Engineering Department

Master's Degree Artificial Intelligence and Data Engineering


Università degli studi di Pisa

Accademic year - 2023-2024

# Large-Scale and Multi-Structured Databases Project: WatchParty

https://github.com/nikisetti01/LargeScaleProject2024

STUDENTS:

**Comini Matteo – Settimelli Niccolò - Vittori Lorenzo**

# Contents

# 1 Introduction

Welcome to WatchParty, your go-to web application for a next-level cinematic adventure! Immerse yourself in a world where movie enthusiasts unite to discover, discuss, and celebrate the magic of film. Search for your favourite movies or uncover hidden gems, read insightful user reviews, and share your own critiques to contribute to a vibrant community.

With WatchParty, you can effortlessly build a personalized watchlist, keeping track of films you are eager to explore or have already enjoyed. Exchange recommendations, and create a social hub for your cinematic interests. Additionally, delve into film communities where engaging discussions, themed groups, and diverse perspectives await.

WatchParty employs Python for data cleaning. Our database strategy integrates MongoDB for flexible document management and Neo4j for intricate graph relationships. The web app, developed with CSS, Php and JavaScript, it’s going to provide a simple and easy to use user interface.

# 2 Requirements

## 2.1 Functional requirements - Unregistered user

- The system must allow an unregistered user to become a registered user by registering Username, Display Name, age, password and city.
- The system must allow an unregistered users to search for information pages about films and read reviews from other registered users.

## 2.2 Functional requirements - Registered user

- The system must allow a user to login (by username and password) to the system.
- Once logged in the system must display the user main page.
- The system must allow a user to perform logout process.
- The system must allow a user to search for information pages about films and read reviews from other registered users.
- The system must allow a user to add a film to his watchlist.
- The system must allow a user to navigate through his watchlist
- The system must allow a user to removing film or Tv series from his watchlist.
- The system must allow a user to review a film or Tv series.
- The system must allow a user to follow a community.
- The system must allow a user to not follow a community previously followed.
- The system must allow a user to post in a community.
- The system must allow a user to comment a post in a community to which he belongs.
- The system must allow a user to navigate through another user main page to see his watchlist.

## 2.3 Functional requirements – Administrator

- The system must allow administrator user to login (by username and password) to the system.
- The system must allow administrators user to explore analytics in a specific page.
- The system must allow administrators user to search and delete a registered user.
- The system must allow administrators user to delete a post in a community.
- The system must allow administrators user to delete a comment from a post in a community.
- The system must allow administrators user to perform logout process.

From now on, users refer to Administrators and registered user.

### 2.6 Non-functional requirements

- The system must be a website application.
- The system must communicate with any user by using a secure communication channel (HTTPS).
- The system must present a large film and Tv series collection.
- The system must have an intuitive User interface.
- The system must be developed by using an OOP language (i.e.: Java, Python, etc.)
- The system must be able to run on at least the following main browsers: Google Chrome, Mozilla, Microsoft Edge.
- There must not be down time and must be fault tolerant

# 3 Statistics and queries

In the following section, we will delve into an exploration of the CRUD operations that shape the system's data management capabilities, supplying insights into how these essential processes empower users to create, retrieve, update, and remove data within the system. The implementation of CRUD operations plays a pivotal role in realizing the functional requirements, enabling seamless user interactions and ensuring the effective management of user profiles, content creation, and engagement within the platform.

### 3.1 CRUD operations

- **Create new reader**

  This query, provided by the system for the unregistered user allow them to sign up and create a new account.

- **Create new administrator**

  This query, provided by the system for administrators only allow them to create a new administrator profile.

- **Add a film to watchlist**

This query, provided by the system for registered users only allow them to add a film to his personal watchlist.

- **Remove a film to watchlist**

This query, provided by the system for registered users only allow them to remove a film, previously added, to his personal watchlist.

- **Remove a user**

This query, provided by the system for administrators only allow them to delete a user from the system.

- **Join a community**

This query, provided by the system for registered users only allow them to join a community.

- **Post into a community**

This query, provided by the system for registered users only allow them to post into a community

- **Comment a post**

This query, provided by the system for registered users only allow them to comment a post into a community

- **Looking a main profile**

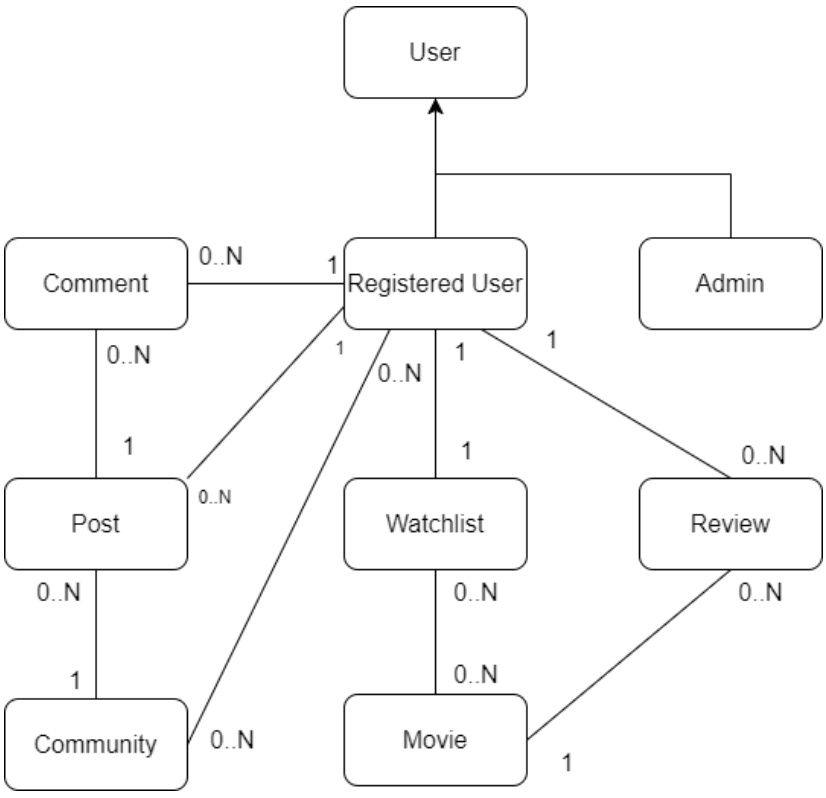
This query, provided by the system for any user allow it to search for another user main profile by username.

- **Delete a movie**

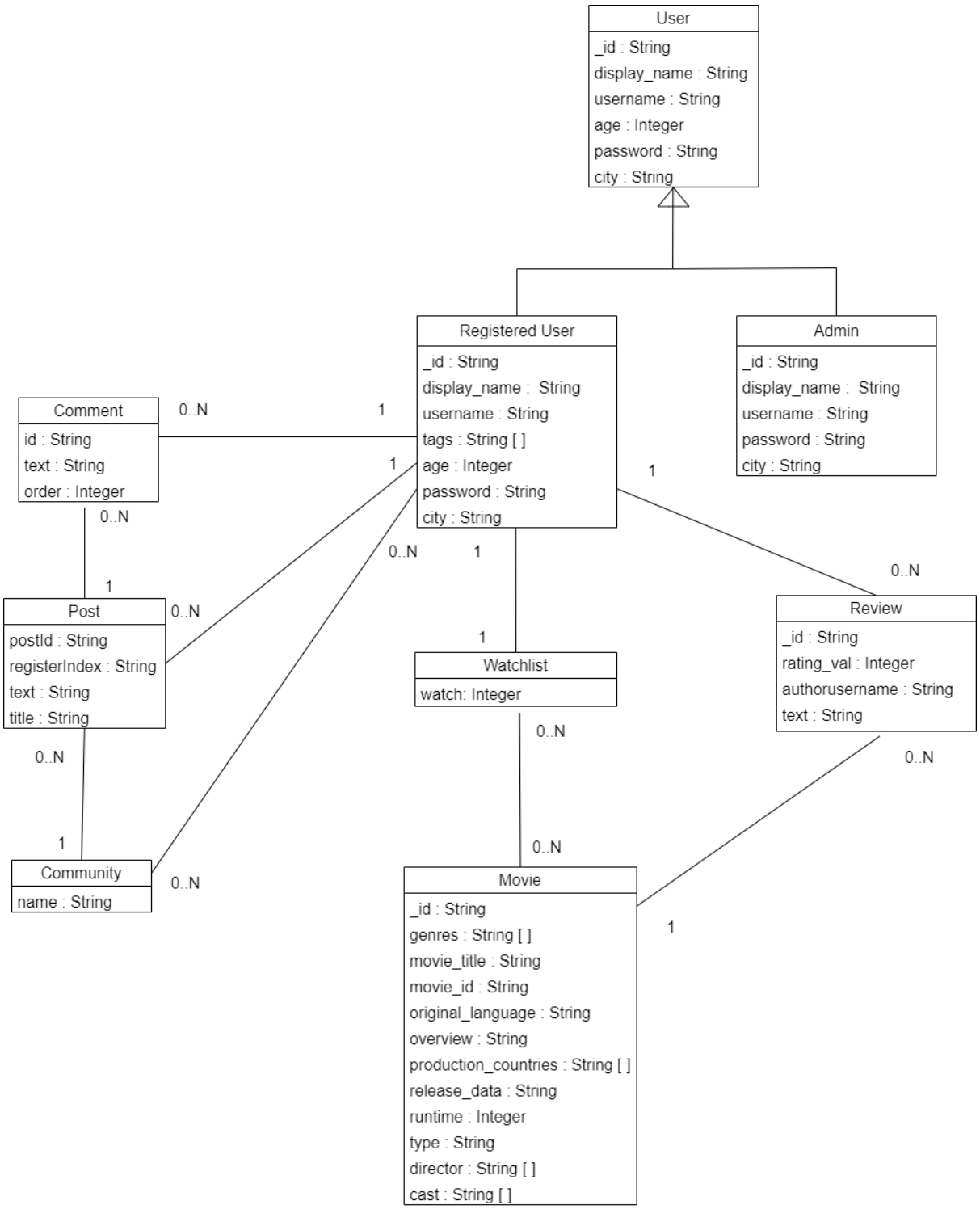
This query, allow to an admin to delete a movie. When an admin deletes a movie, its reviews are also deleted cascade-wise, and all of this happens synchronously.

# 4 UML Use case diagram


# 5 UML Class analysis



# 6 UML Class diagram



# 7 Database

Within our system architecture, we decide to adopt MongoDB (Document database) to handle user profiles, implement film reviews, and manage the watchlist functionality. MongoDB, a NoSQL document database, offers flexibility and scalability, making it an ideal choice for storing user-related data and facilitating efficient operations on profiles, reviews, and watchlists.

In parallel, we have chosen Neo4j, a graph database, to navigate the intricate relationships within communities and user interactions. Neo4j's graph structure excels in representing and querying complex relationships, making it an optimal solution for modelling and managing the dynamic connections between users in our platform's community features.

