UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Master's Degree in Artificial Intelligence and Data Engineering

LARGE SCALE AND MULTI STRUCTURED DATABASES PROJECT

My Akiba

https://github.com/Gianma23/my-akiba.git

Students:

Gianmaria Saggini
Cristina Maria Rita Lombardo
Alessio Franchini

ACADEMIC YEAR 2024/2025

1. Introduction

My Akiba is an application that allows its users to keep track of their read manga and watched anime. It provides a large catalogue of both media allowing users to find new anime or manga to watch or read by also providing them with reviews from other users and the scores given to each media.

This application also requires an administrator, he plays a pivotal role in maintaining Akiba's ecosystem. He can manage the catalogue by adding, updating, or removing media entries, moderate reviews, and access detailed user and media analytics. These analytics enable administrators to identify user trends, influencers, and popular media, as well as oversee system performance.

By integrating these diverse functionalities, My Akiba represents a user-centric environment where fans can seamlessly track their favourite content, connect with users with similar tastes, and stay informed about the latest trends in the world of manga and anime.

2. Design

2.1. Actors

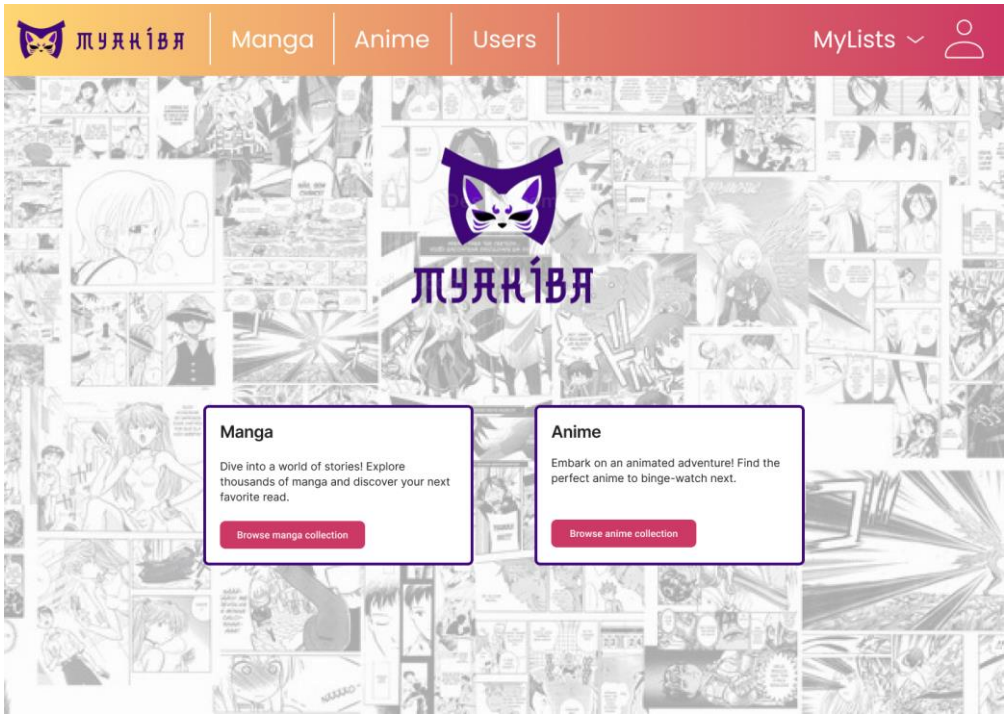
In **My Akiba** there are three main actors:

- The Unregistered User
- The Registered User
- The Administrator

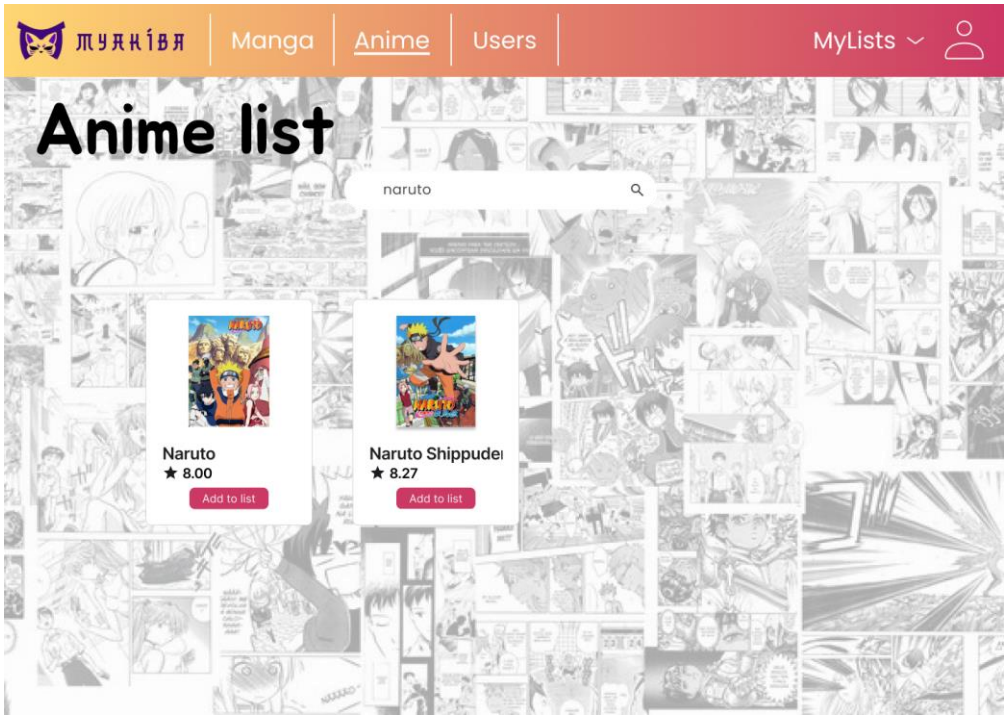
Both Unregistered User and Registered User are the end-users of our application. The first one has limited access to the functionalities, while the second one can use the application in a more meaningful way, creating lists, reviewing and scoring the media. The Administrators have the role of managing both registered users in the application and media in the catalogue.

2.2. Mockups

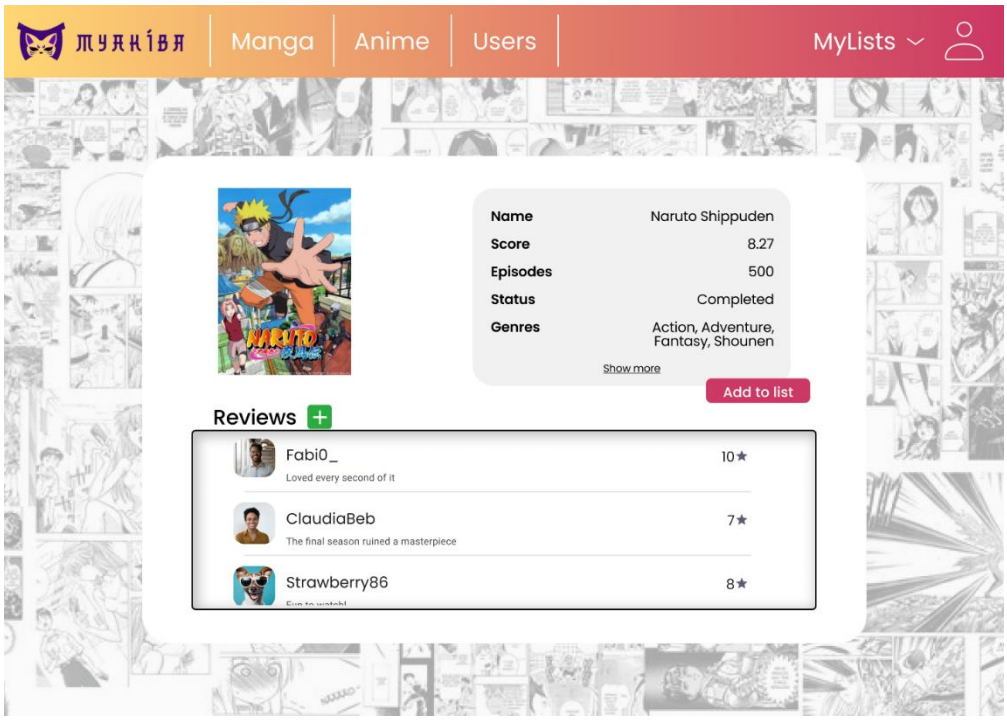
2.2.1. Unregistered user view



Main page where the user can choose to browse manga or anime or login to access other functionalities.

