Feyzan Colak, Flavio Messina, Noemi Cherchi
Large-scale and multi-structured databases project
2023-2024

Contents

Contents **1**

1 Introduction **2**

2 Analysis **3**
2.1 Actors . . . 3
2.2 Functional Requirements . . . 3
2.3 Non Functional Requirements . . . 4
2.4 UML class diagram . . . 5
2.5 UML use case diagram . . . 6
2.6 Scenarios . . . 7
2.7 Data Modeling . . . 13

3 Design **14**
3.1 Document Database . . . 14

4 Implementation **22**
4.1 Development Environment . . . 22
4.2 Main Modules . . . 23
4.3 Adopted Patterns and Techniques . . . 29
4.4 Description of Main Classes . . . 31
4.5 MongoDB queries . . . 34
4.6 GraphDB queries . . . 44

5 Testing **53**
5.1 Structural Testing . . . 53
5.2 Functional Testing . . . 56
5.3 Performance Testing . . . 57

6 Conclusion **59**
6.1 Future Work . . . 59
6.2 Conclusion . . . 59

Introduction

Aim of the project

MangaVerse is a web application project developed for the Large-scale and Multi-structured Databases course at the University of Pisa. This web application aims to create a **comprehensive platform** for exploring a vast collection of **manga** and **anime**, while also fostering **interaction** among users.

The website is accessible without requiring a login, offering a limited set of functionalities. However, once users log in, they gain access to a wide range of features designed to **personalize** their experience, particularly through **social interactions**.

MangaVerse is designed to be a user-friendly, seamless, and engaging platform for **manga and anime enthusiasts**, to connect with like-minded individuals, discover new content, and share their thoughts and opinions.

To explore the project development details, here is the URL of the GitHub repository: https://github.com/xDarkFlamesx/MangaUniverse.

Application Highlights

- **Comprehensive Media Database**: MangaVerse offers a vast collection of **manga** and **anime** entries, complete with detailed information, ratings, and reviews, to help users make informed decisions.
- **Search and Filter Functionality**: Users can easily search for specific **manga** or **anime** titles, genres, authors, and more, using advanced **filtering** options to refine their search results.
- **User Profiles**: Users can create profiles to **personalize** their experience, track their activity, and connect with other users who share similar interests.
- **User and Media Content Suggestions**: The platform provides tailored suggestions based on user interactions, preferences, and information, enhancing the discovery of new **manga**, **anime**, and **users**.
- **Social Features**: Logged-in users can engage with the community through **reviews**, **likes**, and **follows**, fostering a vibrant community of **manga** and **anime enthusiasts**.
- **Managerial Roles and Analytics**: MangaVerse includes **managerial roles** with access to an **analytics dashboard** for tracking media content and user activities. Managers can **add, update, or remove manga** and **anime entries**, monitor trends, and analyze ratings to optimize content offerings.
- **Content Management**: Managers can efficiently manage media content and user accounts, ensuring that the platform remains **up-to-date** and **relevant**.

Analysis

Actors

- **Unregistered User**: A visitor who has not logged in on the platform.
- **Registered User**: A user who has created an account on the platform.
- **Manager**: A registered user with administrative privileges.

Functional Requirements

The system should provide the following features for each type of user:

- **Browse Media Contents**.
- **Search and Filter Media Contents**:
  - Find specific manga or anime by title.
  - Utilize basic filtering options to refine the media content list.
- **View Media Content Trends**.
- **View Media Content**:
  - View limited information about each media content.
- **View Media Content Details**:
  - View detailed information about each media content.
  - View reviews and ratings for each media content.
  - View number of likes for each media content.
- **Browse Users**.
- **Search Users by Username**.
- **View User**:
  - View limited information about each user.
- **View User Details**:
  - View detailed information about each user.
  - View anime and manga liked by the user.
  - View followers and following of the user.

Unregistered User

- **Register/Login**:
  - Create a new account to access additional features.
  - Use valid credentials (email and password) to log into the account.

Registered User

- **Logout**.
- **Profile Management**:
  - Edit and update personal information (e.g., profile picture, bio).
  - Delete own profile.
- **Like/Unlike Media Contents**.
- **Follow/Unfollow Users**.
- **Review Media Contents**:
  - Add comment and rating to manga and anime.
  - Edit/Delete own reviews.
- **Advanced Recommendations**:
  - Receive media content suggestions based on user interactions and personal information.
  - Receive users suggestions based on user interactions.

Manager

- **Logout**.
- **Analytics Dashboard**:
  - View user analytics (distribution and app rating).
  - View manga analytics (trends and average rating).
  - View anime analytics (trends and average rating).
- **Content Management**:
  - Add new media content (manga and anime).
  - Update/Remove existing media content.

Non Functional Requirements

Performance

- **Response Time**: The system should have low latency, with pages loading within an acceptable timeframe.
- **Scalability**: The system should be able to handle an increasing number of users and data without significant degradation in performance.
- **Concurrency**: The application should support multiple users simultaneously without performance bottlenecks. For very high traffic scenarios, acceptable delays may be introduced.
- **Availability**: The system should be available 24/7, with minimal downtime for maintenance.
- **Replication**: The system should have data replication to ensure data availability and fault tolerance.

Security

- **Controlled User Operations**: Users should only be able to perform operations that they are authorized to do.

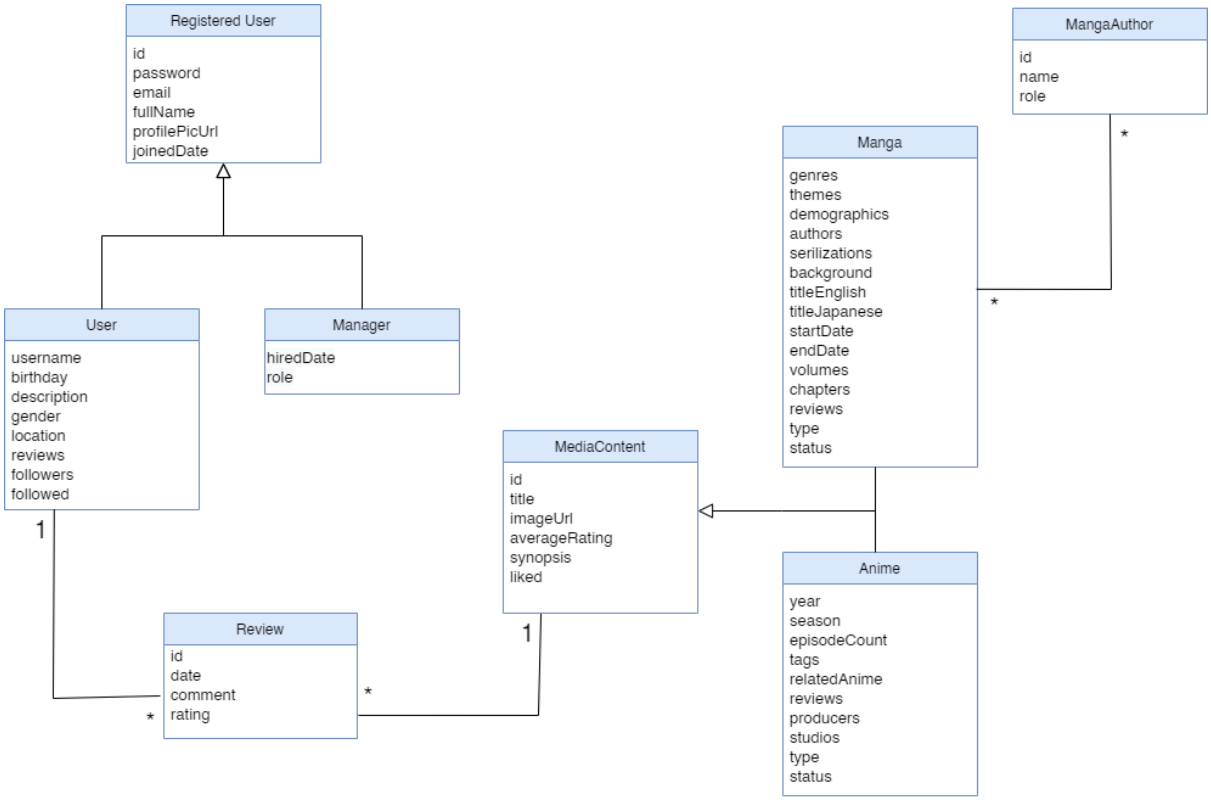
Data Integrity

- **Data Consistency**: The system should maintain data consistency across all components and databases.

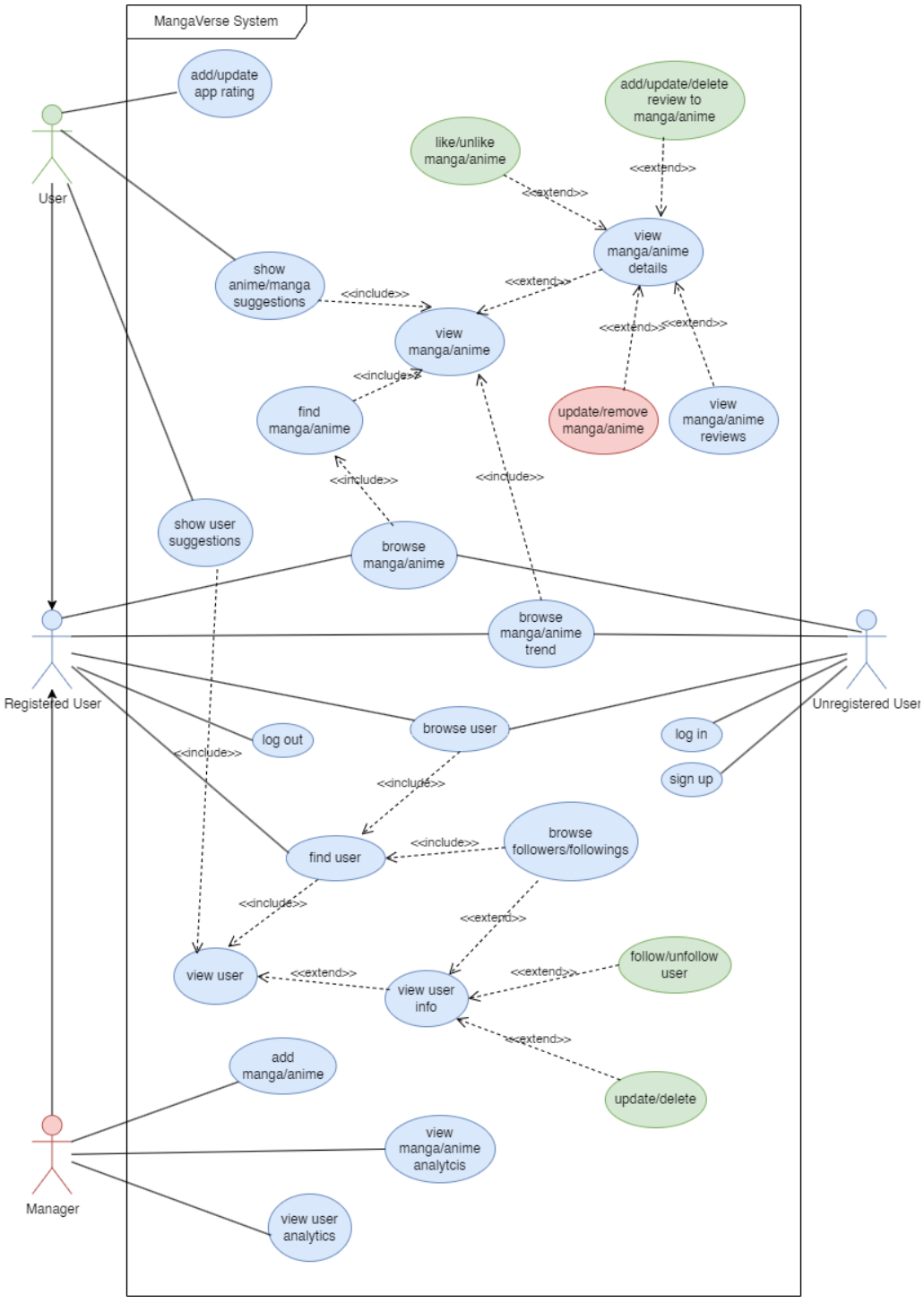
User Interface

- **Responsiveness**: The user interface should be responsive, providing a consistent and seamless experience across various devices and screen sizes.
- **Intuitiveness**: The interface should be user-friendly, with clear navigation and easily understandable features.

