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

Contents

Introduction

MangaVerse is a web application project developed for the Large-scale and multi-structured databases course of the University of Pisa. This web application aims to provide users with a comprehensive platform to explore, search, interact with a vast collection of manga and anime and interact with the other users.

The website is accessible without login providing a limited number of functionalities. Once a user logs in, the platform offers a wide range of features to personalize the user experience, in particular the social features. The application manages user and media content suggestions based on interactions, preferences and user information. Beside the user roles, the web application also has managerial roles. MangaVerse provides an analytics dashboards to track media contents and user activities also managing media contents and user accounts. These features allow manager to add, update, or remove manga and anime entries and monitor trends and rating.

Through its comprehensive set of features, MangaVerse aims to provide a community of manga and anime enthusiasts with a platform to explore, share, and engage with their favorite content. This platform enhances the user experience and facilitates deep engagement with both the content and the community.

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.

Requirements

Unregistered User:

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

Registered User:

- Logout.
- 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.
- 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(Registered User with Administrative Features):

- Logout.
- 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.
- 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 use case diagram

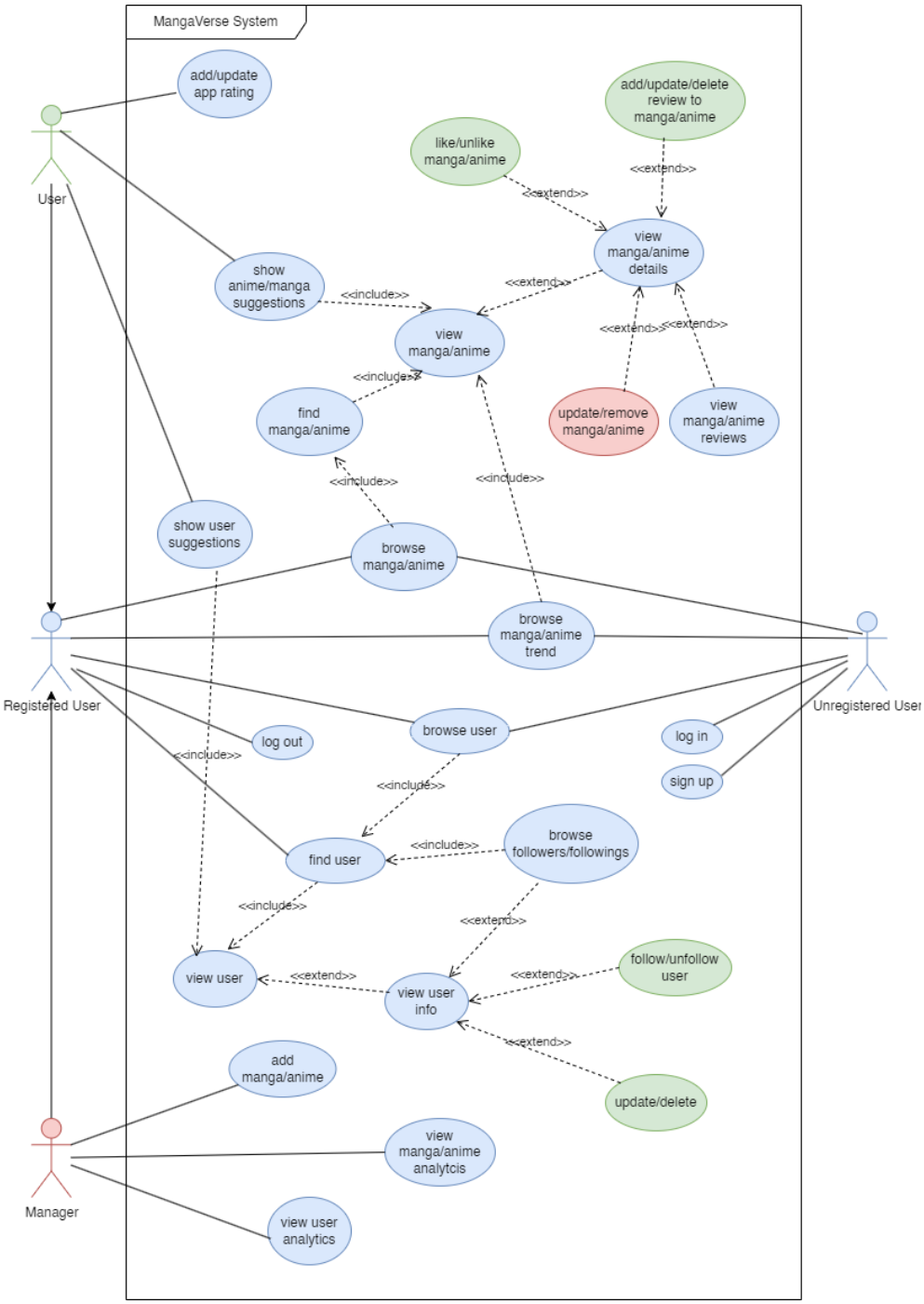

Figure 2.1: UML Use Case Diagram

UML class diagram

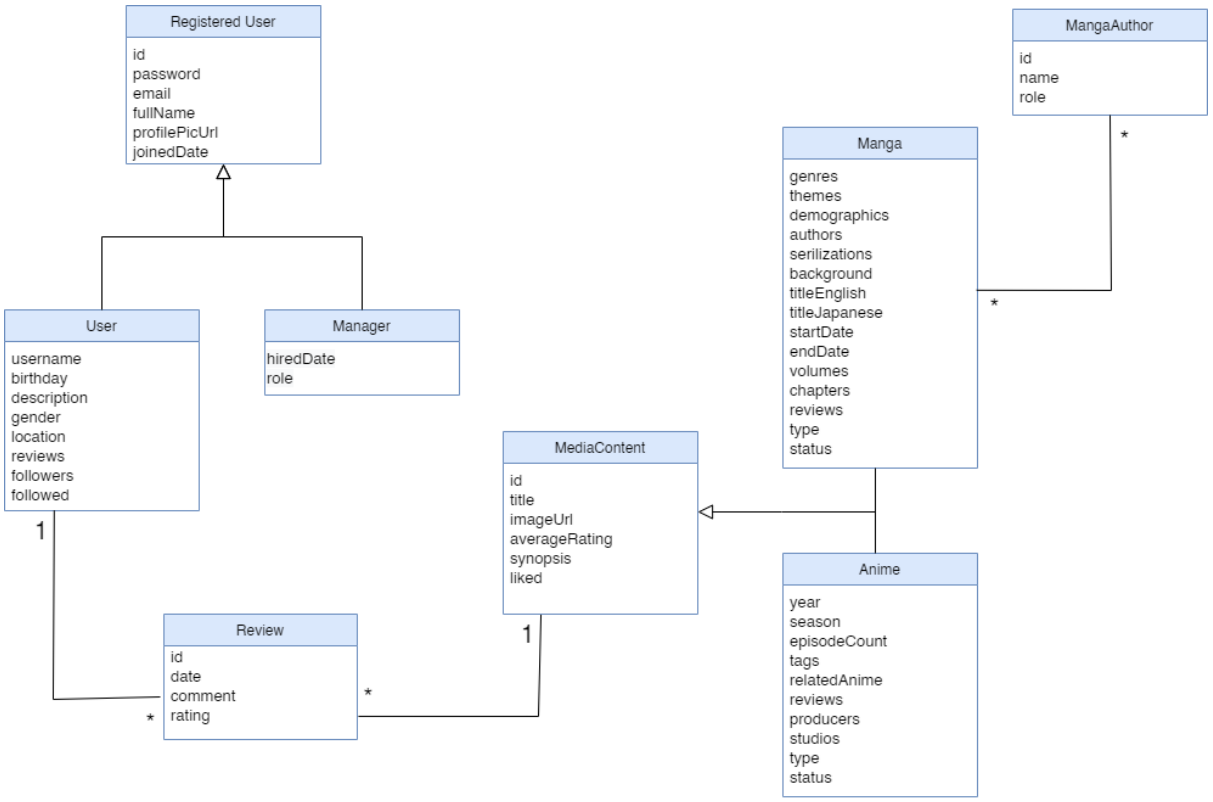

Figure 2.2: UML Class Diagram

Data Modeling

Data Collection

Sources: https://www.kaggle.com/datasets/dbdmobile/myanimelist-dataset?select=users-score-2023.csv, MyAnimeList.net, anilist.com, kitsu.io, livechart.me, anime-planet.com, nofity.moe, anisearch.com, anidb.net

Description: Manga, users and scores datasets were collected from MyAnimeList.net site using the official API and another unofficial API (Jikan). The anime datasets were collected from all the sources.

