UNIVERSITY OF PISA
FACULTY OF ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING / COMPUTER ENGINEERING

Large Scale and Multi Structured Databases

Project Documentation

BGNet

Group: Casu Federico, De Marco Angelo, Pardini Marco

PISA, JANUARY 2023

Contents

1 Link to GitHub

The code is available on GitHub at this link.

2 Introduction

In this section we present our application idea, discussing the final result we expect in terms of User Experience. Moreover, we talk about how we gathered all our data that were used to build this prototype.

2.1 Application highlights

BGNet is a social network-like application that allows users to get in touch with each other. It stands for **Board Games Network**, because the common denominator between who decides to use this software is a (at least) curiosity for Board Games'world.

In our application, we want to guarantee mainly two experiences:

- We want to have a main page for each Game we stored (2). This page has to be the main place in which users can speak about the game. They can post something about the game, comment someone other's post, etc. In this page users are allowed to see tournaments that are planned for that game, and they can join or even create a new tournament.
- We want to personalize the homepage (1) of each user, giving him/her the possibility of seeing different suggestions based on its behave in the software. We want to allow you to discover new games that you can find enjoyable!

Lastly, we want to allow administrators to have access to analysis regarding website activities, to better think about what to change, what is going good and what is going bad.

2.2 Dataset creation

We decided to scrape real data from the web. In particular, to respect the **variety** of data, we have used two different sources:

- We have used **Board Game Geek API** to scrape informations from BGG about users, posts, and games
- We have used **Board Game Arena** to scrape informations about tournaments

2.2.1 Games

We selected from Board Game Geek (3) a pool of 100 Board Games, and we scraped some basic descriptions about them. These information will be shown in the main page of each game.

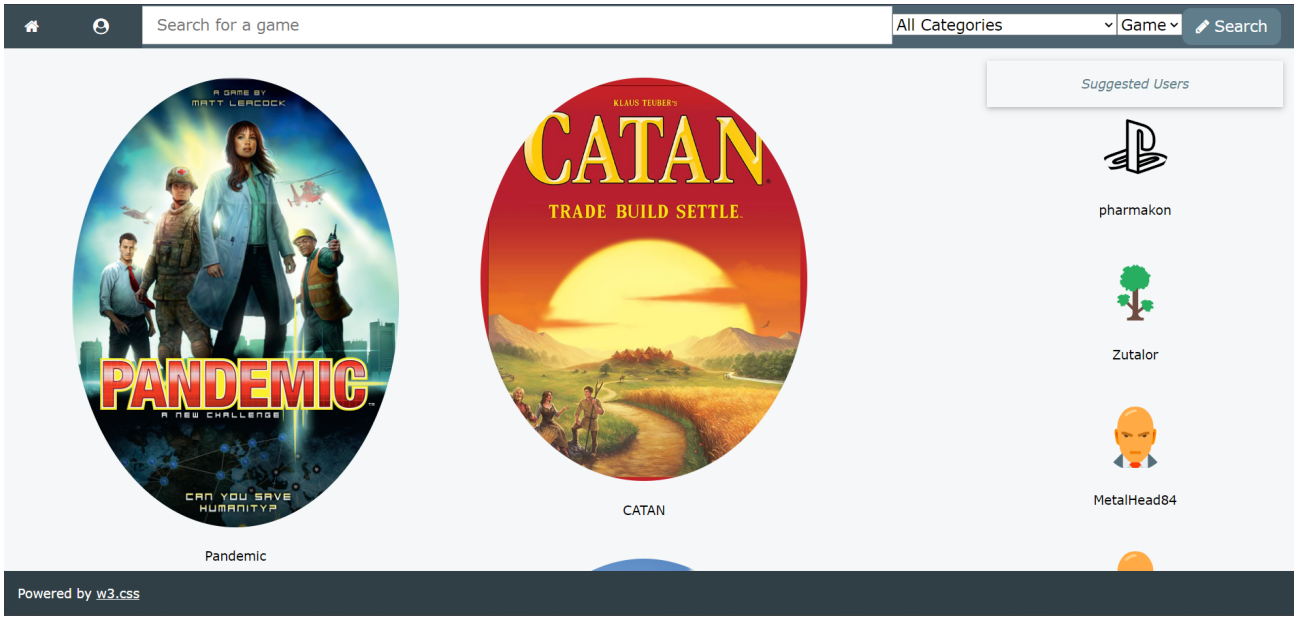

Figure 1: Homepage (Different between different users).

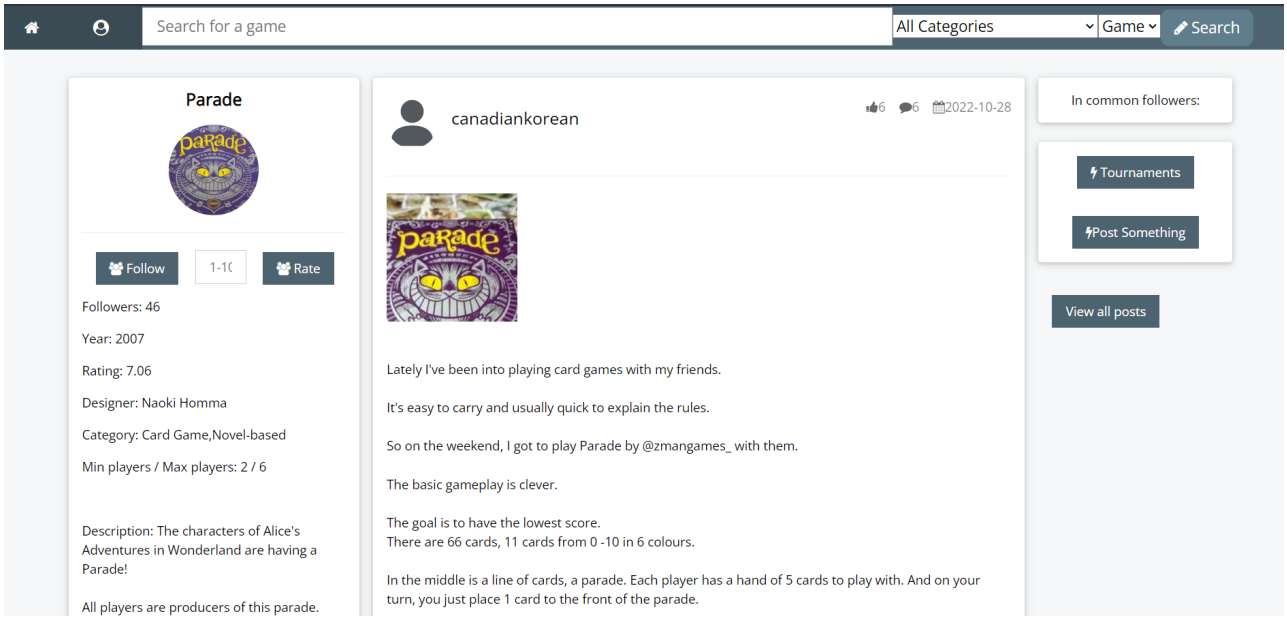

Figure 2: An example of game page.

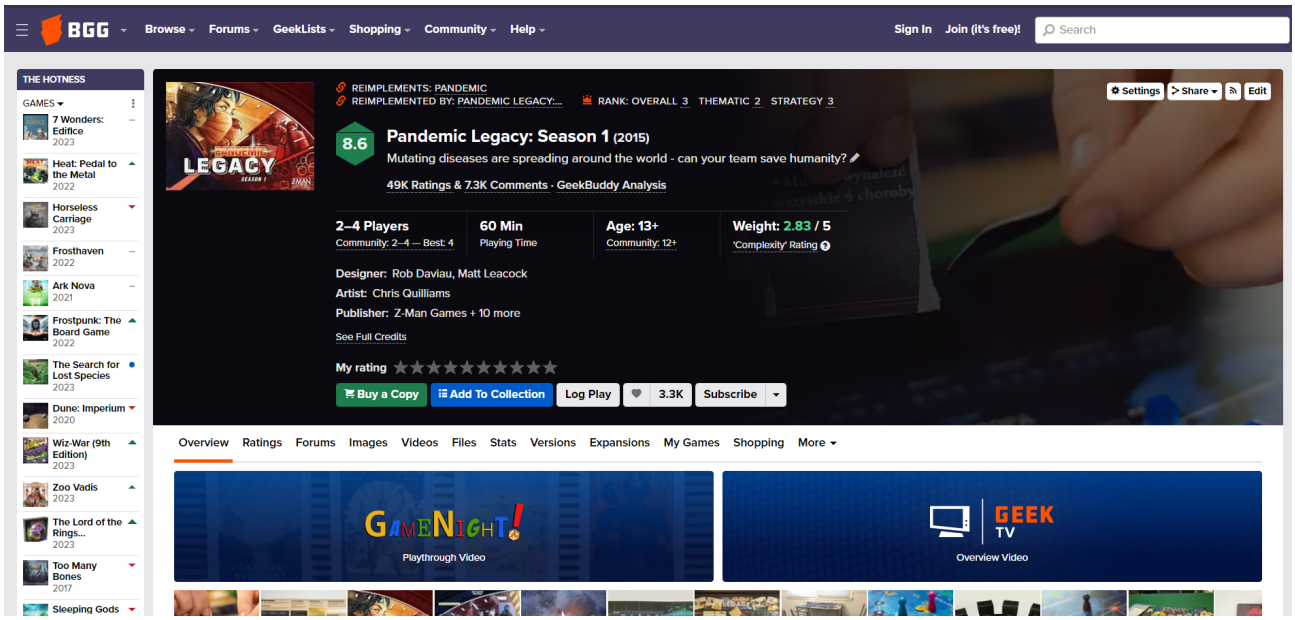

Figure 3: BGG Game Page

2.2.2 Posts and comments

Board Game Geek has an entire section, "forums", dedicated to discussions regarding the game. These discussions are called *Threads*. We name these Threads *post*, and every answer to the Thread a *comment*. In this way we were able to scrape all Posts and Comments we need using a Python script that used the official API made available by boardgamegeek.com to gather all the info we need. For simplicity, we scraped a sample of posts (15k Posts with **all** relative comments) from the games scraped at the previous task.

2.2.3 Users

Regarding users, again we used the BGG's API to get them. In this case, we had some problems, because we tried to scrape every user that own a post or a comment that we previously scraped, but due to request blocks we were able to scrape only 8.500 users. So, we will have posts that are not associated to an existing user in our DB, and to handle with that we will assume that these users don't belong to the website more.

2.2.4 Tournaments

Regarding tournaments, we didn't have the same luck we had for the other things, because our source of tournaments, Board Game Arena, doesn't publish any API. So, we manually scraped data from tournament pages (4). We couldn't scrape data regarding participants' names, because otherwise these will never match our users (scraped from another source). To handle with that, we decided to randomly draw, without replacement, a number of our users equal to *number of participants* and we forced them to join that tournament.

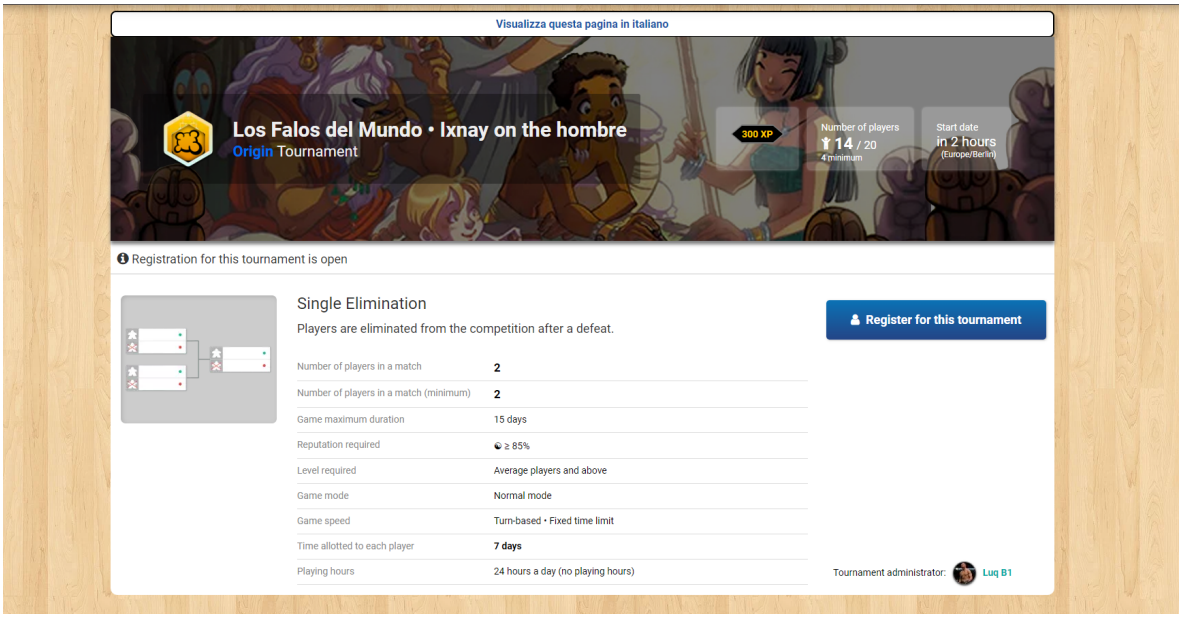

Figure 4: BGA Tournament Page

2.3 Useful scripts and functions

Before proceeding, we want to state that the data we have scraped are not ready for the construction of databases. In particular, we reasoned about the fact that we don't have any follow relationship between user-user and user-game, and no ratings of users over games. All this files and mini-projects are free to view inside the repository.

2.3.1 Most recent posts

We made a script that keeps a number of posts of a specific author and insert it as embedded into `user.json` and `game.json`. To reason about why we decide to embed duplicated information and how many posts are we embedding, see Section 4.

2.3.2 Followers and participants

We made a script that randomly choose a pool of users and assign them to a specific tournament. Moreover, we use a Normal Gaussian Distribution to randomly generate who-follows-who. We will use later this results to populate Neo4j Graph.

2.3.3 Ratings

We scraped the average rate that every game has, but we couldn't know who-rated-what. So, we used again a Normal Gaussian Distribution with a fixed mean (on the average real rate) and 1-variance, to generate couples user-rate. In this way, we randomly chose who rates, but we leave the real rate that a game have.

3 Design

In this section we explain better what our application is composed by, and we try to present an overview of quite complete functional and non-functional requirements that our application satisfies. We will then describe in later sections what architecture choices are led from this premise.

3.1 Main actors

Starting by analysing main actors of our application, we talk about **Unregistered Users**, **Registered Users** and **Admin**:

- **Unregistered Users** are the ones we could call "visitors", those that are not registered on the website but wants anyway to have a look at the website. A visitor should be able to see something that may "attract" them to join the application.
- **Registered Users** are the ones that decided to join our website. They should have a complete access to the website functionalities, except for analytic functions.
- **Admins** are those that rule the website, with the power of banning users, deleting posts and creating games. They can even see analytic functions to eventually talk with the team to propose changes to the application.

3.2 Requirements

3.2.1 Functional Requirements

The system must allow an **Unregistered User** to:

1. Join the social network by providing his/her personal information.
2. Browse games with no filter.
3. Browse games' categories.
4. Browse games by category.
5. Select a game:
   (a) Once a game has been selected, View a description of the game.
   (b) Once a game has been selected, View the number of users who followed that game.
   (c) Once a game has been selected, View the game's related posts.
   (d) Select a post:
      i. Once a post has been selected, View its related comments
6. Find an user by its username.

7. Select an user:
   (a) Once an user has been selected, View a brief description of the user.
   (b) Once a user has been selected, View the number of followers.
   (c) Once an user has been selected, View the user's related posts
   (d) Select a post:
      i. Once a post has been selected, View its related comments.