UML class diagram



UML use case diagram



Scenarios

Use Case	**Register Account**
Primary Actor	Unregistered User
Description	Allows user to sign up
Pre-conditions	The account must not be already exist
Main Event Steps	1. The user navigates to the registration page. 2. The user fills out the registration form with valid information. 3. The system validates the entered information. 4. After successful validation, the system created a new user account
Post-conditions	The user is registered to the system
Correlated Use Cases	-

Use Case	**Log In**
Primary Actor	Registered User
Description	Allows user to login
Pre-conditions	The user must not be already logged in
Main Event Steps	1. The user navigates to the login page. 2. The system shows the login form to the user and asks for email and password 3. The user enters valid login credentials. 4. The system validates the entered credentials 5. After successful validation, the system logs the user.
Post-conditions	1. The user is authenticated and logged in 2. The user is redirected to the main page for registered user.
Correlated Use Cases	-

Use Case	**Log Out**
Primary Actor	Registered User
Description	Allows user to logged out
Pre-conditions	The user must be logged in
Main Event Steps	The system starts the logout process
Post-conditions	The user is logged out. The user is directed to the main page
Correlated Use Cases	-

Use Case	**Browse Anime/Manga**
Primary Actor	Registered User, Unregistered User
Description	Allows both registered and unregistered users to browse anime and manga
Pre-conditions	-
Main Event Steps	1. The user navigates to the browse section. 2. The system displays a list of available anime and manga.
Post-conditions	1. The user will view the list of anime or manga 2. The user can filter or sort the list based on various criteria.
Correlated Use Cases	Find Anime/Manga

Use Case	**Find Anime/Manga**
Primary Actor	Registered User, Unregistered User
Description	Allows users to search for specific anime or manga.
Pre-conditions	-
Main Event Steps	1. The user navigates to the search section of anime/manga. 2. The user enters search criteria (e.g., title). 3. The system retrieves a list of available anime/manga. 4. The user can apply filters or sort the list.
Post-conditions	The system displays a result of the search The user views a list of anime/manga matching with the used criteria
Correlated Use Cases	View Anime/Manga

Use Case	**View Anime/Manga**
Primary Actor	Registered User, Unregistered User
Description	Allows users to view details of a specific anime or manga
Pre-conditions	-
Main Event Steps	1. The user selects an anime or manga from the list or search results. 2. The system displays pre information about the selected anime/manga.
Post-conditions	The user views pre information about the selected anime/manga
Correlated Use Cases	-

Use Case	View Anime/Manga Details
Primary Actor	Registered User, Unregistered User
Description	Allows users to view all the information about the specific anime or manga including reviews
Pre-conditions	-
Main Event Steps	1. The user selects an anime or manga. 2. The system retrieves detailed information, including reviews.
Post-conditions	
Correlated Use Cases	-

Use Case	View Anime/Manga Reviews
Primary Actor	Registered User, Unregistered User
Description	Allows users to view reviews of a specific anime or manga
Pre-conditions	-
Main Event Steps	1. The user navigates to the reviews section of a specific anime or manga. 2. The system retrieves the reviews. 3. The system displays the reviews to the user.
Post-conditions	The user views the reviews of the specific anime or manga
Correlated Use Cases	-

Use Case	Show Anime/Manga Suggestions
Primary Actor	User
Description	Allows users to view anime or manga suggestions based on their likes or followings
Pre-conditions	User must be logged in
Main Event Steps	1. The user navigates to the suggestions part of the main page. 2. The system retrieves suggestions based on user preferences and followings. 3. The system displays the suggestions to the user.
Post-conditions	The user views the suggested anime or manga
Correlated Use Cases	View Manga/Anime

Use Case	Browse Anime/Manga Trend
Primary Actor	Registered USer, Unregistered User
Description	Allows users to browse trending anime or manga
Pre-conditions	User must be logged in
Main Event Steps	1. The user navigates to the trending section of the main page. 2. The system retrieves trending anime or manga. 3. The system displays the trending items to the user.
Post-conditions	-
Correlated Use Cases	View Anime/Manga

Use Case	Like Anime/Manga
Primary Actor	User
Description	Allows users to like an anime or manga
Pre-conditions	User must be logged in
Main Event Steps	1. The user clicks the like or unlike button. 2. The system updates the like status.
Post-conditions	1. The anime or manga is liked or liked by the user and added to the liked anime/manga list 2. The appearance of like button is changed to filled version of the button
Correlated Use Cases	-

Use Case	Add Review to Anime/Manga
Primary Actor	User
Description	Allows user to make comment or rating for specific anime/manga
Pre-conditions	User must be logged in
Main Event Steps	1. The user selects an anime or manga. 2. The user navigates to the review section. 3. The user adds a review that can be either comment or rating the anime/-manga 4. The system ensures that at least one of the comment or rate value is filled 5. The system saves the review and updates the latest reviews
Post-conditions	The review is added to the specific anime/manga and visible on the page
Correlated Use Cases	-

Use Case	Show User Suggestions
Primary Actor	User
Description	Allows users to view suggested users based on similar tastes or may know
Pre-conditions	User must be logged in
Main Event Steps	1. The user navigates to the profile page's suggested users section 2. The system retrieves suggested users based on user preferences and followings 3. The system displays the suggested users to the user
Post-conditions	The user views the suggested users.
Correlated Use Cases	View User

Use Case	Find User
Primary Actor	Registered User, Unregistered User
Description	Allows users to search for specific users according to their usernames
Pre-conditions	-
Main Event Steps	1. The user navigates to the user user search section of the page. 2. The user enters search criteria. 3. The system retrieves matching users based on criteria. 4. The system displays the search results to the user.
Post-conditions	The user views the search results.
Correlated Use Cases	View User

Use Case	Browse Followings/Followers
Primary Actor	Registered User, Unregistered User
Description	Allows users to browse their or others followers and followings
Pre-conditions	-
Main Event Steps	1. The user navigates to the profile page. 2. The user selects the followers or followings section. 3. The system retrieves the list of followers or followings. 4. The system displays the list to the user.
Post-conditions	The user views their followers and followings.
Correlated Use Cases	Find User,View User Info

Use Case	**Follow User**
Primary Actor	Registered User
Description	Allows users to follow another user
Pre-conditions	User must be logged in
Main Event Steps	1. The user selects a user to follow 2. The user navigates to that user's profile page. 3. The systems displays the profile of other user. 4. The user clicks the follow button. 5. The system add the user to the following list of the other user and updates the count on the profile 6. The system updates the follow/unfollow status.
Post-conditions	The user starts to follow the other user
Correlated Use Cases	-

Use Case	**View User Information**
Primary Actor	Registered User, Unregistered User
Description	Allows users to view detailed information about another user
Pre-conditions	-
Main Event Steps	1. The user selects a user from the list or search results or browse 2. The system retrieves detailed information about the selected user. 3. The system displays the detailed information to the user.
Post-conditions	The user views detailed information about the selected user.
Correlated Use Cases	-

Use Case	**View Analytics**
Primary Actor	Manager
Description	Allows managers to view analytics related to users and anime/manga
Pre-conditions	The user must be logged in with manager credentials
Main Event Steps	1. The manager navigates to the analytics section of their page 2. The system retrieves analytics data. 3. The system displays the analytics data to the manager.
Post-conditions	The manager views the analytics data.
Correlated Use Cases	-

Data Modeling

Data Collection

Sources: myAnimeList.net, anilist.co, kitsu.io, livechart.me, anime-planet.com, notify.moe, anisearch.com, anidb.net.

Description: The datasets contain information about anime, manga, users, and scores. Anime and reviews are stored in separate CSV files. The manga dataset is collected from myAnimeList.net, scraped using the Official API and Jikan API.

Variety: Anime are collected from 8 different sources, manga from one source, and users/reviews from the same sources (myAnimeList and Anime-Planet). The data is structured across 4 CSV files and a JSON file.

Volume: ~3 GB. The datasets contain approximately 10 million reviews, 40k anime entries, 70k manga entries, and 200k users.

Data Cleaning and Preprocessing

The data cleaning and preprocessing involved several steps to ensure consistency and usability of the datasets:

- **Data Integration and Reduction**: Integrating anime, reviews, and user data from various sources required dealing with non-unique IDs across datasets. To ensure proper integration, new unique IDs were generated for each entity. Due to issues with data quality, a significant number of users and their associated reviews were removed. This process resulted in a refined dataset containing 10,000 users and 600,000 reviews.
- **Review Comment Generation**: Some sources lacked explicit comments for reviews. To address this, a script was employed to generate generic comments based on the review ratings, ensuring completeness of the review dataset.
- **Synthetic Data Generation**: Many users did not have sufficient reviews for meaningful analytics. Synthetic reviews were generated to supplement these users, enabling more robust analysis of user interactions and content preferences.
- **Data Pruning**: Extraneous information about users, anime, and manga that were not relevant to the project goals were removed to streamline the datasets.
- **Data Augmentation**: Essential user information such as email, password, and profile picture, which were necessary for user management features, were added to the dataset.
- **Consistency Checks**: Before insertion into the Document DB collections, rigorous data consistency checks were conducted to uphold accuracy and reliability. For instance, chronological consistency was enforced for fields such as the joined date, ensuring it fell after the user's birthdate and before any related review dates. Additionally, user locations were validated to ensure they mapped to valid countries. Upon insertion, Document DB's ids were utilized for entity linking and to maintain data integrity.
- **Graph Database Population**: Fake relationships were created, such as follow relationships between users and like relationships between users and anime/manga, to populate the graph database.

These steps were crucial in preparing the datasets for effective utilization within the MangaVerse platform, ensuring data quality and integrity across all components.

Python was predominantly used for data preprocessing tasks, leveraging its flexibility and extensive libraries. Additionally, Java was employed for specific tasks such as adding and updating redundancies in the document database.

Design

The system is designed to handle a substantial volume of data with varying attributes in both number and type. There are several anime and manga entries that contains missing or incomplete information, and the system must be able to handle this, avoiding meaningless memory occupation. To accommodate this, a Document Database was chosen for its flexibility and schema-less nature, ease of use, and high-performance capabilities. This choice enables the execution of complex queries, including advanced filtering across different attribute types.

In addition, the implementation of social networking functionalities necessitated the use of a Graph Database. This allows for efficient traversal of relationships between entities and effectively manages connections between different entities such as users, anime, and manga.

Document Database

The decision to use a document database, specifically MongoDB, was driven by its **flexibility** and **schema-less nature**, which allows for the storage of data with varying attributes in both number and type. This adaptability makes MongoDB ideal for handling diverse and dynamic data models.

MongoDB also provides a **high-performance environment** for executing complex queries, reducing the need for joins and improving overall application performance by minimizing database access. This is particularly beneficial for applications like MangaVerse that require fast response times and can benefit from pre-computed relationships between entities. By storing embedded relationships directly within documents, MongoDB enables quicker data retrieval, enhancing user experience.

To avoid normalized data, **data redundancy** is used to define relationships between entities. Although this technique can lead to increased memory usage, it enables faster queries and fewer database accesses. For applications with high query complexity and rapidly growing data, such as MangaVerse, this is an optimal trade-off.

To define **one-to-many relationships**, the documents linking pattern is used. In this pattern, one entity stores a list of the IDs of related entities, allowing for quick retrieval without multiple queries. This approach is used for retrieving reviews written by a user or reviews associated with a specific anime or manga. Additionally, document embedding is used for storing the latest reviews within the anime or manga documents, enabling fast access to this information when a user views the anime or manga page.

The document database also includes other redundancies between collections and between the document database and the graph database. For example, fields such as the number of likes for a manga or anime, the number of followers and followings for a user, and the average rating of media content are stored redundantly. To maintain consistency without unnecessary database accesses, a flag called **'avg_rating_last_update'** is used to indicate whether the average rating is up-to-date.

MongoDB is also a **scalable database**, capable of handling large volumes of data and traffic. It supports distribution, providing **high availability**, **fault tolerance**, and **data integrity**. This means MongoDB can easily be scaled out to accommodate growth in both data and traffic, ensuring consistent performance and reliability as the application grows.

Collections

The database contains the following collections:

- **Anime**: This collection stores information about anime, including a list of review IDs and the most recent reviews as nested documents.
- **Manga**: This collection stores information about manga, including a list of review IDs and the most recent reviews as nested documents.
- **Reviews**: This collection stores user ratings and comments for media content. To enhance performance and reduce multiple queries, it includes some user and media redundancies, especially for suggestions and analytics.
- **Users**: This collection stores user data along with a list of review IDs.

Analytics and Suggestions

The application performs various analytics on users, manga, and anime to provide the manager with valuable information regarding user distribution and the average rating of the application or media content. These analytics are grouped by different criteria such as genre, season, and year. Additionally, the application offers personalized media content suggestions to users.

The following is a comprehensive list of all the queries that the application should be able to execute:

- **User Analytics**:
  - Retrieve the distribution of users grouped by gender, joined_on date, birthday, and country;
  - Retrieve the average rating of the application grouped by gender, joined_on date, birthday, and country;
- **Media Content Analytics**:
  - Retrieve the average rating of the media content grouped by various criteria (e.g., genre, type, demographics, author, etc.);
  - Retrieve the average rating of a media content per month or year;
- **Suggestions**:
  - Retrieve the top media content suggestions for a user based on the user's location or birthday.