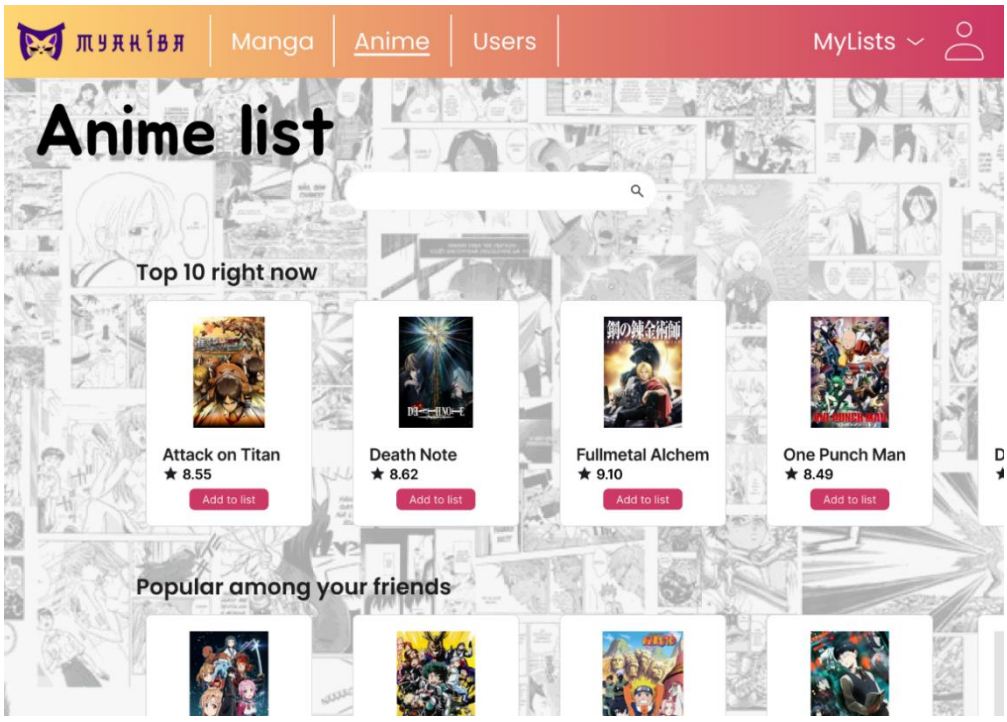


Users can search for a media by its name.

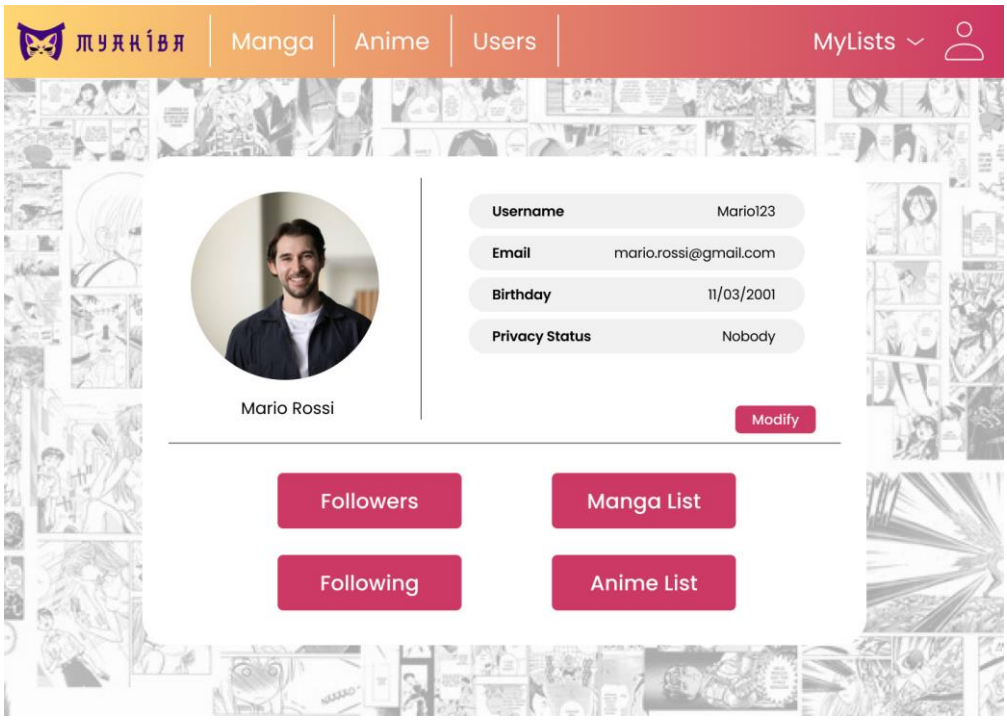


Each media has its info and reviews displayed in a page.

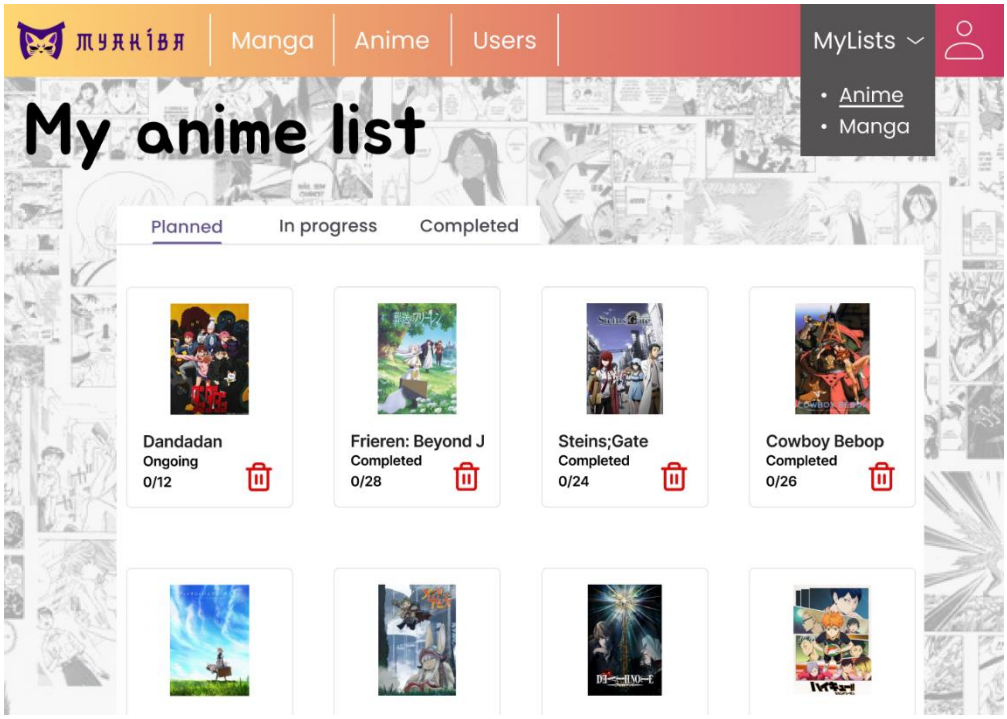
2.2.2. Registered user view



Registered users can browse among media recommended for them.

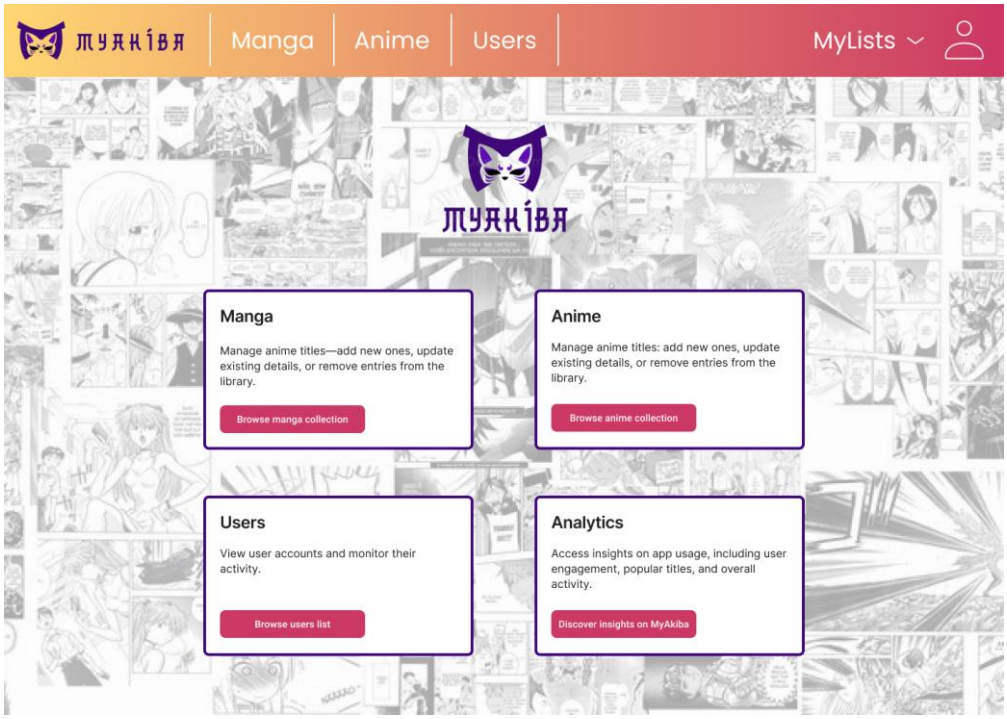


Personal user profile.

