wars", $options: "i" } }
	]
	},
	{ type: "MOVIE" }
	] } },
	{
	$sort: { imdb_score: -1 }
	},
	{
	$project: {
	_id: 1,
	title: 1,
	release_year: 1,
	poster_path: 1,
	imdb_score: 1
	} }
	]).explain("executionStats");

executionStats: { executionSuccess: true, nReturned: 573, executionTimeMillis: 43, totalKeysExamined: 0, totalDocsExamined: 11230,	executionStats: { executionSuccess: true, nReturned: 14, executionTimeMillis: 42, totalKeysExamined: 0, totalDocsExamined: 11230

$db.Movies.createIndex(\{ title: "text", keywords: "text" \}, \{ weights: \{ title: 10, keywords: 2 \} \})$

$db.Movies.createIndex(\{ imdb_{score}: 1 \})$

We create a text index on title and keywords fields, for improving the performance of the query search based on one or more words that appear in the title or in the keywords. Moreover, we use a system of weights to give more importance to the correspondence of the word with the title because the user is more likely to search a movie for its title.

Single word (with index on title & keywords)	***Multiple words (with index on title & keywords)***

db.Movies.aggregate([
{
$match: {
$text: { $search: "love" },
type: "MOVIE"
} },
{
$addFields: {
score: { $meta: "textScore" }
} },
{
$sort: { score: -1, imdb_score: -1 }
},
{
$project: {
_id: 1,
title: 1,
release_year: 1, poster_path: 1,
imdb_score: 1, score: 1 } }
]).explain("executionStats"); }
	db.Movies.aggregate([
	{
	$match: {
	$text: { $search: "star wars" },
	type: "MOVIE"
	} },
	{
	$addFields: {
	score: { $meta: "textScore" }
	} },
	{
	$sort: { score: -1, imdb_score: -1 }
	},
	{
	$project: {
	_id: 1,
	title: 1,
	release_year: 1, poster_path: 1,
	imdb_score: 1, score: 1 } }
	]).explain("executionStats"); }

executionStats: { executionSuccess: true, nReturned: 543, executionTimeMillis: 15, totalKeysExamined: 543, totalDocsExamined: 1086,	executionStats: { executionSuccess: true, nReturned: 417, executionTimeMillis: 14, totalKeysExamined: 436, totalDocsExamined: 842,

MOVIE SEARCH - with filters

Single word (no indexes)	*Multiple words (no indexes)*

db.Movies.aggregate([
{
"$match": {
"$and": [
{ "type": "MOVIE" },
{ "genres": { "$all": ["drama"] } },
{ "release_year": 1954 },
{ "platform": "AMAZON PRIME" },
{ "production_countries": "US" },
{ "age_certification": "PG" },
{ "imdb_score": { "$gte": 5 } },
{ "imdb_votes": { "$gte": 1000 } },
{
"$or": [
{ "title": { "$regex": "love", "$options": "i" } },
{ "keywords": { "$regex": "love", "$options": "i" } }
] }] } },
{ "$sort": { "imdb_score": -1 } },
{
"$project": {
"_id": 1,
"title": 1,
"release_year": 1,
"poster_path": 1,
"imdb_score": 1
} }])
	db.Movies.aggregate([
	{
	"$match": {
	"$and": [
	{ "type": "MOVIE" },
	{ "genres": { "$all": ["drama"] } },
	{ "release_year": 1954 },
	{ "platform": "AMAZON PRIME" },
	{ "production_countries": "US" },
	{ "age_certification": "PG" },
	{ "imdb_score": { "$gte": 5 } },
	{ "imdb_votes": { "$gte": 1000 } },
	{
	"$or": [
	{ "title": { "$regex": "star.*wars", "$options": "i" } },
	{ "keywords": { "$regex": "star.*wars", "$options": "i" } }
	] }] } },
	{ "$sort": { "imdb_score": -1 } },
	{
	"$project": {
	"_id": 1,
	"title": 1,
	"release_year": 1,
	"poster_path": 1,
	"imdb_score": 1
	} }])

executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 38, totalKeysExamined: 0, totalDocsExamined: 11230,	executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 38, totalKeysExamined: 0, totalDocsExamined: 11230,

Single word (with index on title & keywords)	*Multiple words (with index on title & keywords)*

db.Movies.aggregate([
{
"$match": {
"$and": [
{ "type": "MOVIE" },
{ "genres": { "$all": ["drama"] } },
{ "release_year": 1954 },
{ "platform": "AMAZON PRIME" },
{ "production_countries": "US"},
{ "age_certification": "PG" },
{ "imdb_score": { "$gte": 5 } },
{ "imdb_votes": { "$gte": 1000 } },
{ "$text": { "$search": "love" } }
] } },
{
"$addFields": {
score: { "$meta": "textScore" } } },
{
"$sort": {
score: -1,
"imdb_score": -1 } },
{
"$project": {
"_id": 1,
"title": 1,
"release_year": 1,
"poster_path": 1,
"imdb_score": 1
} }]).explain("executionStats");
	db.Movies.aggregate([
	{
	"$match": {
	"$and": [
	{ "type": "MOVIE" },
	{ "genres": { "$all": ["drama"] } },
	{ "release_year": 1954 },
	{ "platform": "AMAZON PRIME" },
	{ "production_countries": "US"},
	{ "age_certification": "PG" },
	{ "imdb_score": { "$gte": 5 } },
	{ "imdb_votes": { "$gte": 1000 } },
	{ "$text": { "$search": "star wars" } }] } },
	{
	"$addFields": {
	score: { "$meta": "textScore" } } },
	{
	"$sort": {
	score: -1,
	"imdb_score": -1 } },
	{
	"$project": {
	"_id": 1,
	"title": 1,
	"release_year": 1,
	"poster_path": 1,
	"imdb_score": 1
	} }]).explain("executionStats");

executionStats: { executionSuccess: true, nReturned: 0, executionTimeMillis: 12, totalKeysExamined: 543, totalDocsExamined: 1086,	executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 7, totalKeysExamined: 436, totalDocsExamined: 842,

CELEBRITY SEARCH

db.Celebrities.createIndex({ name: "text", character: "text" },
{ weights: { name: 3, character: 1 } })

Search by name or character (no index)	***Search by name or character (with index on name)***

db.Celebrities.aggregate([
{
$unwind: "$jobs"
},
{
$match: {
$or: [
{ "name": { $regex: "Cecile", $options: "i" } },
{ "jobs.character": { $regex: "Cecile", $options: "i" } }
]
}
},
{
$group: {
_id: "$_id",
name: { $first: "$name" },
jobs: { $push: "$jobs" },
followers: { $first: "$followers" },
poster: { $first: "$poster" }
}
},
{
$limit: 10
}
]).explain("executionStats");
	db.Celebrities.aggregate([
	{
	$match: {
	$text: { $search: "Cecile" }
	} },
	{
	$addFields: {
	score: { $meta: "textScore" } } },
	{
	$unwind: "$jobs" },
	{
	$group: {
	_id: "$_id",
	name: { $first: "$name" },
	jobs: { $push: "$jobs" },
	followers: { $first: "$followers" },
	poster: { $first: "$poster" },
	score: { $first: "$score" },
	imdb_score: { $first: "$imdb_score" } } },
	{
	$sort: {
	score: -1,
	imdb_score: -1
	} },
	{
	$limit: 10
	}]).explain("executionStats");

executionStats: { executionSuccess: true, nReturned: 115708, executionTimeMillis: 240, totalKeysExamined: 0, totalDocsExamined: 115708	executionStats: { executionSuccess: true, nReturned: 27, executionTimeMillis: 0, totalKeysExamined: 27, totalDocsExamined: 27,

USERS SEARCH

```
db.Users.createIndex( {
name: "text",
surname: "text",
username: "text"},
{weights: { name: 5, surname: 10, username: 7 } })
db.users.createIndex({ country: 1 })
```

No indexes	***With indexes (only name, surname,username)***

db.Users.find({
name: "Aaron",
surname: "Aguilar",
country: "BR"
}).explain("executionStats");
	db.Users.aggregate([
	{
	$match: {
	country: "BR",
	$text: { $search: "Aaron aguilar" } }
	},
	{
	$sort: { score: { $meta: "textScore" } }
	},
	{
	$limit: 5 }
	]).explain("executionStats");

executionStats: { executionSuccess: true, nReturned: 1, executionTimeMillis: 24, totalKeysExamined: 0, totalDocsExamined: 9736,	executionStats: { executionSuccess: true, nReturned: 3, executionTimeMillis: 9, totalKeysExamined: 47, totalDocsExamined: 92,

With indexes (name, surname,username, country)
db.Users.aggregate([{ $match: { country: "BR", $text: { $search: "Aaron aguilar" } } }, { $sort: { score: { $meta: "textScore" } } }, { $limit: 5 }]).explain("executionStats");
executionStats: { executionSuccess: true, nReturned: 3, executionTimeMillis: 0, totalKeysExamined: 47, totalDocsExamined: 92,

POSTS SEARCH

db.Posts.createIndex({ movie_id: 1 })

db.Posts.createIndex({ author: 1 })

Search by movie_id (no index)	*Search by movie_id (with index on movie_id)*
db.Posts.find({ movie_id: "tt0110413" }).explain("executionStats");	db.Posts.find({ movie_id: "tt0110413" }).explain("executionStats")
executionStats: { executionSuccess: true, nReturned: 230, executionTimeMillis: 56, totalKeysExamined: 0, totalDocsExamined: 50550,	executionStats: { executionSuccess: true, nReturned: 230, executionTimeMillis: 6, totalKeysExamined: 230, totalDocsExamined: 230,

COMMENTS SEARCH

db.Comments.createIndex({ post_id: 1 })

db.Comments.createIndex({ author: 1 })

Search by post_id (no index)	***Search by post_id (with index on post_id)***
db.Comments.find({ post_id: ObjectId("67adf4ba630e2552c27d0fb6") }).explain("executionStats");	db.Comments.find({ post_id: ObjectId("67adf4ba630e2552c27d0fb6") }).explain("executionStats");
executionStats: { executionSuccess: true, nReturned: 0, executionTimeMillis: 119, totalKeysExamined: 0, totalDocsExamined: 372900,	executionStats: { executionSuccess: true, nReturned: 53, executionTimeMillis: 9, totalKeysExamined: 53, totalDocsExamined: 53,

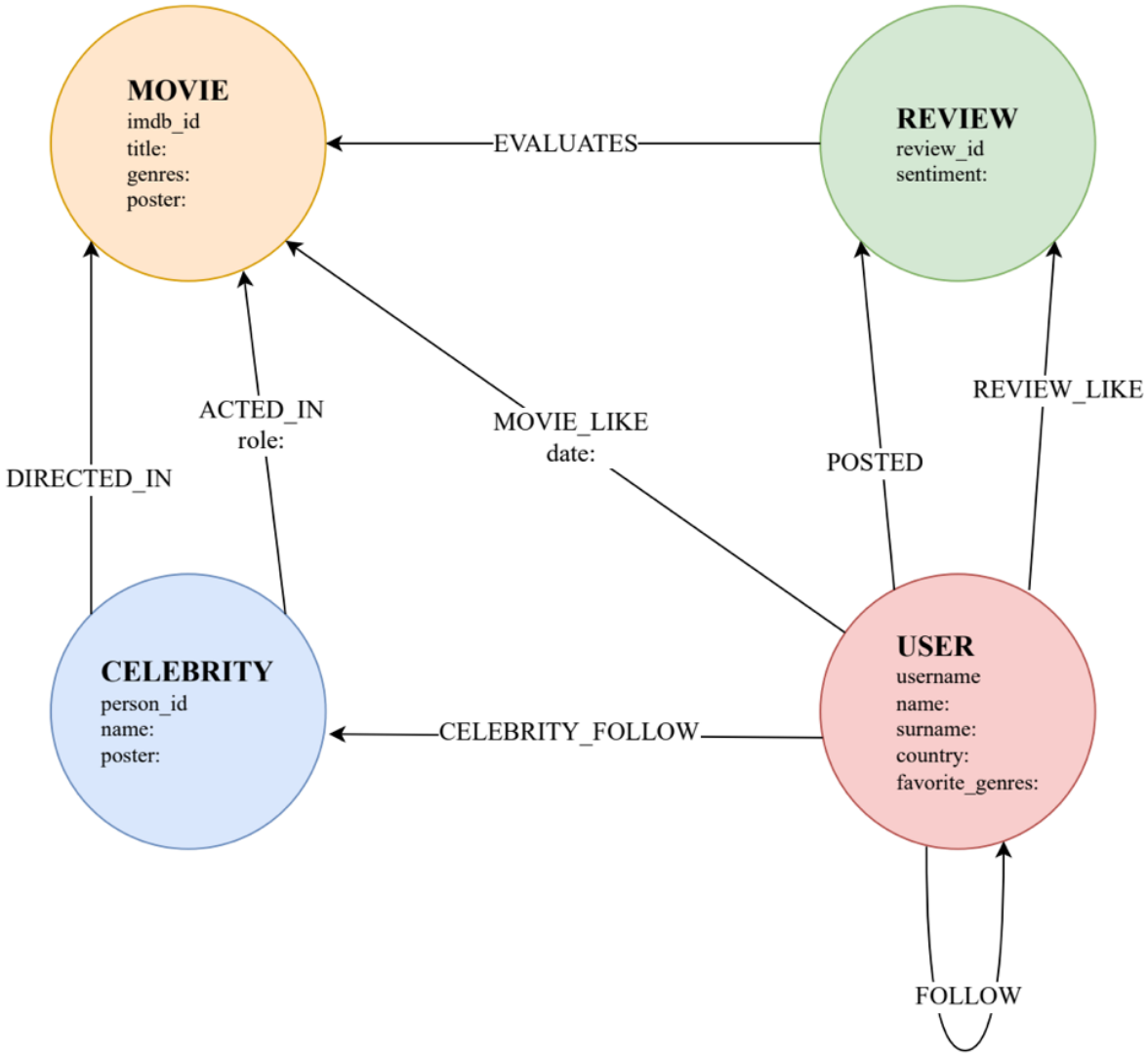
REVIEWS SEARCH

db.Reviews.createIndex({ movie_id: 1 })

Search by movie_id (no index)	***Search by movie_id (with index on movie_id)***
db.Reviews.find({ movie_id: "tt3640424" }).explain("executionStats");	db.Reviews.find({ movie_id: "tt3640424" }).explain("executionStats");
executionStats: { executionSuccess: true, nReturned: 20, executionTimeMillis: 140, totalKeysExamined: 0, totalDocsExamined: 50000,	executionStats: { executionSuccess: true, nReturned: 20, executionTimeMillis: 3, totalKeysExamined: 20, totalDocsExamined: 20,

3.4 *NEO4J*

Neo4j plays a crucial role in the recommendation system of our project, which suggests movies, celebrities, and other registered users to each individual. These recommendations are based on the relationships between users, movies, celebrities, and other users. To manage these complex connections efficiently and provide accurate suggestions, we chose to leverage Neo4j, a graph database that is particularly well-suited for handling such dynamic relationships. Its graph structure allows for fast traversal queries and real-time updates to recommendations as user interactions evolve. This ensures that users always receive personalized and up-to-date recommendations.



3.4.1 Concepts

Neo4j uses graph theory to model relationships and entities, making it an ideal tool for recommendation engines where entities are strongly interconnected. In our case, the primary entities are **Movies**, **Celebrities**, **Users**, and **Reviews**, and these entities are connected by various relationships, enabling us to provide insights and personalized recommendations based on users' interactions with each other, their favorite genres, and the content they engage with.

3.4.2 Entities

Movie

- **title**: The name of the movie.
- **imdb_id**: The unique identifier for the movie in IMDB.
- **genres**: The movie's genres, used to match with the user's favorite genres.

We store only essential movie data, such as the title and genres, which are necessary for query results and matching user preferences. The IMDB ID is also stored for precise movie identification.

Celebrity

- **person_id**: Unique identifier for the celebrity.
- **name**: The name of the celebrity (actor, director, etc.).
- **poster**: The image of the celebrity to visually represent them in query results.

Celebrity information is kept minimal, with just the name and poster for display purposes, helping personalize recommendations based on user interests.

User

- **username**: The unique identifier for the user.
- **name**: The user's first name.
- **surname**: The user's last name.
- **country**: The country where the user resides, allowing location-based recommendations.
- **favourite_genres**: The genres the user enjoys most, used to suggest films that match their personal tastes.

The "country" field allows us to provide recommendations based on geographical location, while "favourite_genres" personalizes suggestions according to the user's preferences.

Review

- **review_id**: The unique identifier for a specific review.
- **sentiment**: The sentiment of the review (positive or negative), to suggest movies that the user is more likely to enjoy based on past feedback.

In Neo4j, we store the relationship between reviews and users to suggest movies based on liked reviews. For example, if User 1 liked a review from User 2 on Movie Y, we may recommend Movie X, which was reviewed by User 2. This creates a personalized experience by leveraging the interactions between users and their review preferences.

