UNIVERSITÀ DI PISA

OkiboAnime

Project of Large Scale and Multi Structured Databases

Marco Galante

Lorenzo Guidotti

Daniel Deiana

Index

Link to GitHub

The code is available on GitHub at this link: https://github.com/daniel-deiana/lsmsd-project

Introduction

In this section we show our application idea. We also talk about how we gathered all data used to build the prototype.

Application highlights

OkiboAnime is an application that allows users to explore the anime world. They can review anime and get in touch with other users following them and viewing their Top10 list. That's because every user can open a daily pack in which he could find cards from characters that he reviewed. In our application, we mainly guarantee:

- An anime page in which user can find anime, see reviews and characters. They have also suggestions on anime to see based on followed users' tastes.
- A shop page in which user can see which characters can find and in which he/her can open a daily pack and find new cards.
- Some personal pages in which user can see his/her cards, his data and suggestion on possible friends

Regarding administrators, they have access to analysis regarding user activities.

Dataset creation

We downloaded the initial dataset from **Kaggle**, then we scraped some information on **Anime Planet** to enrich data and to allow some operations.

Anime

Regarding anime, we selected a pool of 458 Anime, and we scraped some information about them.

Review

Regarding reviews, since they have so long texts in which users indicated some integer scores based for example on Story or Characters, we considered only the text and an overall score on the anime. Since there were not the date in which they were wrote, we added a datetime to make some analytics. We retrieved a totality of about 18’000 reviews.

Users

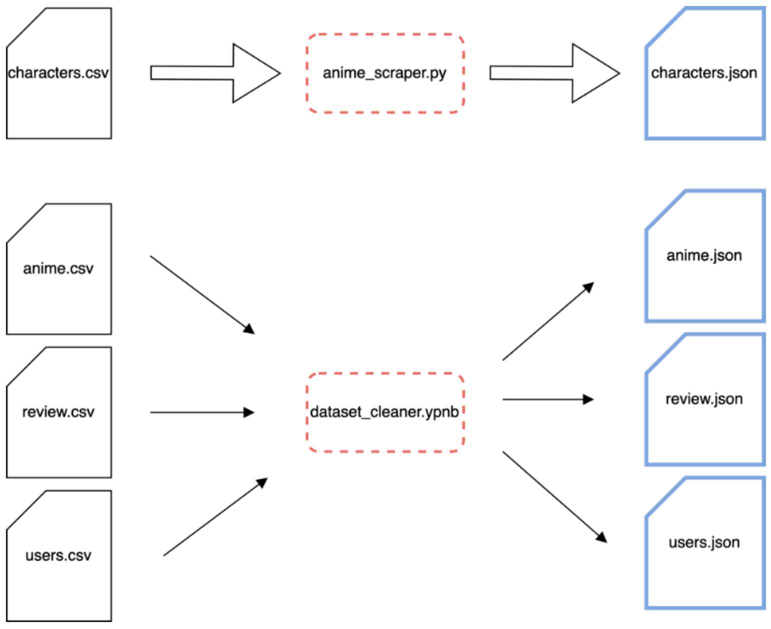
Regarding users, we considered only those ones who had already done some reviews. Then we generate some information like password and country. Totally we have about 10’000 users.

Characters

Regarding characters, we selected some of the most loved characters from AnimePlanet considering our Anime. We retrieved about 2’000 characters.

Useful scripts and functions

Data scraped was not ready for the construction of databases, so we decided to apply some changes in order to build related data. The scraping in this case was done to associate each character to an anime, since in the dataset that we used this information wasn’t stored from the start. We also scraped the image URLs for each character from anime planet in order to show them in the anime page.



Design

Main actors

We have 3 main actors in the application:

- **Unregistered User**.
- **Logged User.**
- **Administrator.**

The first one can only Register or Log in the application if already registered. So, no content is showed to a user who hasn't registered yet.

The logged user is the main actor of the application. After logging he can do all sort of stuff apart from the viewing analytics and doing some few CRUD operation on the dataset.

The administrator has the main purpose of viewing the analytics about the application and inserting some entities.

We also had to create ourselves all the data regarding user followings, user own top 10 of characters and the data regarding the characters "cards" owned by each user. In this case we made the relationships using a python script that generated random characters for each user to be inserted in the top 10 or in the **owned characters** list.

Requirements

Here are the functional requirements of the application:

Functional Requirements

Functional Requirements – Unregistered User

- The system must allow an unregistered user to become a registered one by registering his/her full name, nickname, email address, password, birthday and favorite anime.

Functional Requirements – User

- The system must allow a user to login.
- The system must allow a user to know the most/worst appreciated and long anime
- The system must allow a user to view suggestions about anime to see.
- The system must allow a user to find a specific anime.
- The system must allow a user to display the information of a selected anime.
- The system must allow a user to make a review of an anime.
- The system must allow a user to display his album (the cards that he/she has found).
- The system must allow a user to find a specific card in the album by its name.
- The system must allow a user to add/remove a character he/she owns to the Top-10 list.
- The system must allow a user to open a daily pack and find characters
- The system must allow a user to display some cards that he/she could find
- The system must allow a user to view his/her page with data and information
- The system must allow a user to find another user in the system through his/her username and view his/her information
- The system must allow a user to follow/unfollow another user
- The system must allow a user to view suggested users
- The system must allow a user to display the top-10 list of his/her followed user.
- The system must allow a user to logout.

Functional Requirements – Admin

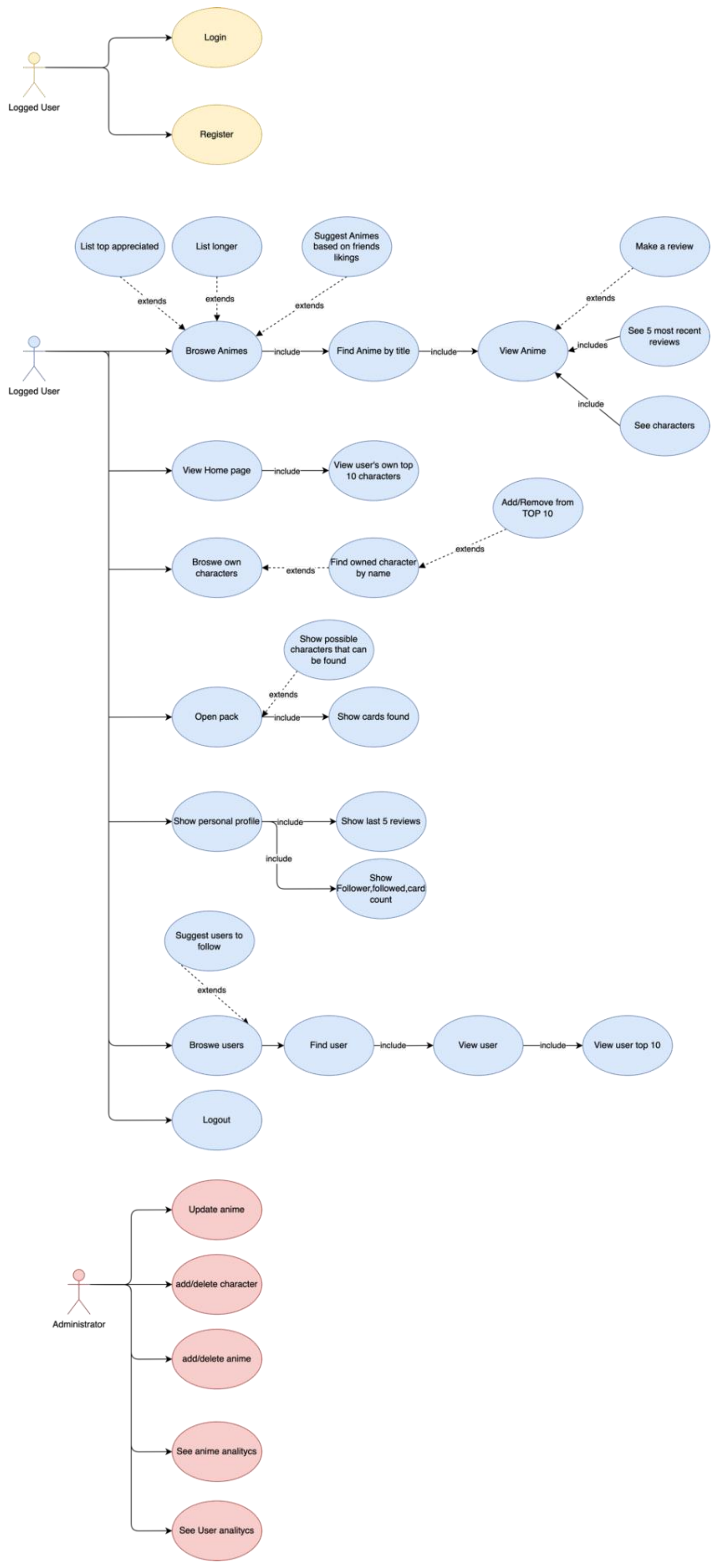
- The system must allow an admin to add/update an anime to the list.
- The system must allow an admin to add/remove a character card to the list.
- The system must allow an admin to delete a user from the list.
- The system must allow an admin to see analytics about users, reviews anime and their characters

Non-Functional Requirements

Here's a list of non-functional requirements for the application:

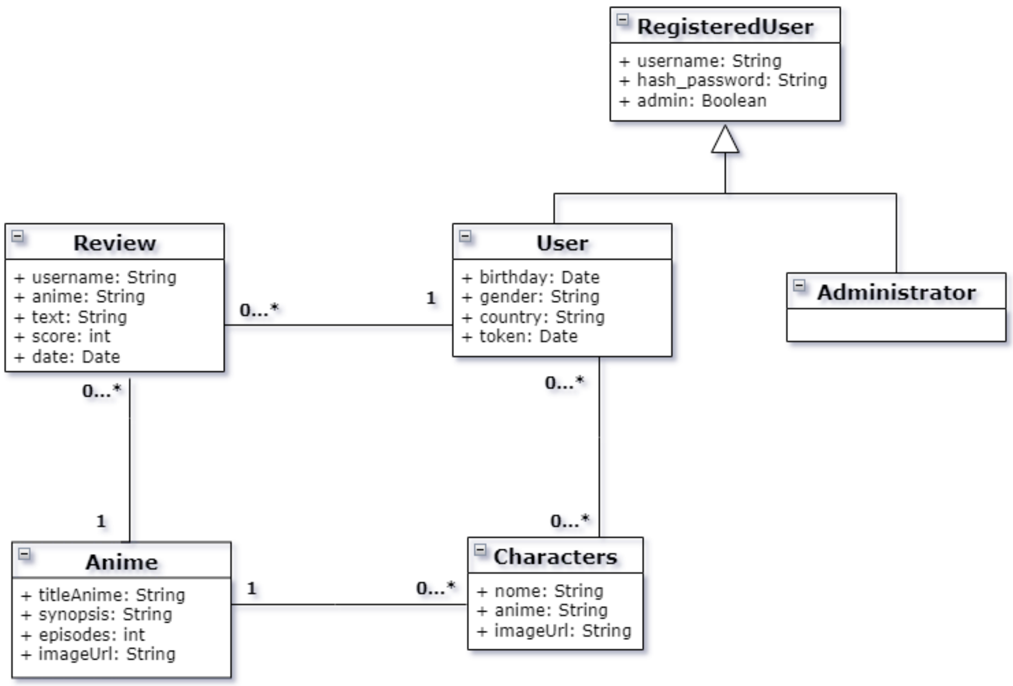
- The system must be a responsive web application.
- Tolerance to loss of the data.
- High availability for the data, at the cost of maybe showing old versions.
- The code must be modular and implemented using OOP paradigm.
- The system must be available to users 24/7
- Avoid, if possible, a single point of failure

Use-Case Diagram



UML Class Diagram

The UML diagram of the class in the application is the following:



Data Modelling

Databases Choices

We decide to use 2 types of databases, **MongoDB** and **Neo4j**. The Graph database was chosen to support the "social network" portion of the application, to make the suggestions of users and anime and to store the data regarding the owning of a character card by a user and the insertion of a character in the user's top 10.

Considerations on volumes

Here we made few assumptions on the volumes of the operations done by users. We made an estimated volume of users each day using the application to be around 10000.

Action	# of occurrences	MongoDB ops.	Neo4j ops.	Total (per day)
Loading Home	3	0	1	30000
Loading Anime	5	1	0	50000
Loading Anime Search	2	0	1	20000
Loading Profile	2	1	2	60000
Loading Friends Page	1	1	1	20000
Open a pack	1	1	1	20000

The nature of the application is a **read-intensive one**, we needed to retrieve in the faster way data to be show for the users such as reviews and the characters of an anime.

MongoDB document collections

We decided to store three collections inside MongoDB:

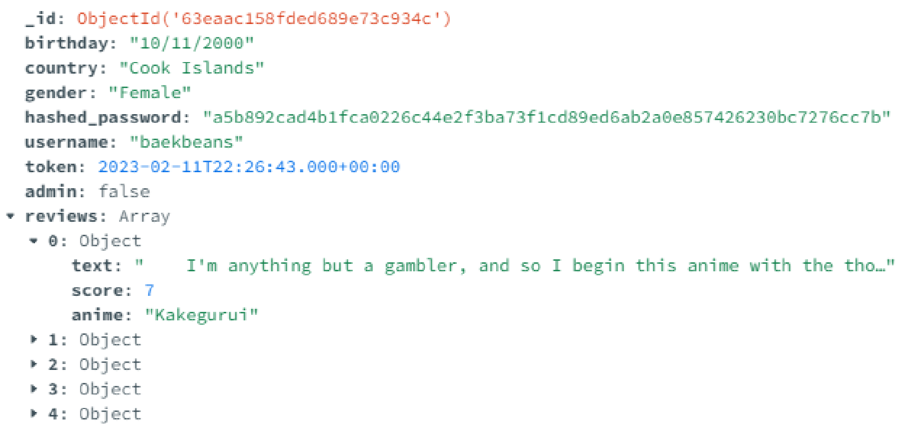
- Users
- Anime
- Reviews

We decided to handle the **one-to-many** relationships regarding the reviews associated to an anime and the characters associated by embedding documents. Those embeddings can be seen in the following sections for the collections of MongoDB.

Collection "Users"

This collection is used to store all the personal data about a user. All the data provided during the signup is stored inside the user collection. We maintain the password in a hashed format (we used the SHA256 algorithm since is a well-known standard). We also added a Boolean attribute to indicate if a user was an admin or not, and a date called *token* to ensure that a user cannot open more than a pack every day. We leave here an example of user document.

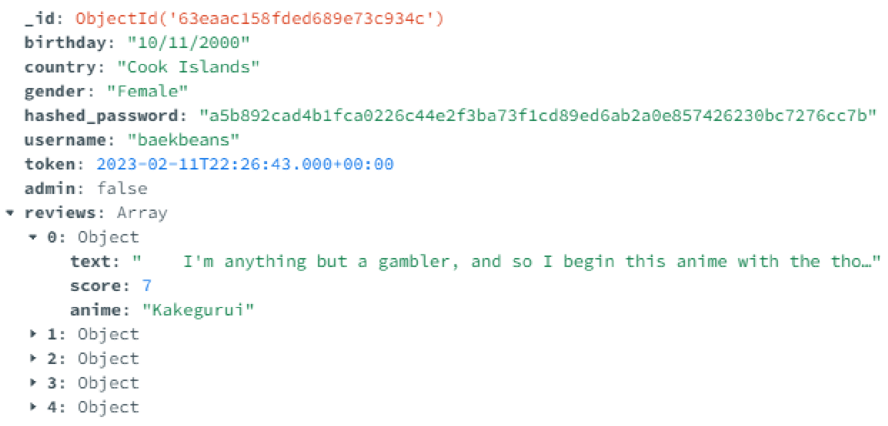
We decided to embed the **last 5 reviews** of a user because we thought it was interesting for a random navigator to see them to make a fast idea of the interests of another user. The embedding is useful to only query the user document to load the profile page **without joining data with other collections**.



```
  _id: ObjectId('63eaac158fded689e73c934c')
  birthday: "10/11/2000"
  country: "Cook Islands"
  gender: "Female"
  hashed_password: "a5b892cad4b1fca0226c44e2f3ba73f1cd89ed6ab2a0e857426230bc7276cc7b"
  username: "baekbeans"
  token: 2023-02-11T22:26:43.000+00:00
  admin: false
▾ reviews: Array
  ▾ 0: Object
      text: "    I'm anything but a gambler, and so I begin this anime with the tho…"
      score: 7
      anime: "Kakegurui"
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
```

Collection "Anime"

This collection stores data about the anime entity, here's an example:

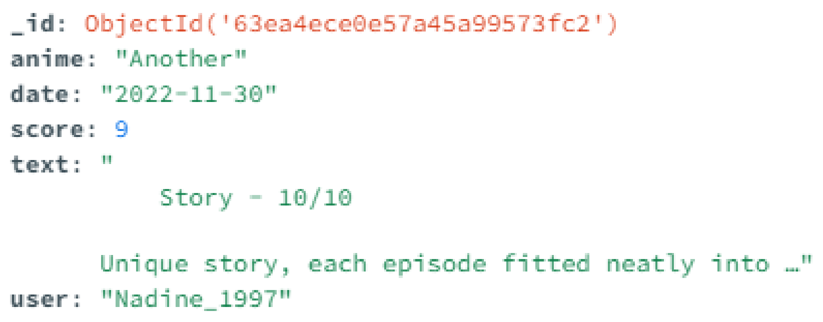


```
  _id: ObjectId('63eaac158fded689e73c934c')
  birthday: "10/11/2000"
  country: "Cook Islands"
  gender: "Female"
  hashed_password: "a5b892cad4b1fca0226c44e2f3ba73f1cd89ed6ab2a0e857426230bc7276cc7b"
  username: "baekbeans"
  token: 2023-02-11T22:26:43.000+00:00
  admin: false
▾ reviews: Array
  ▾ 0: Object
      text: "    I'm anything but a gambler, and so I begin this anime with the tho…"
      score: 7
      anime: "Kakegurui"
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
```

We decided to store a synopsis of the anime to give a fast idea on the nature of the opera, the number of episodes of the latter and a Boolean that signifies if an anime is finished or not plus the URL of an image representing the anime. Here we have the embedding. We embedded **all** the characters of an inside the anime to retrieve in a fast way all the characters inside it. We thought about the heaviness of doing so (storing all characters instead of storing a subset of them and then using another separate collection), and concluded that this reasoning was correct because of the fact that, **in general** the cardinality in this kind of relationship doesn't increase with the time (unlike the reviews for anime i.e.) and the fact that the embedded document has only 2 attributes (character name and image URL) so it's light to store and doesn't impact heavily on the size of the collection (unlike the reviews).