The system must allow a **Registered User** to:

1. Login into the social network by providing his/her username and password.
2. Browse games with no filter.
3. Browse games' categories.
4. Browse games by category.
5. Select a game:
   (a) Once a game has been selected, View a description of the game.
   (b) Once a game has been selected, Follow its page.
   (c) Once a game has been selected, Create a post.
   (d) Once a game has been selected, View the number of users who followed that game.
   (e) Once a game has been selected, View the game's related posts.
   (f) Select a post:
      i. Once a post has been selected, View its related comments.
      ii. Once a post has been selected, Like that post.
      iii. Once a post has been selected, Comment the selected post.
      iv. Once a post has been selected, Select the user who wrote the post
         A. Once a user has been selected in this way, he can perform all actions listed below (7).
   (g) Once a game has been selected, Create a Tournament.
   (h) Once a game has been selected, View the list of Tournaments regarding the game.
      i. Once a Tournament has been selected, View full details about the tournament.
      ii. Once a Tournament has been selected, Join it (if not closed).
      iii. Once a Tournament has been selected, View the list of the participants.
      iv. Once a Tournament has been selected, Leave it (if joined).
      v. Once a Tournament has been selected, Dismiss (if created).

6. Find an user by its username.
7. Select an user:
   (a) Once an user has been selected, View a brief description of the user.
   (b) Once a user has been selected, View the number of followers.
   (c) Once an user has been selected, View the most recent posts that user has written.
   (d) Select a post:
      i. Once a post has been selected, View its related comments.
      ii. Once a post has been selected, Like that post.
      iii. Once a post has been selected, Comment the selected post.
      iv. Once a post has been selected, Go to the game's page in which the post was written.
   (e) Once a user has been selected, View the list of common games.
   (f) Once a user has been selected, Follow him/her.
   (g) Once a user has been selected, View a list of common followers.
   (h) Once a user has been selected, View a list of common joined Tournaments.
8. Access his personal page:
   (a) Once in the profile page, View his number of followers.
   (b) Once in the profile page, View his most recent posts.

The system must allow an **Admin** to behave like a *Registered User*, and so to do all things listed above. In addition the system must allow to:

1. Delete a Game.
2. Delete a Post.
3. Ban an user.
4. Access an *Admin Page*:
   (a) Once the Admin Page has been accessed, Create a Game page.
   (b) Once the Admin Page has been accessed, View the best rated game of a fixed category.
   (c) Once the Admin Page has been accessed, View the worst rated game of a fixed category.
   (d) Once the Admin Page has been accessed, View the distribution of comments and post by month.
   (e) Once the Admin Page has been accessed, View the distribution of joining among continents.

3.2.2 Non-Functional Requirements

The non-functional requirements are listed below:

- The system must be a website application.
- Tolerance to the loss of data.
- Avoid, if possible, a single point of failure.
- High Availability, accepting eventually older versions of data.
- The code must be written in a Object Oriented Programming language.

3.3 Use-Case Diagram

Figure 5 shows the Use-Case diagram. You can find the `.svg` file, to have a better visualization experience, in our GitHub repository.

3.4 UML Class Diagram

We inserted snapshot (6 of the simplified use-case Diagram, even if you can find the complete one in the Repository

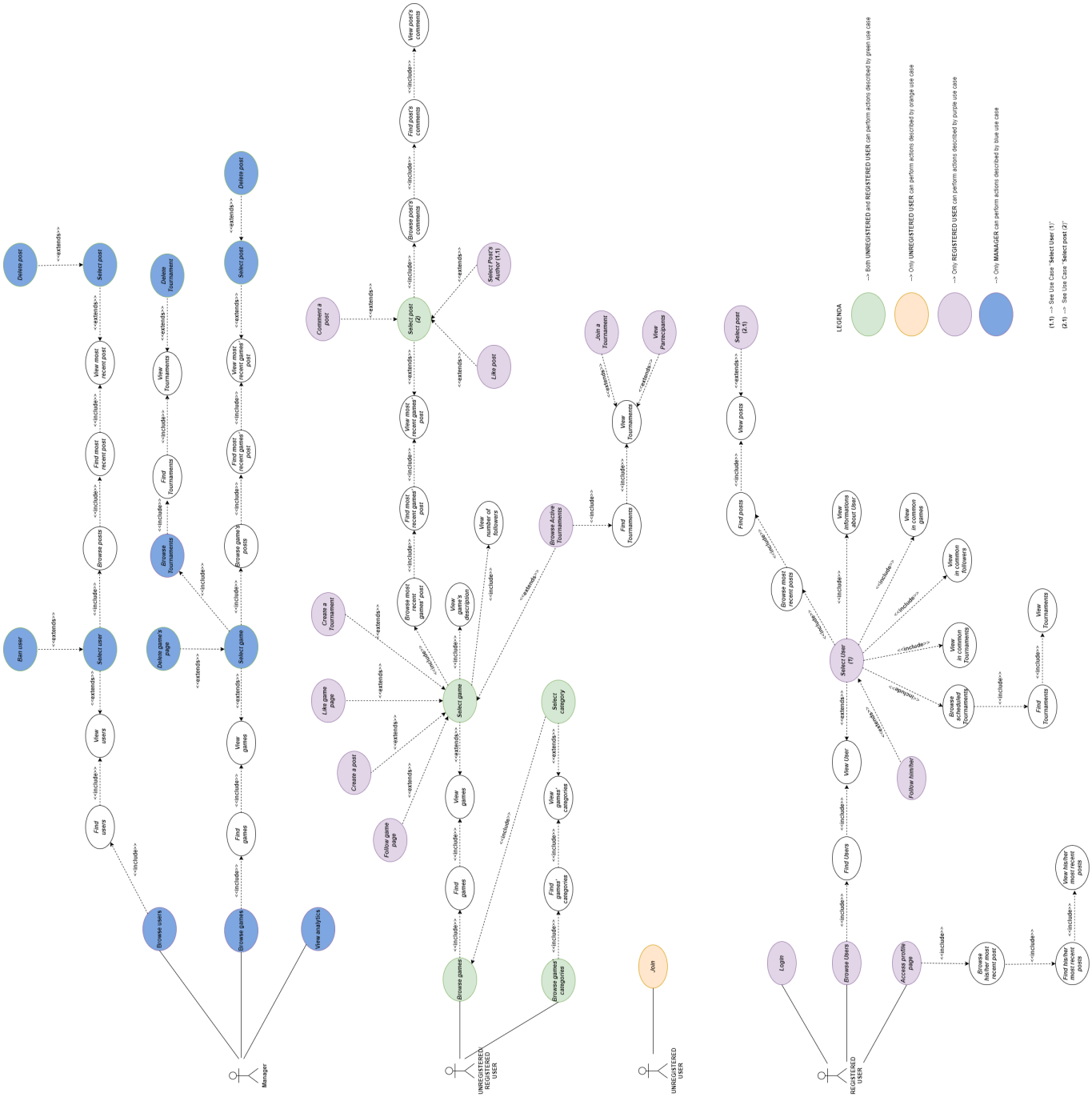

Figure 5: Use Case Diagram.

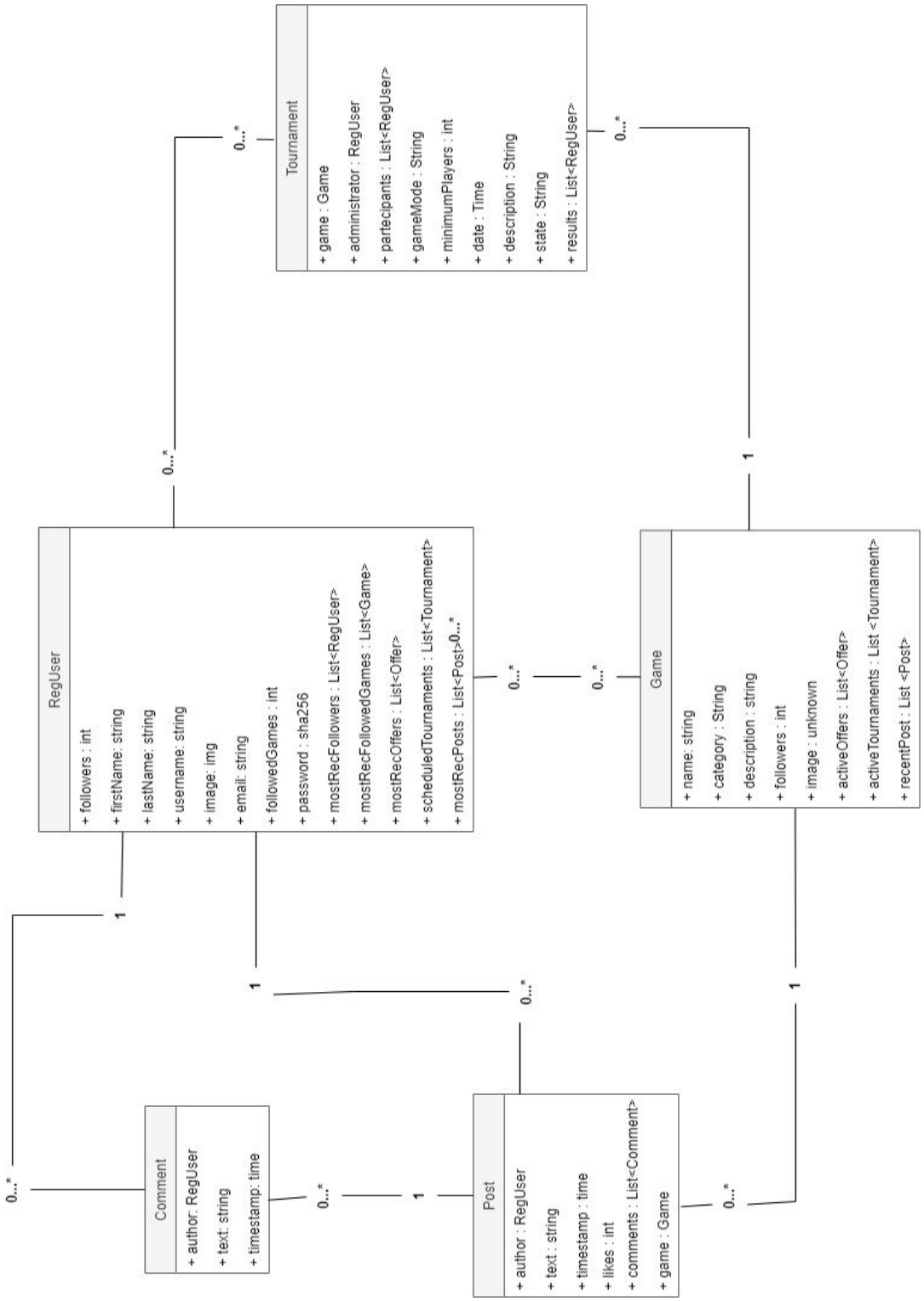

Figure 6: A simplified Snapshot of UML Class Diagram

4 Data Modeling

4.1 Which Databases we are going to use

We decided to use both `MongoDB` and `Neo4j` to handle our entities. The idea of using a graph database derived by the fact that our application has some social-network features like following and recommendation systems.

4.1.1 Comparison with RDBMS

Let's have a look at what we should have done if we had at our disposal only SQL to represent a sub-part of our project, the information of games. Right now games are collected both in `Neo4j` and in `MongoDB`. `Neo4j` stores the relations between users following game and game of tournaments. `MongoDB` on the other hand stores all the static information of a game: name, categories, description, image, recent Posts, ecc. This means that, to display a game page, we have to access only 1 collection (and only 1 document), which is the collection game and the specific game document. To display the in common information (users following the game in common with you) we make access to Neo4j.

Let's have a look now at what we should have done if we had to use only a relational DBMS. We'll consider a 3NF. To display the game page, we would need these tables:

- GAME (game_name, category, description, image, designer, year_published...)
- POST (post_id, game_name, creator, title, text)
- FOLLOW_GAME (username, game_name)
- FOLLOW_USER (username, username)
- TOURNAMENT_GAME (game_name, tournament_id)
- TOURNAMENT (tournament_id, description, date)
- USER (username, image)
- RATINGS (username, game_name)

We would need approximately 8 tables to obtain the information that we need to display. It's easy to understand that joining these 8 tables would be computationally expensive, especially if the size of the database grows much. This is a common problem that relational database have when dealing with object oriented programming. This also shows that `MongoDB` is a natural option to store information about a game.

4.2 Reasons to use `Neo4j`

As we studied, *graph database* are suitable for problem domains easily described in terms of entities and relationship. Here the problem is describing relations between users and other users or users and games (*follow*), and relationship with tournaments (*participate*). These aspects guided us into deciding to adopt `Neo4j` to be responsible of storing information of relationship between entities in our project.

4.3 A little volume study to select the most relevant queries

To better understanding how to organize collections, and how to select the most relevant queries, we introduce now a little table of operations common done every day by a user that access the website. The website is really readings-heavy and pretty interactive, and so we consider only data retrieval operations and common interactive operations. On the column *total volume* we make the assumption of 10'000 users logging in our website every day.

Operation	Frequency(per Day)	MongoDB Operations	GraphDB Operations	Total Volume
Accessing the Homepage	3	0	6	6*10000
Searching for Game/User	20	20	0	20*10000
Seeing Most Recent Post of a Game/User	20	0(*)	0	0
Seeing all Posts of a Game	3	3	0	3*10000
See followers of a Game/User	20	0	3	3*10000
See common followers when in a GamePage/UserPage	20	0	2	2*10000
Seeing all informations about a tournament	4	0	3	3*10000
Comment a Post	4	12(*)	0	12*10000
Like a Post	5	15(*)	0	15*10000
Follow a Game or an User	5	0	5	5*10000
View all comments of a post	30	30	0	30*10000

(*) Due to the frequency of accessing to most recent post of users and games, we decided to embed them (5) in the correspondent collections (see below). Notice that doing in this way we have no accesses (because information are already gathered in the searching operations) and this overcomes the penalty in accesses of likes and comments (due to duplicated data).

4.4 Collections

We have thought about three collections inside `MongoDB`. These collections cover the entities **User**, **Game**, **Post** and , and allow all functionalities of the software except for two functions entirely handled by GraphDB (see below).

4.4.1 User

Inside the **User**'s collection we store all information his/her provided during signup process. You can see a snapshot of the document structure in the figure (7). From these information we load all things we need in the user page, and we decided to embed the most recent posts written by the user thinking about the fact that rarely we will want to see all posts regarding him/her, but frequently we will want to access the last posts he/her has written, as described in the paragraph 4.3.

4.4.2 Game

Inside the **Game**'s collection we store all information an Administrator inserts during the creation of a game and we reflect data structure we have scraped from boardgamegeek.com. You can see a snapshot of the document structure in the figure (8). From these information we load all things we need in the game page and

```
{
  "_id": {
    "$oid": "63d315f408208372609ad3e4"
  },
  "username": "federic0casu",
  "password": "c0dd9b0ce23e54bb55cce85f1daa59ddb4a4650630825d5bfc39bb77c094dca4aa64c448445afa47",
  "firstname": "Federico",
  "lastame": "Casu",
  "yearregistered": 2023,
  "email": "f.casu@studenti.unipi.it",
  "stateorprovince": "Pisa",
  "country": "Italy",
  "continent": "Europe",
  "most_recent_post": [],
  "admin": false,
  "_class": "it.unipi.BGnet.model.User"
}
```


Figure 7: User Document.

we decided, as for users, to embed the most recent posts. An `$unwind` operation in this document is necessary whenever an Administrator wants to analyse the best and the worst category of the website.

