UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Master's Degree in Artificial Intelligence and Data Engineering

BoardVerse

Large Scale and Multistructured Databases Project

Group:
Martina Fabiani
Tommaso Falaschi
Emanuele Respino

Project GitHub Repository:
https://github.com/martiFabia/BoardVerse/tree/Neo4j-Integration

ACADEMIC YEAR 2024/2025

Contents

Chapter 1

Introduction

BoardVerse is an application for true board game fans that allows users to discover, explore and share this passion. Thanks to a wide catalogue of games, users can search for new gaming experiences, read detailed reviews and ratings from other players, or share their opinions to help the community.

You can create a custom list of your favorite games, follow the latest trends, organize tournaments and connect with other fans with similar tastes. The platform also offers spaces for group discussions, where players can exchange strategies, tips and ideas.

This rich catalog is maintained by administrators, who not only add, update or remove games, but also supervise all user activities. For example, they can moderate inappropriate reviews or comments, ensuring that the app remains a welcoming environment and in line with the friendly spirit of the community.

The application uses a combination of technologies such as MongoDB for flexible management of documents and Neo4j to represent relationships between users, games and tournaments in an advanced way.

Chapter 2

Actors and Mockups

2.1 Actors

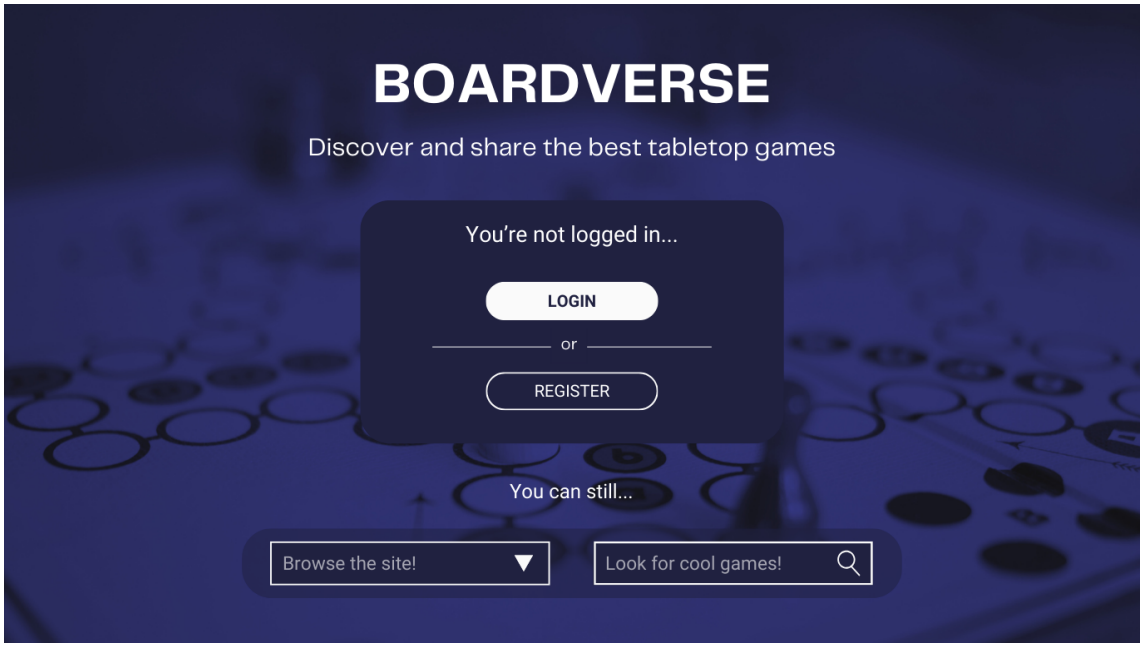
In BoardVerse there are three main actors:

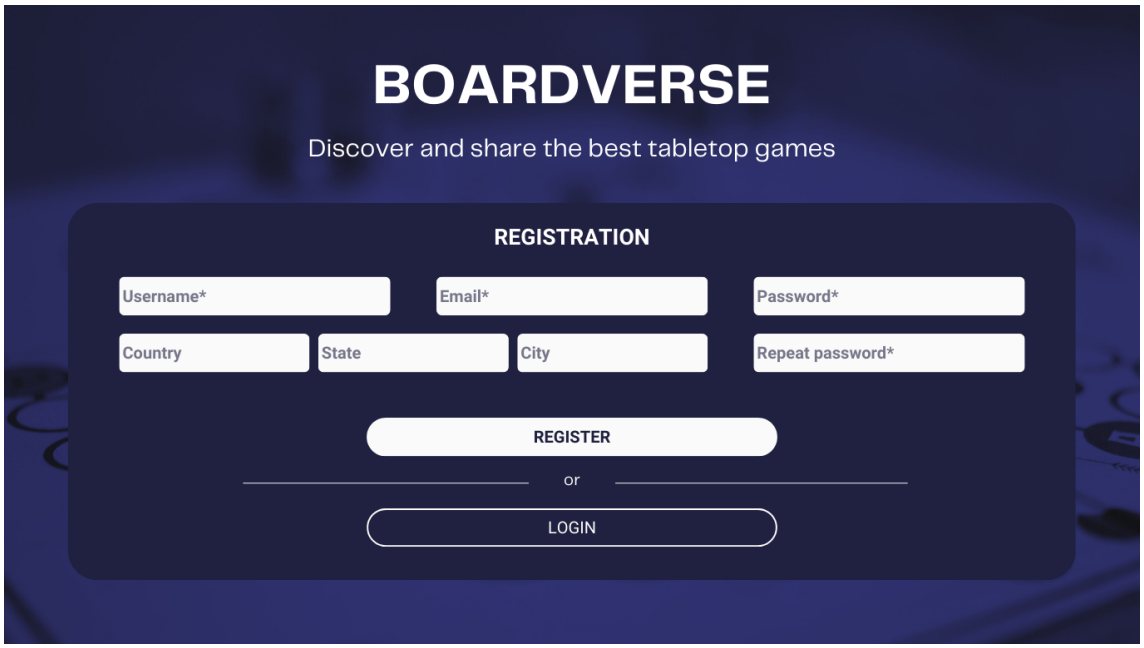
- **Unregistred User**
- **Registred User**
- **Administrator**

The unregistered user has a limited set of actions that he can perform compared to the registered one who has access to all features of the application except those reserved for the administrator who will also manage the catalog of games and moderate the activity of registered users.

2.2 Mockups

2.2.1 Unregistred User

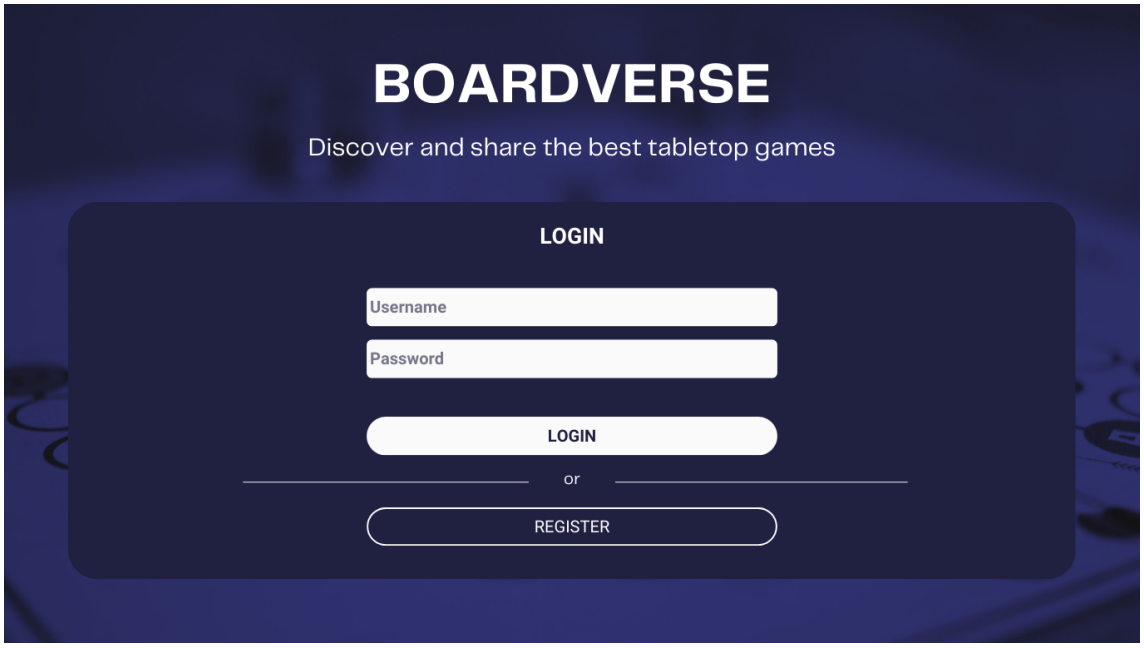


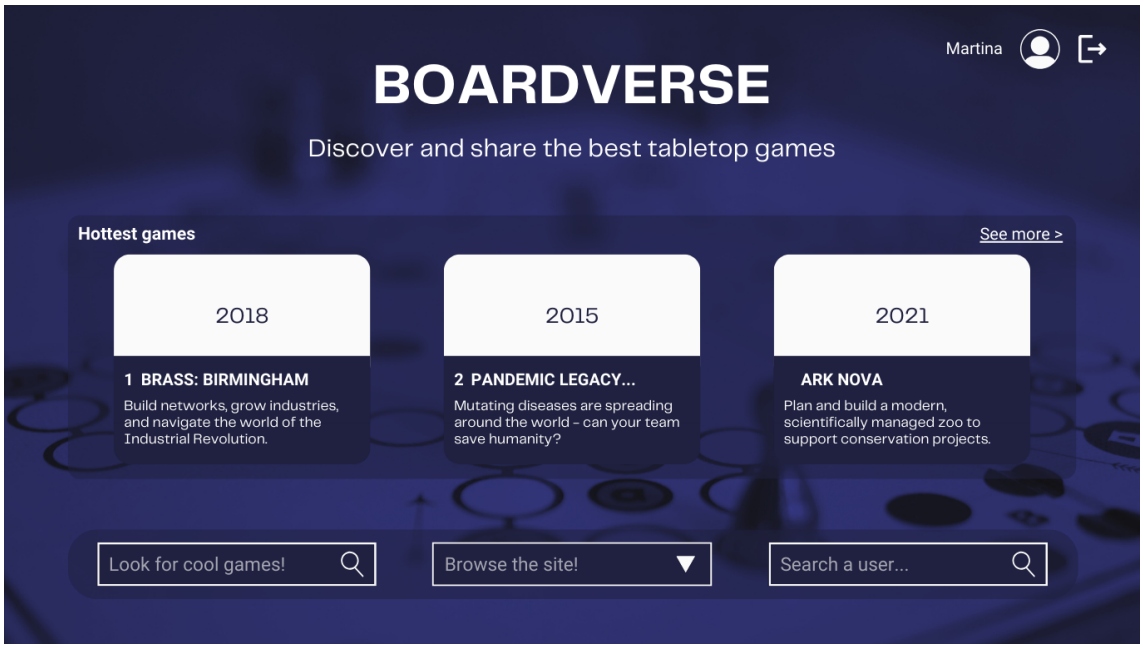


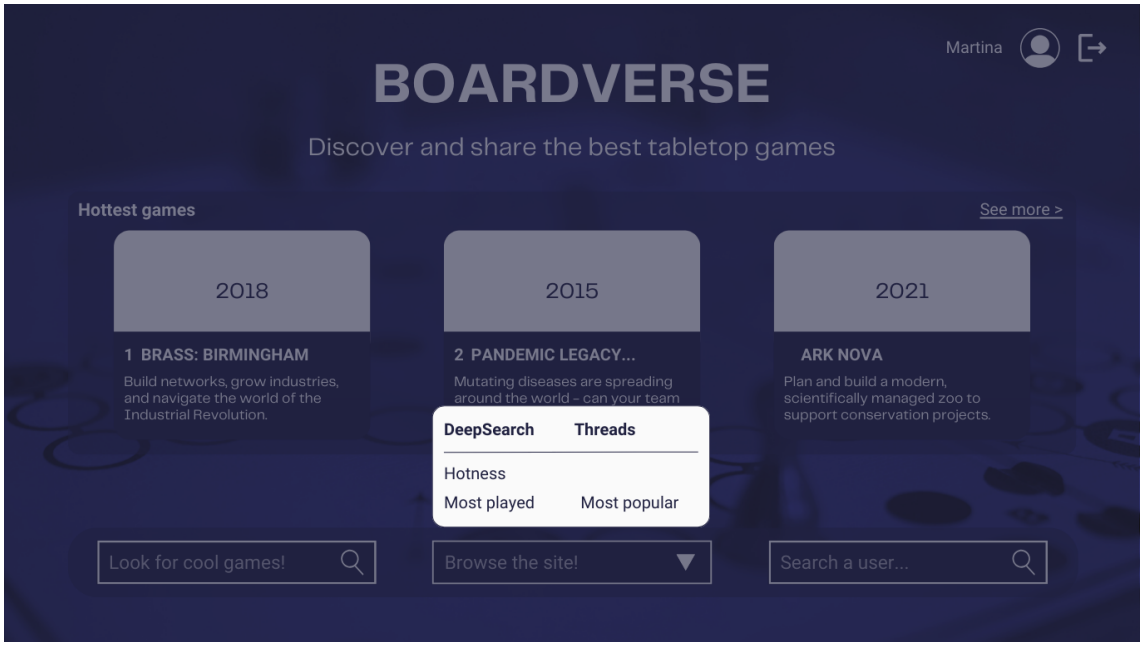
2.2.2 Registred User and Administrator

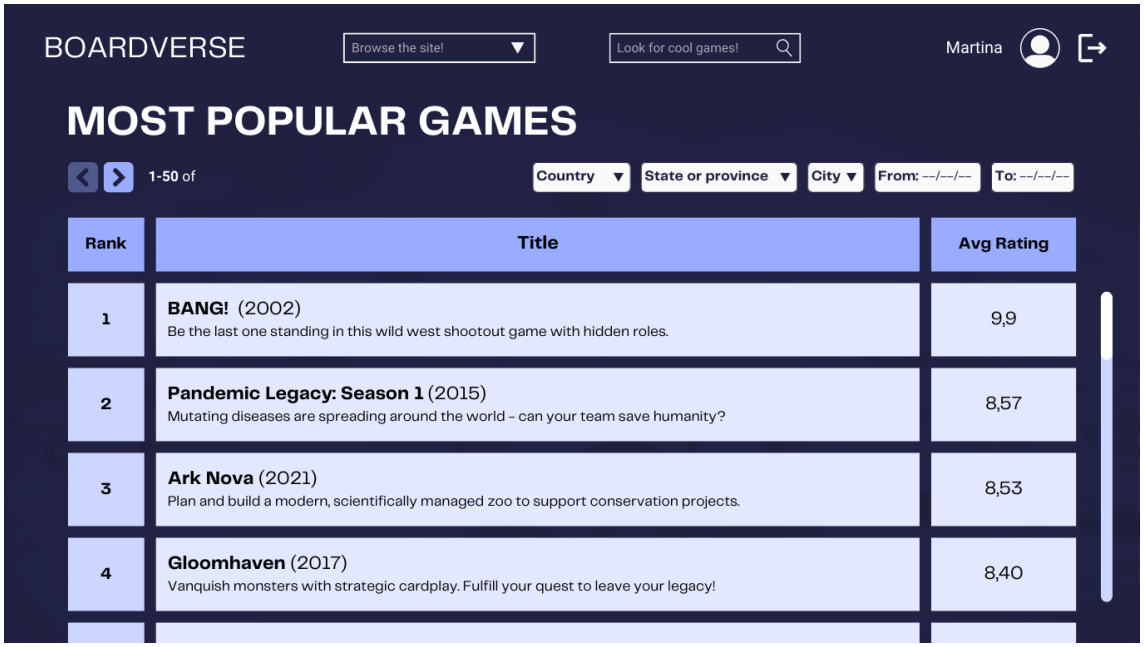
Actions that can be performed by both registred user and admin are highlighted with a red button, while actions that can only be done by

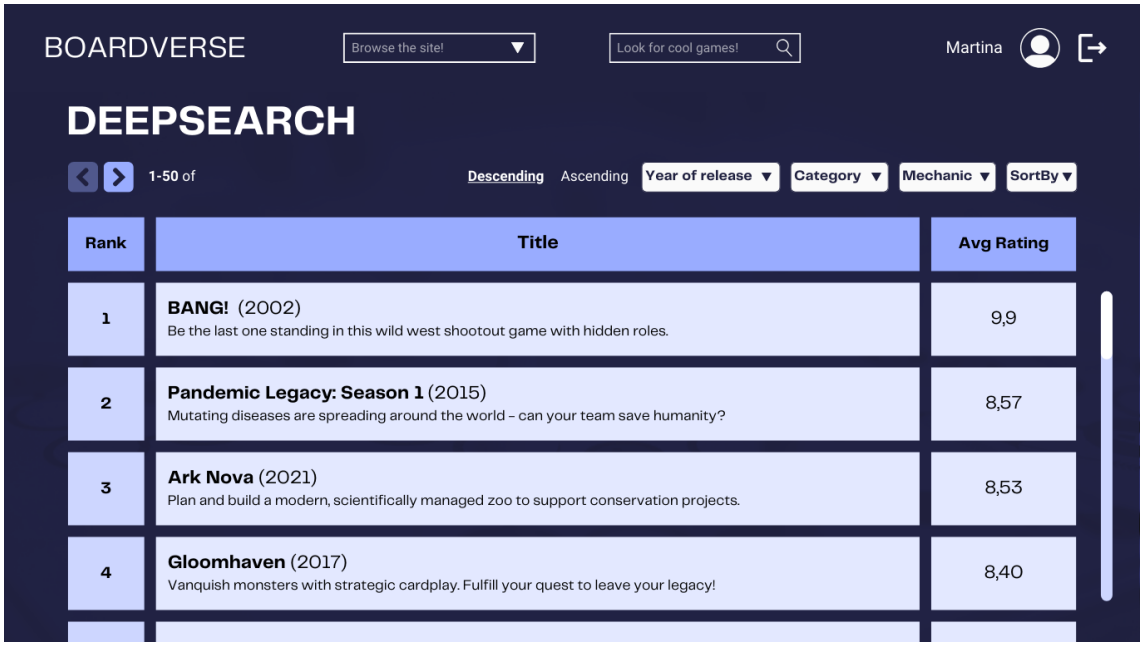
the admin are indicated with a green button. This is the main difference between the actions that can be carried out by the two different actors.