This dual-database approach ensures that our system is finely tuned to handle diverse data needs, providing an effective and responsive environment for both individual user experiences and community interactions

## 7.1 Document database

The entities that we decide to implement on the database are:

- User
    - Registered User
    - Admin
- Watchlist (embedded inside Registered User)
- Review
- Movie

**User**

The collection of users includes information about Registered Users (basically everyone who sign up) and Admin. It’s important to evidence that Admin informations are quite different respect to Registered Users information, for example Admins don’t need any type of Watchlist in their information

**Watchlist**

The watchlist encompasses the collection of movies that a user has either already viewed or plans to watch. While visually differentiated in front-end, in practical terms, these entries will be consolidated within the same embedded structure in user profiles, identifiable through the 'watch' Boolean. We chose to implement them as embedded entities rather than separate collections, given the strong association with individual user profiles. In practice, the watchlist will appear as an embedded array within a user, containing a series of collections as described above.

**Review**

The "review" entity represents a user's judgment or assessment of a specific movie within the system. Each review may include descriptive text (not obligatory) and numerical rate in a scale from 1 to 10.

Reviews serve as a crucial means of interaction, allowing users to share their experiences and providing informative and evaluative insights into the movies featured on the platform.

**Movie**

The movie entity encapsulates information about a specific movie within the system. It includes essential details such as the title, genre, release date, and a synopsis of the movie. Additionally, the entity stores related information like cast, crew, and other metadata, offering a comprehensive overview of each film available on the platform.

### 7.1.1 Collections

Considering the following guidelines:

- Arrange collections with consideration for the queries to be executed, eliminate dependencies between collections, and structure documents to streamline the workflow for each operation.
- Minimize the number of collections managed by the database as much as possible.

And taking into account the criteria previously mentioned in 4.1, it is concluded that the collections to be stored in the database will be **Users**, **Movies**, and **Reviews**.



### 7.1.2 Queries

The list of queries that will be written later pertains to the documents present in the document database:

**Crud operation**

- **Create**
  - User
  - Review
  - Movie
- **Read**
  - All users
  - User by username
  - User by _id
  - Movie by movie_id
  - Movie by movie_title
  - Review by movie_id
  - Review by author
  - Review by movie_title
- **Update**
  - Add new WatchlistItem to Watchlist
  - Delete a WatchlistItem from Watchlist
  - Change status from 'to watch' to 'watched'
  - Change user information
  - Change text and/or the rating value from a review
- **Delete**
  - User
  - Review
  - Movie

Specifically, the deletion of reviews and users is an operation exclusive to the admin and the user in question, while the deletion of movies (and also the addition) is an operation exclusive to the admin.

**Statistics queries**

The more complex queries were specifically designed for the type of database we intended to use, as they make the best use of the capabilities of the document database. Therefore, the following queries will be executed on the data contained in the document database:

- Count Users by City and Movie (to find the number of users who have watched a specific movie in a particular city)
- Count Movies by Age (count how many movies have been watched (watch = 1) in each age group. This can be useful to understand which age groups are more active in movie-watching)

- Find top 5 movies by reviews
- Number of a film review
- Histogram to visualize the number of movies reviewed by user rating
- Histogram to see the number of ratings received by a movie
- Calculate the total runtime of movies watched by a user
- Find 5 random review from the 5 top film

## 7.2 Graph DB

The part of the project that uses a graph database is used to store information about users, communities, posts, and comments. Our graph will have various types of nodes, which are User, Post, and Community. Among these three nodes, we have different types of relationships that connect the nodes to each other. The relationship between User and Community is called "JOIN" and represents the user belonging to the community. This allows them to read and utilize all the features offered by the Community, including Posting and Commenting. These are represented by the "POSTING" and "COMMENT" relationships, which signify the connection between a User and a Post. A User can repeatedly comment on any Post and can make as many posts as they want in the communities they are a part of. The last remaining relationship is related to the association between Posts and Communities, and it is called "BELONG," representing the presence of a post in a specific community.

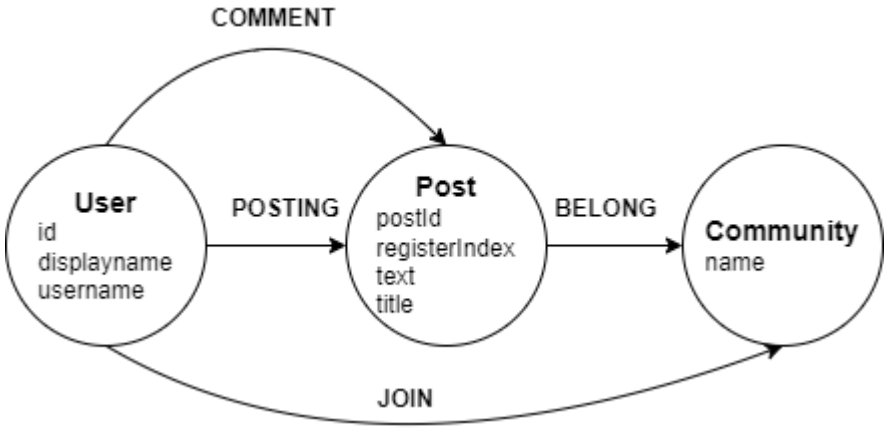
In particular, in the graph database, we describe our User with the following properties:

- id
- display_name
- username.

The post node, on the other hand, will have the following properties:

- postId
- registerIndex
- text
- title.

Finally, the community will only have the "name" attribute within it. This is because the addition of communities is entirely managed by the admin, and it is a sufficient field given the small number of communities. Additionally, since the primary function of communities is to serve as a link between all posts, we do not need to describe them with specific attributes. Another important detail to mention is that the "comment" relationship has an internal attribute, specifically the "text" field and “order”, which the first one holds the content of the comment and the second one maintain the casuality consistency of the user messages.




### 7.2.1 Queries

| CREATE | |
|---|---|
| **Domain-specific** | **Graph-centric** |
| Add a new User | Add new User vertex |
| Admin add a new Community | Admin add new Community vertex |
| User Join a Community | Add new “JOIN” relationship between User and Community |
| User write a new Post | Add new Post vertex , add a “POSTING” relationship between them and add an “BELONG” relationship between Post and Community |
| User write a new Comment | Add new “COMMENT” relationship between User and Post |

| READ | |
|---|---|
| **Domain-specific** | **Graph-centric** |
| Find all Community joined by a User | How many “JOIN” egde are outgoing from a specific User vertex? |
| Find all Posts made by User | How many “POSTING” edges are outgoing from a specific User vertex? |
| Find the User who wrote the Post | The User who initiates the ‘POSTING’ edge towards a specific Post |
| Find all the comments on a Post | All edges of type 'COMMENT' that point to a specific Post |

| DELETE | |
|---|---|
| **Domain-specific** | **Graph-centric** |
| Admin removes a User account | Remove the User node, all the Post nodes linked with “POSTING” relationship and all ingoing edges, all relationship “COMMENT” and “JOIN” between this and the other nodes |

| | |
|---|---|
| Admin removes a Community | Remove the 'Community' node, all connected 'Post' nodes, and recursively remove all 'COMMENT' and 'POSTING' edges directed towards those deleted 'Post' nodes. Also, delete all 'JOIN' relationships directed from 'User' to the community |
| Admin removes a Post | Remove a 'Post' node, all its 'COMMENT' and 'POSTING' connections, and the 'BELONG' relationship to the 'Community' |
| Admin removes a Comment | Remove a 'COMMENT' relationship between User and Post |
| User removes his Post | Remove a 'Post' node, all its 'COMMENT' and 'POSTING' connections, and the 'BELONG' relationship to the 'Community' |

Statistics queries

| **STATISTICS** | |
|---|---|
| **Domain-specific** | **Graph-centric** |
| Which User are the most active in a specific Community? | User vertex with highest number of “POSTING” and “COMMENT” edges as own activities. |
| Recommend a community to a user based on the communities of the user they have interacted with the most, where they are not already a member | Given a user, look at the Posts that are pointed to by the 'COMMENT' and 'POSTING' relationships. For the obtained Posts, find the one with the highest activity (count the number of 'COMMENT' and 'POSTING' relationships pointing to those Posts). Once the ideal user is found, return a Community node where the input user does not have a 'JOIN' relationship |
| Calculate the affinity between two users | Calculate the shortest path between two users, considering edges of type 'BELONG,' 'COMMENT,' and 'POSTING' relationships |

Some queries, such as the one that allows the user to delete a post and the admin to delete a community, are functional and tested in the back-end but not implemented in the front-end. We will discuss their functionality in Chapter 12, which pertains to the testing phase.

# 8 Redundacies

For optimal use of our application, following various considerations, we have pursued the idea that we prefer fast query execution over reduced memory consumption. This is achieved through the implementation of redundancies to avoid performing join operations. I will now discuss the redundancies present in the database.

### 8.1 Watchlist Redundancy

The redundancy of the Watchlist in the user collection is designed to provide quick access to the movies reviewed by the user, while adding other functionalities such as movies the user wants to watch. The Watchlist is presented as an embedded list within the User and consists of Watchlist items that will contain the name of the reviewed movie and the rating given in the respective review. To differentiate between watched movies and those to watch, a 'watch' field has been added, which serves as a flag to indicate to which category each movie belongs. This helps us avoid having the system execute time-consuming queries and operations every time a user accesses their profile. This redundancy allows us to distinguish between movies to watch and those already watched, and it also allows us to easily display in the specific user's profile the movies they have reviewed along with their ratings.

This redundancy is divided into two parts: one related to the movie document, and the other related to the review document.

- **Movie Redundacies:** The movie redundancy includes fields like movie_id, movie_title, and rating_val. It is important and allows us to store simple but useful information about the movie, facilitating our goal of identifying the movies that the user has seen and has yet to see.

- **Review Redundancies:** The review redundancies only include the presence of the rating assigned during the review. This data is obviously not present for movies that have not been watched yet (identified by the 'watch' field, which will be set to 0)."

### 8.2 Tags Redundancy

The tags contained as a list within the user indicate which groups the user is a part of. Their redundancy serves the purpose of quickly displaying, without querying Neo4j, the communities with which a user has a 'join' relationship. This will help us reduce calls and also serve as data replication, considering that the primary data is stored within the GraphDB.

### 8.3 Users Redundancy

Within the Neo4j database, there is redundancy of users, which is essential for the functioning of our communities. This redundancy is necessary because we need to know which users are part of the community and what posts and comments they write. This redundancy of users does not include all attributes but only the simplest ones, such as username, display_name, and user id.

We will delve deeper into the update of the consistency of these redundancies later in a separate chapter that discusses how the CAP theorem has been applied to our project.

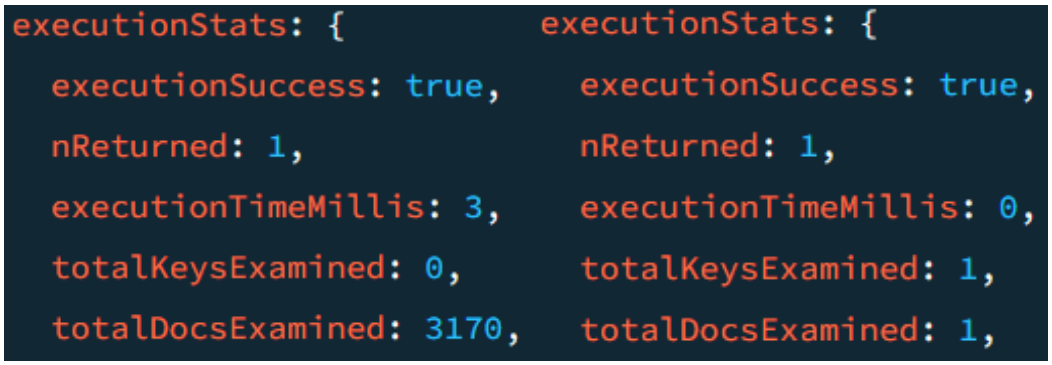
# 9 DocumentDB indexes and constrains design

Index management is a fundamental aspect of structuring a DocumentDB, as it allows for sorting the list of elements and facilitates their retrieval and manipulation in future queries. Indexes can be set up for each collection, and multiple indexes can also be established.