4.4.3 Post

Inside the collection **Post** we store all information regarding one of them, including the Game it is referred to and the user who wrote it (Figure 9). Notice that there's no, in this way, reason to access collections User and Game to show a Post, because duplicating the name of the game and the username of the user who wrote it allows us to show the minimal information a visitor is interested in.

4.5 Concepts Handled By Graph

In the following section we're going to analyze which concepts and entities are handled by Neo4j.

4.5.1 Games

The entity game has 3 properties: the name of the game, the url of the image and an array of categories. The name of the game is the identifier of the entity itself. The url is necessary to show the image of the game when suggesting it, and categories are also needed for suggestions' purposes. This entity can have 2 types of relationship: FOLLOWS, between user and game, and TOURNAMENTGAME between tournament and game.

4.5.2 Users

The entity user has 2 properties: the name of the user, which is used as unique identifier, and the url of the image, which is necessary when showing in common followers between users and suggestions. The entity can have 4 type of relationship: FOLLOWS a game or another user, PARTICIPATE to a tournament, and CREATE a tournament.

```
_id: ObjectId('63bdb97d4f154812d674099c')
name: "Merchants & Marauders"
nrate: "13635"
designer: "Kasper Aagaard"
yearpublished: "2010"
minplayers: "2"
maxplayers: "4"
playingtime: "180"
minplaytime: "180"
maxplaytime: "180"
▾ categories: Array
    0: "Adventure"
    1: "Fighting"
    2: "Nautical"
    3: "Pike and Shot"
    4: "Pirates"
    5: "Transportation"
description: "Merchants &amp;amp; Marauders lets you live the life of an influential mer…"
img: "https://cf.geekdo-images.com/RmKYojYHmV8X-ebtNgaz-g__original/img/wrLH…"
▸ ratings: Array
▸ most_recent_post: Array
```


Figure 8: Game Document.


```
_id: ObjectId('63c0601e5217b04263319d00')
id: 41063917
author: "canadiankorean"
game: "Parade"
subject: "Who doesn't like a Parade?  A quick review of this neat card game"
text: "<div style=''><a href="https://boardgamegeek.com/image/7150301"><img s…"
timestamp: "2022-10-28T22:37:09-05:00"
▾ likes: Array
    0: "Sphere"
    1: "loverman21"
    2: "saroz162"
    3: "Moose Malloy"
    4: "yangtze2000"
▾ comments: Array
  ▾ 0: Object
      author: "Sphere"
      text: "<font color=#2121A4><div class='quote'><div class='quotetitle'><p><b>c…"
      timestamp: "2022-10-30T01:17:54-05:00"
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
  ▸ 5: Object
```


Figure 9: Post Document

4.5.3 Tournaments

We have chosen to keep the entire Tournament entity inside `Neo4j`. The other possibility was to store it in MongoDB (at least store the data of a tournaments inside MongoDB and the relationships in Neo4j). The main concept behind this choice is the fact that we consider tournament to be a strongly dynamic entity. Tournaments do not carry that much information in them, if not the users that are participating, or that have participated to the tournament itself.

Another possibility could be to store recent tournaments inside `Neo4j` and to keep the old ones inside a collection in `MongoDB`. Here we made another assumption, which is that tournaments won't grow that much in size to be a relevant problem. If in future there will be a reason to deal with the size of tournaments, we could think to delete the oldest ones, or to move them into a collection.

The entity has 6 properties: date, duration, isClosed, maxPlayers, modalities and playersPerMatch. IsClosed is a boolean property which indicates if the tournament is already finished or not. This property could be obtained also by summing the date of the tournament with the duration, but in order to keep some flexibility we decided to let the creator only to close the tournament.

This entity can be linked with 3 types of relationships: PARTICIPATE, the game of the tournament (TOURNAMENTGAME), and with the creator of the tournament (CREATED).

Main queries on Graph database:

Domain specific	Graph-centric
Find the tournaments in common between "userA" and "userB"	Which tournaments' vertexes have an incoming edge of type `PARTICIPATE` starting from "userA" and incoming edge, of the same type, starting from "userB".
Find the number of followers of "userA"	How many `FOLLOWS` edges are incoming into the "userA" vertex?

5 Implementation

5.1 Frameworks

We have decided to use the Spring Framework to handle the entire project:

- We used the Spring-Dependency Injection via the `@Autowired` annotation.
- We used the `@SessionAttributes` annotation to save session attributes during the project (like logged status or current game selected).
- We used `@Controller`, `@RestController`, `@Service`, `@Repository`, `@GetMapping` to link functions from front-end to back-end.

5.2 Package Organization

We divided our project in the following packages:

5.2.1 `it.unipi.BGNet.controllers`

Here are grouped all Class labeled with the annotation `@Controller`, which are used to return the html pages upon a get request. We have a controller class for each page (Figure 10). Below we list all controller classes we have introduced, the returned page is included in the name of the class:

- `AdminController`
- `CommentPageController`
- `GamePageController`
- `HomeController`
- `LoginController`
- `SignUpController`
- `PostController`
- `ProfileController`
- `SearchPageController`
- `TournamentController`

```java
@Controller
@SessionAttributes("sessionVariables")
public class HomeController {
    @RequestMapping("/")
    public String home(Model model) {
        if(model.getAttribute("sessionVariables") == null)
            model.addAttribute("sessionVariables", new SessionVariables());
        return "home";
    }
}
```

Figure 10: HomeController, the `@Controller` that returns the landing page.

5.2.2 `it.unipi.BGNet.controllers.api`

Here are grouped all class labeled with the annotation `@RestController`, which are used to return DTOs and answers to /api/* requests (Figure 11). Below we list all `@RestControllers` we have introduced with relative functions:

- `AdminCRUDController` - It answers to the admin request to create a game, to delete a game, to delete a post or to ban an user.
- `AuthController` - It answers to the login request, and checks if the current user is logged or is an admin.
- `SignupController` - It answers to the signup request.
- `GameController` - It answers to all requests regarding a game: it looks for its existence, load its game page, handle follow and rate request. It even handle search requests.
- `LandingController` - It answer to the request of building the landing page, loading suggestions regarding Users and Games.
- `LoadCommentController` - It answers to the request of adding a comment to a post.
- `LoadPostController` - It answers to the request of getting post lists, adding a new post or like/unlike it.
- `LoadProfileController` - It answers to the requests of loading an user page, or to follow/unfollow an user
- `LoadTournamentController` - It answers to the requests regarding tournaments: participate, close, get participants, create and quit.

5.2.3 `it.unipi.BGNet.model`

In this package we list all models that reflects the back-end. Models are only accessible from `Service` and `Repository` levels. We implemented a model for the main entities of our application: Game (Figure 12), Post (Figure 14), Comment (Figure 15), Tournament (Figure 16) and User (Figure 13).

```java
@RestController
@SessionAttributes("sessionVariables")
public class GameController {
    @Autowired
    GameService gameService;
    @GetMapping("/api/loadGamePage")
    public @ResponseBody String loadGamePage(Model model){
        SessionVariables sv = (SessionVariables) model.getAttribute("sessionVariables");
        GamePage page = gameService.getGamePage(sv.myself, sv.gameToDisplay);
        Gson gson = new GsonBuilder().serializeSpecialFloatingPointValues().create();
        return gson.toJson(page);
    }
    @GetMapping("/api/gamePageExists")
    public @ResponseBody String gamePageExists(Model model, @RequestParam("name") String name){
        List<Game> list = gameService.checkExistence(name);
        int counts = list.size();
        if(counts > 0) {
            if(counts > 1) {
                SessionVariables sv = (SessionVariables) model.getAttribute("sessionVariables");
                sv.current_results = list;
                return new Gson().toJson(counts);
            }
            else {
                if(list.get(0).getName().equals(name)) {
                    SessionVariables sv = (SessionVariables) model.getAttribute("sessionVariables");
                    sv.gameToDisplay = name;
                    model.addAttribute("sessionVariables", sv);
                    return new Gson().toJson(1);
                }
                else {
                    SessionVariables sv = (SessionVariables) model.getAttribute("sessionVariables");
                    sv.current_results = list;
                    return new Gson().toJson(0);
                }
            }
        }
        return new Gson().toJson(-1);
    }
```

Figure 11: `GameController`, an example of `@RestController`.

```java
@Data
@NoArgsConstructor
@Document(collection = "game")
public class Game {
    @Id
    private String id;

    private String name;

    private String designer;

    @Field("yearpublished")
    private int yearPublished;

    @Field("minplayers")
    private int minPlayers;

    @Field("maxplayers")
    private int maxPlayers;

    private String playingTime, minPlayTime, maxPlayTime;

    private List<String> categories;

    private String description;

    private String img;

    private List<String> ratings;

    @Field("most_recent_post")
    private List<Post> mostRecentPosts;

    private List<String> followers;

    @Field("avg_rate")
    private float avgRate;

}
```

Figure 12: Game Model

```java
@Document(collection = "user")
public class User {
    @Id
    private String id;

    @NotBlank
    @Size(max = 20)
    private String username;

    @NotBlank
    @Size(max = 100)
    private String password;

    @NotBlank
    @Size(max = 20)
    @Field("firstname")
    private String firstName;

    @NotBlank
    @Size(max = 20)
    @Field("lastame")
    private String lastName;

    @Field("yearregistered")
    private int yearRegistered;

    @NotBlank
    @Size(max = 100)
    @Email
    private String email;

    @Field("stateorprovince")

    private String stateOrProvince, country, continent, img;

    @Field("most_recent_post")
    private List<Post> mostRecentPosts = new ArrayList<>();

    @Field("admin")
    private boolean admin;

}
```

Figure 13: User Model

```java
@Data
@NoArgsConstructor
@Document(collection = "post")
public class Post {
    @Id
    private String id;

    private String author;

    private String game;

    private String text;

    private List<String> likes;

    private String timestamp;

    private List<Comment> comments;

}
```

Figure 14: Post Model

```java
@NoArgsConstructor
public class Comment {
    private String author;

    private String text;

    private String dateTime;

}
```

Figure 15: Comment Model

```java
public class Tournament {
    private int id;

    private String date;

    private String duration;

    private int maxPlayers;

    private String modalities;

    private String playersPerMatch;

    private List<String> partecipants;

    private String tournamentGame;

    private String creator;

    private boolean isClosed;
}
```

Figure 16: Tournament Model

5.2.4 `it.unipi.BGNet.DTO`

Inside this package we list all DTOs that are passed to the controller to help the front end to build pages.

- `Gamepage` - Aggregation of games information we show in game page composed of game description, number of followers, most recent posts, common followers and status variables checking if you already followed or rated the game
- `PostDTO` - Aggregation of post information composed of author, timestamp, text, number of likes and comments
- `GameDTO` - Aggregation of games information we show in result page composed of name, designer, min/max players and categories.
- `UserDTO` - Aggregation of user information we show in the user page. We show information, most recent posts, in common followers and tournaments, number of followers and status variable checking if you already follow that user.
- `TournamentDTO` - Aggregation of tournament information that we show when button tournaments is pressed in the game page. It shows date, duration, maximum number of players, modalities, players per match, number of partecipants, played game, creator. We have 3 status variables: `isClosed` to see if the tournament is closed or open, `isParticipating` to check if the user is already participating, and `isCreator` to check if the current user is the creator or not.

- `InCommonGenericDTO` - Aggregation of information that need to be showed when accessing a game page or a user page, to display some information about in common followers. It shows the username/gamename and the image.

5.2.5 `it.unipi.BGNet.service`

Inside this pages there are collected all classed demanded to service level. Their role is to operate as a bridge between `@RestController` and `@Repository` classes. Controller classes demand at the service a functionality, and the Service-level demands for all necessary repository functions. At the end, these classes ensemble the DTOs, whose details are explained above.

5.2.6 `it.unipi.BGNet.repository`

Inside this package we inserted the `@Repository` level of class Game and User, because they share some low-level functions in both `Neo4j` and `MongoDB`. We inserted TournamentRepository too because it elaborates lower-level functions from TournamentNeo4j.

- `GameRepository` - It handles repository functions, regarding CRUD operations, for a game. Moreover here we can find all repository functions regarding recent posts inside that game (add a new one, update, remove, add a comment) and repository functions regarding graph operations (follow, suggestions).
- `UserRepository` - It handles repository functions (CRUD operations) regarding an user. We can find all repository functions regarding recent posts inside that user (add a new one, update, remove, add a commend) and repository functions regarding graph operations (follow, suggestions).
- `TournamentRepository` - It handles repository function to CRUD operations regarding a tournament.

5.2.7 `it.unipi.BGNet.repository.mongoDB`

In this section we have inserted classes that manage repository level for classes Post and Comment:

- `PostRepository` - Here we insert all CRUD operations regarding posts.
- `CommentRepository` - Here we insert all CRUD operations regarding comments, considering they are always embedded in a *Post* entity.

5.2.8 `it.unipi.BGNet.repository.neo4j`

In this section we can find a lower-level implementation of method declared in the graph section of .repository package. The graphs operations are detailed in Cypher Language, and the most relevant ones are detailed at Section **??**.

5.2.9 it.unipi.BGNet.utilities

Inside this package we put two classes:

- Inside class `Constants` we put PAGE-SIZE and RECENT-SIZE variables, that keeps the number of posts in a page and the number of posts embedded in an User or in a Game document
- Inside class `SessionVariables` we put all session variables we need in our project: myName, status, the current game to display and the current List of game to return in the results page.

5.3 MongoDB Relevant Operations

We put here implementation of both some common operations described at 3.3. and Admin Side analysis functions.

5.3.1 Loading GamePage (MongoDB side)

To load a game page the application calls the following methods:

1. `public @ResponseBody String loadGamePage(Model model)`
2. `public GamePage getGamePage(String myself, String gameName)`
3. `public Optional<Game> getGameByName(String gameName)`
4. `public List<InCommonGenericDTO> findInCommonFollowers(String username, String gamename)`
5. `public boolean isFollowing(String username, String gamename)`
6. `public int getFollowersNumberByGamename(String gamename)`

The first method is a `@RestController` and its task is to handle the get request perfomed when a user access the game page. Once the request has been handled by the `@RestController`, the application calls the `@Service` associated to the game entity. In particular, the method `loadGamePage()` calls the Game's `@Service` `getGamePage(String myself, String gameName)`. At this point the Game's `@Service` uses the methods made available by Game's `@Repository`: to build up the game page it is needed to call `getGameByName(String gameName)`.

```
public Optional<Game> getGameByName(String name){
    Optional<Game> game = Optional.empty();
    try {
        game = gameMongo.findByName(name);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return game;
}
```

Method `findByName(String gameName)` is made available by `MongoRepository<Game, String>` Spring interface. The last method implements for us the following query:

```
db.game.find{(name: "gameName")}
```

Methods 4, 5 and 6 calls `Neo4j` repositories and they are described later.