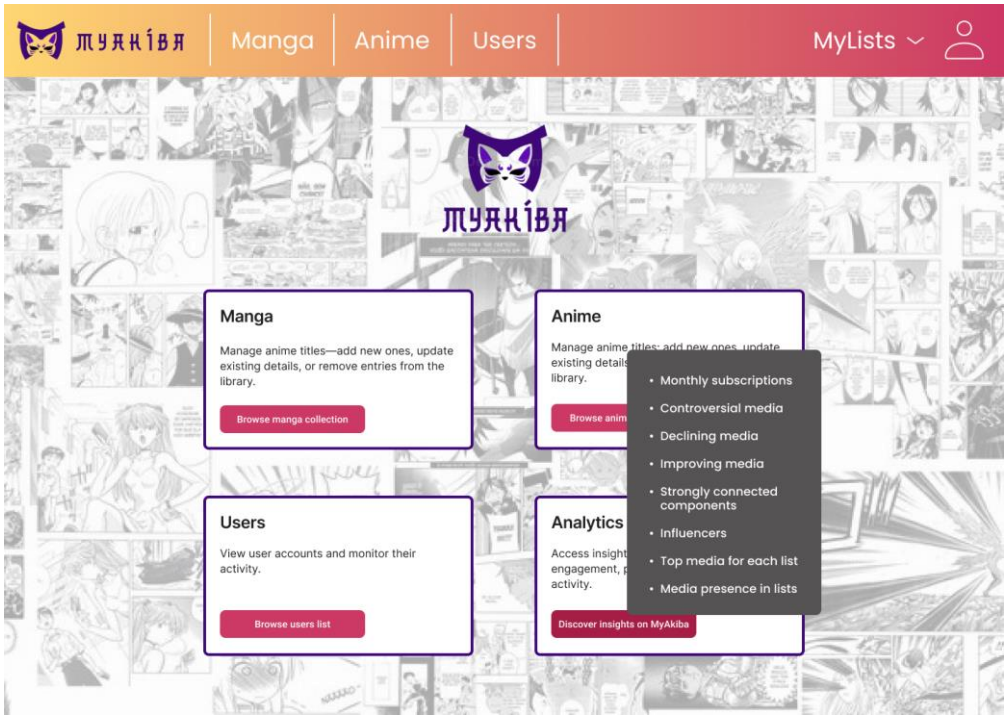


Personal user lists, divided by the user progress for each media.

2.2.3. Administrator view



Main page with all the functionalities offered to the administrator.



List of the analytics available.

2.3. Requirements

2.3.1. Functional requirements

1. Unregistered user

1.1. Browse

1.1.1. The system must allow an unregistered user to browse the list of anime and manga.

1.1.2. The system must allow an unregistered user to see details of a particular anime or manga.

1.1.3. The system must allow an unregistered user to perform a text search on manga or anime.

1.2. Registration

1.2.1. The system must allow an unregistered user to register.

2. Registered user

2.1. Account management

2.1.1. The system must allow a registered user to login.

2.1.2. The system must allow a registered user to delete its account.

2.1.3. The system must allow a registered user to update its personal information.

2.2. Browse

2.2.1. The system must allow a registered user to browse the list of anime and manga.

2.2.2. The system must allow a registered user to see details of a particular anime or manga.

2.2.3. The system must allow a registered user to perform a text search on manga or anime.

2.2.4. The system must allow a registered user to browse other registered users.

2.2.5. The system must allow a registered user to find other registered users by username.

2.3. Lists management

2.3.1. The system must offer 3 lists, both for manga and anime (6 lists in total):

- *Planned* list
- *In Progress* list
- *Completed* list

2.3.2. The system must allow a registered user to add a media to the corresponding *Planned* list.

2.3.3. The system must allow a registered user to update the number of watched episodes of an anime. If the number of watched episodes is higher than zero, the anime is put in the *In Progress* list of anime. If the user has seen all the episode, the anime is put in the *Completed* list of anime.

2.3.4. The system must allow a registered user to update the number of read chapters of a manga. If the number of read chapters is higher than zero, the manga is put in the *In Progress* list of manga. If the user has read all the chapters, the manga is put in the *Completed* list of manga.

2.3.5. The system must allow a registered user to remove a manga or anime from any list.

2.4. Social features

2.4.1. The system must allow a registered user to follow another registered user.

2.4.2. The system must allow a registered user to unfollow another registered user.

2.4.3. The system must allow a registered user to see the list of followed users.

2.4.4. The system must allow a registered user to see the list of their followers.

2.4.5. The system must allow a registered user to specify their privacy status. The privacy status tells who can see the lists of the user. The possible values are:

- Nobody can see my personal lists.
- Only registered users that follow me can see my personal lists.
- All registered users can see my personal lists.

Personal lists mean the followers, following, anime and manga lists.

2.4.6. The system must allow a registered user to see non-personal details of a registered user.

2.4.7. The system must allow a registered user to see the lists of a registered user, according to its privacy status.

2.5. Feedback

2.5.1. The system must allow a registered user to review a media. The review must have a vote from 1 to 10, and optionally a comment.

2.5.2. The system must allow a registered user to delete its own reviews.

2.5.3. The system must allow a registered user to see the list of reviews for a media.

2.6. Recommendations

2.6.1. The system must offer a registered user a list of the highest rated anime or manga, globally and for each genre.

2.6.2. The system must offer a registered user a list of registered users with similar tastes.

2.6.3. The system must offer a registered user a list of manga or anime popular among followed users.

3. Administrator

3.1. Browse

3.1.1. The system must allow an administrator to browse the list of anime and manga.

3.1.2. The system must allow an administrator to see details of a particular anime or manga.

3.1.3. The system must allow an administrator to perform a text search on manga or anime.

3.1.4. The system must allow an administrator to browse registered users.

3.1.5. The system must allow an administrator to find other registered users by username.

3.2. User management

3.2.1. The system must allow an administrator to see all details of a registered user.

3.2.2. The system must allow an administrator to see the lists of a registered user, independently of its privacy status.

3.3. Catalogue management

3.3.1. The system must allow an administrator to add an anime or manga.

3.3.2. The system must allow an administrator to update an anime or manga information.

3.3.3. The system must allow an administrator to delete an anime or manga.

3.3.4. The system must allow an administrator to delete a review.

3.4. Analytics

3.4.1. The system must offer an administrator analytics about users namely:

- For each year, see the month with most registrations.
- Find communities.
- Find users that are influencers.

3.4.2. The system must offer an administrator analytics about anime and manga namely:

- See anime/manga present more in each list.
- For a media, see how many users have it in their planned, in progress and completed lists.
- Retrieve a list of the most controversial media for each genre.
- Retrieve a list of media that are going worse than usual.
- Retrieve a list of media that are going better than usual.

2.3.2. Non-Functional Requirements

1. Product requirements

1.1. The system must be capable of returning to the user quick responses.

1.2. The system must be highly available. That is, there should not be down time.

1.3. The system should be able to scale to accommodate an increase in user traffic, for instance when a new episode of a popular anime is released, without any degradation of the response time and availability.

2. Implementation requirements

2.1. The system must be developed using Object-Oriented Programming languages.

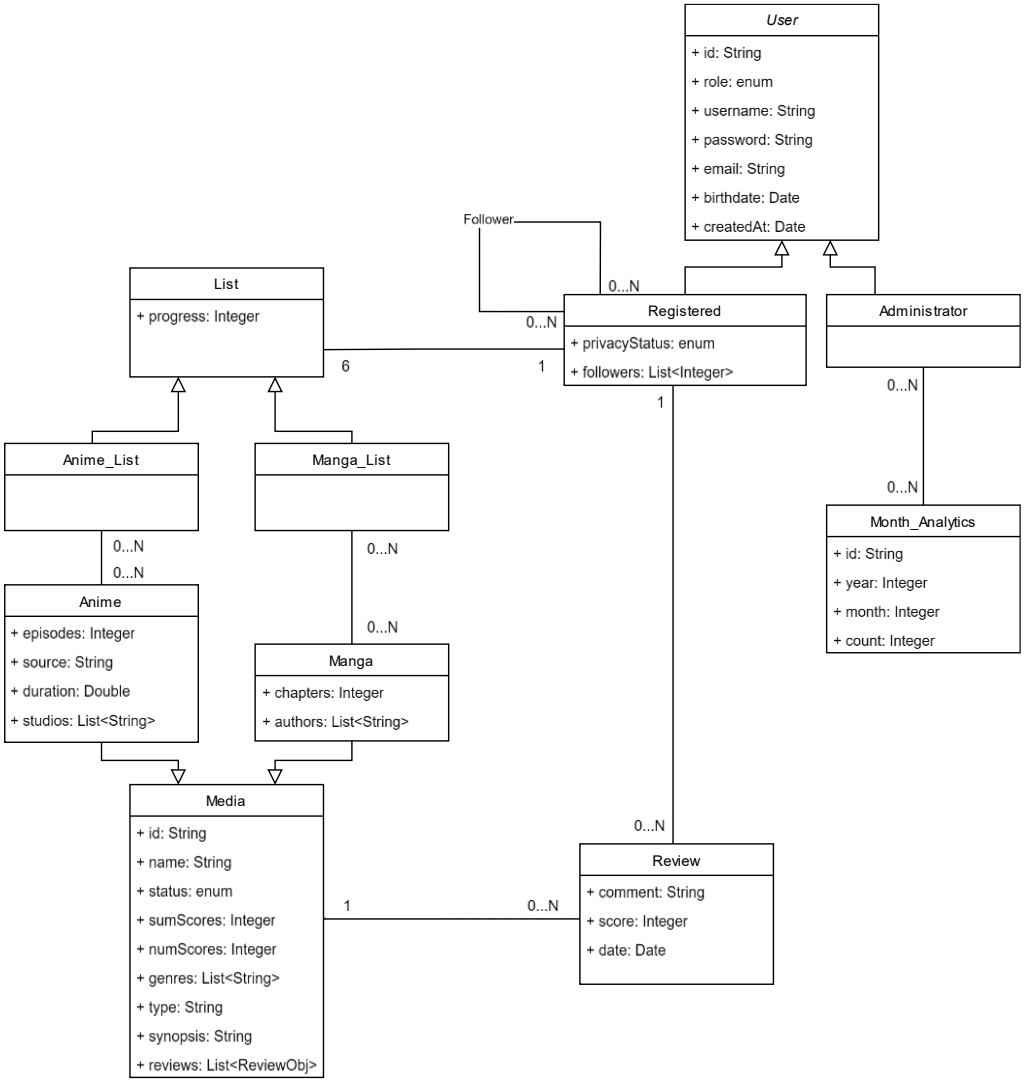
2.2. The system must expose its functionalities by means of RESTful APIs, according to OpenAPI Specification.