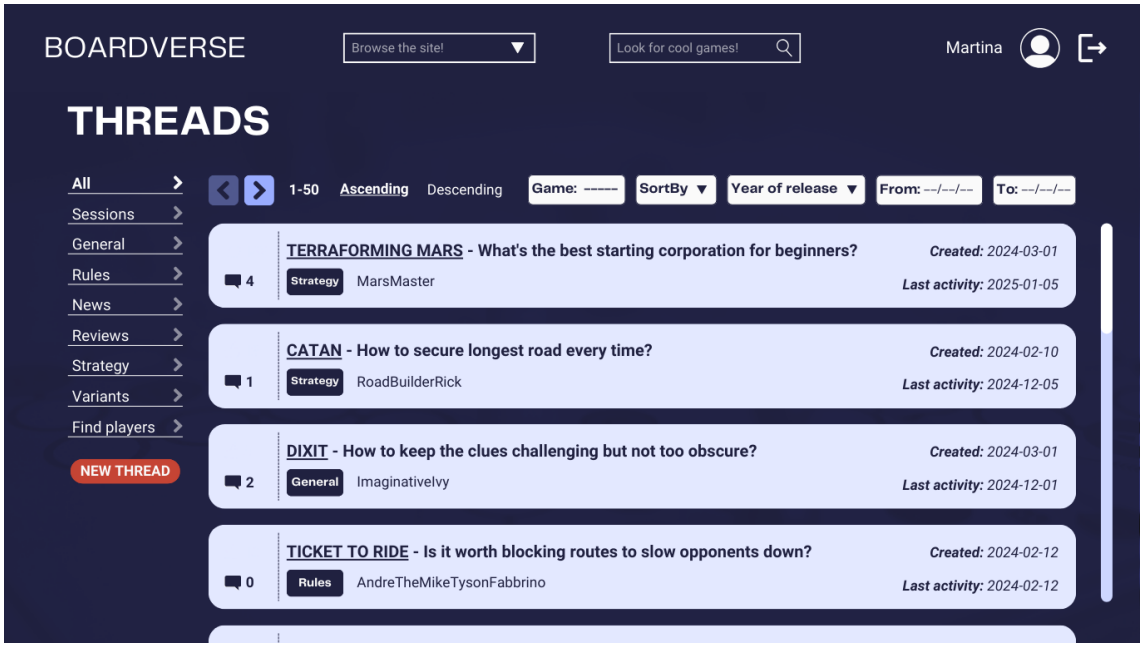


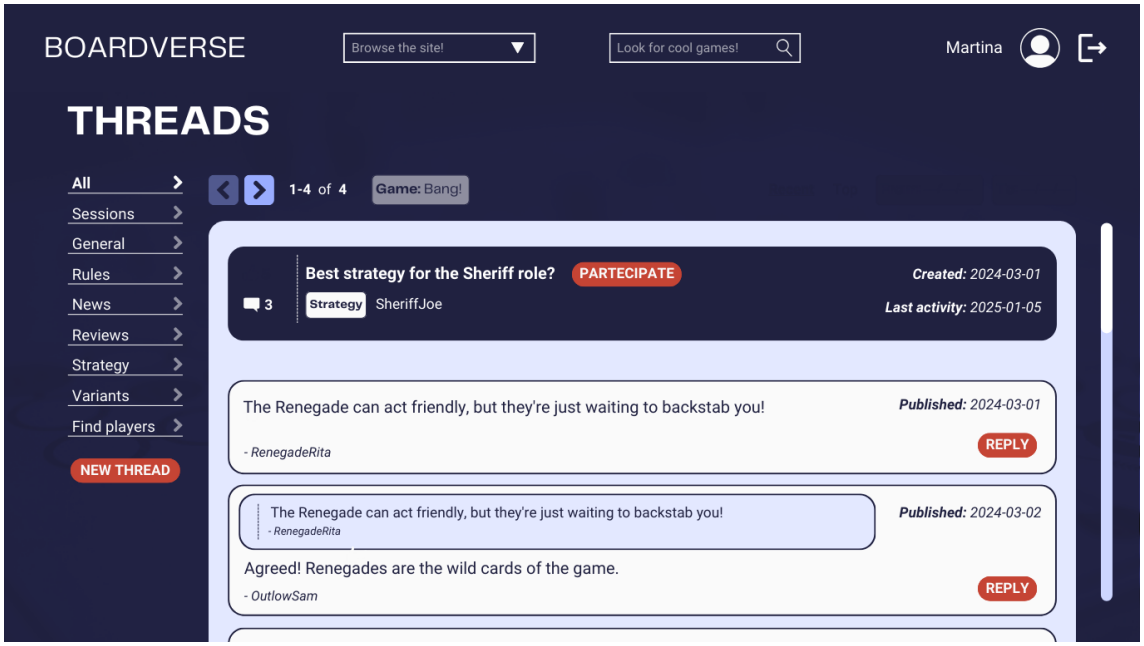


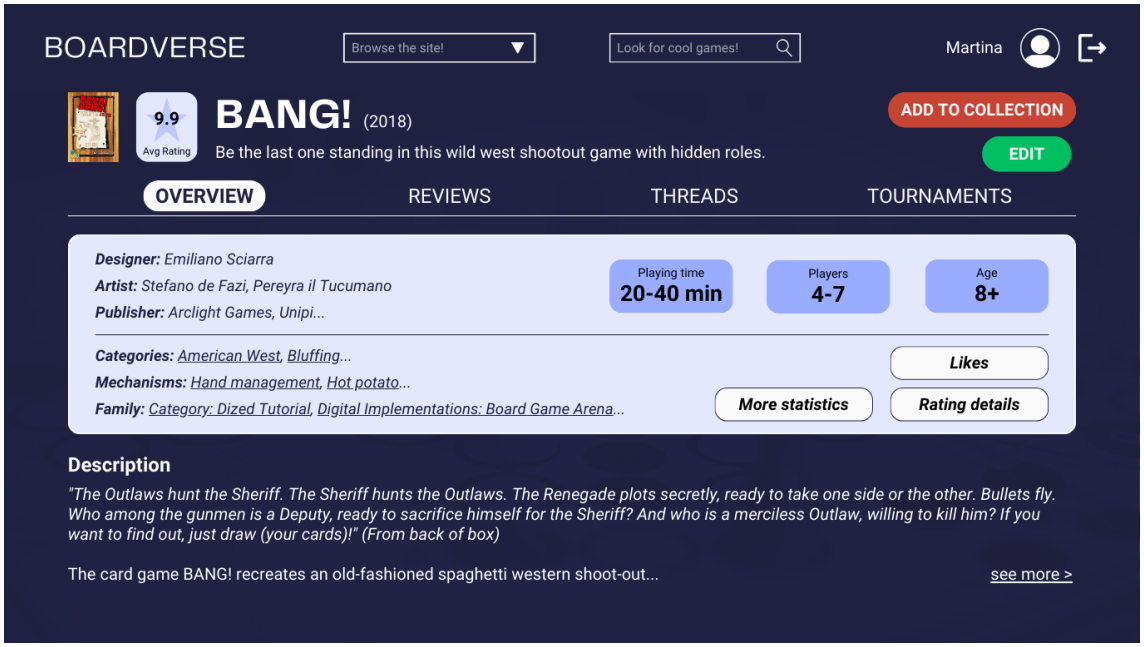

Martina
BOARDVERSE
Discover and share the best tabletop games
Hottest games
See more >
2018
1 BRASS: BIRMINGHAM
Build networks, grow industries, and navigate the world of the Industrial Revolution.
2015
2 PANDEMIC LEGACY...
Mutating diseases are spreading around the world - can your team
2021
ARK NOVA
Plan and build a modern, scientifically managed zoo to support conservation projects.
DeepSearch
Threads
Hotness
Most played
Most popular
Look for cool games!
Browse the site!
Search a user...

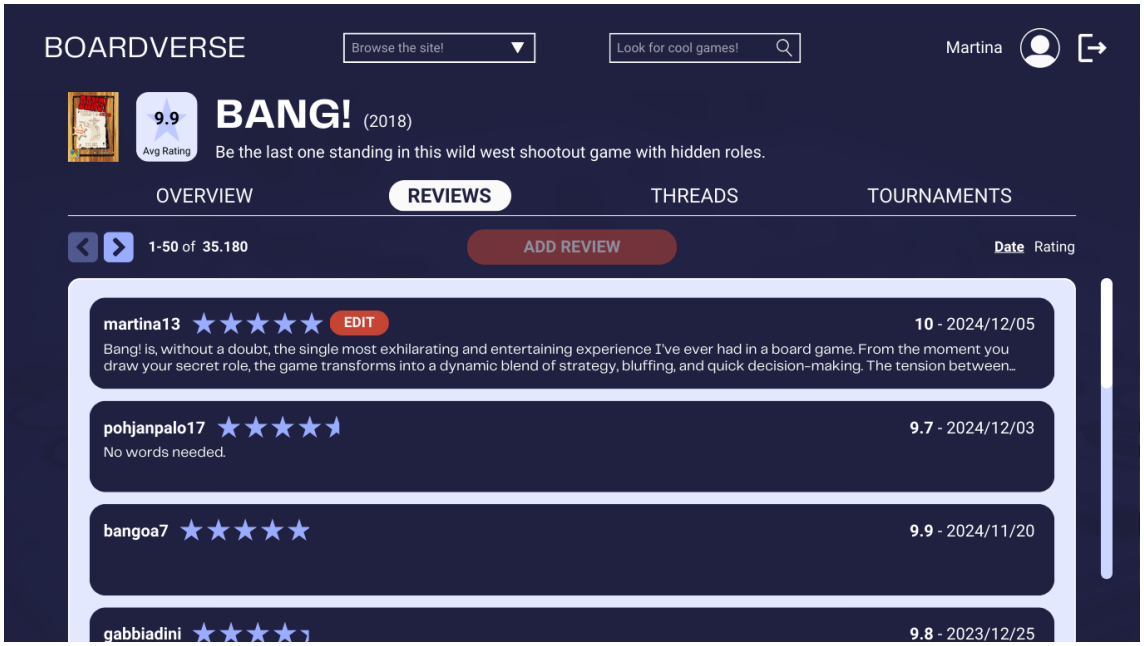

BOARDVERSE
Browse the site!
Look for cool games!
Martina
MOST POPULAR GAMES
1-50 of
Country
State or province
City
From: --/--/--
To: --/--/--
Rank
Title
Avg Rating
1
BANG! (2002)
Be the last one standing in this wild west shootout game with hidden roles.
9,9
2
Pandemic Legacy: Season 1 (2015)
Mutating diseases are spreading around the world - can your team save humanity?
8,57
3
Ark Nova (2021)
Plan and build a modern, scientifically managed zoo to support conservation projects.
8,53
4
Gloomhaven (2017)
Vanquish monsters with strategic cardplay. Fulfill your quest to leave your legacy!
8,40

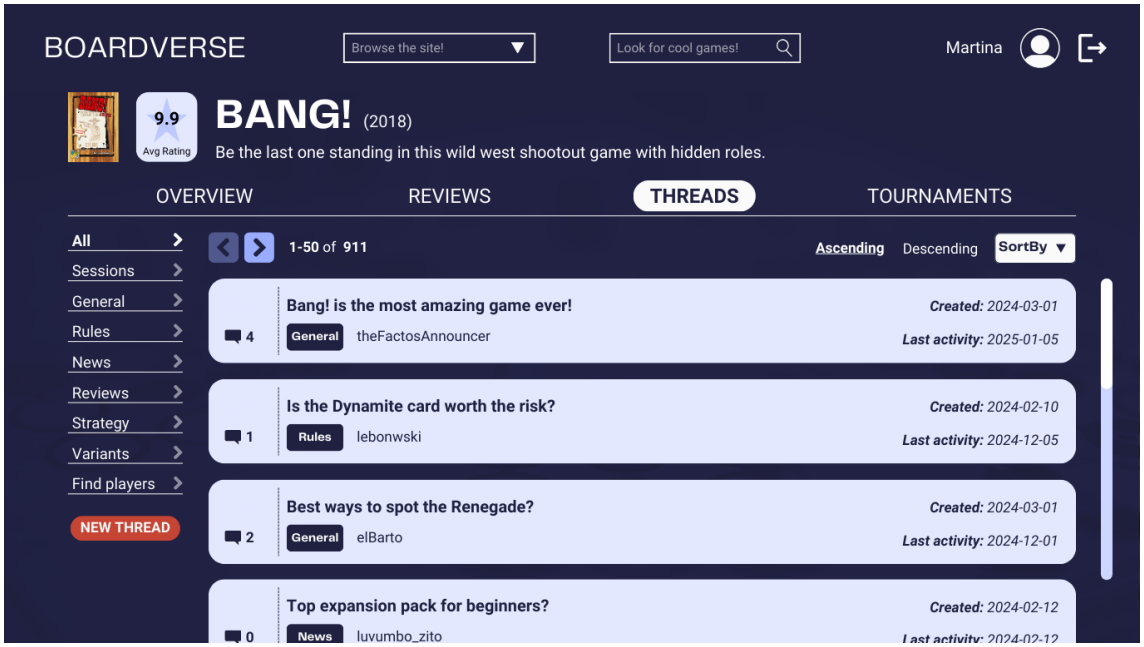

BOARDVERSE
Browse the site!
Look for cool games!
Martina
DEEPSEARCH
1-50 of
Descending
Ascending
Year of release
Category
Mechanic
SortBy
Rank
Title
Avg Rating
1
BANG! (2002)
Be the last one standing in this wild west shootout game with hidden roles.
9,9
2
Pandemic Legacy: Season 1 (2015)
Mutating diseases are spreading around the world – can your team save humanity?
8,57
3
Ark Nova (2021)
Plan and build a modern, scientifically managed zoo to support conservation projects.
8,53
4
Gloomhaven (2017)
Vanquish monsters with strategic cardplay. Fulfill your quest to leave your legacy!
8,40