CRUD operations

- **CREATE**
  - Create a user
  - Create an anime
  - Create a manga
  - Create a review
- **UPDATE**
  - Update user's information
  - Update media content details
  - Update a review
- **DELETE**
  - Delete a user
  - Delete a media content

- Delete a review by its ID
- Delete reviews of a user
- Delete reviews of a media content
- Delete reviews not related to any media content
- Delete a review not related to any user

- **READ**
  - Read a user by the ID
  - Read the first N users by username
  - Read a media content by its ID
  - Read a page of media contents by filters
  - Read reviews of a specified user
  - Read reviews of a specified media content

JSON document example

Anime:


```
{
  "_id": "65789bb52f5d29465d0abcfc",
  "title": "\"Aesop\" no Ohanashi yori: Ushi to Kaeru, Yokubatta Inu",
  "type": "MOVIE",
  "episodes": 1,
  "status": "FINISHED",
  "picture": "https://cdn.myanimelist.net/images/anime/3/65151.jpg",
  "tags": [
    "family friendly",
    "fantasy",
    "frogs",
    "kids"
  ],
  "synopsis": "Based on Aesop's Fables.",
  "latest_reviews": [
    {
      "id": "657ed1b40481d3954cf8d69c",
      "comment": "Struggles to maintain interest; fails to captivate.",
      "date": "2022-04-11T22:00:00.000+00:00",
      "user": {
        "id": "6577877ce683762347607f42",
        "username": "dreadstuff",
        "picture": "https://imgbox.com/7MaTkBQR"
      }
    }
  ],
  "anime_season": {
    "season": "WINTER",
    "year": 1970,
    "average_rating": 4.5,
    "avg_rating_last_update": true
  },
  "review_ids": [],
  "likes": 7
}
```

Manga:


```
    {
      "_id": "657ac61cb34f5514b91eabc1",
      "title": "H20",
      "type": "MANHWA",
      "status": "FINISHED",
      "volumes": 7,
      "chapters": 43,
      "genres": [
        "Romance",
        "Slice of Life"
      ],
      "demographics": [
        "SHOUJO"
      ],
      "authors": [
        {
          "serializations": "Wink"
        }
      ],
      "synopsis": "Menga is simply known as the vice rep and is bullied. Hanako has moved...",
      "title_english": "H20",
      "picture": "https://cdn.myanimelist.net/images/manga/1/10531.jpg",
      "average_rating": 3.67,
      "latest_reviews": [
        {
          "id": "66682d94bebc20d9557bba39",
          "comment": "Beautiful artwork and engaging characters.",
          "date": "2024-06-17T21:33:37.000+00:00",
          "rating": 7,
          "user": {
            "id": "6577877be68376234760635b",
            "username": "Sanji-kun",
            "picture": "images/account-icon.png"
          }
        }
      ],
      "start_date": null,
      "end_date": null,
      "avg_rating_last_update": true,
      "review_ids": [
        "66682d94bebc20d9557bba39"
      ]
    }
```

Reviews:


```
      {
        "_id": "66682a4fbebc20d9557b7544",
        "user": {
          "id": "6577877ce683762347607e35",
          "username": "velmodine",
          "picture": "https://thypix.com/wp-content/uploads/2021/10/anime-avatar-profile-
    pic...",
          "location": "Montenegro"
        },
        "manga": {
          "id": "657ac61bb34f5514b91ea22a",
          "title": "Kaguya-sama wa Kokurasetai: Tensai-tachi no Renai Zunousen",
          "rating": 8,
          "comment": null,
          "date": "2022-07-30T00:00:00.000+00:00"
        }
      }
```


Users:


```
      {
        "_id": "6577877be68376234760585b",
        "email": "millerderek@example.com",
        "password": "6ed60c305f7e923745ec6e3c010faaf8970c1fb8ea73987f9bf6d5ed053aa94c",
        "description": "Embodied by the spirit of a thousand anime protagonists.",
        "picture": "https://thypix.com/wp-content/uploads/2021/10/manga-profile-picture
    -82...",
        "username": "Crystal",
        "gender": "Female",
        "location": "Ukraine",
        "joined_on": "2017-06-06T00:00:00.000+00:00",
        "app_rating": 2,
        "followed": 35,
        "followers": 39,
        "review_ids": []
      }
```


Redundancy

In a document database, data redundancy can be leveraged to enhance application performance and reduce the number of database accesses. However, the application must manage the consistency between the original data and their redundant copies. This consistency maintenance can impact performance and availability. Therefore, optimal strategies are necessary to ensure data consistency while preserving application performance and availability.

The following redundancies are implemented in the database:

- **Latest Reviews**:
  In the anime and manga collections, a field contains the latest 5 reviews for that specific media content. This approach ensures fast retrieval.

- **Average Rating**:
  In the anime and manga collections, a field contains the average rating of the media content. This field is updated every time a new review is written.

- **Number of Likes**:
  In the anime and manga collections, a field contains the number of likes. This field is updated every time a new like relationship is created or deleted.

- **Followers and Followings**:
  In the user collection, fields contain the number of followers and followings. These fields are updated every time a new follow relationship is created or deleted.

- **User Field in Reviews**:
  In the reviews collection, a field contains user data, such as id, username, picture, location, and birthday. This data is used for suggestion purposes.

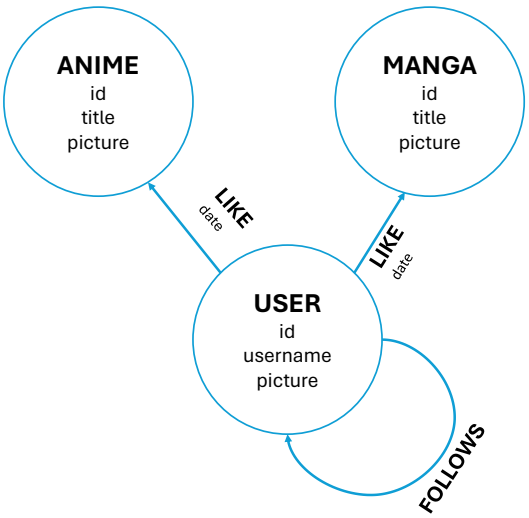
- **Media Content Field in Reviews**:
  In the reviews collection, a field contains information about the anime or manga the review pertains to, including the media content id and title.

- **Review Ids**:
  A list of review ids is stored in the anime, manga, and user collections. This list is used to quickly retrieve the reviews of media content and users.

Graph Database

The inclusion of social networking functionalities, such as user interactions, likes, and follows, necessitated the integration of a graph database to efficiently manage relationships between entities. A graph database operates on the concepts of nodes and relationships, which are ideal for this purpose. Unlike traditional databases that rely on join tables—often resulting in higher memory usage and slower execution times due to multiple database accesses—a graph database optimizes traversal for relationships. This makes it particularly well-suited for managing complex connections between users, anime, and manga, ensuring faster and more efficient data retrieval and manipulation.

Additionally, graph databases provide enhanced flexibility and scalability, allowing for easier implementation of new features and relationships as the application grows. They support advanced queries that can traverse multiple relationships in a single operation, offering a significant performance boost for social networking applications where interconnected data is prevalent.



Nodes

The database will have the following nodes:

- User: This node will store information about users, such as id, usernames, and picture.
- Anime: This node will store information about anime, such as id, titles and picture.
- Manga: This node will store information about manga, such as id, titles and picture.

Relationships

The database will have the following relationships:

- LIKE: This relationship will connect users to anime and manga nodes. It will store the date when the user liked the media content.
- FOLLOW: This relationship will connect users to other users.

CRUD operations

- **CREATE**
  - Create a user node
  - Create an anime node
  - Create a manga node
  - Create a LIKE edge between a user and an anime or manga
  - Create a FOLLOW edge between two users
- UPDATE
  - Update user's picture and/or username
  - Update media content picture and/or title
- DELETE
  - Delete a user node and all edge connected to it
  - Delete a media content node and all edge connected to it
  - Delete a LIKE edge between a user and an anime or manga
  - Delete a FOLLOW edge between two users
- READ
  - Read the numer of FOLLOW edges outcoming from a user
  - Read the number of FOLLOW edges incoming to a user
  - Read the number of LIKE edges incoming to an anime or manga
  - Read the list of followers of a user
  - Read the list of followings of a user

Availability and Partition Tolerance (AP)

MangaVerse, as a social network, adopts the AP configuration of the CAP theorem, emphasizing Availability and Partition Tolerance. This ensures users can reliably access the application, interact with others, and engage with media content, even though data consistency may vary (Eventual Consistency). This approach enhances user experience by prioritizing uninterrupted service over strict data consistency.

Replicas

For MangaVerse, three MongoDB replicas were deployed to ensure high availability, partition tolerance, and rapid response times. This configuration allows the system to remain accessible and responsive even during network partitions or node failures. However, managing data consistency across multiple replicas requires careful handling to achieve eventual consistency without compromising availability.

Sharding

Even though sharding is not currently implemented in MangaVerse, it's crucial to consider its implications. Sharding is a method used to distribute data across multiple machines, essential for managing data growth. As datasets expand, a single machine may become inadequate for storing and managing data with acceptable read and write speeds. Document databases like MongoDB support sharding, which facilitates horizontal scaling.

Choosing the right shard key for each collection is critical for effective sharding in MongoDB. The shard key determines how data is distributed across shards. Factors such as the types of queries performed, document relationships, and document structure must be carefully evaluated when selecting a shard key to ensure optimal performance and scalability.

In its current implementation, MangaVerse does not support sharding. A potential approach could involve sharding anime and manga data by unique identifiers like id or title, and reviews by media content id or title to keep related data together on the same shard. However, this strategy may lead to performance challenges. Queries involving reviews related to a specific user or media content suggestions based on user location and birthday could require accessing multiple shards, potentially reducing application performance. Similarly, analytical tasks like computing average ratings across anime and manga collections would necessitate querying all shards, which can be inefficient.

Implementation

Development Environment

The implementation of the MangaVerse web application was carried out in a development environment chosen from the requirements and the design study precedently discussed. That includes the following tools and technologies:

Programming Languages

- **Backend:** Java.
- **Frontend:** HTML, CSS, JavaScript.
- **Data Preprocessing:** Python and Java.

Database

- **Document Database:** MongoDB.
- **Graph Database:** Neo4j.

Integrated Development Environment

- Intellij IDEA.

Collaborative Development Environment

- Git.
- GitHub.

Web Server

- Apache Tomcat.

Build Automation

- Maven.

Testing

- JUnit.

Main Modules

- Configuration
- Controller
- DAO (Data Access Objects)
- DTO (Data Transfer Objects)
- Model
- Service
- Utils
- User Interface (JSP, CSS ans JS files)

Configuration

The configuration module includes a class named *AppServletContextListener*, which is responsible for managing **database connections**, the **TaskManager** for handling asynchronous operations and their priorities, and the **PeriodicExecutorTaskService** for handling periodic tasks. This configuration class implements the *ServletContextListener* interface.

The *@WebListener* annotation is used to listen for application lifecycle events. The *ServletContextListener* interface contains two methods: *contextInitialized(ServletContextEvent sce)* and *contextDestroyed(ServletContextEvent sce)*.

The *contextInitialized* method is called when the web application starts. It opens the database connections and initiates the TaskManager and PeriodicExecutorTaskService. The *contextDestroyed* method is invoked when the web application shuts down. It closes all database connections and stops the services.

For MongoDB was used this configuration:

```
    settings = MongoClientSettings.builder()
                .applyConnectionString(connectionString)
                .writeConcern(WriteConcern.W1)
                .readPreference(ReadPreference.nearest())
                .retryWrites(true)
                .readConcern(ReadConcern.LOCAL)
                .build();

```

- **WriteConcern.W1**: Write concern describes the level of acknowledgment requested from MongoDB for write operations. WriteConcern.W1 means the write operation must be acknowledged by the primary server before the client proceeds. This ensures that the data is written to at least the primary replica set member but doesn't wait for acknowledgment from secondary members.
- **ReadPreference.nearest()**: Read preference determines how MongoDB clients route read operations to members of a replica set. ReadPreference.nearest() means the client will read from the member of the replica set that has the least network latency, irrespective of whether it's a primary or secondary member. This can help reduce read latency.
- **retryWrites(true)**: RetryWrites enables retryable writes. If a network error occurs during a write operation, MongoDB will automatically retry the write operation once. This can help to handle transient network errors more gracefully.
- **ReadConcern.LOCAL**: Read concern specifies the level of isolation for read operations. ReadConcern.LOCAL ensures that a query returns the most recent data a node has locally, without waiting for data to be replicated to other nodes. This is the default level and it balances consistency and performance.

Controller

The controller module acts as an intermediary between user requests and the backend of the MangaVerse web application through servlet classes. It receives user requests, processes them, and returns the corresponding responses. The controller classes are implemented using *HttpServlet* to handle user requests and responses.

In their intermediary role, the controller classes serve as a bridge between the user interface and backend logic. When a user interacts with the web application, their actions are translated into HTTP requests, which are handled by the relevant servlet class in the controller module. Each controller class utilizes a switch-case structure to determine the requested action and invokes the appropriate handler method accordingly. This structure allows for clear and organized routing of requests to their corresponding handler methods. After processing the request, the servlet generates the appropriate response.

Example code snippet from MediaContentServlet:

```
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    IOException, ServletException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     IOException, ServletException {
        processRequest(request, response);
    }

    protected void processRequest(HttpServletRequest request, HttpServletResponse response
    ) throws IOException, ServletException {
        switch (request.getParameter("action")) {
            case "editProfile" -> handleUpdate(request, response);
            case "deleteProfile" -> handleDelete(request, response);
            case "getReviews" -> handleGetReviewsByUserId(request, response);
            case "rateApp" -> handleRateApp(request, response);
            case "suggestedMediaContent" -> handleSuggestedMediaContent(request, response)
    ;
            case "suggestedUsers" -> handleSuggestedUsers(request, response);
            case null, default -> handleLoadPage(request, response);
        }
    }
}

```

The controller module contains the following classes:

- **Exception:**
  - **NotAuthorizedException:** This exception is thrown when the user is not authorized to access the requested resource.
- **AuthServlet:** to handle rquests on authentication page.
- **MainPageServlet:** to handle requests on the mainPage.
- **ManagerServlet:** to handle requests on the manager page.
- **MediaContentServlet:** to handle requests on the media content page and retrieve or set media content releted info.
- **ProfileServlet:** to handle requests on the profile page.
- **UserServlet:** to handle requests related to user model.