Variety: The datasets contain a variety of data types, including text, numbers, and dates. Anime are collected from 8 different sources. All the information is collected in 4 different csv files.

Volume: The datasets contain a large volume of data, with thousands of entries for anime, manga, users, and scores. The total size of the datasets is around 3 GB.

Data Cleaning and Preprocessing

Python scripts were used to clean and preprocess the data. The following steps were performed: reviews were created by merging the users and scores datasets, and creating comments about the media contents; the anime dataset was created by putting togethere the diffrerent sources; the manga dataset was created from MyAnimeList.net; the users dataset was cleaned and missing information, like email and password, was added.

Design

The web application needs to handle a big amount of data, so we decided to use a combination of different databases to store and manage the data. We will use a document database to store users, media contents and reviews data, and a graph database to store relationships between users and media content. This will allow us to efficiently store and retrieve data, as well as handle complex relationships between data.

Document Database

For the document database, we will use MongoDB. MongoDB is a NoSQL database that stores data in flexible, JSON-like documents. It is a popular choice for applications that require flexibility and scalability. These documents are flexible, meaning they can have different fields and structures. This makes MongoDB a good choice for applications that require flexibility in their data model. MongoDB is also a scalable database, meaning it can handle large amounts of data and traffic. It is designed to scale out, meaning you can add more servers to handle more traffic.

Collections The database will have the following collections:

- Anime: This collection will store information about anime, such as titles, tags, and synopsis.
- Manga: This collection will store information about manga, such as titles, genres, and authors.
- Reviews: This collection will store user ratings and comments for media content.
- Users: This collection will store user data, such as usernames, passwords, email addresses, gender and location.

MongoDB document example

Anime:


```
{
  "_id": "65789bb52f5d29465d0abcfb",
  "title": "0",
  "type": "SPECIAL",
  "episodes": 1,
  "status": "FINISHED",
  "picture": "https://cdn.myanimelist.net/images/anime/12/81160.jpg",
  "tags": [
    "drama",
    "female protagonist",
    "indefinite",
    "music",
    "present"
  ],
  "producers": "Sony Music Entertainment",
  "studios": "Minakata Laboratory",
  "synopsis": "This music video tells how a shy girl with a secret love and curiosity...",
  "latest_reviews": [
    {
      "id": "657b301306c134f18884924c",
      "date": "2023-10-03T22:00:00.000+00:00",
      "rating": 4,
      "user": {
        "id": "6577877ce68376234760745c",
        "username": "Tolstij_Trofim",
        "picture": "https://thypix.com/wp-content/uploads/2021/10/manga-profile-picture
    -10..."
      }
    },
  ],
  "anime_season": {
    "season": "FALL",
    "year": 2013
  },
  "average_rating": 6.7,
  "avg_rating_last_update": true,
  "likes": 4
}
```

Manga:


```
{
  "_id": "657ac61bb34f5514b91ea223",
  "title": "Berserk",
  "type": "MANGA",
  "status": "ONGOING",
  "genres": [
    "Action",
    "Adventure",
    "Award Winning",
    "Drama",
    "Fantasy",
    "Horror",
    "Supernatural"
  ],
  "themes": [
    "Gore",
    "Military",
    "Mythology",
    "Psychological"
  ],
  "demographics": [
    "SEINEN"
  ],
  "authors": [
    {
      "id": 1868,
      "role": "Story & Art",
      "name": "Kentarou Miura"
    },
    {
      "serializations": "Young Animal"
    }
  ],
  "synopsis": "Guts, a former mercenary now known as the \"Black Swordsman,\" is out fo...
    ",
  "title_english": "Berserk",
  "start_date": "1989-08-25T00:00:00.000+00:00",
  "picture": "https://cdn.myanimelist.net/images/manga/1/157897l.jpg",
  "average_rating": 3.33,
  "latest_reviews": [
    {
      "user": {
        "id": "6577877be683762347605ce7",
        "username": "calamity_razes",
        "picture": "https://imgbox.com/7MaTkBQR"
      },
      "date": "2012-12-15T00:00:00.000+00:00",
      "comment": "An insult to the art of manga; avoid at all costs.",
      "id": "657b302206c134f18886f5ef"
    },
  ],
  "anime_season": {
    "season": "FALL",
    "year": 2013
  },
  "average_rating": 6.7,
  "avg_rating_last_update": true,
  "likes": 4
}