5.3.2 Like a Post

When a user like a post the application handles this event calling the following methods:

1. `public @ResponseBody String likePost(Model model, @RequestParam(value = "post") String post, @RequestParam(value = "game") String game)`
2. `public int likeUnlikePost(String id, String username, String game)`

Similarly to what said for loading a game page, the first method is a `@RestController` and its task is to handle the get request perfomed when a user likes a post. Once the get request has been captured, the `@RestController` method `likePost()` calls `likeUnlikePost()`. The latter is made available by the Post's `@Service` class.

```
public int likeUnlikePost(String id, String username, String game) {
    Optional<Post> older = postRepo.getPostById(id);
    if(older.isEmpty())
        return -1;
    boolean flag = false;
    for(String like : older.get().getLikes())
        if(like.equals(username))
            flag = true;
    Post saved;
    if(flag)
        saved = postRepo.unlikePost(older.get(), username);
    else
        saved = postRepo.likePost(older.get(), username);
    if(saved == null)
        return -1;
    gameRepo.updatePost(game, older.get(), saved);
    userRepo.updatePost(older.get().getAuthor(), older.get(), saved);
    return (flag) ? 2 : 1;
}
```

We recall that, since we have introduced the redundancy "most recent posts" in collections `User` and `Game`, we need to verify if the post is included in both user's most recent posts and game's most recent posts. Once

the like has been added / removed from the post we need to propagate the update to both game and user collections.

```
public boolean GameRepository::updatePost(String name, Post olderPost, Post newPost) {
    Optional<Game> game = getGameByName(name);
    if(game.isEmpty())
        return false;
    List<Post> list = game.get().getMostRecentPosts();
    for(Post post: list)
        if(post.getId().equals(olderPost.getId())) {
            list.set(list.indexOf(post), newPost);
            break;
        }
    game.get().setMostRecentPosts(list);
    try {
        gameMongo.save(game.get());
    } catch(Exception e){
        e.printStackTrace();
        return false;
    }
    return true;
}
public boolean UserRepository::updatePost(String name, Post olderPost, Post newPost) {
    Optional<User> user = getUserByUsername(name);
    if(user.isEmpty())
        return false;
    List<Post> list = user.get().getMostRecentPosts();
    list.forEach((post) -> {
        if (post.getId().equals(olderPost.getId()))
            list.set(list.indexOf(post), newPost);
    });
    user.get().setMostRecentPosts(list);
    try{
        userMongo.save(user.get());
    }catch(Exception e){
        e.printStackTrace();
        return false;
    }
    return true;
}
```

5.3.3 Analytic regarding categories (Admin Side)

Ad Administrator can query the database to get the worst rated game and the best rated game for each category. The query is traduced, using Spring Classes, in the following method:

```
public List<AnalyticDTO> findBestAndWorstGamesForCategory() {
    UnwindOperation unwindOperation = unwind("categories");

    ProjectionOperation selectAvgs = project().andExpression("name").as("name")
            .andExpression("categories")
            .as("category").andExpression("avg_rate").as("avg")
            .andExclude("_id");

    SortOperation sortOperation = sort(Sort.by(Sort.Direction.ASC, "avg"));

    GroupOperation groupOperation = group("$category")
            .last("$name").as("BestRatedGame")
            .last("$avg").as("BestRate")
            .first("$name").as("WorstRatedGame")
            .first("$avg").as("WorstRate");

    ProjectionOperation projectionOperation = project()
            .andExpression("_id").as("field1")
            .andExclude("_id")
            .andInclude("BestRatedGame")
            .andInclude("WorstRatedGame");

    Aggregation aggregation = newAggregation(unwindOperation, selectAvgs, sortOperation,
            groupOperation, projectionOperation);

    AggregationResults<AnalyticDTO> result = mongoOperations
            .aggregate(aggregation, "game", AnalyticDTO.class);

    return result.getMappedResults();
}
```

This is the equivalent for the `MongoDB` Aggregation you can find in Figure 17 and that follows these stages:

1. Firstly we need an `$unwind` stage from which we unpack the categories array.
2. Then a `$project` and `$sort` stage to sort the games by their average rate.

3. With a `$group` stage by the field *category*, we select the first and the last element of each group, and we `$project` names in the last stage.

5.3.4 Analytic regarding Post' distribution (Admin Side)

An administrator can query the database to get the distribution of posts and the mean number of comments for each post among each month of the input year. This is done with the following method:

```
public List<AnalyticDTO> getPostAndCommentDistribution(int year){
    ProjectionOperation commentSize = project()
            .andExpression("$comments").size().as("howManyComments")
            .and(year(DateOperators.dateFromString("$timestamp"))).as("year")
            .and(month(DateOperators.dateFromString("$timestamp"))).as("month");

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

    GroupOperation groupMonths = group("month")
            .count().as("posts")
            .sum("howManyComments").as("comments");

    ProjectionOperation projectFields = project()
            .andExpression("_id").as("field1")
            .andExclude("_id")
            .andExpression("posts").as("field2")
            .and(ArithmeticOperators.Divide.valueOf("comments")
            .divideBy("posts")).as("field3");

    SortOperation sortByMonth = sort(Sort.by(Sort.Direction.ASC, "field1"));

    Aggregation aggregation = newAggregation(commentSize, matchYear,
        groupMonths, projectFields, sortByMonth);

    AggregationResults<AnalyticDTO> result = mongoOperations
            .aggregate(aggregation, "post", AnalyticDTO.class);

    return result.getMappedResults();
}
```

This is the equivalent for the MongoDB Aggregation you can find in the fig(18) and that follows these stages:

1. In the first `$project` stage we compute the year and the month of each post.

```
db.game.aggregate([
    {
    $unwind: "$categories"
    },
    {$project:
        {
        name:1, category: "$categories", avg:1
        }
    },
    {
    $sort: {avg: 1}
    },
    {
    $group:
        {
        _id: {category: "$_id.category"},
        biggestGameRate: {$last: "$_id.name"},
        biggestRate: {$last: "$avg"},
        lowestGameRate: {$first: "$_id.name"},
        lowestRate: {$first: "$avg"}
        }
    },
    {
    $project:
        {
        _id:0,
        category: "$_id.category",
        biggestGameRate: 1, lowestGameRate: 1
        }
    }
])
```

Figure 17: Categories Analysis

2. In the second `$match` stage we only look at those posts with the same year as the input.
3. Then we `$group` to obtain the number of posts and comments in each month.
4. Then we `$project` to `$divide` comments by posts and obtain the mean number of comments for each post and lastly we `$sort` for visualization purposes.

5.3.5 Analytic regarding User' geographic distribution

An Administrator can query database to get the geographic distribution of the users who decided to join during the input year. Notice that the query returns for each continent the country with the maximum number of joining, and can return a "not specified" group to represent those users that didn't input it during signup process. The method is the following:

```
public List<AnalyticDTO> getBestCountriesByYearRegistered(int year){
    MatchOperation getYear = match(new Criteria("yearregistered").is(year));
    GroupOperation getNumberOfJoining = group("continent", "stateorprovince")
            .count().as("count");

    SortOperation sortByCount = sort(Sort.by(Sort.Direction.DESC, "count"));

    GroupOperation getCountries = group("_id.continent")
            .first("_id.stateorprovince").as("country")
            .first("count").as("people");

    ProjectionOperation renameThings = project()
            .andExpression("_id").as("field1")
            .andExpression("country").as("field2")
            .andExpression("people").as("field3");

    Aggregation aggregation = newAggregation(getYear, getNumberOfJoining, sortByCount, getCountries
    AggregationResults<AnalyticDTO> result = mongoOperations
            .aggregate(aggregation, "user", AnalyticDTO.class);

    return result.getMappedResults();
}
```

This is the equivalent for the MongoDB Aggregation you can find in the fig(19) and that follows these stages:

1. First we `$match` only those users that was registered on the input year.
2. Then we `$group` to get the number of joining for each country.
3. Then we `$sort` by this number and `$group` again to get best countries for each continent.

```
db.post.aggregate([
    {
    $project:
        {
        comment: {$size : "$comments"},
        year: {$year: { $dateFromString:
                { dateString: "$timestamp"}
                }},
        month: {$month: { $dateFromString:
                { dateString: "$timestamp"}
                }}
        }
    },
    {
    $match: {year: 2022}
    },
    {
    $group:
        {
            _id: {mese: "$month"},
            numeroPost: {$sum:1},
            numeroCommenti: {$sum: "$comment"}
        }
    },
    {
    $project:
        {
            _id: 0,
            mese: "$_id.mese",
            quantiPost: "$numeroPost",
            mediaCommenti : {$divide: ["$numeroCommenti", "$numeroPost"]}
        }
    },
    {
    $sort: {mese: 1}
    }
])
```

Figure 18: Post Distribution Analysis.

```
db.user.aggregate([
    {
    $match:
        {
            yearregistered: 2021
        }
    },
    {
    $group:
        {
            _id: {continent: "$continent", country: "$stateorprovince"},
            count: {$sum: 1}
        }
    },
    {
    $sort: {count:-1}
    },
   {
    $group:
        {
            _id: "$_id.continent",
            country: {$first: "$_id.country"},
            people: {$first: "$count"}
        }
    },
    {
    $project:
        {
            _id:0,
            continent: "$_id",
            country: 1,
            people: 1
        }
    }
])
```

Figure 19: User best countries by continent Analysis

5.3.6 Analytic regarding finding most popular users

An administrator can query Neo4j in order to find the 5 most popular users. The popularity is defined as the number of followers of that user + the number of distinct participants to the tournaments he created. The method is the following:

```
public List<Record> getMostPopularUsers(){
    try{
        return graphNeo4j.read("MATCH (ua:User)-[:FOLLOWS]->(ub:User)" +
                " MATCH (ub)-[:CREATED]->(t)" +
                " MATCH (partecipants:User)-[:PARTICIPATE]->(t)" +
                " RETURN ub.name as username," +
                " (COUNT(DISTINCT(ua.name)) +"
                " COUNT(DISTINCT(partecipants.name))) AS popularity" +
                " ORDER BY popularity DESC" +
                " LIMIT 5");
    } catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

5.4 Analytic regarding finding the most popular games

An administrator can query Neo4j in order to find the 5 most popular games, where popularity is defined as the number of followers of that game + the number of tournaments of that game multiplied by 10. The method is the following:

```
public List<Record> getMostPopularGames(){
    try{
        return graphNeo4j.read("MATCH(g:Game)<-[:FOLLOWS]-(u)" +
                " MATCH (g)<-[:TOURNAMENT_GAME]-(t)" +
                " WHERE t.isClosed=false" +
                " RETURN g.name AS gamename, " +
                " (COUNT(DISTINCT(u))+COUNT(DISTINCT(t))*10) AS popularity" +
                " ORDER BY popularity DESC" +
                " LIMIT 5");
    } catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

5.5 Neo4j Relevant Graph-Domain Queries

Some of the most important query performed in Neo4j are listed below.

5.5.1 Finding the tournaments in common between 2 users

This query is in charge of finding the tournaments in common between 2 users. It's useful when we're logged in and we want to display another user's account. We'd like to show if we're both participating in some tournaments, or if we've done it in the past. Tournaments are a very dynamic entity, and can be used to create new friendships in the BGNet site.

```
MATCH (uA:User)-[:PARTICIPATE]->(t:Tournament)<-[:PARTICIPATE]-(uB:User)
WHERE uA.name = userA AND uB.name = userB
MATCH (t)-[:TOURNAMENT_GAME]->(g:Game)
RETURN id(t) as id,
    t.date as date,
    t.duration as duration,
    t.maxPlayers as maxPlayers,
    t.modalities as modalities,
    t.playersPerMatch as playersPerMatch,
    t.isClosed as isClosed,
    g.name as gameName
```

The query is performed by `public List<Record> getInCommonTournaments(String userA, String userB)`.

5.5.2 Finding the number of followers of a user

This query is in charge of finding the number of followers of a user. It's called every time a user selects another user in the site.

```
MATCH (:User)-[:FOLLOWS]->(u:User)
WHERE u.name=username
RETURN count(*) as numFollowers
```

5.5.3 Follow a game

This query is in charge of following a game. It's called every time a user wants to follow the game he selected.

```
MATCH (u:User) WHERE u.name = username
MATCH (g:Game) WHERE g.name = gamename
MERGE (u)-[:FOLLOWS]->(g)
```

5.5.4 Suggestions

We paid particular attention to the suggested users in the landing page. The objective is to give to every user a tailored suggestion based on their interests/activities. We have 3 levels of suggestions: if the first one does not return at least 4 users, then we go to the second one and so on.

User suggestion based on the tournaments played - We imagined that during a tournament players get in touch with each other and would like to see again people that participated to the tournament. So the idea is to suggest the people that took part at the same tournament, and between them to return the ones that have the highest number of in common followed games.

```
MATCH(u:User{name: username})-[:PARTICIPATE]->(t:Tournament)<-[:PARTICIPATE]-(inComPart:User)
WHERE NOT (inComPart)<-[:FOLLOWS]-(u)
MATCH (inCommonGamers:User)-[:FOLLOWS]->(g:Game)<-[:FOLLOWS]-(u)
WHERE inCommonGamers.name IN inComPart.name
RETURN DISTINCT(inCommonGamers.name) AS suggestedUser,
    inCommonGamers.imgUrl as imgUrl,
    COUNT(*) AS numGames
ORDER BY numGames DESC LIMIT 4
```

User suggestion based on the people that we're following - If we haven't took part to any tournament, we'd like at least to suggest to the user people that are followed by the users that he follows. So we take the first 4 that appear the highest number of times. The main idea behind that is that if I follow some users, and all of them follow another one, maybe I could be interested in following him too.

```
MATCH(u:User {name: username})-[:FOLLOWS]->
    (ub:User)-[:FOLLOWS]->(ut:User)
WHERE NOT (u)-[:FOLLOWS]->(ut)
RETURN ut.name as username,
    ut.imgUrl AS imgUrl,
    COUNT(ut.name) AS numSuggestions
    ORDER BY numSuggestions DESC LIMIT 4
```

Random user suggestion - If we haven't took part to any tournament and if we don't follow any user, we're going to suggest 4 random users, mainly due to responsiveness. We have implemented methods that return the 4 most famous users, where "famousness" is the number of followers of that specific user, but we decided to not use it due to the delay it introduced in the landing page, and because it's not recommended to use query that traverse all the graph in Neo4j.

```
MATCH(u:User)
RETURN u.name AS user, u.imgUrl as imgUrl, rand() as r
ORDER BY r
LIMIT 4
```

```
MATCH ()-[r:FOLLOWS]->(u:User)
RETURN DISTINCT(u.name) AS user,
    u.imgUrl AS imgUrl,
    COUNT(DISTINCT((r))) AS cardinality
ORDER BY cardinality DESC LIMIT 4
```

We also try to suggest games to our users. To do this we have 2 different levels of suggestions, same as before.