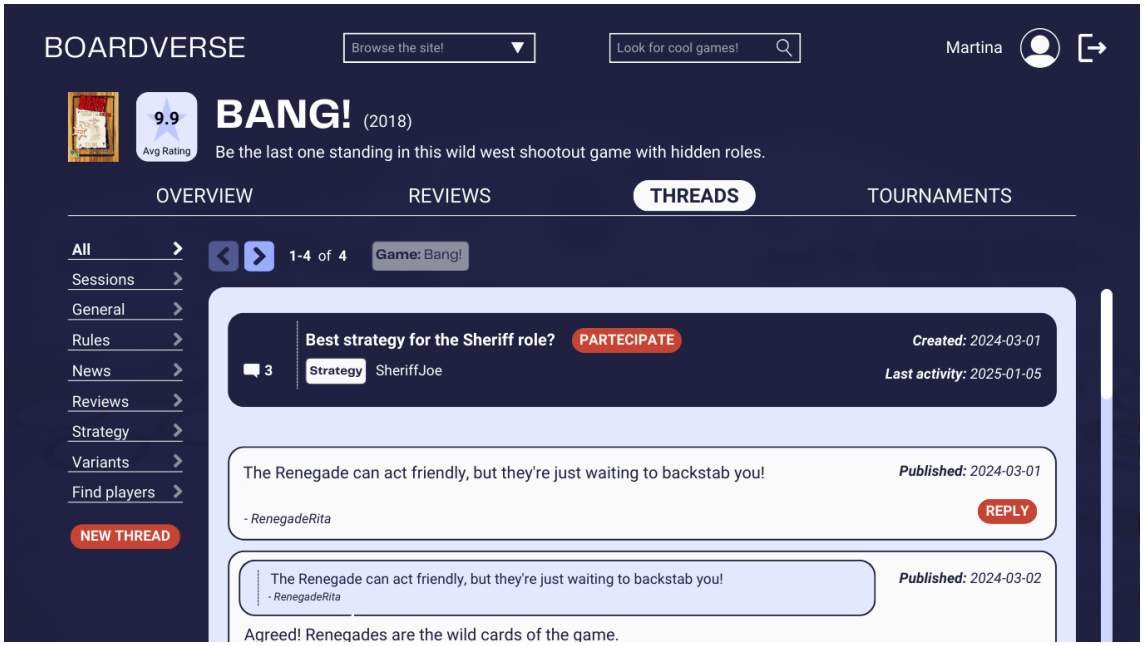

BOARDVERSE
Browse the site!
Look for cool games!
Martina
THREADS
All
Sessions
General
Rules
News
Reviews
Strategy
Variants
Find players
NEW THREAD
1-50
Ascending
Descending
Game: -----
SortBy
Year of release
From: --/--/--
To: --/--/--
4
TERRAFORMING MARS - What's the best starting corporation for beginners?
Strategy
MarsMaster
Created: 2024-03-01
Last activity: 2025-01-05
1
CATAN - How to secure longest road every time?
Strategy
RoadBuilderRick
Created: 2024-02-10
Last activity: 2024-12-05
2
DIXIT - How to keep the clues challenging but not too obscure?
General
Imaginativelvy
Created: 2024-03-01
Last activity: 2024-12-01
0
TICKET TO RIDE - Is it worth blocking routes to slow opponents down?
Rules
AndreTheMikeTysonFabbrino
Created: 2024-02-12
Last activity: 2024-02-12

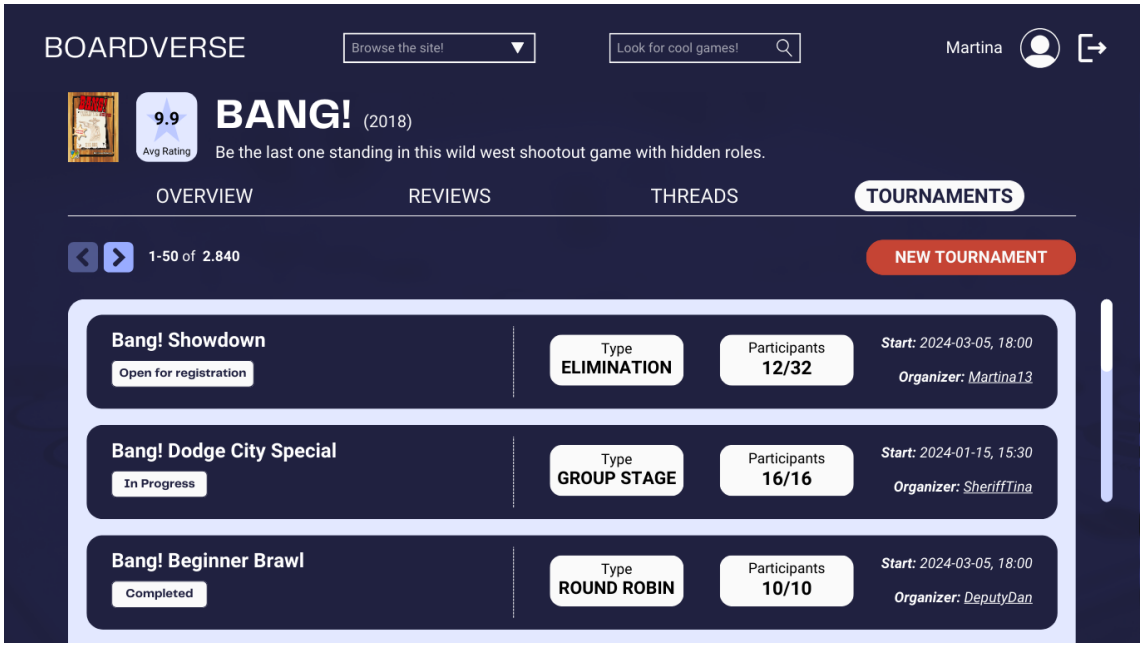

BOARDVERSE
Browse the site!
Look for cool games!
Martina
THREADS
All
Sessions
General
Rules
News
Reviews
Strategy
Variants
Find players
NEW THREAD
1-4 of 4
Game: Bang!
Best strategy for the Sheriff role?
PARTECIPATE
3
Strategy
SheriffJoe
Created: 2024-03-01
Last activity: 2025-01-05
The Renegade can act friendly, but they're just waiting to backstab you!
- RenegadeRita
Published: 2024-03-01
REPLY
The Renegade can act friendly, but they're just waiting to backstab you!
- RenegadeRita
Published: 2024-03-02
Agreed! Renegades are the wild cards of the game.
- OutlowSam
REPLY

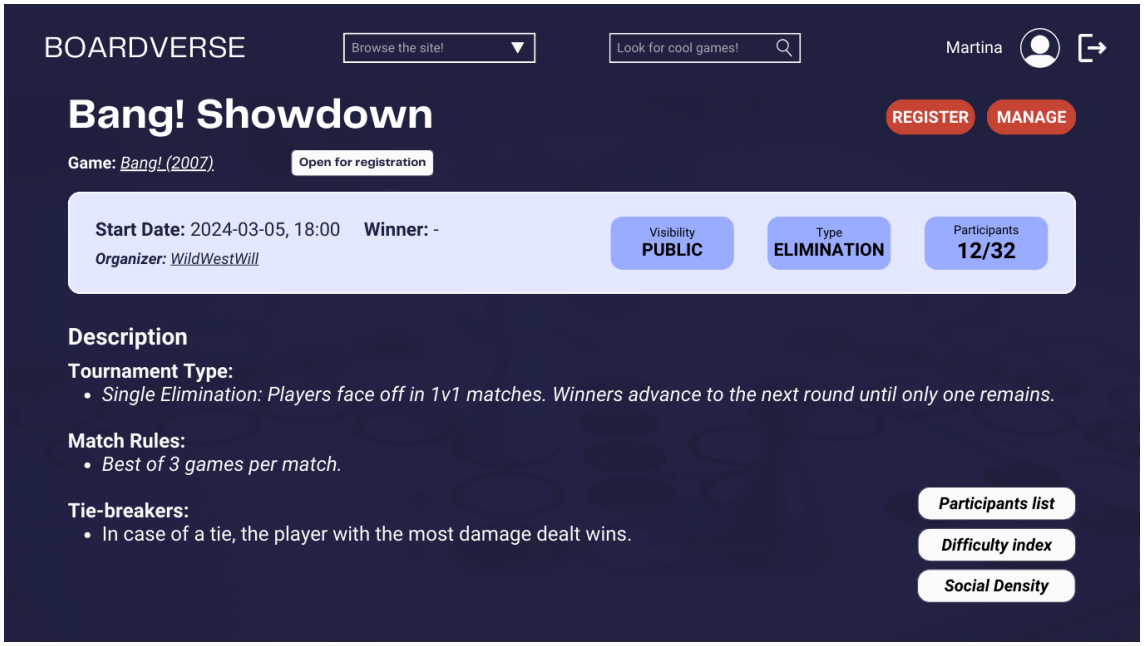

BOARDVERSE
Browse the site!
Look for cool games!
Martina
9.9
Avg Rating
BANG! (2018)
Be the last one standing in this wild west shootout game with hidden roles.
ADD TO COLLECTION
EDIT
OVERVIEW
REVIEWS
THREADS
TOURNAMENTS
Designer: Emiliano Sciarra
Artist: Stefano de Fazi, Pereyra il Tucumano
Publisher: Arclight Games, Unipi...
Playing time
20-40 min
Players
4-7
Age
8+
Categories: American West, Bluffing...
Mechanisms: Hand management, Hot potato...
Family: Category: Dized Tutorial, Digital Implementations: Board Game Arena...
Likes
More statistics
Rating details
Description
"The Outlaws hunt the Sheriff. The Sheriff hunts the Outlaws. The Renegade plots secretly, ready to take one side or the other. Bullets fly. Who among the gunmen is a Deputy, ready to sacrifice himself for the Sheriff? And who is a merciless Outlaw, willing to kill him? If you want to find out, just draw (your cards)!" (From back of box)
The card game BANG! recreates an old-fashioned spaghetti western shoot-out...
see more >

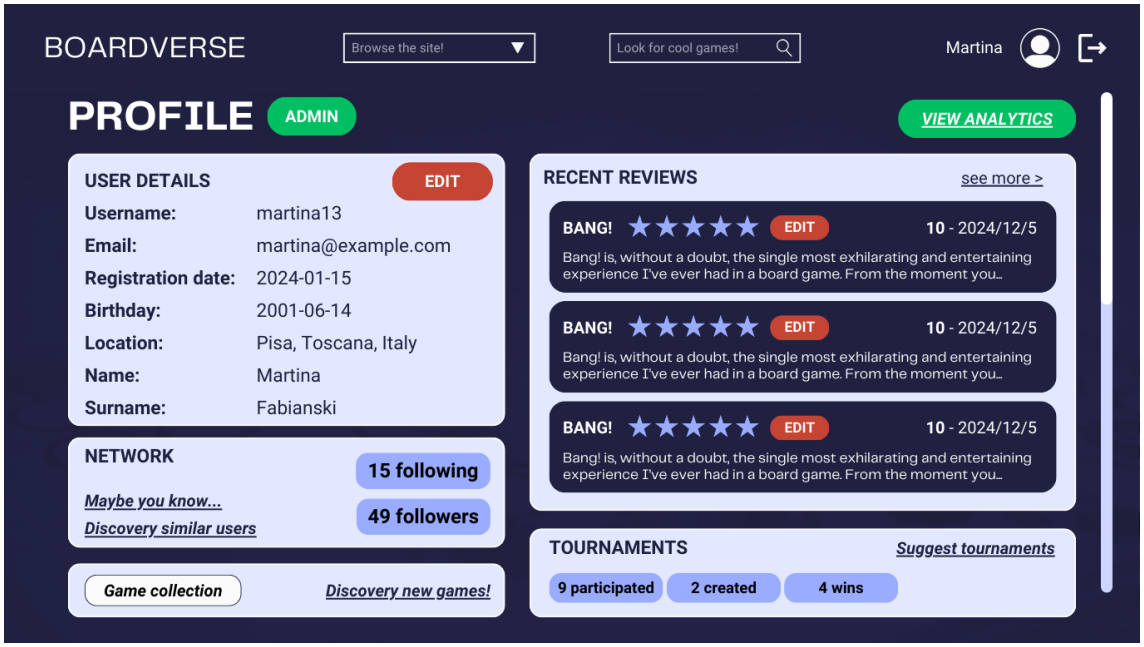

BOARDVERSE
Browse the site!
Look for cool games!
Martina
9.9
Avg Rating
BANG! (2018)
Be the last one standing in this wild west shootout game with hidden roles.
OVERVIEW
REVIEWS
THREADS
TOURNAMENTS
1-50 of 35.180
ADD REVIEW
Date Rating
martina13
EDIT
10 - 2024/12/05
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you draw your secret role, the game transforms into a dynamic blend of strategy, bluffing, and quick decision-making. The tension between...
pohjanpalo17
9.7 - 2024/12/03
No words needed.
bangoa7
9.9 - 2024/11/20
gabbiadini
9.8 - 2023/12/25

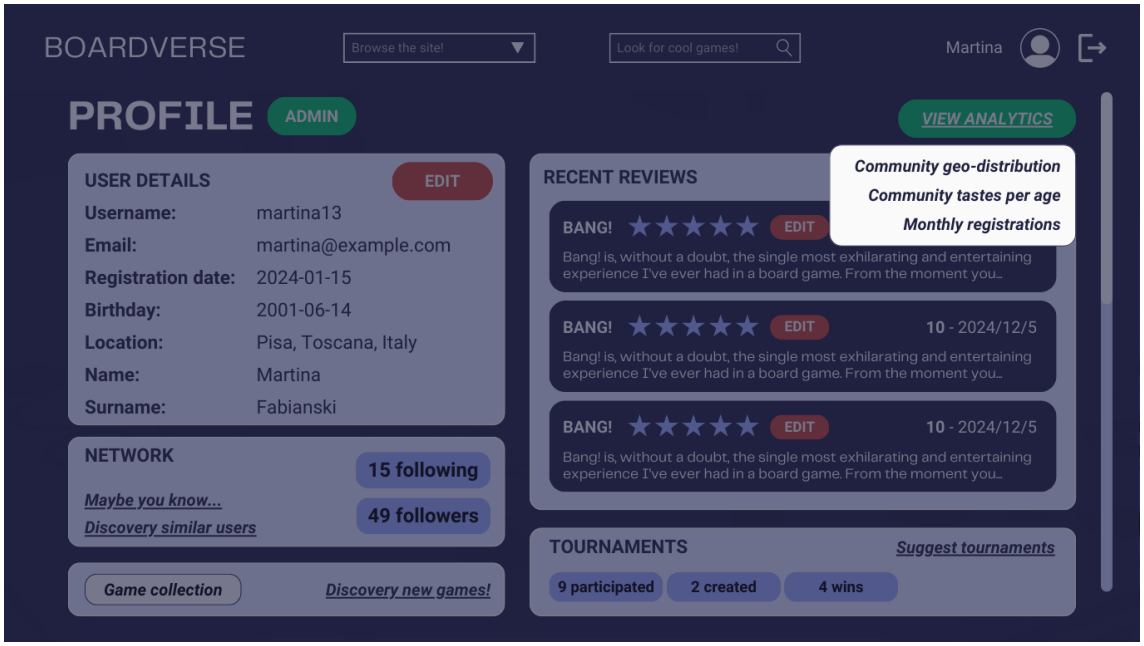

BOARDVERSE
Browse the site!
Look for cool games!
Martina
9.9
Avg Rating
BANG! (2018)
Be the last one standing in this wild west shootout game with hidden roles.
OVERVIEW
REVIEWS
THREADS
TOURNAMENTS
All
Sessions
General
Rules
News
Reviews
Strategy
Variants
Find players
NEW THREAD
1-50 of 911
Ascending
Descending
SortBy
4
Bang! is the most amazing game ever!
General
theFactosAnnouncer
Created: 2024-03-01
Last activity: 2025-01-05
1
Is the Dynamite card worth the risk?
Rules
lebonwski
Created: 2024-02-10
Last activity: 2024-12-05
2
Best ways to spot the Renegade?
General
elBarto
Created: 2024-03-01
Last activity: 2024-12-01
0
Top expansion pack for beginners?
News
luvumbo_zito
Created: 2024-02-12


BOARDVERSE
Browse the site!
Look for cool games!
Martina
9.9
Avg Rating
BANG! (2018)
Be the last one standing in this wild west shootout game with hidden roles.
OVERVIEW
REVIEWS
THREADS
TOURNAMENTS
All
Sessions
General
Rules
News
Reviews
Strategy
Variants
Find players
NEW THREAD
1-4 of 4
Game: Bang!
Best strategy for the Sheriff role?
PARTECIPATE
3
Strategy
SheriffJoe
Created: 2024-03-01
Last activity: 2025-01-05
The Renegade can act friendly, but they're just waiting to backstab you!
- RenegadeRita
Published: 2024-03-01
REPLY
The Renegade can act friendly, but they're just waiting to backstab you!
- RenegadeRita
Published: 2024-03-02
Agreed! Renegades are the wild cards of the game.