DAO (Data Access Objects)

The DAO module includes the logic for accessing and managing data in the database and provides data retrieval, storage and manipulation. This module includes classes with CRUD (create, read, update, delete) operations and query executions. It provides a layer of abstraction between the database and the rest of the application and ensures the separation of concerns. The DAO module contains the following classes:

- Enums
  - DataRepositoryEnum
- Exceptions
- Interfaces
  - MediaContentDAO
  - ReviewDAO
  - UserDAO
- Mongo
  - AnimeDAOMongoImpl
  - BaseMongoDBDAO
  - MangaDAOMongoImpl
  - ReviewDAOMongoImpl
  - UserDAOMongoImpl
- Neo4j
  - AnimeDAONeo4jImpl
  - BaseNeo4jDAO
  - MangaDAONeo4jImpl
  - UserDAONeo4jImpl
- DAOLocator

Example code snippet from MangaDAOMongoImpl:

```
{
    //MongoDB queries
    //Best genres/themes/demographics/authors based on the average rating
    @Override
    public Map<String, Double> getBestCriteria (String criteria, boolean isArray, int page
    ) throws DAOException {
        try  {
            MongoCollection<Document> mangaCollection = getCollection(COLLECTION_NAME);
            int pageOffset = (page-1)*Constants.PAGE_SIZE;

            List<Bson> pipeline;
            if (isArray) {
                pipeline = List.of(
                        match(and(exists(criteria), ne("average_rating", null))),
                        unwind("$" + criteria),
                        group("$" + criteria, avg("criteria_average_rating", "
    $average_rating")),
                        sort(descending("criteria_average_rating")),
                        skip(pageOffset),
                        limit(25)
                );
            } else {
                pipeline = List.of(
                        match(Filters.exists(criteria)),
                        group("$" + criteria, avg("criteria_average_rating", "
    $average_rating")),
                        sort(new Document("criteria_average_rating", -1)),
                        skip(pageOffset),
                        limit(25)
                );
            }

            List <Document> document = mangaCollection.aggregate(pipeline).into(new
    ArrayList<>());
            Map<String, Double> bestCriteria = new LinkedHashMap<>();
            for (Document doc : document) {
                Double avgRating = doc.get("criteria_average_rating") instanceof Integer?
                        doc.getInteger("criteria_average_rating").doubleValue() :
                        doc.getDouble("criteria_average_rating");
                if (criteria.equals("authors")) {
                    bestCriteria.put(doc.get("_id", Document.class).getString("name"),
    avgRating);
                } else {
                    bestCriteria.put(doc.get("_id").toString(), avgRating);
                }
            }

            return bestCriteria;

        } catch (Exception e) {
            throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
        }
    }
}

```

Example code snippet from UserDAONeo4jImpl:

```
{
    /**
    * Retrieves a list of users following a specific user from the Neo4j database.
    *
    * @param userId The ID of the user whose followers are to be retrieved.
    * @param loggedUserId The ID of the user requesting the list of followers.
    * @return A list of RegisteredUserDTO objects representing the followers of the
    specified user.
    * @throws DAOException If an error occurs while retrieving the followers list.
    */
   @Override
   public List<UserSummaryDTO> getFirstNFollowers(String userId, String loggedUserId)
    throws DAOException {
       try (Session session = getSession()) {
           StringBuilder queryBuilder = new StringBuilder("MATCH (follower:User)-[:FOLLOWS
    ]->(:User {id: $userId}) ");
           if (loggedUserId != null) {
               queryBuilder.append("WHERE follower.id <> $loggedUserId ");
           }
           queryBuilder.append("RETURN follower AS user ");
           queryBuilder.append("ORDER BY follower.username ");
           queryBuilder.append("LIMIT 10");
           String query = queryBuilder.toString();

           Map<String, Object> params = new HashMap<>();
           params.put("userId", userId);
           if (loggedUserId != null) {
               params.put("loggedUserId", loggedUserId);
           }

           List<Record> records = session.executeRead(
                   tx -> tx.run(query, params).list()
           );

           return records.isEmpty() ? null : records.stream()
                   .map(this::recordToUserSummaryDTO)
                   .toList();

       } catch (Neo4jException e) {
           throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

       } catch (Exception e) {
           throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
       }
   }
}

```

DTO (Data Transfer Objects)

The DTO modules are the intermediary class between presentation layer and the DAO module in the web application. They transfer data structures between different layers and components of the application in a more standardized way.

Model

- Enums
- Media Content
  - Anime
  - Manga
  - Manga Author
  - Media Content
- Registered User
  - Mangager
  - Registered User
  - User
- Review

Service

Service module has also important role in the web application. The classes in the service module are responsible for containing the business logic and maintaining interaction between the DAO classes and the presentation layer. It handles complex operations with guarantying that the application's core functionalities are executed correctly. Some of the services that are provided in the service module are: *UserService, MediaContentService, ReviewService, TaskManager, ExecuterTaskService.* The package structure of Service module is as follows:

- enums
  - ExecuterTaskService
- exceptions
  - enums
  —— BusinessExceptionType
  - BusinessException
- impl
  - asinc media tasks
  —— CreateMediaTask
  —— DeleteMediaTask
  —— UpdateAverageRatingTask
  —— UpdateMediaRedundancyTask
  —— UpdateMediaTask
  —— UpdateNumberofLikesTask
  - asinc review tasks
  —— RemoveDeletedMediaReviewsTask
  —— RemoveDeletedUserReviewsTask
  —— UpdateReviewRedundancyTask
  - asinc user tasks
  —— CreateUserTask
  —— DeleteUserTask
  —— UpdateNumberOfFollowedTask
  —— UpdateNumberOfFollowersTask
  —- UpdateUserTask
  - AperiodicExecutorTaskServiceImpl
  - ErrorTaskManager
  - MediaContentServiceImpl
  - PeriodicExecutorTaskServiceImpl

- ReviewServiceImpl
- UserServiceImpl
- interfaces
—— ExecuterTaskService
—— MediaContentService
—— ReviewService
—— Task
—— TaskManager
—— UserService

- ServiceLocator

Adopted Patterns and Techniques

Patterns

MVC Pattern

The MVC (*Model-View-Controller*) pattern is a prevalent architectural pattern in software development, especially in web applications. It divides the application into three interconnected components:

- **Model:** Represents the core data and business logic of the application. It interfaces with databases and processes data manipulation and business rules.
- **View:** Handles the presentation layer of the application. It's responsible for rendering data to users and transmitting user interactions to the controller.
- **Controller:** Acts as an intermediary between the model and view components. It receives user input, processes requests, and updates the model state accordingly. It then triggers updates to the view to reflect these changes.

In addition to the MVC components, modern applications often integrate several additional modules to enhance organization and maintainability:

- **DAO (Data Access Object):** Handles access to different databases in an organized manner. It abstracts the data access layer from the rest of the application, providing standardized methods for querying and manipulating data.
- **Service:** Manages the application's business logic and acts as an intermediary between the controller and the DAO module. It encapsulates complex business rules and orchestrates multiple DAO accesses to execute operations requested by users.
- **DTO (Data Transfer Object):** Facilitates the transfer of data between layers and modules of the application. DTOs are lightweight objects that carry data between the client and server, or between different layers of the application. They streamline communication and reduce the amount of data transmitted over the network.

Together, these components (Model, View, Controller, DAO, Service, and DTO) form a cohesive architecture that promotes separation of concerns, modularity, maintainability, and testability in software applications. They enable parallel development of different components by separate teams, support diverse views of the same data, and ensure efficient data transfer across the application.

Locator Pattern

The Locator design pattern effectively manages dependencies and interactions among modules in software applications. By allowing clients to request implementations from a locator instead of directly instantiating implementation classes, it decouples operations. This approach enhances flexibility and modularity within the application's architecture, making it well-suited for integration with the MVC Pattern.

Singleton Pattern

The Singleton pattern ensures that a class has a single instance and provides a global point of access to it. It is particularly valuable in scenarios where multiple components need consistent access to a shared resource, such as database connections, thereby preventing redundant object creation and optimizing resource utilization. Moreover, it finds application in classes that manage thread execution, such as *ErrorTaskManager*, *AperiodicExecutorTaskServiceImpl*, and *PeriodicExecutorTaskServiceImpl*. This pattern ensures that these classes maintain singular control over their respective tasks, enhancing efficiency and coordination within the application.

Techniques

Session Management

Session management in the application optimizes user state using the HttpSession interface. Storing crucial user information in session variables reduces the need for frequent database accesses, thereby enhancing performance and user experience.

Task Abstraction

The class Task represents individual asynchronous operations within the application. Each task is defined by its priority and timestamp and implements the abstract method executeJob to initiate task execution. Various Task implementations manage data redundancy across different MongoDB collections and ensure synchronization between MongoDB and Neo4j databases, leveraging Data Access Objects (DAOs) for efficient data management.

Task Manager

The Task Manager defines methods for managing a task queue capable of handling up to 30 tasks. Tasks are added based on priority or timestamp in case of equal priority. The concrete implementation, ErrorTaskManager, specifically manages tasks encountering transient errors during execution.

Execution of tasks is handled by a dedicated thread, which continuously retrieves and processes tasks from the queue. When the queue is empty, the thread waits for new tasks. Tasks encountering RETRYABLE errors during execution are rescheduled in the queue for future retries, ensuring eventual data consistency across the application. Additionally, tasks failing consecutively five times generate a warning log message to prevent indefinite looping.

Executor Service

The ExecutorService, implemented through the Java interface, manages two types of tasks: Runnable for asynchronous operations without immediate result retrieval, and Callable using the Future type for retrieving data in the main thread upon task completion. This concurrency model significantly boosts application efficiency by parallelizing operations.

Within the application, Runnable tasks manage redundancy and ensure eventual consistency across both MongoDB and Neo4j databases, thereby minimizing performance impacts and enhancing application responsiveness. Callable tasks are utilized in controllers to handle multiple accesses for retrieving necessary information to load pages efficiently.

Description of Main Classes

Controller

Class	Description
AuthServlet	Handles requests on authentication page
MainPageServlet	Handles requests on the mainPage
ManagerServlet	Handles business logic for manager
MediaContentServlet	handle requests on the media content page and retrieve or set media content releted info
ProfileServlet	Handles requests on the profile page
UserServlet	Handles requests related to user model

DAO

Class	Sub-package	Description
MediaContentDAO	interfaces	Collection of methods for media content database related entities on mongoDB
ReviewDAO B	interfaces	Collection of methods for review database related entities on mongoDB
UserDAO	interfaces	Collection of methods for user database related entities on mongoDB
AnimeDAOMongoImpl	mongo	Contains all the method implementation for the MongoDB database anime entities
BaseMongoDBDAO	mongo	Contains all the method implementations for the MongoDB database
MangaDAOMongoImpl	mongo	Contains all the method implementations for the MongoDB database manga entities
ReviewDAOMongoImpl	mongo	Contains all the method implementations for the MongoDB database review entities
UserDAOMongoImpl	mongo	Contains all the method implementations for the MongoDB database user entities
AnimeDAONeo4jImpl	neo4j	Contains all the method implementation for the Neo4j database anime entities
BaseNeo4jDAO	neo4j	Contains all the method implementations for the Neo4j database
MangaDAONeo4jImpl	neo4j	Contains all the method implementation for the Neo4j database manga entities
UserDAONeo4jImpl	neo4j	Contains all the method implementation for the Neo4j database user entities
DAOLocator		Implements the locator pattern for accessing DAOs based on the specified data repository

DTO

Class	Sub-package	Description
AnimeDTO	mediaContent	Represents data transfer object containing attributes for animes
MangaDTO	mediaContent	Represents data transfer object containing attributes for mangas
MediaContentDTO	interfaces	Defines common attributes for media content
DashboardDTO	statistics	Contains statistical data for the dashboard
MongoDBStats	statistics	Provides statistics specific to MongoDB
LoggedUserDTO		Holds information about a logged-in user.
PageDTO		Represents pagination details
ReviewDTO		Contains attributes for reviews
UserRegistrationDTO		Holds data for user registration
UserSummaryDTO		Provides a summary of user information

Model

Class	Sub-package	Description
Anime	mediaContent	Provides unique anime attributes by extending parent class MediaContent and related getter and setter methods.
Manga	mediaContent	Provides unique manga attributes by extending parent class MediaContent and related getter and setter methods.
MangaAuthor	mediaContent	Contains manga author attributes and related getter and setter methods.
MediaContent	mediaContent	Contains all the attributes used by types of media contents and their getter and setter methods.
Manager	registeredUser	Provides unique manager attributes by extending parent class RegisteredUser and related getter and setter methods.
RegisteredUSer	registeredUser	Contains all the attributes used by types of registered users and their getter and setter methods.
User	registeredUser	Provides unique user attributes by extending parent class RegisteredUser and related getter and setter methods.
Review		Contains review attributes and related getter and setter methods.