### User Index

For the index selection in the user collection, we have chosen the 'username' field to be sorted in ascending order since we perform all searches based on this field. This is because there cannot be two users with the same username, making it a unique field that can be used for searches within the collection. The crucial part is to have it sorted to optimize all queries in our collection. The strategy around the uniqueness of the username attribute has been applied to make the implementation of the web application more straightforward.

db.user.find({ username: "vinsim27" }).explain("executionStats")

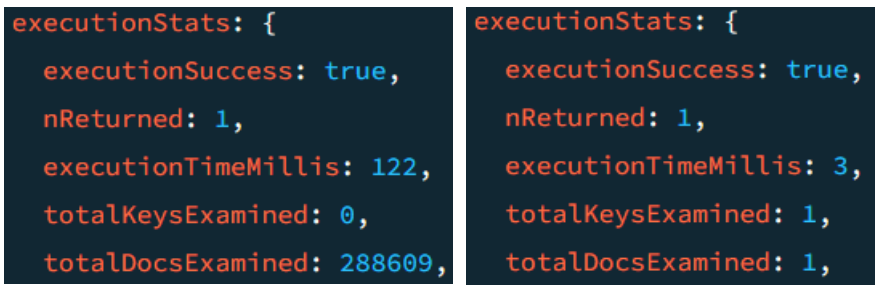



(performance without and with username index)

### Movie Index

For the index selection in the movie collection, we have adopted a similar approach to that of users. We have set the index on 'movie_id' since it also serves as an identifier. Inside 'movie_id,' the movie's name is contained, but it remains a field where duplicates are not allowed because additional characteristics are added (for example, in our movie list, there are many movies with the title "Dracula," but they all have different 'movie_id's, such as "dracula-1974," etc.). This index is also useful for all specific movie search functions.

db.movie.find({ movie_id: "dracula-1974" }).explain("executionStats")



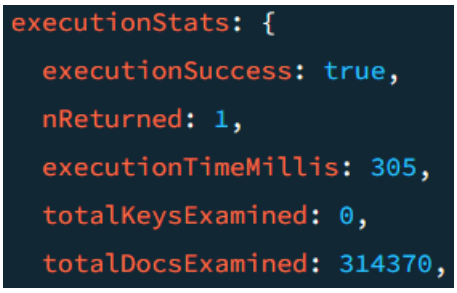
(performance with and without index)

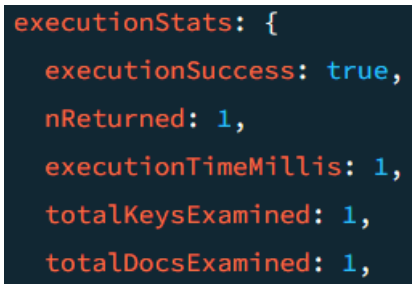
### Review Index

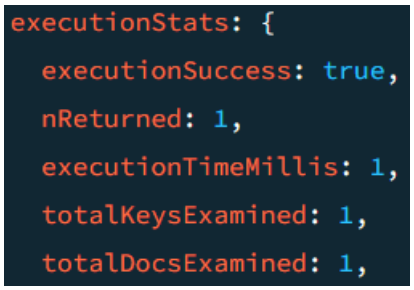
The case of reviews is special since it contains redundancies of movies. We will sort by 'movie_id' to obtain the reviews of a specific movie as quickly as possible. Furthermore, we will use a compound index consisting of 'movie_id' and 'rating_val' in descending order. This way, we can easily find the desired movie and facilitate the search for its best reviews. By ordering as the second index by 'rating_val' in descending order, we obtain the top reviews as the first results. This second index will optimize the query for finding the top 5 movies with their 5 best reviews.

The choice to use 'movie_id' as the index rather than 'authorusername' is based on the fact that we are not interested in searching for reviews by author since we already have their reviews in the 'watchlist' field within the user collection. This leads us to choose 'movie_id' as the primary index.

db.review.find({“movie.movie_id”:"dracula-1974"}).explain("executionStats")



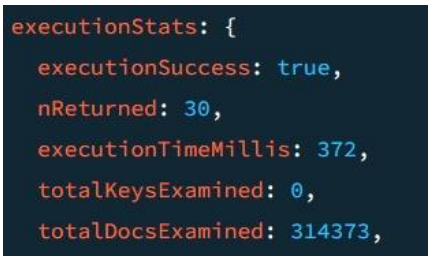



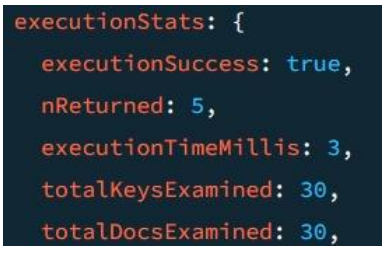

(performance without index, with movie_id index and with compound index movie_id and rating_val)

db.review.find(

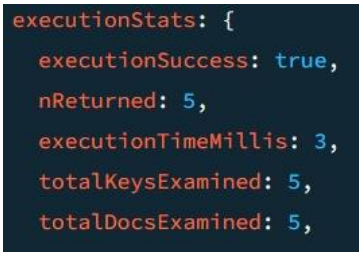
{ "movie.movie_id": "on-the-basis-of-sex" },

{ _id: 1, rating_val: 1, authorusername: 1, "movie.movie_id": 1, "movie.movie_title": 1
}).explain(“executionStats”)




This is the query on which the aggregation is based to determine the top 5 films with the highest average ratings and also retrieve the top 5 best reviews.



# 11 System architecture

The decision regarding the architecture to adopt is a crucial aspect in the development of an application. To ensure flexibility and foresight, it is essential to describe the architecture in general terms before tying it to specific technological implementations. This approach ensures that architectural choices are not overly tied to technologies that may become obsolete or unsuitable for the future needs of the application.

## 11.1 General Description

The system's structure is designed to meet the previously detailed requirements, with the goal of reducing development complexity by leveraging standardized components that provide common functionality. The heart of the architecture is represented by data management, a topic extensively discussed in the preceding chapters. In particular, attention is focused on data distribution and replication, which are fundamental aspects, although the final configuration will be influenced by available resources and will be detailed in a dedicated section.

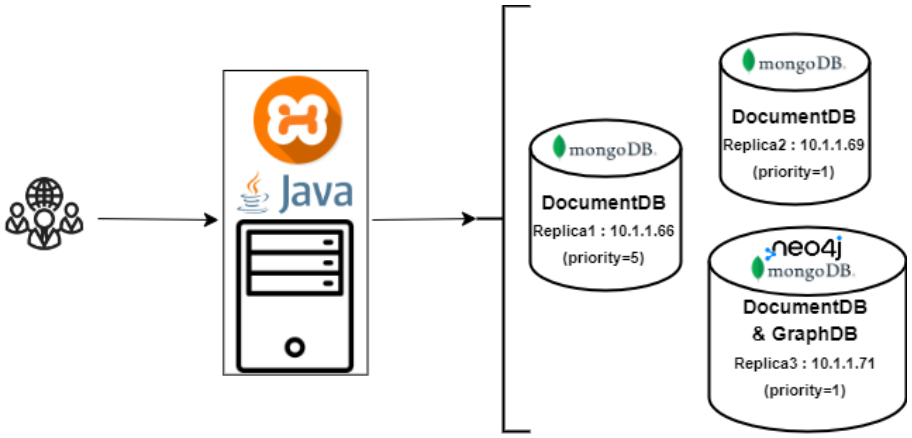
The system uses a graph database and a document database with three replicas of the latter. Each replica holds all the data, ensuring correct operation of our application in case of failures. The replicas are not all at the same level; we have one primary replica with priority 5, while the other two have priority 1. In the event of a failure of the primary replica, one of the two secondaries will become the primary to ensure correct and fast operation.

## 11.2 Framework and Components

Our application is structured into various layers: user interface, application logic, and data persistence. The user interface utilizes standard web technologies that provide a simplified and intuitive interface for users.

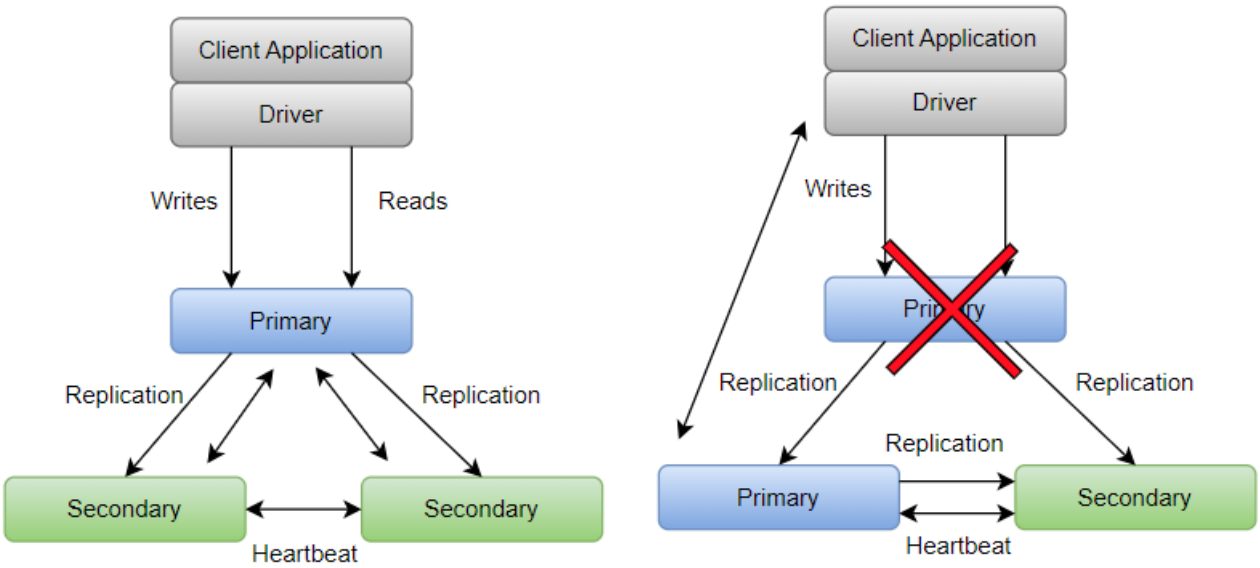
For the server-side, in the development environment configuration, we use xampp, which allows us to have our web application locally, while for the back-end, we use the Java programming language.

Finally, MongoDB and Neo4j have been chosen as databases to manage our data. MongoDB contains the more "informative" part of the application, such as user data, movie data, and their reviews, while Neo4j, in addition to containing user redundancy, also houses the more social-network aspect of our project. Replicas have been designed only for the document database, but in the future, they could be extended to the data contained in Neo4j.