BOARDVERSE
Browse the site!
Look for cool games!
Martina
9.9
Avg Rating
BANG! (2018)
Be the last one standing in this wild west shootout game with hidden roles.
OVERVIEW
REVIEWS
THREADS
TOURNAMENTS
1-50 of 2.840
NEW TOURNAMENT
Bang! Showdown
Open for registration
Type
ELIMINATION
Participants
12/32
Start: 2024-03-05, 18:00
Organizer: Martina13
Bang! Dodge City Special
In Progress
Type
GROUP STAGE
Participants
16/16
Start: 2024-01-15, 15:30
Organizer: SheriffTina
Bang! Beginner Brawl
Completed
Type
ROUND ROBIN
Participants
10/10
Start: 2024-03-05, 18:00
Organizer: DeputyDan


BOARDVERSE
Browse the site!
Look for cool games!
Martina
Bang! Showdown
REGISTER
MANAGE
Game: Bang! (2007)
Open for registration
Start Date: 2024-03-05, 18:00
Winner: -
Organizer: WildWestWill
Visibility
PUBLIC
Type
ELIMINATION
Participants
12/32
Description
Tournament Type:
• Single Elimination: Players face off in 1v1 matches. Winners advance to the next round until only one remains.
Match Rules:
• Best of 3 games per match.
Tie-breakers:
• In case of a tie, the player with the most damage dealt wins.
Participants list
Difficulty index
Social Density


BOARDVERSE
Browse the site!
Look for cool games!
Martina
PROFILE
ADMIN
VIEW ANALYTICS
USER DETAILS
EDIT
Username: martina13
Email: martina@example.com
Registration date: 2024-01-15
Birthday: 2001-06-14
Location: Pisa, Toscana, Italy
Name: Martina
Surname: Fabianski
NETWORK
15 following
49 followers
Maybe you know...
Discovery similar users
Game collection
Discovery new games!
RECENT REVIEWS
see more >
BANG! EDIT 10 - 2024/12/5
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you...
BANG! EDIT 10 - 2024/12/5
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you...
BANG! EDIT 10 - 2024/12/5
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you...
TOURNAMENTS
Suggest tournaments
9 participated
2 created
4 wins


BOARDVERSE
Browse the site!
Look for cool games!
Martina
PROFILE
ADMIN
VIEW ANALYTICS
Community geo-distribution
Community tastes per age
Monthly registrations
USER DETAILS
EDIT
Username: martina13
Email: martina@example.com
Registration date: 2024-01-15
Birthday: 2001-06-14
Location: Pisa, Toscana, Italy
Name: Martina
Surname: Fabianski
RECENT REVIEWS
BANG! EDIT
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you...
BANG! EDIT 10 - 2024/12/5
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you...
BANG! EDIT 10 - 2024/12/5
Bang! is, without a doubt, the single most exhilarating and entertaining experience I've ever had in a board game. From the moment you...
NETWORK
15 following
Maybe you know...
Discovery similar users
49 followers
Game collection
Discovery new games!
TOURNAMENTS
Suggest tournaments
9 participated
2 created
4 wins

Chapter 3

Requirements

3.1 Functional Requirements

- **Unregistered User**

  1. The system must allow an unregistered user to browse games by name, category, year released and mechanics.
  2. The system must allow an unregistered user to see the hottest, most popular, most played game rankings.
  3. The system must allow an unregistered user to view detailed information about a selected game, including related reviews and threads.
  4. The system must allow an unregistered user to browse threads by game, last post date and tag.
  5. The system must allow an unregistered user to display messages in a specific thread.
  6. The system must allow an unregistered user to register by providing personal information.

- **Registered User**
  In addition to the previous requirements:

  1. The system must allow a registered user to login by providing username and password.
  2. The system must allow a registered user to edit personal information.

3. The system must allow a registered user to view their personal page.
4. The system must allow a registered user to find other registered user by username.
5. The system must allow a registered user to add a review to a game. A review can include a vote from 1 to 10, a comment or both.
6. The system must allow a registered user to edit a own review.
7. The system must allow a registered user to delete a own review.
8. The system must allow a registered user to create a thread for a game. A thread must include a subject, a tag and a content.
9. The system must allow a registered user to edit a own thread.
10. The system must allow a registered user to delete a own thread.
11. The system must allow a registered user to add a message or reply to a message in a thread.
12. The system must allow a registered user to delete a own message in a thread.
13. The system must allow a registered user to edit a own tournament.
14. The system must allow a registered user to delete a own tournament.
15. The system must allow a registered user to display tournaments related to a specific game for which he's allowed to.
16. The system must allow a registered user to view detailed information about a selected tournament, including participants, social density index and difficulty level.
17. The system must allow a registered user to create a tournament for a game. A tournament must include a name, type, description, location, minimum and maximum number of participants, start date and visibility.
18. The system must allow a registered user to participate to a tournament.

19. The system must allow a registered user to remove participation from a tournament.
20. The system must allow a registered user to update information related to a tournament of its own.
21. The system must allow a registered user to select a winner for a tournament of its own.
22. The system must allow a registered user to add a game to its own collection.
23. The system must allow a registered user to remove a game to its own collection.
24. The system must allow a registered user to follow another registered user.
25. The system must allow a registered user to unfollow another registered user.
26. The system must allow a registered user to see the list of followed users.
27. The system must allow a registered user to see the list of followers.
28. The system must allow a registered user to see the list of collected games.
29. The system must suggest a registered user a list of registered user who might know.
30. The system must suggest a registered user a list of registered user with similar tastes.
31. The system must suggest a registered user a list of games that may be of interest for the user.
32. The system must suggest a registered user a list of tournaments that may be of interest for the user.

- **Administrator**
  In addition to all the previous requirements:

  1. The system must allow ad administrator to add a game.
  2. The system must allow ad administrator to update a game.

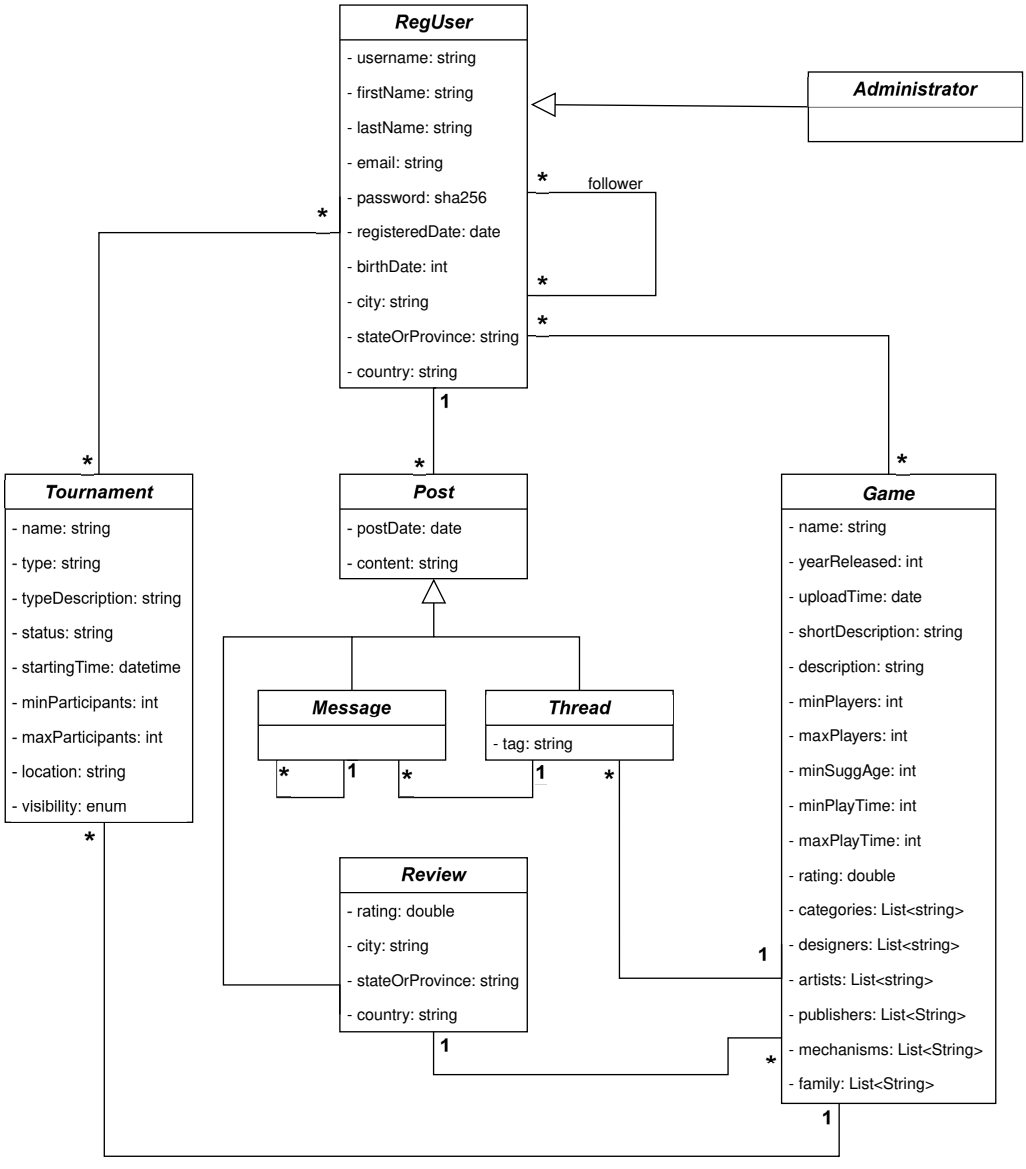
3. The system must allow ad administrator to delete a game.
4. The system must allow ad administrator to delete a user.
5. The system must allow ad administrator to delete a review.
6. The system must allow ad administrator to delete a thread.
7. The system must allow ad administrator to delete a message.
8. The system must allow ad administrator to delete a tournament.
9. The system must allow ad administrator to view the community geographical distribution analytic.
10. The system must allow ad administrator to view the community tastes per age bucket analytic.
11. The system must allow ad administrator to view the overall monthly registration distribution analytic.

3.2 Non-Functional Requirements

1. The system must be accessible as a web application compatible with major modern web browsers (Chrome, Firefox, Safari, Edge).
2. The system must be developed in an OOP language.
3. The system must ensure high availability.
4. The system should be capable of handling growth in user traffic by scaling effectively.
5. The system must be fault-tolerant and capable of maintaining essential functionality in the event of partial system failures.
6. The system must use a basic authentication protocol to secure access.
7. The system must encrypt users' passwords.

Chapter 4

UML Class Diagram



Chapter 5

Database

5.1 Operations Volumes

The following table presents an estimated number of occurrences for each of the primary operations performed by the application. These figures are based on an assumption of 100,000 active users per day.

Action	**%**	**#Daily occurrences**	**MongoDB**	**Neo4j**	**Daily total**
Register user	0.01	10	1	1	20
Login user	100	100,000	1	0	100,000
Browse users	20	20,000	1	0	20,000
User page load	80	2 × 80,000	1	0	80,000
Update user info	0.01	10	4	1	40
Browse games	120	3 × 120,000	1	0	360,000
Load game page	120	3 × 120,000	1	0	360,000
Load ranking	50	50,000	1	0	50,000
Load hottest games	30	30,000	1	0	30,000
Write a review	10	10,000	3	0	30,000
Load game reviews	50	50,000	1	0	50,000
Create a thread	1	1,000	1	0	1,000
Write a message	5	5,000	1	0	5,000
Delete a message	0.01	10	1	0	10
Load game threads	30	30,000	1	0	30,000
Find threads	10	10,000	1	0	10,000
Load thread	25	3 × 25,000	1	0	75,000
Create a tournament	0.1	100	2	1	300
Load game tournaments	25	25,000	1	0	25,000
Load tournament info	10	10,000	1	0	10,000
Follow a user	5	5,000	1	1	10,000
Unfollow a user	0.5	500	1	1	1,000
Game recommendation	20	20,000	0	1	20,000
User recommendation	15	15,000	0	1	15,000
Tournament recommendation	5	5,000	0	1	5,000
Like a game	10	10,000	0	1	10,000
Participate in a tournament	0.5	500	1	1	500

5.2 Document DB

5.2.1 Collections

- **USER:** Contains all documents related to registered users and admin with their personal information provided at the time of registration along with other useful fields for application purposes such as the following:
  - **registeredDate:** timestamp of registration.
  - **location:** country, state and city of the user.
  - **followers:** number of users following the user.
  - **following:** number of users followed by the user.
  - **tournaments:** number of created, partecipated and won tournaments.
  - **mostRecentReviews:** array of the three last reviews left by the user, made using the embedding of documents.
- **GAME:** The collection contains all documents about games information, with additional information among which we can find:
  - **uploadTime:** timestamp where the admin added the game.
  - **averageRating:** average rating of the game.
  - **ratingVoters:** number of votes given to the game.
- **REVIEW:** This collection contains all the reviews made for each game along with user metadata, such as:
  - **authorBirthDate:** Date of birth of the review author.
  - **location:** country, state and city of the user at the time of review.
- **THREAD:** collection containing thread-related documents. Threads contains the messages and information about the related game.
- **TOURNAMENT:** collection contains information about tournaments. One of its key fields is visibility, which determines who can access and participate in the tournament. The *visibility* field can have three possible values:

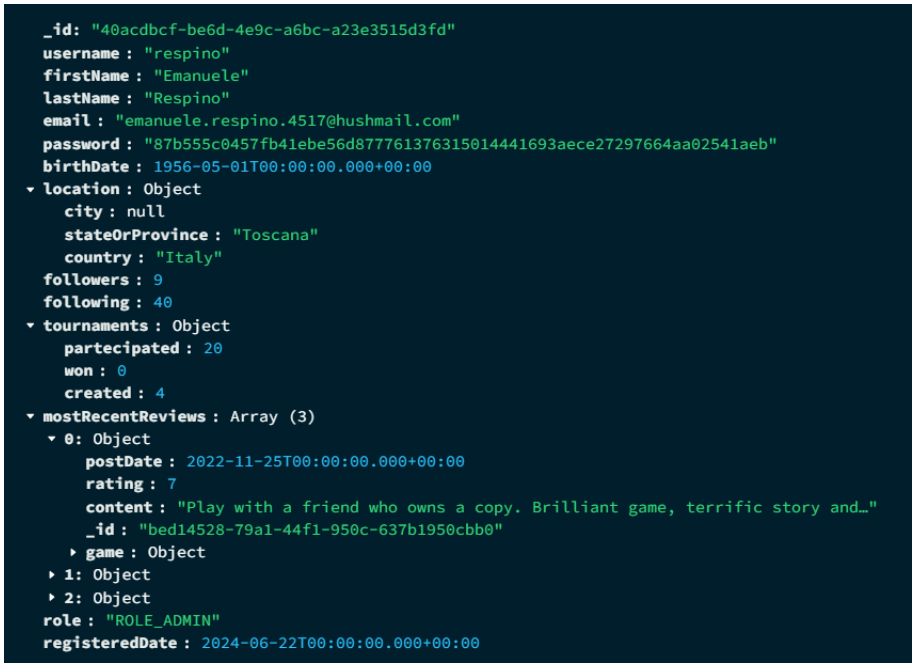
– PUBLIC: The tournament is open to everyone, and any user can visualize in tournaments list and join it.

– INVITE: Only users who receive a direct invitation from the administrator can participate.

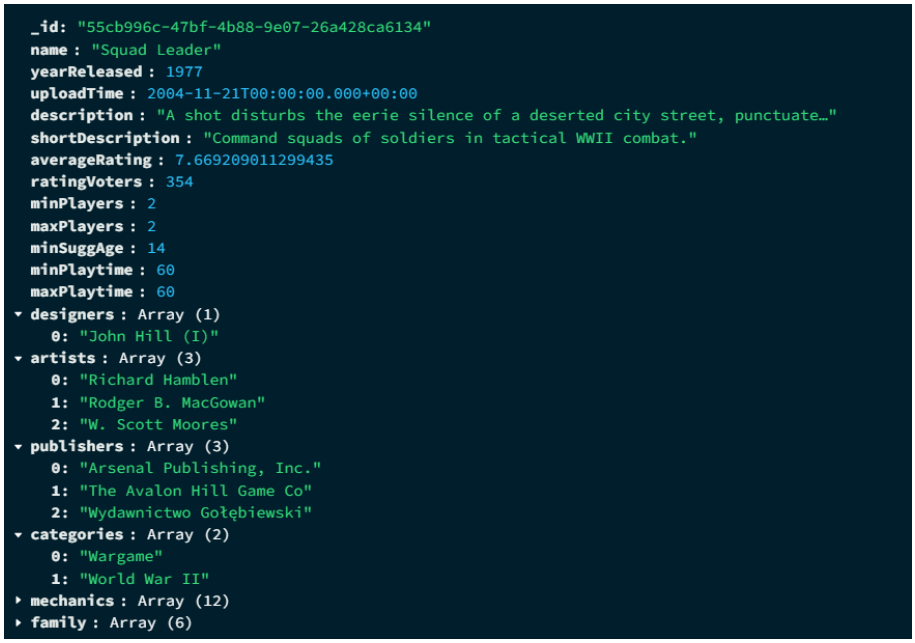
– PRIVATE: The tournament is restricted, and only specific users can access it. The administrator explicitly lists the allowed users by storing their IDs in the *allowed* field.