3.4.3 Relationships

In Neo4j, the relationships between entities are fundamental for the recommendation system, enabling us to provide personalized suggestions based on users' interactions with movies, celebrities, and other users. These relationships define the connections and allow for meaningful traversal queries that dynamically update based on user actions. Below are the main relationships used in our system:

ACTED_IN

This relationship connects a **Celebrity** (specifically an actor) with a **Movie**. It indicates that the actor has performed in that particular movie. This relationship is key to recommending movies based on the movies that a user has enjoyed featuring specific actors or celebrities they follow.

Example:

(Celebrity)-[:ACTED_IN]->(Movie)

"Tom Kane" -[:ACTED_IN]-> "Star Wars"

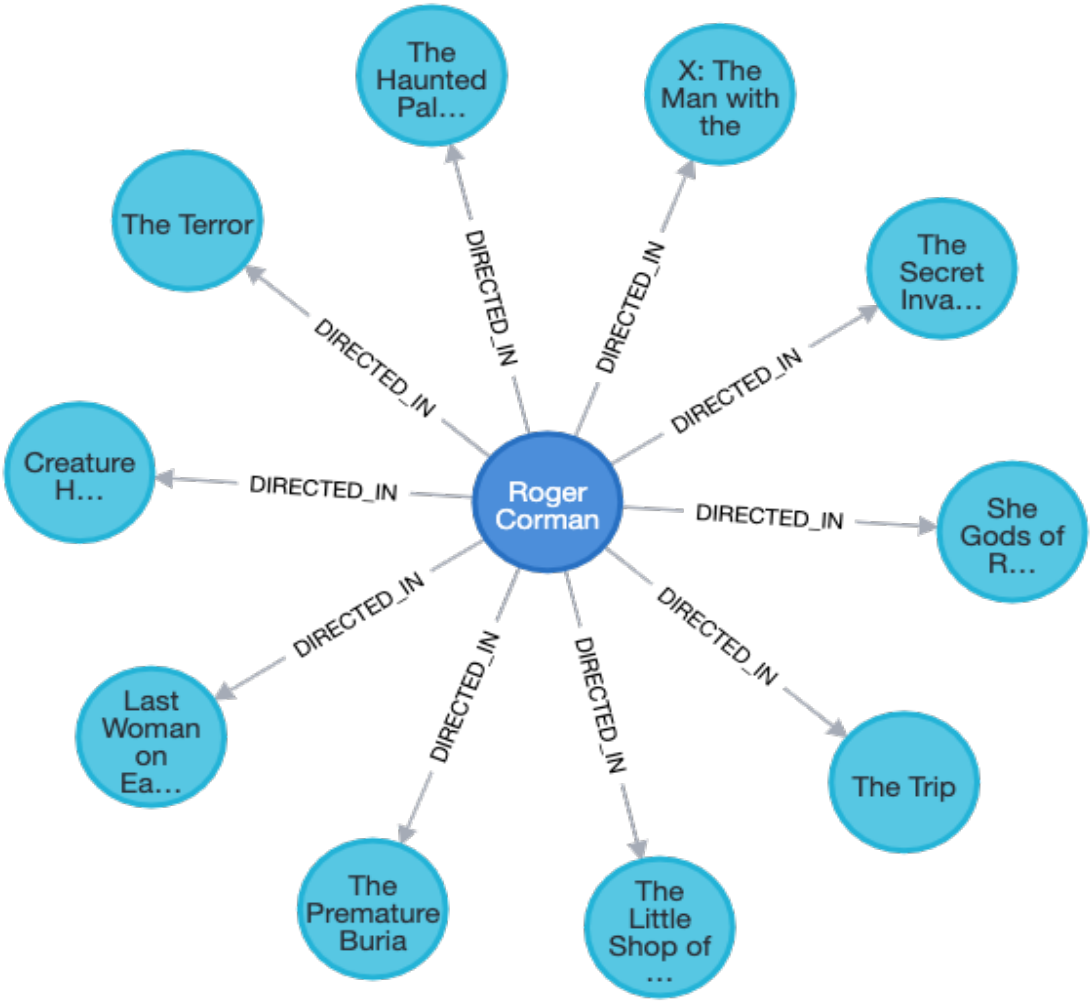
DIRECTED_IN

This relationship connects a **Celebrity** (in this case, a director) with a **Movie**. It shows which movies a particular director has been involved in. It helps users discover films based on directors they admire, offering director-centric recommendations.

Example:

Celebrity)-[:DIRECTED_IN]->(Movie)

"Roger Corman Nolan" -[:DIRECTED_IN]-> "The Trip"



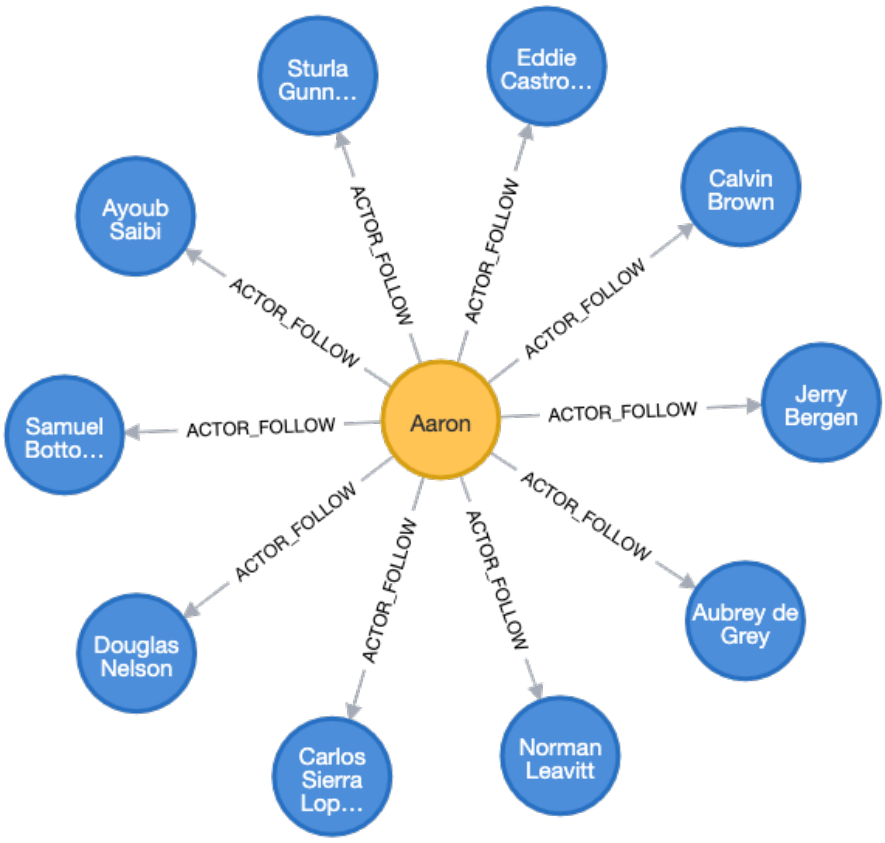
ACTOR_FOLLOW

This relationship tracks which **users** are following specific **celebrities**. It also stores the **date** when the user began following the celebrity, allowing for historical tracking of user preferences and engagement with celebrities.

Example:

(User)-[:ACTOR_FOLLOW {since: "2022-01-01"}]->(Celebrity)

"Aaron"-[:ACTOR_FOLLOW {since: "2023-05-15"}]-> "Calvin Brown"



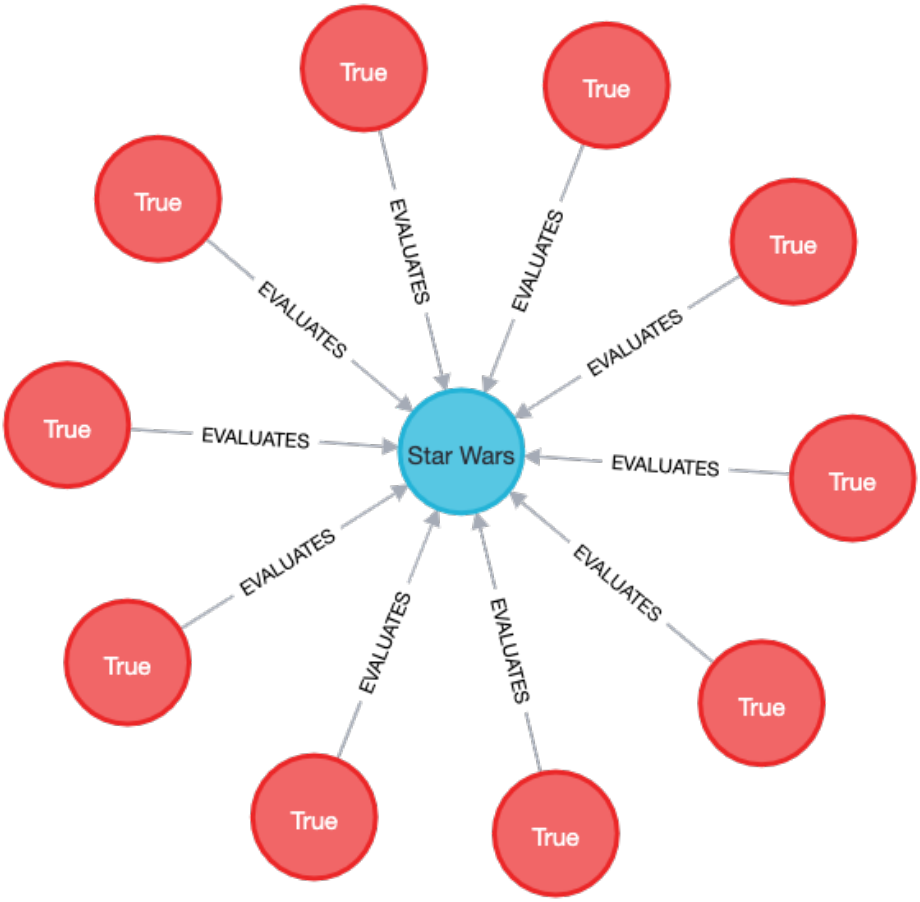
EVALUATES

The **EVALUATES** relationship links a **User** to the **Movie** they have reviewed. It indicates that the user has provided a review for that movie and can be used to recommend movies similar to those the user has evaluated highly.

Example:

(User)-[:EVALUATES]->(Movie)

"Aaron" -[:EVALUATES]-> "Star Wars"



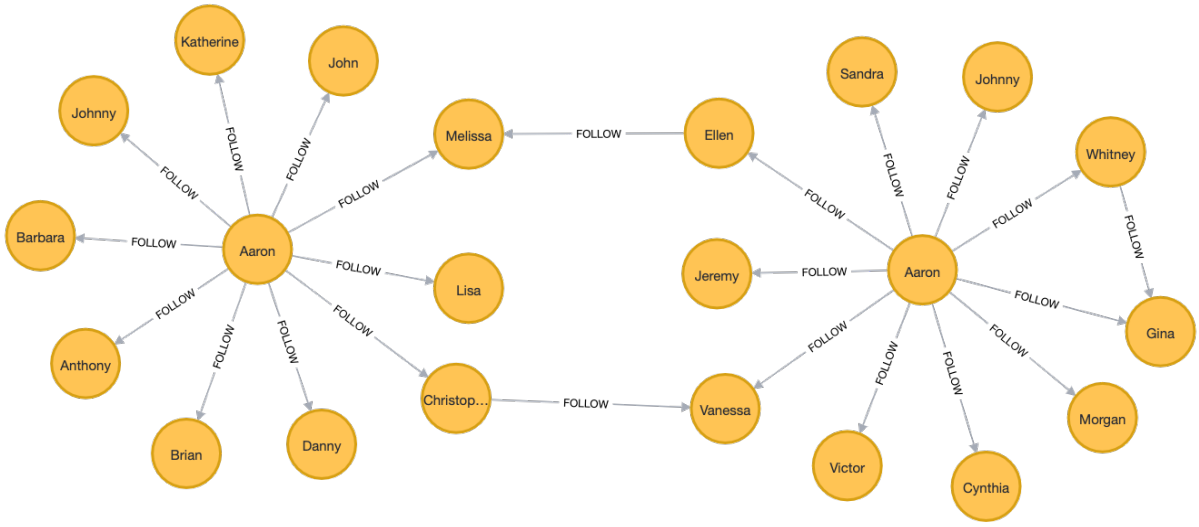
FOLLOW

This relationship connects users to other users they follow. It is not bidirectional, meaning that while User A may follow User B, User B does not necessarily follow User A back. This relationship is useful for providing social-based recommendations, such as suggesting movies that a user's followed friends have rated highly.

Example:

(User)-[:FOLLOW]->(User)

"Aaron" -[:FOLLOW]-> "Melissa"



MOVIE_LIKE

This relationship tracks when a user likes a movie. It is an important relationship for identifying the movies a user is interested in, which can be used for recommending similar films based on the user's likes.

Example:

(User)-[:MOVIE_LIKE]->(Movie)

"Aaron" -[:MOVIE_LIKE]-> "Pusher 3"

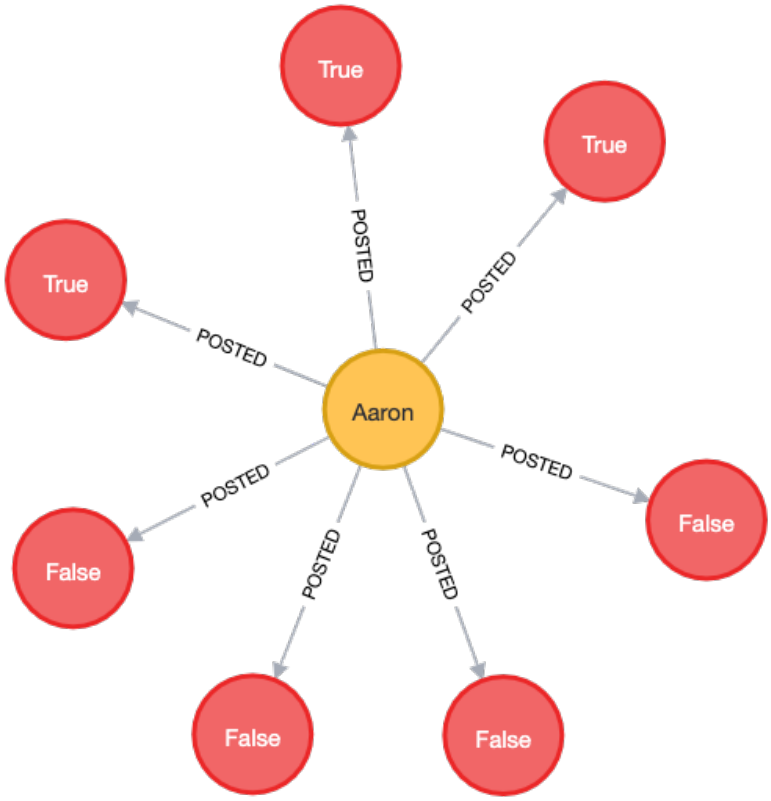
POSTED

The POSTED relationship indicates that a user has posted a review for a movie. This allows the system to track the reviews users have contributed and recommend films based on reviews that users have shared about other movies.

Example:

(User)-[:POSTED]->(Review)

"Aaron" -[:POSTED]-> "Positive"



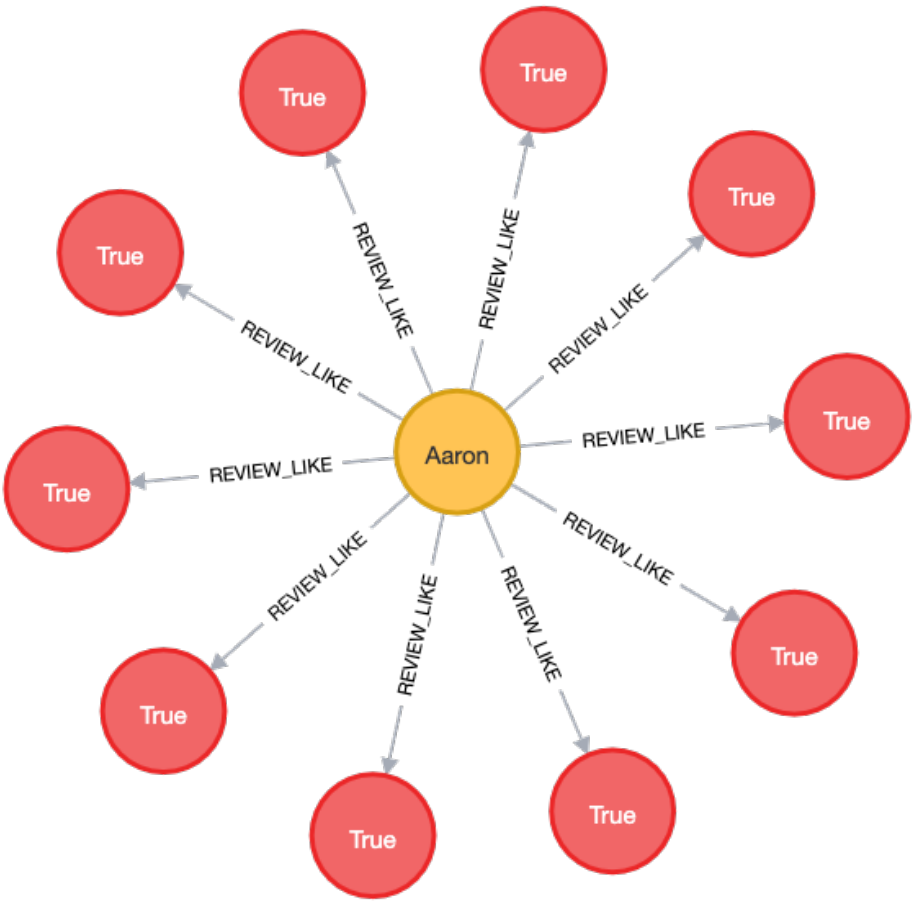
REVIEW_LIKE

This relationship tracks when a user likes a specific review posted by another user. It is used to identify reviews that resonate with a particular user, and it can influence future recommendations based on the reviews a user has found appealing.