## 11.3 Replica

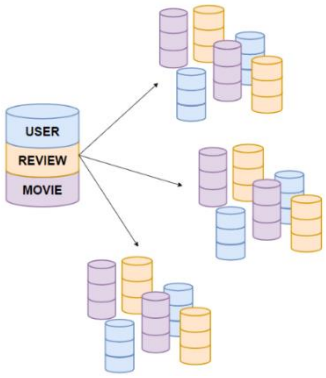
In our MongoDB replica set configuration, we have one primary node and two secondary nodes. The primary node is the core of the replication architecture: it is the only node that handles write operations, ensuring data integrity and consistency. The secondary nodes, on the other hand, synchronize with the primary by replicating write operations after they have been applied to the primary. Within this configuration, we can assign different priorities to the various nodes to influence the primary node election process in case of failover. For example, if we have a primary node with an assigned priority of 5, this indicates "high priority." This means that during the election of the primary node, the node with the highest priority is more likely to be chosen as the new primary if the current one were to fail or become inaccessible. The secondary nodes, with a priority of 1, can receive and serve read operations, offloading the load on the primary node and providing data redundancy. This mechanism is particularly useful to ensure that the system remains operational and accessible even in the presence of high workloads or hardware failures. Although secondary nodes with lower priority can become primaries, they typically do so only if the node with the highest priority is unavailable. It is important to note that despite redundancy and load distribution, all writes still need to pass through the primary node to maintain the atomicity of operations. This can be a bottleneck in systems with extremely high write volumes, but it is a necessary trade-off to maintain data consistency. In conclusion, MongoDB's replica set configuration offers a balance between performance, availability, and consistency, allowing systems to remain



robust and resilient in the face of failures and interruptions, while maintaining centralized write management to avoid conflicts and ensure data integrity.

| Component | Virtual Machine | Priority | IP address |
|---|---|---|---|
| MongoDB #1 | largescale2023 | 5 | 10.1.1.66 |
| MongoDB #2 | largescale2023 | 1 | 10.1.1.69 |
| MongoDB #2 | largescale2023 | 1 | 10.1.1.71 |
| Neo4jDB | Largescale2023 | - | 10.1.1.71 |

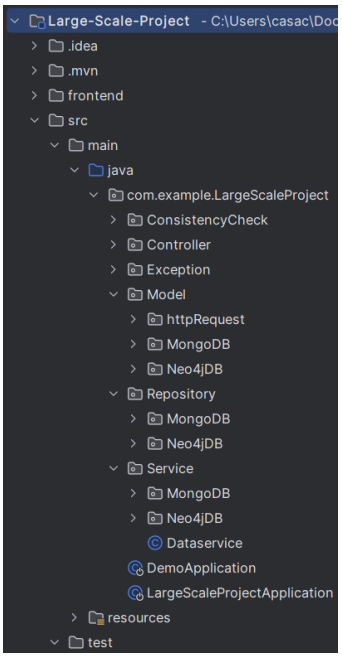
## 11.4 Sharding




Our approach to implementing sharding is based on user and age as a secondary key, and it is a strategic decision that reflects careful consideration of user behavioral characteristics and their interactions with the database. The decision to use age as one of the sharding criteria is particularly clever because it leverages the natural tendencies of similar demographic groups, which often exhibit similar behavioral patterns, such as a preference for certain movie genres. This not only can enhance query accuracy but can also expedite their execution, as relevant reviews and movie data will be physically closer within the same shard or adjacent shards. Hash-based partitioning of users creates a uniform distribution of data across shards, reducing the risk of hotspots where one shard might be overloaded compared to others. This method of partitioning, when executed thoughtfully, can significantly contribute to the overall system efficiency by balancing the load and maximizing performance. Additionally, cascading the reviews and movies based on user age groups establishes a logical association between the data and is a strategy that keeps related data together, facilitating operations that often involve analyzing relationships between users, their reviews, and corresponding movies. This can not only reduce query latency but also simplify data management because reviews and movies that are inherently linked are positioned close to the relevant user data. This approach would speed up access when queries are made for fields such as age and username, just like our admin-side query called countMoviesByAgeGroup(). The configuration and maintenance of a sharding system require continuous monitoring and proper resource management. If implemented and managed correctly, this approach can not only improve system performance but also provide a scalable foundation for future growth, ensuring that the database can effectively serve user needs as data volume increases.

# 12 Implementation

The system has been developed faithfully following the specifications outlined in the design phase. To conclude, we will analyze the most important queries and provide specific insights into redundancy handling within the system. The system has been developed faithfully following the specifications outlined in the design phase. We will proceed to present the system using a bottom-up approach, starting from the Java classes that serve as the foundation of the project, moving on to the repositories where the queries, which will be discussed in more detail in the following chapters, are located. There will also be a description of the exception modules used to maintain the system's integrity. Subsequently, we will describe the services responsible for implementing the core functions upon which the controllers are based. The controllers, in turn, handle communication with the front-end.




## 12.1 Spring boot

Our project is built on the foundation of Spring, a framework used to initialize the project and establish communication between databases and the back-end. It is a tool that simplifies and accelerates the development of web applications and microservices through three main features: automatic configuration, the adoption of a declarative approach to configuration, and the ability to create standalone applications.

## 12.2 Model

The system's models are divided into three main categories: MongoDB, Neo4j, and httpRequest. This organization has been chosen to ensure clear and orderly management of different entities and data within our system.

In the MongoDB category, you will find basic models related to entities also present in our database, such as "movie" and "registered user." Additionally, we have created specific classes to receive the results of our more complex aggregation functions. These models are designed to facilitate access and manipulation of MongoDB-related data within the application.

The Neo4j directory contains models related to entities present in the Neo4j graph database, such as "community." Similarly, we have implemented classes to receive the results of more complex queries within the Neo4j database. These models are essential for effectively managing interactions with the graph database and obtaining specific information from linked entities.

Finally, the httpRequest category houses models dedicated to manipulating data from the "model" class to make it more understandable and suitable for visualization by the front-end.

This organization of models allows us to maintain a clean and modular structure within our system, facilitating data management and access based on the different needs of various parts of the application.

## 12.3 Repository

The repository modules, both for Neo4j and MongoDB, constitute an essential component of our system, facilitating interaction with databases through Spring Boot. These modules harness the power of Spring Boot to communicate with the database and retrieve data efficiently. We use annotations like @Query to define custom queries and @Aggregate to implement complex aggregation operations, enabling flexible data management. Thanks to Spring Boot and its repository features, we can optimize data access and ensure high performance in our project.

Our repository modules are extensions of Spring Boot's MongoDB and Neo4j libraries, offering a simplified implementation of CRUD (Create, Read, Update, Delete) operations. These extensions allow us to easily perform common tasks, such as finding a user by ID or managing entities in the database. Through the integration of Spring Boot's repository capabilities with our databases, we streamline the data access process, improving efficiency and maintainability in our system.

## 12.4 Service

The service modules play a crucial role in our system as intermediaries between the user interface and data access via the repository modules. Their primary function is to conduct checks and potentially raise exceptions before performing operations on the data through the repository modules. These checks ensure data validity and apply specific business logic prior to database operations. Once the checks are satisfied, the service modules coordinate data access operations through the repository modules, ensuring consistent and compliant execution with the application's requirements. This communication between the modules is facilitated through a dependency configuration. When a service module needs to access or manipulate data, it utilizes the configured dependencies associated with the repository modules, which may involve calling repository methods to interact with the database. The repository modules, in turn, execute the requested operations on the data, interacting directly with the underlying database. Once the operations are completed, the results are returned to the service module, which can then further process them or pass them to the appropriate part of the application. The service modules act as a bridge between the user interface and data access, enabling organized management of the business logic within our system.

### 12.4.1 Dataservice

The Dataservice class is the central component of our Spring-based Java application because it communicates with two different databases: MongoDB and Neo4j. Its main function is to provide a service interface for user management, movie management, reviews, watchlists, and communities.

Let's start with dependency injection. The Dataservice class is annotated with @Service, which means it is a Spring service component and is managed by the Spring framework. Inside the class, there are various fields annotated with @Autowired, which are used to inject dependencies into other service classes. These dependencies include services for MongoDB and Neo4j, which will be used to communicate with their respective databases. Each method of the Dataservice class performs a specific task:

- User registration and deletion are managed in both the MongoDB and Neo4j databases. This means that when a user is registered, corresponding records are created in both databases. Similarly, when a user is deleted, records are removed from both databases.
- User search is performed in the MongoDB database, allowing you to find a user based on their username.
- Movie management involves searching for movies by title or ID in the MongoDB database.
- Access control is carried out by comparing the provided credentials with those stored in the MongoDB database.
- User information can be retrieved from the MongoDB database and returned in an InfoRequest object.
- Watchlist management involves adding movies to the list of movies to watch and retrieving these movies from the watchlist. These operations involve the MongoDB database.
- Reviews are handled by saving new reviews and displaying them. These operations also involve the MongoDB database.
- The class uses a cache (cachedRatings) to store user reviews to avoid recalculating them multiple times.
- A review histogram is calculated by counting the number of reviews for each rating value. This histogram is used to display review statistics.
- The ability to retrieve the top movies based on the number of reviews and average ratings is also provided, using the MongoDB service.
- The class can retrieve posts and their related comments from a specific community in the Neo4j database. These operations involve retrieving complex data from the Neo4j database.
- Additionally, you can add new posts to the community through the addPost method.

Therefore, the Dataservice acts as an intermediary between the application and the databases, providing a set of high-level operations for data management. Thanks to

dependency injection, it can efficiently communicate with the databases, allowing the application to perform complex operations such as user registration, review management, and interactions with communities. It is an essential component for accessing and manipulating data within the application.

## 12.5 Controller

The controller is responsible for managing communication between the front-end and the back-end. It receives requests from the front-end, routes operations within the application based on the received data, and sends appropriate responses to the front-end. It uses the information in HTTP requests to determine the action to be taken and interacts with service modules to implement business logic. Once processing is complete, it generates responses for the front-end, ensuring effective communication between the two parts of the system.

Our controller is divided into multiple files, one for each part of the front-end interface. This allows the functions elaborated in the data service to be called in each controller endpoint, which then processes the data for the front-end page using the httpRequest classes.

## 12.6 Exception

This directory is responsible for exception handling, with one class for each entity present in our application. All classes extend the RuntimeException class, enabling the raising of exceptions in case of errors.

## 12.7 ConsistencyCheck

Directory containing the WatchlistConsistencyChecker class, which we will delve into further later, serves to periodically check the consistency between the watchlist and reviews.