5.2.2 Examples of documents

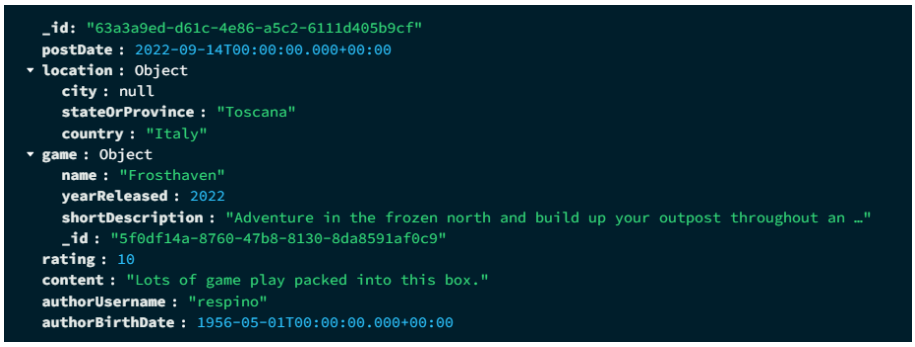
User document



Game document

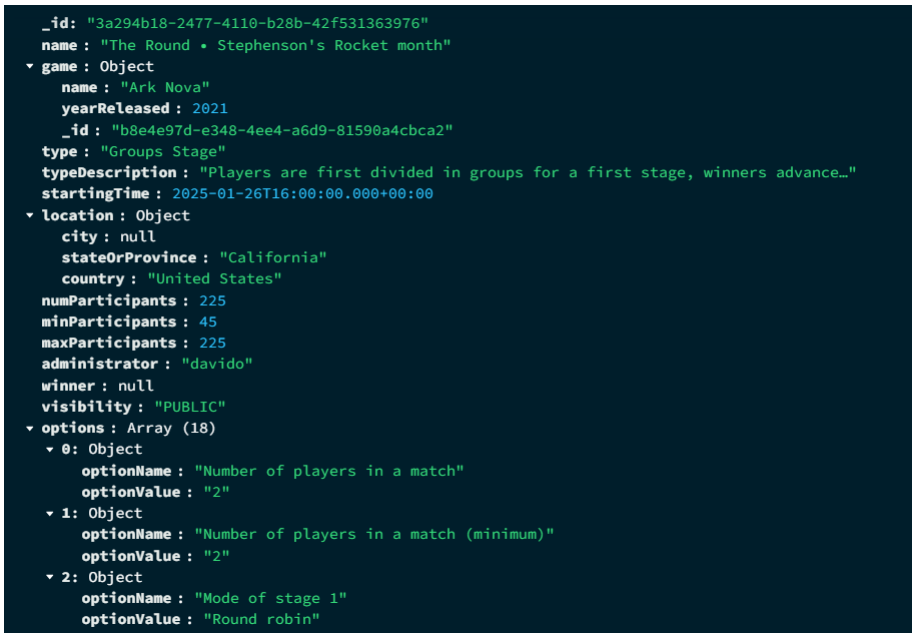


Review document



Thread document

Tournament document



5.2.3 CRUD Operations

- Create
  - User.

  - Game.
  - Review.
  - Thread.
  - Tournament.

- **Read**
  - User by id.
  - User by username.
  - Game by id.
  - Game by name.
  - Game by category.
  - Game by yearReleased.
  - Game by mechanic.
  - Review by game id.
  - Review by author username.
  - Thread by id.
  - Thread by game id.
  - Thread by game yearReleased.
  - Thread by postDate.
  - Thread by tag.
  - Tournament by id.
  - Tournament by game id.

- **Update**
  - Update user information.
  - Update game information.
  - Update review details.
  - Update thread details.
  - Add a new message in a thread.
  - Reply to a message in a thread.

  - Delete a message in a thread.
  - Edit a message in a thread.
  - Update tournaments details.

- **Delete**
  - User.
  - Game.
  - Review.
  - Thread.
  - Tournament.

5.2.4 Queries

Queries are described in more detail in the appropriate chapter.

- **Games ranking based on reviews filtered by postDate and user Location**
- **Top 10 Most Played Games**
- **Hottest games based on threads activity**
- **Game rating details**
- **Community geographical distribution**
- **Monthly user registrations by Year**
- **Community tastes per age bucket**

5.3 Graph DB

Within the graph we decided to track part of the activity of the user within the platform, starting from the interaction with other users through the follow feature, interaction with games allowing to place a game among the favorites and also everything that concerns the competitive side of a user with tournaments. Thanks to the implementation of the graph database it is possible to provide an improved and customized user experience.

5.3.1 Nodes

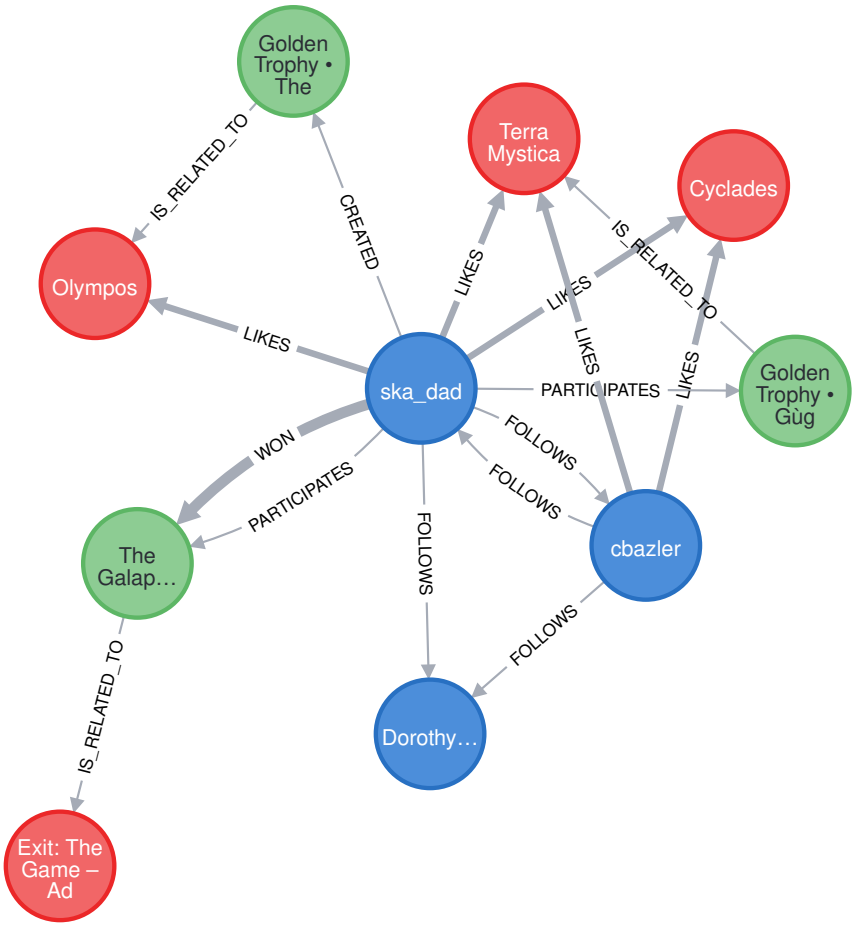
- **User:** Simple node for the registered user only contains the username.
- **Game:** node containing id, name, yearReleased, shortDescription, categories.
- **Tournament:** containing id, name, visibility, maxParticipants, startingTime.

5.3.2 Relationship

- **USER - FOLLOW → USER**: Relationship between two users created when one user starts following another one.
- **USER - LIKES → GAME**: Relationship between a user and a game, represents the inclusion of that game in the user's favorites list.
- **USER - CREATED → TOURNAMENT**: Tournament user relationship created when the user creates a tournament. The user who has this relationship with the tournament is the administrator of the tournament.
- **USER - PARTECIPATES → TOURNAMENT**: Relationship that specifies the user's participation in the tournament.
- **USER - WON → TOURNAMENT**: Relationship created at the end of a tournament that specifies a user's win in a tournament.
- **TOURNAMENT - IS_RELATED_TO → GAME**: Relationship created when a user creates a tournament that allows you to associate a particular tournament with the game it was created for.

All these relationships, except for IS_RELATED_TO, have the timestamp of when they were created.

5.3.3 Examples of Graph



5.3.4 CRUD Operations

CREATE	
Domain-specific	**Graph-centric**
Add a new User	Add new User vertex
Admin add a new Game	Add new Game vertex
User create a new Tournament	User add new Tournament vertex and add new "CREATED" relationship and a relationship "IS_RELATED_TO" is created from the tournament node to the game node.
User follows another user	Add new “FOLLOWS” relationships between the two users from the first user to the second one
User likes a Game	Add new “LIKES” relationship between User and Game
User Join a Tournament	Add new “PARTECIPATES” relationship between User and Tournament
User won a Tournament	Add new “WON” relationship between User and Tournament

READ	
Domain-specific	**Graph-centric**
Find all friends of an User	How many "FOLLOWS" egde are outgoing from a specific User vertex?
Find all the tournaments a user is participating in	How many "PARTECIPATES" egde are outgoing from a specific User vertex?
Find all tournaments a user has won	How many "WON" egde are outgoing from a specific User vertex?
Find all tournaments that a user is a creator of	How many "CREATED" egde are outgoing from a specific User vertex?
Find all the games that the user likes	How many "LIKES" egde are outgoing from a specific User vertex?

UPDATE	
Domain-specific	**Graph-centric**
User update username	Update username property in the user node.
Creator of a tournament update the tournament info	Update tournament node properties
Admin update game info	Update game node properties

DELETE	
Domain-specific	**Graph-centric**
User is deleted	Remove a User node, all the Tournament nodes linked with "CREATED" relationship and all others "FOLLOWS", "PARTECIPATES" and "LIKES" relationship.
Game is deleted	Remove a Game node, all tournament nodes connected via "IS_RELATED_TO" relationship, and all "LIKES", "CREATED", "PARTECIPATES", "WON", "IS_RELATED_TO" relationships.
User removes a like to a game	Remove a "LIKES" relationship between User and Game
User no longer participates in a tournament	Remove a "PARTECIPATES" relationship between User and Tournament.
Tournament owner cancels the tournament	Remove a "TOURNAMENT" node and all its "OWNS", "PARTECIPATES" and "IS_RELATED_TO" connections
User removes follow to another user	Remove "FOLLOWS" relationship from first user to second one.

5.3.5 Queries

Queries are described in more detail in the appropriate chapter.

- **Suggest new friends**
- **Suggest users with similar tastes**
- **Board games recommendation**
- **Tournaments recommendation**

Chapter 6

Distributed Database Design

As outlined in section 3.2, our system is designed to guarantee High Availability and Partition Tolerance, ensuring fast responses to user requests. In alignment with the CAP theorem, the application adopts an AP-oriented approach, prioritizing availability and partition tolerance while compromising on consistency by implementing eventual consistency. In this chapter, we will analyze in detail the strategies for data distribution and replication, which form the foundation of the system's design.

6.1 Replicas

To ensure high availability, we have configured a replica set using the virtual machines available for this project. Specifically, the cluster includes three replicas dedicated to the document database (MongoDB). The organization of the replica set is as follows:

- **Primary node:** The node with the IP address 10.1.1.20 acts as the primary server and is responsible for handling all incoming requests. It has a priority level of 5, making it the preferred leader in the replica set.
- **Secondary nodes:** Two additional nodes, with IP addresses 10.1.1.21 (priority 2) and 10.1.1.22 (priority 1), store synchronized copies of

the data from the primary server. These nodes ensure data redundancy and support failover mechanisms in case the primary node becomes unavailable.

This setup allows the system to continue functioning even if the primary node becomes unavailable. In such a case, an automatic election process ensures that one of the secondary nodes is promoted to primary, minimizing downtime and maintaining service continuity.

To enhance system reliability, we have configured the nodes with different priority levels. The node with the highest priority (priority 5) is most likely to become the primary node during an election, ensuring a stable and predictable leadership structure. The node with the lowest priority (priority 1) also hosts the Neo4j service, and this configuration minimizes its workload by making it a candidate for primary only as a last resort. This approach carefully balances the workload between MongoDB and Neo4j, preventing resource conflicts and maintaining the performance of both databases.

Although the replica set currently includes three replicas exclusively for MongoDB and only one replica for Neo4j, this configuration is designed with future improvements in mind. While Neo4j's single replica is sufficient for this phase of the project, additional replicas can be added later to enhance its fault tolerance and availability. For MongoDB, the existing three-node cluster already provides strong support for high availability and redundancy.

Overall, the replica set design provides a solid foundation for a distributed database system that is highly available, fault-tolerant, and prepared for future scalability.

In our MongoDB configuration, we introduced the following constraints in the URI connection string:

mongodb://10.1.1.20:27017,10.1.1.21:27017,10.1.1.22:27017/?
replicaSet=lsmdb&w=majority&readPreference=nearest&retryWrites=true

- **readPreference=nearest:** This parameter directs read operations to the nearest node in terms of network latency. This improves performance by reducing the time required to fulfill read requests, particularly in distributed environments.It also helps distribute the

workload more evenly across the replica set, as secondary nodes can participate in serving read requests, reducing the pressure on the primary node. While this configuration may occasionally return slightly stale data due to the eventual consistency model of MongoDB, it is a reasonable tradeoff in scenarios where speed and availability are prioritized.

- **w=majority:** This parameter ensures write operations are acknowledged only after being committed to the majority of nodes in the replica set. In our setup, this means at least two out of three nodes must confirm the write operation. This configuration guarantees data durability, as the system can recover from a primary node failure without losing data, provided the write has been replicated to the majority. Additionally, it maintains a high level of consistency, ensuring that the most recent updates are preserved across the replica set. While this approach may slightly increase write latency due to the need for replication confirmation, the tradeoff is well worth it for the added reliability and fault tolerance.

Together, these configurations create a balance between performance and resilience. The system delivers faster read responses by leveraging the nearest available node while ensuring that data remains safe and consistent through majority write acknowledgments.

6.2 Handling Inter-DB Consistency

Having two different types of databases in the system introduces some redundant data, which creates the need to carefully manage operations that interact with both databases. These operations, such as user registration or game information updates, require maintaining consistency between the two. To achieve this, we use **retryable** Spring functions that ensure operations are executed reliably.

The process begins with attempting the operation on the Neo4j database, as this is the more fragile system due to having only a single replica node. If the operation on Neo4j is successful, it then proceeds to execute the corresponding operation on the MongoDB database. If both operations are completed successfully, the function returns a success response to

the user. However, if either of the two operations fails, the system automatically retries the entire process up to three times, with a fixed delay between retries. If all retries fail, the operation is marked as unsuccessful, and an error response is returned to the user.
It's important to note that this approach does not include a rollback mechanism for changes made to Neo4j if the MongoDB operation fails. This could potentially lead to temporary inconsistencies between the databases. However, the likelihood of this happening is very low, given that MongoDB operates with a replica set of three nodes, providing greater reliability.

In the event that MongoDB fails during an operation, the retry mechanism starts from the very beginning of the function, including reattempting the write operation on Neo4j. This is not problematic because the write function is designed to be idempotent, meaning that repeating the operation will not create duplicate or conflicting entries. This design ensures that operations remain reliable and consistent across both databases, even in the face of potential failures.

6.3 Sharding

In our project, we have identified the collections that will benefit from sharding: Users, Games, Reviews, Threads and Tournaments. The selection of shard keys and the corresponding sharding strategies has been carefully designed, taking into account factors such as data access patterns, the anticipated growth rate of each collection, and the need to maintain optimal query performance. This approach ensures efficient data distribution and scalability across all collections, tailored to their specific usage scenarios.

- **Users collection**
  Sharding key: Hashed Username

  We have chosen to use the hashed username as the shard key to ensure a more uniform distribution of data across shards, preventing imbalances and facilitating scalability. This approach is also highly effective given the frequent queries based on the username, as it allows the system to directly route the query to the appropriate shard. This significantly enhances performance and efficiency.