Service

Class	Sub-package	Description
CreateMediaTask	impl/ asinc_media_tasks	Implementation of methods for media task creation for MediaContentService
DeleteMediaTask B	impl/ asinc_media_tasks	Implementation of methods for media task deletion for MediaContentService
RefreshLatestReviewsTasks	impl/ asinc_media_tasks	Implementation of methods for refreshing latest reviews for MediaContentService
UpdateAverageRatingTask	impl/ asinc_media_tasks	Implementation of methods for updating average rating for MediaContentService
UpdateMediaRedundancyTask	impl/ asinc_media_tasks	Implementation of methods for updating media redundancy for MediaContentService
UpdateMediaTask	impl/ asinc_media_tasks	Implementation of methods for updating media for MediaContentService
UpdateNumberofLikesTask	impl/ asinc_media_tasks	Implementation of methods for updating numbers of likes for MediaContentService
RemoveDeletedMedia ReviewsTask	impl/ asinc_review_tasks	Implementation of methods for removing reviews of deleted media for ReviewService
RemoveDeletedUser ReviewsTask	impl/ asinc_review_tasks	Implementation of methods for removing reviews of deleted user for ReviewService
UpdateReviewRedundancyTask	impl/ asinc_review_tasks	Implementation of methods for updating review redundancy for ReviewService
CreateUserTask	impl/ asinc_user_tasks	Implementation of methods for user creation for UserService
DeleteUserTask	impl/ asinc_user_tasks	Implementation of methods for user deletion for UserService
UpdateNumberOfFollowedTask B	impl/ asinc_user_tasks	Implementation of methods for updating number of followed for UserService
UpdateNumberOfFollowersTask	impl/ asinc_user_tasks	Implementation of methods for updating number of followers for UserService
UpdateUserTask	impl/ asinc_user_tasks	Implementation of methods for updating user for MediaContentService
AperiodicExecutor TaskServiceImpl	impl	Implementation of aperiodic tasks for ExecutorTaskService
ErrorTaskManager	impl	Implementation of TaskManager interface to handle error
MediaContentServiceImpl	impl	Implementation of MediaContentService, providing media content operations
PeriodicExecutor TaskServiceImpl	impl	Implementation of periodic tasks for ExecutorTaskService
ReviewServiceImpl	impl	Implementation of ReviewService, providing review operations
UserServiceImpl	impl	Implementation of UserService, providing user operations

Class	Sub-package	Description
ExecutorTaskService	interfaces	Collection of methods for task management
MediaContentService	interfaces	Collection of methods for media content service
ReviewService	interfaces	Collection of methods for review service
Task	interfaces	Collection of methods for execution operations
TaskManager	interfaces	Collection of methods for managing task prioritization
UserService	interfaces	Collection of methods for user service
ServiceLocator		Implements locator pattern for services

MongoDB queries

Some of the most important MongoDB queries for analytic and suggestion purposes.

USER:

Get Distribution
GetDistribution query to get the user's location, birthday year that gave the highest rating to the application

- Java Implementation:

```
public Map<String, Integer> getDistribution(String criteria) throws DAOException {
    try {
        MongoCollection<Document> usersCollection = getCollection(COLLECTION_NAME);

        List<Bson> pipeline = new ArrayList<>();
        if (criteria.equals("birthday") || criteria.equals("joined_on")) {
            pipeline.addAll(List.of(
                    match(exists(criteria)),
                    project(fields(computed("year", new Document("$year", "$" + criteria)),
    include("app_rating" ))),
                    group("$year", sum("count", 1)),
                    sort(descending("count"))));
        } else if (criteria.equals("location") || criteria.equals("gender")) {
            pipeline.addAll(List.of(
                    match(exists(criteria)),
                    project(fields(include(criteria, "app_rating"))),
                    group("$" + criteria, sum("count", 1)),
                    sort(descending("count"))));
        } else {
            throw new Exception("UserDAOMongoImpl: getDistribution: Invalid criteria");
        }

        List<Document> aggregationResult = usersCollection.aggregate(pipeline).into(new
    ArrayList<>());
        if (aggregationResult.isEmpty()) {
            throw new MongoException("UserDAOMongoImpl: getDistribution: No data found");
        }

        Map<String,Integer> map = new LinkedHashMap<>();
        for (Document doc : aggregationResult) {
            if (criteria.equals("birthday") || criteria.equals("joined_on")) {
                map.put(String.valueOf(doc.getInteger("_id")), doc.getInteger("count"));
            } else {
                map.put(doc.getString("_id"), doc.getInteger("count"));
            }
        }
        return map;

    } catch (MongoException e){
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
```

```
    } catch (Exception e){
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Mongo Shell Query:

```
db.collection.aggregate([
    {
        // Match stage to filter documents where 'criteriaOfSearch' exists
        $match: {
            [criteriaOfSearch]: { $exists: true }
        }
    },
    // Project stage to include 'criteriaOfSearch' and 'app_rating' fields
    {
        $project: {
            [criteriaOfSearch]: 1,
            app_rating: 1
        }
    },
    // Group stage to count occurrences of each 'criteriaOfSearch'
    {
        $group: {
            _id: "$" + criteriaOfSearch,
            count: { $sum: 1 }
        }
    },
    // Sort stage to sort documents by 'count' in descending order
    {
        $sort: {
            count: -1
        }
    }
]);
```

Average App Rating

Calculates the average application rating based on the specified search criteria

- Java Implementation:

```
public Map<String, Double> averageAppRating(String criteria) throws DAOException {
try {
    MongoCollection<Document> usersCollection = getCollection(COLLECTION_NAME);

    List<Bson> pipeline = List.of(
            match(and(exists(criteria), exists("app_rating"))),
            group("$" + criteria, avg("averageAppRating", "$app_rating")),
            sort(descending("averageAppRating"))
    );

    List<Document> aggregationResult = usersCollection.aggregate(pipeline).into(new
ArrayList<>());
    if (aggregationResult.isEmpty()) {
        throw new MongoException("UserDAOMongoImpl: averageAppRating: No data found");
    }

    Map<String,Double> map = new LinkedHashMap<>();
    for (Document doc : aggregationResult) {
        map.put(doc.getString("_id"), doc.getDouble("averageAppRating"));
    }
    return map;

} catch (MongoException e){
    throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
}
catch (Exception e){
    throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
}
}
```

- Mongo Shell Query:

```
db.getCollection.aggregate([
    {
        // Match stage: Filters documents to include only those that
        // have both the specified 'criteria' field and 'app_rating' field.
        $match: {
                $and: [
                { [criteria]: { $exists: true } },
                { app_rating: { $exists: true } }
            ]
        }
    },
    {
        // Group stage: Groups the filtered documents by the 'criteria' field.
        // Calculates the average value of 'app_rating' for each group.
        $group: {
            _id: "$" + criteria,
            averageAppRating: { $avg: "$app_rating" }
        }
    },
    {
        // Sort stage: Sorts the groups in descending order by 'averageAppRating'.
        $sort: {
            averageAppRating: -1
        }
    }
]).toArray();
```

Average App Rating By Age

Calculates the average app rating for users grouped by age ranges. The age ranges are defined as follows:
- 0-13 years
- 13-20 years
- 20-30 years
- 30-40 years
- 40-50 years
- 50+ years

- Java Implementation:

```
public Map<String, Double> averageAppRatingByAgeRange() throws DAOException {
    try {
        MongoCollection<Document> usersCollection = getCollection(COLLECTION_NAME);

        // Define the boundaries for the age ranges and the output fields
        List<Long> boundaries = Arrays.asList(0L, 13L, 20L, 30L, 40L, 50L);
        BsonField[] outputFields = {
                new BsonField("avg_app_rating", new Document("$avg", "$app_rating"))
        };
        BucketOptions options = new BucketOptions()
                .defaultBucket(50L)
                .output(outputFields);

        List<Bson> pipeline = List.of(
                match(and(exists("birthday"), exists("app_rating"))),
                project(fields(
                        computed("age", new Document("$floor", new Document("$divide",
                        Arrays.asList(
                                new Document("$subtract", Arrays.asList(new Date(), "$birthday
    ")),
                                1000L * 60 * 60 * 24 * 365
                        )))),
                        include("app_rating")
                )),
                bucket("$age", boundaries, options)
        );

        List<Document> aggregationResult = usersCollection.aggregate(pipeline).into(new
    ArrayList<>());

        if (aggregationResult.isEmpty()) {
            throw new MongoException("UserDAOMongoImpl: averageAppRatingByAgeRange: No data
    found");
        }
```

```
32
33          Map<String, Double> map = new LinkedHashMap<>();
34          for (Document doc : aggregationResult) {
35              String ageRange = convertIntegerToAgeRange(doc.getLong("_id"));
36              map.put(ageRange, doc.getDouble("avg_app_rating"));
37          }
38
39          return map;
40
41      } catch (MongoException e){
42          throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
43      } catch (Exception e){
44          throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
45      }
46  }
```

- Mongo Shell Query:

```
1  db.getCollection('COLLECTION_NAME').aggregate([
2      {
3          // Match stage: Filters documents to include only those that
4          // have both the 'birthday' field and 'app_rating' field.
5          $match: {
6              $and: [
7                  { birthday: { $exists: true } },
8                  { app_rating: { $exists: true } }
9              ]
10         }
11     },
12     {
13         // Project stage: Adds a new field 'age' calculated by subtracting
14         // the 'birthday' from the current date, converting the difference
15         // from milliseconds to years, and taking the floor of the result.
16         // Also includes the 'app_rating' field.
17         $project: {
18             age: {
19                 $floor: {
20                     $divide: [
21                         { $subtract: [ new Date(), "$birthday" ] },
22                         1000 * 60 * 60 * 24 * 365
23                     ]
24                 }
25             },
26             app_rating: 1
27         }
28     },
29     {
30         // Bucket stage: Groups the documents into buckets based on the 'age' field.
31         // Specifies boundaries for the buckets and assigns documents with an age
32         // outside these boundaries to the default bucket (50+ years).
33         // For each bucket, calculates the average value of 'app_rating'.
34         $bucket: {
35             groupBy: "$age",
36             boundaries: [0, 13, 20, 30, 40, 50],
37             default: 50,
38             output: {
39                 avg_app_rating: { $avg: "$app_rating" }
40             }
41         }
42     }
43 ]).toArray();
```

REVIEW:

Get Media Content Rating By Year

Retrieves the average ratings for a specific media content (anime or manga) by year within a specified range. The aggregation pipeline performs the following steps:

1. Matches the reviews for the specified media content ID and date range, ensuring the reviews have a rating.
2. Groups the reviews by year and calculates the average rating for each year.

3. Projects the results to include the year and the calculated average rating.
4. Sorts the results by year in ascending order.

- Java Implementation:

```java
public Map<String, Double> getMediaContentRatingByYear(MediaContentType type, String
    mediaContentId, int startYear, int endYear) throws  DAOException {
    try  {
        // Get media content rating by year
        MongoCollection<Document> reviewCollection = getCollection(COLLECTION_NAME);

        String nodeType = type.equals(MediaContentType.ANIME) ? "anime" : "manga";
        Date startDate = ConverterUtils.localDateToDate(LocalDate.of(startYear, 1, 1));
        Date endDate = ConverterUtils.localDateToDate(LocalDate.of(endYear + 1, 1, 1));
        List<Bson> pipeline = List.of(
                match(and(
                        eq(nodeType + ".id", new ObjectId(mediaContentId)),
                        exists("rating", true),
                        gte("date", startDate),
                        lt("date", endDate)
                )),
                group(new Document("$year", "$date"), avg("average_rating", "$rating")),
                project(fields(
                        excludeId(),
                        computed("year", "$_id"),
                        include("average_rating"))
                ),
                sort(ascending("year"))
        );
        List<Document> result = reviewCollection.aggregate(pipeline).into(new ArrayList<>());

        // Initialize the result map with years and default values
        Map<String, Double> resultMap = new LinkedHashMap<>();
        for (int year = startYear; year <= endYear; year++) {
            resultMap.put(String.valueOf(year), null);
        }

        // Populate the result map with the average ratings
        for (Document document : result) {
            Double averageRating = document.getDouble("average_rating");
            Integer year = document.getInteger("year");
            resultMap.put(String.valueOf(year), averageRating);
        }
        return resultMap;

    } catch (MongoException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Mongo Shell Query:

```javascript
// Match stage to filter documents based on specified conditions
db.collection.aggregate([
    {
        // Filters documents to include only those where:
        // 1. The nested 'id' field under 'nodeType' matches the specified 'mediaContentId'.
        // 2. The 'rating' field exists.
        // 3. The 'date' field is within the specified date range (startDate to endDate).
        $match: {
            [`${nodeType}.id`]: new ObjectId(mediaContentId),
            rating: { $exists: true },
            date: { $gte: startDate, $lt: endDate }
        }
    },
    {
        // Groups the filtered documents by year extracted from the 'date' field.
        // Calculates the average value of 'rating' for each year.
        $group: {
            _id: { $year: "$date" },
            average_rating: { $avg: "$rating" }
        }
```

```
    },
    {
        // Projects the result to include the 'year' and 'average_rating' fields,
        // excluding the '_id' field.
        $project: {
            _id: 0,
            year: "$_id",
            average_rating: 1
        }
    },
    {
        // Sort stage to sort documents by year in ascending order
        $sort: { year: 1 }
    }
]);
```

Get Media Content Rating By Month

Retrieves the average ratings for a specific media content (anime or manga) by month for a specified year. The aggregation pipeline performs the following steps:

1. Matches the reviews for the specified media content ID and year, ensuring the reviews have a rating.
2. Groups the reviews by month and calculates the average rating for each month.
3. Projects the results to include the month and the calculated average rating.
4. Sorts the results by month in ascending order.