## 12.8 Main Classes Overview

This section provides a concise overview of the primary classes implemented within the application. These classes are categorized by the package to which they belong. While interfaces are not explicitly listed, the descriptions of the respective implementations will indicate the interfaces they implement.

| **Model** | | |
|---|---|---|
| **Class** | **Sub-Package** | **Brief description** |
| Admin | MongoDB | Extends the parent class User providing Admin unique attributes and their getters and setter |
| Movie | MongoDB | Contains all the attributes for describe movie and all the getter and setter |
| RegisteredUser | MongoDB | Extends the parent class User providing Registered User unique |

| | | |
|---|---|---|
| | | attributes and their getters and setter |
| Review | MongoDB | Contains all review attributes and their getter and setter |
| User | MongoDB | Contains all the attributes common to all types of users in the system and their getters and setters |
| WatchlistItem | MongoDB | Contains all Watchlist item attributes and their getter and setter |
| AgeGroupMovieCountResult | Aggregate | Contains all the attributes for get data from the aggregate countMoviesByAgeGroup |
| MostWatchedMovieResult | Aggregate | Contains all the attributes for get data from the aggregate countUsersByCityAndMovie |
| MovieDetailsResult | Aggregate | Contains all the attributes for get data from the aggregate findMovieDetails |
| MovieRandomReviewsResult | Aggregate | Contains all the attributes for get data from the aggregate findRandomReviewsForMovie |
| MovieReviewCountResult | Aggregate | Contains all the attributes for get data from the aggregate findReviewCountByMovie |
| RatingCountResult | Aggregate | Contains all the attributes for get data from the aggregate findRatingHistogramForUser |
| RatingDistributionResult | Aggregate | Contains all the attributes for get data from the aggregate findRatingDistributionByMovie |
| ReviewAggregateResult | Aggregate | Contains all the attributes for get data from the aggregate findTopMoviesByReviews |
| UserWatchTimeResult | Aggregate | Contains all the attributes for get data from the aggregate calculateUserWatchTime |
| Comment | Neo4j | Contains the attributes common to all types of comment in the system and their getters and setters |
| Community | Neo4j | Contains the attributes common to all types of community in the system and |

| | | |
|---|---|---|
| | | their getters and setters |
| Post | Neo4j | Contains the attributes common to all types of post in the system and their getters and setters |
| User | Neo4j | Contains the attributes common to all types of users in the system and their getters and setters |
| RecommendedCommunity | QueryReturn | Contains all the attributes for get data from the neo4j query getMostActiveUserAndCommunity |
| UserActivity | QueryReturn | Contains all the attributes for get data from the neo4j query findMostActiveUsersInCommunity |
| CommentRequest | httpRequest | Required to facilitate communication with the front-end |
| InfoRequest | httpRequest | Required to facilitate communication with the front-end |
| LoginRequest | httpRequest | Required to facilitate communication with the front-end |
| PostRequest | httpRequest | Required to facilitate communication with the front-end |
| ReviewRequest | httpRequest | Required to facilitate communication with the front-end |
| Topfilmrequest | httpRequest | Required to facilitate communication with the front-end |
| WatchlistAddRequest | httpRequest | Required to facilitate communication with the front-end |

| **Repository** | |
|---|---|
| **Class** | **Sub-Package** |
| AdminRepository | MongoDB |
| MovieRepository | MongoDB |
| UserRepository | MongoDB |
| RegisteredUserRepository | MongoDB |
| ReviewRepository | MongoDB |
| Neo4jCommentRepository | Neo4j |
| Neo4jCommunityRepository | Neo4j |
| Neo4jPostRepository | Neo4j |
| Neo4jUserRepository | Neo4j |

| **Service** |
|---|

| Class | Sub-Package |
|---|---|
| Dataservice | - |
| MongodbWatchlistItemservice | MongoDB |
| MongodbMovieService | MongoDB |
| MongodbUserService | MongoDB |
| MongodbReviewService | MongoDB |
| Neo4jCommentService | Neo4j |
| Neo4jCommunityService | Neo4j |
| Neo4jPostService | Neo4j |
| Neo4jUserService | Neo4j |

| Controller | |
|---|---|
| **Class** | **Sub-Package** |
| CommunityController | - |
| MovieController | - |
| ReviewController | - |
| UserController | - |
| AdminController | - |

| Exception | |
|---|---|
| **Class** | **Sub-Package** |
| CommminityException | - |
| MovieException | - |
| MovieNotFoundException | - |
| PostException | - |
| ReviewException | - |
| UserException | - |
| UserNotFoundException | - |

| ConsistencyCheck | |
|---|---|
| **Class** | **Sub-Package** |
| WatchlistConsistencyChecker | - |

### 12.9 Queries

Now I will show the queries used, presenting the code and explaining them in detail step by step, to analyze how the result is obtained. Using the @Query and @Aggregation methods provided by Spring, the structure of our database queries closely resembles those used in the shells of the two databases.

#### 12.9.1 Document DB

The majority of queries in our application are performed in MongoDB. However, most of them are simple CRUD operations that do not require a detailed explanation. We will focus on the more complex or core functionalities queries.

## countMoviesByAgeGroup

```
@Aggregation(pipeline = {
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 10] }, { $lt: ['$age', 20] } ] }, then: '10-19', else: null } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 20] }, { $lt: ['$age', 30] } ] }, then: '20-29', else: '$ageGroup' } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 30] }, { $lt: ['$age', 40] } ] }, then: '30-39', else: '$ageGroup' } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 40] }, { $lt: ['$age', 50] } ] }, then: '40-49', else: '$ageGroup' } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 50] }, { $lt: ['$age', 60] } ] }, then: '50-59', else: '$ageGroup' } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 60] }, { $lt: ['$age', 70] } ] }, then: '60-69', else: '$ageGroup' } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $and: [ { $gte: ['$age', 70] }, { $lt: ['$age', 80] } ] }, then: '70-79', else: '$ageGroup' } } } }",
        "{ $addFields: { ageGroup: { $cond: { if: { $gte: ['$age', 80] }, then: '80+', else: '$ageGroup' } } } }",
        "{ $group: { _id: '$ageGroup', movieCount: { $sum: { $size: { $filter: { input: '$watchlist', cond: { $eq: ['$$this.watch', 1] } } } } } } }",
        "{ $project: { _id: 0, ageGroup: '$_id', movieCount: 1 } }"
})
List<AgeGroupMovieCountResult> countMoviesByAgeGroup();
```

This query begins with the "@Aggregation" annotation, indicating its use as a MongoDB aggregation operation. It sets up a data processing pipeline composed of multiple stages, each responsible for a specific data operation. Data flows through these stages sequentially. In the initial stage, we employ "$addFields" to introduce a new field named "ageGroup" to the documents. This field will be used to categorize users based on their age. The "$cond" function serves as a condition, checking if a user's age falls between 10 and 19 years. If the condition holds true, it assigns "10-19" to the "ageGroup" field; otherwise, it assigns "null." Subsequent stages repeat this process for various age groups (e.g., 20-29, 30-39, etc.), updating the "ageGroup" field when a user's age falls within a specific range. Once each user has been assigned an age group, the query proceeds to the "$group" stage. Here, documents are grouped based on the "ageGroup" field, and the count of movies is calculated for each age group. This calculation is performed using "$sum" and "$size" to count movies within each user's "watchlist" that have a "watch" value of 1. Finally, in the "$project" stage, we define the structure of the final result. The previous ID is excluded, and the result is returned as a list of objects with "ageGroup" and "movieCount" fields. In essence, this query divides users into different age groups, calculates the movie count for each age group, and returns the result as a list of objects representing the number of movies for each age bracket.

This query is considered fundamental for the statistical analysis by the admin because the relationship between age groups and movies watched reveals the target marketing of our application.

## countUsersByCityAndMovie

```
@Aggregation(pipeline = {
        "{ $match: { 'city': ?0, 'watchlist.movie.movie_id': ?1, 'watchlist.watch': 1 } }",
        "{ $count: 'userCount' }"
})
Optional<MostWatchedMovieResult> countUsersByCityAndMovie(String city, String movieId);
```

This query is designed to count the number of users in a specific city who have added a specific movie to their watchlist and have watched this movie. The query uses the "watch" field with a value of 1 to indicate that the movie has been watched.

The query consists of two distinct aggregation stages executed sequentially. In the first stage, using { $match: { 'city': ?0, 'watchlist.movie.movie_id': ?1, 'watchlist.watch': 1 } }, documents that meet three conditions are filtered: the city must match the "city" parameter passed to the method, there must be a movie with a "movie_id" matching the "movieId" parameter in the watchlist, and the "watch" field must be equal to 1. This filter reduces the set of documents on which subsequent aggregation operations will be performed. In the second stage, with { $count: 'userCount' }, a counting operation is performed on the filtered results from the first stage. This operation counts how many documents satisfy the specific conditions specified in the preceding phase and returns the result with a field named 'userCount.' The method countUsersByCityAndMovie accepts two parameters: "city" and "movieId." The query determines how many users in the specified city have added the specified movie to their watchlist and have watched it, returning the count as a value encapsulated in the "MostWatchedMovieResult" object. In this context, "watch" with a value of 1 indicates that users have watched the movie.

In summary, this query provides the count of users in a specific city who have added a specific movie to their watchlist and have watched it.

## calculateUserWatchTime

```
@Aggregation(pipeline = {
        "{ $match: { 'username': ?0 } }",
        "{ $project: { " +
                "    totalRuntime: { " +
                "        $reduce: { " +
                "            input: { " +
                "                $filter: { " +
                "                    input: '$watchlist', " +
                "                    as: 'item', " +
                "                    cond: { $eq: ['$$item.watch', 1] } " +
                "                } " +
                "            }, " +
                "            initialValue: 0, " +
                "            in: { $add: ['$$value', '$$this.movie.runtime'] } " +
                "        } " +
                "    } " +
                "}} "
})
Optional<UserWatchTimeResult> calculateUserWatchTime(String username);
```

- { $match: { 'username': ?0 } }: This part of the aggregation pipeline filters the data to consider only documents with the specified 'username' field (provided as a parameter).
- { $project: { "totalRuntime": { $reduce: { input: { $filter: { input: '$watchlist', as: 'item', cond: { $eq: ['$$item.watch', 1] } } }, initialValue: 0, in: { $add: ['$$value',

'$$this.movie.runtime'] } } } } }: This section projects a new field called 'totalRuntime' by applying a series of operations to the 'watchlist' array.

- $filter: It filters the 'watchlist' array to include only items where the 'watch' field is equal to 1.
- $reduce: This operator iterates over the filtered array and calculates the total viewing time by summing up the 'runtime' of each movie. The 'initialValue' starts at 0, and the 'in' expression adds the current movie's runtime to the accumulated value.

The result of this query is an aggregation that returns 'totalRuntime' for the specified user, which represents the total time spent watching movies by that user, considering only movies with the 'watch' field set to 1 in their watchlist. The result is encapsulated in an Optional<UserWatchTimeResult> object, allowing you to retrieve this information in a structured way in your code.

## findWatchlisttoWatch

```
@Aggregation(pipeline = {

        "{ $match: { 'username': ?0 } }",

        "{ $project: { 'username': 1, 'watchlist': { $filter: { input: '$watchlist', as: 'item', cond: { $eq: ['$$item.watch', 0] } } } } }",

        "{ $project: { 'username': 1, 'watchlist': { $slice: ['$watchlist', 10] } } }"

})
RegisteredUser findWatchlisttowatch(String username);
```

This query is designed to retrieve a specific portion of a user's watchlist, excluding the movies they have already watched. It is useful to improve the performance of the user's main page, by loading only the movies that the user has yet to watch.

Here's how the query works: in the first stage, { $match: { 'username': ?0 } }, document filtering takes place. The query searches for documents where the "username" field matches the parameter username passed to the method. This is used to identify the specific user whose watchlist you want to retrieve. In the second stage, { $project: { 'username': 1, 'watchlist': { $filter: { input: '$watchlist', as: 'item', cond: { $eq: ['$$item.watch', 0] } } } } }, a data projection process occurs. In this stage, the user's watchlist is filtered using the $filter operator. Only items in the watchlist where the "watch" field has a value of 0, indicating that the user has not yet watched those movies, are included. The result is a watchlist containing only the movies that the user has yet to watch. In the third stage, { $project: { 'username': 1, 'watchlist': { $slice: ['$watchlist', 10] } } }, another projection is performed. In this case, the query focuses on limiting the number of movies to display in the user's watchlist. The $slice operator is used to select the first 10 movies from the filtered watchlist. This is useful to prevent the user's main page from being overloaded with a lengthy list of movies.