3. Security requirements

3.1. The system must use a basic authentication protocol to secure access.

3.2. The system must encrypt users' passwords.

2.4. UML Class Diagram



2.5. Data Models

2.5.1. Document DB Collections

The document DB has been used to store the data of the API creating the collections:

- Users
- Anime
- Manga
- Month_analytics

The **Users** collection stores all documents related to the Registered Users and the Administrators. It stores personal information provided at the registration and other useful details, some of the main attributes that can be found in such documents are:

- *createdAt*: the timestamp when the account was created.
- *privacyStatus*: privacy level chosen by the user for the visibility of their profile and lists:
    - "NOBODY": Profile is not visible to anyone.
    - "FOLLOWERS": Profile is visible to followers only.
    - "ALL": Profile is visible to everyone.
- *followers*: the array of users' ids that follow this account, using the document linking to connect a user to its followers.

The **Anime** collection stores the documents providing information about the anime media, allowing a performant access to all the data a user may be interested in for all the anime present in the database. Some attributes related to the anime documents are:

- *episodes*: the current number of episodes released.
- *duration*: the mean duration of the episodes of an anime, in seconds.
- *source*: the original source of the anime, if it is an original, a transposition of a manga or other writings.

The **Manga** collection stores the documents providing information about the manga media, allowing a performant access to all the data a user may be interested in for all the manga present in the database. Some attribute related to the manga documents are:

- *chapters*: the current number of released chapters of a manga.
- *authors:* the list of the authors that have participated in the writing of a manga.

The **anime** and **media documents** are stored in different collections for a faster retrieval and there is no case where they need to be retrieved together, however their structures share some common elements:

- *status*: providing information about the release status of the media, allowing the user, and the system to know if it's "ONGOING" or "COMPLETE".
- *sumScores*: the sum of the ratings given by the users to the media in a range from 1 to 10.
- *numScores*: the number of the scores. With the previous attribute they allow to calculate the mean score given to the media without recalculating it every time a new score is added.
- *type*: provides information about the type of media released. If it is serialised, a movie or a special episode for an anime; if it is a light novel or a manga for the manga media.
- *reviews*: using document embedding we find in this attribute the documents related to the reviews, containing information on the reviewer (*userId* and *username*), the *score* given to the media, the *timestamp* of the review and, optionally, a *comment* provided by the user.

The **Month Analytics** collection stores the data generated by analytics provided to the administrator, specifically it keeps, for each year, the month with most registrations to the application, allowing the system to retrieve such data directly from the database and not execute the count each time the analytic is requested.

2.5.1.1. Document Examples

User document


```
{
  _id: '6a1c1930-9811-4b00-aa85-64bea06770e8',
  role: 'USER',
  username: 'gianma',
  password: '$2a$12$e2dOgfNSb5T2oIS2R0HDHuKFRdAmIGklUTe260kZcsGe2NOavLLKO',
  email: 'gianmaria.saggini@gmail.com',
  privacyStatus: 'FOLLOWERS',
  birthdate: 2001-03-01T00:00:00.000Z,
  createdAt: 2025-01-08T21:06:53.252Z,
  _class: 'it.unipi.myakiba.model.UserMongo',
  followers: [
    '65397404-1b08-421b-a21e-4d6c54e7ecfb'
  ]
}
```


Anime document


```
{
  _id: '4742',
  name: 'Papa to Odorou',
  status: 'COMPLETE',
  type: 'tv',
  episodes: 16,
  duration: 390,
  studios: [
    'StudioDeen'
  ],
  source: 'manga',
  genres: [
    'Comedy'
  ],
  synopsis: 'Imagine two irresponsible siblings (
  numScores: 1,
  sumScores: 7,
  reviews: [
    {
      userId: '503',
      username: 'vatu',
      score: 7,
      timestamp: 2022-08-05T07:42:48.000Z
    }
  ]
}
```

Manga document

Month Analytics document

2.5.2. Graph DB Structure

This database choice has been selected to manage the social features of the system, meaning the relationship between users in a follower-followed pattern, and to keep track of the different lists available to the users.

2.5.2.1. Graph Nodes

In the graph database there are three kinds of nodes:

- **User** nodes with only information regarding their *id*, *username* and *privacyStatus*.
- **Anime** nodes having information about *id*, *name*, *status*, *episodes* and *genres* of the media.
- **Manga** nodes having the information about *id*, *name*, *status*, *chapters* and *genres* of the media.

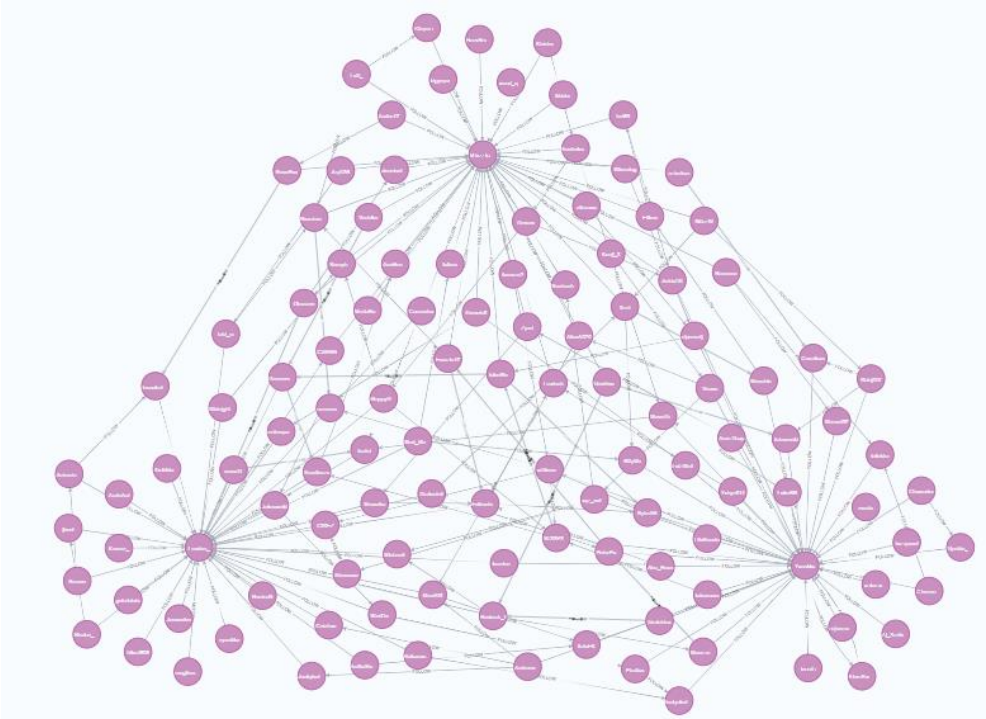
2.5.2.2. Graph Relationship

The relations between the nodes can be of two types:

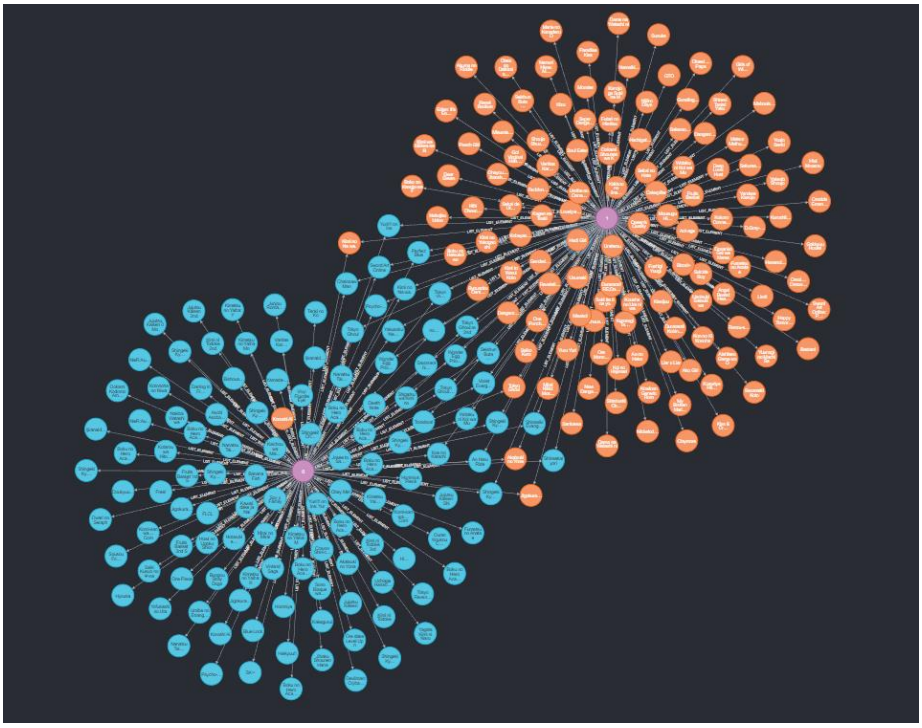
- **:FOLLOW**: relationship between two different User nodes
- **:LIST_ELEMENT**: relationship between a User node and an Anime or Manga node. This arch has the attribute *progress* to allow a user to keep track of their progress related to the media, and the system to define the type of list the media belongs to.

2.5.2.3. Graph Examples

:User − [: FOLLOW] → :User



:User − [: LIST_ELEMENT { progress }] → :Anime (orange)
:User − [: LIST_ELEMENT { progress }] → :Manga (blue)



2.6. Distributed Database Design

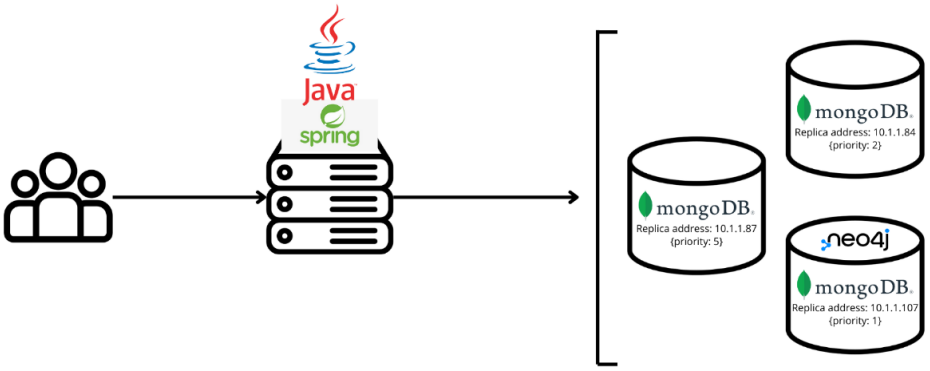
According to the non-functional requirements written in paragraph 2.2.2. our system must ensure High Availability and Partition Tolerance allowing quick responses to users' requests. Considering the CAP theorem our application has been developed according to the AP solution compromising in the consistency side of the theorem, implementing eventual consistency. For this objective the fundamental aspect of the design stands in the data distribution and replication approach.

2.6.1. Replicas

Our Mongo DB replica set is made up of a cluster of three nodes provided for the project. Out of these nodes we deploy three replicas only for the document DB, and we leave the replication of the graph DB, implemented with Neo4j, for future improvements, keeping only one replica in this project. The result is 3 replicas for MongoDB and 1 replica for Neo4j.

The replica set is made of one primary node and two secondary nodes. The nodes are configured with different priorities, one with priority 5, making it the most likely to become the primary node when the election starts and two with a lower priority of 2 and 1, which will be the secondary nodes. This last node, with the lower priority value, is also the node in which we have installed the Neo4j service; assigning it the lowest

priority ensures that it is selected as the primary node for MongoDB only as a last resort, thereby minimizing its workload under normal circumstances.



Read Operations (read preference: nearest)

Given the high volume of read requests expected in our system and the requirement of having a quick response to users' requests we chose to implement the queries to the Mongo DB replica set with the reading preference set to *nearest*, retrieving the data from the nearest replica in terms of latency. This choice also helps to balance the database workload, preventing bottlenecks on the primary node and ensuring smoother operation during peak usage times. Additionally, this approach improves scalability by leveraging secondary nodes for read operations, making the system capable of handling a growing user base.

Write Operations (write concern: majority)

Considering the functionalities of our application the write operation has a high volume, even if much lower than the read one, especially in the context of users lists, and their update. Even with these considerations we have thought that in these exceptional circumstances (having only three replicas) the choice of *majority* as the write concern is the best trade off, ensuring a strong consistency, waiting for the write operation to be executed on two replicas. This choice, considering also the setting for the read operations, guarantees that most reads (even from secondaries) reflect the most recent data. The slightly increasing write latency shouldn't be an issue considering the limited number of replicas used. This choice can be revaluated with an increase in the number of replicas, limiting the number of acknowledgments waited to end the operation.

Operations Volumes

What follows is a table with an estimated number of occurrences for each of the main operations provided by the application. The values have been calculated by estimating 100.000 active users per day.

Action	%	**# Daily occurrences**	**MongoDB**	**Neo4j**	**Daily total**
Register user	0.01	10	1	1	20
Login User	100	100000	1	0	100000
Browse Users	10	10000	1	0	10000
Find User	1.5	1500	1	0	1500
Update User Info	0.9	900	5	2	90000
Browse List	150 x2[1]	150000 x2[1]	0	1	150000
Add Media to List	5 x2[1]	5000 x2[1]	0	1	10000
Update List Progress	20 x2[1]	2000 x2[1]	0	1	4000
Remove From List	0.5	500	0	1	500
Follow a User	3	3000	1	1	6000
Unfollow User	1	1000	1	1	2000
Browse Followers	5	5000	0	1	5000
Post Review	1	1000	2	0	2000
Browse Media	120 x2[1]	120000 x2[1]	1	0	240000
Find Media	50 x2[1]	50000 x2[1]	1	0	100000
Media Recommendation	40 x2[1]	40000 x2[1]	0	1	80000
User Recommendation	3	3000	0	1	3000
Find User (ADMIN)		5	1	0	5
Run Analytics		10	2	1	30
Add Media		50 x2[1]	1	1	200
Update Media Info		10 x2[1]	1	1	40
Delete Media		5 x2[1]	1	1	20
Delete Review		20	1	0	20

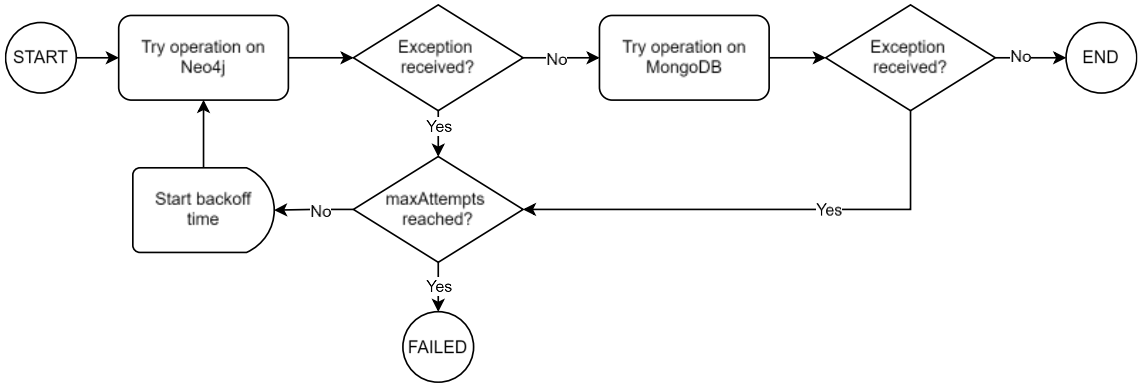
[1]With the consideration of having two types of media some operations must be considered twice

2.6.2. Sharding

We thought carefully about how to shard our data and decided to use document IDs as the shard key, along with hashing as the partition algorithm. This approach ensures that the data is distributed evenly across shards and avoids performance bottlenecks. We also considered using other fields. However, most searches are based on document IDs, so using another field would have added an extra step to find the sharding key. This would have made queries slower and unnecessarily complicated without providing real benefits.