Example:

(User)-[:REVIEW_LIKE]->(Review)

"Chris" -[:REVIEW_LIKE]-> "Positive"



4. IMPLEMENTATION

This section describes the structure of the *Movieland* project, which follows the MVC (Model-View-Controller) pattern to ensure clear separation of concerns and better maintainability.

The application is organized into the following packages:

- **Controller (it.unipi.movieland.controller)**: handles HTTP requests and coordinates interactions between the service layer and the clients.
- **Service (it.unipi.movieland.service)**: implements the business logic and provides centralized functionality.
- **Repository (it.unipi.movieland.repository)**: manages access to and handling of persistent data.
- **Model (it.unipi.movieland.model)**: defines the core domain entities of the application.
- **DTO (it.unipi.movieland.dto)**: transfers data between different layers while keeping domain logic separated.
- **Exception (it.unipi.movieland.exception)**: manages exceptions to ensure clear and centralized error handling.
- **Utils (it.unipi.movieland.utils)**: contains utility classes and methods that can be reused across the application.

4.1 MVC DESIGN PATTERN

We have chosen to adopt the **Model-View-Controller (MVC)** architectural pattern for the development of the **Movieland** project. This choice was motivated by the need to maintain a clear separation of responsibilities among the different components of the application, improving code readability, maintainability, and collaboration within the development team.

- **Model**: handles the data and business logic, representing the domain entities and their relationships. In our case, the data is stored in MongoDB and Neo4j, and the model classes reflect the structure of the documents and nodes in these databases.
- **Controller**: manages HTTP requests, orchestrating interactions between the **Model** and the **Service**. Controllers validate incoming data, delegate logic to the services, and return clear and consistent responses.
- **Service**: implements the application's business logic, centralizing the management of operations between the various databases (MongoDB and

Neo4j). Services execute complex operations, combine data from different sources, and provide the necessary responses to the controllers.

Adopting this pattern has allowed us to maintain a well-organized, scalable, and easily extendable architecture, simplifying the integration of new features and the management of existing code.

4.2 PROJECT STRUCTURE

4.2.1 it.unipi.movieland.dto

The **DTO** package contains classes that facilitate **data transfer** between the backend and clients. DTOs prevent direct exposure of database entities, securing internal structures and reducing the amount of transmitted data. A DTO can aggregate information from MongoDB and Neo4j, returning a single object with specific fields. DTOs can be manually converted from entities or mapped using tools like MapStruct. Using DTOs improves maintainability, enhances security, and allows better control over API responses.

CLASS	FOLDER	DESCRIPTION
CelebrityMongoDTO	Celebrity	Data transfer object for Celebrity information in MongoDB
CelebrityNeo4jDTO	Celebrity	Data transfer object for Celebrity information in Neo4j
JobDTO	Celebrity	DTO for handling job-related data for celebrities
ActorDTO	Celebrity	DTO for actor-related data and information
CelebrityRecommendationsDTO	Celebrity	DTO for celebrity recommendations
CombinedPercentageDTO	Movie	DTO for combining percentages of movie ratings or views
CommentDTO	Comment	DTO for transferring comment data
GenreRecommendationsDTO	Movie	DTO for genre-based movie recommendations
ListIdDTO	User	DTO for handling user's movie list identifiers
MovieRecommendationsDTO	Movie	DTO for movie recommendations
PostActivityDTO	Post	DTO for post activity data and interactions

PostDTO	Post	DTO for transferring post-related data
ResponseWrapper	General	DTO that wraps responses to standardize the output format
SearchNewTitleDTO	Movie	DTO for handling searches for new movie titles
SearchTitleDTO	Movie	DTO for searching movies by specific criteria
StringCountDTO	General	DTO for mapping string data to counts
UpdateTitleDTO	User	DTO for updating movie title data
UserInfluencerDTO	User	DTO for handling influencer data related to users
UserRecommendationsDTO	User	DTO for user-based recommendations
WatchlistDTO	User	DTO for managing user watchlists

4.2.2 it.unipi.movieland.controller

The **controller** package manages the application's **RESTful interface**, exposing endpoints for data access. Classes are annotated with `@RestController`, defining API methods using `@GetMapping`, `@PostMapping`, `@PutMapping`, and `@DeleteMapping`. Controllers handle HTTP requests, validate input, and delegate logic to services, returning JSON responses via `ResponseEntity<?>`. Best practices suggest avoiding direct exposure of database entities, instead using **DTOs** to control which fields are returned. Controllers should remain lightweight, focusing only on request processing.

CLASS	FOLDER	DESCRIPTION
CelebrityController	Celebrity	Endpoints for handling searching, updating, adding and deleting a celebrity.
CommentController	Comment	Endpoints for handling searching, updating, adding and deleting the comments.
ManagerController	Manager	Endpoints for the manager for managing inconsistencies, analytics and admin account.

MovieController	Movie	Endpoints for handling searching, updating, adding and deleting a title.
PostController	Post	Endpoints for handling searching, updating, adding and deleting the posts.
ReviewController	Review	Endpoints for handling searching, updating, adding and deleting a review.
RecommendationController	User	Endpoints for the recommendation system of movies, users, celebrity for the users.
UserController	User	Endpoints for handling searching, updating, adding, deleting, handle personal movie lists, account managing.

4.2.3 it.unipi.movieland.model

The **model** package contains classes that represent the **application domain**, defining the entities stored in the databases. For **MongoDB**, each class is annotated with `@Document(collection = "collection_name")` and includes an `@Id` field as a unique identifier. For **Neo4j**, classes are annotated with `@Node("Label")` and use `@Id` and `@GeneratedValue(UUID)` for the id management.

CLASS	FOLDER	DESCRIPTION
CelebrityMongoDB	Celebrity	Represent the celebrity document in MongoDB
CelebrityNeo4j	Celebrity	Represent the celebrity node in Neo4j
Job	Celebrity	Represent the jobs array in the celebrity document in MongoDB
Comment	Comment	Represent the comment document in MongoDB
Manager	Manager	Represent the manager document in MongoDB
Movie	Movie	Represent the movie document in MongoDB
MovieCelebrity	Movie	Represent the celebrity array in the movie document in MongoDB
MovieReview	Movie	Represent the review array in the movie document in MongoDB

MovieNeo4j	Movie	Represent the movie node in Neo4j
Post	Post	Represent the post document in MongoDB
ReviewMongoDB	Review	Represent the review document in MongoDB
ReviewNeo4j	Review	Represent the review node in MongoDB
UserCelebrity	User	Represent the celebrity array in the user document in MongoDB
UserMongoDB	User	Represent the user document in MongoDB
UserMovie	User	Represent the liked_movies array in the document in MongoDB
UserNeo4J	User	Represent the user node in MongoDB
UserReview	User	Represent the review array in the user document in MongoDB
CountryEnum		Enum for the production countries of the movies
GenderEnum		Enum for the user genders
GenreEnum		Enum for the genres of the movies
PlatformEnum		Enum for the streaming platforms
TitleTypeEnum		Enum for the type of the title (MOVIE/SHOW)

4.2.4 it.unipi.movieland.service

The **service** package contains **business logic**, separating data access from controllers. Classes are annotated with `@Service` and orchestrate operations between MongoDB and Neo4j, combining data from both databases. For example, a service might retrieve a document from MongoDB and enrich it with related data from Neo4j before returning it. The service layer improves **maintainability** by keeping complex logic outside controllers. It is also an ideal place to integrate validations, and error handling. By following this approach, the application remains scalable and well-structured.

CLASS	FOLDER	DESCRIPTION
CelebrityService	Celebrity	Business logic for request regarding celebrities
CommentService	Comment	Business logic for request regarding comments
ManagerService	Manager	Business logic for request regarding the manager
MovieService	Movie	Business logic for request regarding the movies

PostService	Post	Business logic for request regarding posts
ReviewService	Review	Business logic for request regarding reviews
UserService	User	Business logic for request regarding the users

4.2.5 it.unipi.movieland.repository

The **repository** package contains interfaces responsible for **data access** in MongoDB and Neo4j. For **MongoDB**, interfaces extend `MongoRepository<T, ID>`, providing built-in CRUD operations without implementation. Custom queries can be defined using the `@Query` annotation in JSON format. For **Neo4j**, interfaces extend `Neo4jRepository<T, ID>`, allowing Cypher queries with `@Query("MATCH ... RETURN ...")`. Both repositories are injected into services to handle data access, ensuring proper separation of concerns and reusability. This approach prevents direct database access from controllers and enhances maintainability by centralizing data operations.

CLASS	FOLDER	DESCRIPTION
ManagerRepository	Manager	Handle the communication with the manager document in MongoDB.
CelebrityMongoDBRepository	Celebrity	Handle the communication with the celebrity document in MongoDB.
CelebrityNeo4JRepository	Celebrity	Handle the communication with the celebrity node in Neo4j.
MovieMongoDBRepository	Movie	Handle the communication with the movie document in MongoDB.
MovieMongoDBImplementation	Movie	Implements two queries for titles search in MongoDB.
MovieNeo4jRepository	Movie	Handle the communication with the movie node in Neo4j.
PostMongoDBRepository	Post	Handle the communication with the post document in MongoDB.
CommentMongoDBRepository	Comment	Handle the communication with the comment document in MongoDB.

ReviewMongoDBRepository	Review	Handle the communication with the review document in MongoDB.
ReviewNeo4JRepository	Review	Handle the communication with the review node in Neo4j.
UserMongoDBRepository	User	Handle the communication with the user document in MongoDB.
UserNeo4JRepository	User	Handle the communication with the user node in Neo4j.

4.2.6 it.unipi.movieland.exception

The **exception** package centralizes **error handling** and defines custom exceptions to improve API robustness. Instead of returning generic errors, it provides meaningful messages and proper HTTP status codes. Custom exceptions extend `RuntimeException` and are thrown when specific conditions occur. This approach ensures clean, maintainable code, prevents unnecessary stack traces in responses, and improves the user experience by providing clear error messages.

CLASS	FOLDER	DESCRIPTION
CelebrityNotFoundException		Exception thrown when a celebrity
MovieNotFoundException		Exception thrown when a movie is not found on MongoDB
BusinessException		Thrown for generic exception

4.2.7 it.unipi.movieland.utils

The utils package contains the code for the APIs request used for adding new title to the platform and the de-serializer for handling the response given by the API request.

CLASS	FOLDER	DESCRIPTION
APIRequest	apiMDB	Contains the code for the API requests for searching and adding a new title to the platform
SearchNewTitleDTODeserializer	deserializers	Deserialize json object returned by searching a new title by name
TitleDeserializer	deserializers	Deserialize json object returned by searching a new title by id

4.3 MONGODB RELEVANT OPERATIONS

4.3.1 Movies

Search Movie by Title or Keywords

This method allows users to search for a title using a string (label) that can be part of the title or a keyword. It also requires a title type parameter (MOVIE or SHOW), handled with the TitleTypeEnum. The query filters by type and uses a text index to score titles based on how well they match the label. The results are sorted by this score and IMDb score. To display only relevant information, we project a few fields, ensuring a brief description. Pagination is used to manage large results, showing only the required number of titles per page.

```
public Page<SearchTitleDTO> getTitleByTitleOrKeyword(TitleTypeEnum type, String label, Pageable pageable) throws BusinessException {
    try {
        MatchOperation matchOperation = Aggregation.match(
                Criteria.where( key: "$text").is(new Document("$search", label)).and( key: "type").is(type)
        );

        ProjectionOperation addScoreField = Aggregation.project( ...fields: "_id", "title", "release_year", "poster_path", "imdb_score")
                .andExpression( expression: "{$meta: 'textScore'}").as( alias: "score");

        SortOperation sortOperation = Aggregation.sort(
                org.springframework.data.domain.Sort.by(
                        org.springframework.data.domain.Sort.Order.desc( property: "score"),
                        org.springframework.data.domain.Sort.Order.desc( property: "imdb_score")
                )
        );

        ProjectionOperation finalProjection = Aggregation.project( ...fields: "_id", "title", "release_year", "poster_path", "imdb_score");

        Aggregation aggregation = Aggregation.newAggregation(
                matchOperation,
                addScoreField,
                sortOperation,
                finalProjection,
                Aggregation.skip((long) pageable.getOffset()),
                Aggregation.limit(pageable.getPageSize())
        );

        AggregationResults<SearchTitleDTO> results = mongoTemplate.aggregate(aggregation, collectionName: "Movies", SearchTitleDTO.class);
        List<SearchTitleDTO> movies = results.getMappedResults();

        return new PageImpl<>(movies, pageable, movies.size());
    } catch (Exception e) {
        throw new BusinessException("Error retrieving titles", e);
    }
```

Search Movie by Title or Keywords with Filters

This method allows users to search for a title using keywords or a title, with the added option of applying multiple filters. Users can specify filters such as genre, release year, platform, production countries, age certification, IMDb score, and IMDb votes to narrow down the results. This enables a more tailored search experience, making it easier to find specific movies based on both their title or keywords and various criteria.

```java
public Page<SearchTitleDTO> getTitlewithFilters(TitleTypeEnum type,  1 usage  Leo-Cecchini *
        Optional<String> label,
        Optional<List<GenreEnum>> genres,
        Optional<Integer> release_year,
        Optional<PlatformEnum> platform,
        Optional<CountryEnum> production_countries,
        Optional<String> age_certification,
        Optional<Integer> imdb_scores,
        Optional<Integer> imdb_votes,
        Pageable pageable
) throws BusinessException {

    try {
        List<Criteria> criteriaList = new ArrayList<>();

        // Base criteria for type
        criteriaList.add(Criteria.where( key: "type").is(type));
        // Handle text search for label if present
        if (label.isPresent() && !label.get().isEmpty()) {
            criteriaList.add(Criteria.where( key: "$text").is(new Document("$search", label.get())));
        }
        // Add optional criteria only if they are present
        genres.ifPresent(g -> criteriaList.add(Criteria.where( key: "genres").all(g)));
        release_year.ifPresent(y -> criteriaList.add(Criteria.where( key: "release_year").is(y)));
        platform.ifPresent(p -> criteriaList.add(Criteria.where( key: "platform").is(p.getDisplayName())));
        production_countries.ifPresent(pc -> criteriaList.add(Criteria.where( key: "production_countries").is(pc)));
        age_certification.ifPresent(ac -> criteriaList.add(Criteria.where( key: "age_certification").is(ac)));
        imdb_scores.ifPresent(score -> criteriaList.add(Criteria.where( key: "imdb_score").gte(score)));
        imdb_votes.ifPresent(votes -> criteriaList.add(Criteria.where( key: "imdb_votes").gte(votes)));

        // Create the match operation with all criteria
        Criteria combinedCriteria = new Criteria().andOperator(criteriaList.toArray(new Criteria[0]));
        MatchOperation matchOperation = Aggregation.match(combinedCriteria);

        // Project fields and add text score
        ProjectionOperation addScoreField = Aggregation.project()
                .and( name: "_id").as( alias: "_id")
                .and( name: "title").as( alias: "title")
                .and( name: "release_year").as( alias: "release_year")
```

```
            .and( name: "poster_path").as( alias: "poster_path")
            .and( name: "imdb_score").as( alias: "imdb_score")
            .andExpression( expression: "{$meta: 'textScore'}").as( alias: "score");

    // Sort by text score and imdb_score
    SortOperation sortOperation = Aggregation.sort(Sort.by(Sort.Direction.DESC, ...properties: "score", "imdb_score"

    // Final projection to clean up output
    ProjectionOperation finalProjection = Aggregation.project()
            .and( name: "_id").as( alias: "_id")
            .and( name: "title").as( alias: "title")
            .and( name: "release_year").as( alias: "release_year")
            .and( name: "poster_path").as( alias: "poster_path")
            .and( name: "imdb_score").as( alias: "imdb_score");

    // Count total matches for pagination
    Aggregation countAggregation = Aggregation.newAggregation(matchOperation);
    AggregationResults<Document> countResults = mongoTemplate.aggregate(
            countAggregation, collectionName: "Movies", Document.class);
    long total = countResults.getMappedResults().size();

    // Build final aggregation with pagination
    Aggregation aggregation = Aggregation.newAggregation(
            matchOperation,
            addScoreField,
            sortOperation,
            finalProjection,
            Aggregation.skip(pageable.getOffset()),
            Aggregation.limit(pageable.getPageSize())
    );
    AggregationResults<SearchTitleDTO> results = mongoTemplate.aggregate(
            aggregation, collectionName: "Movies", SearchTitleDTO.class);
    List<SearchTitleDTO> movies = results.getMappedResults();

    return new PageImpl<>(movies, pageable, total);
} catch (Exception e) {
    throw new BusinessException("Error retrieving titles", e);
}
```

Similar to the previous query, we utilize a text index when the user enters a label. Then, for each selected filter, we create a new criterion to refine the search results. Since all filter parameters are optional, we first check if they have been provided by the user using the ifPresent() method. In the end, we select only a few key fields to display and ensure the results are sorted by IMDb score and relevance to the search query. This approach enhances the search experience, delivering relevant results based on both the user's input and selected filters.