In summary, this query retrieves a portion of a user's watchlist, excluding the movies they have already watched. It is particularly useful on the user's main page to improve performance and display only the movies that the user has yet to watch.

## findWatchlistWatched

```
@Aggregation(pipeline = {
        "{ $match: { 'username': ?0 } }",
        "{ $project: { 'username': 1, 'watchlist': { $filter: { input: '$watchlist', as: 'item', cond: { $eq: ['$$item.watch', 1] } } } } }",
        "{ $project: { 'username': 1, 'watchlist': { $slice: ['$watchlist', 10] } } }"
})
RegisteredUser findWatchlistwatched(String username);
```

In the first stage, { $match: { 'username': ?0 } }, document filtering occurs. The query searches for documents where the "username" field matches the username parameter passed to the method. This identifies the specific user whose watchlist you want to retrieve. In the second stage, { $project: { 'username': 1, 'watchlist': { $filter: { input: '$watchlist', as: 'item', cond: { $eq: ['$$item.watch', 1] } } } } }, a data projection process takes place. In this case, the user's watchlist is filtered using the $filter operator. Only items in the watchlist where the "watch" field has a value of 1, indicating that the user has already watched those movies, are included. The result is a watchlist containing only the movies that the user has already seen. In the third stage, { $project: { 'username': 1, 'watchlist': { $slice: ['$watchlist', 10] } } }, an additional projection is performed. In this case, the query focuses on limiting the number of movies to display in the user's watchlist. The $slice operator is used to select the first 10 movies from the filtered watchlist. This is useful to prevent overloading the user's main page with a lengthy list of movies.

In summary, this query retrieves a portion of a user's watchlist, but includes only the movies that the user has already watched. It is particularly useful on the user's main page to improve performance and present only the movies that the user has already seen, avoiding overwhelming the page with a complete list of movies.

## findRatingHistogramForUser

```
@Aggregation(pipeline = {
        "{ $match: { 'authorusername': ?0 } }",
        "{ $group: { _id: '$rating_val', count: { $sum: 1 } } }",
        "{ $sort: { _id: 1 } }"
})
List<RatingCountResult> findRatingHistogramForUser(String username);
```

In the first stage, { $match: { 'authorusername': ?0 } }, document filtering takes place. The query searches for reviews where the "authorusername" field matches the username parameter passed to the method. This identifies the reviews written by the specific user for whom you want to calculate the rating histogram. In the second stage, { $group: { _id: '$rating_val', count: { $sum: 1 } } }, a grouping operation occurs. Documents are grouped based on the "rating_val" field (rating value) of the reviews. The $sum operator is used to count the number of reviews in each group. In other words, it counts how many times the user has assigned a particular rating value. The

result includes the ID of the rating value and the count of reviews for each value. In the third stage, { $sort: { _id: 1 } }, the results are sorted based on the ID of the rating value in ascending order. This is useful to ensure that the results are presented in an orderly manner in a scale from 1 to 10.

In summary, this query calculates the histogram of ratings given by the user for reviews. This histogram will show how many times the user has assigned each possible rating value and can be used to create a bar chart in the user's profile, allowing other users to see the user's rating preferences.

The histogram is particularly interesting because by summing up all ten values, we obtain the total number of reviews. This data helps us determine how many movies a user has actually watched. Thanks to this information, we can simplify a query to retrieve all the movies watched by a user, obtaining the desired value directly by summing those ten values.

### findRandomReviewforMovie

```
@Aggregation(pipeline = {
        "{ $match: { 'movie.movie_id': ?0 } }",
        "{ $sample: { size: 5 } }",
        "{ $group: { _id: '$movie.movie_title', randomReviews: { $push: '$$ROOT' } } }",
        "{ $project: { _id: 1, randomReviews: 1 } }"
})
List<MovieRandomReviewsResult> findRandomReviewsForMovie(String movie_id);
```

In the first stage, { $match: { 'movie.movie_id': ?0 } }, document filtering is performed. The query searches for reviews where the "movie.movie_id" field matches the movie_id parameter passed to the method. This identifies all the reviews related to the specific movie. In the second stage, { $sample: { size: 5 } }, a random sampling of documents takes place. The $sample operator is used to randomly select a subset of 5 reviews from all those related to the movie. This way, you obtain a random selection of reviews for the movie. In the third stage, { $group: { _id: '$movie.movie_title', randomReviews: { $push: '$$ROOT' } } }, the results are grouped by the movie's title. The $push operator is used to create an array called "randomReviews" that contains the complete documents of the previously selected reviews. This step organizes the random reviews for the specific movie. In the fourth stage, { $project: { _id: 1, randomReviews: 1 } }, a data projection is performed. Only the fields "_id" (representing the movie's title) and "randomReviews" (containing the random reviews) are selected. These fields are included in the result.

In summary, this query retrieves 5 random reviews for a specific movie and organizes them into a result that includes the movie's title and the list of reviews. These random reviews can be used to display a random selection of comments to users of the website or application, enriching the movie viewing experience.

## findRatingDistributionByMovie

```
@Aggregation(pipeline = {
        "{ $match: { 'movie.movie_title': ?0 } }",
        "{ $group: { _id: '$rating_val', count: { $sum: 1 } } }",
        "{ $sort: { _id: 1 } }"
})
List<RatingDistributionResult> findRatingDistributionByMovie(String movie_title);
```

In the first stage, { $match: { 'movie.movie_title': ?0 } }, document filtering is performed. The query looks for reviews where the "movie.movie_title" field matches the movie_title parameter passed to the method. This identifies all reviews related to a specific film. In the second stage, { $group: { _id: '$rating_val', count: { $sum: 1 } } }, a data grouping operation takes place. Documents are grouped based on the "rating_val" (rating value) field of the reviews. The $sum operator is used to count the number of reviews in each group. In other words, it counts how many times different rating values have been assigned for that film. The result includes the ID of the rating value and the count of reviews for each value. In the third stage, { $sort: { _id: 1 } }, the results are sorted based on the ID of the rating value in ascending order. This is useful to ensure that the results are presented in an ordered manner, in a scale from 1 to 10.

In summary, this query calculates the distribution of ratings given to reviews of a specific film. It shows how many times different rating values have been assigned for that film and orders them to present the distribution clearly and in an organized manner. This information about the rating distribution can be used to better understand how a particular film has been rated by users.

## findReviewCountByMovie

```
@Aggregation(pipeline = {
        "{ $group: { _id: '$movie.movie_title', totalReviews: { $sum: 1 } } }"
})
List<MovieReviewCountResult> findReviewCountByMovie();
```

In the main stage of the query, { $group: { _id: '$movie.movie_title', totalReviews: { $sum: 1 } } }, a data grouping operation is performed. Documents are grouped based on the "movie.movie_title" field, which represents the title of the film. The $sum operator is used to calculate the total number of reviews within each group. This means that the query sums 1 for each review in the group, thus counting the total reviews for each specific film. The result of this query will be a list of MovieReviewCountResult objects, each of which contains the film's title (_id) and the total number of reviews (totalReviews) for that specific film.

We use this query to assess the popularity of a film based on the number of reviews it has received. The more reviews a film has, the greater its popularity or the interest it has generated among users. This information can be useful for providing guidance to potential viewers on which films are more discussed or appreciated within our platform, helping them make informed decisions about which films to watch.

## findTomMovieByReviews

```
@Aggregation(pipeline = {
        "{ $group: { _id: '$movie.movie_id', totalReviews: { $sum: 1 }, avgRating: { $avg: '$rating_val' } } }",
        "{ $sort: { totalReviews: -1, avgRating: -1 } }",
        "{ $limit: 5 }"
})
List<ReviewAggregateResult> findTopMoviesByReviews();
```

In the first stage of the query, { $group: { _id: '$movie.movie_id', totalReviews: { $sum: 1 }, avgRating: { $avg: '$rating_val' } } }, a data grouping phase takes place. The review documents are grouped based on the "movie.movie_id" field, which represents the unique identifier of the movie. In this stage, two values are calculated for each movie:

- totalReviews: It's the total number of reviews received by the movie, obtained by summing 1 for each review in the group.
- avgRating: It's the average rating of the movie, calculated as the average of the "rating_val" values in the reviews related to the movie.

In the second stage, { $sort: { totalReviews: -1, avgRating: -1 } }, the results are sorted based on the total number of reviews in descending order (totalReviews) and the average rating in descending order (avgRating). This ensures that movies with the highest number of reviews and the highest average rating will be positioned at the top of the list. In the third stage, { $limit: 5 }, a limit of 5 movies is applied. This means that only the top 5 movies from the sorted list will be returned, representing the best movies based on user reviews.

In summary, this query identifies the most popular movies on the platform by considering the number of reviews received and the average rating. The results are used to create the "Top Movie" page, where the best movies are presented based on user opinions. The query is executed once a day to ensure that the information is up-to-date and reflects changes in user reviews over time.

### 12.9.2 GraphDB

### findMostActiveUsersInCommunity

```
@Query("MATCH (u:User)-[activity:POSTING|COMMENT]->(p:Post)-[:BELONG]->(c:Community {name: $communityname}) " +
        "RETURN u.username AS username, " +
        "COUNT(CASE WHEN TYPE(activity) = 'POSTING' THEN p END) AS numPostings, " +
        "COUNT(CASE WHEN TYPE(activity) = 'COMMENT' THEN p END) AS numComments, " +
        "COUNT(p) AS totalActivity " +
        "ORDER BY totalActivity DESC " +
        "LIMIT $limit")
List<UserActivity> findMostActiveUsersInCommunity(@Param("communityname") String communityName,@Param("limit") int  limit);
```

The query uses the Cypher language, which is specific to Neo4j, a graph database. Here's how it works:

- { $match ... }: In this part of the query, user nodes (u) connected to post nodes (p) or comments (c) through "POSTING" or "COMMENT" activity relationships within a specified community are selected. The filter is based on the community name passed as the $communityname parameter.
- RETURN ...: The query returns the following information for each user: u.username AS username: The user's username.
- COUNT(CASE WHEN TYPE(activity) = 'POSTING' THEN p END) AS numPostings: The number of posts made by the user within the community.
- COUNT(CASE WHEN TYPE(activity) = 'COMMENT' THEN p END) AS numComments: The number of comments written by the user within the community.
- COUNT(p) AS totalActivity: The total activities (sum of posts and comments) performed by the user in the community.
- ORDER BY totalActivity DESC: The results are sorted based on the total activities in descending order. This means that users with the highest amount of activity will be at the top of the list.
- LIMIT $limit: A limit is applied to the number of returned results, which is specified as the $limit parameter. This allows controlling the number of most active users returned by the query.

In conclusion, this query allows community administrators to identify the most active users within that community, considering both posts and comments. It is useful for assessing user engagement and rewarding those who contribute significantly to the community's life.