Collection "Reviews"

The Review collection stores data about anime reviews done by the users. here's an example:



```
_id: ObjectId('63ea4ece0e57a45a99573fc2')
anime: "Another"
date: "2022-11-30"
score: 9
text: "
          Story - 10/10

      Unique story, each episode fitted neatly into …"
user: "Nadine_1997"
```

We store data about the user that made a review, the date and the anime which the review is about and obviously the text of the review and the score given by the user. We decide only to embed a subset of this collection inside the user and the anime because reviews will likely increase in time (both the reviews on an anime and the reviews done by user) so we thought that making a separate collection was correct following this reasoning.

Concepts Handled by Graph Database

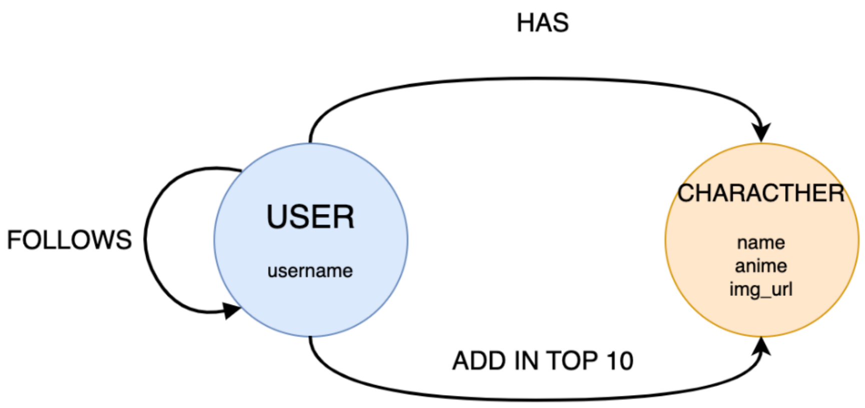
User

The user entity has got only the property of its **username**. This entity has a recursive relationship called FOLLOW (for implementing the friends' system in the application) and two other relationships oriented towards the **character** called "ADDTOTOP10" and "HAS".

Character

We stored the character in the Graph to implement the relationships with the user about the top 10 and the card owning. We decided to store in the character node the name of the character, the image URL, and the anime for suggestion purposes.

Here's a diagram of the graph relationships:



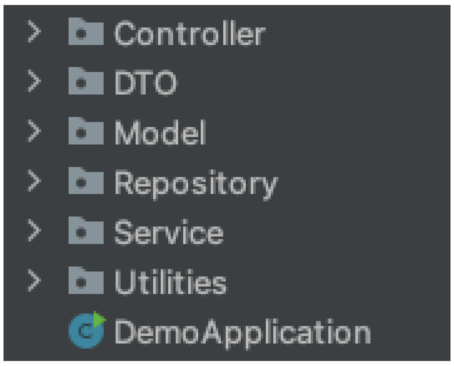
Examples of queries on the graph database

Domain-Specific	Graph-centric
Suggest the users (that you don't already follow) followed by your friends.	Which user vertexes have an incoming arc of type follow by the vertexes that I have an arc incoming to.
Suggest users by their top 10 list	Which user vertex have in incoming arc of type "ADDTOTOP10 " towards character vertexes in which the user vertex has incoming arcs of type "ADDTOTOP10"

Implementation

Frameworks

We used **BootStrap** to get the front-end done and developed the backend for the application in java, using the **spring framework** to handle the interaction between frontend and backend (using the Controller with the @**PostMapping**, @**GetMapping** annotations). and used the @**Autowired** notation to handle dependency injections. The structure of the application is divided in this way:



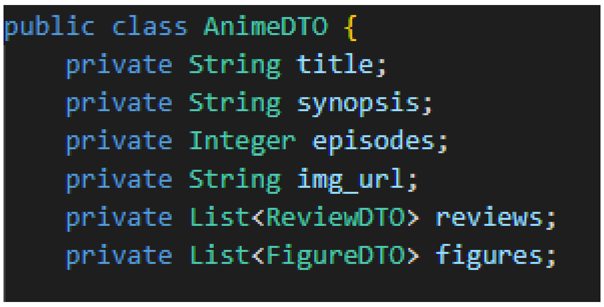
```
> Controller
> DTO
> Model
> Repository
> Service
> Utilities
  DemoApplication
```

Here is a brief description of the content of directories:

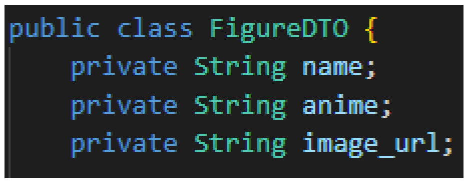
- **Controller:** We have controllers for the returning of the static templates (each for every page) and a sub-directory called **Api.** In the latter we have all the REST controllers which are responsible for returning data to the front end i.e. (the controller used to return the data for the anime data)
- **DTO:** in the DTO we have all the classes used to store the data that the repository classes return to the REST controllers.
- **Model**: This directory contains all the classes that we use to store data when we interact with the databases.
- **Service:** The service contains classes associated to the application entities (i.e. anime). those classes provide an interface between the REST controllers and the repository level.
- **Repository:** The repository contains all the classes with their methods used to interact with the databases.

DTO Examples

We provide some examples of the DTO classes (Figure and Anime):



```java
public class AnimeDTO {
    private String title;
    private String synopsis;
    private Integer episodes;
    private String img_url;
    private List<ReviewDTO> reviews;
    private List<FigureDTO> figures;
```

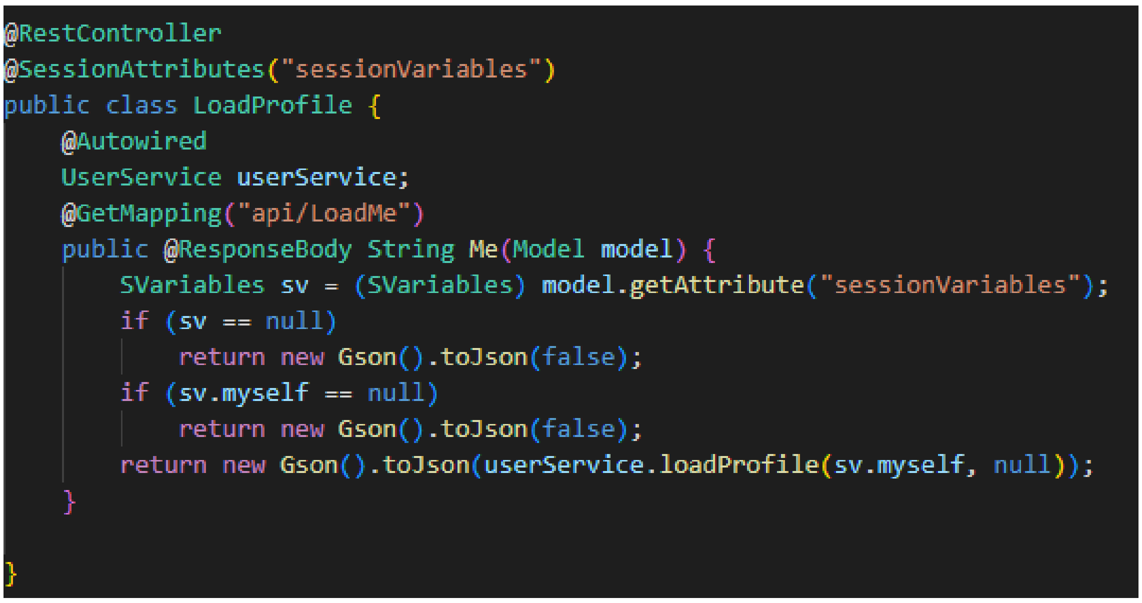


```java
public class FigureDTO {
    private String name;
    private String anime;
    private String image_url;
```

MongoDB Relevant operations

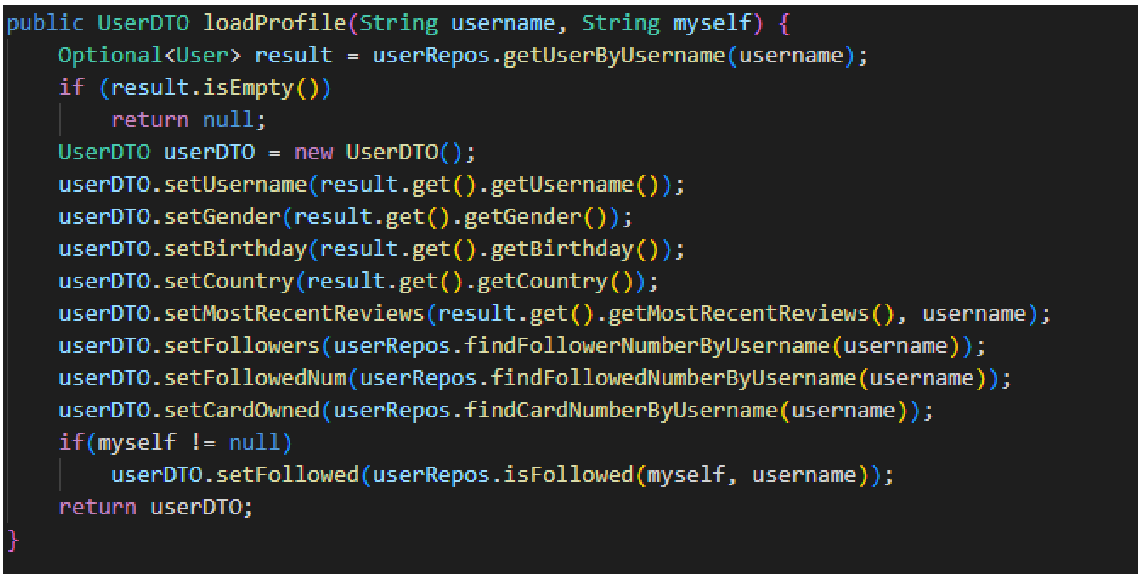
Loading profile page

We start from the controller which maps the ajax request with the URL "**api/LoadMe**".



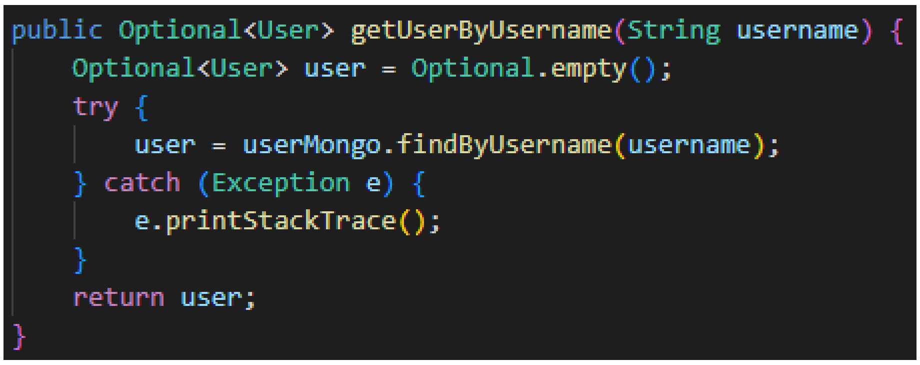
```java
@RestController
@SessionAttributes("sessionVariables")
public class LoadProfile {
    @Autowired
    UserService userService;
    @GetMapping("api/LoadMe")
    public @ResponseBody String Me(Model model) {
        SVariables sv = (SVariables) model.getAttribute("sessionVariables");
        if (sv == null)
            return new Gson().toJson(false);
        if (sv.myself == null)
            return new Gson().toJson(false);
        return new Gson().toJson(userService.loadProfile(sv.myself, null));
    }

}
```

This Controller interacts with the user service to load the profile data, the method "**loadProfile**" returns a **userDTO** instance. **LoadProfile** interacts with the repository level by calling "**getUserByUsername**".

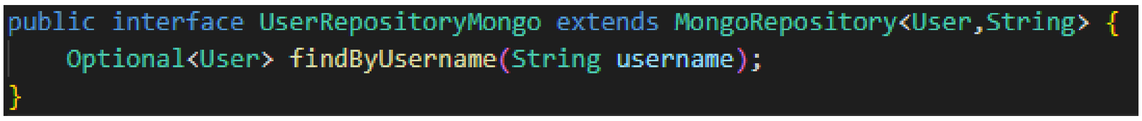


```java
public UserDTO loadProfile(String username, String myself) {
    Optional<User> result = userRepos.getUserByUsername(username);
    if (result.isEmpty())
        return null;
    UserDTO userDTO = new UserDTO();
    userDTO.setUsername(result.get().getUsername());
    userDTO.setGender(result.get().getGender());
    userDTO.setBirthday(result.get().getBirthday());
    userDTO.setCountry(result.get().getCountry());
    userDTO.setMostRecentReviews(result.get().getMostRecentReviews(), username);
    userDTO.setFollowers(userRepos.findFollowerNumberByUsername(username));
    userDTO.setFollowedNum(userRepos.findFollowedNumberByUsername(username));
    userDTO.setCardOwned(userRepos.findCardNumberByUsername(username));
    if(myself != null)
        userDTO.setFollowed(userRepos.isFollowed(myself, username));
    return userDTO;
}
```

at repository level the method interacts with the mongoDB database using the interface defined here:



```java
public Optional<User> getUserByUsername(String username) {
    Optional<User> user = Optional.empty();
    try {
        user = userMongo.findByUsername(username);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return user;
}
```

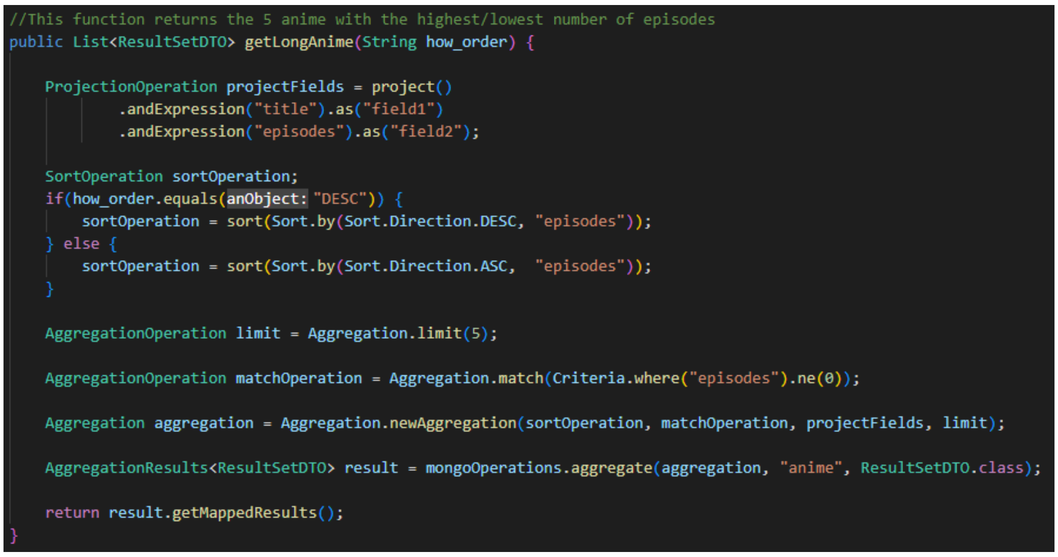


```java
public interface UserRepositoryMongo extends MongoRepository<User,String> {
    Optional<User> findByUsername(String username);
}
```