- Java Implementation:

```
public Map<String, Double> getMediaContentRatingByMonth(MediaContentType type, String
    mediaContentId, int year) throws DAOException {
    try  {
        // Get media content rating by month
        MongoCollection<Document> reviewCollection = getCollection(COLLECTION_NAME);

        String nodeType = type.equals(MediaContentType.ANIME) ? "anime" : "manga";
        Date startDate = ConverterUtils.localDateToDate(LocalDate.of(year, 1, 1));
        Date endDate = ConverterUtils.localDateToDate(LocalDate.of(year + 1, 1, 1));
        List<Bson> pipeline = List.of(
                match(and(
                        eq(nodeType + ".id", new ObjectId(mediaContentId)),
                        exists("rating", true),
                        gte("date", startDate),
                        lt("date", endDate)
                )),
                group(new Document("$month", "$date"),
                        avg("average_rating", "$rating")
                ),
                project(fields(
                        excludeId(),
                        computed("month", "$_id"),
                        include("average_rating")
                )),
                sort(ascending("month"))
        );
        List<Document> result = reviewCollection.aggregate(pipeline).into(new ArrayList<>());

        // Initialize the result map with months and default values
        Map<String, Double> resultMap = new LinkedHashMap<>();
        for (Month month : Month.values()) {
            resultMap.put(month.getDisplayName(TextStyle.FULL, Locale.ENGLISH), null);
        }

        // Populate the result map with the average ratings
        for (Document document : result) {
            Object ratingObj = document.get("average_rating");
            Double averageRating = ratingObj instanceof Integer ratingInt ? ratingInt.
    doubleValue() : (Double) ratingObj;
            Integer month = document.getInteger("month");
            resultMap.put(Month.of(month).getDisplayName(TextStyle.FULL, Locale.ENGLISH),
    averageRating);
        }
        return resultMap;

    } catch (MongoException e) {
```

```
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Mongo Shell Query:

```
db.getCollection.aggregate([
    {
        // Match stage: Filters documents to include only those that meet the specified conditions.
        $match: {
            $and: [
                // Includes documents where the nested 'id' field under 'nodeType' matches 'mediaContentId'.
                { [nodeType + ".id"]: mediaContentId },
                // Includes documents where the 'rating' field exists.
                { rating: { $exists: true } },
                // Includes documents where the 'date' field is greater than or equal to 'startDate'.
                { date: { $gte: startDate } },
                // Includes documents where the 'date' field is less than 'endDate'.
                { date: { $lt: endDate } }
            ]
        }
    },
    {
        // Group stage: Groups the filtered documents by the month extracted from the 'date' field.
        // Calculates the average value of 'rating' for each month.
        $group: {
            _id: { $month: "$date" },
            average_rating: { $avg: "$rating" }
        }
    },
    {
        // Project stage: Shapes the output documents to include 'month' and 'average_rating' fields.
        // Excludes the '_id' field.
        $project: {
            _id: 0,
            month: "$_id",
            average_rating: 1
        }
    },
    {
        // Sort stage: Sorts the documents by 'month' in ascending order.
        $sort: {
            month: 1
        }
    }
]).toArray();
```

Suggest Media Content

Suggests media content (anime or manga) based on user criteria (location or birthday year). The aggregation pipeline performs the following steps:

1. Matches the reviews with a rating, the specified media content type and the user criteria.
2. Groups the reviews by media content ID and calculates the average rating for each media content.
3. Projects the results to include the media content title and the calculated average rating.
4. Sorts the results by average rating in descending order.
5. Limits the results to 20 entries.

- Java Implementation:

```
public List<MediaContentDTO> suggestMediaContent(MediaContentType mediaContentType, String criteriaType, String criteriaValue) throws DAOException {
    try  {
        // Suggest media content based on user criteria
        MongoCollection<Document> reviewCollection = getCollection(COLLECTION_NAME);
        String nodeType = mediaContentType.equals(MediaContentType.ANIME) ? "anime" : "manga";

```

```
        Bson filter = and(
                exists("rating", true),
                exists(nodeType, true)
        );

        if (criteriaType.equals("location")) {
            filter = and(filter, eq("user.location", criteriaValue));
        } else if (criteriaType.equals("birthday")) {
            Date startDate = ConverterUtils.localDateToDate(LocalDate.of(Integer.parseInt(
    criteriaValue), 1, 1));
            Date endDate = ConverterUtils.localDateToDate(LocalDate.of(Integer.parseInt(
    criteriaValue) + 1, 1, 1));
            filter = and(filter, gte("user.birthday", startDate), lt("user.birthday", endDate)
    );
        } else {
            throw new Exception("ReviewDAOMongoImpl: suggestMediaContent: Invalid criteria
    type");
        }

        List<Bson> pipeline = new ArrayList<>(List.of(
                match(filter),
                group("$" + nodeType + ".id",
                        first("title", "$" + nodeType + ".title"),
                        avg("average_rating", "$rating")),
                sort(descending("average_rating")),
                project(include("title")),
                limit(20)));

        List<Document> result = reviewCollection.aggregate(pipeline).into(new ArrayList<>());
        if (result.isEmpty()) {
            throw new MongoException("ReviewDAOMongoImpl: suggestMediaContent: No reviews
    found");
        }

        List<MediaContentDTO> entries = new ArrayList<>();
        for (Document document : result) {
            String contentId = String.valueOf(document.getObjectId("_id"));
            String title = document.getString("title");

            MediaContentDTO mediaContentDTO;
            if (nodeType.equals("anime")) {
                mediaContentDTO = new AnimeDTO(contentId, title);
            } else {
                mediaContentDTO = new MangaDTO(contentId, title);
            }
            entries.add(mediaContentDTO);
        }
        return entries;

    } catch (MongoException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Mongo Shell Query:

```
db.collection.aggregate([
    {
        // Match documents based on a dynamic user criteria
        // It dynamically matches documents where a field in the 'user' object
        // (specified by 'criteriaType') has the value 'criteriaValue'.
        $match: {
        ["user." + criteriaType]: criteriaValue
        }
    },
    {
        // Group stage: Groups documents by the node type's ID.
        // For each group, it retrieves the first title and calculates the average rating.
        $group: {
        _id: "$" + nodeType + ".id", // Group by the node type's ID
        title: { $first: "$" + nodeType + ".title" }, // Get the first title in the group
```

```
        average_rating: { $avg: "$rating" } // Calculate the average rating for the group
        }
    },
    {
        // Sort stage: Sorts the grouped documents by average rating in descending order.
        $sort: { average_rating: -1 }
    },
    {
        // Limit stage: Limits the number of results to a constant defined by 'Constants.
    PAGE_SIZE'.
        $limit: Constants.PAGE_SIZE
    }
    ]);
```

MANGA/ANIME:

Get Best Criteria

Retrieves the best criteria based on the average rating of the Anime objects in the MongoDB database.

- Java Implementation:

```
public Map<String, Double> getBestCriteria (String criteria, boolean isArray, int page) throws
     DAOException {
    try {
        MongoCollection<Document> animeCollection = getCollection(COLLECTION_NAME);
        int pageOffset = (page - 1) * Constants.PAGE_SIZE;

        List<Bson> pipeline;
        if (isArray) {
            pipeline = List.of(
                    match(and(exists(criteria), ne("average_rating", null))),
                    unwind("$" + criteria),
                    group("$" + criteria, avg("criteria_average_rating", "$average_rating")),
                    sort(descending("criteria_average_rating")),
                    skip(pageOffset),
                    limit(25)
            );
        } else {
            pipeline = List.of(
                    match(Filters.exists(criteria)),
                    group("$" + criteria, avg("criteria_average_rating", "$average_rating")),
                    sort(new Document("criteria_average_rating", -1)),
                    skip(pageOffset),
                    limit(25)
            );
        }

        List <Document> document = animeCollection.aggregate(pipeline).into(new ArrayList<>())
    ;
        Map<String, Double> bestCriteria = new LinkedHashMap<>();
        for (Document doc : document) {
            Double avgRating = doc.get("criteria_average_rating") instanceof Integer?
                    doc.getInteger("criteria_average_rating").doubleValue() :
                    doc.getDouble("criteria_average_rating");
            bestCriteria.put(doc.get("_id").toString(), avgRating);
        }

        return bestCriteria;

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Mongo Shell Query:

```
db.collection.aggregate([
    // Match stage to filter documents where 'criteria' exists and 'average_rating' is not
    null
    {
        $match: {
            criteria: { $exists: true },
```

```
            average_rating: { $ne: null }
        }
    },
    // Unwind stage to deconstruct the 'criteria' array field
    {
        $unwind: "$" + criteria
    },
    // Group stage to calculate the average rating for each criteria
    {
        $group: {
            // Group by each individual 'criteria' element
            _id: "$" + criteria,
            // Calculate the average rating for each 'criteria'
            criteria_average_rating: { $avg: "$average_rating" }
        }
    },
    // Sort stage to sort documents by 'criteria_average_rating' in descending order
    {
        $sort: {
            criteria_average_rating: -1
        }
    },
    // Skip stage to skip the first 'pageOffset' documents
    {
        $skip: pageOffset
    },
    // Limit stage to limit the results to 25 documents
    {
        $limit: 25
    }
]);
```

GraphDB queries

Some of the most important Neo4j queries for analytic and suggestion purposes.

USERS:

Suggest User By Common Likes
Retrieves a list of suggested users for a specific user based on common likes from the Neo4j database. The method performs the following steps:
1. Retrieve users who like the same media content as the specified user in the last 6 month.
2. Retrieve users who like the same media content as the specified user in the last year.
3. Retrieve users who like the same media content as the specified user.

- Java Implementation:

```
public List<UserSummaryDTO> suggestUsersByCommonLikes(String userId, Integer limit,
    MediaContentType type) throws DAOException {

    try (Session session = getSession()) {
        if (type == null) {
            throw new IllegalArgumentException("Media content type must be specified");
        }

        int n = limit == null ? 5 : limit;
        int remaining;

        StringBuilder queryBuilder = new StringBuilder();
        if (type == MediaContentType.ANIME)
            queryBuilder.append("MATCH (u:User {id: $userId})-[r:LIKE]->(media:Anime)<-[:LIKE
    ]-(suggested:User) ");
        else
            queryBuilder.append("MATCH (u:User {id: $userId})-[r:LIKE]->(media:Manga)<-[:LIKE
    ]-(suggested:User) ");
        queryBuilder.append("""
                WHERE u <> suggested AND r.date >= date($date)
                WITH suggested, COUNT(DISTINCT media) AS commonLikes
                WHERE commonLikes > $min
                RETURN suggested AS user, commonLikes
                ORDER BY commonLikes DESC
                LIMIT $n
                """);
        String query1 = queryBuilder.toString();
        Value params1 = parameters("userId", userId, "n", n, "date", LocalDate.now().
    minusMonths(6), "min", 5);

        List<UserSummaryDTO> suggested = session.executeRead(
                tx -> tx.run(query1, params1).list()
        ).stream()
                .map(this::recordToUserSummaryDTO)
                .collect(Collectors.toList());

        remaining = n - suggested.size();

        if (remaining > 0) {
            Value params2 = parameters("userId", userId, "n", n, "date", LocalDate.now().
    minusYears(1), "min", 5);

            List<Record> records = session.executeRead(tx -> tx.run(query1, params2).list());
            for (Record record : records) {
                UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
                if (!suggested.contains(userDTO))
                    suggested.add(userDTO);
                if (suggested.size() == n)
                    break;
            }

            remaining = n - suggested.size();
        }

        if(remaining > 0) {
```

```
            StringBuilder queryBuilder3 = new StringBuilder();
            if (type == MediaContentType.ANIME)
                queryBuilder3.append("MATCH (u:User {id: $userId})-[r:LIKE]->(media:Anime)<-[:LIKE]-(suggested:User) ");
            else
                queryBuilder3.append("MATCH (u:User {id: $userId})-[r:LIKE]->(media:Manga)<-[:LIKE]-(suggested:User) ");
            queryBuilder3.append("""
                    WHERE u <> suggested
                    WITH suggested, COUNT(DISTINCT media) AS commonLikes
                    RETURN suggested AS user, commonLikes
                    ORDER BY commonLikes DESC
                    LIMIT $n
                    """);
            String query2 = queryBuilder3.toString();
            Value params3 = parameters("userId", userId, "n", n);

            List<Record> records = session.executeRead(tx -> tx.run(query2, params3).list());
            for (Record record : records) {
                UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
                if (!suggested.contains(userDTO))
                    suggested.add(userDTO);
                if (suggested.size() == n)
                    break;
            }
        }

        return suggested.isEmpty() ? null : suggested;

    } catch (Neo4jException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Neo4j Query:

```
// Match the user with the given userId who has liked media of type Manga
MATCH (u:User {id: $userId})-[r:LIKE]->(media:Manga)<-[:LIKE]-(suggested:User)
// Filter out the original user from the suggested users and only consider likes after the specified date
WHERE u <> suggested AND r.date >= $date
// Pass the suggested users and count of distinct media liked by both users to the next stage
WITH suggested, COUNT(DISTINCT media) AS commonLikes
// Filter out users with common likes less than or equal to the minimum threshold
WHERE commonLikes > $min
// Return the suggested users and the count of common likes
RETURN suggested AS user, commonLikes
// Order the results by the count of common likes in descending order
ORDER BY commonLikes DESC
// Limit the number of results to the specified maximum
LIMIT $n
```

Suggest Users By Common Followings

Retrieves a list of suggested users for a specific user based on common followings from the Neo4j database. The method performs the following steps:

1. Retrieve users that follow user's followings and have more than 5 common followings.
2. Retrieve users that are followed by user's followings and have more than 5 connections.
3. Retrieve users that follow user's followings.