- **Games collection**
  Sharding key: Hashed id
  The hashed ID has been chosen as the sharding key due to the high frequency of queries that rely on the ID to identify games. Additionally, this approach ensures seamless scalability. When new shards are added, data can be automatically redistributed across the cluster, thanks to the hashing mechanism, maintaining an optimal balance. This design enhances the system's resilience and makes it well-equipped to handle a growing volume of data efficiently.

- **Reviews collection**
  Sharding key: Hashed gameID
  For the Reviews collection, hashed gameID was chosen as the sharding key for several reasons:

  - *Uniform Data Distribution:* By hashing the gameID, the reviews are distributed evenly across all shards, preventing any single shard from becoming overloaded. This is particularly important given that some games may have significantly more reviews than others.
  - *Query Optimization:* Many queries in the Reviews collection are likely to focus on retrieving reviews for a specific game. Using gameID as the sharding key allows these queries to be directed to the relevant shard directly, significantly improving query performance by avoiding inter-shard operations.
  - *Scalability:* The hashed gameID ensures that as the number of reviews grows, new shards can be added seamlessly, and the data will be automatically redistributed in a balanced way. This makes the system capable of handling increased data volumes efficiently.
  - *Avoiding Hotspots:* Games with a large number of reviews could create hotspots if the data were not evenly distributed. Hashing the gameID breaks any natural patterns in the distribution of data, ensuring that the load is balanced across the cluster.

- **Threads collection**
  Sharding key: Hashed gameID

As with the Reviews collection, the Threads collection also uses the hashed gameID as the sharding key. This choice was made because gameID is a field present in all documents, making it a reliable and consistent key for sharding. Additionally, the decision is supported by the same reasons outlined above, including uniform data distribution, query optimization, and improved scalability.

- **Tournaments collection**
  Sharding key: Hashed gameID

  Since tournaments can only be accessed from the corresponding game's page, we have chosen to use hashed gameID as the sharding key. This ensures that all tournament data related to a specific game is stored within the same shard, optimizing query performance by allowing direct access to the relevant shard when retrieving tournament information. Additionally, hashing the gameID ensures an even distribution of tournament data across shards, maintaining a balanced load and supporting scalability as the number of games and tournaments grows. This approach aligns with the hierarchical relationship between games and tournaments, streamlining data access and system efficiency.

Chapter 7

Dataset description

The core of this application revolves around managing and analyzing a large-scale dataset in the context of board games. To achieve this, it was essential to retrieve a robust and relevant dataset that would serve as the foundation for our application. For this purpose, we relied on two well-known platforms in the board game industry: **BoardGameGeek (BGG)** and **BoardGameArena (BGA)**. These platforms provided the raw material for our project about games, users, reviews, forums, and tournaments.

Our objective was to construct a dataset capable of simulating real-world scenarios while maintaining consistency across all data points. For data that was unavailable, we supplemented the dataset by injecting carefully crafted artificial data to ensure completeness and realism.

We can decompose this overall process, made using *Jupyter notebooks* and *Python* in two distinct and serialized phases: *data retrieval* and *data processing*.

7.1 Data retrieval process

First step was to download data from the up cited platforms. This required different techniques and some precautions due to policies, the huge amount of data available and time constrains. The overall raw data volume scraped is around **950MB**.

7.1.1 BoardGameGeek

BoardGameGeek provides a structured XML API (`API2`), which allowed us to efficiently gather various types of data. This platform was our primary source for:

- **Games**. We selected the top *2,000 games* based on their popularity ranking. For each game, we retrieved its detailed metadata.
- **Reviews**. During game data retrieval, user reviews were also collected for the selected games, including ratings, comments, and the associated usernames. To maintain consistency, we limited the number of reviews to a subset of the most recent ones (*500 maximum*) for each downloaded game. For our specific purpose, we downloaded the reviews which contained a text comment.
- **Threads**. Each game has associated forums with threads and messages: we retrieved forum data by first fetching the list of forums for a game, followed by the threads within each forum, and finally the individual messages within the threads. Forum activity was included to capture user engagement and community interactions around specific games. Due to the huge amount of data available, we only selected a reduced number of forums, avoiding the retrieval of the ones with less number of messages, from the first *300 games* in the original ranking, resulting in *70 threads per game* and *40 messages per thread* on average.
- **Users**. During reviews and threads data retrieval, we collected the usernames we found. Then, we selected *5000 users* for which we download the available metadata, including their favorite games.

7.1.2 BoardGameArena

Unlike BGG, BGA lacks an official API, requiring us to use web scraping to collect data. Custom scripts were developed from scratch to extract relevant tournament information, accordingly to rate limits and platform policies. However, this resulted in a limited data retrieval, downloading metadata for approximately *200 games*, with *10 tournaments per game* on average.

7.2 Data processing

Data has been initially stored in `JSON` format, simplifying its successive management and manipulation. To achieve our goal, we had to enhance data quality with many techniques such as *locations cleaning*, *introducing data redundancy*, *thread extraction* from forums and *messages refactoring*, ratings correction and so on. In addition, we needed to provide missing data injecting generated metadata: time constrains, tournaments replication, user activity redistribution to the downloaded users and a **user network** creation based on user similarity in their activity (likes and post on threads).

7.3 Results

The final dataset volume, approximately **1.2GB** is provided by several JSON files, reflecting the entities and the specific database in which then data has been stored:

File	Size
`games.json`	5.4MB
`reviews.json`	672.8MB
`threads.json`	223.4MB
`tournaments.json`	61.5MB
`users.json`	14.3MB

Table 7.1: File sizes for MongoDB

File	Size
games.json	589kB
tournaments.json	7.4MB
users.json	1.2MB
likes.json	123.9MB
follows.json	31.2MB
partecipates.json	86.2MB
created.json	3.2MB
won.json	1.5MB
is_related_to.json	2.6MB

Table 7.2: File sizes for Neo4j

Chapter 8

Implementation

In this chapter, we will delve into the detailed implementation of the project using Spring Boot, a powerful and widely used framework for building robust and scalable applications. The aim is to provide a clear and structured explanation of the project's architecture, focusing on the organization of the codebase and the specific roles of each component.

The implementation is divided into several key packages, each designed to address a specific concern in the application's structure. This modular approach ensures clarity, maintainability, and scalability.

8.1 Spring Boot

In this project, Spring Boot plays a crucial role by streamlining the development process with its robust framework. It simplifies dependency management, reduces boilerplate code, and provides an embedded server for running the application. Additionally, Spring Boot seamlessly integrates with various tools and technologies, enabling the efficient creation of REST APIs and ensuring scalability and maintainability.

8.2 Security

In this system, we configure Spring Security to ensure robust application security by implementing a range of security features. The authentication system supports login, registration, and role-based authorization, leveraging JWT (JSON Web Token) for secure token-based authentication, Spring Security for core security management, and Spring Data

MongoDB for efficient data handling.

To achieve this, we use a carefully designed security configuration, such as the following method:

```
@Bean  Martina *
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
            .exceptionHandling( ExceptionHandlingConfigurer<HttpSecurity> exception -> exception.authenticationEntryPoint(unauthorizedHandler))
            .sessionManagement( SessionManagementConfigurer<HttpSecurity> session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
                    // Consenti l'accesso agli endpoint Swagger
                    .requestMatchers("/v3/api-docs/**", "/swagger-ui/**", "/swagger-ui.html").permitAll()
                    .requestMatchers("/api/admin/**").hasRole("ADMIN")
                    .requestMatchers("/api/users/**", "/api/suggestions/**").authenticated()
                    .anyRequest().permitAll()
            );

    http.authenticationProvider(authenticationProvider());
    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

This function plays a pivotal role in managing security configurations. It disables CSRF protection for stateless applications, sets up exception handling to manage unauthorized access, and configures session management to operate in a stateless mode. It also defines role-based access control, permitting unrestricted access to endpoints while restricting administrative endpoints to users with the "ADMIN" role.

Moreover, the function integrates an authentication provider and a JWT filter for secure authentication. This comprehensive setup ensures that the application is protected against unauthorized access while remaining user-friendly and maintaining seamless integration with essential tools like Swagger for API documentation. For declarative authorization management, the @PreAuthorize annotation is used. This is particularly valuable for endpoints that require diverse authorization rules, as it keeps the code clean, readable, and centralizes control over access permissions.

8.3 Config

This package contains essential configuration classes for the application. Specifically:

- **GlobalExceptionHandler**: This class centralizes the management of exceptions across the application. It ensures consistent error

handling and structured responses for various types of exceptions, improving maintainability and user experience.

- **SwaggerConfig**: This class configures Swagger to generate and manage API documentation for the application. It simplifies the process to explore and test the API, while seamlessly integrating security features like JWT authentication into the documentation. This ensures a clear understanding of secured endpoints and provides an interactive interface for testing the API with proper authentication.

8.4 Model

The model package defines the data structures used throughout the application. The system's models are divided into two types:

- **MongoDB**
- **Neo4j**

8.4.1 MongoDB models

8.4.1.1 User Model

```java
@Document(collection = "users")  Martina +1 *
@Data
public class UserMongo {
    @Id
    private String id;

    @NotBlank
    @Size(max = 20)
    @Indexed(unique = true)
    private String username;
    @NotBlank(message = "First name is required")
    private String firstName;
    @NotBlank(message = "Last name is required")
    private String lastName;
    @NotBlank
    @Size(max = 50)
    @Email(message = "Email should be valid")
    private String email;
    @NotBlank
    @Size(min = 8, max = 20)
    private String password;
    private Date registeredDate;
    @Past(message = "Birthdate must be in the past")
    private Date birthDate;
    @Schema(description = "UserMongo Location (Country, State, City)")
    private Location location;
    private int followers;
    private int following;
    private TournamentsUser tournaments;
    private Role role;     // Ruolo enum: "ROLE_USER" o "ROLE_ADMIN"
    private List<ReviewUser> mostRecentReviews=new ArrayList<>();
```

8.4.1.2 Game Model

```java
@Document(collection = "games") 29 usages  LeBonWskii +1 *
@Data
public class GameMongo {

    @Id
    private String id;

    @NotBlank(message = "Name cannot be blank")
    @Indexed
    private String name;
    @NotNull(message = "Year released cannot be null")
    private Integer yearReleased;
    private Date uploadTime;

    private String description;
    private String shortDescription;
    @Min(value = 0, message = "Average rating must be positive")
    private Double averageRating;
    private int ratingVoters;
    private int minPlayers;
    private int maxPlayers;
    private int minSuggAge;
    private int minPlayTime;
    private int maxPlayTime;
    private List<String> designers;
    private List<String> artists;
    private List<String> publisher;
    @NotNull(message = "Categories cannot be null")
    private List<String> categories;
    private List<String> mechanics;
    private List<String> family;
```

8.4.1.3 Post Model

```java
@Data  5 usages  3 inheritors  Martina
public abstract class Post {
    private String authorUsername;
    private Date postDate;
    private String content;
}
```

8.4.1.4 Review Model

```
@Document(collection = "reviews") 12 usages  Martina +1
@Data
public class ReviewMongo extends Post {

    @Id
    private String id;

    private Date authorBirthDate;
    private Location location;
    private GameReview game;

    @Min(1) @Max(value = 10, message = "Rating must be between 1 and 10")
    @NotNull
    private Double rating;
```

8.4.1.5 Thread Model

```
@Document(collection = "threads") 11 usages  Martina +2
@Data
public class ThreadMongo extends Post {
    private String id;
    private String tag;
    private GamePreviewEssential game;
    private Date lastPostDate;

    private List<Message> messages;
}
```

8.4.1.6 Tournament Model

```java
@Data 15 usages  Martina +1
@Document(collection = "tournaments")
public class TournamentMongo {
    @Id
    private String id;
    @NotBlank
    private String name;
    @NotBlank
    private GamePreviewEssential game;

    private String type;
    private String typeDescription;
    private Date startingTime;

    private Location location;

    @Positive
    private Integer numParticipants;
    @Positive
    private Integer minParticipants;
    @Positive
    private Integer maxParticipants;

    private String administrator;
    private String winner;

    private VisibilityTournament visibility;

    private List<OptionsTournament> options=new ArrayList<>();
    private List<String> allowed;
```

8.4.2 Neo4j models

8.4.2.1 User Model

```java
@Node("User")  9 usages  LeBonWskii
@Data
public class UserNeo4j {

    @Id
    @Property("username")
    private String username;
}
```

8.4.2.2 Game Model

```java
@Node("Game") 13 usages  LeBonWskii +1
@Data
public class GameNeo4j {
    @Id
    @Property("_id")
    private String id;

    @Property("name")
    @NotBlank
    private String name;

    @Property("yearReleased")
    @NotNull
    @PositiveOrZero
    private int yearReleased;

    @Property("shortDescription")
    @NotBlank
    private String shortDescription;

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

8.4.2.3 Tournament Model

```java
@Node("Tournament") 3 usages  LeBonWskii +1
@Data
public class TournamentNeo4j {
    @Id
    @Property("_id")
    private String id;

    @Property("name")
    @NotBlank
    private String name;

    @Property("visibility")
    @NotBlank
    private String visibility;

    @Property("maxParticipants")
    @NotNull
    @Positive
    private int maxParticipants;

    @Property("startingTime")
    @NotNull
    private Date startingTime;
}
```

8.5 Repository

The repository modules in our system play a critical role in facilitating interaction with both MongoDB and Neo4j databases. Leveraging Spring Boot's repository features, they provide a abstraction layer enabling efficient implementation of CRUD functionalities and simplifying data management.

For MongoDB and Neo4j, we utilize annotations like @Query for custom queries and @Aggregation for handling advanced operations such as analytics and recommendations. These features ensure flexibility in accessing and processing data while maintaining high performance.

8.6 Service

The service layer is the backbone of our application, bridging the gap between the controllers and repositories. It serves as the central hub for business logic, validating data, managing transactions, and ensuring a clear separation of responsibilities.

Service modules are responsible for applying business rules, verifying data integrity, and handling potential errors before interacting with the repository layer. They act as a safeguard, ensuring that only valid and compliant operations are executed on the database.

Beyond simply coordinating database interactions, the service layer processes and transforms data retrieved from repositories, making it ready for use by the controller or other parts of the application. By centralizing key functionalities within the service layer, the system achieves greater modularity, maintainability, and resilience, enabling consistent and efficient data management while adhering to business requirements.

8.6.1 Analytics Service

Module to provide the administrator with analytics functionality for generating statistics for better understanding of application performance, For example, features that allow users to be displayed across countries and track the monthly trend of user registrations.

8.6.2 Auth Service

Module that implements the registration and login of a user that through encryption of passwords and JWT tokens allows to increase the security of the application in these delicate phases. Also if a user registration is successful, it creates a new document in the User collection in MongDB and a new node in Neo4j.

8.6.3 Game Service

Manages all CRUD operations related to games, from creating a new game, to updating the information about it, deleting and searching for

games through numerous filters including name, release year and categories, returning lists of games with the support of pagination. In addition to this are implemented some queries such as obtaining the ranking of games based on rating, even in this case filterable for numerous parameters, and get the most played games.

8.6.4 Review Service

Allows you to manage all the CRUD operations related to reviews, such as creation of new reviews, updating of the review by the author, Deletion of a review by the administrator's author or obtaining reviews for a specific game or user.

8.6.5 Suggestion Service

It allows to generate important suggestions to improve the user experience on the platform by exploiting the connections of the graph database Neo4j. In particular, it allows a user to suggest other users whom he or she might know or have interests in communities or to suggest tournaments which the user might be interested in participating in.

8.6.6 Thread Service

Implements all thread management, starting from the creation, deletion, update and addition of messages or replies allowing users to interact exchanging advice and opinions. It also allows you to get threads related to a game or to get threads simply by filtering through other parameters such as the date of creation of the thread to allow users to always stay up-to-date with the latest discussions.

8.6.7 Tournament Service

Manages all the features related to tournaments both MongoDb side and NEo4j side, allowing you to perform CRUD operations related to tournaments that allow you to implement the competitive side of table games loved by users,

8.6.8 User Service

It realizes all those functionalities related to the users, such as the user's profile, view of a user's profile or personal profile allowing also to modify the information of the latter. In addition to the deletion of the profile it is also possible to obtain the references left by the user.

8.7 Controller

8.7.1 Admin Controller

Provides all the functionality needed by the admin, such as game management and moderation of user activity. It also allows the administrator to get important analytics on the application. This controller is connected with game, user and analytic services for a management of the key parts of the application.