Query and analytic for registered user

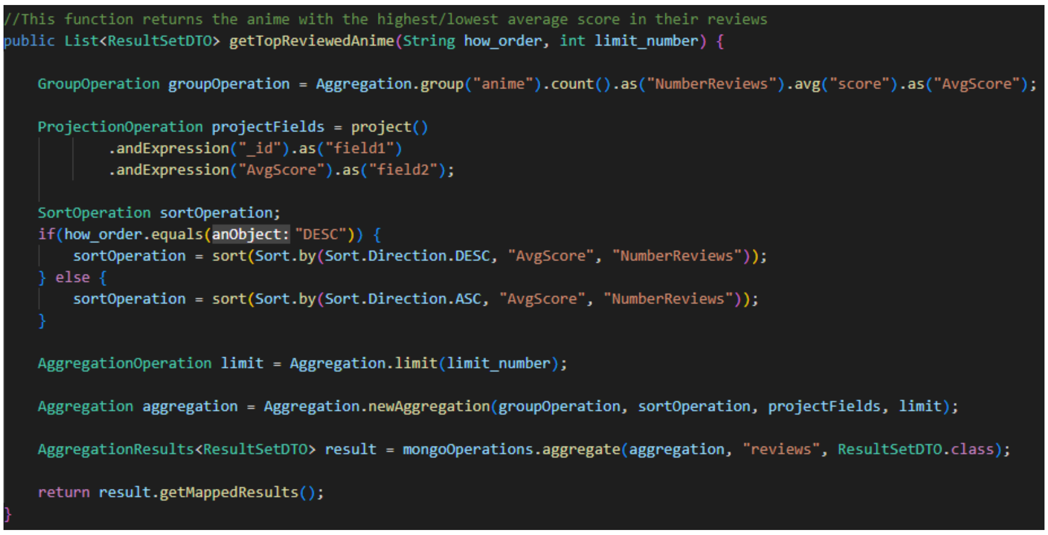
The first interaction between a user and the information get from monngoDB are:

- *"The longest/shortest anime projecting the number of episodes of 5 anime"*.



```java
//This function returns the 5 anime with the highest/lowest number of episodes
public List<ResultSetDTO> getLongAnime(String how_order) {

    ProjectionOperation projectFields = project()
            .andExpression("title").as("field1")
            .andExpression("episodes").as("field2");

    SortOperation sortOperation;
    if(how_order.equals(anObject: "DESC")) {
        sortOperation = sort(Sort.by(Sort.Direction.DESC, "episodes"));
    } else {
        sortOperation = sort(Sort.by(Sort.Direction.ASC,  "episodes"));
    }

    AggregationOperation limit = Aggregation.limit(5);

    AggregationOperation matchOperation = Aggregation.match(Criteria.where("episodes").ne(0));

    Aggregation aggregation = Aggregation.newAggregation(sortOperation, matchOperation, projectFields, limit);

    AggregationResults<ResultSetDTO> result = mongoOperations.aggregate(aggregation, "anime", ResultSetDTO.class);

    return result.getMappedResults();
}
```

- *"The Most/Least Appreciated anime projecting the average score for the reviews of 5 anime"*.



```java
//This function returns the anime with the highest/lowest average score in their reviews
public List<ResultSetDTO> getTopReviewedAnime(String how_order, int limit_number) {

    GroupOperation groupOperation = Aggregation.group("anime").count().as("NumberReviews").avg("score").as("AvgScore");

    ProjectionOperation projectFields = project()
            .andExpression("_id").as("field1")
            .andExpression("AvgScore").as("field2");

    SortOperation sortOperation;
    if(how_order.equals(anObject: "DESC")) {
        sortOperation = sort(Sort.by(Sort.Direction.DESC, "AvgScore", "NumberReviews"));
    } else {
        sortOperation = sort(Sort.by(Sort.Direction.ASC, "AvgScore", "NumberReviews"));
    }

    AggregationOperation limit = Aggregation.limit(limit_number);

    Aggregation aggregation = Aggregation.newAggregation(groupOperation, sortOperation, projectFields, limit);

    AggregationResults<ResultSetDTO> result = mongoOperations.aggregate(aggregation, "reviews", ResultSetDTO.class);

    return result.getMappedResults();
}
```

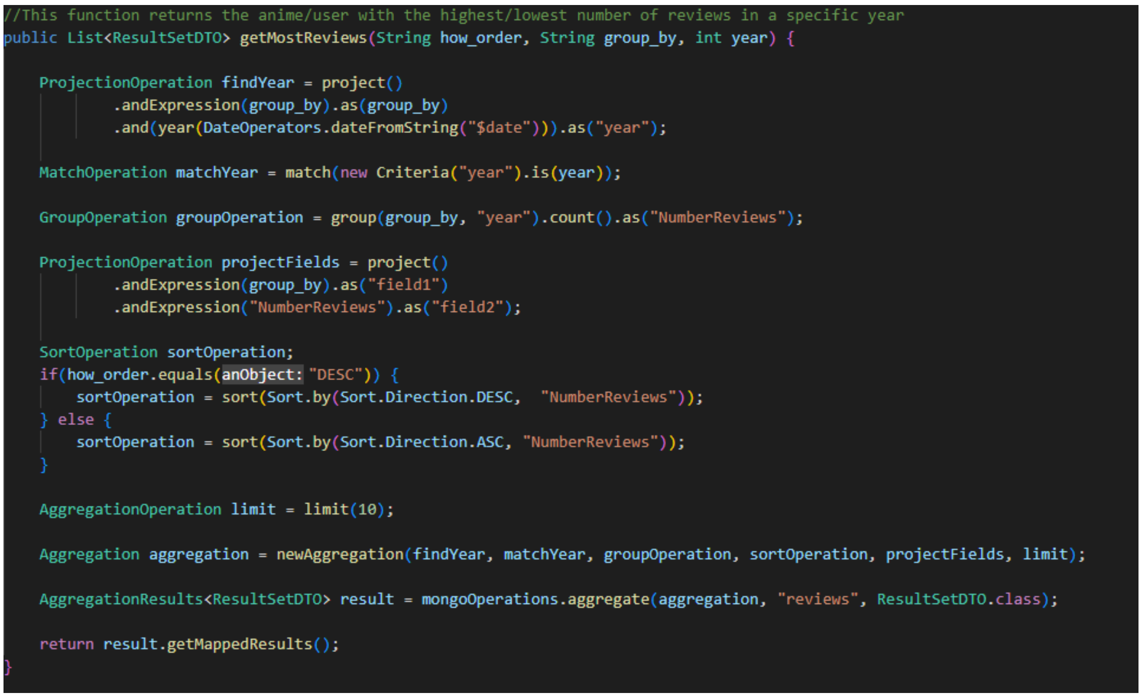
The result set of these queries are showed in **anime_search** page, with the aim to suggest an anime to view.

Query and analytic for admin

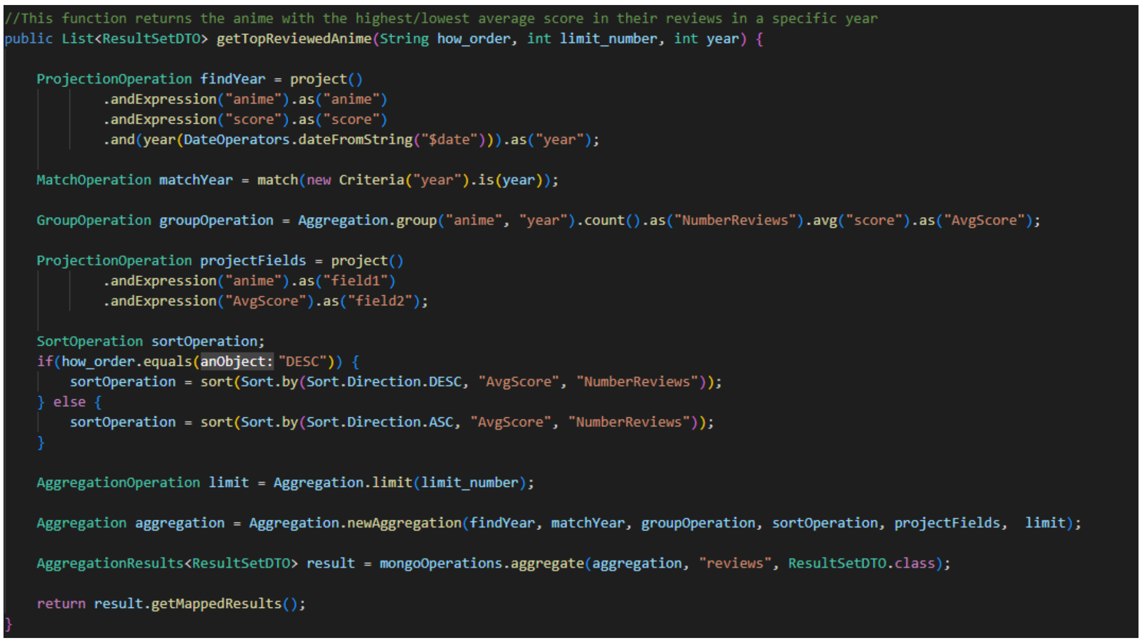
In the admin page, it can be possible to see the result of the following analytics:

- For Anime:
    - **Most/Reviewed Anime**
    - **Most Positive Rated Anime**
    - **Most Positive Weighted Anime**
- For Users:
    - **Most Active Users**
    - **Most Frequent Countries**
    - **Most Popular Users**
- For Characters:
    - **Most Loved Character**
    - **Most Rare Character**
    - **Most Unused Character**

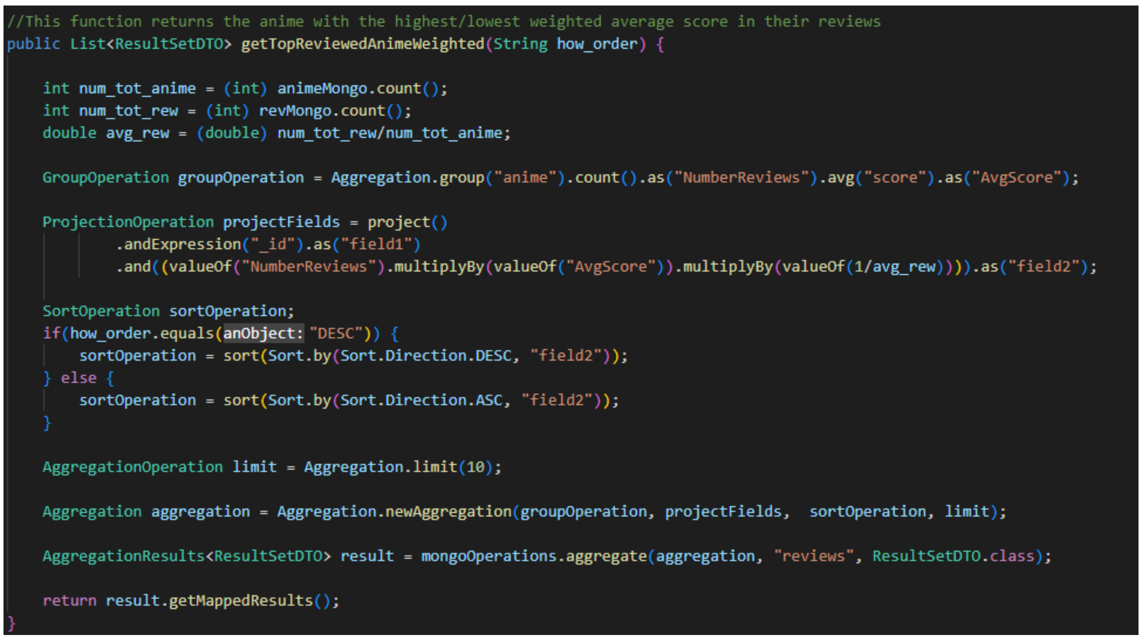
Here we report some of these aggregations, but for more details we advise to check the comments before each of them in the **following snaphsots.**



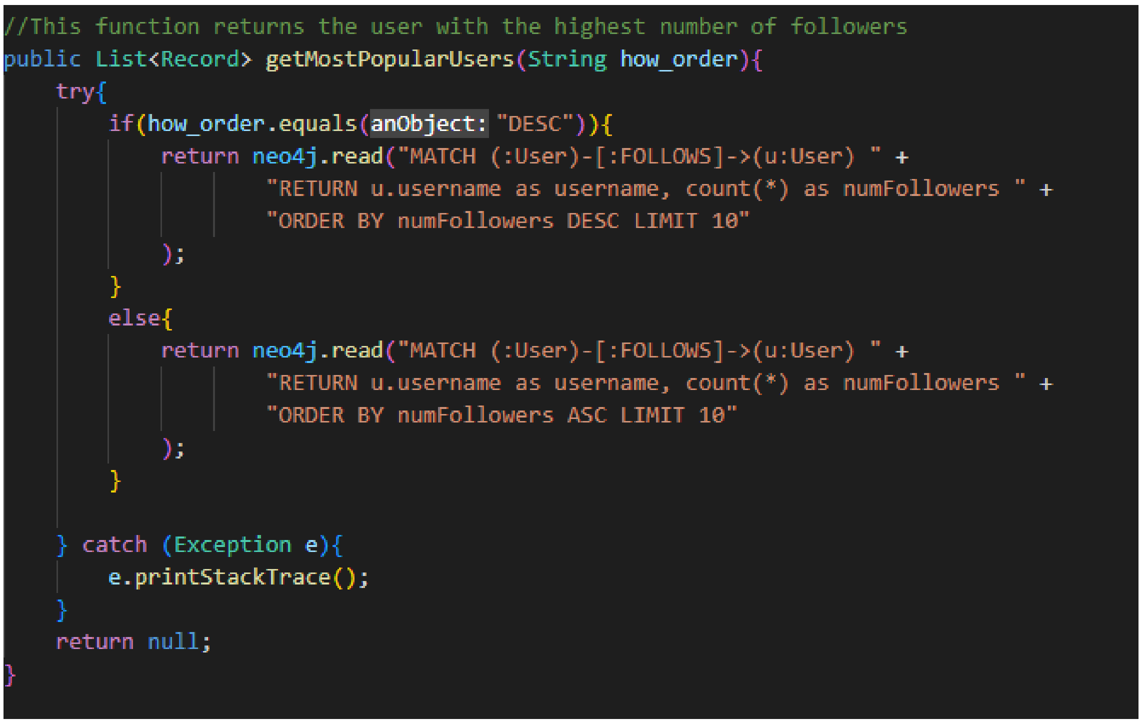
```
//This function returns the anime/user with the highest/lowest number of reviews in a specific year
public List<ResultSetDTO> getMostReviews(String how_order, String group_by, int year) {

    ProjectionOperation findYear = project()
            .andExpression(group_by).as(group_by)
            .and(year(DateOperators.dateFromString("$date"))).as("year");

    MatchOperation matchYear = match(new Criteria("year").is(year));

    GroupOperation groupOperation = group(group_by, "year").count().as("NumberReviews");

    ProjectionOperation projectFields = project()
            .andExpression(group_by).as("field1")
            .andExpression("NumberReviews").as("field2");

    SortOperation sortOperation;
    if(how_order.equals(anObject: "DESC")) {
        sortOperation = sort(Sort.by(Sort.Direction.DESC,  "NumberReviews"));
    } else {
        sortOperation = sort(Sort.by(Sort.Direction.ASC, "NumberReviews"));
    }

    AggregationOperation limit = limit(10);

    Aggregation aggregation = newAggregation(findYear, matchYear, groupOperation, sortOperation, projectFields, limit);

    AggregationResults<ResultSetDTO> result = mongoOperations.aggregate(aggregation, "reviews", ResultSetDTO.class);

    return result.getMappedResults();
}
```