Most Frequent Actors in Specific Genres

Retrieve the most frequent actors in each genre provided by the user. Return the actor's ID, name, poster, and the count of movies the actor has participated in for each genre.

```
@Aggregation(pipeline = {  1 usage  Leo-Cecchini
        "{ $match: { genres: { $all: ?0 } } }",
        "{ $unwind: '$actors' }",
        "{ $group: { " + "_id: '$actors.id', " + "name: { $first: '$actors.name' }, " + "poster: { $first: '$actors.poster' }, " +
                "movieCount: { $sum: 1 } " +
                "} }",
        "{ $sort: { movieCount: -1 } }",
        "{ $limit: 5 }",
        "{ $project: { " + "actor_id: '$_id', " + "name: 1, " + "poster: 1, " + "movieCount: 1 " + "} }"
})
List<ActorDTO> mostFrequentActorsSpecificGenres(List<String> genres);
```

Most Voted Movies in Specific Genres

Retrieve the total number of IMDb votes for each genre. Return the genre name and the combined number of votes collected by all movies belonging to that genre.

```
@Aggregation(pipeline = {  1 usage  Leo-Cecchini
        "{ $sort: { imdb_votes: -1 } }",
        "{ $limit: 1000 }",
        "{ $unwind: \"$genres\" }",
        "{ $group: { _id: \"$genres\", count: { $sum: \"$imdb_votes\" } } }",
        "{ $sort: { count: -1 } }",
        "{ $project: { label: \"$_id\", count: 1 } }"
})
List<StringCountDTO> mostVotedMoviesByGenres();
```

Most Popular Celebrities

Sort the movies by IMDb votes, and for each actor, count the number of movies they have participated in. Sort the actors by their movie participation count. Return the actor's ID, name, poster, and the total number of movies they have appeared in.

```
@Aggregation(pipeline = {  1 usage  Leo-Cecchini
        "{ $sort: { imdb_votes: -1 } }",
        "{ $limit: 1000 }",
        "{ $unwind: \"$genres\" }",
        "{ $group: { _id: \"$genres\", count: { $sum: \"$imdb_votes\" } } }",
        "{ $sort: { count: -1 } }",
        "{ $project: { label: \"$_id\", count: 1 } }"
})
List<StringCountDTO> mostVotedMoviesByGenres();
```

Best Platforms

Retrieve the platform with the highest number of movies from the top 1000, sorted by IMDb votes across the databases. Return the platform name and the total number of movies it hosts.

```
@Aggregation(pipeline = {  1 usage   Leo-Cecchini
        "{ $sort: { imdb_votes: -1 } }",
        "{ $limit: 1000 }",
        "{ $unwind: '$platform' }",
        "{ $group: { " + "_id: '$platform', " + "count: { $sum: 1 } " + "} }",
        "{ $sort: { count: -1 } }",
        "{ $project: { " + "label: '$_id', " + "count: 1 " + "} }"
})
List<StringCountDTO> bestPlatformForTop1000Movies();
```

Keyword Combination Percentage Analysis

Given one or more keywords, retrieve all the keywords that appear most frequently in titles containing the user-provided keywords. Return both the keyword and its combination percentage with the given keyword(s).

```
@Aggregation(pipeline = {  1 usage   Leo-Cecchini
        "{ $match: { keywords: { $all: ?0 } } }",
        "{ $group: { _id: null, totalMovies: { $count: {} }, keywordCounts: { $push: \"$keywords\" } } }",
        "{ $unwind: \"$keywordCounts\" }",
        "{ $unwind: \"$keywordCounts\" }",
        "{ $group: { _id: \"$keywordCounts\", movieCount: { $count: {} }, totalMovies: { $first: \"$totalMovies\" } } }",
        "{ $project: { label: \"$_id\", movieCount: 1, combinedPercentage: { $multiply: [ { $divide: [ \"$movieCount\", \"$totalMovies\" ] }, 100 ] } } }",
        "{ $sort: { combinedPercentage: -1 } }"
})
List<CombinedPercentageDTO> percentageOfCombinedKeywords(List<String> keywords);
```

4.3.2 Celebrities

Search a Celebrity by Id

This method retrieves a celebrity from MongoDB using their ID. It calls findById on the celebrityMongoRepository to find the celebrity. If the celebrity is not found, it throws a CelebrityNotFoundException. If found, it maps the entity to a CelebrityMongoDto and returns it.

```
//METHOD TO RETRIEVE A CELEBRITY USING THEIR ID (MONGODB)
public CelebrityMongoDto getCelebrityByIdMongo(int id) {

    CelebrityMongoDB celebrity = celebrityMongoRepository.findById(id)
            .orElseThrow(() -> new CelebrityNotFoundException("CELEBRITY WITH ID " + id + " NOT FOUND."));

    return CelebrityMongoDto.fromEntity(celebrity);
}
```

Search for an Actor by Text

This method searches for actors by name or character in MongoDB using the provided searchTerm. It calls searchActorsAndCharacters on the celebrityMongoRepository to find matching celebrities. If no results are found, a RuntimeException is thrown. If matches are found, the method maps the entities to CelebrityMongoDto and returns a list of DTOs.

```
// METHOD TO SEARCH FOR AN ACTOR BY NAME OR CHARACTER (MONGODB)
public List<CelebrityMongoDto> searchActorsByCharacterMongo(String searchTerm) {
    List<CelebrityMongoDB> celebrities = celebrityMongoRepository.searchActorsAndCharacters(searchTerm);

    if (celebrities.isEmpty()) {
        throw new RuntimeException("NO CELEBRITIES FOUND FOR THE SEARCH TERM: " + searchTerm);
    }

    return celebrities.stream() Stream<CelebrityMongoDB>
            .map(CelebrityMongoDto::fromEntity) Stream<CelebrityMongoDto>
            .collect(Collectors.toList());
}
```

```
//FIND ACTORS BY NAME OR CHARACTER
@Aggregation(pipeline = {
        "{ $match: { $text: { $search: '?0' } } }",
        "{ $addFields: { score: { $meta: 'textScore' } } }",
        "{ $unwind: '$jobs' }",
        "{ $group: { " +
                "_id: '$_id', " + "name: { $first: '$name' }, " + "jobs: { $push: '$jobs' }, " +
                "followers: { $first: '$followers' }, " + "poster: { $first: '$poster' }, " +
                "score: { $first: '$score' }, " + "imdb_score: { $first: '$imdb_score' } } }",
        "{ $sort: { score: -1, imdb_score: -1 } }",
        "{ $project: { name: 1, jobs: 1, followers: 1, poster: 1 } }",
        "{ $limit: 10 }"
})
List<CelebrityMongoDB> searchActorsAndCharacters(String searchTerm);
```

Get Jobs for Celebrity

This method allows you to retrieve all the jobs associated with a specific celebrity, identified by their celebrityId. It looks up the celebrity in the database and returns a list of their roles (e.g., actor, director) in different movies. If no jobs are found, it returns an empty list with a relevant message. If the celebrity doesn't exist, it returns a "not found" error. This ensures easy access to all job-related information for a given celebrity.

```
//METHOD FOR SEARCHING FOR A CELEBRITY'S JOBS
@Transactional
public ResponseEntity<Object> getJobsForCelebrity(int celebrityId) {
    try {
        CelebrityMongoDB celebrity = celebrityMongoRepository.findById(celebrityId)
                .orElseThrow(() -> new IllegalArgumentException("CELEBRITY WITH ID " + celebrityId + " NOT FOUND!"));

        List<Job> jobs = celebrity.getJobs();

        if (jobs == null || jobs.isEmpty()) {
            return ResponseEntity.status(HttpStatus.OK)
                    .body(Map.of( k1: "message",  v1: "NO JOBS FOUND FOR CELEBRITY WITH ID " + celebrityId,  k2: "jobs", List.of()));
        }

        return ResponseEntity.status(HttpStatus.OK)
                .body(Map.of( k1: "message",  v1: "JOBS RETRIEVED SUCCESSFULLY",  k2: "jobs", jobs));

    } catch (IllegalArgumentException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(Map.of( k1: "message", e.getMessage()));
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of( k1: "message",  v1: "An unexpected error occurred",  k2: "error", e.getMessage()));
    }
}
```

4.3.3 Users

Search User by Username

This method retrieves a user from MongoDB by their username. If the user is not found, it throws a NoSuchElementException. If the user exists, it returns the user's data from MongoDB.

```
public UserMongoDB getUserByUsername(String username) {
    Optional<UserMongoDB> user = mongoRepository.findById(username);
    if (user.isEmpty()) {throw new NoSuchElementException("User " + username + " not found");}
    return user.get();
}
```

Retrieve User's Watchlist

This method retrieves the watchlist of a specific user. It checks if the user exists in the MongoDB database, then fetches the user's watchlist. If any issues arise during the retrieval process, an exception is thrown. The watchlist is returned in the form of a WatchlistDTO object, which contains the list of movies the user has added to their watchlist.

```
public WatchlistDTO getWatchlist(String username) {
    if (!mongoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    try {
        UserMongoDB user= mongoRepository.getWatchlist(username);
        return new WatchlistDTO(user.getWatchlist());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Search for Users by Text

This method allows users to search for other users based on partial matches of their usernames or names. It uses MongoDB's text-based search functionality to find and rank users based on the similarity of the input to the users' details. The results are sorted by relevance, showing the closest matches at the top.

```
@Aggregation(pipeline = {
        "{ $match: { $text: { $search: '?0' }, $or: [ { 'country': '?1' }, { '?1': null }, { '?1': '' } ] } }",
        "{ $sort: { score: { $meta: 'textScore' } } }",
        "{ $skip: ?2 }",
        "{ $limit: ?3 }"
})
List<UserMongoDB> searchUser(String query, String country, int offset, int limit);
```

Add Movie to Watchlist

This method allows a user to add a movie to their watchlist. It checks if the user exists in the MongoDB database, ensures the user hasn't exceeded the watchlist limit of 20 movies, and verifies that the movie is not already in the watchlist. Additionally, it ensures the movie exists in the database. If all conditions are met, the movie is added to the user's watchlist in MongoDB. If any issue arises, an exception is thrown.

```
@Transactional
public void addToWatchlist(String username, String movieId) {
    Optional<UserMongoDB> user = mongoRepository.findById(username);
    if (user.isEmpty()) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (user.get().getWatchlist().size()>20) {
        throw new IllegalArgumentException("Can't have more than 20 movies in the watchlist");
    } else if  (mongoRepository.isMovieInWatchlist(username, movieId)) {
        throw new IllegalArgumentException("Movie '" + movieId + "' is already in the watchlist");
    } else if (!movieMongoRepository.existsById(movieId)) {
        throw new NoSuchElementException("Movie '" + movieId + "' doesn't exists");
    }
    try {
        mongoRepository.addToWatchlist(username,movieId);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

The addToWatchList query finds the users and film and iserts the second one in the user's watchlist.

```
@Aggregation(pipeline = {
        "{ $match: { _id: ?0 } }",
        "{ $lookup: { from: 'Movies', let: { movieId: ?1 }, pipeline: " +
                "[ { $match: { $expr: { $eq: ['$_id', '$$movieId'] } } } ], as: 'watchMovie' } }",
        "{ $set: { watchMovie: { $first: '$watchMovie' } } }",
        "{ $set: { watchlist: { $concatArrays: ['$watchlist', [{ film_id: '$watchMovie._id', " +
                "title: '$watchMovie.title', poster: '$watchMovie.poster_path' }]] } } }",
        "{ $merge: { into: 'Users', whenMatched: 'merge', whenNotMatched: 'insert' } }"
})
void addToWatchlist(String userId, String movieId);
```

Remove Movie from Watchlist

This method allows a user to remove a movie from their watchlist. It checks if the user exists in the MongoDB database and verifies that the movie is present in their watchlist. If the movie is found, it removes it from the user's watchlist in MongoDB. If any issues occur during the process, an exception is thrown.

```
@Transactional
public void removeFromWatchlist(String username, String movieId) {
    if (!mongoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!mongoRepository.isMovieInWatchlist(username, movieId)) {
        throw new IllegalArgumentException("Movie '" + movieId + "' is not in the watchlist");
    }
    try {
        mongoRepository.removeFromWatchlist(username,movieId);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

The removeFromWatchList query finds the and removes the movie from the watchlist.

```
@Query("{ '_id': ?0 }")
@Update("{ $pull: { 'watchlist': { 'film_id': ?1 } } }")
void removeFromWatchlist(String userId, String movieId);
```

4.3.4 Reviews

Update Reviews

The updateReviews method refreshes a movie's reviews by selecting the five most recent ones. It ensures that only the latest reviews are retained, helping keep the movie's review data up-to-date. This operation helps maintain the relevancy of review information displayed to users.

```
@Aggregation(pipeline = {
        "{ $match: { _id: ?0 } }",
        "{ $lookup: { from: 'Reviews', localField: '_id', foreignField: 'movie_id', as: 'reviews' } }",
        "{ $unwind: '$reviews' }",
        "{ $sort: { 'reviews.timestamp': -1 } }",
        "{ $limit: 5 }",
        "{ $group: { _id: '$_id', title: { $first: '$title' }, type: { $first: '$type' }, " +
                "description: { $first: '$description' }, release_year: { $first: '$release_year' }, " +
                "genres: { $first: '$genres' }, keywords: { $first: '$keywords' }, " +
                "production_countries: { $first: '$production_countries' }, runtime: { $first: '$runtime' }, " +
                "poster_path: { $first: '$poster_path' }, imdb_score: { $first: '$imdb_score' }, " +
                "imdb_votes: { $first: '$imdb_votes' }, platform: { $first: '$platform' }, " +
                "director: { $first: '$director' }, actors: { $first: '$actors' }, " +
                "reviews: { $push: { id: '$reviews._id', review: '$reviews.review', " +
                "sentiment: '$reviews.sentiment', username: '$reviews.username', timestamp: '$reviews.timestamp' } }
                "revenue: { $first: '$revenue' }, budget: { $first: '$budget' }, " +
                "age_certification: { $first: '$age_certification' }, seasons: { $first: '$seasons' } } }",
        "{ $merge: { into: 'Movies', whenMatched: 'merge', whenNotMatched: 'insert' } }"
})
```

Fetch Reviews by Movie Id

This method retrieves reviews for a specific movie based on the provided movieId. It uses pagination with page and size parameters to manage large sets of reviews efficiently. If the movie does not exist, it throws an exception. The method ensures that reviews are fetched in the correct range for the given pagination parameters.

```
public List<ReviewMongoDB> getReviewsByMovieId(String movieId, int page, int size) {
    if (!movieMongoDBRepository.existsById(movieId)) {
        throw new NoSuchElementException("Movie '" + movieId + "' doesn't exists");
    }
    try {
        Pageable pageable = PageRequest.of(page, size);
        return reviewMongoRepository.findByMovieId(movieId,(int)pageable.getOffset(),size);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

It then returns all the reviews filtered by the movie ID and paginated.

```
@Aggregation(pipeline = {  1 usage   matteo
        "{ $match: { movie_id: ?0 } }",
        "{ $sort: { timestamp: -1 } }",
        "{ $skip: ?2 }",
        "{ $limit: ?3 }"
})
List<ReviewMongoDB> findByMovieId(String movieId,int offset, int limit);
```

Fetch Reviews by Username

This method retrieves all reviews written by a user, identified by userId, using pagination to manage large sets of reviews. It ensures the user exists in the database before fetching their reviews. If the user is found, the reviews are returned based on the page and size parameters, offering efficient handling of large amounts of data.

```
public List<ReviewMongoDB> getReviewsByUsername(String userId, int page, int size) {
    if (!userMongoDBRepository.existsById(userId)) {
        throw new NoSuchElementException("User '" + userId + "' doesn't exists");
    }
    try {
        Pageable pageable = PageRequest.of(page, size);
        return reviewMongoRepository.findByUsername(userId,(int)pageable.getOffset(),size);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

It then returns all the reviews filtered by the user ID and paginated.