8.7.2 Auth Controller

Manages the parts related to the authentication of a user by connecting to the auth service to ensure greater security of the login and registration phases.

8.7.3 Game Controller

Allows to manage all requests concerning the games by connecting with the game service to allow a correct return of what the user had requested.

8.7.4 Review Controller

Connects with the review service to provide the functionality of creating, obtaining, updating and deleting reviews.

8.7.5 Suggestion Controller

Provides useful suggestion functionality to users by leveraging the suggestion service.

8.7.6 Thread Controller

It allows to map all user requests regarding thread related features, such as creating, deleting or adding messages to populate the discussion. To do this, of course, uses the thread service.

8.7.7 Tournament Controller

Requests for creation, updating, deletion and obtaining, both for each game that a user, for tournaments are provided also to the connection with tournament service.

8.7.8 User Controller

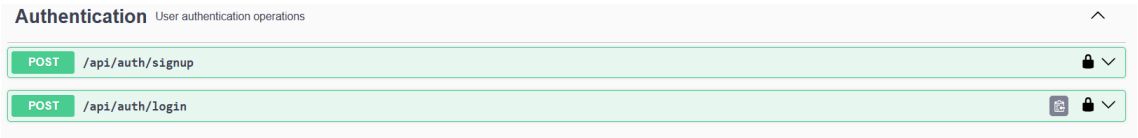
This controller is responsible for handling various user-related functionalities, such as searching for users, fetching user details, updating their profiles, and deleting accounts. It also manages user preferred games, allowing users to add or remove a like to a certain game. Furthermore, it oversees social interactions like following or unfollowing other users. The controller communicates with the User Service to ensure efficient processing of all requests.

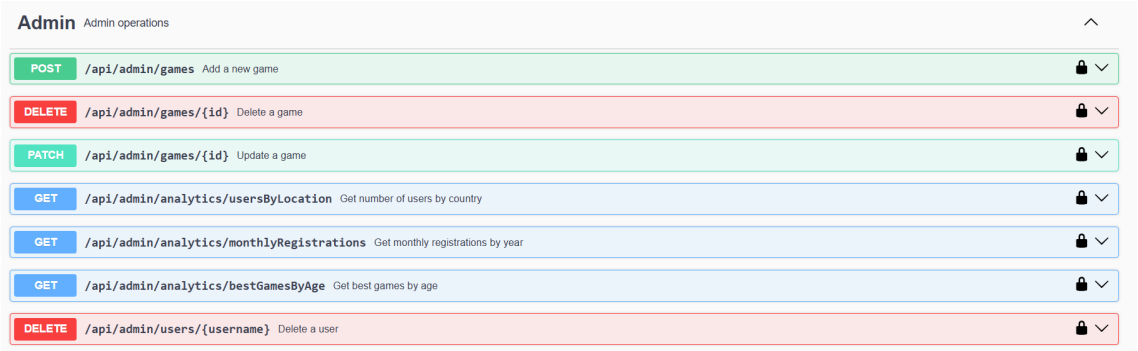
8.8 Exception

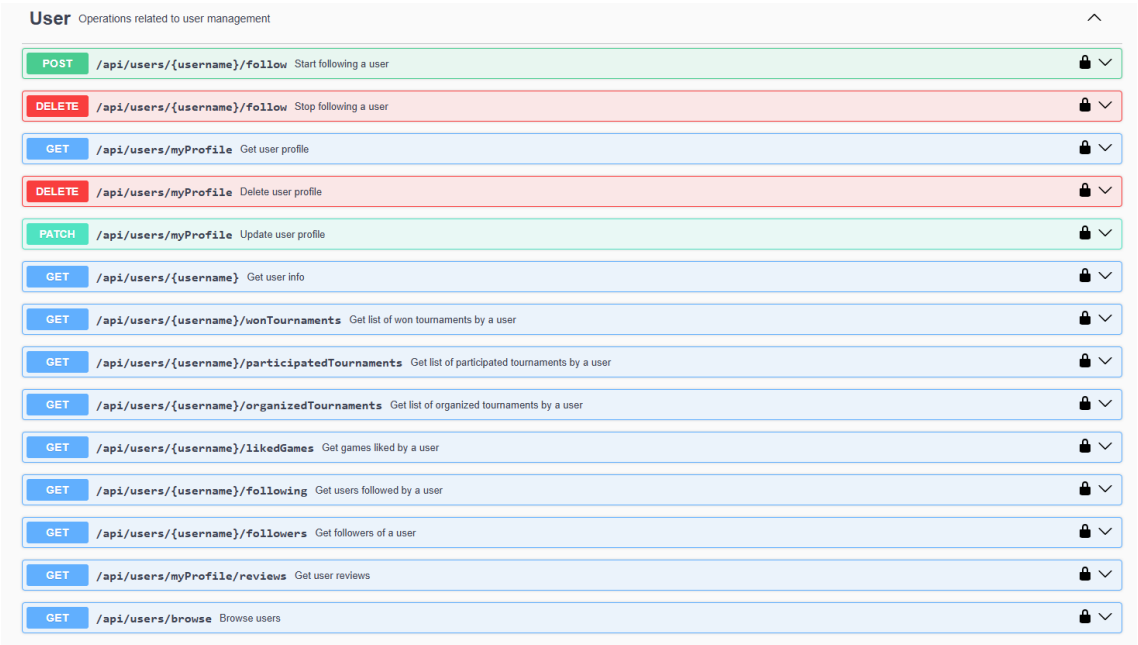
This package contains custom exception classes:

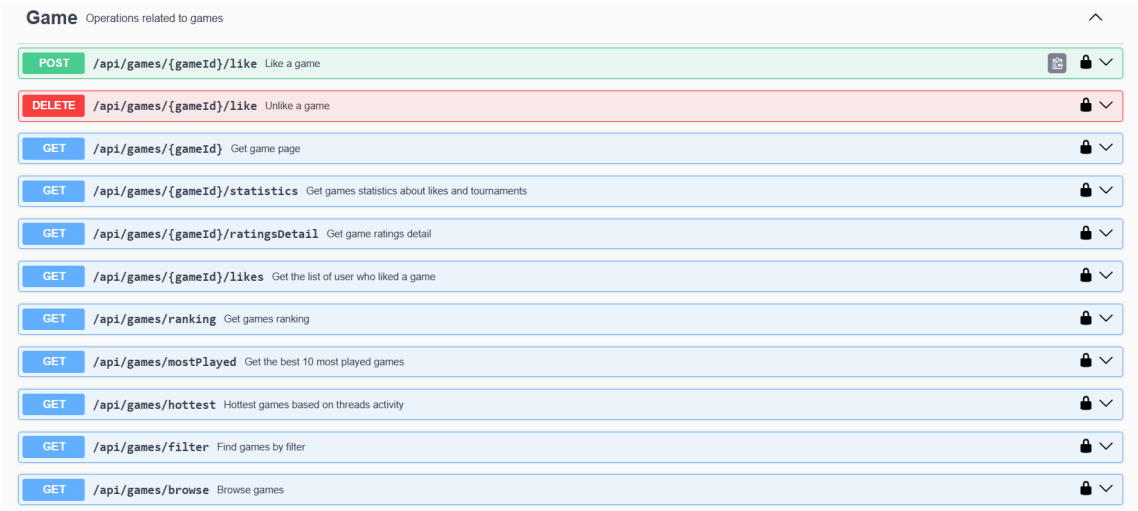
- **AlreadyExistsException:** This exception is likely thrown when an entity (e.g., a user or game) already exists in the system and a duplicate operation is attempted (e.g., trying to register a user with an existing username).
- **NotFoundException:** This exception is thrown when a requested entity or resource cannot be found (e.g., looking for a non-existent user or game).

8.9 Endpoints

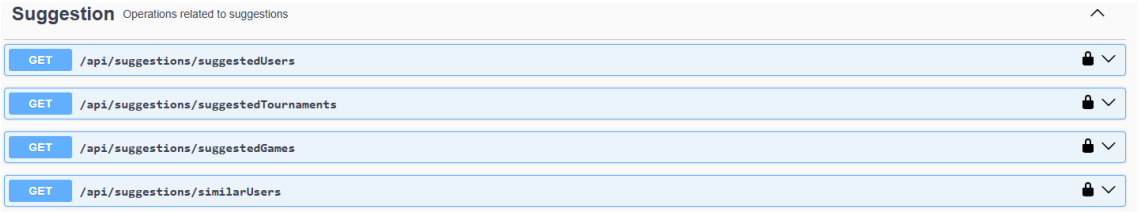


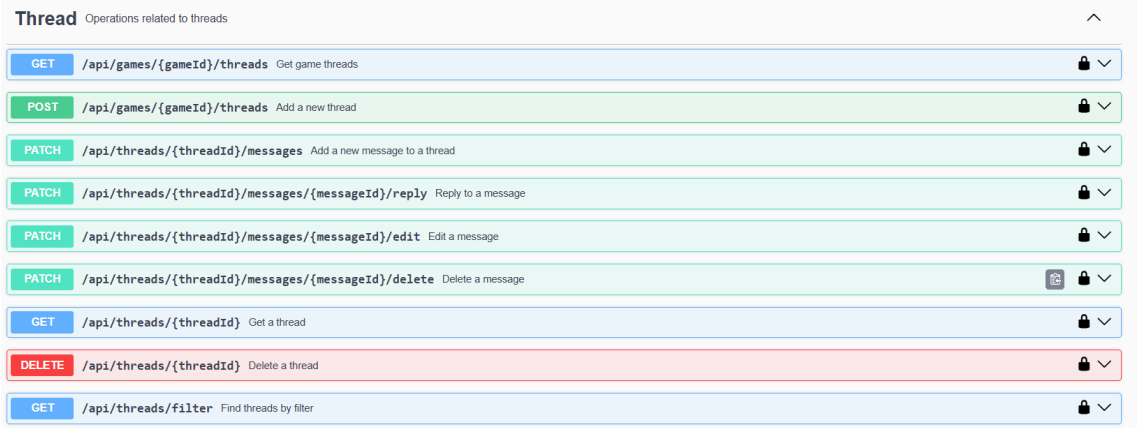


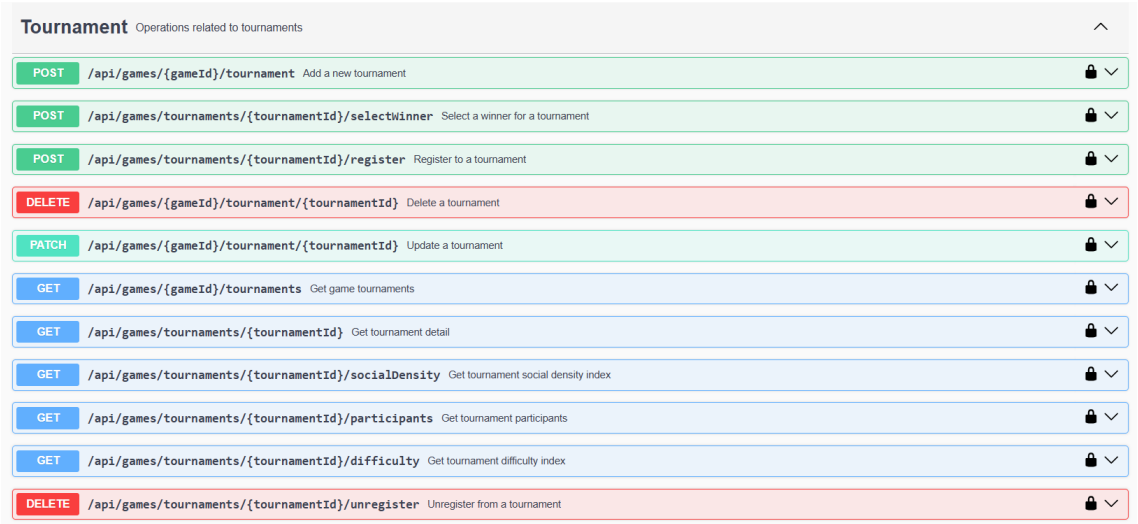



Game Operations related to games
POST /api/games/{gameId}/like Like a game
DELETE /api/games/{gameId}/like Unlike a game
GET /api/games/{gameId} Get game page
GET /api/games/{gameId}/statistics Get games statistics about likes and tournaments
GET /api/games/{gameId}/ratingsDetail Get game ratings detail
GET /api/games/{gameId}/likes Get the list of user who liked a game
GET /api/games/ranking Get games ranking
GET /api/games/mostPlayed Get the best 10 most played games
GET /api/games/hottest Hottest games based on threads activity
GET /api/games/filter Find games by filter
GET /api/games/browse Browse games


Review Operations related to reviews
GET /api/games/{gameId}/reviews Get game reviews
POST /api/games/{gameId}/reviews Add a new review
DELETE /api/games/{gameId}/reviews/{reviewId} Delete a review
PATCH /api/games/{gameId}/reviews/{reviewId} Update a review


Suggestion Operations related to suggestions
GET /api/suggestions/suggestedUsers
GET /api/suggestions/suggestedTournaments
GET /api/suggestions/suggestedGames
GET /api/suggestions/similarUsers


Thread Operations related to threads
GET /api/games/{gameId}/threads Get game threads
POST /api/games/{gameId}/threads Add a new thread
PATCH /api/threads/{threadId}/messages Add a new message to a thread
PATCH /api/threads/{threadId}/messages/{messageId}/reply Reply to a message
PATCH /api/threads/{threadId}/messages/{messageId}/edit Edit a message
PATCH /api/threads/{threadId}/messages/{messageId}/delete Delete a message
GET /api/threads/{threadId} Get a thread
DELETE /api/threads/{threadId} Delete a thread
GET /api/threads/filter Find threads by filter


Tournament
Operations related to tournaments
POST /api/games/{gameId}/tournament Add a new tournament
POST /api/games/tournaments/{tournamentId}/selectWinner Select a winner for a tournament
POST /api/games/tournaments/{tournamentId}/register Register to a tournament
DELETE /api/games/{gameId}/tournament/{tournamentId} Delete a tournament
PATCH /api/games/{gameId}/tournament/{tournamentId} Update a tournament
GET /api/games/{gameId}/tournaments Get game tournaments
GET /api/games/tournaments/{tournamentId} Get tournament detail
GET /api/games/tournaments/{tournamentId}/socialDensity Get tournament social density index
GET /api/games/tournaments/{tournamentId}/participants Get tournament participants
GET /api/games/tournaments/{tournamentId}/difficulty Get tournament difficulty index
DELETE /api/games/tournaments/{tournamentId}/unregister Unregister from a tournament

Chapter 9

Queries

9.1 MongoDB

In the following chapter, we will illustrate the queries implemented in MongoDB, providing detailed explanations of their structure and functionality.

9.1.1 User operations

Ranking of games based on reviews filtered by postDate and user Location

This query is useful for generating a ranked list of games based on user reviews within a specific time frame and location.
For example:

- Ranking games by average rating for users in a particular country or city.
- Generating a leaderboard of games based on reviews during a specific promotional period or event.

It provides insights into the most highly-rated games within a given context and time frame, helping identify popular games Each entry in the result represents a game and includes the following information: game Id, game Name, Short Description, Year Released and average rating.
The aggregation performs the following steps:

1. Match: Filters reviews to include only those within the specified date range (startDate to endDate). Additional location-based filters are applied dynamically, ensuring that optional parameters such as country, state, and city are handled flexibly. 2. Group: Groups reviews by the unique game identifier (gameId). For each group, the aggregation retrieves the game information and calculates the average rating of the reviews for that game. 3. Sort: Orders the grouped results in descending order of the calculated averageRating, ensuring the highest-rated games appear first.

```
@Aggregation(pipeline = {  1 usage  new *
        "{ '$match': { 'postDate': { '$gte': ?0, '$lte': ?1 } } }",
        "{ '$match': { '$and': [ "
                + "  { '$or': [ { '$expr': { '$eq': [:#{#country}, null] } }, { 'location.country': :#{#country} } ] }, "
                + "  { '$or': [ { '$expr': { '$eq': [:#{#state}, null] } }, { 'location.stateOrProvince': :#{#state} } ] }, "
                + "  { '$or': [ { '$expr': { '$eq': [:#{#city}, null] } }, { 'location.city': :#{#city} } ] } "
                + "] } }",
        "{ '$group': { "
                + "  '_id': '$game._id', 'name': { '$first': '$game.name' }, "
                + "  'yearReleased': { '$first': '$game.yearReleased' }, "
                + "  'shortDescription': { '$first': '$game.shortDescription' }, "
                + "  'averageRating': { '$avg': '$rating' } "
                + "} }",
        "{ '$sort': { 'averageRating': -1 } }",
        "{ '$project': { "
                + "  '_id': 1, 'name': 1, 'shortDescription': 1, 'yearReleased': 1, "
                + "  'averageRating': { '$round': [ '$averageRating', 2 ] } "
                + "} }"
})
Slice<GameRankPreviewDTO> findAverageRatingByPostDateLocation(
        Date startDate, Date endDate, String country, String state, String city, Pageable pageable);
```

Top 10 Most Played Games

This aggregation query calculates and retrieves the Top 10 games with the highest average weighted participation based on tournaments held within a specified time frame. The query incorporates weighted calculations based on the visibility of the tournaments (e.g., public, invite, or private). Here's a detailed explanation of its steps:

1. Match: Filters the tournaments to include only those that started within the specified date range and excludes tournaments with fewer than 10 participants.