2.6.3. Handling the inter database consistency

Having two distinct types of databases involves the presence of redundant data, this leads to the need of a control over the operations that interact with data in both databases, such as user registration or updates in the media information, handling the consistency between them. This consistency is managed using retriable spring functions: the operation is firstly tried on Neo4j DB, the most likely to fail having only one replica node, if the operation succeed then the operation occurs in MongoDB and if there are no issues the function returns a success response to the user. If any of the two phases fails the operation is retried for a limited number of times (3) after a fixed backoff time. If the number of retries reaches a fixed limit, then the operation will fail and return an error response to the user. This setting implies that if the operation in MongoDB fails there are no rollbacks in the Neo4j DB, leading to an inconsistency in the database. However, the chances of this case occurring are very slim due to the presence of three nodes handling this database replicas. When the operation fails on MongoDB the retry starts from the beginning of the function retrying also the operation on Neo4j, this will not be an issue having the write function as idempotent.



3. Dataset and Data Retrieval

The focus of this application is handling large scale of data and managing them in the best way possible, to do so it is mandatory to procure the raw material itself: data.

For this reason, we opted for the retrieval of what we needed from one of the most famous web sites in the field of anime and manga catalogues providers: MyAnimeList. This allowed us to obtain all data we needed for our application, starting from users' general information, their lists with the score associated with the media and the media information themselves. With this objective we wrote a script to gather, process, and organize user, manga and anime data into a final dataset suitable for storage and further analysis. Given the high number of resources needed and the time required, we could retrieve only a little fraction of the entire database present. This lack of completeness has been considered when creating the final databases assuring the consistency of the data, if a user has a media in one of the lists or a score associated to it, the media will be present in the database.

The initial step involves retrieving club (a community in the web application) information from the website, allowing us to retrieve users with similar tastes generating a more interesting dataset. Next, we focus on gathering usernames from the clubs using the Jikan API, an unofficial and open-source API for the MyAnimeList. It iterates through the club ids, fetching member usernames and saving them to a file. The usernames are then shuffled and assigned new IDs, creating a users.csv file that maps usernames to unique user Ids.

For both media data, we retrieve the lists for each user exploiting the official MyAnimeList API. We fetch users' lists details saving them in individual user files. Additionally, we compile a list of unique media IDs from the users' lists, ensuring that all relevant media are included in the final dataset. At the end we retrieve detailed information for each unique media. This detailed media data is saved to a csv file for each type of media (one for manga and another for anime). At the end we also retrieve reviews for both manga and anime, scraping reviews from the Jikan API and saving them to individual files, processing the reviews and scores and preparing them for inclusion in the final dataset. The followers' array in the users' information has been generated randomly by sampling a random number of users (between 0 and 50).

Finally, we combine all retrieved data into a cohesive dataset suitable for MongoDB import, processing manga and anime files, calculating the number of scores and sum of scores and serializing reviews as JSON. The same operation is repeated for what needed in the Neo4j database, creating a csv file for each type of node and another file for every kind of relationship between them.

At the end of the script we obtain the following files for mongo import:

- *users_collection.csv* 828 kB
- *anime_collection.csv* 13 MB

- *manga_collection.csv* 14MB

The files obtained for the creation of the database with Neo4j are the following:

- *user_nodes.csv* 76 kB
- *anime_nodes.csv* 1.3 MB
- *manga_nodes.csv* 2 MB
- *follow_relationship.csv* 1.4 MB
- *anime_relationship.csv* 16.4 MB
- *manga_relationship.csv* 3.1 MB

4. Implementation

My Akiba system has been implemented following all the requirements and design listed and using the most recent technologies available. The application is implemented using the Spring framework, while using the Controller-Service-Repository structure to organise the code.

4.1. Spring

The project has been built using the Spring framework for the initialisation and to establish the connection between the server and the databases. It is a tool that simplifies and accelerates the development of web applications and microservices through three main features: automatic configuration, the adoption of a declarative approach to configuration, and the ability to create standalone applications.
Another useful feature used of the Spring framework is the Spring Security token management to allow a more fluid and persistent login of the users to the system, handling the registration and the login via a JWT (JSON Web Token), to be included in any further request's header once logged. This security management is handled with a filter chain, allowing some requests to proceed without checking the authorisation if it's not needed (registration, login and browsing media), and using the token provided in the header to permit the others. This implementation allows to perform the security check and authorization management requested in the non-functional requirements.

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(AbstractHttpConfigurer::disable) // Disable CSRF protection
            .httpBasic(AbstractHttpConfigurer::disable) // Disable HTTP Basic authentication
            .formLogin(AbstractHttpConfigurer::disable) // Disable form login
            .authorizeHttpRequests(request -> request
                    .requestMatchers("/api-docs.html", "/swagger-ui/**", "/v3/api-docs/**").permitAll()
                    .requestMatchers("/api/auth/**").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/media/{mediaType:[a-zA-Z]+}", "/api/media/{mediaType:[a-zA-Z]+}/{mediaId:[a-zA-Z0-9\\-]+}").permitAll()
                    .requestMatchers("/api/users").authenticated() // Only authenticated users can access this endpoint
                    .requestMatchers("/api/admin/**").hasRole("ADMIN")
                    .anyRequest().hasRole("USER"))
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

4.2. Model

The models for this application are divided into two main types: the MongoDB models and the Neo4j ones.

4.2.1. MongoDB Models

User Model

```java
public class UserMongo {
    @Id
    private String id;

    @NotBlank(message = "Role cannot be blank")
    private String role;

    @NotBlank(message = "Username cannot be blank")
    @Indexed(unique = true)
    private String username;

    @NotBlank(message = "Password cannot be blank")
    private String password;

    @NotBlank(message = "Email cannot be blank")
    @Email(message = "Email should be valid")
    @Indexed(unique = true)
    private String email;

    @NotBlank(message = "Privacy status cannot be blank")
    private PrivacyStatus privacyStatus;

    private List<String> followers;

    @Past(message = "Birthdate must be in the past")
    private Date birthdate;

    private Date createdAt;
}
```

User Principal Model

This class implements the native user class used by Spring Security for the management of authorizations.

```java
public class UserPrincipal implements UserDetails {

    private final UserMongo user;

    public UserPrincipal(UserMongo user) { this.user = user; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        System.out.println("ROLE_" + user.getRole().toUpperCase());
        return Collections.singleton(() -> "ROLE_" + user.getRole().toUpperCase());
    }

    @Override
    public String getPassword() { return user.getPassword(); }

    @Override
    public String getUsername() { return user.getEmail(); }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }

    public UserMongo getUser() { return user; }
```

Media Model

This abstract class implements the abstract entity of the media including all common attributes between the two types.

```java
public abstract class MediaMongo {
    @Id
    private String id;

    @NotBlank
    private String name;

    @NotBlank
    private MediaStatus status;

    @NotBlank
    private int sumScores = 0;

    private int numScores = 0;

    @NotBlank
    private List<String> genres;

    @NotEmpty
    private String type;

    @NotEmpty
    private String synopsis;

    private List<ReviewDto> reviews;
}
```

Anime Model

```java
public class AnimeMongo extends MediaMongo{
    @NotBlank
    private int episodes;

    private String source;

    private double duration;

    private List<String> studios;
}
```

Manga Model

```java
public class MangaMongo extends MediaMongo {
    @NotBlank
    private int chapters;

    private List<String> authors;
}
```

Month Analytics Model

```java
public class MonthAnalytic {
    @Id
    private String id;

    @Indexed(unique = true)
    private int year;

    @NotBlank
    private int month;

    private int count = 0;
}
```

4.2.2. Neo4j Models

User Model

```java
public class UserNeo4j {

    @Id
    private String id;

    @Property("username")
    private String username;

    @Property("privacyStatus")
    private PrivacyStatus privacyStatus;
}
```

Anime Model

```java
public class AnimeNeo4j {

    @Id
    private String id;

    @Property("name")
    private String name;

    @Property("status")
    private MediaStatus status;

    @Property("episodes")
    private int episodes;

    @Property("genres")
    private List<String> genres;
}
```

Manga Mongo

```java
public class MangaNeo4j {

    @Id
    private String id;

    @Property("name")
    private String name;

    @Property("status")
    private MediaStatus status;

    @Property("chapters")
    private int chapters;

    @Property("genres")
    private List<String> genres;
}
```

4.3. Repository

The repository modules extended by Spring Data classes provide an abstraction layer for database operations, simplifying the implementation of the API. In our application we use repositories for both MongoDB and Neo4j using simple queries for the databases management and users' requests and more complex ones for analytics and recommendation functionalities, these last ones will be taken up and explored in depth in the next chapter.

4.4. Service

The service layer acts as an intermediary between the controller and repository layers in the implementation of an application, representing its core functionalities and main business logic implementations. It encapsulates business logic and orchestrates interactions with repositories, ensuring a clean separation of concerns. Additionally, the service modules allow better management of transactions and error handling, making the system more robust and maintainable.