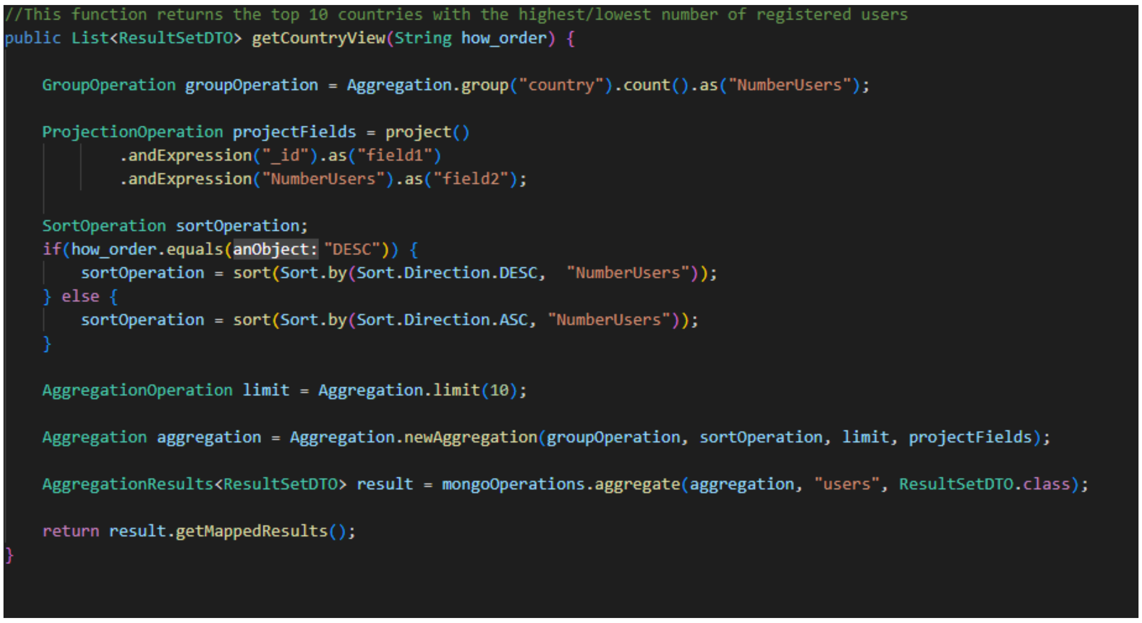
```java
//This function returns the anime with the highest/lowest average score in their reviews in a specific year
public List<ResultSetDTO> getTopReviewedAnime(String how_order, int limit_number, int year) {

    ProjectionOperation findYear = project()
            .andExpression("anime").as("anime")
            .andExpression("score").as("score")
            .and(year(DateOperators.dateFromString("$date"))).as("year");

    MatchOperation matchYear = match(new Criteria("year").is(year));

    GroupOperation groupOperation = Aggregation.group("anime", "year").count().as("NumberReviews").avg("score").as("AvgScore");

    ProjectionOperation projectFields = project()
            .andExpression("anime").as("field1")
            .andExpression("AvgScore").as("field2");

    SortOperation sortOperation;
    if(how_order.equals(anObject: "DESC")) {
        sortOperation = sort(Sort.by(Sort.Direction.DESC, "AvgScore", "NumberReviews"));
    } else {
        sortOperation = sort(Sort.by(Sort.Direction.ASC, "AvgScore", "NumberReviews"));
    }

    AggregationOperation limit = Aggregation.limit(limit_number);

    Aggregation aggregation = Aggregation.newAggregation(findYear, matchYear, groupOperation, sortOperation, projectFields,  limit);

    AggregationResults<ResultSetDTO> result = mongoOperations.aggregate(aggregation, "reviews", ResultSetDTO.class);

    return result.getMappedResults();
}
```



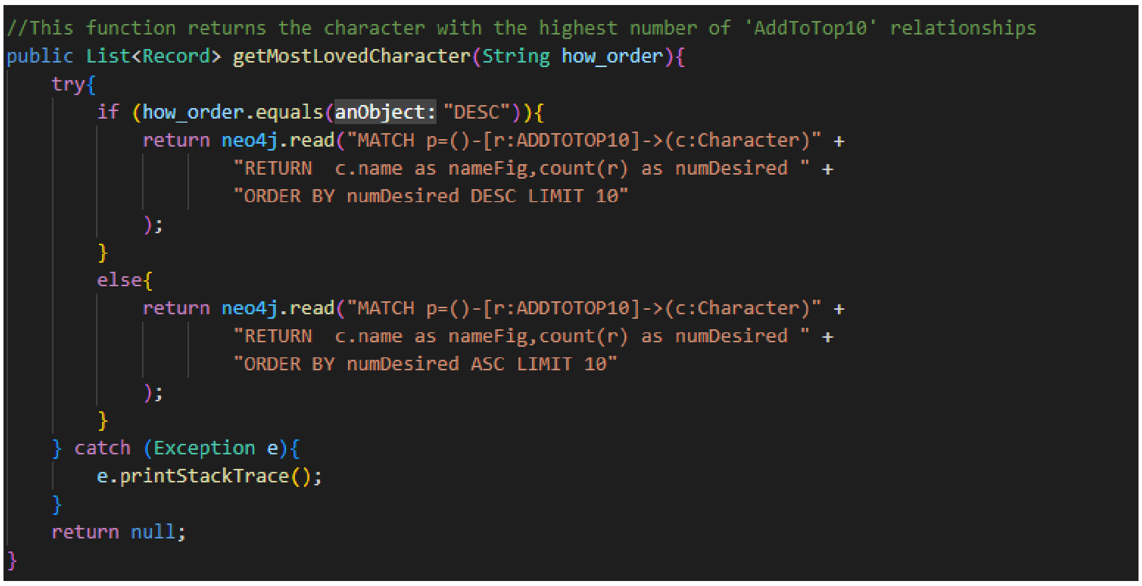
```java
//This function returns the anime with the highest/lowest weighted average score in their reviews
public List<ResultSetDTO> getTopReviewedAnimeWeighted(String how_order) {

    int num_tot_anime = (int) animeMongo.count();
    int num_tot_rew = (int) revMongo.count();
    double avg_rew = (double) num_tot_rew/num_tot_anime;

    GroupOperation groupOperation = Aggregation.group("anime").count().as("NumberReviews").avg("score").as("AvgScore");

    ProjectionOperation projectFields = project()
            .andExpression("_id").as("field1")
            .and((valueOf("NumberReviews").multiplyBy(valueOf("AvgScore")).multiplyBy(valueOf(1/avg_rew)))).as("field2");

    SortOperation sortOperation;
    if(how_order.equals(anObject: "DESC")) {
        sortOperation = sort(Sort.by(Sort.Direction.DESC, "field2"));
    } else {
        sortOperation = sort(Sort.by(Sort.Direction.ASC, "field2"));
    }

    AggregationOperation limit = Aggregation.limit(10);

    Aggregation aggregation = Aggregation.newAggregation(groupOperation, projectFields,  sortOperation, limit);

    AggregationResults<ResultSetDTO> result = mongoOperations.aggregate(aggregation, "reviews", ResultSetDTO.class);

    return result.getMappedResults();
}
```



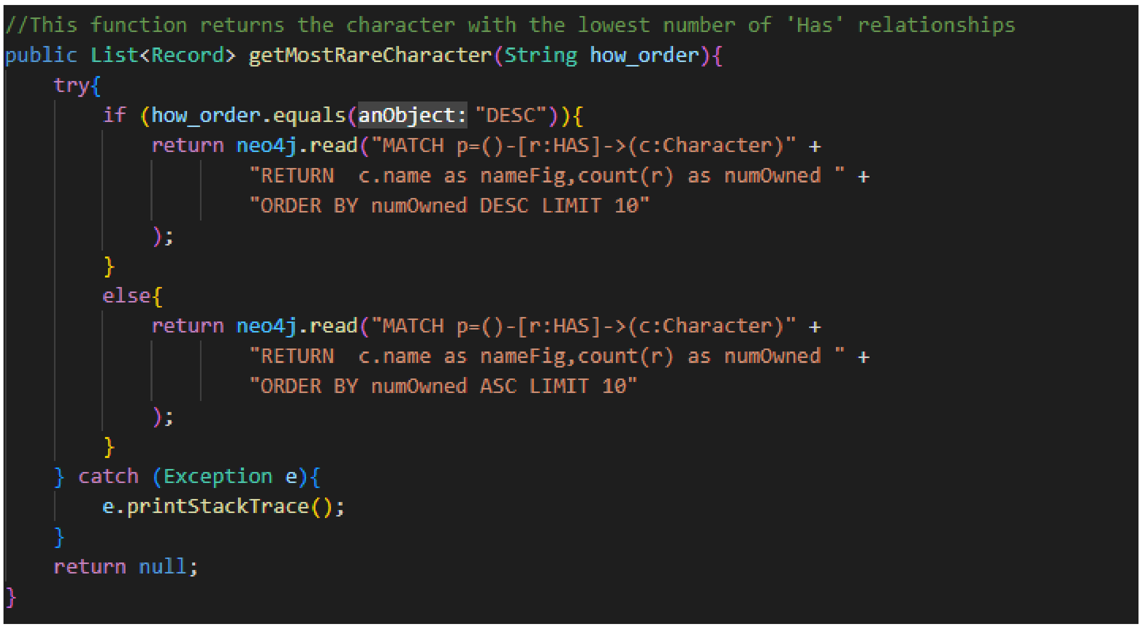
```
//This function returns the user with the highest number of followers
public List<Record> getMostPopularUsers(String how_order){
    try{
        if(how_order.equals(anObject: "DESC")){
            return neo4j.read("MATCH (:User)-[:FOLLOWS]->(u:User) " +
                    "RETURN u.username as username, count(*) as numFollowers " +
                    "ORDER BY numFollowers DESC LIMIT 10"
            );
        }
        else{
            return neo4j.read("MATCH (:User)-[:FOLLOWS]->(u:User) " +
                    "RETURN u.username as username, count(*) as numFollowers " +
                    "ORDER BY numFollowers ASC LIMIT 10"
            );
        }

    } catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```



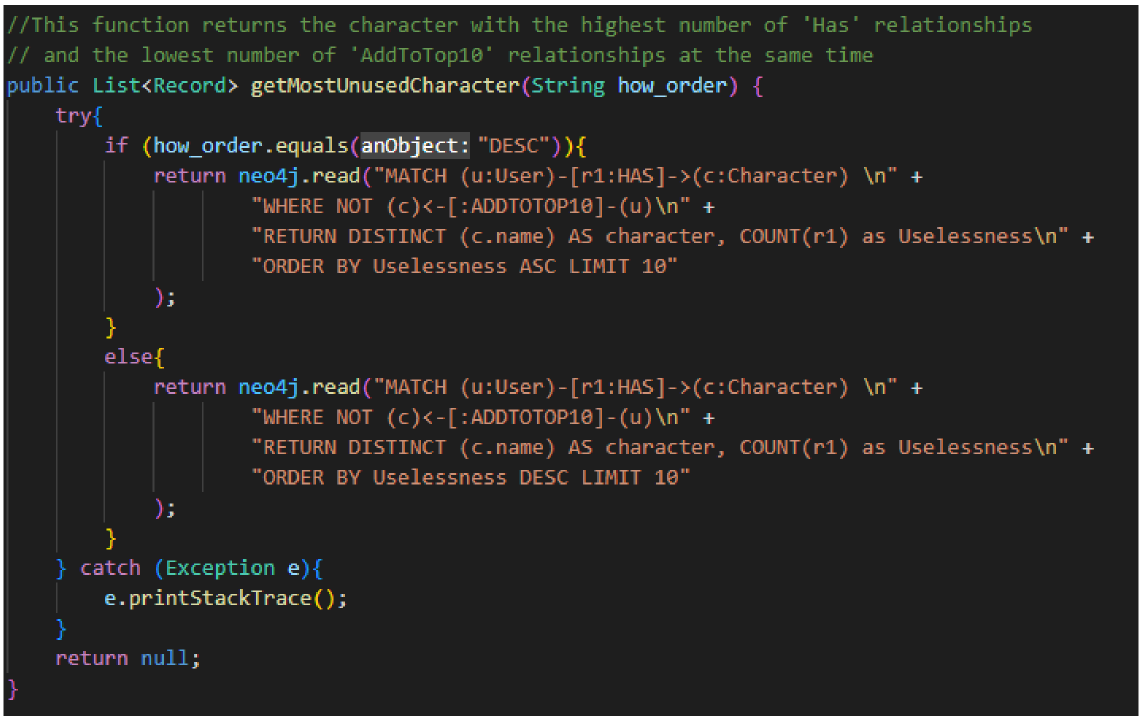
```
//This function returns the top 10 countries with the highest/lowest number of registered users
public List<ResultSetDTO> getCountryView(String how_order) {

    GroupOperation groupOperation = Aggregation.group("country").count().as("NumberUsers");

    ProjectionOperation projectFields = project()
            .andExpression("_id").as("field1")
            .andExpression("NumberUsers").as("field2");

    SortOperation sortOperation;
    if(how_order.equals(anObject: "DESC")) {
        sortOperation = sort(Sort.by(Sort.Direction.DESC,  "NumberUsers"));
    } else {
        sortOperation = sort(Sort.by(Sort.Direction.ASC, "NumberUsers"));
    }

    AggregationOperation limit = Aggregation.limit(10);

    Aggregation aggregation = Aggregation.newAggregation(groupOperation, sortOperation, limit, projectFields);

    AggregationResults<ResultSetDTO> result = mongoOperations.aggregate(aggregation, "users", ResultSetDTO.class);

    return result.getMappedResults();
}
```



```java
//This function returns the character with the highest number of 'AddToTop10' relationships
public List<Record> getMostLovedCharacter(String how_order){
    try{
        if (how_order.equals(anObject: "DESC")){
            return neo4j.read("MATCH p=()-[r:ADDTOTOP10]->(c:Character)" +
                    "RETURN  c.name as nameFig,count(r) as numDesired " +
                    "ORDER BY numDesired DESC LIMIT 10"
            );
        }
        else{
            return neo4j.read("MATCH p=()-[r:ADDTOTOP10]->(c:Character)" +
                    "RETURN  c.name as nameFig,count(r) as numDesired " +
                    "ORDER BY numDesired ASC LIMIT 10"
            );
        }
    } catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```



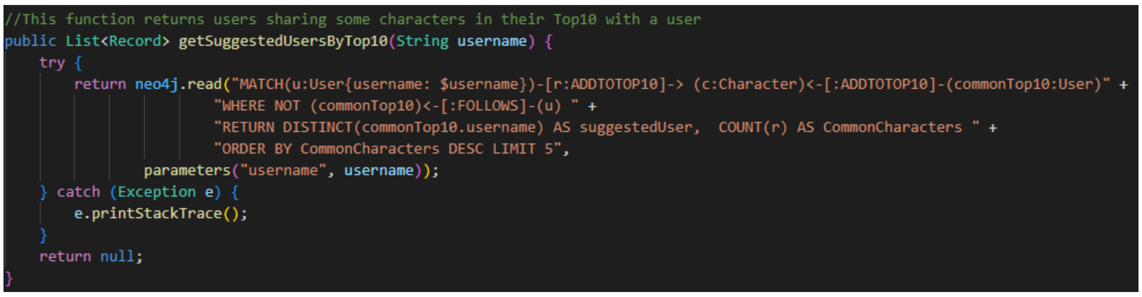
```java
//This function returns the character with the lowest number of 'Has' relationships
public List<Record> getMostRareCharacter(String how_order){
    try{
        if (how_order.equals(anObject: "DESC")){
            return neo4j.read("MATCH p=()-[r:HAS]->(c:Character)" +
                    "RETURN  c.name as nameFig,count(r) as numOwned " +
                    "ORDER BY numOwned DESC LIMIT 10"
            );
        }
        else{
            return neo4j.read("MATCH p=()-[r:HAS]->(c:Character)" +
                    "RETURN  c.name as nameFig,count(r) as numOwned " +
                    "ORDER BY numOwned ASC LIMIT 10"
            );
        }
    } catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```



```
//This function returns the character with the highest number of 'Has' relationships
// and the lowest number of 'AddToTop10' relationships at the same time
public List<Record> getMostUnusedCharacter(String how_order) {
    try{
        if (how_order.equals(anObject: "DESC")){
            return neo4j.read("MATCH (u:User)-[r1:HAS]->(c:Character) \n" +
                    "WHERE NOT (c)<-[:ADDTOTOP10]-(u)\n" +
                    "RETURN DISTINCT (c.name) AS character, COUNT(r1) as Uselessness\n" +
                    "ORDER BY Uselessness ASC LIMIT 10"
            );
        }
        else{
            return neo4j.read("MATCH (u:User)-[r1:HAS]->(c:Character) \n" +
                    "WHERE NOT (c)<-[:ADDTOTOP10]-(u)\n" +
                    "RETURN DISTINCT (c.name) AS character, COUNT(r1) as Uselessness\n" +
                    "ORDER BY Uselessness DESC LIMIT 10"
            );
        }
    } catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

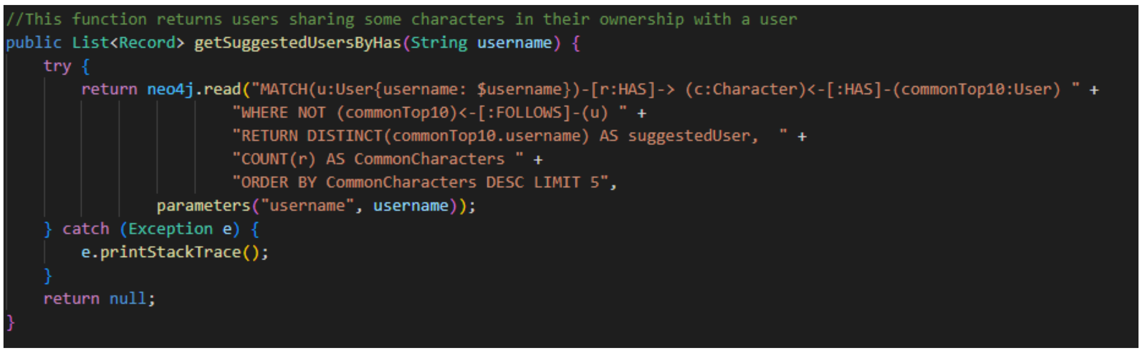
User Suggestions

The **suggestion** we present are all used by exploiting the graph. The application suggests to a user other users based on **three levels** in order to find at least 5 users. In this kind of suggestion, we select the users that have incoming arcs to characters that we have in the top 10 counting the characters in common between their top 10 and ours.