## findShortestPath

```
@Query("MATCH p = shortestPath((u1:User{username:$username1})-[:POSTING|BELONG|COMMENT*..50]-(u2:User{username:$username2})) " +
        "RETURN length(p) as pathdistance")
Integer findShortestPath(@Param("username1") String username1, @Param("username2") String  username2);
```

Here's how it works:

- MATCH p = shortestPath(...): In this part of the query, the shortest path (shortestPath) between two user nodes (u1 and u2) within the graph is searched for. These two user nodes are identified by the specific usernames provided as parameters ($username1 and $username2). The path can traverse relationships of different types, such as "POSTING," "BELONG," or "COMMENT," and has a maximum length of 50.
- RETURN length(p) as pathdistance: The query returns the length of the path (pathdistance) between the two users as the result. This value represents the number of relationships or connections required to link the two users within the social network of the platform.

Furthermore, it's important to note that this query is useful for determining how closely connected or similar two users are within the social network of the platform. A shorter distance indicates a stronger connection or closer relationship between the two users, while a longer distance suggests they may be less connected or have fewer interactions. In essence, the query helps evaluate the degree of connection between two users, providing a measure of their proximity within the social network of the platform.

## getMostActiveUserAndCommunity

```
@Query("MATCH (selectedUser:User {username: $selectedUsername})-[:POSTING|COMMENT]->(activity:Post) " +
        "WITH selectedUser, activity " +
        "MATCH (otherUser:User)-[:POSTING|COMMENT]->(activity) " +
        "WHERE otherUser <> selectedUser " +
        "WITH otherUser, COUNT(activity) AS activityCount " +
        "ORDER BY activityCount DESC " +
        "LIMIT 1 " +
        "MATCH (userA:User {username: otherUser.username})-[:JOIN]->(communityA:Community) " +
        "WHERE NOT EXISTS { " +
        "  MATCH (userA)-[:JOIN]->(communityA)<-[:JOIN]-(userB:User {username: $selectedUsername}) " +
        "} " +
        "RETURN otherUser.username AS mostActiveUser, activityCount, COLLECT(communityA.name)[0] AS comname")
RecommendedCommunity getMostActiveUserAndCommunity(@Param("selectedUsername") String selectedUsername);
```

This query is designed to recommend a community to a user who is not currently a member of it. Here's how it works:

- MATCH (selectedUser:User {username: $selectedUsername})-[:POSTING|COMMENT]->(activity:Post): In this part of the query, the user of interest is selected, identified by the username specified as a parameter

($selectedUsername). It looks for "POSTING" and "COMMENT" relationships between this user and the posts (activity) they have interacted with.
- WITH selectedUser, activity: The results of this search are retained, including both the selected user and the activity.
- MATCH (otherUser:User)-[:POSTING|COMMENT]->(activity) WHERE otherUser <> selectedUser: It looks for another user (otherUser) who has interacted with the same activity (post) but is not the selected user. In other words, it searches for a different user who has participated in the same activities.
- WITH otherUser, COUNT(activity) AS activityCount: It counts the activities (posts or comments) in which the other user (otherUser) has participated. The count is stored as "activityCount."
- ORDER BY activityCount DESC: The results are sorted by the count of activities in descending order, putting the user who has participated in the most activities with the selected user at the top.
- LIMIT 1: It limits the result to only one user, specifically the one who has participated in the most activities with the selected user.
- MATCH (userA:User {username: otherUser.username})-[:JOIN]->(communityA:Community): It searches for the newly identified most active user (otherUser) and checks if they are a member of a community (communityA) to which they have subscribed.
- WHERE NOT EXISTS { MATCH (userA)-[:JOIN]->(communityA)<-[:JOIN]-(userB:User {username: $selectedUsername}) }: It verifies that the selected user ($selectedUsername) is not already a member of the same community to which the most active user identified earlier belongs. In other words, it attempts to avoid recommending a community to the user that they are already a member of through another user.
- RETURN otherUser.username AS mostActiveUser, activityCount, COLLECT(communityA.name)[0] AS comname: Finally, it returns the username of the most active user (mostActiveUser), the count of shared activities (activityCount), and the name of the community to which this user belongs and is recommended to the selected user (comname).

In summary, this query suggests a community to a user who is not a member of it, based on the most active user they have interacted with and who belongs to the recommended community. This way, personalized community recommendations are provided based on users' interactions within the platform.

# 12 Unit Test

## Introduction

Unit tests are small pieces of code that verify individual parts of our software, helping to catch errors early and ensuring greater stability. We have decided to provide a

test file for each service, both in MongoDB and Neo4j. Unit tests play a crucial role in both establishing a general structure and division between the two main modules and the functions that make them up, as well as aiding in the debugging process during the project's development. We decide to implement in memory operation to test the database and then cleanup operation to restore the precedent memory state. Below, you'll find a description of each module along with an explanation of the primary tests for each of them.

## 12.1 MongoDB Unit Tests

### - User Tests

#### Setup

In the setup method, we prepare the environment for testing. Specifically, we test if the Registered user with the username testUser already exists, it is deleted to ensure a clean slate for testing.

#### Adding a User (addUser)

This test case assesses the successful addition of a user to the system. We create a new RegisteredUser and add it using the addUser method. The test checks whether the user is added correctly and whether the returned result contains the expected user information.

#### Adding a User with Duplicate Username (addUser_DuplicateUsername)

In this test, we evaluate the behaviour when attempting to add a user with a duplicate username. We first add a user with a specific username and then attempt to add another user with the same username. The test expects that the second user addition will not succeed, and it asserts that the returned result is not present.

#### Adding a User with Duplicate Username (addUser_DuplicateUsername)

In this test, we evaluate the behaviour when attempting to add a user with a duplicate username. We first add a user with a specific username and then attempt to add another user with the same username. The test expects that the second user addition will not succeed, and it asserts that the returned result is not present.

**Deleting a Non-Existent User (deleteUser_NonExistentUser)**

Here, we test the scenario where an attempt is made to delete a user with a non-existent ID. The test expects an appropriate message indicating that the user does not exist.

**Finding a User by Username (findByUserName)**

This test verifies the ability to find a user by their username. We add a test user and then attempt to retrieve it using the findByUserName method. The test ensures that the user is found and that the retrieved user's username matches the expected value.

**Finding a Non-Existent User by Username (findByUserName_NonExistentUser)**

In this test, we evaluate the behavior when trying to find a user with a non-existent username. The test anticipates a UserNotFoundException to be thrown with a message indicating that the user does not exist.

**Counting Users by City and Movie (countUsersByCityAndMovie)**

This test case checks the functionality of counting users based on their city and a specific movie they interacted with. It counts the users from the city "Roma" who interacted with the movie "Football Freaks." The test expects a specific count of users.

**Counting Movies by Age Group (countMoviesByAgeGroup)**

This test provides if the output of this analytics function, that aim to show the userbase age distribution, is correct and the output (provided with a specific class made the receive the query) is readable.

**Finding Rating Histogram for a User (FindHistogramTest)**

This test provides an example of printing a rating histogram for a user (sonnyc). It demonstrates how the findRatingHistogramForUser method works by displaying the user's rating distribution as an histogram (front-end drawn) of valuations.

**Cleanup**

In the @AfterAll method, we perform cleanup tasks. Specifically, we delete the test user created during the setup phase to leave the system in a clean state after testing.

- ## Movie test

### Introduction

The MovieTest class consists of a series of unit tests designed to evaluate various functionalities related to managing movies within our Java-based project. These tests cover scenarios such as adding movies, deleting movies, and searching for movies by title. From this the on all the tests use an Order method to order the execution of the tests so the output will be more linear and easier to comprehend.

### Test Setup

Before discussing the individual test cases, let's look at the setup phase. In this class, we utilize the @Autowired annotation to inject the MovieRepository and MongodbMovieService, which are essential components for movie-related operations. We also use the @SpringBootTest and @ActiveProfiles("test") annotations to create a testing environment. Now, let's explore the specific test cases within this class:

### Adding a Movie Successfully (testAddMovieSuccess)

This test evaluates the successful addition of a movie to the system. We create a Movie object with specific attributes and then add it using the addMovie method from the MongodbMovieService. The test ensures that the movie is added correctly and that the returned result matches the expected movie.

### Adding a Movie with Duplicate Key (testAddMovieDuplicateKeyException)

In this test case, we attempt to add a movie that has the same key (movieId) as one that already exists in the system. The test verifies that attempting to add such a duplicate movie result in an exception.

### Deleting a Movie Successfully (testDeleteMovieSuccess)

Here, we test the functionality of deleting a movie from the system. After creating a movie, we delete it using the DeleteMovie method from the MongodbMovieService. The test asserts that the movie is successfully removed and no longer exists in the database with the existById repository function.

### Deleting a Non-Existent Movie (testDeleteMovieNotFound)

In this test, we examine the behaviour when trying to delete a movie that does not exist in the system. The test expects a MovieNotFoundException to be thrown with an appropriate message indicating that the movie does not exist.

### Finding Movies by Title (testFindByTitle)

This test case verifies the ability to search for movies by their title. We select a movie title (Aftermath) that has multiple versions in the database, each with different attributes such as release year and runtime. The test retrieves all versions of the movie and prints their details to the console. It ensures that the expected number of movies is retrieved.

## - Review Test

### Introduction

The ReviewTest class consists of a series of unit tests designed to evaluate various functionalities related to managing movie reviews within our project. These tests cover scenarios such as adding reviews, deleting reviews, searching for reviews by username, and performing aggregate functions on movie reviews.

### Setup for Test Data

In the @BeforeAll method, a test user named "Test User" with the username "testUser" is added to the system. This user is used for various review-related tests.

### Adding a Review (TestaddReview)

This test evaluates the successful addition of a review to the system. We prepare the necessary data, including a movie and review details, and then add the review using the addReview method from the MongodbReviewService. The test ensures that the review is added correctly and that the returned result matches the expected review.

### Deleting a Review (TestdeleteReview)

Here, we test the functionality of deleting a review from the system. After creating a review, we delete it using the deleteReview method from the MongodbReviewService. The test asserts that the review is successfully removed.

### Finding Reviews by Username (TestFindUserReviews)

This test case verifies the ability to search for reviews authored by a specific user. We execute the FindReviewsAuthor method, which retrieves a list of reviews authored by the user "testUser." The test ensures that the list is not empty, indicating that reviews by the user exist.

### Finding Top Movies by Reviews (TestFindTopMoviesByReviews)

This test retrieves the top-rated movies based on reviews. It uses the findTopMoviesByReviews method to obtain a list of ReviewAggregateResult objects, each representing a movie and its review statistics. The test prints information such as movie titles, total reviews, and average ratings for the top-rated movies.

### Finding Movie Details (TestfindMovieDetails)

This test retrieves detailed information about a specific movie, "The Hangover Part II." It calls the findMovieDetails method, which returns a list of MovieDetailsResult objects containing user data, average ratings, and more for the movie. The test ensures that the expected movie details are retrieved.

### Finding Random Reviews for a Movie (TestFindRandomReviewsForMovie)

Here, we test the ability to retrieve random reviews for a specific movie, "The Hangover Part II." The findRandomReviewsForMovie method is used to obtain a list of MovieRandomReviewsResult objects, each containing random reviews for the movie. The test prints review details, including ratings, authors, and review text.

## - Watchlist test

### Introduction

The MongodbWatchlistserviceTest class contains a series of unit tests that assess various functionalities related to managing user watchlists and movie tracking within our Java-based project. These tests evaluate scenarios such as adding movies to a watchlist, removing movies, marking movies as watched, and retrieving watchlist data.