```
@Aggregation(pipeline = {
        "{ $match: { username: ?0 } }",
        "{ $sort: { timestamp: -1 } }",
        "{ $skip: ?2 }",
        "{ $limit: ?3 }"
})
List<ReviewMongoDB> findByUsername(String userId,int offset, int limit);
```

Get Review by Id

This method retrieves a review by its unique id from the database. If the review is found, it is returned; otherwise, a NoSuchElementException is thrown. The method ensures efficient handling by catching any potential exceptions and throwing a RuntimeException for further handling.

```
public ReviewMongoDB getReviewById(String id) {
    try {
        return reviewMongoRepository.findById(id).orElseThrow(NoSuchElementException::new);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

```
public List<UserNeo4J> findUserLikeReview(String reviewId){
    if (!reviewNeo4JRepository.existsById(reviewId)) {
        throw new NoSuchElementException("Review '" + reviewId + "' doesn't exists");
    }
    try {
        return reviewNeo4JRepository.findUserLikeReview(reviewId);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Find Best Movies by Review Ratio

The method finds the top 10 movies by selecting the 500 most-reviewed ones, calculating the ratio of positive to total reviews, and returning those with the highest percentages.

```
@Aggregation({
        "{ $group: { _id: '$movie_id', positiveReviews: { $sum: { $cond: [ { $eq: ['$sentiment', true] }, 1, 0 ] } }, totalReviews: { $sum: 1 } } }",
        "{ $sort: { totalReviews: -1 } }",
        "{ $limit: 500 }",
        "{ $project: { id: '$_id',positiveReviews: 1, totalReviews: 1, ratio: { $cond: { if: { $eq: ['$totalReviews', 0] }," +
                " then: 0, else: { $divide: ['$positiveReviews', '$totalReviews'] } } } } }",
        "{ $sort: { ratio: -1 } }",
        "{ $limit: 10 }"
})
List<ReviewRatioDTO> findTopMoviesByReviewRatio();
```

4.3.5 Posts

Create a Post

This method creates a new post by accepting the text, author, and movie ID. It first checks if the author and movie exist in the respective repositories; if either is missing, it throws a 404 Not Found exception. If both validations pass, it creates a new Post object and saves it to the database. The method then returns the created post, ensuring that only valid users can post about existing movies.

```
//METHOD TO CREATE A POST
public Post createPost(String text, String author, String movieId) {
    if (!userRepository.existsById(author)) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "AUTHOR WITH ID : " + author + " NOT FOUND.");}

    if(!movieMongoDBRepository.existsById(movieId)) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "MOVIE WITH ID " + movieId + " NOT FOUND");}

    Post post = new Post(text,author,movieId, comment: null);
    return postMongoDBRepository.save(post);
}
```

Update a Post

This method updates an existing post by accepting the post ID and the new text. It first checks if the post with the provided ID exists in the database; if not, it throws a 404 Not Found exception. If the post is found, it updates the text and sets the current date and time as the new datetime. The updated post is then saved back into the database, and the updated post is returned.

```
//METHOD TO UPDATE AN EXISTING POST
public Post updatePost(String id, String text) {

    Post existingPost = postMongoDBRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "POST WITH ID " + id + " NOT FOUND"));

    existingPost.setText(text);
    existingPost.setDatetime(LocalDateTime.now());

    return postMongoDBRepository.save(existingPost);
}
```

Get Post by Id

This method retrieves a post using its unique ID. It queries the database for the post with the given ID and returns it if found. If the post does not exist, it throws a 404 Not Found exception with an appropriate error message. If the post is found, it is returned to the caller.

```
//METHOD TO RETRIEVE A POST BY ID
public Post getPostById(String id) {  1 usage   new *
    Optional<Post> post = postMongoDBRepository.findById(id);

    if (post.isEmpty()) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "COMMENT WITH ID " + id + " NOT FOUND");
    }
    return post.get();
}
```

Delete a Post

This method is responsible for deleting a post identified by its unique ID. It first checks if the post exists in the database. If the post is found, it is deleted from the repository. If the post doesn't exist, a 404 Not Found exception is thrown, notifying the user that the post with the given ID is not available for deletion.

```
//METHOD TO DELETE A POST BY ID
public void deletePost(String id) {

    Post existingPost = postMongoDBRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "POST WITH ID " + id + " NOT FOUND"));

    postMongoDBRepository.delete(existingPost);
}
```

Fetch Posts by Movie Id

This method retrieves posts related to a specific movie by its ID. It accepts the movie ID along with pagination parameters, such as the page number and size. It uses these parameters to create a PageRequest and fetches the relevant posts from the repository. The result is returned as a paginated list of PostDTO objects, allowing the client to manage large amounts of data efficiently.

```
//METHOD TO GET POSTS BY MOVIE ID
public Page<PostDTO> getPostsByMovieId(String movie_id, int page, int size) {

    PageRequest pageRequest = PageRequest.of(page, size);

    return postMongoDBRepository.findByMovieId(movie_id, pageRequest);
}
```

Retrieve User's Post Activity

This method fetches a list of post activity details by calling the getPostActivity method from the postMongoDBRepository. It returns a collection of PostActivityDTO objects, which typically contain insights such as the number of posts, interactions, or other relevant metrics.

```
@Aggregation(pipeline = {
        "{ $project: { hour: { $hour: { $dateFromString: { dateString: \"$datetime\" } } } } }",
        "{ $group: { _id: \"$hour\", postCount: { $count: {} } } }",
        "{ $sort: { _id: 1 } }",
        "{ $project: { hour: \"$_id\", postCount: 1 } }"
})
List<PostActivityDTO> getPostActivity();
```

Get Influencers Report

This method retrieves a report of the most influential users. It calls the getInfluencersReport() method from the postMongoDBRepository, which likely returns a list of UserInfluencerDTO objects. These objects may contain information such as user activity, engagement, and influence within the platform.

```
@Aggregation(pipeline = {
        "{ $group: { _id: \"$author\", totalPosts: { $count: {} }, totalComments: { $sum: { $size: \"$comment\" } } } }",
        "{ $addFields: { commentsPerPost: { $cond: { if: { $eq: [ \"$totalPosts\", 0 ] }, then: 0, else: { $divide: [ \"$totalComments\", \"$totalPosts\" ] } } } } }",
        "{ $sort: { totalComments: -1, totalPosts: -1, commentsPerPost: -1 } }",
        "{ $project: { username: \"$_id\", totalPosts: 1, totalComments: 1, commentsPerPost: 1 } }"
})
List<UserInfluencerDTO> getInfluencersReport();
```

Fetch Posts Within a Data Range

This method retrieves a paginated list of posts created between two specified dates. It accepts the start date, end date, page number, and page size as parameters, then uses the findByDatetimeBetween method of postMongoDBRepository to fetch the results. The pagination ensures efficient handling of large datasets.

```
public Page<PostDTO> getPostsByDateRange(LocalDateTime startDate, LocalDateTime endDate, int page, int size) {
    PageRequest pageRequest = PageRequest.of(page, size);
    return postMongoDBRepository.findByDatetimeBetween(startDate, endDate, pageRequest);
}
```

4.3.6 Comments

Create a Comment

To create a comment, the system first checks if the author exists in the database by verifying the user's ID. If the author doesn't exist, an error is returned. Next, it checks if the post to which the comment is being added also exists. If the post doesn't exist, another error is generated. If both checks pass, the system creates a new Comment object with the text, author ID, and post ID. Finally, the comment is saved in the repository, completing the operation.

```
//METHOD TO CREATE A COMMENT
public Comment createComment(String text, String authorId,String postId) {
    if (!userRepository.existsById(authorId)) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "AUTHOR WITH ID : " + authorId + " NOT FOUND.");}

    if (!postRepository.existsById(postId)) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "POST WITH ID :" + postId + "NOT FOUND.");}
    Comment comment = new Comment(text,authorId,postId);
    return commentRepository.save(comment);
}
```

Fetch Comments by Post Id

This method fetches a comment by its ID. It first checks if the comment exists using findById on the commentRepository. If found, it returns the comment. If not, a 404 error is thrown, ensuring missing comments are handled gracefully and the user receives a clear error message.

```
//METHOD TO RETRIEVE A COMMENT BY ID
public Comment getCommentById(String id) {
    Optional<Comment> comment = commentRepository.findById(id);

    if (comment.isEmpty()) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "COMMENT WITH ID " + id + " NOT FOUND");
    }
    return comment.get();
}
```

Delete a Comment

This method deletes a comment using the provided ID. It first checks if the comment exists with findById on the commentRepository. If found, it deletes the comment. If not found, a 404 error is thrown. Deleting the comment ensures that unnecessary data is removed, maintaining data integrity.

```
//METHOD TO DELETE A COMMENT BY ID
public void deleteComment(String id) {

    Comment existingComment = commentRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "COMMENT WITH ID " + id + " NOT FOUND"));

    commentRepository.delete(existingComment);
}
```

Update a Comment

This method updates a comment's content based on the provided ID. It first checks if the comment exists using findById on the commentRepository. If found, it updates the text and sets the datetime to the current timestamp. If not found, a 404 error is thrown. After updating, the comment is saved back to the database using the save method, returning the updated comment.

```
//METHOD TO UPDATE AN EXISTING COMMENT
public Comment updateComment(String id, String text) {

    Comment existingComment = commentRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "COMMENT WITH ID " + id + " NOT FOUND"));

    existingComment.setText(text);
    existingComment.setDatetime(LocalDateTime.now());

    return commentRepository.save(existingComment);
}
```

Fetch Comment by Author

This method checks if the author exists using existsById from the userRepository. If not, a 404 error is thrown. Then, it fetches the comments written by the author using findByAuthor with pagination (PageRequest.of(page, size)). If no comments are found, another 404 error is thrown. If comments exist, the paginated list is returned.

```
//METHOD TO GET COMMENTS BY AUTHOR
public Page<Comment> getCommentsByAuthor(String authorId, int page, int size) {

    boolean userExists = userRepository.existsById(authorId);

    if (!userExists) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
                String.format("AUTHOR WITH ID %s NOT FOUND", authorId));
    }

    PageRequest pageRequest = PageRequest.of(page, size);
    Page<Comment> comments = commentRepository.findByAuthor(authorId, pageRequest);

    if (comments.isEmpty()) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
                String.format("NO COMMENTS FOUND FOR THE AUTHOR WITH ID %s", authorId));
    }
    return comments;
}
```

Fetch Comments by Date Range

This method retrieves comments within a specified date range. It takes startDate and endDate as parameters, along with pagination values (page and size). The method uses findByDatetimeBetween from the commentRepository to fetch the comments and returns a paginated list based on the provided date range.

```
//METHOD TO GET COMMENTS BY DATE
public Page<Comment> getCommentsByDateRange(LocalDateTime startDate, LocalDateTime endDate, int page, int size) {
    PageRequest pageRequest = PageRequest.of(page, size);
    return commentRepository.findByDatetimeBetween(startDate, endDate, pageRequest);
}
```

Fetch Comments by Post Id

This method retrieves comments for a specific post based on the provided post ID. It first checks if the post exists using existsById on the postRepository. If the post is not found, a 404 error is thrown. Then, it fetches the comments using findCommentsByPostId from the commentRepository with pagination, returning a paginated list of comments for the specified post.

```
//METHOD TO GET COMMENTS BY POST ID
public Slice<Comment> getCommentsByPostId(ObjectId postId, int page, int size) {
    if (!postRepository.existsById(String.valueOf(postId))) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "POST WITH ID:" + postId + "NOT FOUND.");
    }

    Pageable pageable = PageRequest.of(page, size);
    return commentRepository.findCommentsByPostId(postId, pageable);
}
```

4.4 NEO4J GRAPH-DOMAIN QUERIES

4.4.1. Users

Movies Recommendations

The system uses second-degree relationships to generate personalized recommendations for movies, users, and celebrities. For movies, it identifies those liked by the people a user follows or movies featuring their followed celebrities. User recommendations are based on individuals commonly followed by the user's connections. Celebrity suggestions come from identifying actors, directors, or other public figures who have collaborated with followed celebrities or appeared in movies the user has liked.

```
@Query("MATCH (user:User {username:$userId})-[:FOLLOW|ACTOR_FOLLOW]->(intermediate)-[:ACTED_IN|DIRECTED_IN|MOVIE_LIKE]->(recommendedMovie:Movie) " +
        "WHERE NOT (user)-[:MOVIE_LIKE]->(recommendedMovie) " +
        "WITH recommendedMovie, COUNT(DISTINCT intermediate) AS numRelations, user " +
        "OPTIONAL MATCH (recommendedMovie)<-[:MOVIE_LIKE]-(likingUser:User) WITH recommendedMovie, numRelations, COUNT(likingUser) AS LikeCount, user " +
        "WITH recommendedMovie, numRelations, LikeCount, size(apoc.coll.intersection(user.favorite_genres, recommendedMovie.genres)) AS commonGenres " +
        "RETURN recommendedMovie.title AS title, recommendedMovie.imdb_id AS id, recommendedMovie.Poster_path as poster, LikeCount AS votes, numRelations, commonGenres " +
        "ORDER BY (0.5*commonGenres + numRelations) DESC, LikeCount DESC " +
        "SKIP $skip LIMIT $limit")
Slice<MovieRecommendationsDTO> recommendMovies(String userId,Pageable pageable);
```

Celebrities Recommendation

The system suggests celebrities who have worked in movies the user has liked or are followed by the user's followed users.

```
@Query("MATCH (user:User {username: $userId})-[:FOLLOW|MOVIE_LIKE]->(intermediate)-" +
        "[:ACTOR_FOLLOW|DIRECTED_IN|ACTED_IN]-(recommendedCelebrity:Celebrity) " +
        "WHERE NOT (user)-[:ACTOR_FOLLOW]->(recommendedCelebrity) " +
        "WITH recommendedCelebrity, COUNT(DISTINCT intermediate) AS numRelations " +
        "OPTIONAL MATCH (recommendedCelebrity)<-[:ACTOR_FOLLOW]-(follower:User) " +
        "WITH recommendedCelebrity, numRelations, COUNT(follower) AS FollowerCount " +
        "RETURN recommendedCelebrity.name AS name, recommendedCelebrity.person_id AS id, " +
        "recommendedCelebrity.Poster AS poster, numRelations, FollowerCount AS followers " +
        "ORDER BY numRelations DESC, FollowerCount DESC " +
        "SKIP $skip LIMIT $limit")
Slice<CelebrityRecommendationsDTO> recommendCelebrities(String userId,Pageable pageable);
```

Users Recommendations

The system identifies users who have liked the same movies or celebrities, as well as users followed by the user's followed users.

```
@Query("MATCH (user:User {username: $userId})-[:FOLLOW|ACTOR_FOLLOW|MOVIE_LIKE]->(intermediate)-" +
        "[:FOLLOW|ACTOR_FOLLOW|MOVIE_LIKE]-(recommendedUser:User) " +
        "WHERE NOT (user)-[:FOLLOW]->(recommendedUser) AND user <> recommendedUser " +
        "WITH recommendedUser, COUNT(DISTINCT intermediate) AS numRelations " +
        "OPTIONAL MATCH (recommendedUser)-[:FOLLOW]->(follower:User) " +
        "WITH recommendedUser, numRelations, COUNT(follower) AS FollowerCount " +
        "RETURN recommendedUser.username AS username, recommendedUser.country AS country, " +
        "numRelations, FollowerCount AS followers " +
        "ORDER BY numRelations DESC, FollowerCount DESC " +
        "SKIP $skip LIMIT $limit")
Slice<UserRecommendationsDTO> recommendUsers(String userId,Pageable pageable);
```

Liked Reviews Based Recommendations

The system suggests movies that have received positive reviews from reviewers whose reviews the user has previously liked. It identifies these reviewers and recommends movies positively reviewed by them.

```
@Query("MATCH (user:User {username: $userId})-[:REVIEW_LIKE]->(Review)<-[:POSTED]-(User)-" +
        "[:POSTED]->(r:Review)-[:EVALUATES]->(recommendedMovie:Movie) " +
        "WHERE r.sentiment=true AND NOT (user)-[:MOVIE_LIKE]->(recommendedMovie) " +
        "WITH recommendedMovie, COUNT(DISTINCT r) AS numRelations, user " +
        "OPTIONAL MATCH (recommendedMovie)<-[:MOVIE_LIKE]-(likingUser:User) " +
        "WITH recommendedMovie, numRelations, COUNT(likingUser) AS LikeCount, user " +
        "WITH recommendedMovie, numRelations, LikeCount, " +
        "size(apoc.coll.intersection(user.favorite_genres, recommendedMovie.genres)) AS commonGenres " +
        "RETURN recommendedMovie.title AS title, recommendedMovie.imdb_id AS id, " +
        "recommendedMovie.Poster_path as poster, LikeCount AS votes, numRelations, commonGenres " +
        "ORDER BY (0.5*commonGenres + numRelations) DESC, LikeCount DESC " +
        "SKIP $skip LIMIT $limit")
Slice<MovieRecommendationsDTO> recommendByReview(String userId,Pageable pageable);
```

Cast Based Recommendations

The system suggests celebrities who have frequently collaborated with those the user follows. It analyzes shared projects between followed celebrities and identifies other actors with the most collaborations.