```

Reviews:


```
{
  "_id": "657b300806c134f18882f2f1",
  "user": {
    "id": "6577877be68376234760596d",
    "username": "Dragon_Empress",
    "picture": "images/account-icon.png",
    "location": "Columbus, Georgia",
    "birthday": "1987-07-29T00:00:00.000+00:00",
    "rating": 7
  },
  "anime": {
    "id": "65789bbc2f5d29465d0b18b7",
    "title": "Slayers Revolution",
    "date": "2023-07-23T06:27:54.000+00:00",
    "comment": "Above-average quality in animation and soundtrack."
  }
}

```


Users:


```
{
  "_id": "6577877be683762347605859",
  "email": "xdavis@example.com",
  "password": "290cb38a679d5eb68d11b9ea1e21f48234eba6de19f95612dbcb70ce0c7e4e78",
  "description": "Liberating the mind from stress with the power of anime zen.",
  "picture": "https://thypix.com/wp-content/uploads/2021/10/manga-profile-picture-44",
  "username": "Xinil",
  "gender": "Male",
  "birthday": "1985-03-04T00:00:00.000+00:00",
  "location": "Libya",
  "joined_on": "2014-05-29T00:00:00.000+00:00",
  "app_rating": 5,
  "followed": 40,
  "followers": 29
}

```


The field `"app_rating"` is used to know the general satisfaction of the user with the application.

CRUD operations

- Create: This operation will allow users to create new documents in the database. For example, users can create new reviews for anime and manga.
- Read: This operation will allow users to read documents from the database. For example, users can read information about anime and manga and about other users.
- Update: This operation will allow users to update documents in the database. For example, users can update their reviews for anime and manga, they can also update their own profile, the manager can update media contents.
- Delete: This operation will allow users to delete documents from the database. For example, users can delete their reviews for anime and manga, the manager can delete media contents.

Graph Database

For the graph database, we will use Neo4j. Neo4j is a graph database that stores data in nodes and relationships. It is a popular choice for applications that require complex relationships between data. Neo4j is a graph database, which means it stores data in nodes and relationships. Nodes represent entities, such as users or products, and relationships represent connections between nodes. This makes Neo4j a good choice for applications that require complex relationships between data. Neo4j is also a scalable database, meaning it can handle large amounts of data and traffic. It is designed to scale out, meaning you can add more servers to handle more traffic. This makes Neo4j a good choice for applications that need to scale quickly.

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.

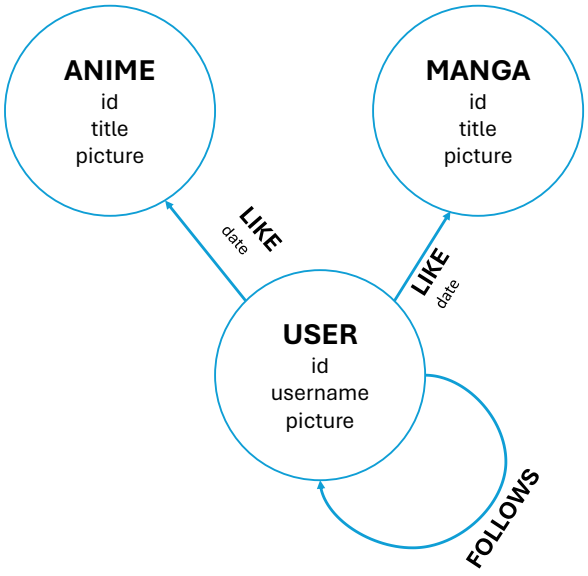

Figure 3.1: GraphDB

CRUD operations

- Create: This operation will allow users to create new nodes and relationships in the database. For example, users can create new relationships between users and media content:

  A user can LIKE a media content:

```
    MATCH (u:User {id: $userId}), (a:Anime {id: $animeId})
    WHERE NOT (u)-[:LIKE]->(a)
    CREATE (u)-[r:LIKE {date: $date} ]->(a)
    RETURN r