### Test Setup

In the @BeforeAll method, a test user with the username "testUser" is added to the system to facilitate various watchlist-related tests.

**Adding a Movie to Watchlist (testAddWatchlistItemSuccess)**

This test evaluates the successful addition of a movie to a user's watchlist. It creates a WatchlistItem for the movie "Football Freaks" and adds it to the watchlist of the test user "testUser." The test ensures that the movie is added correctly to the watchlist.

**testRemoveWatchlistItem_UserNotFound**

the test attempts to remove a movie from a nonexistent user's watchlist. It expects a UserNotFoundException to be thrown, indicating that the user does not exist.

**testRemoveWatchlistItem_ItemNotFound**

the test attempts to remove a nonexistent movie from an existing user's watchlist. It expects a UserException to be thrown, signifying an error during the removal process.

**Watching a Movie (testWatchMovie)**

This test verifies the functionality of marking a movie as watched by a user. It marks the movie "Football Freaks" as watched by the test user "testUser" and checks whether the watch status is correctly updated.

**Retrieving First 5 Watched Movies (testgetFirst10WatchedMovies)**

In this test, we retrieve the first 5 watched movies for the user "madmann14" and print the watchlist items. This test helps ensure that the correct list of watched movies is obtained. We use an already in database user to show how the system retrieve only 5 reviews and not all of them after the query.

**Retrieving First 5 Unwatched Movies (testgetFirst10UnwatchedMovies)**

Like the previous test, this one retrieves the first 5 unwatched movies for the user "madmann14" and prints the watchlist items. It verifies that unwatched movies are correctly identified and listed.

**Calculating User Watch Time (testcalculateUserWatchTime)**

This test calculates and prints the total watch time for the user "vinsim27" based on their watchlist. It utilizes the calculateUserWatchTime method and checks whether

the watch time calculation is accurate. In this test too we use an already in database user to facilitate

#### Test Cleanup

In the end, with the @BeforeAll method we remove the user previously created.

## 12.2 Neo4j Unit Tests

### - User Test

#### Introduction

The Neo4jUserTest class contains a series of unit tests that evaluate various functionalities related to user management and community interactions in a Neo4j database within our project. These tests cover scenarios such as creating users, managing users in communities, posting content, and recommending communities.

#### Test Setup

The tests are ordered using @Order annotations to ensure that they run in a specific sequence.

#### Creating a User (createUser_Success)

This test evaluates the successful creation of a user in the Neo4j database. It creates a new user with a unique username and display name, verifies the user's existence, and then deletes the user from the database as part of cleanup.

#### Creating a User with Duplicate Username (createUser_UsernameAlreadyExists)

In this test, we attempt to create a user with a username that already exists in the database. It verifies that the creation of a user with a duplicate username fails and ensures that no duplicate users are added to the database.

#### Deleting a User (deleteUser_Success)

This test assesses the functionality of deleting a user from the Neo4j database. It creates a user, deletes the user, and verifies that the user is successfully removed from the database.

**Deleting a Non-Existent User (deleteUser_UserDoesNotExist)**

This test attempts to delete a user that does not exist in the database. It verifies that attempting to delete a non-existent user returns a failure result.

**Getting a User by Username (getUserByUsername_Success)**

This test creates a user in the database and then retrieves the user by their username. It ensures that the user is correctly retrieved based on the provided username.

**Getting a Non-Existent User by Username (getUserByUsername_NotExistingUser)**

In this test, we attempt to retrieve a user by a non-existent username. It ensures that the method returns an empty optional when the requested user does not exist.

**Joining a community (testJoinCommunity_Success)**

This test creates a user and a community in the database. It then verifies the user's successful addition to the community using the joinCommunity method. Additionally, it checks whether the user's community list is correctly updated.

**Joining a Non-Existent Community (testJoinCommunity_NotExistingCommunity)**

In this test, we attempt to add a user to a community that does not exist in the database. It ensures that attempting to join a non-existent community result in an empty optional and that the user's state remains unchanged.

**Recommending a community (testRecommendaCommunity)**

This test retrieves recommended communities for a specific user, "vinsim27," based on their interactions within the Neo4j database. It checks whether the recommended community result is not null.

**Finding All Posts by User (testFindAllPostByUser)**

This test retrieves all posts created by a specific user, "worg," and verifies that the list of posts is not null. It also confirms that each post belongs to the specified user.

### Establishing a Posted Relationship (testPostedRelation)

In this test, we simulate the existence of a user, a post, and a community in the database. It establishes relationships between these entities, such as adding a post to a community and associating a user with the community. The test then verifies that the "postedrelation" method correctly connects the user to the post.

- **Community Test**

### Introduction

The Neo4jCommunityServiceTest class contains a series of unit tests that evaluate various functionalities related to community management and user. These tests cover scenarios such as creating communities, retrieving communities by name, deleting communities, and finding the most active users in a community.

### Test Setup

The tests are ordered using @Order annotations to ensure that they run in a specific sequence.

### Creating a Community (testCreateCommunity_Success)

This test evaluates the successful creation of a new community in the Neo4j database. It creates a new community with a unique name, verifies the community's existence, and confirms that the community is successfully created.

### Creating a Community with Duplicate Name (testCreateCommunity_DuplicateName)

In this test, we attempt to create a community with a name that already exists in the database (as created in the previous test). It verifies that the creation of a community with a duplicate name fails and ensures that no duplicate communities are added to the database.

### Finding a Community by Name (testFindCommunityByName_Success)

This test searches for a community by name, specifically the "NewCommunity" created in the first test. It ensures that the community is found based on its name, retrieves it successfully, and verifies the correctness of the retrieved community.

#### Finding a Non-Existent Community by Name (testFindCommunityByName_NotFound)

In this test, we attempt to find a community by a name that does not exist in the database. It ensures that the method returns an empty optional when the requested community does not exist.

#### Deleting a Community by ID (testDeleteCommunityById_Success)

This test deletes a community from the Neo4j database. It first retrieves the community created in the first test by its name, deletes it, and verifies the successful deletion of the community.

#### Getting the Most Active Users in a Community (testGetMostActiveUsersInCommunity)

This test retrieves the most active users in a specific community, "Discussion," and specifies a limit of 10 users. It calls the service method to retrieve this information and ensures that the result is not null. The result is printed to the console for examination.

- ## Comment Test

#### Introduction

The CommentTest class contains a series of unit tests that evaluate various functionalities related to comments within a Neo4j database. These tests cover scenarios such as creating comments, retrieving comments associated with a post, and verifying comment creation.

#### Creating a Comment (testCreateComment)

This test evaluates the successful creation of a new comment in the Neo4j database. It creates a new post, a new user, and then creates a comment associated with the post. The test verifies that the comment is successfully created, including checking its text and presence.

#### Retrieving Comments for a Post (testPostComments)

In this test, we initialize a user, create a post, and then create three comments associated with the post. Afterward, we retrieve comments that are associated with the post and verify their correctness. This test ensures that comments are correctly associated with the post and can be retrieved as expected.

- ## Post Test

### Introduction

The PostTest class contains a series of unit tests that evaluate various functionalities related to posts within our project. These tests cover scenarios such as creating posts, handling duplicate post IDs, deleting posts, and establishing relationships between posts and communities.

### Test Setup

The tests are ordered using @Order annotations to ensure that they run in a specific sequence.

### Creating a Post (testCreatePost)

This test evaluates the successful creation of a new post in the Neo4j database. It creates a new post with a unique ID, verifies that the post is created successfully, and then cleans up the database by deleting the created post.

### Creating a Post with Duplicate Post ID (createPost_DuplicatePostId_ExceptionThrown)

In this test, we attempt to create a post with a duplicate post ID. The test checks that an exception is thrown when trying to create a post with an existing ID. It ensures that the system correctly handles the scenario of duplicate post IDs.

### Deleting a Post (testDeletePost)

This test verifies the functionality of deleting a post from the Neo4j database. It creates a post, saves it, deletes it, and then checks that the post no longer exists in the database.

### Deleting a Non-Existing Post (deletePost_PostNotFound_PostExceptionThrown and deletePost_PostNotFound_ReturnsFalse)

These tests assess the behavior of the system when attempting to delete a post that does not exist in the database. The first test (deletePost_PostNotFound_PostExceptionThrown) checks that a specific exception (PostException) is thrown when trying to delete a non-existing post. The second test

(deletePost_PostNotFound_ReturnsFalse) verifies that the method returns false when attempting to delete a non-existing post.

**Establishing a Relationship Between a Post and a Community (testBelongRelationship)**

This test creates a community, a post, and establishes a relationship between the post and the community. It ensures that the relationship is correctly established, and the test takes care of cleaning up the database by deleting the created community and post.

# 13 Consistency, availability and partition tolerance

CAP Schema

We prioritize eventual consistency with a PA (Partition Tolerance and Availability) approach for the following reasons:

User Interaction: Our primary goal is to maximize user interaction within the system. Users should be able to perform actions and access information without experiencing delays caused by strict data consistency checks.

Up-to-Date Watchlists: Ensuring that users always have access to up-to-date watchlists is crucial. Users want their movie and content lists to reflect their latest actions, such as adding or removing items.

Lower Data Security Requirements: Our system does not handle highly sensitive data that necessitates strong consistency guarantees. While data accuracy is important, it doesn't need to be enforced rigorously at all times.

Watchlist Redundancies

We leverage redundancy in critical areas, such as user watchlists, to strike a balance between data consistency and availability. For example, when a user adds a new review, we follow this process:

Immediate Review Addition: Initially, we add the review immediately to the "review" field. This ensures that users see their actions reflected promptly.

Asynchronous Redundancy Updates: Simultaneously, we utilize Java CompletableFuture to create a background thread. This thread asynchronously updates the user's watchlist, adding the new movie, and computes statistics for histograms and total viewing time. By doing this asynchronously, we prevent overloading the main partition with multiple immediate requests, enhancing system availability. Consequently, users have immediate access to their watchlists.

Trade-off with Perfect Consistency: This approach results in a trade-off with perfect consistency between a user's watchlist and their reviews. While we maintain monotonic read consistency, we acknowledge that there may be eventual inconsistencies between the two datasets.

Tag Redundancies

A similar approach applies to tags, which draw data from Neo4j communities. When a user joins a new community, we immediately update the Neo4j replica. This ensures that users can immediately engage with these new communities. Simultaneously, we asynchronously update the "tags" redundancy in MongoDB.

As evident from these examples, we prioritize asynchronous updates to redundancies, favoring availability over strict consistency. Nevertheless, we recognize the need to periodically verify data consistency. Our service functions include checks for query aborts or rollbacks, but these are reserved for extreme cases and not applied to these redundancies.

For our most vital redundancy, the Watchlist, we've introduced a "WatchlistConsistencyChecker" class. When triggered in the background, this class identifies and rectifies any inconsistencies between the watchlist and reviews. We have opted to trigger this checker not periodically, but in response to user actions, specifically when a user clicks the "expand" button to view their entire watchlist. This way, the checker activates when it's likely that inconsistencies may arise, without resorting to frequent rollback checks. This approach effectively manages inconsistencies caused by delays or replica issues, preserving both availability and reasonable data integrity.

This strategy ensures that our system provides a responsive user experience while maintaining data integrity, striking a balance between consistency and availability.