Games suggestions based on my favourite categories and users followed - The idea is to find the categories that the user likes the most by counting them. We select the first 4 ones, and then we take a look at games followed by the users that we follow. We consider only the ones whose first category is in the list created before, and we suggest to the user the most frequent ones.

```
MATCH (u:User {name: username})-[:FOLLOWS]->(g:Game)
UNWIND g.category AS categories
WITH COLLECT(categories) AS bestCategoriesList, COUNT(*) AS numGames
ORDER BY numGames DESC LIMIT 4
MATCH (u:User {name: username})-[:FOLLOWS]->(following:User)
MATCH (following)-[:FOLLOWS]->(g:Game)
WHERE NOT (u)-[:FOLLOWS]->(g) AND g.category[0] IN bestCategoriesList
RETURN g.name AS gameName,
    g.imgUrl AS imgUrl,
    COUNT(g) AS numFollowers
ORDER BY numFollowers DESC LIMIT 4
```

Random Games suggestions - It could happen that the user is not following enough users or games so that the first suggestion has 4 results. If this is the case, then we're going to suggest random games in the network. The reason is explained above, and it is the same.

```
MATCH(g:Game)
RETURN g.name AS game, g.imgUrl as imgUrl, rand() as r
ORDER BY r
LIMIT 4
```

```
MATCH ()-[r:FOLLOWS]->(g:Game)
RETURN DISTINCT(g.name) AS game, g.imgUrl AS imgUrl,
COUNT(DISTINCT((r))) AS cardinality
ORDER BY cardinality DESC LIMIT 4
```

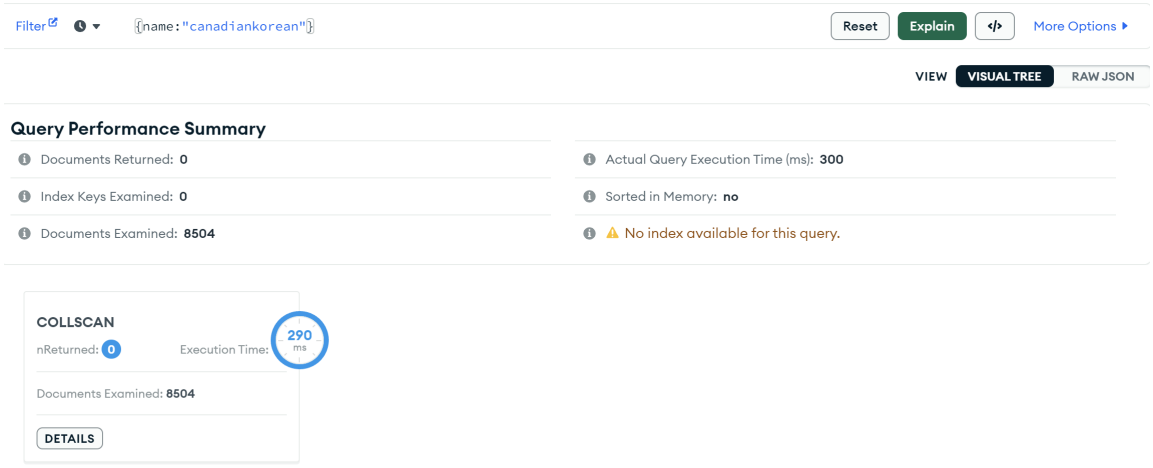

Figure 20: Before inserting the index on username in the User collection

6 Databases Choices

In this section we're going to make an analysis on the database architecture, and in particular on indexes, distributed database design, sharding, CAP theorem and replicas.

6.1 Indexes

As we've seen in 3.3, some of the most used operations in the application are searching for a game using the game name, searching for a user using the username, displaying posts of a certain user or of a certain game. We can assume to deal with a read-heavy application. In this case, the use of several indexes allows the user to quickly access to the database and to retrieve documents fast. The results shown below are taken in local, but we assume them to be the same if collected in remote.

6.1.1 MongoDB Indexes

In order to have fast readings for the user page, which is searched by username, we decided to insert a regular index in the user collection using the username itself.

MongoDB Compass shows that, before inserting the index we had to search between 8503 documents, spending 300 ms. After inserting the index we obtain much better results in terms of documents inspected: 1 only document, and 5 ms spent in searching.

We did the same for the collection game, where we have a regular index on the game name.

On the Post collection we decided to add 1 index, on game name, in order to have fast readings in accessing a game page. Before inserting the index for the game name, searching all posts about the game "Libertalia" took 37 ms and 15039 documents inspected, while after inserting the index it took 0 ms.

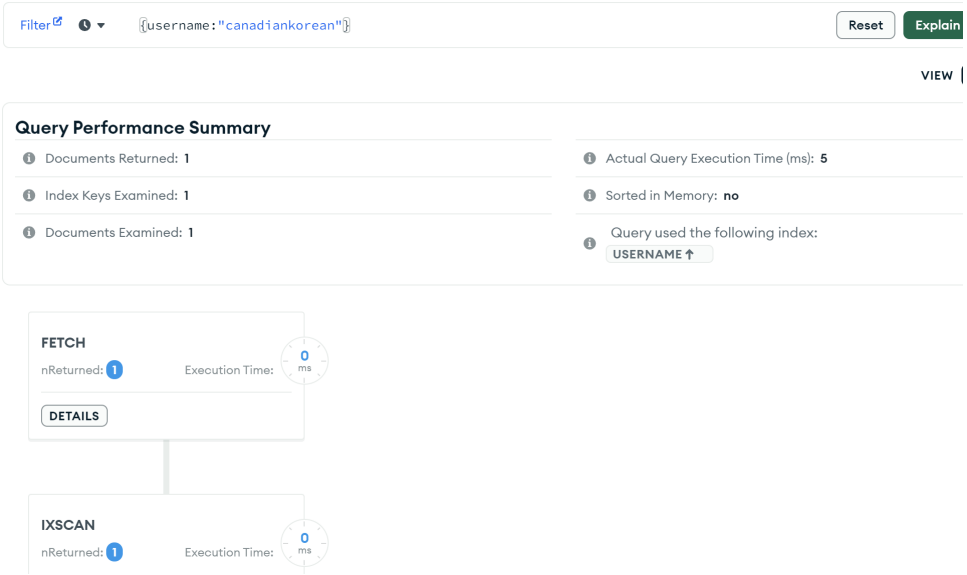

Figure 21: After inserting the index on username in the User collection

6.1.2 Neo4j Indexes

Since every time we search for a user or for a game we have to display in common relationships, we decided to add indexes on Neo4j too.

The first index is on the username for the User entity. From the image we can see that, without this index, searching a node by username would spend 24 ms, while after inserting the index we need only 4 ms. We decided to do the same for the Game entity, inserting an index on the gamename.

6.2 Distributed Database Design

6.2.1 Sharding

In order to improve the availability of the system, we propose to adopt sharding concurrently to the data replication. We propose sharding for only MongoDB collections, because community Edition of Neo4j didn't allow it.

- Regarding *User Collection* we propose **Username** as sharding key because e are used to retrieve the user document by username, that is always present. This sharding proposal doesn't affect the actual collection structure.
- Regarding *Game Collection* we propose **Name** as sharding key for the same reasons of above.
- Regarding *Post Collection* we propose **Game** as sharding key because posts are naturally balanced between games, and so sharding on this field make a natural load balancing on server cluster.

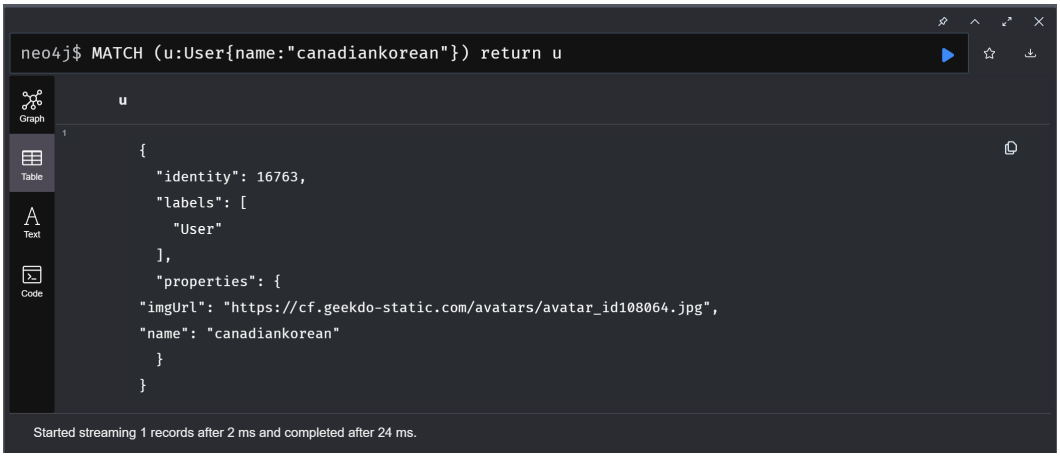

Figure 22: Before inserting index on username on Neo4j

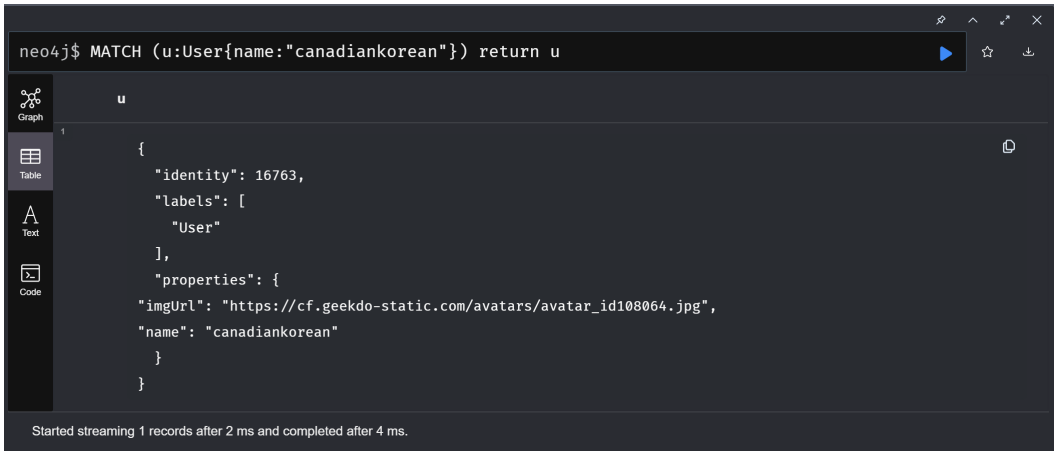

Figure 23: After inserting index on username on Neo4j

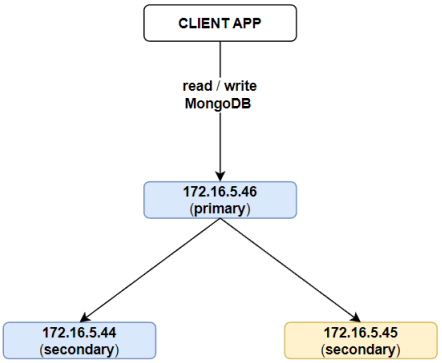

Figure 24: Replica set configuration used for `MongoDB`. Server 172.16.5.45 contains also an instance of `Neo4j` database.

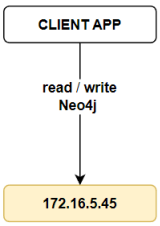

Figure 25: Configuration used for Neo4j.

6.3 Replicas

One of the *non-functional requirements* that we have in mind is to ensure **high-availability**. To reach this goal we have designed the following replica set configuration:

- Regarding MongoDB, the replica set is composed of a primary replica, the latter takes client requests, and two secondaries which are the servers that keep copies of the primary's data.
- Regarding Neo4j, we have only one Neo4j instance[1].

6.3.1 `MongoDB` Replica configuration

The configuration is shown below:

```
rsconf = {
    _id: "lsmdb",
    members: [
```

[1]To the best of our knowledge, student's version of Neo4j does not permit replica set configuration

```
        {_id: 0, host: "172.16.5.45:27017", priority:1},
        {_id: 1, host: "172.16.5.44:27017", priority:2},
        {_id: 2, host: "172.16.5.46:27017", priority:5}]
};
```

We put the lowest level of priority on 172.16.5.45 machine because it is the one containing the only instance of `Neo4j`.

We decided to introduce both **Write Concern** and **Read Preferences** constraints and set this configuration:

```
spring.data.mongodb.uri=mongodb://172.16.5.46:27017,172.16.5.44:27017,172.16.5.45:27017/
replicaSet=lsmdb&w=1&readPreference=nearest&retryWrites=true
```

- `w = 1` - The application regains control once the first copy is written, without waiting that the primary server propagate the write to the secondary ones. This situation could let to lost some data if the primary nodes crashes before propagate the writing and after acknowledge the write request to the application. We accept it, because we want to advantage *Availability* over *Consistency*. About the write operation, we decide to set `retryWrties = true` in the connection options for `MongoDB`. In this way, we allow `MongoDB` drivers to automatically retry write operations a single time if they encounter network errors, or if they cannot find a healthy primary in the replica sets.
- `readPreference = nearest` - We allow the `MongoDB` driver to read from any of the replica because of the non-functional requirements low latency and high availability. We accept the fact that data obtained from the `MongoDB` server could be not updated to the latest version.

6.4 CAP Theorem Issue and Consistency chosen

Because of our non-functional requirements, we decided to stay on the AP side of the triangle. In particular, we decide to don't have a strict consistency, and infact as you can see we have chosen a w=1 as write concern in replicas. We so accept to see not updated data, adopting the eventual consistency, and we even take the risk to don't see some data at all, due to the risk the primary could fall before updating secondaries.

6.5 Databases consistency management

Operations of creating a Game or User and deleting one of them are operations that involve both databases. To ensure consistency between them, you can see the flow of operations in figures (26) and (27)

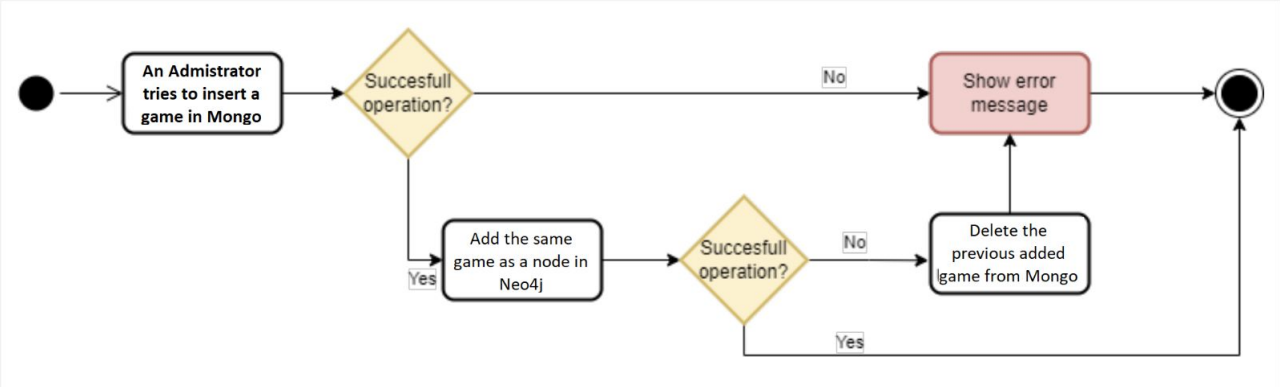

Figure 26: Flow in case of Create() functions.

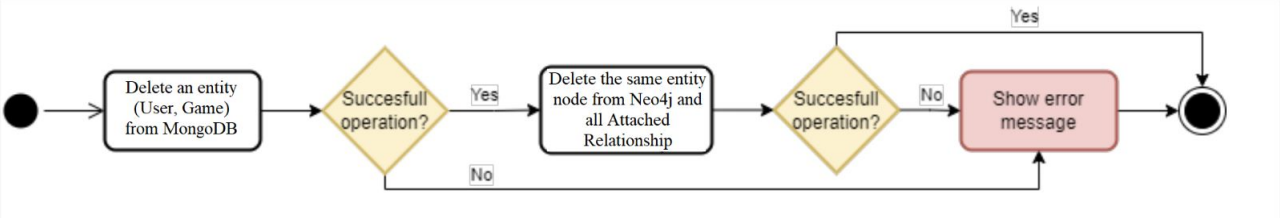

Figure 27: Flow in case of Delete() functions.