```

Listing 3.1: Create Like Relationship

  A user can FOLLOW another user:

```
    MATCH (u:User {id: $userId}), (f:User {id: $followedUserId})
    WHERE NOT (u)-[:FOLLOWS]->(f)
    CREATE (u)-[r:FOLLOWS]->(f)
    RETURN r

```

Listing 3.2: Create Follow Relationship

- Read: This operation will allow users to read nodes and relationships from the database. For example, users can read information about anime and manga and relationships between users and media content. A user can read the list of liked media contents:

```

    MATCH (u:User {id: $userId})-[:LIKE]->(a:Anime)
    RETURN a

```

Listing 3.3: Read Liked Media Contents

  A user can read the list of followers:

```
    MATCH (u:User {id: $userId})<-[:FOLLOWS]-(f:User)
    RETURN f

```

Listing 3.4: Read Followers

- Update: This operation will allow users to update nodes and relationships in the database. For example, users can update their likes for anime and manga and relationships between users.
- Delete: This operation will allow users to delete nodes and relationships from the database. For example, users can delete their likes for anime and manga and relationships between users.

  A user can unlike a media content:

```
    MATCH (u:User {id: $userId})-[r:LIKE]->(a:Anime {id: $animeId})
    DELETE r
    RETURN r

```

Listing 3.5: Delete Like Relationship

A user can unfollow another user:

```
MATCH (:User {id: $followerUserId})-[r:FOLLOWS]->(:User {id: $followingUserId})
DELETE r
RETURN r

```

Listing 3.6: Delete Follow Relationship

Availability and Partition Tolerance

MangaVerse, as a social network, gives priority to the AP configuration of the CAP theorem, ensuring Availability and Partition Tolerance. This allows users to access the application and interact with other users and media content, even if the data is not always consistent (Eventual Consistency).

Redundancy

The performance of the application is critical, so we need to ensure that the system is highly available and fault-tolerant. To achieve this, we gave priority to fast responses, rather than reducing memory consumption.

Latest reviews

In the anime and manga collections, there's a field containing the latest 5 reviews written for that specific media content, in this way it's fast to retrieve.

Average rating

In the anime and manga collections, there's a field containing 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, there's a field containing 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, there are fields containing the number of followers and followings, this field is updated every time a new follow relationship is created or deleted.

User field in Reviews

In the reviews collection, there's a field containing the user data, such as id, username, picture, and also location and birthday, which are used for suggestion porpouses.

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

Replicas

A cluster of three nodes is available for this project, allowing deployment of replicas: however, replicas were only implemented in MongoDB, as Neo4j required the Enterprise version for it. We have 3 replicas for MongoDB and 1 for Neo4J. In MongoDB we have one primary and two secondary replicas, the primary is used for write operations and the secondaries are used for read operations. This will allow us to distribute the load and improve the performance of the application. In case of failure of the primary node, one of the secondary nodes will be promoted to primary, ensuring high availability of the system.

Sharding

Sharding is a method for distributing data across multiple machines to meet the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. Even if not implemented, the database design is ready for sharding, as the data is distributed in a way that allows for easy sharding. The user, anime, manga, and reviews collections are sharded by the user id, anime id, manga id, and review id, respectively. This will allow us to distribute the data across multiple machines and improve the performance of the application.

Implementation

Development Environment

To ensure efficient and successful Implementation of MangaVerse web application, choosing the appropriate development environment is one of the most important points of the project.

Programming Languages

- **Backend:** Java is the main programming language used in the project's backend development.
- **Frontend:** HTML, CSS, JavaScript are utilized for building user interface in the project.
- **Data Preprocessing:** Python and java were used in the project to conduct data preprocessing task with the help of its powerful libraries and ease of use features.

Database

- **Document Database:** MongoDB is used in the project to store and manage document-based data with the help of its flexibility and scalability features.
- **Graph Database:** Neo4j is used in the project to manage and query graph data and handle complex relationships and connections between user entities and media contents in an efficient way.

Integrated Development Environment

Intellij IDEA was used as an primary IDE. It is powerful Java integrated development environment for developing software in an efficient way.

Version Control

Github was used to provide a collaborative development with its version control system.

Web Server

Apache Tomcat was used as a web server to provide reliable environment for deploying and running the java based web application.