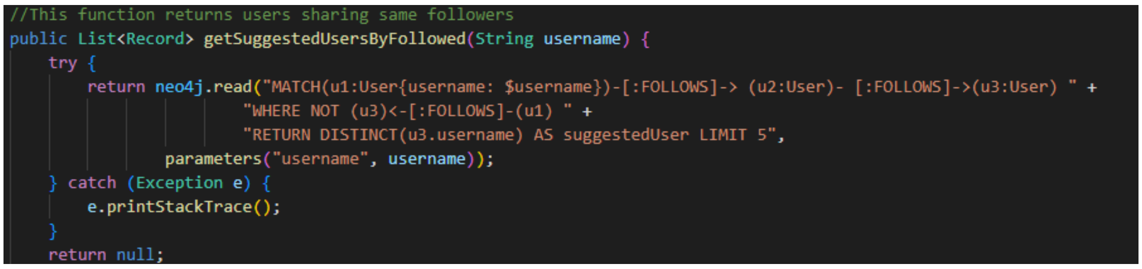
```
//This function returns users sharing some characters in their Top10 with a user
public List<Record> getSuggestedUsersByTop10(String username) {
    try {
        return neo4j.read("MATCH(u:User{username: $username})-[r:ADDTOTOP10]-> (c:Character)<-[:ADDTOTOP10]-(commonTop10:User)" +
                "WHERE NOT (commonTop10)<-[:FOLLOWS]-(u) " +
                "RETURN DISTINCT(commonTop10.username) AS suggestedUser,  COUNT(r) AS CommonCharacters " +
                "ORDER BY CommonCharacters DESC LIMIT 5",
            parameters("username", username));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Another kind of suggestion of users on the graph is the one involving the same reasoning ad before, but involving the characters owned in common between a user and other users.



```java
//This function returns users sharing some characters in their ownership with a user
public List<Record> getSuggestedUsersByHas(String username) {
    try {
        return neo4j.read("MATCH(u:User{username: $username})-[r:HAS]-> (c:Character)<-[:HAS]-(commonTop10:User) " +
                                "WHERE NOT (commonTop10)<-[:FOLLOWS]-(u) " +
                                "RETURN DISTINCT(commonTop10.username) AS suggestedUser,  " +
                                "COUNT(r) AS CommonCharacters " +
                                "ORDER BY CommonCharacters DESC LIMIT 5",
                        parameters("username", username));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

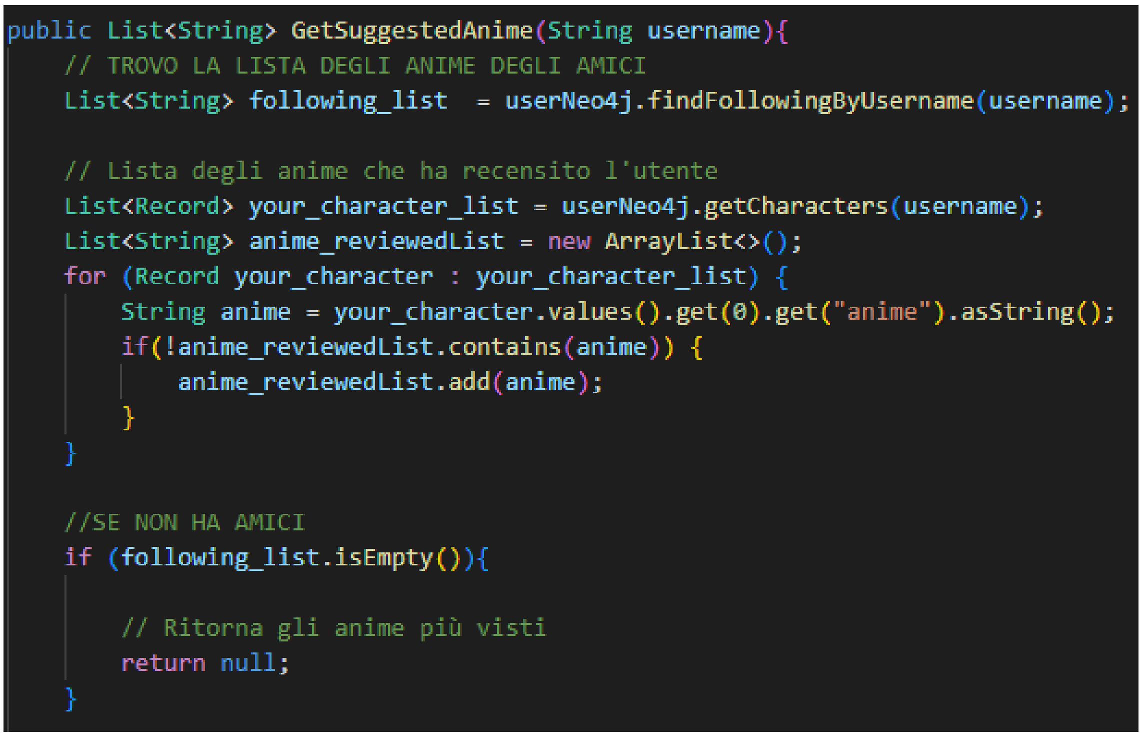
If there are not enough users in this way, we search for users that are followed by our followings.



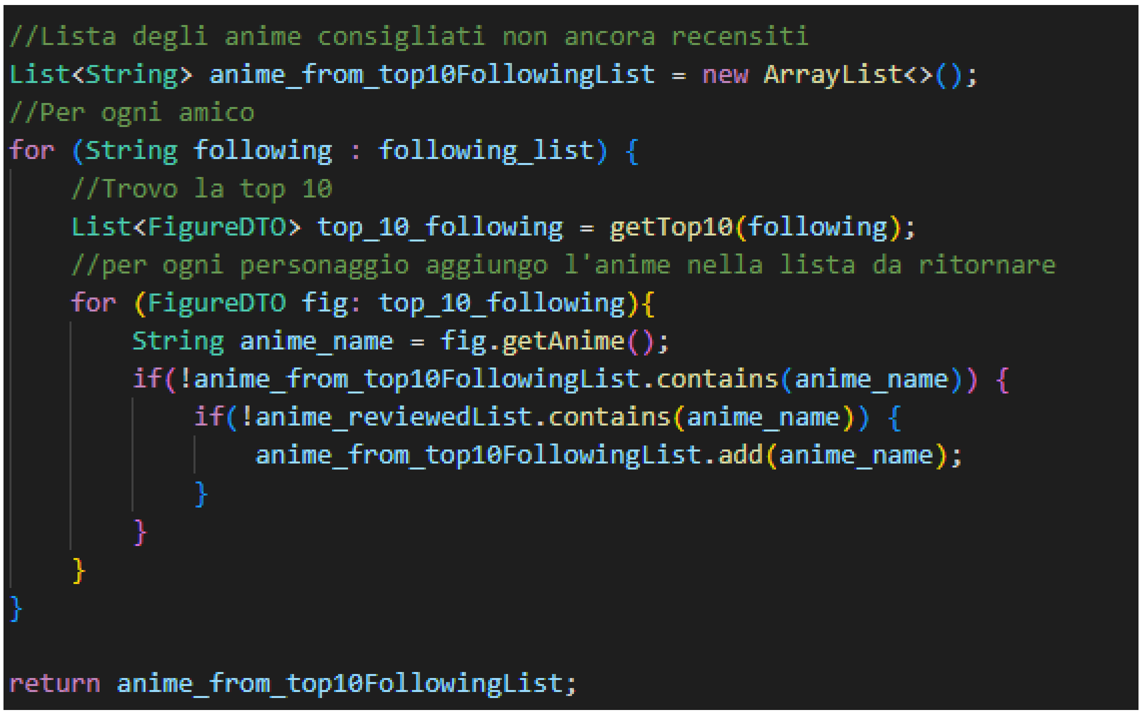
```java
//This function returns users sharing same followers
public List<Record> getSuggestedUsersByFollowed(String username) {
    try {
        return neo4j.read("MATCH(u1:User{username: $username})-[:FOLLOWS]-> (u2:User)- [:FOLLOWS]->(u3:User) " +
                                "WHERE NOT (u3)<-[:FOLLOWS]-(u1) " +
                                "RETURN DISTINCT(u3.username) AS suggestedUser LIMIT 5",
                        parameters("username", username));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
```

Anime Suggestions

The application suggests to a user anime to see based on a **two levels suggestion**. In this kind of suggestion, the application suggests anime considering anime of the Top10 of a user’s friends and the anime of the cards owned by them.



```java
public List<String> GetSuggestedAnime(String username){
    // TROVO LA LISTA DEGLI ANIME DEGLI AMICI
    List<String> following_list  = userNeo4j.findFollowingByUsername(username);

    // Lista degli anime che ha recensito l'utente
    List<Record> your_character_list = userNeo4j.getCharacters(username);
    List<String> anime_reviewedList = new ArrayList<>();
    for (Record your_character : your_character_list) {
        String anime = your_character.values().get(0).get("anime").asString();
        if(!anime_reviewedList.contains(anime)) {
            anime_reviewedList.add(anime);
        }
    }

    //SE NON HA AMICI
    if (following_list.isEmpty()){

        // Ritorna gli anime più visti
        return null;
    }
```



```java
//Lista degli anime consigliati non ancora recensiti
List<String> anime_from_top10FollowingList = new ArrayList<>();
//Per ogni amico
for (String following : following_list) {
    //Trovo la top 10
    List<FigureDTO> top_10_following = getTop10(following);
    //per ogni personaggio aggiungo l'anime nella lista da ritornare
    for (FigureDTO fig: top_10_following){
        String anime_name = fig.getAnime();
        if(!anime_from_top10FollowingList.contains(anime_name)) {
            if(!anime_reviewedList.contains(anime_name)) {
                anime_from_top10FollowingList.add(anime_name);
            }
        }
    }
}

return anime_from_top10FollowingList;
```

Database Choices

In this section we discuss about index choices for MongoDB, Replicas handling and data sharding.

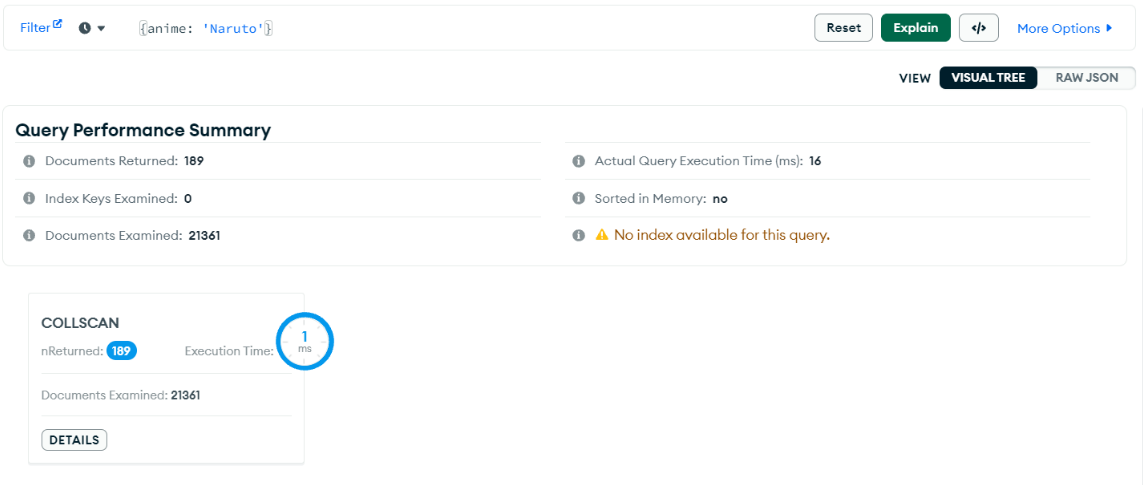
Indexes

As seen, this application is a **read intensive one**. So overall we wanted to enhance read performance we decided to use indexes on attributes involved on the most relevant and **frequent** operations that users do.

Regarding mongoDB collections, the following attributes were chosen as indexes:

- **Anime title in the collection "Anime":** The reasoning behind this is related to the fact that we estimate that the operation that is most frequent is searching for an anime, so putting an index here results in a clear performance enhancement in this kind of operation. Especially considering that inserting a new anime is an operation which is done with a much lower rate than searching one.
- **Anime title in the collection "Reviews":** Since reviews are strictly related to anime, and in the application, you can see all the reviews associated to an anime, but not those related to a user, we decided to increase the performance of performing that research.
- **Username in the collection “Users”:** For the same reasons as anime.

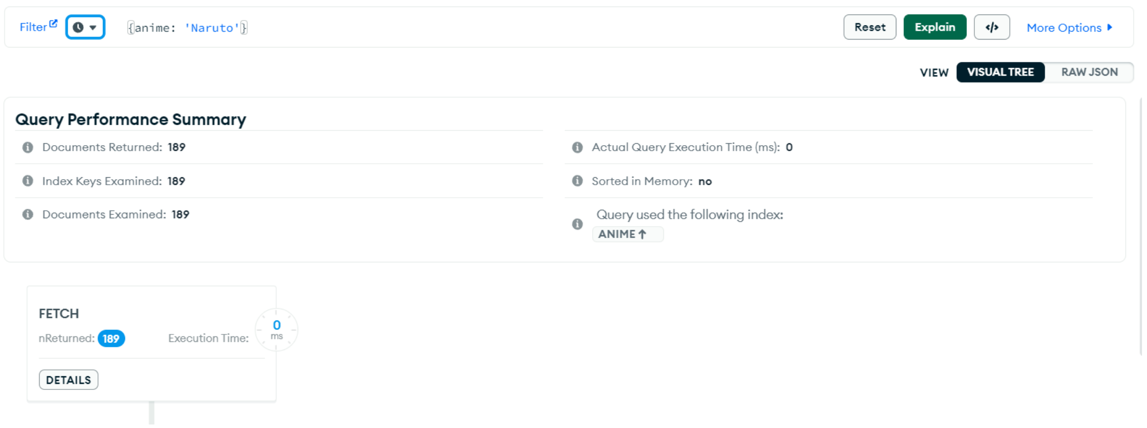
Those indexes, as we can see in the snapshots, increased the performances of query like the researching of an anime, or of a user.


Filter {anime: 'Naruto'} Reset Explain More Options
VIEW VISUAL TREE RAW JSON

Query Performance Summary

Documents Returned: 189
Index Keys Examined: 0
Documents Examined: 21361
Actual Query Execution Time (ms): 16
Sorted in Memory: no
No index available for this query.