Auth Service

This module implements the main functionalities regarding user registration and login. It verifies user credentials, generates JWT tokens upon successful login, and ensures secure password storage through encryption. During registration, it creates new user entries in both MongoDB and Neo4j, ensuring consistency across data layers.

User Service

This module implements all the requirements related to the user, starting from his own profile, his lists and the social relationship with those he follows. This service manages user-related operations. It supports user CRUD operations such as retrieving, updating, and deleting user data. Additionally, it handles user lists (media tracking), enabling users to add, modify, or remove media. It also manages social features like following or unfollowing users and fetching followers or following lists while respecting privacy settings.

MyUserDetails Service

The MyUserDetailsService module combined with the UserPrincipal model integrates with Spring Security to load user details for authentication purposes. It retrieves users either by email or ID from MongoDB and wraps them in a UserPrincipal for use within the security context.

Media Service

This service manages CRUD operations for media entities, including anime and manga. It supports searching and browsing media with pagination, retrieving detailed information, adding new media entries, updating existing ones, and deleting media. Additionally, it enables users to post and delete reviews, ensuring media ratings are updated dynamically.

Analytics Service

This module provides the Administrator functionalities for generating insights and analytics from the data. It includes methods to identify trends, such as the month with the most registrations, controversial or trending media, and media with improving or declining popularity. It also analyses user interactions to determine influential users and provides statistics on media appearances in user lists.

Recommendation Service

This service provides the users useful information and focuses on generating personalized recommendations. It identifies users with similar tastes, suggests popular media among a user's follows, and provides top-rated media based on genres and user interactions.

4.5. Controller

Auth Controller

This module handles user authentication processes, including registration and login. It validates user input during registration, securely creates new user accounts, and allows users to log in by providing credentials. This controller uses the Auth Service to manage authentication tasks.

User Controller

This controller maps user-related operations such as browsing users, retrieving user details, updating user information, and deleting users. It also handles user media lists, allowing users to add, modify, and remove media from their lists, as well as managing user follow and unfollow actions. This controller interacts with the User Service to comply with all requests.

Media Controller

This controller handles operations related to the media catalogue mapping all HTTP requests directed to the media objects. This controller requires the Media Service to provide the needed output to the user.

Admin Controller

The Admin Controller provides administrative functionalities by mapping the Admin HTTP requests, including adding, updating, and deleting media, as well as deleting reviews. It also allows administrators to retrieve user details and access various

analytics. This controller connects with the User, Media and Analytics Services to provide a quick response to the requests.

Recommendation Controller

This module relies on the Recommendation Service to map the users' requests concerning recommendation functionalities.

4.6. Exception handler

Our application includes a Global Exception Handler. By centralizing exception handling logic we can make the code cleaner and more maintainable, assuring also a better management and collection of all possible exceptions. It ensures consistent error responses across the application and reduces the need for repetitive try-catch blocks in individual controllers.

4.7. RESTful API Endpoints

Auth Endpoints

- POST /api/auth/register
- POST /api/auth/login

User Endpoints

- GET /api/users
- GET /api/user
- GET /api/user/{id}
- PATCH /api/user/{id}
- DELETE /api/user/{id}
- GET /api/user/lists/{mediaType}
- GET /api/user/{userId}/lists/{mediaType}
- POST /api/user/lists/ {mediaType}/{mediaId}
- PATCH /api/user/lists/ {mediaType}/{mediaId}
- DELETE /api/user/lists/ {mediaType}/{mediaId}
- GET /api/user/followers
- GET /api/user/{userId}/followers
- GET /api/user/following
- GET /api/user/{userId}/following
- POST /api/user/follow/{userId}
- DELETE /api/user/follow/{userId}

Media Endpoints

- `GET /api/media/{mediaType}`
- `GET /api/media/{mediaType}/{mediaId}`
- `POST /api/media/{mediaType}/{mediaId}/review`
- `DELETE /api/media/{mediaType}/{mediaId}/review/{reviewId}`

Admin Endpoints

- `POST /api/admin/media`
- `PATCH /api/admin/media/{mediaId}`
- `DELETE /api/admin/media/{mediaType}/{mediaId}`
- `DELETE /api/admin/reviews/{mediaType}/{mediaId}/review/{reviewId}`
- `GET /api/admin/user/{userId}`
- `GET /api/admin/analytics/monthlyregistrations`
- `GET /api/admin/analytics/controversial/{mediaType}`
- `GET /api/admin/analytics/declining/{mediaType}`
- `GET /api/admin/analytics/improving/{mediaType}`
- `GET /api/admin/analytics/scc`
- `GET /api/admin/analytics/influencers`
- `GET /api/admin/analytics/listcounter/{mediaType}`
- `GET /api/admin/analytics/mediainlists/{mediaType}/{mediaId}`

Recommendation Endpoints

- `GET /api/recommendations/similar-users`
- `GET /api/recommendations/popular-among-follows/{mediaType}`
- `GET /api/recommendations/top10media/{mediaType}`

5. Most relevant queries

5.1. MongoDB queries

Most relevant queries in mongoDB are implemented using aggregation pipelines. The first three queries are for analytics, while the last one is for recommendations.

Month with most registrations for each year

For each year, the admin can see the month with most user registrations. To speed up computation, data of past years are saved in the *month_analytics* collection. The service method finds the last year present in the collection and calculates only the data yet to be computed. The result of the aggregation is then saved in the collection and all the documents inside are returned.
The aggregation does the following:

1. Match: keep only users of years that weren't already computed.
2. Group: group by year and month, count the total users for each month.
3. Sort: sort by year and users count.
4. Group: group by year and keep the month with most registrations.

```
//   For each year, see the month with most registrations
public List<MonthAnalytic> getMonthlyRegistrations() {
    MonthAnalytic maxDocument = monthAnalyticRepository.findTopByOrderByYearDesc();
    int lastYearCalculated = maxDocument != null ? maxDocument.getYear() : 2000;
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR, lastYearCalculated);
    cal.set(Calendar.DAY_OF_YEAR, 1);
    Date firstDay = cal.getTime();
    List<MonthAnalytic> results = userMongoRepository.findMaxMonthByYearGreaterThan(firstDay);
    monthAnalyticRepository.saveAll(results);
    return monthAnalyticRepository.findAllByOrderByYear();
}
@Aggregation(pipeline = {
        "{ '$match': { 'createdAt': { '$gte': ?0 } } }",
        "{ '$group': { '_id': { 'year': { '$year': '$createdAt' }, 'month': { '$month': '$createdAt' } }, 'count': { '$sum': 1 } } }",
        "{ '$sort': { '_id.year': 1, 'count': -1 } }",
        "{ '$group': { '_id': '$_id.year', 'maxMonth': { '$first': { 'month': '$_id.month', 'count': '$count' } } } }",
        "{ '$project': { '_id': 0, 'year': '$_id', 'month': '$maxMonth.month', 'count': '$maxMonth.count' } }"
})
List<MonthAnalytic> findMaxMonthByYearGreaterThan(Date year);
```

Most controversial media for each genre

An admin can be interested in finding the most controversial media, that is, anime or manga with most variance in the scores.

1. AddFields: calculate the variance of the scores.
2. Unwind: unwind genres array.
3. Sort: sort by genres and variance.
4. Group: group by genre and keep the media with highest variance.

```
@Aggregation(pipeline = {
        "{ '$addFields': { 'variance': { '$pow': [{ '$stdDevPop': '$reviews.score' }, 2] } } }",
        "{ '$unwind': '$genres' }",
        "{ '$sort': { 'genres': 1, 'variance': -1 } }",
        "{ '$group': { " +
                "    '_id': '$genres', " +
                "    'anime': { '$first': { 'id': '$_id', 'name': '$name', 'variance': '$variance' } } " +
                "} }",
        "{ '$project': { '_id': 0, 'genre': '$_id', 'id': '$anime.id', 'name': '$anime.name' } }"
})
List<ControversialMediaDto> findTopVarianceAnime();
```

Top declining/improving media

Declining or improving means that the media is going worse or better than usual. To do so, the average score of the 5 most recent reviews is compared with the total average score.

1. AddFields: calculate the total average score and keep the 5 most recent reviews.
2. AddFields: calculate recent average score.
3. Match: keep media with a declining/improving average score.
4. AddFields: calculate the average score difference to return.
5. Sort: sort by this difference.
6. Limit: keep only the top 10.

```
@Aggregation(pipeline = {
        "{ '$addFields': { " +
                "    'averageScore': { '$cond': { " +
                "        if: { '$gt': ['$numScores', 0] }, " +
                "        then: { '$divide': ['$sumScores', '$numScores'] }, " +
                "        else: 0 " +
                "    } }, " +
                "    'reviews': { '$slice': [ { '$sortArray': { 'input': '$reviews', 'sortBy': { 'timestamp': -1 } } }, 5 ] } " +
        "} }",
        "{ '$addFields': { 'recentAverageScore': { '$avg': '$reviews.score' } } }",
        "{ '$match': { '$expr': { '$lt': ['$recentAverageScore', '$averageScore'] } } }",
        "{ '$addFields': { 'scoreDifference': { '$subtract': ['$averageScore', '$recentAverageScore'] } } }",
        "{ '$sort': { 'scoreDifference': -1 } }",
        "{ '$limit': 10 }",
        "{ '$project': { '_id': 0, 'id': '$_id', 'name': '$name', 'scoreDifference': 1 } }"
})
List<TrendingMediaDto> findTopDecliningAnime();
```

Top 10 media by average score

A user can be interested in knowing the top 10 media overall or for each genre.