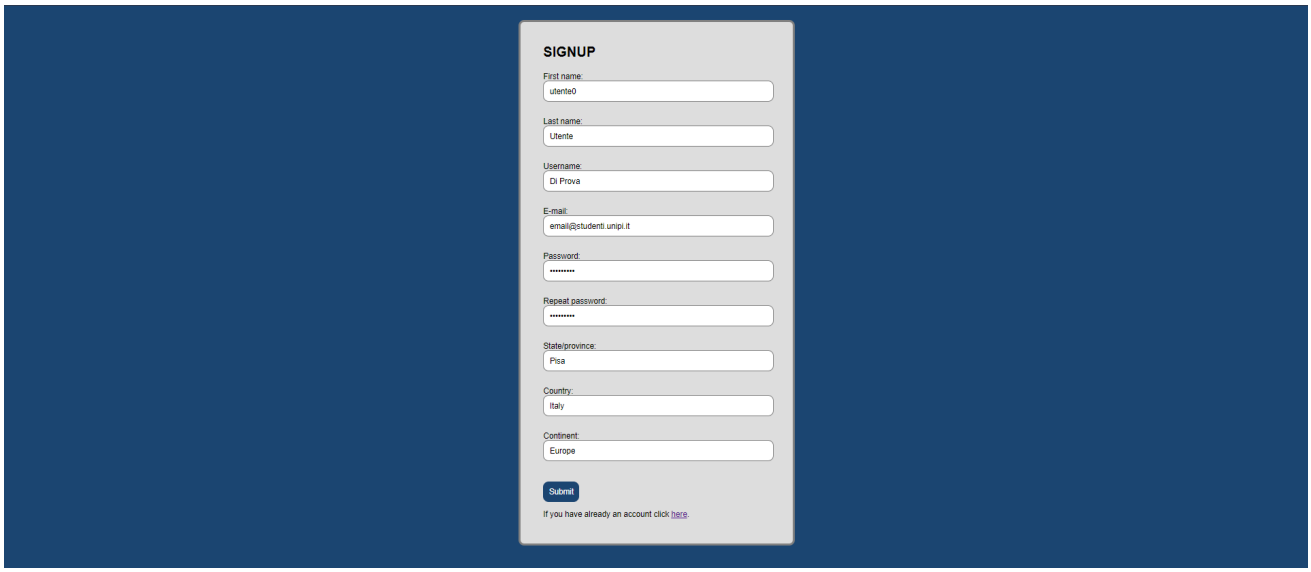

Figure 28: Signup page

7 Usage Manual

7.1 User Manual

- If a visitor wants to fully experience our application, the first thing he will try to to is to signup. Here, Him/Her will be asked to insert some personal details (Figure 28), including First and Last Name, Country, etc.
- Whatever signup has been just done, or if it has been already done, you will be asked to login (Figure 29), inserting username and password provided during signup process.
- When logged in, homepage will be showed. From here (Figure 30), a number of information are showed and a number of possibilities are allowed.
  - A navbar will always be on top of the page. From there you can search for games, users, and you can click on the *home icon* to return to your landing page, or on the *user icon* to go to your personal page (Figure **??**).
  - Four games will appear to you in big circles, covering the major part of the page. These are the games suggested to you. Clicking on one of them will bring you to the game's page.
  - Four users will appear on the right column. These are the users suggested to you, and you share common interests with them. Clicking on one of them will bring you to the user's page.
- If you search for a game from your navbar, you can use the drop-down selection to filter game by categories. Whatever you choice to filter results, you can decide to enter a sub-pattern of the game's name, or the exact name of an existent one.

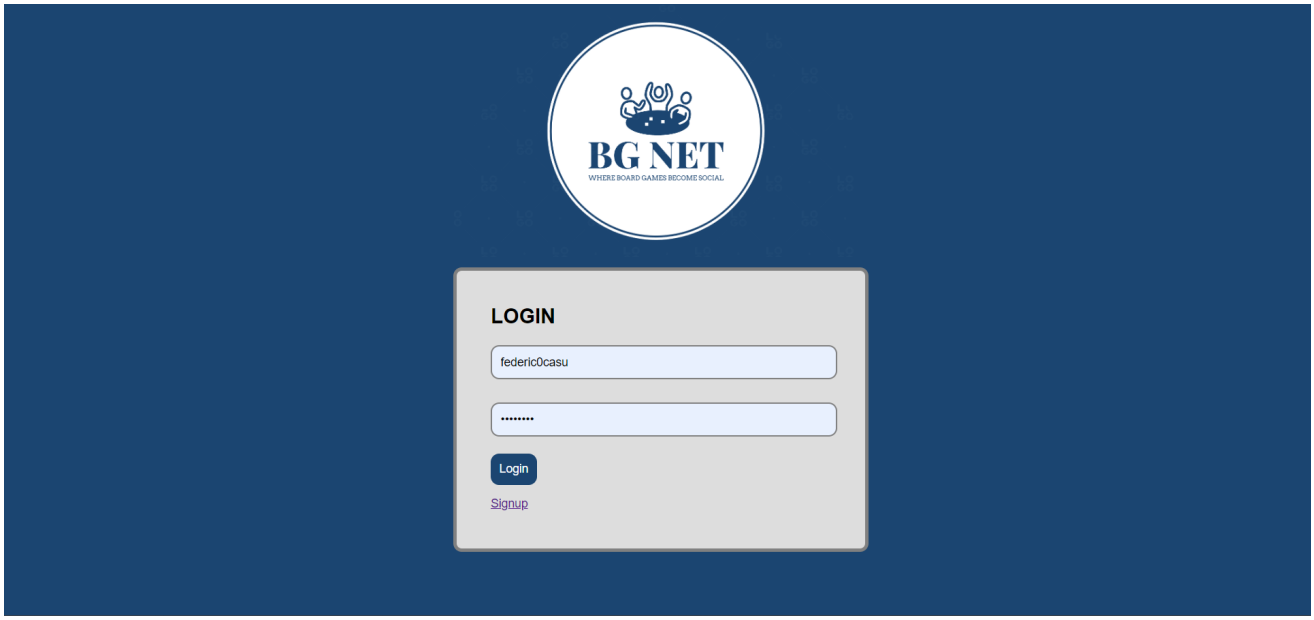

Figure 29: Login page

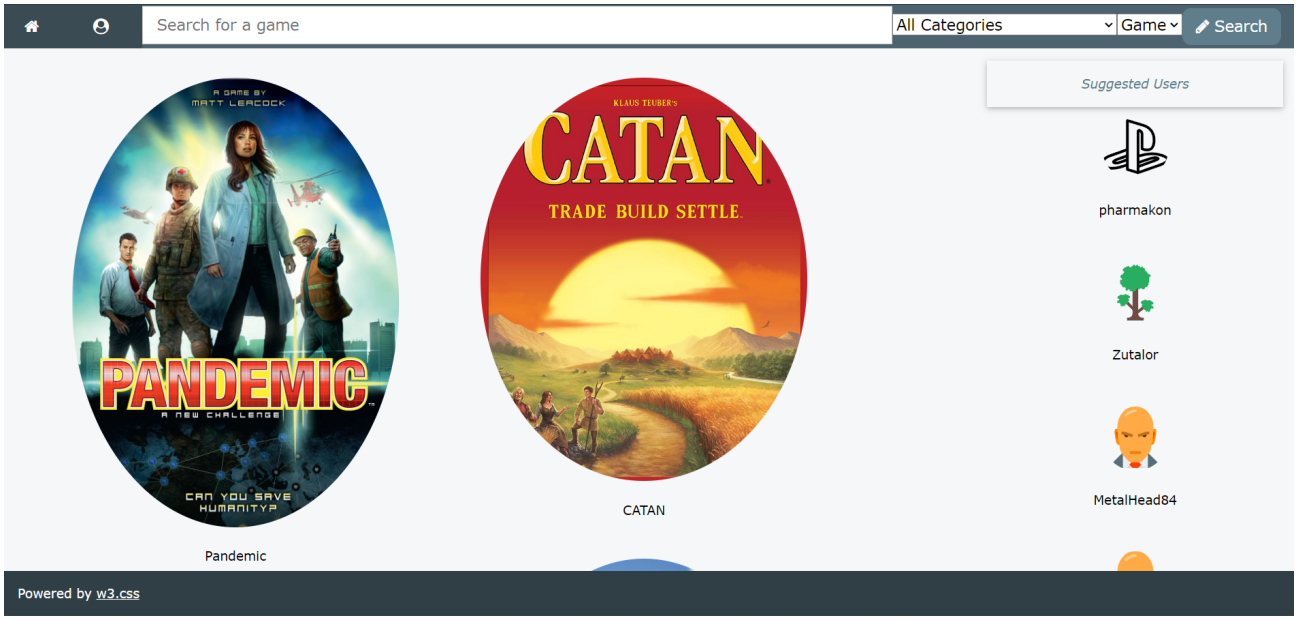

Figure 30: Homepage

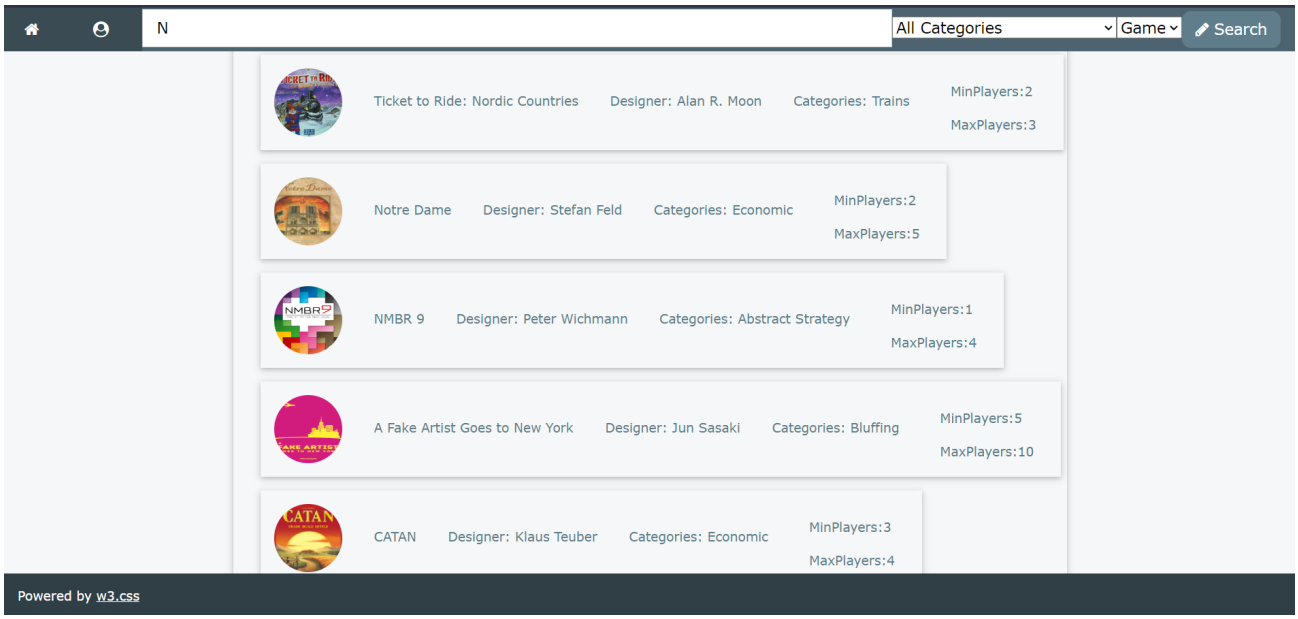

Figure 31: Results page

  - If you choose a subset-search, you will land in a results page (Figure 31), from where you can find all games that match your filters. From here, you can see a brief description of games, and you can click on one of them to arrive at its page.
  - If you choose the exact-search, then you will directly land in the game's page.

- Whenever you arrive at a game's page (Figure 32), you will see a description of the game and a number of information, including its average rating and its number of followers. You even see a list of the most recent posts regarding that game. If you follow the game, you will even see a list of users that follow both you and the game. From here:
  - You can decide to follow the game (or unfollow it) by pressing the *follow (unfollow) button*.
  - You can decide to rate the game (if you haven't done it yet) by inserting a number between 1 and 10 in the rate field and pressing the *rate button*.
  - You can like posts or dislike them by pressing the *like button*
  - You can decide to view the full list of posts of that game by pressing the *view all posts button*. If so, you will land in a page that allows you to browse page with a pagination (Figure 34) system.
  - You can select a post and view his comments by pressing the *view comments button*. This will bring you in a page from where you can browse all comments (Figure 35) regarding that post.
  - You can decide to create a post by pressing the *Post Something button*. This will make a textarea appear (Figure 36). By pressing the *Submit button*, your post will be persistently written (Figure 37) on the game page.

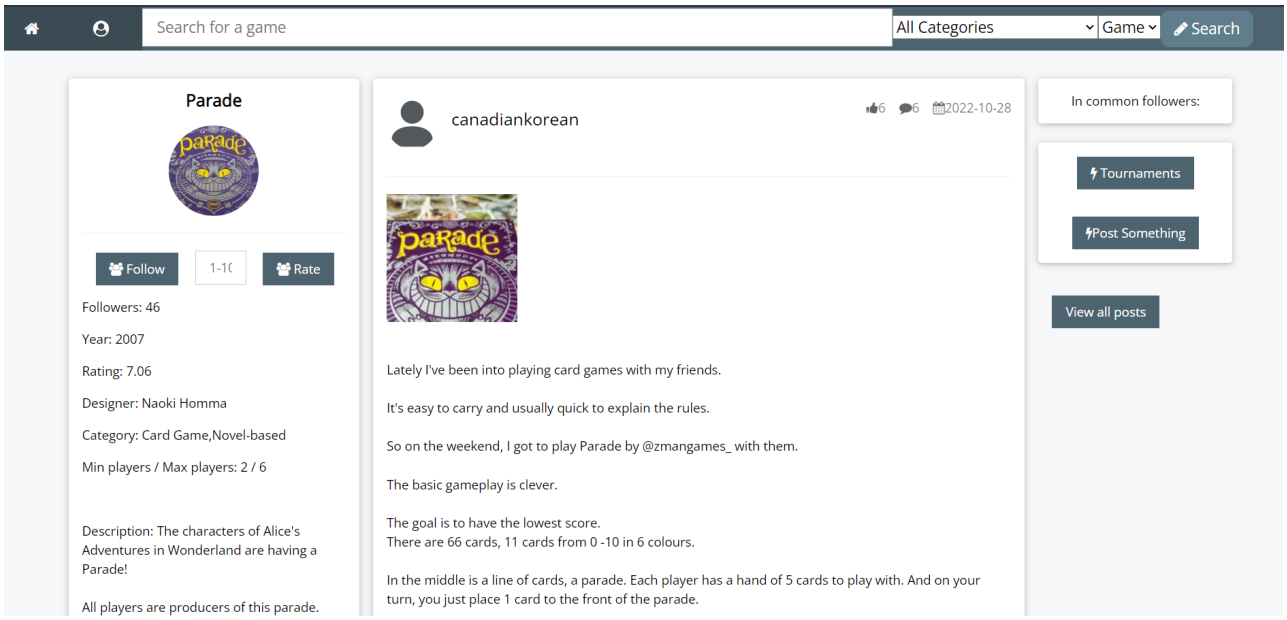

Figure 32: Page of a Game.

  - You can decide to open the tournament page by pressing the *Tournaments button*.

- Whenever you arrive at a Tournament page (Figure 38), you will see a list of tournaments (planned and expired) regarding that game. You can decide to participate (or quit) a tournament by pressing relative buttons. You can even decide to create a tournament, by pressing the *Create button* and inserting the required information (Figure 39).

- Whenever you arrive at a User page (Figure 40), you will see a description of the user, including some personal information and the number of his followers. You can even see a list of users that both follow you and the user you are an, and a list of tournaments you have both enrolled in. You can see his most recently written post, like them. Lastly, you can follow him by pressing the *follow button*.