Build Automation

Maven was used as a build automation tool. It is used to manage the project's build, reporting, and documentation from a central piece of information.

Testing

JUnit was used as a testing framework for Java code. It is used to write and run repeatable automated tests. This ensures the reliability and efficiency of the codebase throughout the development process.

Main Modules

- Configuration
- Controller
- DAO (Data Access Objects)

- DTO (Data Transfer Objects)
- Model
- Service
- Utils
- User Interface

Configuration

Configuration module contains a class named *AppServletContextListener* which is responsible for initializing and managing database connections for the web application. The configuration class implements ServletContextListener interface. *@WebListener* annotation is used to provide listening for application lifecycle events. This annotation contains two methods, which are *contextInitialized(ServletContextEvent sce)* and *contextDestroyed(ServletContextEvent sce)*. The first method is called when the web application is started and the second method is called when the web application is shut down.

Database Connection Management: Database connection is provided with *openConnection()* and *closeConnection()* methods. They are both initialized for managing connection for MongoDB and Neo4j databases. Connections are managed with corresponding DAO classes which are BaseMongoDBDAO and BaseNeo4jDAO.

With using the configuration module for database connection, web application ensures robustness and reliability in its data access layer.

Controller

The controller modules plays a role as intermediary between the user requests and backend of the MangaVerse wab application as servlet classes. They receives the user requests, process them and returns with the corresponding response. The controller classes are implemented using HttpServlet to handle user requests and responses. Within the scope of their intermediary role, the controller classes are responsible of being a bridge between the user interface and backend logic. When a user interacts with the web application, their actions are translated into a HTTP request and these requests are handled by the related servlet class in the controller module. To be able to do do request translation in an efficient way each controller class extend 'HttpServlet' and has various methods to handle HTTP requests like GET and POST. Each controller class utilized a switch-case structure to determine the action requested and invokes the appropriate handler method accordingly. This structure allows for clear and organized routing of request to their corresponding handler method. After processing the request, the servlet generates a requested response.

Example code snippet from MediaContentServlet:

```
{
    protected void processRequest(HttpServletRequest request, HttpServletResponse response
    ) throws ServletException, IOException {
        String action = request.getParameter("action");

        switch (action) {
            case "toggleLike" -> handleToggleLike(request, response);
            case "addReview" -> handleAddReview(request, response);
            case "deleteReview" -> handleDeleteReview(request, response);
            case "editReview" -> handleEditReview(request, response);
            case "getMediaContent" -> handleGetMediaContentById(request,response);
            case "getMediaContentByTitle" -> handleSearchMediaContentByTitle(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
  The AuthServlet class handles the user authentication and authorization processes. It includes login, logout and sign up functions
- MainPageServlet
  The MainPageServlet class is responsible for handling the main page of the web application. It includes the main page of the web application and the search functionality. It provides request related to displaying main page and searching media contents.
- ManagerServlet
  The ManagerServlet class manages administrative requests in manager page. These request are primarily about manga, anime and user analytics such *averageRatingByMonth(), trendMediaContentByYear(), getBestCriteria()...*
- MediaContentServlet
  The MediaContentServlet class is responsible for managing request related with media contents. These requests include like, adding,deleting or editing reviews and retrieving media content details.
- ProfileServlet
  The ProfileServlet class is responsible for managing user profile related requests. These requests include updating user profile, following/unfollowing other users, getting user profile details such as liked anime and manga and user reviews.
- UserServlet
  The UserServlet class is responsible for managing user related requests and interactions. These requests include retrieving followers list, following list and user information.

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

Techniques

Task Manager:
Task Manager class which is located in the service module of the system provides asynchronous task execution with using *PriorityBlockingQueue*. It helps to order the tasks according to their prioritizes. After that prioritization, it ensures that higher priority tasks will be executed first and if two tasks have the same priority the one which is created before will be executed first. While Task Manager class is able to start and stop the tasks within the functions inside, it can also take tasks to the queue in a thread-safe way. By using taskComparator for ordering the tasks, the system provides also effective scheduling and execution.

Aperiodic Executor Task Service:
Executor Task Service class which is located inside the service module of the system is an important part for providing the eventual consistency. Executing tasks in asynchronous way with threads guarantees eventual consistency across different collections, mongoDB and neo4j and different replicas. With the help of the Executor Task Service, tasks that are needed to be executed in an asynchronous way are handled by ensuring that changes propagate correctly across different part of the system. While using multiple databases and data replicas for this web application, it is important for maintain data integrity and eventual consistency. Executing the tasks in an asynchronous way using threads allows to perform operations without blocking the main execution flow. Aperiodic executer task service class is implemented by using the interface of executor service.

Description of Main Classes

Controller

Class	Description
AuthServlet	Handles business logic for authentication
MainPageServlet	Handles business logic for main page
ManagerServlet	Handles business logic for manager
MediaContentServlet	Handles business logic for media content
ProfileServlet	Handles business logic for profile
UserServlet	Handles business logic for user

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

Class	Sub-package	Description
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

Class	Sub-package	Description
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
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");
        }

        Map<String, Double> map = new LinkedHashMap<>();
        for (Document doc : aggregationResult) {
            String ageRange = convertIntegerToAgeRange(doc.getLong("_id"));
            map.put(ageRange, doc.getDouble("avg_app_rating"));
        }

        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('COLLECTION_NAME').aggregate([
    {
        // Match stage: Filters documents to include only those that
        // have both the 'birthday' field and 'app_rating' field.
        $match: {
            $and: [
                { birthday: { $exists: true } },
                { app_rating: { $exists: true } }
            ]
        }
    },
    {
        // Project stage: Adds a new field 'age' calculated by subtracting
        // the 'birthday' from the current date, converting the difference
        // from milliseconds to years, and taking the floor of the result.
        // Also includes the 'app_rating' field.
        $project: {
            age: {
                $floor: {
                    $divide: [
                        { $subtract: [ new Date(), "$birthday" ] },
                        1000 * 60 * 60 * 24 * 365
                    ]
                }
            },
            app_rating: 1
        }
    },
    {
        // Bucket stage: Groups the documents into buckets based on the 'age' field.
        // Specifies boundaries for the buckets and assigns documents with an age
        // outside these boundaries to the default bucket (50+ years).
        // For each bucket, calculates the average value of 'app_rating'.
        $bucket: {
            groupBy: "$age",
            boundaries: [0, 13, 20, 30, 40, 50],
            default: 50,
            output: {
                avg_app_rating: { $avg: "$app_rating" }
            }
        }
    }
]).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:

```
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:

```
// 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();
        }

```

```
50          if(remaining > 0) {
51              StringBuilder queryBuilder3 = new StringBuilder();
52              if (type == MediaContentType.ANIME)
53                  queryBuilder3.append("MATCH (u:User {id: $userId})-[r:LIKE]->(media:Anime)<-[:
    LIKE]-(suggested:User) ");
54              else
55                  queryBuilder3.append("MATCH (u:User {id: $userId})-[r:LIKE]->(media:Manga)<-[:
    LIKE]-(suggested:User) ");
56              queryBuilder3.append("""
57                      WHERE u <> suggested
58                      WITH suggested, COUNT(DISTINCT media) AS commonLikes
59                      RETURN suggested AS user, commonLikes
60                      ORDER BY commonLikes DESC
61                      LIMIT $n
62                      """);
63              String query2 = queryBuilder3.toString();
64              Value params3 = parameters("userId", userId, "n", n);
65
66              List<Record> records = session.executeRead(tx -> tx.run(query2, params3).list());
67              for (Record record : records) {
68                  UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
69                  if (!suggested.contains(userDTO))
70                      suggested.add(userDTO);
71                  if (suggested.size() == n)
72                      break;
73              }
74          }
75
76          return suggested.isEmpty() ? null : suggested;
77
78      } catch (Neo4jException e) {
79          throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
80
81      } catch (Exception e) {
82          throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
83      }
84  }
```

- Neo4j Query:

```
1  // Match the user with the given userId who has liked media of type Manga
2  MATCH (u:User {id: $userId})-[r:LIKE]->(media:Manga)<-[:LIKE]-(suggested:User)
3  // Filter out the original user from the suggested users and only consider likes after the
       specified date
4  WHERE u <> suggested AND r.date >= $date
5  // Pass the suggested users and count of distinct media liked by both users to the next stage
6  WITH suggested, COUNT(DISTINCT media) AS commonLikes
7  // Filter out users with common likes less than or equal to the minimum threshold
8  WHERE commonLikes > $min
9  // Return the suggested users and the count of common likes
10 RETURN suggested AS user, commonLikes
11 // Order the results by the count of common likes in descending order
12 ORDER BY commonLikes DESC
13 // Limit the number of results to the specified maximum
14 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:

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