COLLSCAN
nReturned: 189
Execution Time: 1 ms
Documents Examined: 21361
DETAILS


Filter {anime: 'Naruto'} Reset Explain More Options
VIEW VISUAL TREE RAW JSON

Query Performance Summary

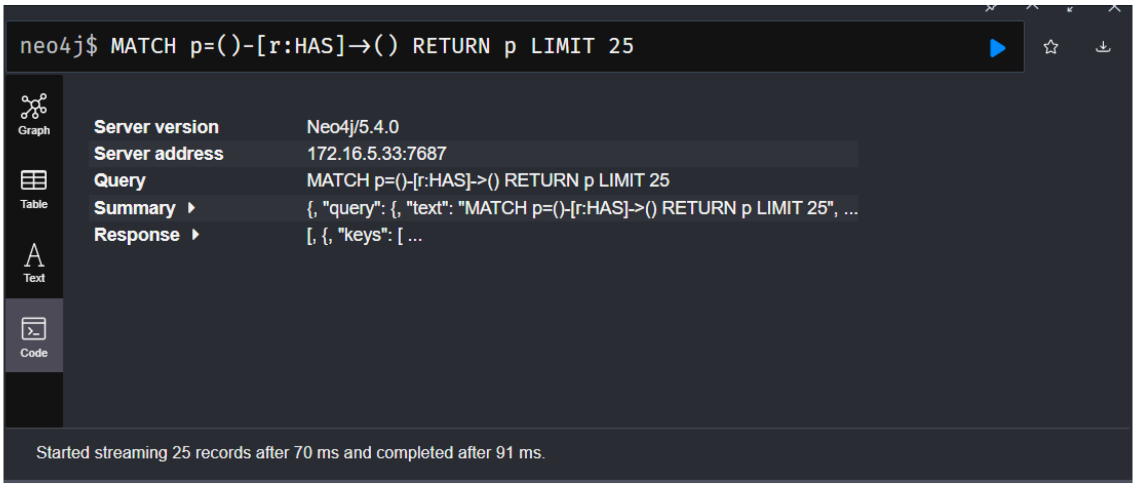
Documents Returned: 189
Index Keys Examined: 189
Documents Examined: 189
Actual Query Execution Time (ms): 0
Sorted in Memory: no
Query used the following index:
ANIME ↑

FETCH
nReturned: 189
Execution Time: 0 ms
DETAILS

Regarding Neo4J, the following attributes were chosen as indexes:

- **Username for the entity “User”**
- **Character name for the entity “Character”**

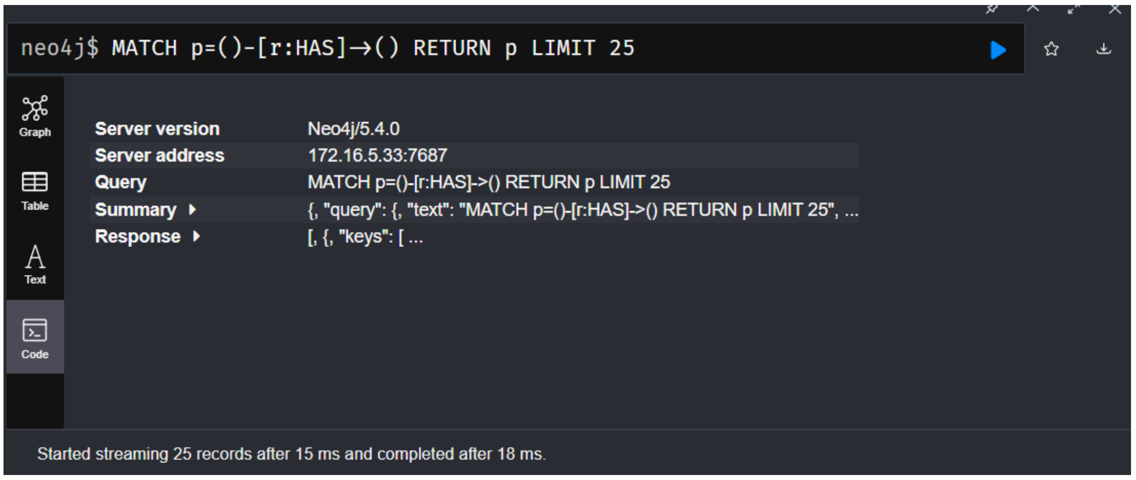
Since every time we search for a user or for a character, we have also to consider relationships, we decided to add indexes also in Neo4J.



```
neo4j$ MATCH p=()-[r:HAS]→() RETURN p LIMIT 25
```

```
Server version    Neo4j/5.4.0
Server address    172.16.5.33:7687
Query             MATCH p=()-[r:HAS]->() RETURN p LIMIT 25
Summary ▸         {, "query": {, "text": "MATCH p=()-[r:HAS]->() RETURN p LIMIT 25", ...
Response ▸        [, {, "keys": [ ...
```

Started streaming 25 records after 70 ms and completed after 91 ms.



```
neo4j$ MATCH p=()-[r:HAS]→() RETURN p LIMIT 25
```

```
Server version    Neo4j/5.4.0
Server address    172.16.5.33:7687
Query             MATCH p=()-[r:HAS]->() RETURN p LIMIT 25
Summary ▸         {, "query": {, "text": "MATCH p=()-[r:HAS]->() RETURN p LIMIT 25", ...
Response ▸        [, {, "keys": [ ...
```

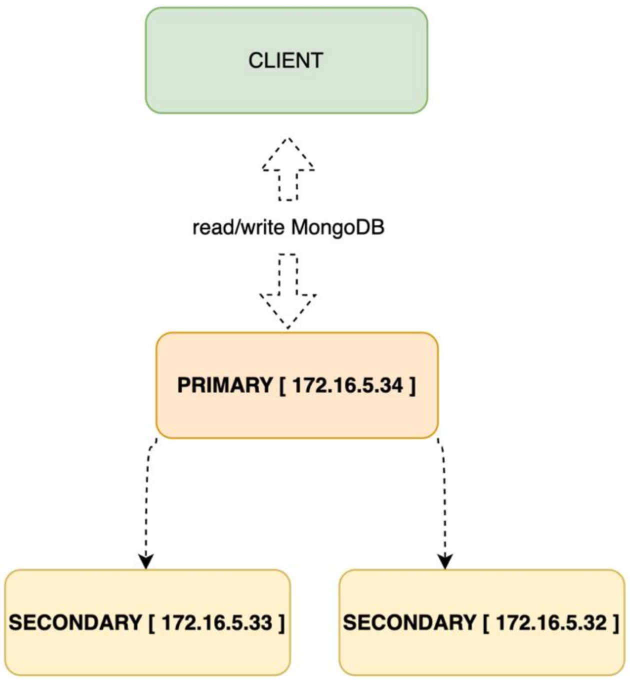
Started streaming 25 records after 15 ms and completed after 18 ms.

Distributed Database Design

Since the most important non-functional requirements is about **high availability,** we decided to orient replica design in order to meet this requirement.

Replica set (MongoDB)

The replica set consists in one primary replica that is used to respond to clients' requests and then other two secondary replicas used to replicate the one of the primaries.



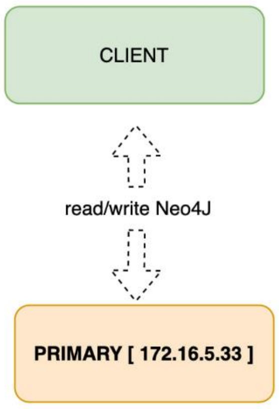
Replica configuration of MongoDB

We decided to set the write concern to 1 because we want high performances, so given the fact that we prefer availability over consistency we want to regain control after only the data in the primary replica has been written. We know that this could cause problems if the primary crashes before propagating writes to the secondary replicas, but we think that for the sake of the application we can accept this scenario. So, the properties set for the configuration are the followings:

- **Write concern** = 1
- **readPreference** = nearest

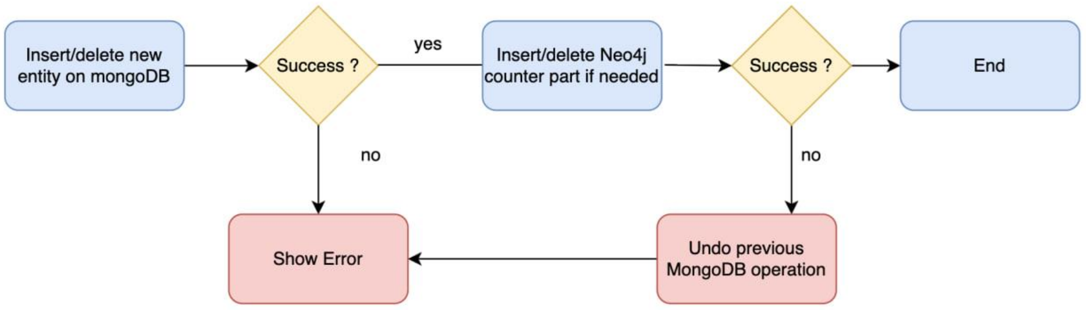
Replica set (Neo4J)

for Neo4j we are limited to only have 1 replica, we put the replica in a VM different from the one of mongo for avoiding single point of failure.



Consistency between databases

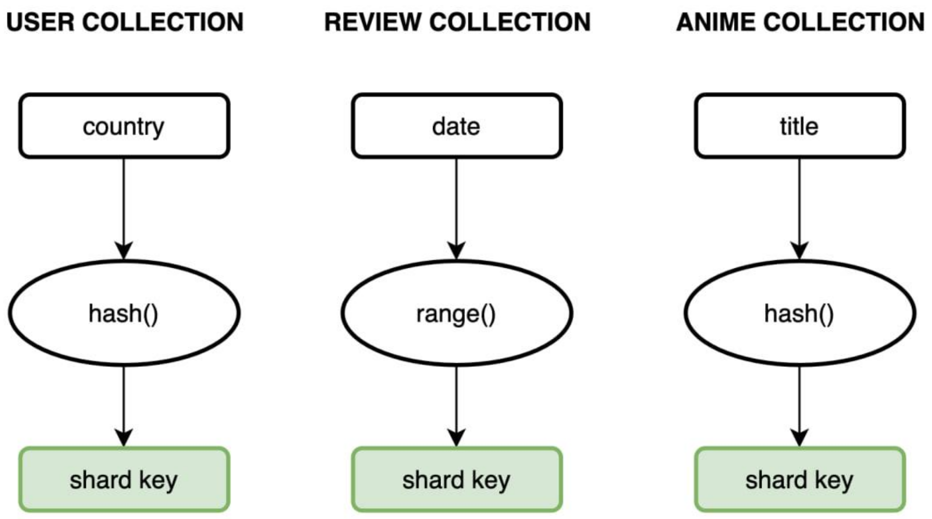
We manage the consistency of data between databases as described in the following diagram (for both insertion/deletion):


Insert/delete new entity on mongoDB
Success ?
yes
no
Insert/delete Neo4j counter part if needed
Success ?
no
End
Show Error
Undo previous MongoDB operation

Sharding proposal

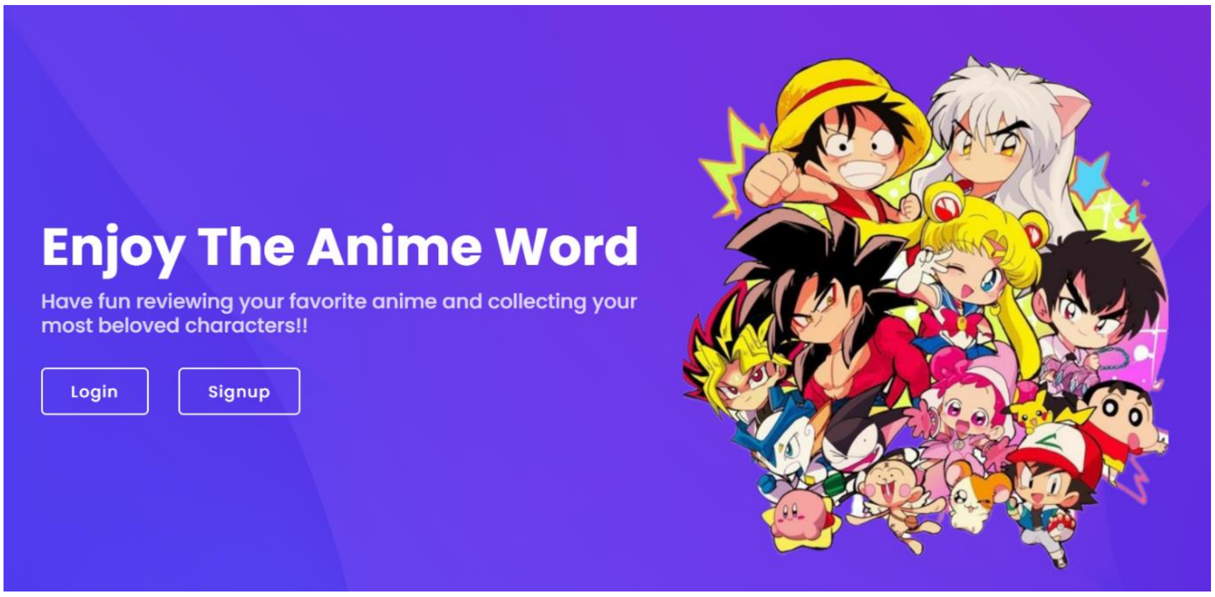
We propose the sharding attributes for the collections of mongoDB being:

- ***country*** for the **User**: The reasoning in this is related to the performance of certain analytics on the user that require grouping by Country.
- ***date*** for the **review:** For the same analytic performance reasoning seen before we use a range of dates and associate it to a shard.
- ***title*** for the **anime** collection

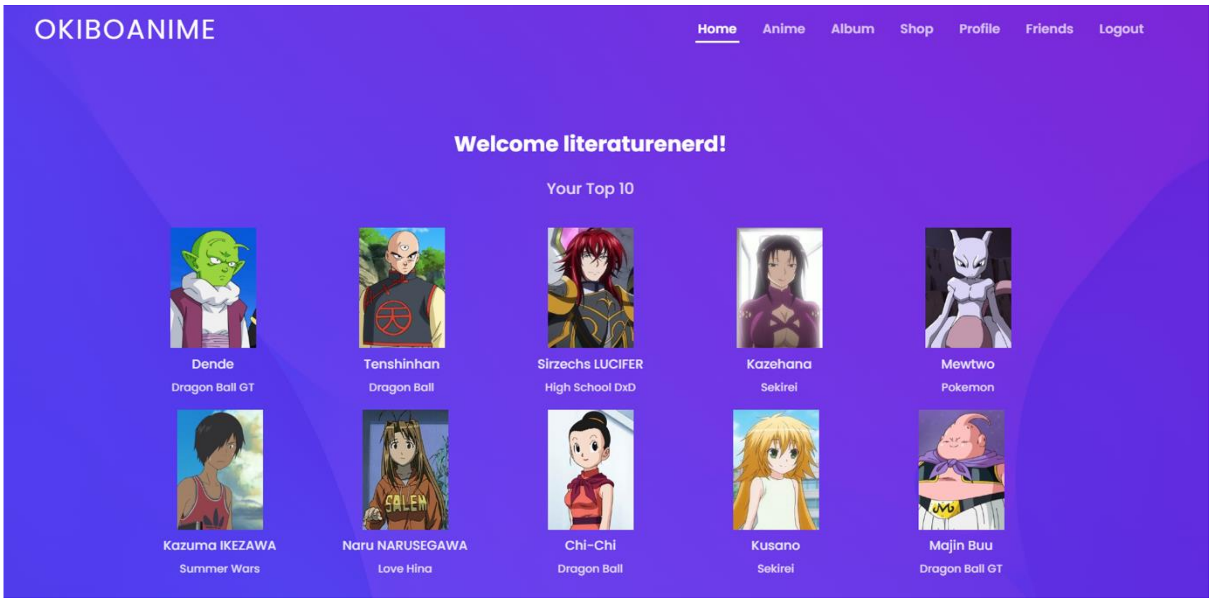


Usage Manual

In the index page you can login or signup to the application


Enjoy The Anime Word
Have fun reviewing your favorite anime and collecting your most beloved characters!!
Login
Signup

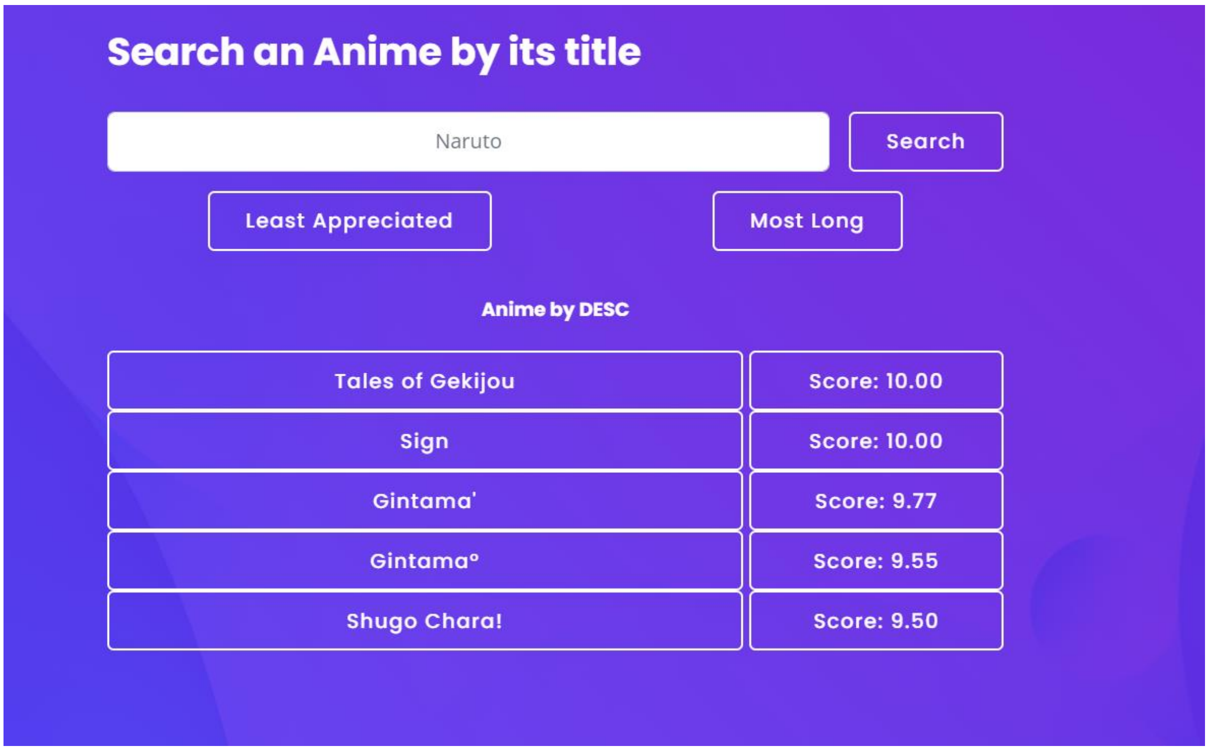
After compiled the form for login or signup, if you are not an administrator, you will be redirected to the home page.