- Java Implementation:

```
public List<UserSummaryDTO> suggestUsersByCommonFollowings(String userId, Integer limit) throws DAOException {
    try (Session session = getSession()) {
        int n = limit == null ? 5 : limit;
        int remaining;

        // suggest users that follow user's followings and have more than 5 common followings
```

```
        String query = """
                MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)<-[:FOLLOWS]-(suggested:User)
                WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
                WITH suggested, COUNT(DISTINCT following) AS commonFollowings
                WHERE commonFollowings > 5
                RETURN suggested as user
                ORDER BY commonFollowings DESC
                LIMIT $n
                """;
        Value params = parameters("userId", userId, "n", n);

        List<UserSummaryDTO> suggested = session.executeRead(
                tx -> tx.run(query, params).list()
        ).stream()
                .map(this::recordToUserSummaryDTO)
                .collect(Collectors.toList());

        remaining = n - suggested.size();

        // if there are not enough suggestions, suggest users that are followed by the user's followings and have more than 5 connections
        if (remaining > 0) {
            String query2 = """
                    MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)-[:FOLLOWS]->(suggested:User)
                    WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
                    WITH suggested, COUNT(DISTINCT following) AS commonUsers
                    WHERE commonUsers > 5
                    RETURN suggested as user
                    ORDER BY commonUsers DESC
                    LIMIT $n
                    """;
            Value params2 = parameters("userId", userId, "n", n);

            List<Record> records = session.executeRead(tx -> tx.run(query2, params2).list());
            for (Record record : records) {
                UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
                if (!suggested.contains(userDTO))
                    suggested.add(userDTO);
                if (suggested.size() == n)
                    break;
            }

            remaining = n - suggested.size();
        }

        // if there are still not enough suggestions, suggest users that follow the user's followings
        if (remaining > 0) {
            String query3 = """
                    MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)<-[:FOLLOWS]-(suggested:User)
                    WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
                    WITH suggested, COUNT(DISTINCT following) AS commonFollowings
                    RETURN suggested as user
                    ORDER BY commonFollowings DESC
                    LIMIT $n
                    """;
            Value params3 = parameters("userId", userId, "n", n);

            List<Record> records = session.executeRead(tx -> tx.run(query3, params3).list());
            for (Record record : records) {
                UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
                if (!suggested.contains(userDTO))
                    suggested.add(userDTO);
                if (suggested.size() == n)
                    break;
            }
        }

        return suggested.isEmpty() ? null : suggested;

```

```
    } catch (Neo4jException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Neo4j Query:

```
// Match the user with the given userId who follows other users
MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)<-[:FOLLOWS]-(suggested:User)
// Ensure that the suggested user is not already followed by the original user and they are not the same user
WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
// Calculate the count of distinct users that both the original user and suggested user follow
WITH suggested, COUNT(DISTINCT following) AS commonFollowers
// Filter out suggested users who have less than or equal to 5 common followers
WHERE commonFollowers > 5
// Return the suggested users along with the count of common followers
RETURN suggested as user, commonFollowers
// Order the results by the count of common followers in descending order
ORDER BY commonFollowers DESC
// Limit the number of results to the specified maximum
LIMIT $n
```

ANIME/MANGA:

Get Trend Media Content By Year

Retrieves a list of trending MangaDTO objects for a specific year from the Neo4j database.

- Java Implementation:

```
public Map<MediaContentDTO, Integer> getTrendMediaContentByYear(int year, Integer limit) throws DAOException {
    int n = limit == null ? 5 : limit;
    try (Session session = getSession()) {
        LocalDate startDate = LocalDate.of(year, 1, 1);
        LocalDate endDate = LocalDate.of(year + 1, 1, 1);

        String query = """
        MATCH (m:Manga)<-[r:LIKE]-(u:User)
        WHERE r.date >= date($startDate) AND r.date < date($endDate)
        WITH m, count(r) AS numLikes
        ORDER BY numLikes DESC
        RETURN m AS manga, numLikes
        LIMIT $n
        """;

        Value params = parameters("startDate", startDate, "endDate", endDate, "n", n);

        Map<MediaContentDTO, Integer> result = new LinkedHashMap<>();
        session.executeRead(
                tx -> tx.run(query, params).list()
        ).forEach(record -> {
            MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
            Integer likes = record.get("numLikes").asInt();
            result.put(mangaDTO, likes);
        });

        return result;

    } catch (Neo4jException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Neo4j Query:

```
// Match Anime nodes that are liked by Users, with a LIKE relationship and a date constraint
MATCH (a:Anime)<-[r:LIKE]-(u:User)
WHERE r.date >= $startDate AND r.date < $endDate
// Aggregate the results to count the number of likes for each Anime node
WITH a, count(r) AS numLikes
// Order the Anime nodes by the number of likes in descending order
ORDER BY numLikes DESC
// Return the Anime node and the number of likes, limiting the results to the specified
    maximum
RETURN a AS anime, numLikes
LIMIT $n
```

Get Media Content Trend By Likes

Retrieves a list of trending MangaDTO objects by likes from the Neo4j database. The method performs the following steps:

1. Retrieve the trending Manga by likes in the last 6 months.
2. If there are not enough trending Manga, retrieve more results from the last year.
3. If there are still not enough trending Manga, retrieve more results from the last 5 years.

- Java Implementation:

```
public List<MediaContentDTO> getMediaContentTrendByLikes(Integer limit) throws DAOException {
    try (Session session = getSession()) {
        int n = limit == null ? 5 : limit;
        int remaining;
        LocalDate now = LocalDate.now();

        // Try to get trending content based on likes in the last 6 months
        String query1 = """
            MATCH (u:User)-[r:LIKE]->(m:Manga)
            WHERE r.date >= date($startDate)
            WITH m, COUNT(r) AS numLikes
            WHERE numLikes > 10
            RETURN m AS manga, numLikes
            ORDER BY numLikes DESC, m.title ASC
            LIMIT $n
            """;

        Value params1 = parameters("startDate", now.minusMonths(6), "n", n);
        List<MediaContentDTO> trendingContent = session.executeRead(
                        tx -> tx.run(query1, params1).list()
                ).stream()
                .map(record -> (MangaDTO) recordToMediaContentDTO(record))
                .collect(Collectors.toList());

        remaining = n - trendingContent.size();

        // If not enough results, add more results from the last year
        if (remaining > 0) {
            Value params2 = parameters("startDate", now.minusYears(1), "n", remaining);

            List<Record> records = session.executeRead(tx -> tx.run(query1, params2).list());
            for (Record record : records) {
                MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
                if (!trendingContent.contains(mangaDTO))
                    trendingContent.add(mangaDTO);
                if (trendingContent.size() == n)
                    break;
            }

            remaining = n - trendingContent.size();
        }

        // If still not enough results, add more results from the last 5 years
        if (remaining > 0) {
            String query2 = """
            MATCH (u:User)-[r:LIKE]->(m:Manga)
            WHERE r.date >= date($startDate)
            WITH m, COUNT(r) AS numLikes
            RETURN m AS manga, numLikes
            ORDER BY numLikes DESC, m.title ASC
```

```
            LIMIT $n
            """;
            Value params3 = parameters("startDate", now.minusYears(5), "n", remaining);

            List<Record> records = session.executeRead(tx -> tx.run(query2, params3).list());
            for (Record record : records) {
                MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
                if (!trendingContent.contains(mangaDTO))
                    trendingContent.add(mangaDTO);
                if (trendingContent.size() == n)
                    break;
            }
        }

        return trendingContent.isEmpty() ? null : trendingContent;

    } catch (Neo4jException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Neo4j Query:

```
// Match User nodes who have liked Anime nodes with a LIKE relationship and date constraint
MATCH (u:User)-[r:LIKE]->(a:Anime)
WHERE r.date >= $startDate
// Aggregate the results to count the number of LIKE relationships for each Anime node
WITH a, COUNT(r) AS numLikes
// Order the Anime nodes by the number of likes in descending order
ORDER BY numLikes DESC
// Return the Anime node and the number of likes, limiting the results to the specified maximum
RETURN a AS anime, numLikes
LIMIT $n
```

Get Suggested By Followings

Retrieves a list of suggested MangaDTO objects for a user from the Neo4j database. The method performs the following steps:

1. Retrieve Manga that the user's followings have liked in the last 6 months.
2. If there are not enough suggestions, retrieve Manga that the user's followings have liked in the last 2 years.
3. If there are still not enough suggestions, retrieve Manga that the user's followings have liked.

- Java Implementation:

```
public List<MediaContentDTO> getSuggestedByFollowings(String userId, Integer limit) throws DAOException {
    try (Session session = getSession()) {
        int n = limit == null ? 5 : limit;
        int remaining;
        LocalDate now = LocalDate.now();

        // try to get suggestions based on likes in the last 6 months
        String query1 = """
            MATCH (u:User {id: $userId})-[:FOLLOWS]->(f:User)-[r:LIKE]->(m:Manga)
            WHERE NOT (u)-[:LIKE]->(m) AND r.date >= date($startDate)
            WITH m, COUNT(DISTINCT f) AS num_likes
            RETURN m AS manga
            ORDER BY num_likes DESC, m.title ASC
            LIMIT $n
            """;
        Value params1 = parameters("userId", userId, "n", n, "startDate", now.minusMonths(6));

        List<MediaContentDTO> suggested = session.executeRead(
                        tx -> tx.run(query1, params1).list()
                ).stream()
                .map(record -> (MangaDTO) recordToMediaContentDTO(record))
                .collect(Collectors.toList());

```

```
        remaining = n - suggested.size();

        // if there are not enough suggestions, add more results from the last 2 years
        if (remaining > 0) {
            Value params2 = parameters("userId", userId, "n", n, "startDate", now.minusYears
    (2));

            List<Record> records = session.executeRead(tx -> tx.run(query1, params2).list());
            for (Record record : records) {
                MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
                if (!suggested.contains(mangaDTO))
                    suggested.add(mangaDTO);
                if (suggested.size() == n)
                    break;
            }

            remaining = n - suggested.size();
        }

        // if there are still not enough suggestions, add more results based on all likes
        if (remaining > 0) {
            String query2 = """
                MATCH (u:User {id: $userId})-[:FOLLOWS]->(f:User)-[r:LIKE]->(m:Manga)
                WHERE NOT (u)-[:LIKE]->(m)
                WITH m, COUNT(DISTINCT f) AS num_likes
                RETURN m AS manga
                ORDER BY num_likes DESC, m.title ASC
                LIMIT $n
                """;
            Value params3 = parameters("userId", userId, "n", n);

            List<Record> records = session.executeRead(tx -> tx.run(query2, params3).list());
            for (Record record : records) {
                MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
                if (!suggested.contains(mangaDTO))
                    suggested.add(mangaDTO);
                if (suggested.size() == n)
                    break;
            }
        }

        return suggested.isEmpty() ? null : suggested;

    } catch (Neo4jException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Neo4j Query:

```
// Match the User node with the specified userId who follows other User nodes,
// and those followed Users who have liked Anime nodes with a LIKE relationship and date
    constraint.
MATCH (u:User {id: $userId})-[:FOLLOWS]->(f:User)-[r:LIKE]->(a:Anime)
// Ensure that the User does not directly like the Anime and that the LIKE relationship's date
    is within the specified range.
WHERE NOT (u)-[:LIKE]->(a) AND r.date >= $startDate
// Aggregate the results to count the number of distinct Users who have liked each Anime node.
WITH a, COUNT(DISTINCT f) AS num_likes
// Return the Anime node, ordered by the number of likes in descending order.
RETURN a AS anime
ORDER BY num_likes DESC
// Limit the number of results returned to the specified maximum.
LIMIT $n
```

Get Suggested By Likes

Retrieves a list of suggested MangaDTO objects for a user from the Neo4j database. The method performs the following steps:

1. Retrieve Manga that other users with similar taste have liked in the last 6 months.

2. If there are not enough suggestions, retrieve Manga that other users with similar taste have liked in the last 2 years.
3. If there are still not enough suggestions, retrieve Manga that other users with similar taste have liked.