1. Match: if a genre is specified, keep only media that have it in the genres array.
2. AddFields: calculate the average score.
3. Sort: sort by this.
4. Limit: keep only the top 10.

```
@Aggregation(pipeline = {
        "{ '$match': { '$expr': { '$or': [ { '$eq': [?0, null] }, { '$in': [?0, '$genres'] } ] } } }",
        "{ '$addFields': { 'averageScore': { '$cond': { if: { '$gt': ['$numScores', 0] }, then: { '$divide': ['$sumScores', '$numScores'] }, else: 0 } } } }",
        "{ '$sort': { 'averageScore': -1 } }",
        "{ '$limit': 10 }",
        "{ '$project': { 'id': 1, 'name': 1, 'averageScore': 1 } }"
})
List<MediaAverageDto> findTop10Anime(String genre);
```

5.2. Neo4j queries

The graph is mostly used for recommendations. However, it can give to the admin insight about followers and media lists relationships.
Most of the analytics are graph traversals, but since they are invoked sporadically by the admin, they don't degrade the performance of the system.

Get users with similar tastes

Domain-specific query: given a user, find users that have similar lists. That is, that have a lot of media in common.

Graph-centric query: given a User node, find other User nodes with the highest Jaccard similarity.

To find the similarity between users, we used the Graph Data Science (GDS) plugin. To speed up the computation, we project only the nodes and relationships that the plugin method will use.

```
@Query("""
        MATCH (u:User {id: $userId})-[:LIST_ELEMENT]->(target)<-[:LIST_ELEMENT]-(other:User)
        WITH collect(u) + collect(other) AS sourceNodes, collect(target) AS targetNodes
        CALL gds.graph.project(
          'myGraph',
          {
            User: {
              label: 'User'
            }
          },
          {
            LIST_ELEMENT: {
              type: 'LIST_ELEMENT'
            }
          }
        )
        YIELD graphName

        CALL gds.nodeSimilarity.stream('myGraph')
        YIELD node1, node2, similarity
        WITH gds.util.asNode(node2) AS user1, similarity
        WHERE gds.util.asNode(node1).id = $userId
        RETURN user1.id AS id, user1.username AS username, similarity
        ORDER BY similarity DESC
        LIMIT 10
        """)
List<UserIdUsernameDto> findUsersWithSimilarTastes(String userId);
```

Get popular media among followed users

Domain-specific query: given a user, find most present media in followed users' list.

Graph-centric query: given a User node, find most present Anime or Manga nodes connected to followed User nodes.

```
@Query("""
        MATCH (user:User {id: $userId})-[:FOLLOW]->(f:User)-[:LIST_ELEMENT]->(media)
        WHERE ((media:Anime AND $mediaType = 'ANIME') OR (media:Manga AND $mediaType = 'MANGA'))
              AND f.privacyStatus <> 'NOBODY'
        RETURN media.id AS id, media.name AS name, count(media.id) AS count
        ORDER BY count DESC
        LIMIT 10
        """)
List<MediaIdNameDto> findPopularMediaAmongFollows(String mediaType, String userId);
```

Find influencers (graph traversal)

Domain-specific query: find most followed users.

Graph-centric query: calculate the in-degree centrality for each User node, using FOLLOW edges.

As a measure of centrality, we decided to use the in-degree centrality, which is known to be a simple but good indicator of influence.

```
@Query("""
        MATCH (u:User)<-[:FOLLOW]-(f:User)
        WITH u, count(f) AS followersCount
        ORDER BY followersCount DESC
        LIMIT 20
        RETURN u.id AS userId, u.username AS username, followersCount
        """)
List<InfluencersDto> findMostFollowedUsers();
```

Find Strongly Connected Components (graph traversal)

Domain-specific query: find users' community.

Graph-centric query: using FOLLOW edges, apply a community detection algorithm to find well connected User nodes.

We define a community as a set of Strongly Connected Component (SCC). From the Graph Data Science documentation: *"The Strongly Connected Components (SCC) algorithm finds maximal sets of connected nodes in a directed graph. A set is considered a strongly connected component if there is a directed path between each pair of nodes within the set."*

```
@Query("""
        MATCH (source:User)-[:FOLLOW]->(target:User)
        WITH collect(source) AS sourceNodes, collect(target) AS targetNodes
        CALL gds.graph.project(
          'graph',
          ['User'],
          {
            FOLLOW: {
              type: 'FOLLOW'
            }
          }
        )
        YIELD graphName

        CALL gds.scc.stream('graph', {})
        YIELD componentId, nodeId
        WITH componentId, collect(gds.util.asNode(nodeId)) AS users
        WHERE size(users) > 1
        RETURN componentId,
               size(users) AS componentSize,
               [user IN users | {id: user.id, username: user.username}] AS userDetails
        ORDER BY componentSize DESC
        """)
List<SCCAnalyticDto> findSCC();
```

Media present more in each list (graph traversal)

Domain-specific query: for each media list, find the 10 most present media.

Graph-centric query: categorize each LIST_ELEMENT in a list type. Then, for each Anime or Manga node count the list of different types.

```
@Query("""
        MATCH (user:User)-[relationship:LIST_ELEMENT]->(anime:Anime)
        WITH anime, relationship,
             CASE
                 WHEN relationship.progress = 0 THEN 'PLANNED'
                 WHEN relationship.progress = anime.episodes AND anime.status = 'COMPLETE' THEN 'COMPLETED'
                 ELSE 'IN_PROGRESS'
             END AS listType
        WITH  anime, listType, count(DISTINCT relationship) AS listCount
        ORDER BY listType, listCount DESC
        WITH listType, collect({id: anime.id, name: anime.name, count: listCount})[0..10] AS topMedia
        RETURN listType, topMedia
        """)
List<ListCounterAnalyticDto> findListCounters();
```

How many times a media is in each list

Domain-specific query: given a media, find in how many lists it is, for each list type.

Graph-centric query: given a Manga or Anime node, categorize the incoming LIST_ELEMENT edges.

```
@Query("""
        MATCH (user:User)-[relationship:LIST_ELEMENT]->(anime:Anime)
        WITH anime, relationship,
             CASE
                 WHEN relationship.progress = 0 THEN 'PLANNED'
                 WHEN relationship.progress = anime.episodes AND anime.status = 'COMPLETE' THEN 'COMPLETED'
                 ELSE 'IN_PROGRESS'
             END AS listType
        WITH  anime, listType, count(DISTINCT relationship) AS listCount
        WITH anime, collect({listType: listType, listCount: listCount}) AS appearances
        RETURN anime.id AS mediaId, anime.name AS mediaName, appearances
        """)
List<MediaInListsAnalyticDto> findAnimeAppearancesInLists();
```

5.3. Indexes

5.3.1. MongoDB indexes

In this paragraph, we discuss the possible indexes that could improve the performance of read operations for each collection. Afterward, each index will be taken into analysis to decide whether it should be added.

User collection

- **Email**
  This index is fundamental to speed up the login and the registration.
- **Username**
  This index would help with the registration, but most importantly it would improve the performance of users browsing, which uses a regex on the username. To confirm our decision to add this index, we performed a test with and without the index while browsing users with a regex.

	Without index	**With index**
nReturned	1	1
executionTimeMillis	23	13
totalKeysExamined	0	3510
totalDocsExamined	3510	1

- **CreatedAt**
  This index would speed up the computation of the *"Month with most registrations for each year*" aggregation. Moreover, without this index, the *month_analytics* collection would be useless, since all the users documents must be scanned for the aggregation. To confirm our decision to add this index, we performed a test with and without the index while executing the aggregation.

The aggregation was executed with `{'createdAt':{'$gte': ISODate("2020-01-01T00:00:00.000Z")}}`.

	Without index	**With index**
nReturned	60	60
executionTimeMillis	123	10
totalKeysExamined	0	1029
totalDocsExamined	3510	0

- **Followers**
  This index would improve the check on followers when the privacyStatus is on FOLLOWERS. Nevertheless, it would slow down the follow and unfollow write operation.
  Since there are 1500 find on a user, of which only about a third will be with privacyStatus=FOLLOWERS, and since there are 4000 follow and unfollow daily, it is not worth adding the index.

Anime and Manga collections

- **Name**
  As the index on the user's name, this index would speed up the browse on anime or manga. Performance tests are similar to the ones on users' name.

5.3.2. Neo4j indexes

Since all neo4j queries that compute starting from a certain node (User, Anime or Manga) find it using the *id* property, we created indexes for the id field for each type of node.