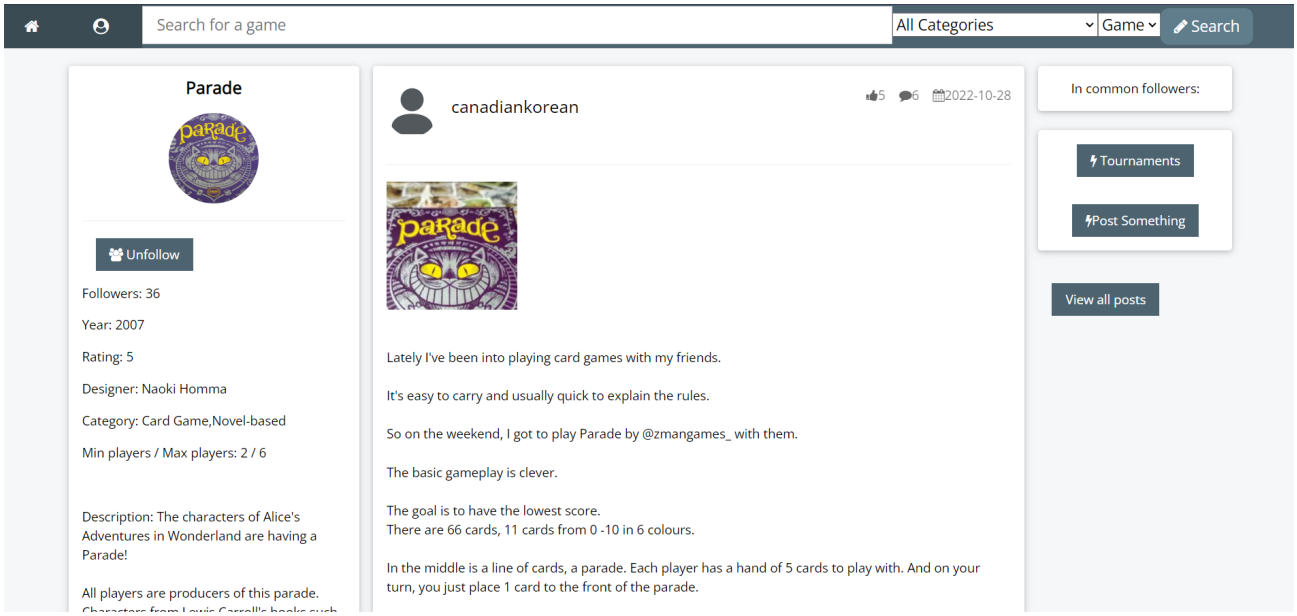

Figure 33: Page of a Game you have rated and followed.

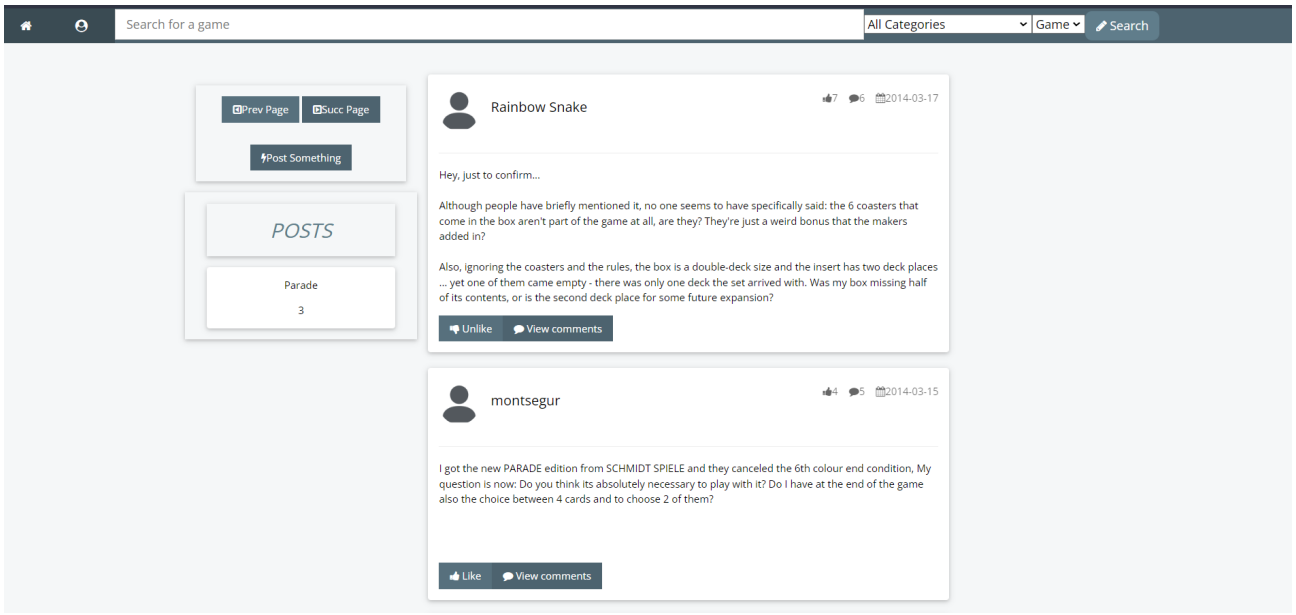

Figure 34: View all posts of a game.

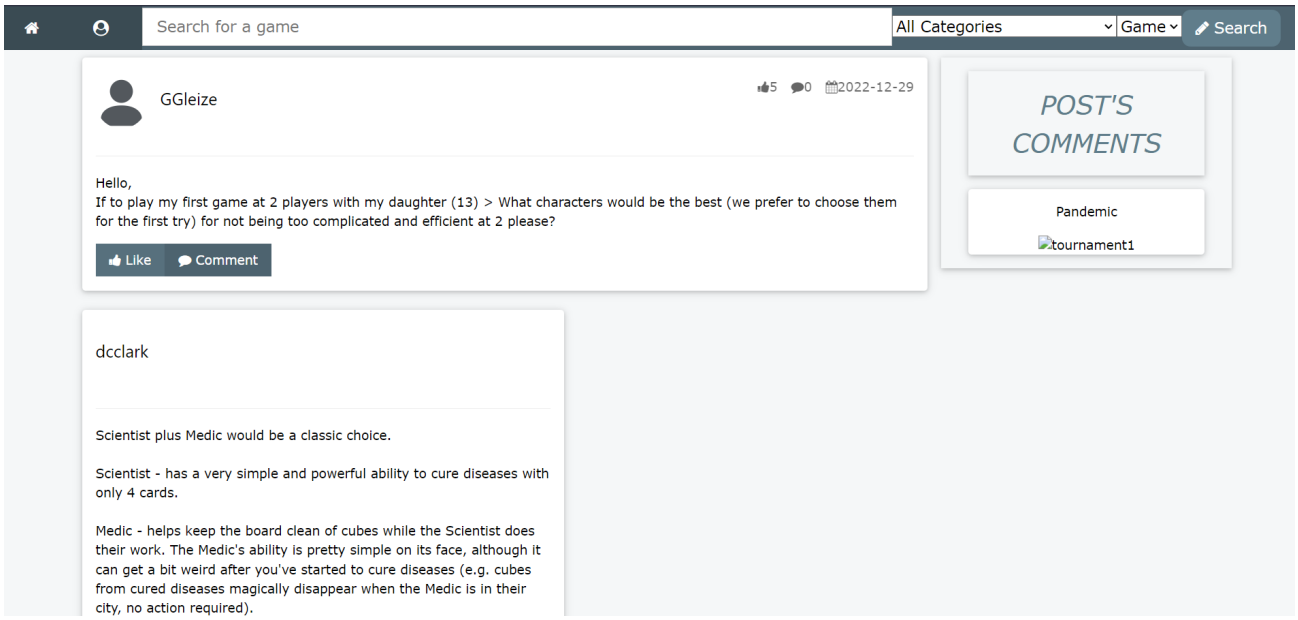

Figure 35: See all comments of a post.

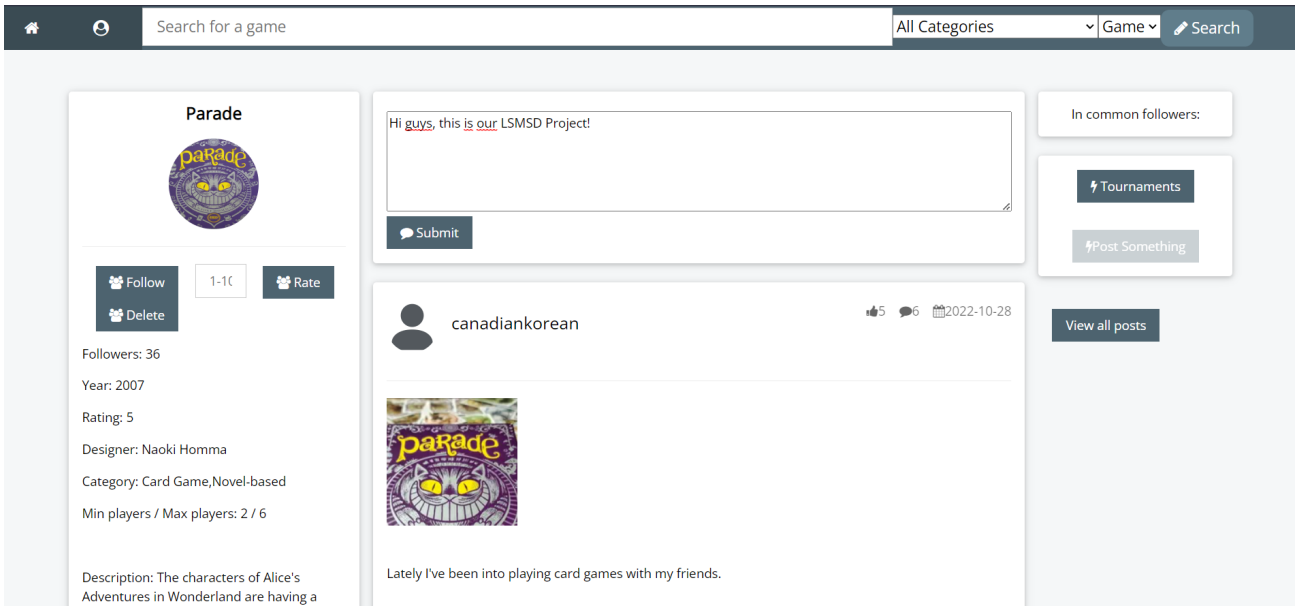

Figure 36: A TextArea appears when you want to post.

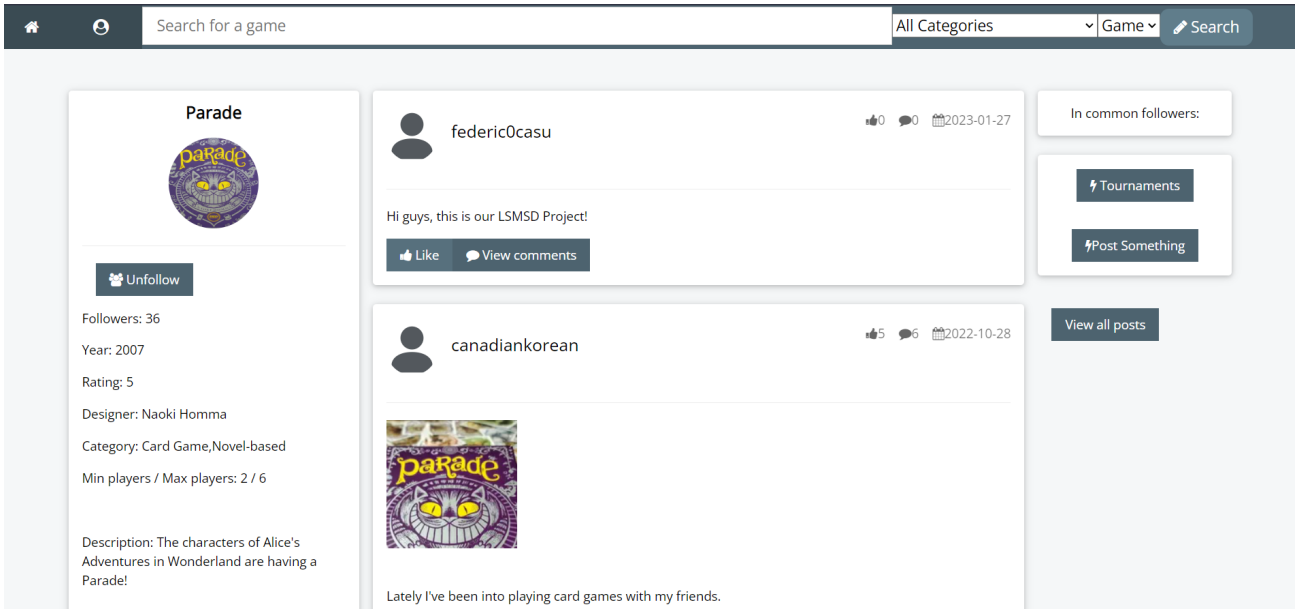

Figure 37: Submitted post.

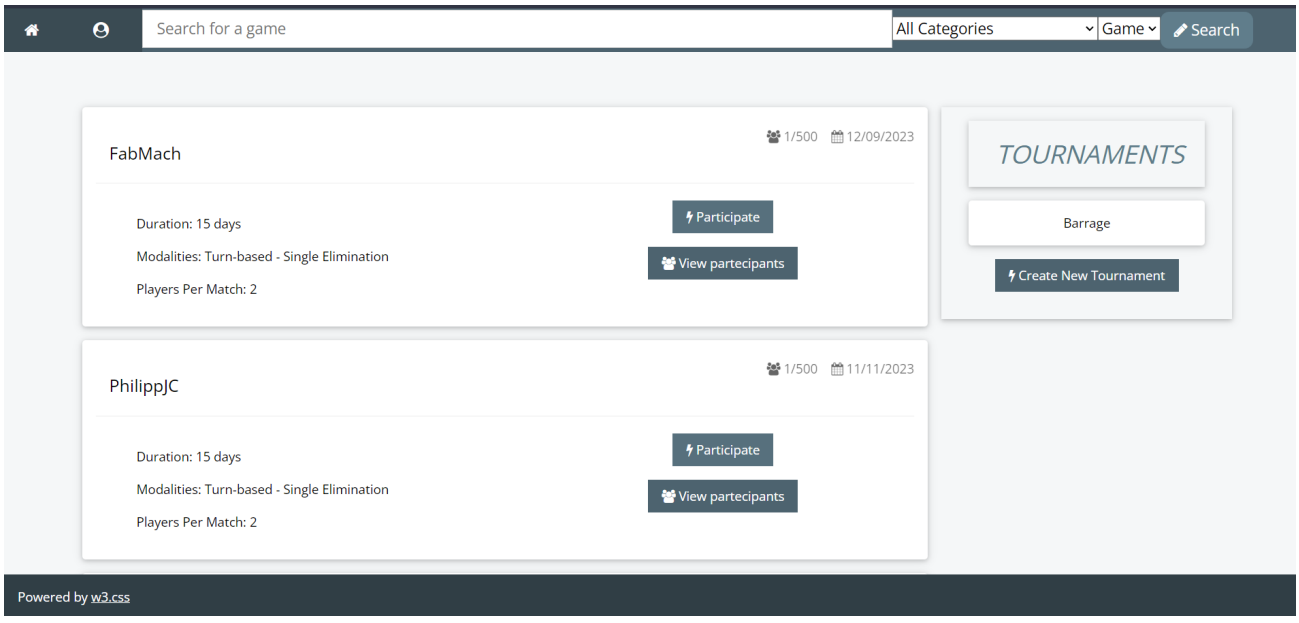

Figure 38: Tournaments Page.