```
@Query("MATCH (user:User {username:$userId})-[:MOVIE_LIKE]->(likedMovie:Movie)<-[:ACTED_IN]-" +
        "(intermediate:Celebrity)-[:ACTED_IN]->(recommendedMovie:Movie) " +
        "WHERE NOT (user)-[:MOVIE_LIKE]->(recommendedMovie) " +
        "WITH recommendedMovie, COUNT(DISTINCT intermediate) AS numRelations, user " +
        "OPTIONAL MATCH (recommendedMovie)<-[:MOVIE_LIKE]-(likingUser:User) " +
        "WITH recommendedMovie, numRelations, COUNT(likingUser) AS LikeCount, user " +
        "WITH recommendedMovie, numRelations, LikeCount, " +
        "size(apoc.coll.intersection(user.favorite_genres, recommendedMovie.genres)) AS commonGenres " +
        "RETURN recommendedMovie.title AS title, recommendedMovie.imdb_id AS id, " +
        "recommendedMovie.Poster_path as poster, LikeCount AS votes, numRelations, commonGenres " +
        "ORDER BY (0.5*commonGenres + numRelations) DESC, LikeCount DESC " +
        "SKIP $skip LIMIT $limit")
Slice<MovieRecommendationsDTO> recommendByCast(String userId,Pageable pageable);
```

4.4.1 Reviews

Find Users Who Liked a Review

This method retrieves a list of users who have liked a specific review, identified by its reviewId. It first checks if the review exists in the Neo4j database, throwing a NoSuchElementException if not. If the review exists, the method fetches all users who have liked the review. Any errors during the process are caught and thrown as a RuntimeException.

```
public List<UserNeo4J> findUserLikeReview(String reviewId){
    if (!reviewNeo4JRepository.existsById(reviewId)) {
        throw new NoSuchElementException("Review '" + reviewId + "' doesn't exists");
    }
    try {
        return reviewNeo4JRepository.findUserLikeReview(reviewId);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

4.5 MONGODB AND NEO4J COMBINED QUERIES

4.5.1. Movies

Create a New Movie

To add a new title to the platform, we use the **mdblistAPI**, which leverages the **imdbAPIs** to retrieve movie information. The process is divided into two main steps:

1. **Get the IMDb ID**: The administrator searches for the movie title since it is unlikely they already have the IMDb ID. The API returns a list of movies with the ID, title, release year, and type, allowing the administrator to choose the correct movie and obtain the ID.

2. **Add the movie**: Once the IMDb ID is found, a second request is made to retrieve the movie details, which are then saved in both the MongoDB and Neo4j databases.

Important notes: Some details, such as budget, keywords, or actors, may be missing in the data received, but the administrator can manually add them. Additionally, **mdblistAPI** is a free service with a daily limit of 1000 requests, allowing us to retrieve nearly all the necessary information. This system makes it easy to add new movies with accurate data, minimizing the need for manual input.

a. Search for a new Title by Name

The first method is used when a user wants to search for a movie by its title. The user inputs the movie title, and the system displays a list of movies that match the given string. The main piece of information shown will be the **IMDb ID**, as it's crucial for adding the movie to the platform. To accomplish this, the system will first call the API, retrieving the list of related movies along with their IMDb IDs.

```java
public String getIdByTitle(String type, String title, Optional<Integer> year) throws IOException, InterruptedException {
    String baseUrl = "https://api.mdblist.com/search/";
    String yearParam = (year.isPresent()) ? "&year=" + year.get() : "&";
    String apiKey = "9penwmkrxop9yv1nch91du21l";
    //creating the url and the request
    String requestUrl = String.format("%s%s?query=%s%s&apikey=%s",
            baseUrl,
            type,
            URLEncoder.encode(title, StandardCharsets.UTF_8),
            yearParam,
            apiKey
    );
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(requestUrl))
            .header( name: "accept",  value: "application/json")
            .build();
    //sending the request
    HttpResponse<String> response = HttpClient.newHttpClient().send(request, HttpResponse.BodyHandlers.ofString());
    return response.statusCode() == 200 ? response.body() : null;
}
```

```java
public Map<Integer,List<SearchNewTitleDTO>> searchNewTitleByName(TitleTypeEnum type,  1 usage  Leo-Cecchini +1
                                                                 String title,
                                                                 Optional<Integer> year) throws IOException, InterruptedException {

    Map<Integer,List<SearchNewTitleDTO>> map = new HashMap<>();
    //creating api request and getting the response
    APIRequest api = new APIRequest();
    String response = api.getIdByTitle(type.name(), title, year);

    if(response == null){   //api error: code 2
        map.put(2, null);
        return map;
    }
    //parsing the response to a json object
    JsonObject jsonResponse = JsonParser.parseString(response).getAsJsonObject();
    //creating an array of json object because I can have a list of more than one title
    JsonArray searchArray = jsonResponse.getAsJsonArray( memberName: "search");

    if(searchArray.isEmpty()){    //no title found: code 1
        map.put(1, null);
        return map;
    }

    //gson for the deserialization of the json object
    Gson gson = new GsonBuilder()
            .registerTypeAdapter(SearchNewTitleDTO.class, new SearchNewTitleDTODeserializer())
            .create();

    Type listType = new TypeToken<List<SearchNewTitleDTO>>() {}.getType();  Leo-Cecchini
    List<SearchNewTitleDTO> movies = gson.fromJson(searchArray, listType); //converting the titles to searchNewTitleDTO

    map.put(0, movies);     //no errors: code 0
    return map;
}
```

After retrieving the movie data, the next step is to process the returned object. We need to parse it into a JSON object for easier manipulation of its fields. Since multiple movies may be returned, we will create a **List** of SearchNewTitleDTO objects to efficiently handle and store the data for each movie. This structure will allow us to access each movie's details and use them in the next steps, such as displaying the titles and IMDb IDs for selection.

b. Add Title by Id

Once the user selects a movie from the list by its IMDb ID, the second step is to add the title to the system. To do this, we need to make another API call, this time using the selected IMDb ID to retrieve the full details of the movie. This will provide all the necessary information such as the title, release year, cast, keywords, etc. After fetching the details, we can proceed to add the movie to both the MongoDB and Neo4j databases, ensuring that all relevant data is stored for later use.

```
public String getMoviebyId(String type, String id) throws IOException, InterruptedException {  1 usage  Leo-Cecchini
    type = type.toLowerCase();
    String baseUrl = "https://api.mdblist.com/imdb/";
    String apiKey = "9penwmkrxop9yv1nch91du21l";

    String requestUrl = String.format("%s%s/%s?apikey=%s",
            baseUrl,
            URLEncoder.encode(type, StandardCharsets.UTF_8),
            URLEncoder.encode(id, StandardCharsets.UTF_8),
            apiKey);

    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(requestUrl))
            .header( name: "accept",  value: "application/json")
            .build();

    HttpResponse<String> response = HttpClient.newHttpClient().send(request, HttpResponse.BodyHandlers.ofString());
    return response.statusCode() == 200 ? response.body() : null;
}
```

```java
public int addTitleById(TitleTypeEnum typeEnum, String id) throws IOException, InterruptedException { 1 usage
    Optional<Movie> title = movieMongoDBInterface.findById(id);
    if(title.isPresent())   //title already in the database
        return 1;

    //creating api request and getting the response
    APIRequest api = new APIRequest();
    String response = api.getMoviebyId(typeEnum.name(), id);

    if(response == null){ //api error
        return 2;
    }
    //parsing the response to a json object
    JsonObject jsonResponse = JsonParser.parseString(response).getAsJsonObject();
    //creating an array of json object because I can have a list of more than one title
    JsonArray searchArray = jsonResponse.getAsJsonArray( memberName: "search");

    if( searchArray != null && searchArray.isEmpty()){  //no titles found
        return 1;
    }

    //fetching fields of json object
    Gson gson = new GsonBuilder()
            .registerTypeAdapter(Movie.class, new TitleDeserializer())
            .create();
    //create movie
    Movie movie = gson.fromJson(response, Movie.class);
    movie.setType(movie.getType());

    MovieNeo4j movieNeo4j = new MovieNeo4j(movie.get_id(), movie.getTitle(), movie.getGenre());

    //add movie to mongodb
    movieMongoDBInterface.insert(movie);
    //add movie to neo4j
    movieNeo4jRepository.save(movieNeo4j);
    return 0;
}
```

After retrieving the movie details using the IMDb ID, we parse the response and extract the necessary fields like title, release year, and cast. We then create a new Movie entity and insert it into MongoDB. Simultaneously, we establish the relevant relationships in Neo4j to link the movie with its actors and directors. This ensures the movie data is stored in both databases.

4.5.2. Celebrities

Create a Celebrity

This method is responsible for creating a celebrity in two different databases: MongoDB and Neo4j. First, it checks if the celebrity already exists in either of the two databases. If the celebrity exists in both, it returns a message indicating that the celebrity with the given ID already exists in both databases. If the celebrity does not exist in either database, the method proceeds by creating a new CelebrityMongoDB object and saving it in the MongoDB database using the save method of the MongoDB repository. Then, it creates a CelebrityNeo4J object with the same ID and other information of the celebrity and saves it in the Neo4j database using the respective repository. Finally, the method returns a success message confirming that the celebrity has been successfully created in both databases. If an error occurs during the creation process, an exception is thrown with a message detailing the error, ensuring that any issues are properly handled and reported.

```
// METHOD TO CREATE A CELEBRITY IN BOTH DATABASES
public String createCelebrityInBothDatabases(int id, String name, String poster) {
    try {
        CelebrityMongoDto existingMongoCelebrity = null;
        CelebrityNeo4JDto existingNeo4jCelebrity = null;

        try {
            existingMongoCelebrity = getCelebrityByIdMongo(id);
        } catch (CelebrityNotFoundException e) { }

        try {
            existingNeo4jCelebrity = getCelebrityByIdNeo4j(String.valueOf(id));
        } catch (CelebrityNotFoundException e) { }

        if (existingMongoCelebrity != null && existingNeo4jCelebrity != null) {
            return "CELEBRITY WITH ID " + id + " ALREADY EXISTS IN BOTH DATABASES";
        }

        CelebrityMongoDB newMongoCelebrity = new CelebrityMongoDB(id, name, poster);
        CelebrityMongoDB savedMongoCelebrity = celebrityMongoRepository.save(newMongoCelebrity);

        CelebrityNeo4J newNeo4jCelebrity = new CelebrityNeo4J(String.valueOf(savedMongoCelebrity.getId()), name, poster);
        celebrityNeo4JRepository.save(newNeo4jCelebrity);

        return "CELEBRITY CREATED SUCCESSFULLY IN BOTH DATABASES";

    } catch (Exception e) {
        throw new RuntimeException("ERROR CREATING CELEBRITY IN BOTH DATABASES: " + e.getMessage());
    }
}
```

Delete a Celebrity

This method deletes celebrity from both MongoDB and Neo4j databases. It first checks if the celebrity exists in MongoDB; if not, it throws an exception. If found, the celebrity is deleted. The same process is then done for Neo4j. The @Transactional annotation ensures that both deletions happen as part of a single transaction, ensuring data consistency. If an error occurs during either deletion, the transaction is rolled back to prevent data inconsistency.

```
// METHOD TO DELETE A CELEBRITY IN BOTH DATABASES
@Transactional
public void deleteCelebrityInBothDatabases(int id) {

    Optional<CelebrityMongoDB> mongoCelebrity = celebrityMongoRepository.findById(id);
    if (mongoCelebrity.isEmpty()) {
        throw new CelebrityNotFoundInMongoException("CELEBRITY WITH ID " + id + " NOT FOUND IN THE MONGO DATABASE");
    }
    celebrityMongoRepository.deleteById(id);

    Optional<CelebrityNeo4J> neo4jCelebrity = celebrityNeo4JRepository.findById(String.valueOf(id));
    if (neo4jCelebrity.isEmpty()) {
        throw new CelebrityNotFoundInNeo4jException("CELEBRITY WITH ID " + id + " NOT FOUND IN THE NEO4J DATABASE");
    }
    celebrityNeo4JRepository.deleteById(String.valueOf(id));
}
```

Add Job to Actor

This method adds a job (role) to an actor (celebrity) for a specific movie. It first verifies if both the actor and the movie exist in the database; if either is not found, a 404 error is returned. The method then checks if the job already exists for that movie and character. If it does, a 400 error is returned to avoid duplicates. If the job doesn't exist, a new job ID is generated, and the job is added to the actor's job list. The job is saved in MongoDB, and a relationship is created in Neo4j to link the actor to the movie. If the movie has fewer than five actors, the actor is also added to the movie's actor list, and the movie is updated. A successful response with a 201 status code and job details is returned, or a 500 error is issued in case of failure.

```java
//METHOD TO ADD JOBS TO AN ACTOR
@Transactional
public ResponseEntity<Object> addJobToActor(int id, String movie_id, String character) {
    try {
        CelebrityMongoDB celebrity = celebrityMongoRepository.findById(id)
                .orElseThrow(() -> new CelebrityNotFoundException("CELEBRITY WITH ID " + id + " NOT FOUND"));

        Movie movie = movieMongoDBRepository.findById(movie_id)
                .orElseThrow(() -> new MovieNotFoundException("MOVIE WITH ID " + movie_id + " NOT FOUND"));

        if (celebrity.getJobs() == null) { celebrity.setJobs(new ArrayList<>()); }

        boolean jobExists = celebrity.getJobs().stream()
                .anyMatch( Job job -> movie_id.equals(job.getMovie_id()) && character != null && character.equals(job.getCharacter()));
        if (jobExists) {...}

        String jobId = generateJobIdForCelebrity(celebrity);
        Job newJob = new Job(celebrity, role: "ACTOR", movie_id, movie.getTitle(), character);
        newJob.setJob_id(jobId);
        celebrity.getJobs().add(newJob);
        celebrityMongoRepository.save(celebrity);

        celebrityNeo4JRepository.addActedInRelationship(String.valueOf(id), movie_id, character);

        if (movie.getActors() == null) { movie.setActors(new ArrayList<>());}

        if (movie.getActors().size() >= 5) {
            return ResponseEntity.status(HttpStatus.OK) .body(Map.of( k1: "message", v1: "JOB ADDED SUCCESSFULLY FOR ACTOR"));}

        boolean actorExists = movie.getActors().stream().anyMatch( MovieCelebrity actor -> actor.getId().equals(id));
        if (!actorExists) {...}

        JobDto jobDto = JobDto.fromEntity(newJob);
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(Map.of( k1: "message", v1: "JOB ADDED SUCCESSFULLY FOR ACTOR WITH ID " + id, k2: "job", jobDto));

    } catch (CelebrityNotFoundException | MovieNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(Map.of( k1: "error", ex.getMessage()));
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of( k1: "error", v1: "FAILED TO ADD JOB FOR ACTOR", k2: "details", e.getMessage()));
    }
}
```

Add Job to Director

This method assigns a director's role to a celebrity (either actor or director) for a specific movie. It first verifies if both the celebrity and the movie exist in the database, throwing an error if either is missing. Next, it checks if the celebrity already holds a director role for that movie; if so, a message is returned to prevent duplication. If the director role is absent, a new job entry is created for the celebrity and associated with the movie. The job is added to the celebrity's list, and changes are saved in MongoDB. Additionally, a relationship linking the celebrity to the movie is established in Neo4j. The method then checks the movie's director list and, if necessary, adds the celebrity to it. Upon success, the method returns a success message. Any errors are caught and handled appropriately with error messages. This ensures both MongoDB and Neo4j reflect the director's role in the movie.

```
//METHOD TO ADD JOBS TO AN DIRECTOR
@Transactional
public ResponseEntity<Object> addJobToDirector(int id, String movie_id) {
    try {
        CelebrityMongoDB celebrity = celebrityMongoRepository.findById(id)
                .orElseThrow(() -> new IllegalArgumentException("CELEBRITY WITH ID " + id + " NOT FOUND"));

        Movie movie = movieMongoDBRepository.findById(movie_id)
                .orElseThrow(() -> new IllegalArgumentException("MOVIE WITH ID " + movie_id + " NOT FOUND"));

        if (celebrity.getJobs() == null) {...}

        boolean jobAlreadyExists = celebrity.getJobs().stream()
                .anyMatch( Job job -> job.getMovie_id().equals(movie_id) && job.getRole().equalsIgnoreCase( anotherString: "Director"));

        if (jobAlreadyExists) {...}

        String jobId = generateJobIdForCelebrity(celebrity);
        Job newJob = new Job();
        newJob.setJob_id(jobId);
        newJob.setRole("DIRECTOR");
        newJob.setMovie_id(movie_id);
        newJob.setMovie_title(movie.getTitle());

        celebrity.getJobs().add(newJob);
        celebrityMongoRepository.save(celebrity);

        celebrityNeo4JRepository.addDirectorInRelationship(String.valueOf(id), movie_id);

        if (movie.getDirectors() == null) {...}

        if (movie.getDirectors().size() >= 5) {...}

        boolean directorExists = movie.getDirectors().stream()
                .anyMatch( MovieCelebrity director -> director.getId().equals(id));
        if (!directorExists) {...}

        return ResponseEntity.status(HttpStatus.CREATED)
                .body(Map.of( k1: "message",  v1: "JOB ADDED SUCCESSFULLY TO CELEBRITY WITH ID: " + id,  k2: "jobId", jobId));

    } catch (IllegalArgumentException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(Map.of( k1: "error", e.getMessage()));
    } catch (Exception e) {...}
}
```