```
4           int remaining;
5
6           // suggest users that follow user's followings and have more than 5 common followings
7           String query = """
8                   MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)<-[:FOLLOWS]-(
    suggested:User)
9                   WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
10                  WITH suggested, COUNT(DISTINCT following) AS commonFollowings
11                  WHERE commonFollowings > 5
12                  RETURN suggested as user
13                  ORDER BY commonFollowings DESC
14                  LIMIT $n
15                  """;
16          Value params = parameters("userId", userId, "n", n);
17
18          List<UserSummaryDTO> suggested = session.executeRead(
19                  tx -> tx.run(query, params).list()
20          ).stream()
21                  .map(this::recordToUserSummaryDTO)
22                  .collect(Collectors.toList());
23
24          remaining = n - suggested.size();
25
26          // if there are not enough suggestions, suggest users that are followed by the user's
    followings and have more than 5 connections
27          if (remaining > 0) {
28              String query2 = """
29                      MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)-[:FOLLOWS]->(
    suggested:User)
30                      WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
31                      WITH suggested, COUNT(DISTINCT following) AS commonUsers
32                      WHERE commonUsers > 5
33                      RETURN suggested as user
34                      ORDER BY commonUsers DESC
35                      LIMIT $n
36                      """;
37              Value params2 = parameters("userId", userId, "n", n);
38
39              List<Record> records = session.executeRead(tx -> tx.run(query2, params2).list());
40              for (Record record : records) {
41                  UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
42                  if (!suggested.contains(userDTO))
43                      suggested.add(userDTO);
44                  if (suggested.size() == n)
45                      break;
46              }
47
48              remaining = n - suggested.size();
49          }
50
51          // if there are still not enough suggestions, suggest users that follow the user's
    followings
52          if (remaining > 0) {
53              String query3 = """
54                      MATCH (u:User {id: $userId})-[:FOLLOWS]->(following:User)<-[:FOLLOWS]-(
    suggested:User)
55                      WHERE NOT (u)-[:FOLLOWS]->(suggested) AND u <> suggested
56                      WITH suggested, COUNT(DISTINCT following) AS commonFollowings
57                      RETURN suggested as user
58                      ORDER BY commonFollowings DESC
59                      LIMIT $n
60                      """;
61              Value params3 = parameters("userId", userId, "n", n);
62
63              List<Record> records = session.executeRead(tx -> tx.run(query3, params3).list());
64              for (Record record : records) {
65                  UserSummaryDTO userDTO = recordToUserSummaryDTO(record);
66                  if (!suggested.contains(userDTO))
67                      suggested.add(userDTO);
68                  if (suggested.size() == n)
69                      break;
70              }
71          }
```

```
72
73          return suggested.isEmpty() ? null : suggested;
74
75      } catch (Neo4jException e) {
76          throw new DAOException(DAOExceptionType.DATABASE_ERROR, e.getMessage());
77
78      } catch (Exception e) {
79          throw new DAOException(DAOExceptionType.GENERIC_ERROR, e.getMessage());
80      }
81  }
```

- 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	-	-
	Search	Yes	10	0	1

Continued on next page

Table 5.2 – continued from previous page

Collection	Operation	Index	Time (ms)	Total Keys Examined	Total Docs Examined
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

Conclusion

The MangaVerse is a web application project that provides a comprehensive web application for dynamic social platform for manga and anime enthusiasts. The web application allows users to explore, search media content and be in contact with other users by review system. Having a user-friendly interface, the application is designed to have a robust set of features. The applications offers functionalities for both unregistrered user and registered user including manager purposes such as browse media content, personalized recommendations, profile management and analytics checking for management purposes.

Beside the functional requirements, the application has also well-defined development process and architecture using different technologies and techniques. While java is used for main backend development programming language, as a database MongoDB and Neo4j are used.

Future Work

For the future: manager will be able to update add delete anime and manga and delete user.

Security

- Data Encryption: All user data, including passwords, should be securely encrypted during transmission and storage.
- Delete user accounts if necessary.user management