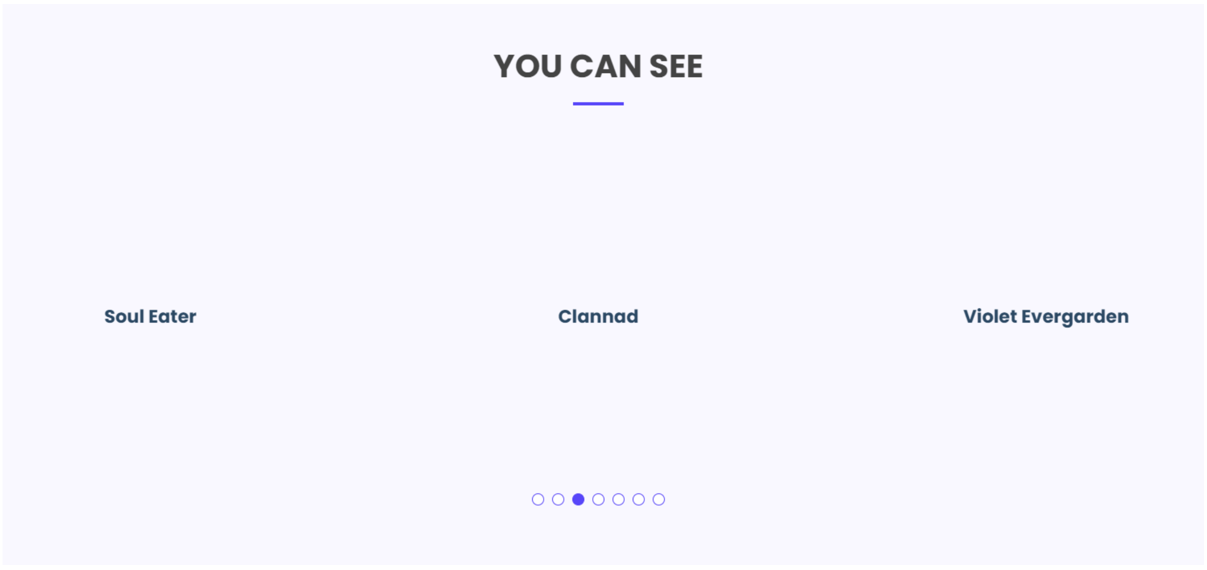
Here you can view your Top10 List and you can access through the navbar to the other sections.

Anime

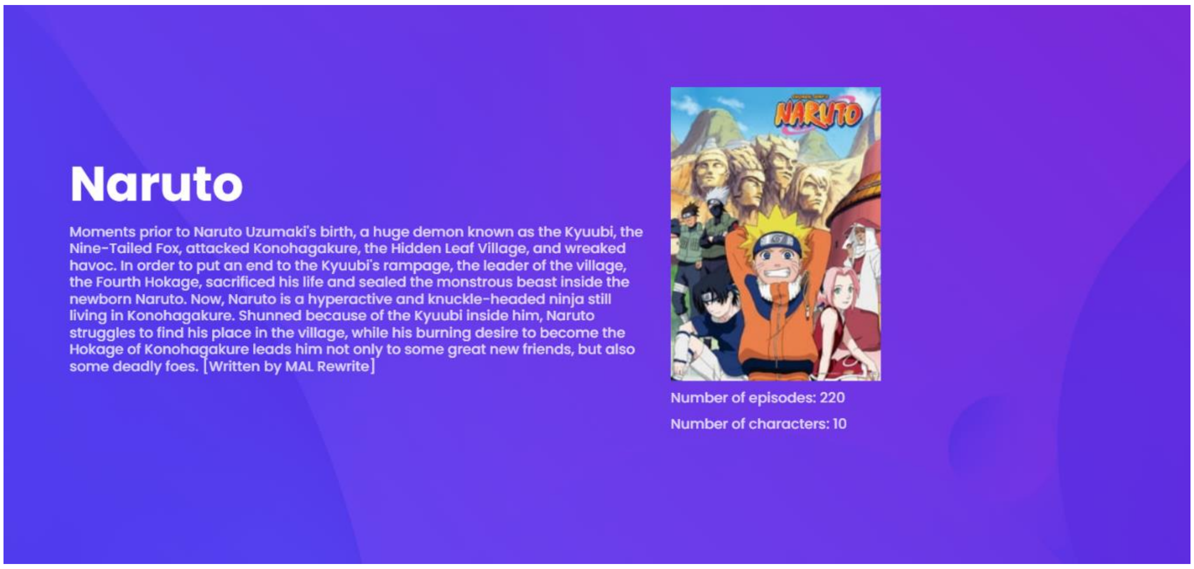
In this page you can search an Anime by the input element, you can see which are the most appreciated and long anime (and the worst), and some suggestions on anime. If you click on an anime you will be redirected to its page.



Tales of Gekijou	Score: 10.00
Sign	Score: 10.00
Gintama'	Score: 9.77
Gintama°	Score: 9.55
Shugo Chara!	Score: 9.50



Here you can view characteristics of the anime (including characters), the last 3 reviews. You can also see your review or write one if you did not do it before. If you click on “*View all the reviews”* you will be redirected to a page in which you can see all the reviews of the anime.

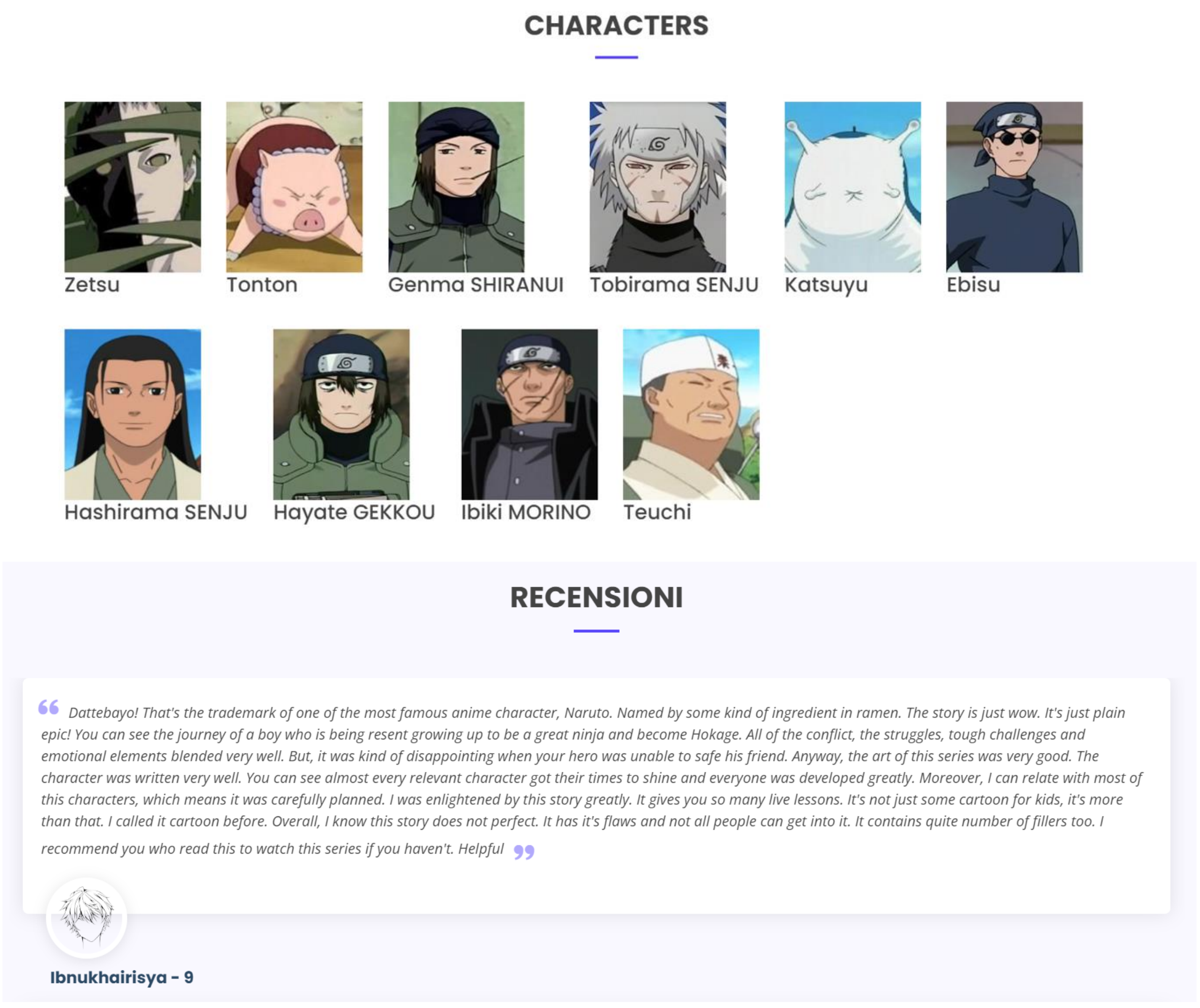


Naruto

Moments prior to Naruto Uzumaki's birth, a huge demon known as the Kyuubi, the Nine-Tailed Fox, attacked Konohagakure, the Hidden Leaf Village, and wreaked havoc. In order to put an end to the Kyuubi's rampage, the leader of the village, the Fourth Hokage, sacrificed his life and sealed the monstrous beast inside the newborn Naruto. Now, Naruto is a hyperactive and knuckle-headed ninja still living in Konohagakure. Shunned because of the Kyuubi inside him, Naruto struggles to find his place in the village, while his burning desire to become the Hokage of Konohagakure leads him not only to some great new friends, but also some deadly foes. [Written by MAL Rewrite]

Number of episodes: 220

Number of characters: 10



CHARACTERS

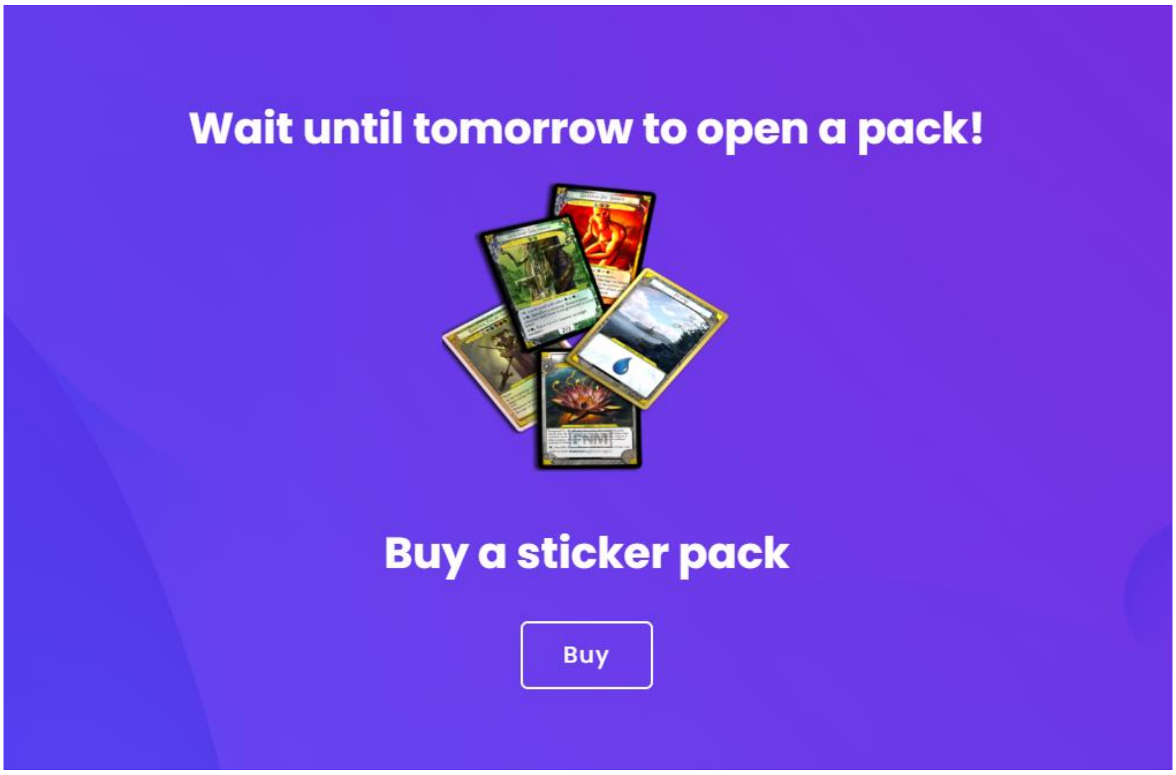
RECENSIONI

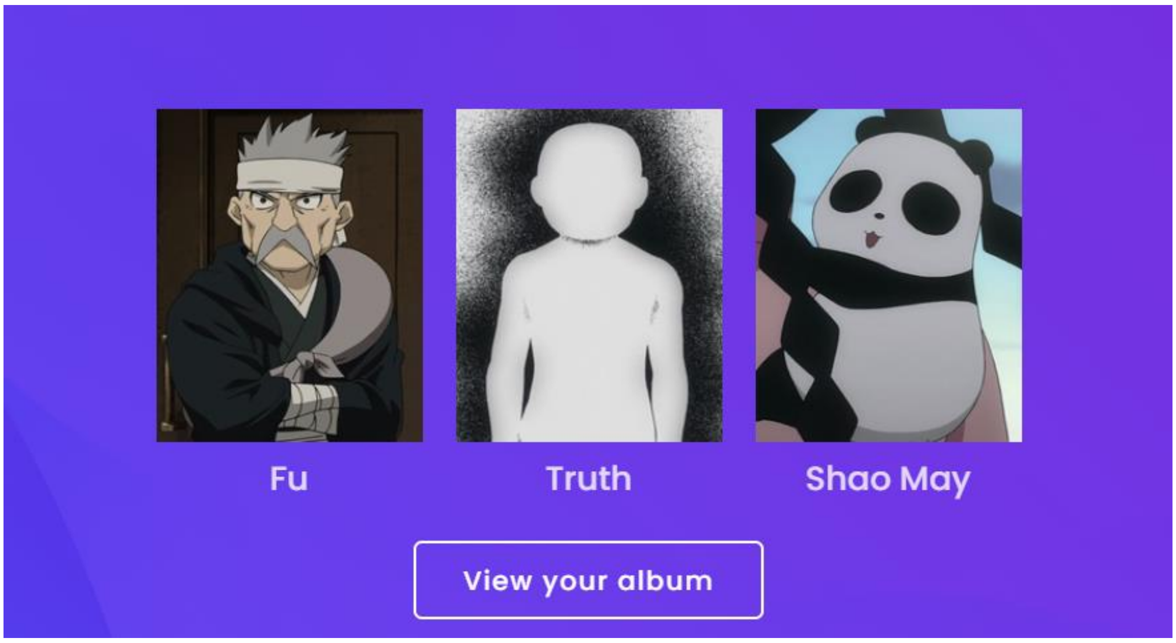
Dattebayo! That's the trademark of one of the most famous anime character, Naruto. Named by some kind of ingredient in ramen. The story is just wow. It's just plain epic! You can see the journey of a boy who is being resent growing up to be a great ninja and become Hokage. All of the conflict, the struggles, tough challenges and emotional elements blended very well. But, it was kind of disappointing when your hero was unable to safe his friend. Anyway, the art of this series was very good. The character was written very well. You can see almost every relevant character got their times to shine and everyone was developed greatly. Moreover, I can relate with most of this characters, which means it was carefully planned. I was enlightened by this story greatly. It gives you so many live lessons. It's not just some cartoon for kids, it's more than that. I called it cartoon before. Overall, I know this story does not perfect. It has it's flaws and not all people can get into it. It contains quite number of fillers too. I recommend you who read this to watch this series if you haven't. Helpful

Ibnukhairisya - 9

Shop

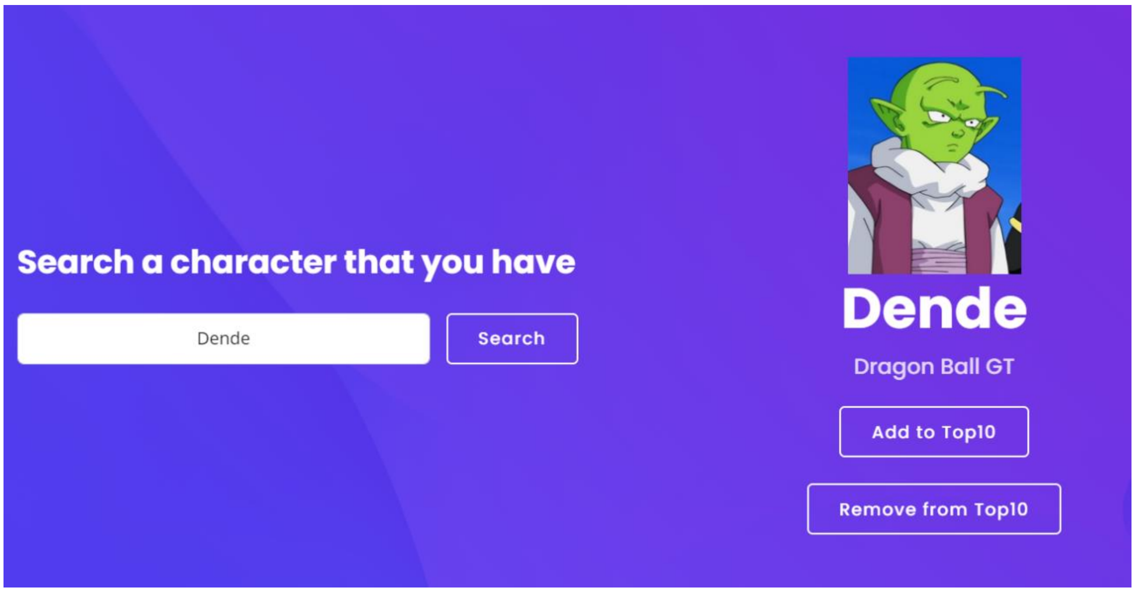
Here you can open your daily pack if you did not do it today, and you can see which cards you can find at the bottom of the page.