Remove Jobs from Celebrity

This method removes a job associated with a celebrity in both MongoDB and Neo4j databases. It starts by verifying if the celebrity with the given ID exists, returning an error if not found. Once the celebrity is located, the method checks if the specified job exists in their list. If the job is found, it is removed from the list, and the celebrity's data is updated in MongoDB. Depending on whether the job is related to acting or directing, the corresponding relationship in Neo4j is deleted — either removing the actor's or the director's link to the movie. The method also updates the movie's list of actors or directors accordingly and saves the movie back to MongoDB. If the job is successfully removed, the method returns a success message. If any error occurs during the process, the exception is caught, and an appropriate error message is returned.

```
//REMOVE JOBS TO ACTOR O DIRECTOR
@Transactional
public ResponseEntity<Object> removeJobById(int celebrityId, String jobId) {
    try {
        CelebrityMongoDB celebrity = celebrityMongoRepository.findById(celebrityId)
                .orElseThrow(() -> new CelebrityNotFoundException("CELEBRITY WITH ID " + celebrityId + " NOT FOUND!"));

        List<Job> jobs = Optional.ofNullable(celebrity.getJobs()).orElse(Collections.emptyList());


        Job jobToRemove = jobs.stream() Stream<Job>
                .filter( Job job -> jobId.equals(job.getJob_id()))
                .findFirst() Optional<Job>
                .orElse( other: null);

        if (jobToRemove == null) {...}

        jobs.remove(jobToRemove);
        celebrityMongoRepository.save(celebrity);

        switch (jobToRemove.getRole()) {...}

        Movie movie = movieMongoDBRepository.findById(jobToRemove.getMovie_id())
                .orElseThrow(() -> new MovieNotFoundException("MOVIE WITH ID " + jobToRemove.getMovie_id() + " NOT FOUND"));

        if ("ACTOR".equalsIgnoreCase(jobToRemove.getRole())) {
            movie.getActors().removeIf( MovieCelebrity actor -> actor.getId().equals(celebrityId));
        } else if ("DIRECTOR".equalsIgnoreCase(jobToRemove.getRole())) {...}

        movieMongoDBRepository.save(movie);

        return ResponseEntity.status(HttpStatus.OK)
                .body(Map.of( k1: "message",  v1: "JOB WITH ID " + jobId + " REMOVED SUCCESSFULLY FROM CELEBRITY"));

    } catch (CelebrityNotFoundException | MovieNotFoundException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(Map.of( k1: "message", e.getMessage()));
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of( k1: "message",  v1: "AN ERROR OCCURRED WHILE REMOVING THE JOB",  k2: "error", e.getMessage()));
    }
}
```

Update Celebrity

This method updates a celebrity's information in both MongoDB and Neo4j databases. It first checks if the celebrity exists in MongoDB by searching for their ID. If the celebrity is not found, a "not found" error message is returned. If the celebrity exists, it proceeds to update the celebrity's name and poster image, if new values are provided. After updating MongoDB, the method attempts to update the same celebrity information in Neo4j using a custom update query. If the update is successful in Neo4j, a success message is returned, indicating that the celebrity was updated in both databases. If Neo4j fails to update, an error message is returned. In case of any other exceptions, the method catches the error and returns a 500 internal server error message with the exception details.

```
//METHOD TO UPDATE A CELEBRITY
@Transactional
public ResponseEntity<Object> updateCelebrity(String personId, String name, String poster) {
    try {
        int id = Integer.parseInt(personId);
        CelebrityMongoDB mongoCelebrity = celebrityMongoRepository.findById(id).orElse( other: null);

        if (mongoCelebrity == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                    .body(Map.of( k1: "message",  v1: "CELEBRITY WITH ID " + personId + " NOT FOUND IN THE DATABASES"));
        }

        if (name != null && !name.trim().isEmpty()) {
            mongoCelebrity.setName(name);
        }

        if (poster != null && !poster.trim().isEmpty()) {
            mongoCelebrity.setPoster(poster);
        }

        celebrityMongoRepository.save(mongoCelebrity);

        Optional<CelebrityNeo4J> neo4jCelebrityOptional = celebrityNeo4JRepository.updateCelebrity(personId, name, poster);

        if (neo4jCelebrityOptional.isPresent()) {
            return ResponseEntity.status(HttpStatus.OK)
                    .body(Map.of( k1: "message",  v1: "CELEBRITY UPDATED SUCCESSFULLY IN BOTH DATABASES."));
        } else {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .body(Map.of( k1: "message",  v1: "FAILED TO UPDATED CELEBRUTY IN NEO4J."));
        }

    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of( k1: "message",  v1: "An error occurred while updating the celebrity",  k2: "error", e.getMessage()));
    }
}
```

4.5.3. Users

Add a User

This method allows the creation of a new user by adding them to both MongoDB and Neo4j databases. First, it checks if the username already exists in either database and if the email is already in use. If any of these conditions are met, an exception is thrown. If the validations pass, the password is encrypted, and a new user is created in both MongoDB and Neo4j. The user data, such as their personal details and preferences, is saved in both databases. If any error occurs during the process, the newly created user is removed from MongoDB to maintain data consistency.

```
@Transactional
public UserMongoDB addUser(String username, String email, String name, String surname,
                           String password, CountryEnum country, String phoneNumber,
                           List<GenreEnum> favoriteGenres, GenderEnum gender, LocalDate birthday) {
    if (mongoRepository.existsById(username) || neoRepository.existsById(username)) {
        throw new IllegalArgumentException("User '" + username + "' already exists");
    } else if (mongoRepository.existsByEmail(email)) {
        throw new IllegalArgumentException("Mail'" + email + "' already used");
    }
    try {
        String passwordE=encrypt(password, secretKey: "MovieLand0123456");
        UserMongoDB mongoUser = new UserMongoDB(username, email, name, surname, passwordE,
                country, phoneNumber, favoriteGenres, gender, birthday);
        UserNeo4J neoUser = new UserNeo4J(username, name, surname, country, favoriteGenres);
        mongoRepository.save(mongoUser);
        neoRepository.save(neoUser);
        return mongoUser;
    } catch (Exception e) {
        mongoRepository.deleteById(username);
        throw new RuntimeException(e);
    }
}
```

Delete a user

This method deletes a user from both MongoDB and Neo4j. It first checks if the user exists in either database. If not, it throws an exception. If the user exists, it attempts to delete the user from both databases. If an error occurs during deletion, the user's data is restored in MongoDB to maintain consistency.

```
@Transactional
public void deleteUser(String username) {
    if (!mongoRepository.existsById(username) && !neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    UserMongoDB userOld =mongoRepository.findById(username)
            .orElseThrow(() -> new NoSuchElementException("User '" + username + "' doesn't exists"));
    try {
        mongoRepository.deleteById(username);
        neoRepository.deleteById(username);
    } catch (Exception e) {
        mongoRepository.save(userOld);
        throw new RuntimeException(e);
    }
}
```

Update Users

This method updates the details of an existing user in both the MongoDB and Neo4j databases. It first checks if the user exists in MongoDB by retrieving their data using the username. If the user does not exist, an exception is thrown. If any of the updated fields (name, surname, country, phone number, favorite genres, gender) are null, the method will retain the user's previous values. The method then attempts to update the user's data in both databases. If an error occurs during the update process, the user's data is reverted to its original state in MongoDB, ensuring consistency across the databases.

```
@Transactional
public void updateUser(String username, String name, String surname, CountryEnum country, String phoneNumber,
                       List<GenreEnum> favoriteGenres, GenderEnum gender) {
    Optional<UserMongoDB> userOld = mongoRepository.findById(username);
    if (userOld.isEmpty()) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    if (name == null) {name = userOld.get().getName();}
    if (surname == null) {surname = userOld.get().getSurname();}
    if (country == null) {country = userOld.get().getCountry();}
    if (phoneNumber == null) {phoneNumber = userOld.get().getPhone_number();}
    if (favoriteGenres == null) {favoriteGenres = userOld.get().getFavorite_genres();}
    if (gender == null) {gender = userOld.get().getGender();}
    try {
        mongoRepository.updateUser(username, name, surname, country, phoneNumber, favoriteGenres, gender);
        neoRepository.updateUser(username, name, surname, country, favoriteGenres);
    } catch (Exception e) {
        mongoRepository.updateUser(username,  userOld.get().getName(), userOld.get().getSurname(),
                userOld.get().getCountry(), userOld.get().getPhone_number(), userOld.get().getFavorite_genres(),
                userOld.get().getGender());
        throw new RuntimeException(e);
    }
}
```

User Authentication

This method authenticates a user by comparing the provided password with the one stored in MongoDB. It first checks if the user exists; if not, it throws a NoSuchElementException. If the user is found, it encrypts the provided password and compares it with the stored one. If they match, authentication is successful; otherwise, it returns false.

```
public boolean authenticate(String username, String password) {
    if (!mongoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    try {
        String pass=mongoRepository.findById(username).get().getPassword();
        String passwordE=encrypt(password, secretKey: "MovieLand0123456");
        return pass.equals(passwordE);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Get Liked Movies

This method retrieves a paginated list of movies liked by a specific user. It first checks if the user exists in the Neo4j database, throwing a NoSuchElementException if not. Then, using the provided page and size parameters, it queries the database for the user's liked movies, returning the results in a Slice. If any error occurs during the process, a RuntimeException is thrown.

```
public Slice<UserMovie> getLikedMovies(String username, int page, int size) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    try {
        Pageable pageable = PageRequest.of(page, size);
        return neoRepository.getLikedMovies(username,pageable);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

The getLikedMovies query retrieves from Neo4J the MOVIE_LIKE relationships of the user.

```
@Query("MATCH (u:User)-[r:MOVIE_LIKE]->(m:Movie) WHERE u.username = $userId " +
        "ORDER BY r.date DESC SKIP $skip LIMIT $limit " +
        "RETURN m.title AS title, m.imdb_id AS movieId, m.Poster_path AS poster ")
Slice<UserMovie> getLikedMovies(String userId, Pageable pageable);
```

Add Liked Movie

This method allows a user to like a movie by adding it to their list of liked movies. It first verifies that the user and the movie exist in their respective databases (Neo4j and MongoDB), throwing an exception if either is missing. Additionally, it checks if the user has already liked the movie to prevent duplicates. If all checks pass, it updates both the Neo4j and MongoDB repositories to reflect the new "liked" movie. In case of any errors during the process, a RuntimeException is thrown.

```
@Transactional
public void addLikedMovie(String username, String movieId) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!movieNeoRepository.existsById(movieId)) {
        throw new NoSuchElementException("Movie '" + movieId + "' doesn't exists");
    } else if (neoRepository.isMovieLiked(username, movieId)) {
        throw new IllegalArgumentException("Movie '" + movieId + "' is already liked");
    }
    try {
        neoRepository.addToLikedMovies(username,movieId);
        mongoRepository.addToLikedMovies(username, movieId);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

The AddToLikedMovies Queries create the relationship between the user and the movie and push the movie into the array liked_movies in the user document, keeping it'size equal to 5.

```
@Aggregation(pipeline = {
        "{ $match: { _id: ?0 } }",
        "{ $lookup: { from: 'Movies', let: { movieId: ?1 }, pipeline: " +
                "[ { $match: { $expr: { $eq: ['$_id', '$$movieId'] } } } ], as: 'likedMovie' } }",
        "{ $set: { likedMovie: { $first: '$likedMovie' } } }",
        "{ $set: { liked_movies: { $concatArrays: ['$liked_movies', [{ film_id: '$likedMovie._id', " +
                "title: '$likedMovie.title', poster: '$likedMovie.poster_path' }]] } } }",
        "{ $set: { liked_movies: { $slice: ['$liked_movies', -5] } } }",
        "{ $merge: { into: 'Users', whenMatched: 'merge', whenNotMatched: 'insert' } }"
})
void addToLikedMovies(String userId, String movieId);
```

Remove Liked Movie

This method removes a movie from a user's list of liked movies. It first ensures that both the user and the movie exist in their respective databases (Neo4j and MongoDB). It also checks if the movie is currently liked by the user, throwing an exception if the movie isn't in the liked list. Once these checks pass, the method updates both the Neo4j and MongoDB repositories to remove the movie from the user's liked movies. In case of any errors during the removal process, a RuntimeException is thrown.

```
@Transactional
public void removeLikedMovie(String username, String movieId) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!movieNeoRepository.existsById(movieId)) {
        throw new NoSuchElementException("Movie '" + movieId + "' doesn't exists");
    } else if (!neoRepository.isMovieLiked(username, movieId)) {
        throw new IllegalArgumentException("Movie '" + movieId + "' isn't liked");
    }
    try {
        List<UserMovie> newMovies = neoRepository.removeFromLikedMovies(username,movieId);
        mongoRepository.removeFromLikedMovies(username,newMovies);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

The RemoveFromLikedMovies Queries delete the relationship between the user and the movie and sets the liked_movies array in the user documenta as the five most recent movies liked.

```
@Query("MATCH (u:User)-[r:MOVIE_LIKE]->(m:Movie) WHERE u.username = $userId AND m.imdb_id = $movieId" +
        " DELETE r WITH u MATCH (u)-[r2:MOVIE_LIKE]->(m2:Movie) " +
        "ORDER BY r2.likedAt DESC LIMIT 5 " +
        "RETURN m2.title AS title, m2.imdb_id AS movieId, m2.Poster_path AS poster")
List<UserMovie> removeFromLikedMovies(String userId, String movieId);
```

Get Followed Celebrities

This method retrieves a paginated list of celebrities followed by a user. It first checks if the user exists in the Neo4j database. If the user is found, it proceeds to query the list of followed celebrities, returning a slice of data based on the specified page and size. If any error occurs during the process, it logs the error and throws a RuntimeException.

```
public Slice<UserCelebrity> getFollowedCelebrities(String username,int page, int size) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    try {
        Pageable pageable = PageRequest.of(page, size);
        return neoRepository.getFollowedCelebrities(username,pageable);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

```
@Query("MATCH (u:User)-[r:ACTOR_FOLLOW]->(m:Celebrity) WHERE u.username = $userId" +
        " ORDER BY r.date DESC SKIP $skip LIMIT $limit" +
        " RETURN m.name AS name, toInteger(m.person_id) AS celebrityId ")
Slice<UserCelebrity> getFollowedCelebrities(String userId,Pageable pageable);
```

Add Followed Celebrity

This method adds a celebrity to a user's followed list. It first checks if the user and celebrity exist in their respective databases (Neo4j). If the celebrity is already followed by the user, an error is thrown. If all checks pass, the celebrity is added to the user's followed list in both Neo4j and MongoDB, and the celebrity's follower count is increased in the MongoDB repository. Any errors encountered during the process are logged and followed by a RuntimeException.