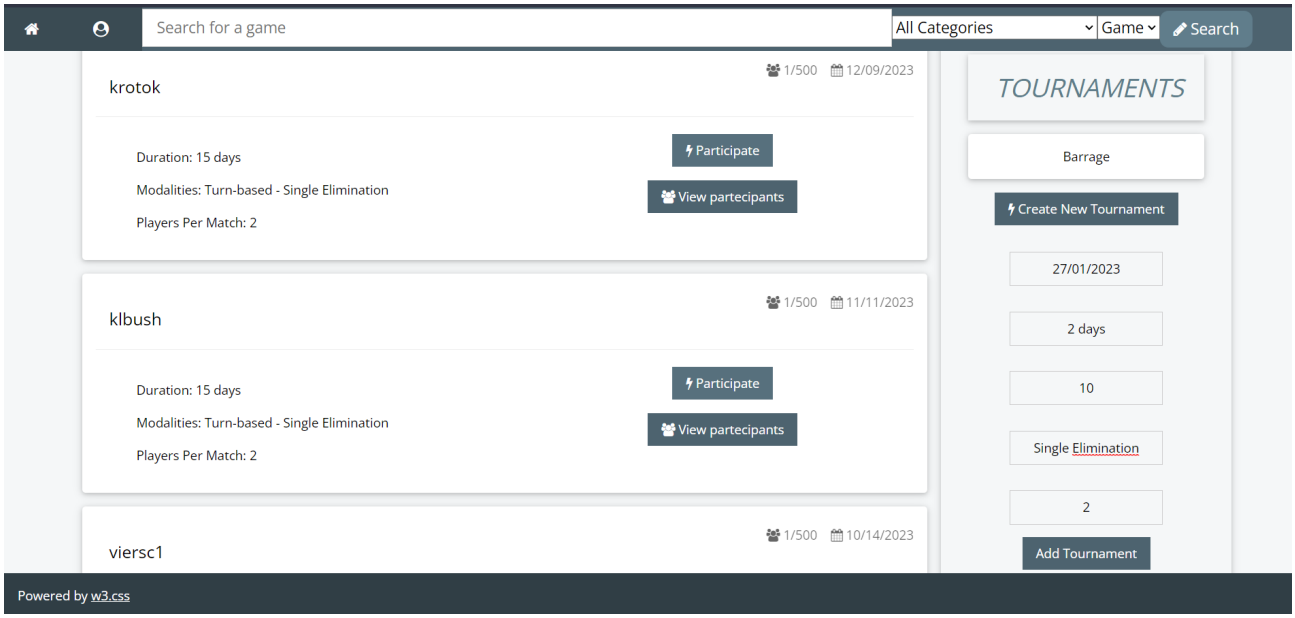

Figure 39: Create a Tournament.

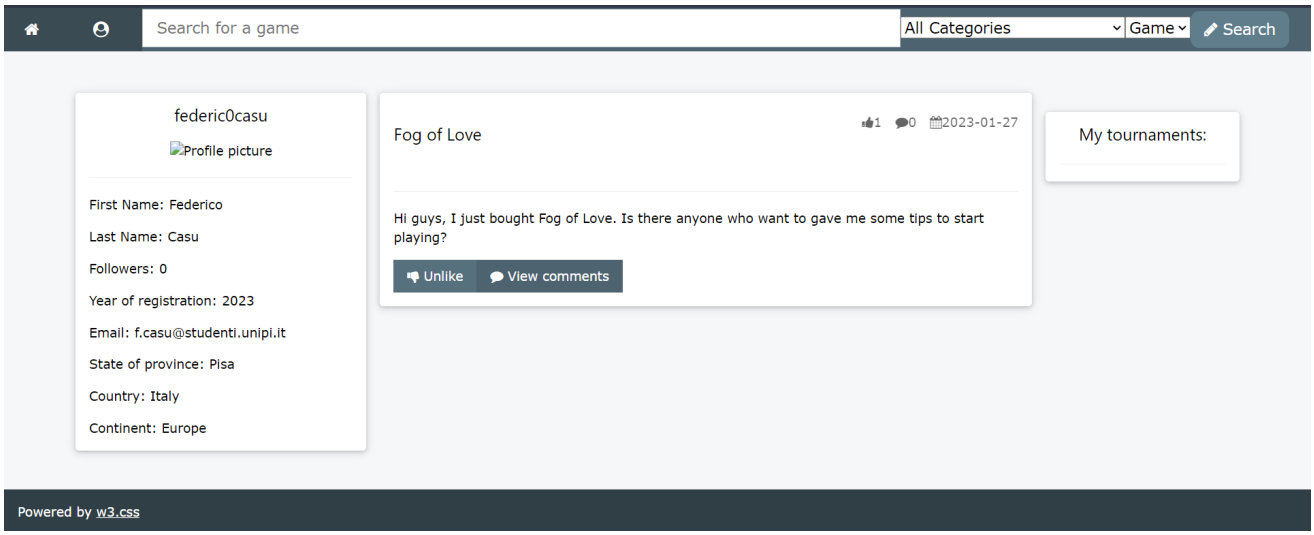

Figure 40: User's page.

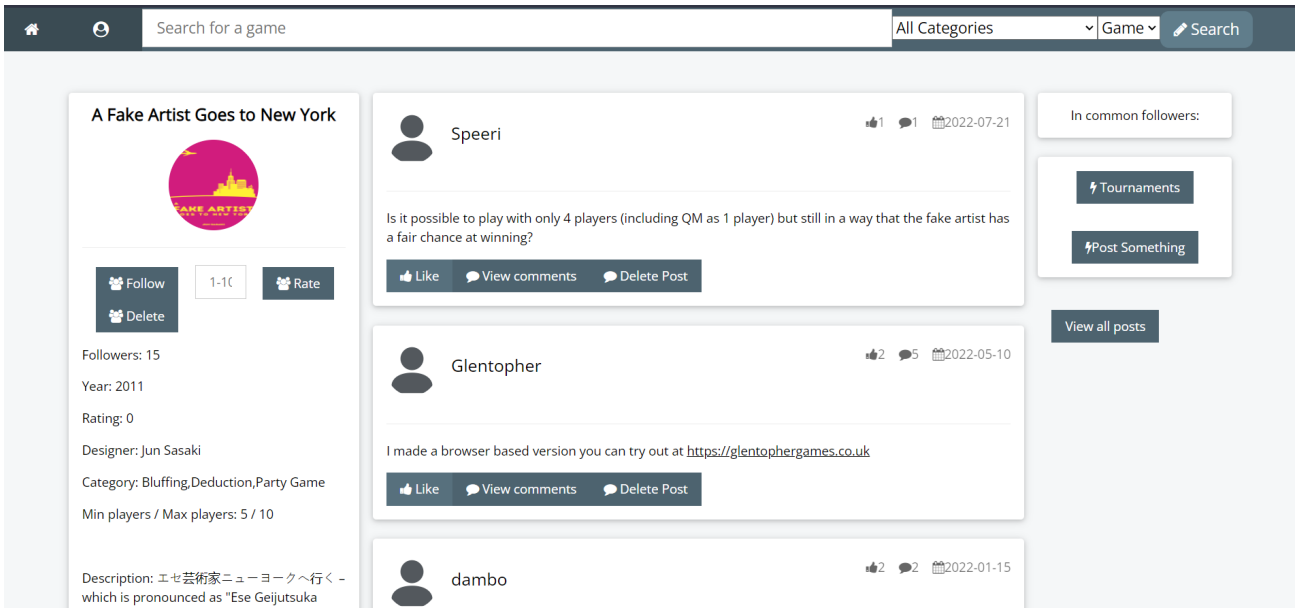

Figure 41: Page of a Game if you are an administrator

7.2 Admin Manual

- As an Administrator, you have a little different view of a game's page (Figure 41) and user's page (Figure 42), because your option to delete a game, a post, or to ban an user are added to the view.
- You can click the button in the shape of a stylized little man (it is located to the upper-left of the home page) to land in a control panel (Figure 43) from where you can access some analytics and see their results. From there you can even add a new game in the system by pressing *Inser New game Button* and inserting the required information (Figure 44).

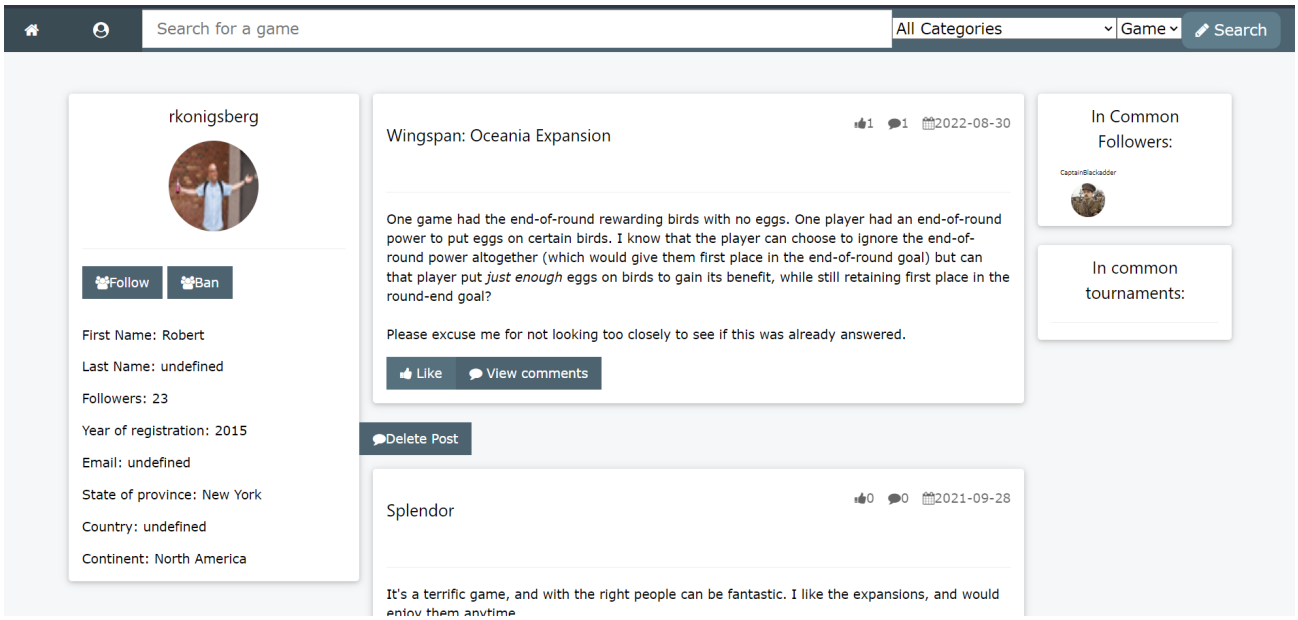

Figure 42: Page of an User if you are an administrator

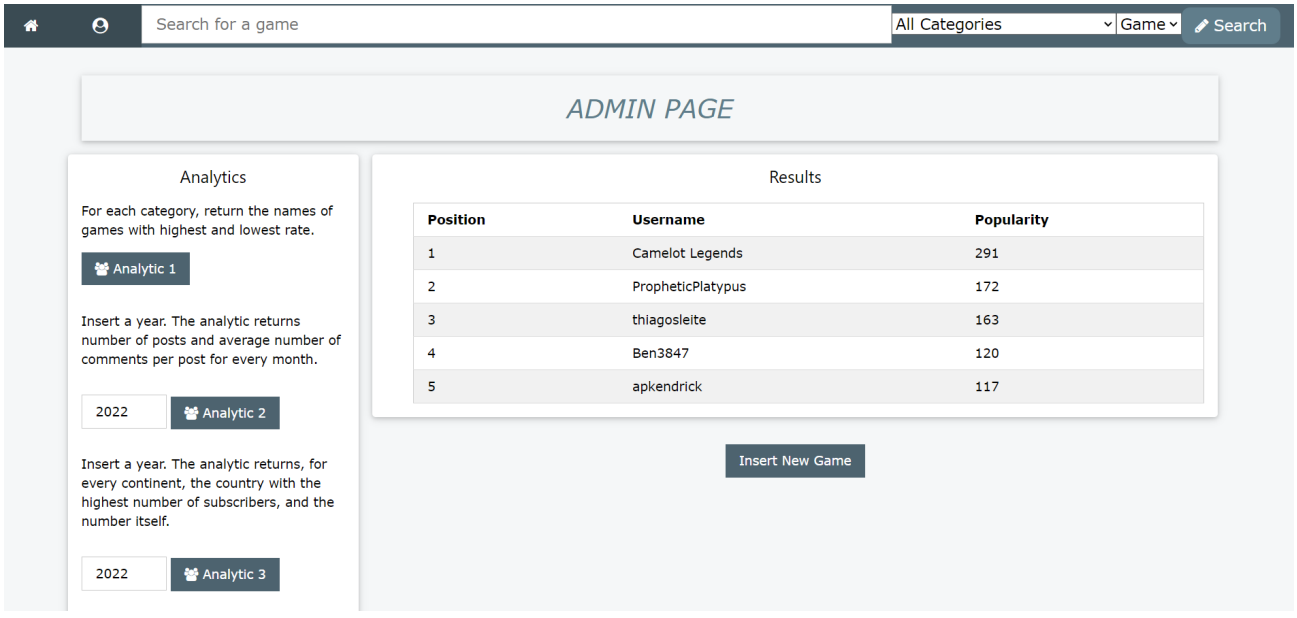

Figure 43: Administrator Control Panel.

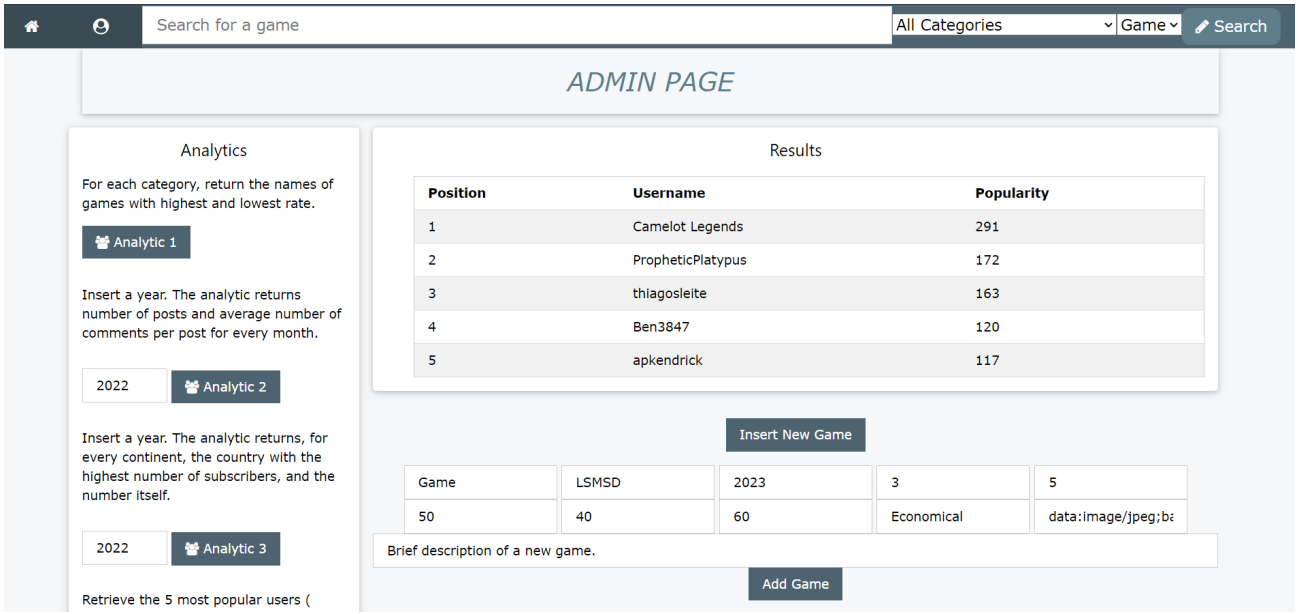

Figure 44: New game creation.

8 Future Implementations

8.1 Possibly Functionalities we could add in future

- Adding the possibility to view the complete list of followers of a game/user
- Adding the possibility to modify your rating to a game
- Adding the possibility to like a comment
- Adding the possibility to reply to a comment
- Adding the possibility to udpate a written post or a written comment