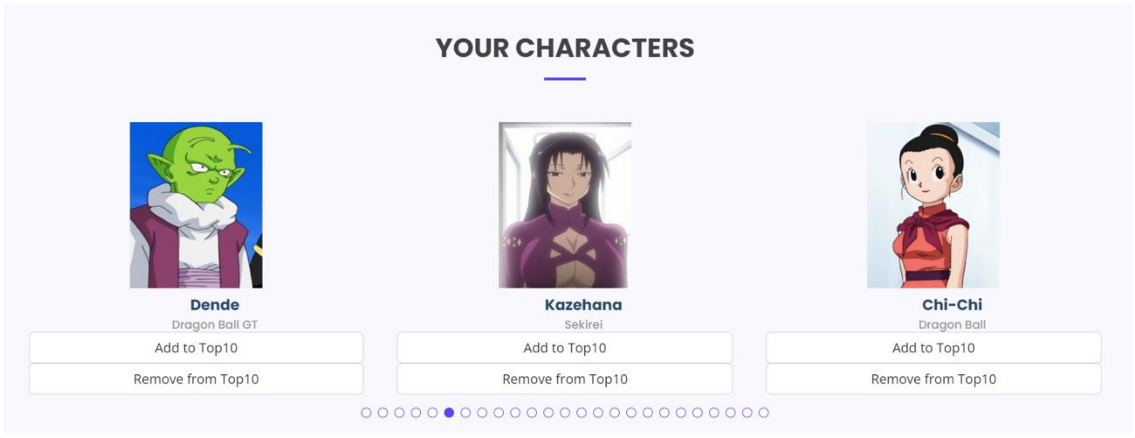




Album

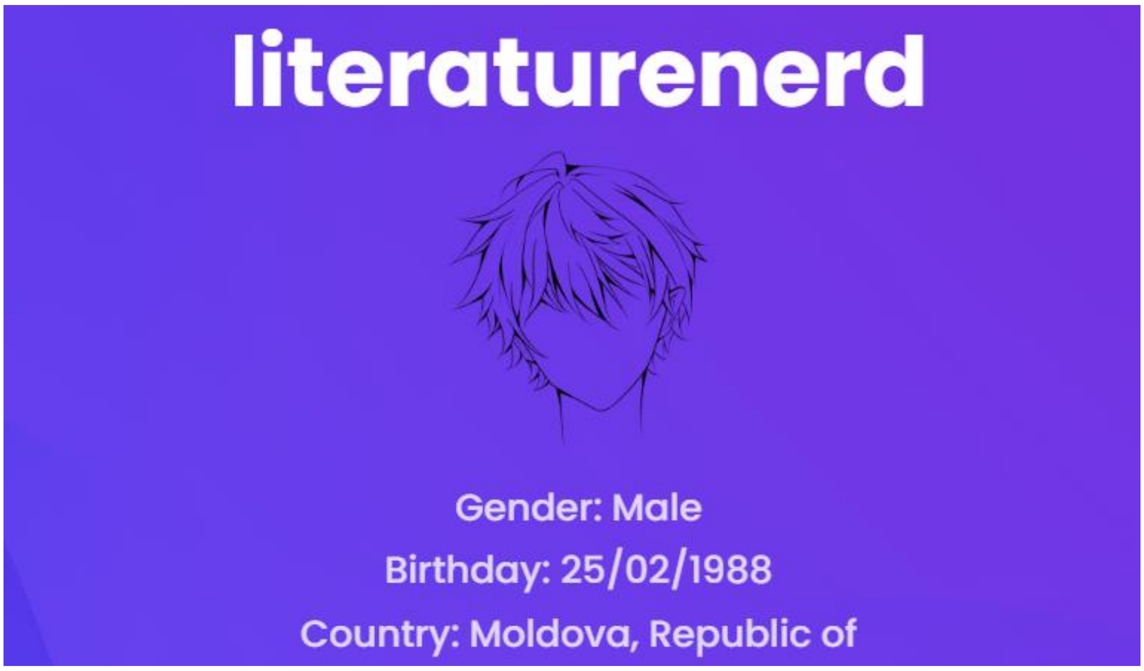
Here you can view all your cards searching them, or a set of them at the bottom of the page. You can also add or remove them from your personal Top10.





Profile

Here you can see your data, your last 5 reviews, the follower number, the followed number and the number of card owned.


literaturenerd

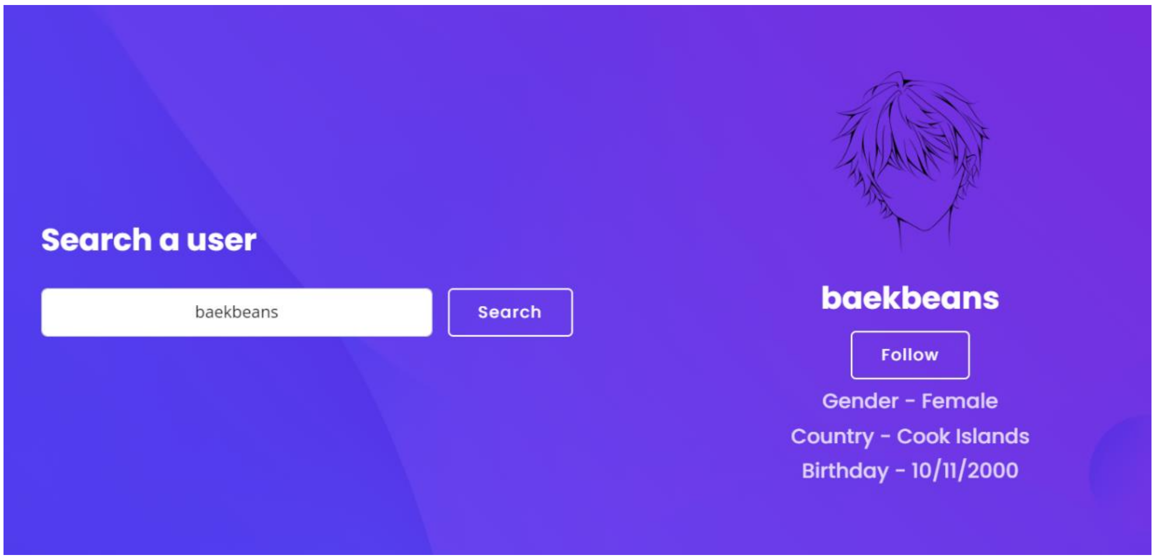
Gender: Male

Birthday: 25/02/1988

Country: Moldova, Republic of

Friends

Here you can search users and see their information including last reviews. You can also follow/unfollow a user unlocking the possibility to view his/her Top10. There are also some suggestions of users with interests like yours.

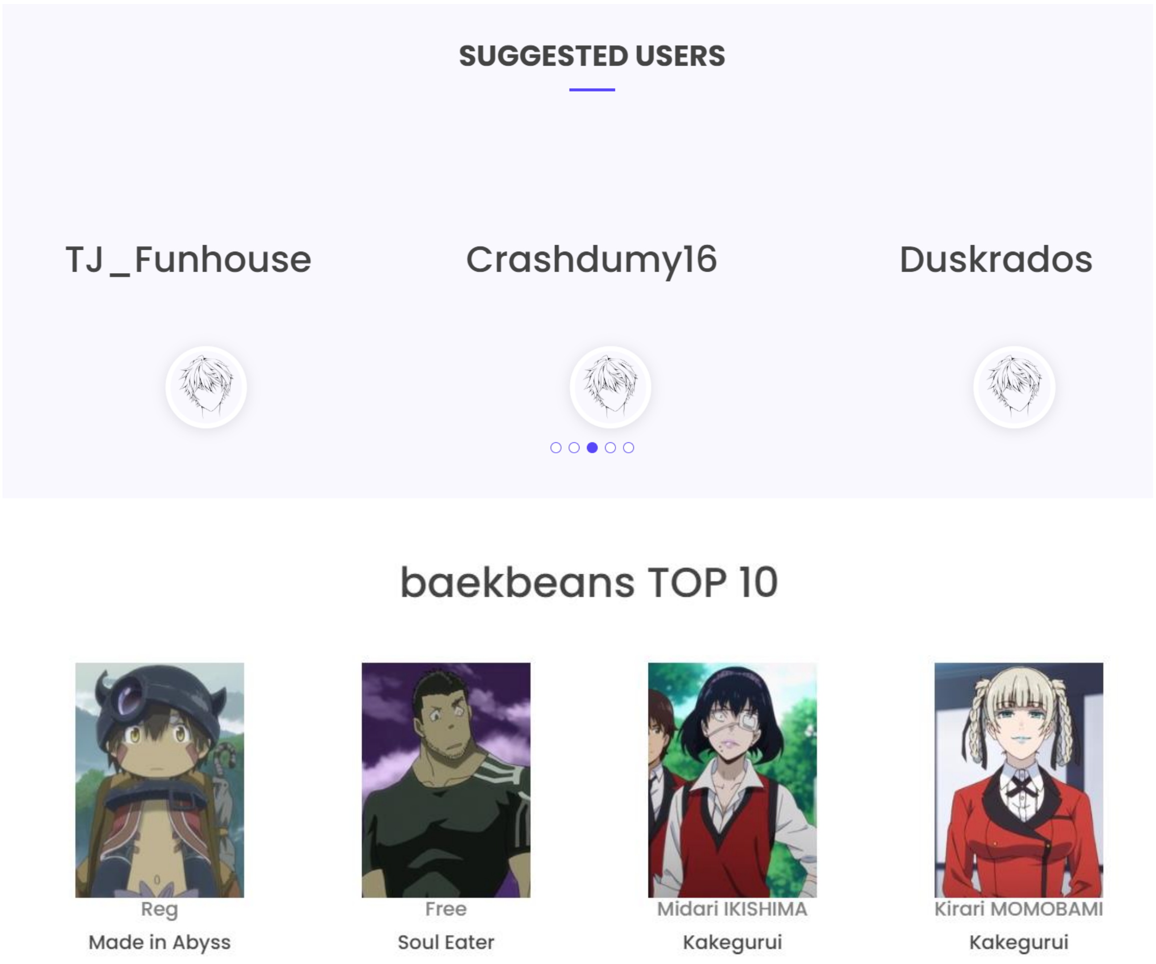


baekbeans

Gender - Female

Country - Cook Islands

Birthday - 10/11/2000



SUGGESTED USERS

TJ_Funhouse

Crashdumy16

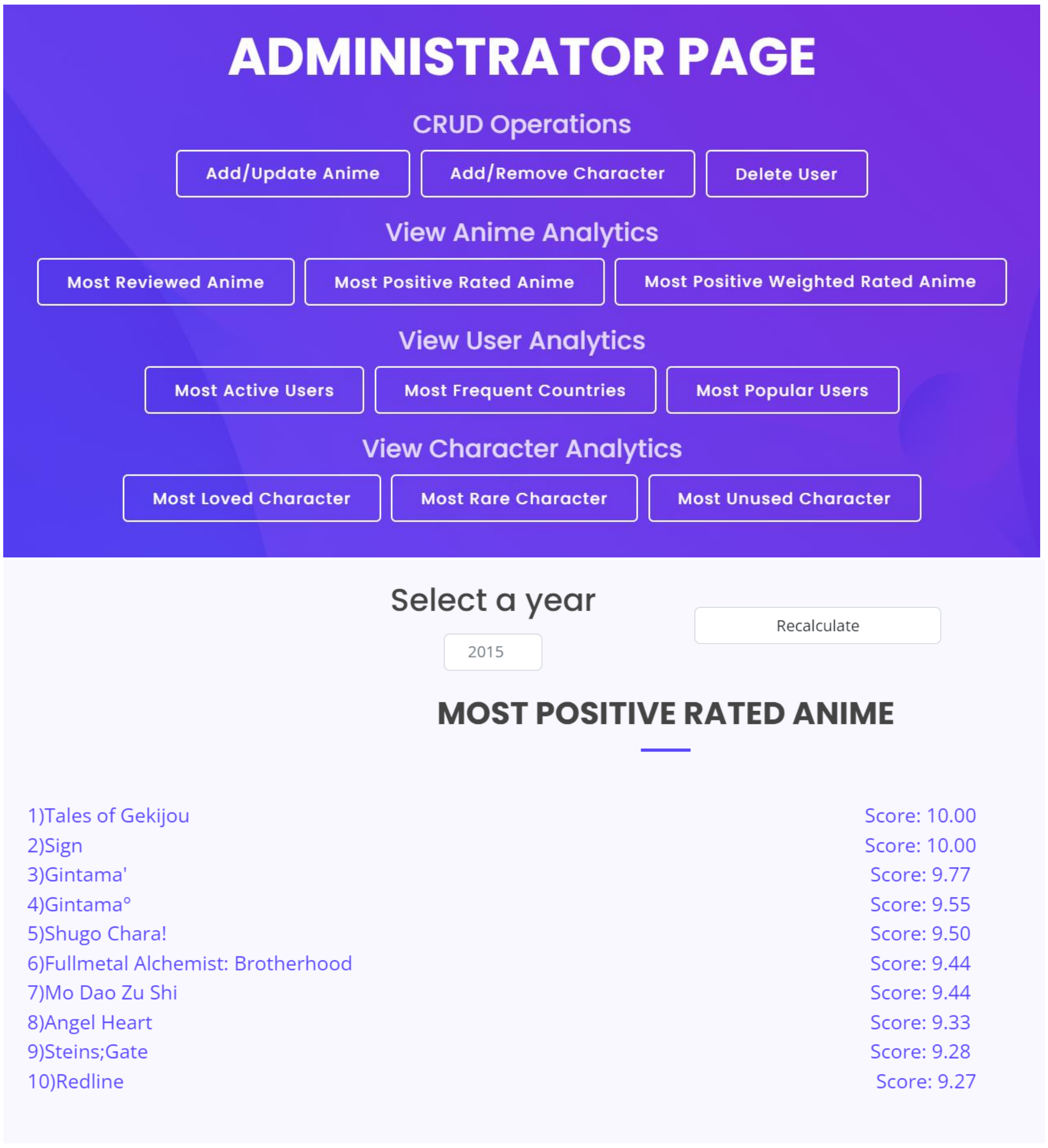
Duskrados

baekbeans TOP 10

Reg	Free	Midari IKISHIMA	Kirari MOMOBAMI
Made in Abyss	Soul Eater	Kakegurui	Kakegurui

Admin

If you are an administrator of the application (username: 'admin', password: 'admin') you can do some CRUD operations, and view analytics on users, anime and characters.



MOST POSITIVE RATED ANIME

1)Tales of Gekijou	Score: 10.00
2)Sign	Score: 10.00
3)Gintama'	Score: 9.77
4)Gintama°	Score: 9.55
5)Shugo Chara!	Score: 9.50
6)Fullmetal Alchemist: Brotherhood	Score: 9.44
7)Mo Dao Zu Shi	Score: 9.44
8)Angel Heart	Score: 9.33
9)Steins;Gate	Score: 9.28
10)Redline	Score: 9.27

Future updates

This application gives the possibility to add a lot of possible functionalities as:

- Adding the possibility of viewing the complete list of followers of a user.
- Adding the possibility of exchanging character cards with other users.
- Adding the possibility of updating a written review.
- Adding the possibility of commenting others reviews or like them.