```
@Transactional
public void addFollowedCelebrity(String username, int celebrityId) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!celebrityNeoRepository.existsById(String.valueOf(celebrityId))) {
        throw new NoSuchElementException("Celebrity '" + celebrityId + "' doesn't exists");
    } else if (neoRepository.isCelebrityFollowed(username, String.valueOf(celebrityId))) {
        throw new IllegalArgumentException("Celebrity '" + celebrityId + "' is already followed");
    }
    try {
        neoRepository.addToFollowedCelebrities(username,String.valueOf(celebrityId));
        mongoRepository.addToFollowedCelebrities(username, celebrityId);
        celebrityMongoRepository.increaseFollowers(celebrityId);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

The AddToFollowedCelebrities Queries create the relationship between the user and the celebrity and push the celebrity into the array followed_celebrities in the user document, keeping it'size equal to 5.

```
@Query("MATCH (u:User), (m:Celebrity) WHERE u.username = $userId AND m.person_id = $celebrityId " +
        "MERGE (u)-[r:ACTOR_FOLLOW]->(m) SET r.Date = datetime()")
void addToFollowedCelebrities(String userId, String celebrityId);
```

```
@Aggregation(pipeline = {
        "{ $match: { _id: ?0 } }",
        "{ $lookup: { from: 'Celebrities', let: { celebrityId: ?1 }, pipeline: " +
                "[ { $match: { $expr: { $eq: ['$_id', '$$celebrityId'] } } } ], as: 'followedCelebrity' } }",
        "{ $set: { followedCelebrity: { $first: '$followedCelebrity' } } }",
        "{ $set: { followed_celebrities: { $concatArrays: ['$followed_celebrities', " +
                "[{ person_id: '$followedCelebrity._id', name: '$followedCelebrity.name', " +
                "poster: '$followedCelebrity.Poster'}]] } } }",
        "{ $set: { followed_celebrities: { $slice: ['$followed_celebrities', -5] } } }",
        "{ $merge: { into: 'Users', whenMatched: 'merge', whenNotMatched: 'insert' } }"
})
void addToFollowedCelebrities(String userId, int celebrityId);
```

Remove Followed Celebrity

This method removes a celebrity from the user's followed list. It first checks If the user and the celebrity exist in their respective databases (Neo4j). If the celebrity is not followed by the user, an error is thrown. If all checks pass, the celebrity is removed from the user's followed list in both Neo4j and MongoDB, and the celebrity's follower count is decreased in the MongoDB repository. Any errors encountered during the process are logged and followed by a RuntimeException.

```
@Transactional
public void removeFollowedCelebrity(String username, int celebrityId) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!celebrityNeoRepository.existsById(String.valueOf(celebrityId))) {
        throw new NoSuchElementException("Celebrity '" + celebrityId + "' doesn't exists");
    } else if (!neoRepository.isCelebrityFollowed(username, String.valueOf(celebrityId))) {
        throw new IllegalArgumentException("Celebrity '" + celebrityId + "' isn't followed");
    }
    try {
        List<UserCelebrity> newCelebrities = neoRepository.removeFromFollowedCelebrities(
                username,String.valueOf(celebrityId));
        mongoRepository.removeFromFollowedCelebrities(username,newCelebrities);
        celebrityMongoRepository.decreaseFollowers(celebrityId);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

The RemoveFromFollowedCelebrities Queries delete the relationship between the user and the movie and sets the followed_celebrities array in the user documenta as the five most recent celebrities followed.

```
@Query("MATCH (u:User)-[r:ACTOR_FOLLOW]->(m:Celebrity) " +
        "WHERE u.username = $userId AND m.person_id = $celebrityId DELETE r WITH u " +
        "MATCH (u)-[r2:ACTOR_FOLLOW]->(m2:Celebrity) " +
        "ORDER BY r2.date DESC LIMIT 5 RETURN m2.name AS name, toInteger(m2.person_id) AS celebrityId ")
List<UserCelebrity> removeFromFollowedCelebrities(String userId, String celebrityId);
@Query("{ '_id': ?0 }")
@Update("{ $set: { 'followed_celebrities': ?1 } }")
void removeFromFollowedCelebrities(String userId, List<UserCelebrity> celebrities);
```

Follow a User

This method enables a user to follow another user. It first checks if both users exist in the system. If they do, it verifies if the user is already following the target user and ensures that a user cannot follow themselves. If the conditions are met, a “FOLLOW” relationship is created in the Neo4j database, and the follower and following counts are updated in both the MongoDB and Neo4j databases.

```
@Transactional
public void addFollowedUser(String username, String followedUsername) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!neoRepository.existsById(followedUsername)) {
        throw new NoSuchElementException("User '" + followedUsername + "' doesn't exists");
    } else if (neoRepository.isUserFollowed(username, followedUsername)) {
        throw new IllegalArgumentException("User '" + followedUsername + "' is already followed");
    } else if (username.equals(followedUsername)) {
        throw new IllegalArgumentException("Users can't follow itself");
    }
    try {
        neoRepository.followUser(username,followedUsername);
        mongoRepository.increaseFollowed(username);
        mongoRepository.increaseFollowers(followedUsername);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

Unfollow a Users

This method allows a user to unfollow another user. It first verifies that both users exist and that the user is already following the target user. If these conditions are satisfied, the "FOLLOW" relationship is removed from the Neo4j database, and the follower and following counts are updated in both MongoDB and Neo4j. If any errors occur during the process, an exception is thrown, and the operation is rolled back.

```
@Transactional
public void removeFollowedUser(String username, String followedUsername) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    } else if (!neoRepository.existsById(String.valueOf(followedUsername))) {
        throw new NoSuchElementException("User '" + followedUsername + "' doesn't exists");
    } else if (!neoRepository.isUserFollowed(username, followedUsername)) {
        throw new IllegalArgumentException("User '" + followedUsername + "' isn't followed");
    }
    try {
        neoRepository.unfollowUser(username,followedUsername);
        mongoRepository.decreaseFollowed(username);
        mongoRepository.decreaseFollowers(followedUsername);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

Get Followed Users

This method retrieves a paginated list of users followed by the specified user in Neo4j. It first checks if the user exists in the database; if not, it throws an exception. Then, using pagination, it fetches the followed users from Neo4j. In case of any error during the process, a RuntimeException is thrown.

```
public Slice<UserNeo4J> getFollowersUsers(String username,int page, int size) {
    if (!neoRepository.existsById(username)) {
        throw new NoSuchElementException("User '" + username + "' doesn't exists");
    }
    try {
        Pageable pageable = PageRequest.of(page, size);
        return neoRepository.getFollowers(username,pageable);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

4.5.4. Reviews

Create a Review for a Movie

This method allows a user to add a review for a movie. It first checks if both the user and the movie exist in the database. If they do, it creates a new review in both MongoDB and Neo4j, associating the review with the user and the movie. Additionally, it updates the user's recent reviews and the movie's review count. If any error occurs during the process, the review is not added, and an exception is thrown.

```
@Transactional
public ReviewMongoDB addReview(String movieId, String userId, String txt, boolean sentiment) {
    if (!userMongoDBRepository.existsById(userId)) {
        throw new NoSuchElementException("User '" + userId + "' doesn't exists");
    } else if (!movieMongoDBRepository.existsById(movieId)) {
        throw new NoSuchElementException("Movie '" + movieId + "' doesn't exists");
    }
    try {
        ReviewMongoDB review = new ReviewMongoDB(txt, sentiment, movieId, userId);
        ReviewNeo4J review1 = new ReviewNeo4J(review.get_id(),sentiment);
        reviewMongoRepository.save(review);
        reviewNeo4JRepository.save(review1);
        reviewNeo4JRepository.setUser(review.get_id(),userId);
        reviewNeo4JRepository.setMovie(review.get_id(),movieId);
        userMongoDBRepository.setRecentReview(userId);
        movieMongoDBRepository.updateReviews(movieId);
        return review;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

SetRecentReview replaces the user's last recorded review with the most recent one they have submitted.

```
@Aggregation(pipeline = {
        "{ '$match': { '_id': ?0 } }",
        "{ '$lookup': { 'from': 'Reviews', 'let': { 'username': '$_id' }, " +
                "'pipeline': [{ '$match': { '$expr': { '$eq': [ '$username', '$$username' ] } } } , " +
                "{ '$sort': { 'timestamp': -1 } }, { '$limit': 1 }, " +
                "{ '$lookup': { 'from': 'Movies', 'localField': 'movie_id', 'foreignField': '_id', 'as': 'movie' } }, " +
                "{ '$unwind': '$movie' }, { '$project': { 'review_id': '$_id', 'movie_title': '$movie.title', " +
                "'sentiment': '$sentiment', 'content': '$review' } } ], 'as': 'recent_review' } }",
        "{ '$set': { 'recent_review': { '$arrayElemAt': [ '$recent_review', 0 ] } } }",
        "{'$unset': 'recent_review._id'}",
        "{ '$merge': { 'into': 'Users', 'on': '_id', 'whenMatched': 'replace', 'whenNotMatched': 'discard' } }"
})
void setRecentReview(String userId);
```

Delete a Review

This method deletes a review by its unique id from both MongoDB and Neo4j databases. It first retrieves the review using the getReviewById method. If the review doesn't exist, a NoSuchElementException is thrown. Upon successful deletion, it also updates the recent review for the user and refreshes the movie's reviews in MongoDB. If any errors occur during the deletion process, a RuntimeException is thrown to handle the issue.

```
@Transactional
public void deleteReview(String id) {
    ReviewMongoDB review = getReviewById(id);
    if (review == null) {
        throw new NoSuchElementException("Review '" + id + "' doesn't exists");
    }
    try {
        reviewMongoRepository.deleteById(id);
        reviewNeo4JRepository.deleteById(id);
        userMongoDBRepository.setRecentReview(review.getUsername());
        movieMongoDBRepository.updateReviews(review.getMovie_id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Update Review

This method allows updating the text of a review identified by its unique id. It first retrieves the review using the getReviewById method, and if it doesn't exist, a NoSuchElementException is thrown. Once the review is found, it updates the review text and the timestamp of the review in MongoDB. Additionally, it refreshes the recent review for the user and updates the movie's review details. If any error occurs during the process, a RuntimeException is thrown to handle the exception.

```
@Transactional
public void updateReview(String id, String txt) {
    ReviewMongoDB review = getReviewById(id);
    if (review == null) {
        throw new NoSuchElementException("Review '" + id + "' doesn't exists");
    }
    try {
        LocalDateTime time = LocalDateTime.now();
        reviewMongoRepository.updateReview(id,txt,time);
        userMongoDBRepository.setRecentReview(review.getUsername());
        movieMongoDBRepository.updateReviews(review.getMovie_id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Like a Review

This method allows a user to like a review. It first checks if the review and the user exist in the database. If the review or the user does not exist, a NoSuchElementException is thrown. Additionally, if the review has already been liked by the user, an IllegalArgumentException is raised. If all conditions are satisfied, the review is liked by the user by updating both the MongoDB and Neo4j databases. Any exceptions encountered during the process are caught and thrown as a RuntimeException.

```
@Transactional
public void likeReview(String id,String userId) {
    if (!reviewMongoRepository.existsById(id)) {
        throw new NoSuchElementException("Review '" + userId + "' doesn't exists");
    } else if (!userMongoDBRepository.existsById(userId)) {
        throw new NoSuchElementException("User '" + userId + "' doesn't exists");
    } else if (reviewNeo4JRepository.isReviewLiked(userId,id)) {
        throw new IllegalArgumentException("Review '" + id + "' is already liked");
    }
    try {
        reviewNeo4JRepository.likeReview(id,userId);
        reviewMongoRepository.likeReview(id);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Unlike a Review

This method allows a user to unlike a review. It first checks if the review and the user exist in the database. If either the review or the user is missing, a NoSuchElementException is thrown. Additionally, if the review has not been liked by the user, an IllegalArgumentException is raised. If all conditions are met, the review is unliked by the user, updating both MongoDB and Neo4j databases. Any errors that occur during the process are caught and thrown as a RuntimeException.

```
@Transactional
public void unlikeReview(String id,String userId) {
    if (!reviewMongoRepository.existsById(id)) {
        throw new NoSuchElementException("Review '" + userId + "' doesn't exists");
    } else if (!userMongoDBRepository.existsById(userId)) {
        throw new NoSuchElementException("User '" + userId + "' doesn't exists");
    } else if (!reviewNeo4JRepository.isReviewLiked(userId,id)) {
        throw new IllegalArgumentException("Review '" + id + "' isn't liked");
    }
    try {
        reviewNeo4JRepository.unlikeReview(id,userId);
        reviewMongoRepository.unlikeReview(id);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

4.5.5. Manager

Check Database Inconsistencies in MongoDB and Neo4j

These two methods are used to identify inconsistencies between Neo4j and MongoDB. They check if one of the databases contains outdated data, such as missing documents in MongoDB or missing nodes in Neo4j. These methods ensure that both databases remain synchronized and accurate.

```
public List<String> inconsistenciesNeo() {  1 usage   matteo
    List<String> mongoDb=movieMongoDBInterface.findAllIds().getAllIds();
    List<String> neo4j=movieNeo4jRepository.findAllIds();
    return findDifference(neo4j, mongoDb);
}
public List<String> inconsistenciesMongo() {  1 usage   matteo
    List<String> mongoDb=movieMongoDBInterface.findAllIds().getAllIds();
    List<String> neo4j=movieNeo4jRepository.findAllIds();
    return findDifference(mongoDb, neo4j);
}
```

5. DATA REPLICATION

Consideration on CAP theorem

One of the main features of the platform, being a social-media-like application, is to guarantee the availability of the data in any moment, so that the users can stay engaged into the platform not only to interact with other people, but also to discover and find titles and celebrities. We thought that, in our case, guaranteeing the availability of our data is more important than the consistency of it, since we don't store sensible information about users or about the titles, that must be consistent every time someone accesses it. So, in conclusion, to keep our data always available we are willing to pay the price of losing consistency.

With reference to the CAP theorem, we choose the AP solution (Availability and Partition tolerance), to guarantee our platform to be accessible even during network failures.

For implementing the replication in our system, we had at our disposal three virtual machines with addresses: 10.1.1.13, 10.1.1.11, 10.1.1.10. The main structure of the nodes is composed by a primary node and two secondary nodes and is illustrated in the following table.

MongoDB NODE	**Neo4j NODE**	**ADDRESS**	**USED FOR**	**PRIORITY**
Primary	-	10.1.1.13	Collecting writes and receive read requests	5
Secondary	-	10.1.1.11	Store replication of the primary node and receive read requests	2
Secondary	Primary	10.1.1.10	Store replication of the primary node and receive read requests.	1

MONGODB REPLICATION

Read-preferences: nearest
It can guarantee a good request distribution and then a good load balance. Sending all the requests to the primary node can be not optimal because this node may have a lot of incoming requests especially during moments of the day where the traffic in the platform is at its highest. So, since we can tolerate inconsistent data, we can distribute the incoming requests to the nearest node (primary or secondary) to the user, based on latency, so that the response time is kept acceptable for the user engagement, which is our main aim.

Write-concern: W1
We decided to return the control to the application after the write acknowledgment from the primary node. We preferred to do so instead of waiting for the ack from at least two members (using majority or w2) because we want to guarantee a fast responsiveness of the platform, and we thought that it was not necessary to store the write in more members because it could slow down the system.

NEO4J REPLICATION

We deployed our database in a different virtual machine instead of the same primary node used for MongoDB for two reasons:

- Avoid a single point of failure: we wanted to avoid having both Neo4j and MongoDB databases in the same node because if this node goes down, we lose both primary nodes
- Having a better requests distribution: since Neo4j is principally used for recommendation systems while MongoDB for search queries we thought that having the two principal members of the databases in two different nodes was a better solution for the writing requests.

As we can see in the table, the primary node for Neo4j is deployed on the third virtual machine (10.1.1.10) which is the one with the lowest priority for MongoDB, so that if the primary node of MongoDB (10.1.1.13) has a failure, the new primary selected will probably be the second virtual machine (10.1.1.11) which has a higher priority. In this way we try to handle the requests traffic in a balanced way by not having both the primary members in the same node.

6. SHARDING STRATEGY

We have selected the potential sharding keys based on the most frequently executed queries for each collection. Our goal is to optimize query performance and ensure an even distribution of data across shards.

Movies

The most frequent queries on the movie dataset include:

- Finding a movie by _id
- Text search on movie titles and keywords
- Filtering movies by attributes such as year, votes, score, country, and platform
- Adding a movie to a watchlist

Among these, filtering queries would benefit the most from sharding. However, sharding by release_year would concentrate new movies on a single shard, and imdb_score is not evenly distributed, with most scores clustering between 6 and 7.

For this reason, the most suitable sharding keys are:

- imdb_id with hashing to ensure an even distribution.
- Movie title by alphabetical range, which could help balance queries related to text search.

Celebrities

The most frequent queries include:

- Finding a celebrity by _id
- Text search on celebrity names and roles

Since text search still requires scanning the jobs array, sharding does not significantly improve performance for these queries. The best options are still:

- Alphabetical sharding by name
- Hashing by _id to maximize even distribution

Users

The most frequent queries are:

- Finding users by _id
- Text search by username and name

Like celebrities, none of these queries benefit much from sharding. The best approaches we would opt for:

- Alphabetical sharding by name
- Hashing by _id, ensuring an even distribution across shards

Reviews

The most frequent queries include:

- Retrieving reviews by movie_id
- Retrieving reviews by username
- Posting a new review

Since queries are more likely to retrieve reviews by movie_id, the optimal sharding key is:

- movie_id, ensuring reviews are stored close together for a given movie.

Posts

The most frequent queries include:

- Retrieving posts by movie_id
- Retrieving posts by username
- Creating a new post

Like reviews, movie_id is expected to be queried more frequently. The best sharding approach is:

- movie_id with hashing, distributing posts evenly across shards.

Comments

The most frequent queries include:

- Retrieving comments by post_id
- Retrieving comments by username
- Creating a new comment

Since comments are more likely to be retrieved based on the associated post, the best sharding key is:

- post_id with hashing, ensuring comments are evenly distributed and efficiently retrieved.

This strategy ensures that data is evenly distributed while optimizing performance for the most frequently executed queries.