- Java Implementation:

```
public List<MediaContentDTO> getSuggestedByLikes(String userId, Integer limit) throws DAOException {
    try (Session session = getSession()) {
        int n = limit == null ? 5 : limit;
        int remaining;
        LocalDate today = LocalDate.now();

        // Try to get suggestions based on likes in the last 6 months
        String query1 = """
                MATCH (u:User {id: $userId})-[r1:LIKE]->(m:Manga)<-[:LIKE]-(f:User)
                WHERE r1.date >= $startDate
                WITH u, f, COUNT(m) AS common_likes
                ORDER BY common_likes DESC
                LIMIT 20
                MATCH (f)-[:LIKE]->(m2:Manga)
                WHERE NOT (u)-[:LIKE]->(m2)
                WITH m2, COUNT(DISTINCT f) AS num_likes
                RETURN m2 AS manga
                ORDER BY num_likes DESC, m2.title ASC
                LIMIT $n
                """;
        Value params1 = parameters("userId", userId, "n", n, "startDate", today.minusMonths(6));

        List<MediaContentDTO> suggested = session.executeRead(
                        tx -> tx.run(query1, params1).list()
                ).stream()
                .map(record -> (MangaDTO) recordToMediaContentDTO(record))
                .collect(Collectors.toList());

        remaining = n - suggested.size();

        // If there are not enough suggestions, add more results from the last 2 years
        if (remaining > 0) {
            Value params2 = parameters("userId", userId, "n", n, "startDate", today.minusYears(2));

            List<Record> records = session.executeRead(tx -> tx.run(query1, params2).list());
            for (Record record : records) {
                MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
                if (!suggested.contains(mangaDTO))
                    suggested.add(mangaDTO);
                if (suggested.size() == n)
                    break;
            }

            remaining = n - suggested.size();
        }

        // If there are not enough suggestions, add more results based on all likes
        if (remaining > 0) {
            String query2 = """
                    MATCH (u:User {id: $userId})-[r1:LIKE]->(m:Manga)<-[:LIKE]-(f:User)
                    WITH u, f, COUNT(m) AS common_likes
                    ORDER BY common_likes DESC
                    MATCH (f)-[:LIKE]->(m2:Manga)
                    WHERE NOT (u)-[:LIKE]->(m2)
                    WITH m2, COUNT(DISTINCT f) AS num_likes
                    RETURN m2 AS manga
                    ORDER BY num_likes DESC, m2.title ASC
                    LIMIT $n
                    """;
            Value params3 = parameters("userId", userId, "n", n);

            List<Record> records = session.executeRead(tx -> tx.run(query2, params3).list());
            for (Record record : records) {
```

```
                MangaDTO mangaDTO = (MangaDTO) recordToMediaContentDTO(record);
                if (!suggested.contains(mangaDTO))
                    suggested.add(mangaDTO);
                if (suggested.size() == n)
                    break;
            }
        }

        return suggested.isEmpty() ? null : suggested;

    } catch (Neo4jException e) {
        throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());

    } catch (Exception e) {
        throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
    }
}
```

- Neo4j Query:

```
// Match the User node with the specified userId who likes Anime nodes (r1) and those Anime nodes are liked by other Users (f).
MATCH (u:User {id: $userId})-[r1:LIKE]->(a:Anime)<-[:LIKE]-(f:User)
// Ensure that the User's LIKE relationship's date is within the specified range.
WHERE r1.date >= $startDate
// Aggregate the results to count the number of common Anime nodes liked by both the User (u) and other Users (f).
WITH u, f, COUNT(a) AS common_likes
// Order the results by the number of common likes in descending order and limit to 20 results.
ORDER BY common_likes DESC
LIMIT 20
// Match Users (f) who like Anime nodes (a2) that are not liked by the User (u).
MATCH (f)-[:LIKE]->(a2:Anime)
WHERE NOT (u)-[:LIKE]->(a2)
// Aggregate the results to count the number of distinct Users (f) who like each Anime node (a2).
WITH a2, COUNT(DISTINCT f) AS num_likes
// Return the Anime node (a2) ordered by the number of likes by distinct Users (num_likes) in descending order.
RETURN a2 AS anime
ORDER BY num_likes DESC
// Limit the number of Anime nodes returned to the specified maximum ($n).
LIMIT $n
```

Testing

Testing is a substantial part of the MangaVerse web application project. Testing helps to ensure application's reliability, performance and correctness. To be able to conduct efficient testing process, two kind of tests are preformed. They are JUnit testing as a structural testing and functional testing.

Structural Testing

Structural testing also with other name white-box testing is based on testing the internal structure of the working application and it guarantees that the methods are working as expected.JUnit testing framework is used to conduct structural testing. JUnit testing is performed by testing different modules of the application such as DAOs and services. With that process each methods efficiency and correctness is guaranteed. Some examples of JUnit testing are shown below.

Example code snippet from AnimeDAOMongoImplTest:

```
class AnimeDAOMongoImplTest {

@BeforeEach
public void setUp() throws Exception {
    BaseMongoDBDAO.openConnection();
}

@AfterEach
public void tearDown() throws Exception {
    BaseMongoDBDAO.closeConnection();
}
// test 1 : search for an anime by name
// test 2 : search for an anime by filters
@Test
void searchTest() {
    AnimeDAOMongoImpl animeDAO = new AnimeDAOMongoImpl();

    // test 1
    System.out.println("Search by title");
    assertDoesNotThrow(() -> {
        List<MediaContentDTO> animeList = animeDAO.search(List.of(Pair.of("title", "Attack
     on Titan")), Map.of("title", 1), 1, false).getEntries();
        for (MediaContentDTO anime : animeList) {
            System.out.println("Id: " + anime.getId() + ", Title: " + anime.getTitle());
        }
    });

    // test 2
    System.out.println("Search by filters");
    assertDoesNotThrow(() -> {
        for (int i = 1; i < 5; i++) {
            PageDTO<MediaContentDTO> animePage = animeDAO.search(List.of(Pair.of("$in",Map
    .of("tags", List.of("school clubs", "manwha")))), Map.of("title", 1), i, false);
            if (!animePage.getEntries().isEmpty()) {
                for (MediaContentDTO anime : animePage.getEntries()) {
                    System.out.println("Id: " + anime.getId() + ", Title: " + anime.
    getTitle());
                }
            }

        }
    });
}
```

```
}

```

Example code snippet from Neo4jDAOImplTest:

```
public class Neo4JDAOImplTest{

@BeforeEach
public void setUp() throws Exception {
    BaseMongoDBDAO.openConnection();
    BaseNeo4JDAO.openConnection();
}

@AfterEach
public void tearDown() throws DAOException {
    BaseMongoDBDAO.closeConnection();
    BaseNeo4JDAO.closeConnection();
}
@Test
public void testFollowUser() throws DAOException {
    try {
        UserDAONeo4JImpl neo4JDAO = new UserDAONeo4JImpl();
        neo4JDAO.follow("6577877be68376234760585a", "6577877be683762347605859");
    } catch (DAOException e) {
        fail("Exception not expected: " + e.getMessage());
    }

}

@Test
public void testUnlikeAnime() throws DAOException {
    try {
        AnimeDAONeo4JImpl dao = new AnimeDAONeo4JImpl();
        dao.unlike("6577877be68376234760585f","65789bb52f5d29465d0abd09");
    } catch (DAOException e) {
        fail("Exception not expected: " + e.getMessage());
    }

}
}

```

Example code snippet from ReviewServiceImpl:

```
class ReviewServiceImplTest {
    private static final ExecutorTaskService aperiodicTaskService = ServiceLocator.
    getExecutorTaskService(ExecutorTaskServiceType.APERIODIC);
    private static final TaskManager errorTaskManager = ServiceLocator.
    getErrorsTaskManager();
    @BeforeEach
    public void setUp() throws Exception {
        BaseMongoDBDAO.openConnection();
        BaseNeo4JDAO.openConnection();
        aperiodicTaskService.start();
        errorTaskManager.start();
    }

    @AfterEach
    public void tearDown() throws Exception {
        BaseMongoDBDAO.closeConnection();
        BaseNeo4JDAO.closeConnection();
        aperiodicTaskService.stop();
        errorTaskManager.stop();
    }
    @Test
    void updateReview() {
        ReviewServiceImpl reviewService = new ReviewServiceImpl();
        try {
            ReviewDTO reviewAnime = createSampleAnimeReview();
```

```
            assertDoesNotThrow(() -> reviewService.addReview(reviewAnime));
            reviewAnime.setComment("This is an updated test review");
            reviewAnime.setRating(4);
            assertDoesNotThrow(() -> reviewService.updateReview(reviewAnime));
            System.out.println("Anime review updated: " + reviewAnime);

            ReviewDTO reviewManga = createSampleMangaReview();
            assertDoesNotThrow(() -> reviewService.addReview(reviewManga));
            reviewManga.setComment("This is an updated test review");
            reviewManga.setRating(4);
            assertDoesNotThrow(() -> reviewService.updateReview(reviewManga));
            System.out.println("Manga review updated: " + reviewManga);
        } catch (BusinessException e) {
            fail(e);
        }
    }
}

```

Functional Testing

Functional testing also with other name black-box testing is based on testing the application's external functionalities. It checks the application from end-user's perspective. It ensures that specified requirements are provided efficiently by the web application and expected is outcome is created. With the help of the use cases and real world scenarios, functional testing is conducted. Some examples of functional testing are shown below.

Table 5.1: Functional Test Cases

Id	Description	Input	Expected Output	Output	Outcome
User_01	Login with correct information	email: nmiller@example.com, password: f6d6b3ffecb44a...	The user logs in successfully		
User_02	Login with wrong information	email: wrong@example.com, password: wrong	The user is not able to log in successfully		
User_03	Signup with all the mandatory info are filled				
User_04	Signup with missing info				
User_05	Update user information	description: manga lover	User profile is updated with new info.		
User_06	Follow another user	-	User is followed.		
User_07	Unfollow another user	-	User is unfollowed.		
User_08	Search manga by title	title: "Slam Dunk"	The list of manga which includes the words of "Slam Dunk" is shown.		
User_09	Search manga by detailed filtering				
User_10	Like anime	-	The anime is liked		
User_11	Add review to anime	review:"I like the anime"	The review is added to the anime and displayed in the anime page		
User_12	Update review	review: "I dont like this anime anymore"	The review is updated with the new one.		

Id	Description	Input	Expected Output	Output	Outcome
Admin_01	See users distribution analytics	-			
Admin_02	See manga analytics for get average rating by month	Year:2020	Average rating for each month in 2020 is displayed in the page		
Admin_03	See anime analytics for get trend media content by year				
Admin_04					
Admin_05					

Performance Testing

Performance testing is conducted to ensure that MangaVerse web application is able to handle the a high volume of operations efficiently and provides a smooth experience for users. It is important to test the application's performance to ensure that it can handle the expected load. Performance testing is applied on MongoDB and Neo4j databases. Specifically, the aim of the performance testing is to see the impacts of indexing on CRUD operations and aggregation operations.

MongoDB Performance Testing

Indexes are used in MongoDB to improve the performance of queries. Indexes are used to quickly locate data without having to search every document in a collection. This limiting the search with indexes, results can get with faster response time. For the mongoDB performance testing, *username* is used to indexing Users collection and *title* is used to indexing Anime and Manga collections. The tests are conducted for creating new documents and searching for documents. As it can be shown in the table below, there are significant improvement in operation times with indexing, especially for search tasks

Table 5.2: MongoDB Performance Test Results

Collection	Operation	Index	Time (ms)	Total Keys Examined	Total Docs Examined
Users	Insert	No	3	-	-
	Search	No	17	0	10007
	Insert	Yes	5	-	-
	Search	Yes	5	1	1
Anime	Insert	No	3	-	-
	Search	No	88	0	30113
	Insert	Yes	3	-	-

Continued on next page

Table 5.2 – continued from previous page

Collection	Operation	Index	Time (ms)	Total Keys Examined	Total Docs Examined
	Search	Yes	10	0	1
Manga	Insert	No	3	-	-
	Search	No	141	0	41677
	Insert	Yes	3	-	-
	Search	Yes	10	1	1

Neo4j Performance Testing

Similarly, also for Neo4j database, performance testing is conducted to observe the impacts of indexing on some CRUD operations. Indexing with ids is used for both anime, manga and users nodes. The tests are conducted for creating new nodes and searching for nodes. The test result in the table below shows that indexing has a significant impact on operations in Neo4j database especially for search. Indexing time decreases with using indexes for search tasks because it prevents to check all the nodes in the database.

Table 5.3: Neo4j Performance Test Results

Collection	Operation	Index	Time (ms)
Anime	Insert	No	5
	Insert	Yes	5
	Search	No	45
	Search	Yes	2
Manga	Insert	No	7
	Insert	Yes	5
	Search	No	46
	Search	Yes	9
Users	Insert	No	56
	Insert	Yes	9
	Search	No	40
	Search	Yes	3

Conclusion

Future Work

In the future, MangaVerse aims to expand its capabilities and enhance user experience through several key initiatives:

- **Enhanced Media Management:** Introduce functionalities for managers to update, add, and delete anime and manga entries. This will empower administrators to keep the platform's content relevant and up-to-date.
- **Advanced User Management:** Implement robust user management features, including the ability to delete user accounts if necessary. This ensures compliance with user data protection regulations and helps maintain a secure and trustworthy community environment.
- **Improved Recommendation System:** Refine personalized recommendation algorithms based on user preferences, viewing history, and community interactions. This will enhance user engagement by providing tailored content suggestions.
- **Analytics and Insights:** Develop comprehensive analytics tools for management purposes. These tools will provide insights into user behavior, content popularity, and platform usage trends. Analytics-driven decisions will guide future development and marketing strategies.
- **Enhanced Security Measures:** Strengthen security protocols with continuous monitoring and updates. Implement data encryption for all user data, including passwords, both in transit and at rest. Regular security audits and compliance checks will ensure MangaVerse remains a safe and reliable platform for users.
- **Community Engagement Features:** Introduce features to foster community interaction, such as forums, group discussions, and event notifications. These features will encourage active participation among manga and anime enthusiasts, creating a vibrant and supportive community.

Conclusion

MangaVerse has successfully established itself as a dynamic platform for manga and anime enthusiasts, offering a comprehensive suite of features including media exploration, personalized recommendations, and user profile management. The application leverages a robust technological stack, utilizing Java for backend development and MongoDB and Neo4j for database management, to ensure a seamless and efficient user experience.

Looking ahead, the future development roadmap focuses on refining management functionalities, enhancing security measures, and implementing advanced analytics and community engagement features. By prioritizing user satisfaction, data security, and innovative feature development, MangaVerse is poised to solidify its position as a leading platform for manga and anime enthusiasts.