2. AddFields: Adds two new fields for each tournament:

   Weight: Assigns a weighting factor based on the tournament's visibility (PUBLIC, INVITE, or PRIVATE).

WeightedParticipants: Calculates the weighted number of participants by multiplying numParticipants by the corresponding weight.

3. Group: Groups the tournaments by the unique identifier of the associated game.

4. AddFields: Calculates the average weighted participation for each game by dividing the total weighted participants by the total weight.

5. Sort: Orders the games in descending order of their average weighted participation, ensuring the most played games appear first.

6. Limit: keep only the top 10

```
@Aggregation(pipeline = {  1 usage   Martina
        "{ $match: { 'startingTime': { $gte: ?0, $lt: ?1 }, 'numParticipants': { $gt: 10 } } }",
        "{ $addFields: { 'weight': { '$switch': { 'branches': [ " +
                "{ 'case': { '$eq': ['$visibility', 'PUBLIC'] }, 'then': 2 }, " +
                "{ 'case': { '$eq': ['$visibility', 'INVITE'] }, 'then': 1.5 }, " +
                "{ 'case': { '$eq': ['$visibility', 'PRIVATE'] }, 'then': 1 } ], " +
                "'default': 1 } }, " +
                "'weightedParticipants': { '$multiply': ['$numParticipants', " +
                "{ '$switch': { 'branches': [ " +
                "{ 'case': { '$eq': ['$visibility', 'PUBLIC'] }, 'then': 2 }, " +
                "{ 'case': { '$eq': ['$visibility', 'INVITE'] }, 'then': 1.5 }, " +
                "{ 'case': { '$eq': ['$visibility', 'PRIVATE'] }, 'then': 1 } ], " +
                "'default': 1 } }] } } }",
        "{ $group: { '_id': '$game._id', 'name': { '$first': '$game.name' }, 'yearReleased': { '$first': '$game.yearReleased' }, " +
                "'totalWeightedParticipants': { '$sum': '$weightedParticipants' }, 'totalWeight': { '$sum': '$weight' } } }",
        "{ $addFields: { 'averageWeightedParticipants': { '$divide': ['$totalWeightedParticipants', '$totalWeight'] } } }",
        "{ $sort: { 'averageWeightedParticipants': -1 } }",
        "{ $limit: 10 }",
        "{ $project: { '_id': 0, 'gameID': '$_id', 'name': 1, 'yearReleased': 1, 'averageWeightedParticipants': 1 } }"
})
List<MostPlayedGameDTO> findTop10GamesWithHighestAverageParticipation(Date startDate, Date endDate);
```

Hottest games based on threads activity

This query is designed to rank games based on the importance of associated messages, taking into account both the number of messages and their time relevance within a specified period. It provides a normalized view of the game's activity or significance, potentially useful for identifying trending games or games with the most impactful discussions during a specific time frame. The aggregation performs the following steps:

1. addFields: the query first filters the messages array within each document, retaining only messages where the postDate falls within the specified date range.

2. addFields: next, it processes the filtered messages array using map. For each message:

   - A variable dateDif is calculated, representing the difference between the range limits.

   - The weight of each message is computed using the formula:

   0.1 + (1 * ((postDate - startDate) / (endDate - startDate)))

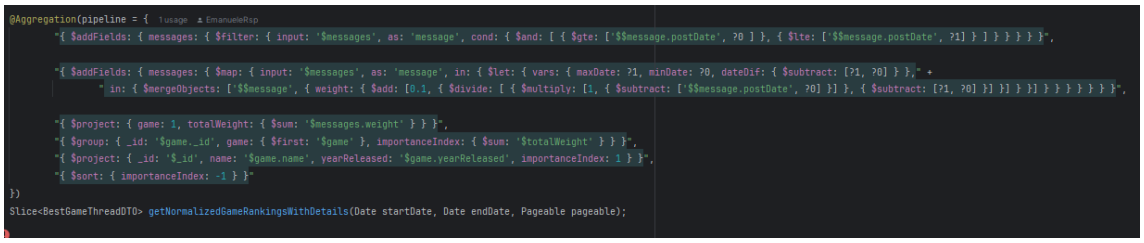
   which normalizes the weight based on the postDate relative to the reference range. The computed weight is then merged back into the original message object.

3. project: a new field, totalWeight, is added to each document. This field represents the sum of all calculated weights for the messages in the messages array.

4. group: the documents are grouped by gameId. For each group: the game details are retrieved and the totalWeight values for all messages associated with the game are summed to compute an importanceIndex.

5. project: in the final stage, only the required fields are selected

6. sort: the results are sorted in descending order of importanceIndex, ensuring that games with the highest importance appear first.



Game rating details

This query is designed to calculate and retrieve rating statistics for a specific game based on its gameId. It provides a detailed breakdown of average ratings, the standard deviation of ratings, and a distribution of ratings by rounded values. Here's an explanation of the query stages:

1. match: filters documents to include only ratings associated with the specified gameId, ensuring the rating field is not null.

2. project: retains the original rating and adds a roundedRating field, which rounds the rating to the nearest whole number for distribution analysis.

3. facet: Splits into two parallel computations:

   - Stats: Calculates the average rating (avgRating) and standard deviation (stdDeviation) across all ratings.

   - Distribution: Groups ratings by roundedRating, counts occurrences in each category, and sorts them in ascending order.

4. project: Combines the stats and distribution results into a single document, providing both a statistical summary and a detailed distribution of ratings for the game.

```
@Aggregation(pipeline = {  1 usage   Martina
        "{ '$match': { 'game._id': ?0, 'rating': { '$ne': null } } }",
        "{ '$project': { 'rating': 1, 'roundedRating': { '$round': ['$rating', 0] } } }",
        "{ '$facet': { " +
                "'stats': [ " +
                "{ '$group': { '_id': null, 'avgRating': { '$avg': '$rating' }, 'stdDeviation': { '$stdDevPop': '$rating' } } } " +
                "], " +
                "'distribution': [ " +
                "{ '$group': { '_id': '$roundedRating', 'count': { '$sum': 1 } } }, " +
                "{ '$sort': { '_id': 1 } }, " +
                "{ '$project': { 'rating': '$_id', 'count': 1, '_id': 0 } } " +
                "] " +
                "} }",
        "{ '$project': { " +
                "'avgRating': { '$arrayElemAt': ['$stats.avgRating', 0] }, " +
                "'stdDeviation': { '$arrayElemAt': ['$stats.stdDeviation', 0] }, " +
                "'distribution': '$distribution' " +
                "} }"
})
RatingDetails findRatingDetailsByGameId(String gameId);
```

9.1.2 Admin operations

Community geographical distribution

This query calculates the number of users grouped by country and state or province, organizing the results into a hierarchical structure with states nested under their respective countries. It is ideal for generating a location-based user distribution, providing insights into how users are geographically distributed by country and state/province. It can be

used for dashboards, reporting, or analytics to understand regional user engagement. The aggregation performs the following steps:

1. match: Filters the documents to include only users with valid location information.

2. group: groups the users by country and stateOrProvince, calculating the total number of users for each state or province.

3. group: regroups the results by country, creating a hierarchical structure where each country contains an array of its states along with their respective user counts.

4. sort: orders the results alphabetically by country for better readability and presentation.

5. project: formats the final output to include: country and states, an array of states within the country, each showing the user count.

```
@Aggregation(pipeline = {  1 usage   Martina
        "{ '$match': { 'location.country': { '$ne': null }, 'location.stateOrProvince': { '$ne': null }} }",
        "{ '$group': { '_id': { 'country': '$location.country', 'state': '$location.stateOrProvince'}, 'count': { '$sum': 1 } } }",
        "{ '$group': { '_id': '$_id.country', 'states': { '$push': { 'state': '$_id.state', 'count': '$count' } } } }",
        "{ '$sort': { '_id': 1 } }",  // Ordinamento alfabetico su country
        "{ '$project': { '_id': 0, 'country': '$_id', 'states': 1 } }"
})
Slice<CountryAggregation> countUsersByLocationHierarchy(Pageable pageable);
```

Monthly user registrations by Year

This query is designed for the admin and provides the monthly user registrations for a selected year. It aggregates the number of users registered per month within the specified time range, allowing the admin to analyze registration trends for the chosen year. This query is specifically tailored for admin users to:

- Visualize monthly registration trends for a given year.
- Identify months with high or low registration activity.
- Analyze the effectiveness of campaigns or events influencing user registrations during specific periods.

Here's how it works:

1. match: filters the user documents to include only those whose registeredDate falls within the specified year.
2. group: Groups the documents by month and calculates the total number of user registrations for each month.
3. sort: Orders the results in ascending order based on the month number.

The query produces a list of objects, where each object contains:

- Month: The numeric representation of the month (e.g., 1 for January, 2 for February) .
- Registrations: The total number of user registrations for that specific month.

```
@Aggregation(pipeline = {  1 usage   Martina
        "{ '$match': { 'registeredDate': { '$gte': ?0, '$lt': ?1 } } }",
        "{ '$group': { '_id': { '$month': '$registeredDate'  }, 'registrations': { '$sum': 1 } } }",
        "{ '$sort': { '_id': 1 } }"
})
List<MonthlyReg> monthlyRegistrations(Instant startDate, Instant endDate);
```

Community tastes per age bucket

This query is designed to determine the best-rated games grouped by age brackets of users who provided ratings. It calculates the average rating for each game within defined age groups and selects the game with the highest average rating for each bracket. It helps developers and publishers tailor their content, marketing strategies, or updates to better suit specific audience segments. Below is an explanation of how the query works:

The query filters reviews to include only those with a non-null rating, ensuring that only valid data is processed. It then calculates a new field, age, by determining the user's age as the difference in years between their birth date (authorBirthDate) and the current date. Based on this, it creates an additional field, ageBracket, grouping users into 10-year age ranges. Reviews are then grouped by both age bracket and game ID,

and games with fewer than 30 reviews are excluded to ensure statistical significance. The results are sorted first by age bracket and then by average rating in descending order. For each age bracket, the game with the highest average rating is selected. Finally, the age brackets are sorted in ascending order to produce the final output.

```
@Aggregation(pipeline = {  1 usage   Martina
        "{ $match: { rating: { $ne: null } } }",
        "{ $addFields: { age: { $dateDiff: { startDate: \"$authorBirthDate\", endDate: \"$$NOW\", unit: \"year\" } } } }",
        "{ $match: { age: { $gte: 10, $lte: 99 } } }",
        "{ $addFields: { ageBracket: { $concat: [ " +
                "{ $toString: { $multiply: [10, { $floor: { $divide: [\"$age\", 10] } }] } }, \"-\", " +
                "{ $toString: { $add: [ { $multiply: [10, { $floor: { $divide: [\"$age\", 10] } }] }, 9 ] } } ] } } }",
        "{ $group: { " +
                "    _id: { ageBracket: \"$ageBracket\", game: \"$game._id\" }, " +
                "    name: { $first: \"$game.name\" }, " +
                "    yearReleased: { $first: \"$game.yearReleased\" }, " +
                "    averageRating: { $avg: \"$rating\" }, " +
                "    totalReviews: { $sum: 1 } " +
                "} }",
        "{ $match: { totalReviews: { $gte: 30 } } }",
        "{ $sort: { \"_id.ageBracket\": 1, \"averageRating\": -1 } }",
        "{ $group: { " +
                "    _id: \"$_id.ageBracket\", " +
                "    gameID: { $first: \"$_id.game\" }, " +
                "    name: { $first: \"$name\" }, " +
                "    yearReleased: { $first: \"$yearReleased\" }, " +
                "    bestAvgRating: { $first: \"$averageRating\" }, " +
                "    totalReviews: { $first: \"$totalReviews\" } " +
                "} }",
        "{ $sort: { \"_id\": 1 } }"
})
List<BestGameAgeDTO> findBestGameByAgeBrackets();
```

9.2 Neo4j

9.2.1 Suggest new friends

Domain-specific: Suggest new friends to user A, suggesting users who have more friends in common with user A but are not friends with user A

Graph-centric: Considering the current User vertex (User A), identify all the User vertices directly connected to User A through an outgoing "FOLLOWS" edge. Then, for each of these connected User vertices, find all other User vertices connected to them through an outgoing "FOLLOWS" edge. Count the occurrences of these User vertices to determine the number of mutual friends they share with User A. Exclude any User vertex that is already directly connected to User A via a "FOLLOWS"

edge or is User A itself. Sort the remaining User vertices by the number of mutual friends in descending order and return them as potential new friends.

```
@Query("""
    MATCH (currentUser:User {username: $username})-[:FOLLOWS]->(followedUser:User)-[:FOLLOWS]->(suggestedUser:User)
    WHERE NOT (currentUser)-[:FOLLOWS]->(suggestedUser) AND suggestedUser.username <> $username
    RETURN suggestedUser.username AS username, COUNT(suggestedUser) AS commonFollowers
    ORDER BY commonFollowers DESC
    SKIP $pageSize * ($pageNumber - 1)
    LIMIT $pageSize
""")
List<UserFollowRecommendationDTO> getUsersRecommendationBySimilarNetwork(String username, int pageSize, int pageNumber);
```

9.2.2 Suggest users with similar tastes

Domain-specific: Suggest users with similar tastes to user A, suggesting users who have more liked games in common with user A but are not friends with user A

Graph-centric: Considering the current User vertex (User A), identify all the User vertices directly connected to User A through an outgoing "FOLLOWS" edge. Then, for each of these connected User vertices, find all other User vertices connected to them through an outgoing "FOLLOWS" edge. Locate all the games liked by these users. Then locate all games connected with a "LIKES" relationship with user A. Count the games liked in common between user A and other users. Return the list of users who have at least 1 liked game in common with user A in descending order of liked games in common

```
@Query("""
        MATCH (user:User {username: $username})-[:FOLLOWS]->(:User)-[:FOLLOWS]->(suggestedUser:User)
        WHERE NOT (user)-[:FOLLOWS]->(suggestedUser) AND suggestedUser <> user
        MATCH (user)-[:LIKES]->(commonGame:Game)<-[:LIKES]-(suggestedUser)
        WITH suggestedUser, COUNT(commonGame) AS sharedGames
        WHERE sharedGames > 0
        RETURN
          suggestedUser.username AS username,
          sharedGames
        ORDER BY sharedGames DESC
        SKIP $pageSize * ($pageNumber - 1)
        LIMIT $pageSize
""")
List<UserTastesSuggestionDTO> getUsersRecommendationBySimilarTastes(@Param("username") String username, @Param("pageSize") int pageSize, @Param("pageNumber") int pageNumber);
```

9.2.3 Board games recommendation

Domain-specific: Board games recommendation based on the user's favorite board games and categories

Graph-centric: Starting from the current user node A, all other user nodes connected to the A node are identified through a "FOLLOWS" relationship. Between these nodes identify all games connected through

a "LIKES" relationship that are not connected with a"LIKES" relationship with the user node A. Then find all the game nodes connected to the A node through a "LIKES" relationship. Extract all categories of potentially suggestible games and games liked by the user. Compare and count the number of categories in common. Return games with at least 1 common category in descending order of the number of common categories.

```
@Query("""
    MATCH (u:User {username: $username})<-[:FOLLOWS]-(follower:User)-[:LIKES]->(g:Game)
        WHERE NOT (u)-[:LIKES]->(g)
        WITH DISTINCT g, u
        // Compute common categories
        MATCH (u)-[:LIKES]->(likedGame:Game)
        UNWIND likedGame.categories AS userCategory
        UNWIND g.categories AS gameCategory
        WITH g, userCategory, gameCategory
        WHERE userCategory = gameCategory
        WITH g, COUNT(userCategory) AS commonCategories
        // Order and return results
        WHERE commonCategories > 0
        RETURN
          g.name AS name,
          g._id AS id,
          g.yearReleased AS yearReleased,
          g.shortDescription AS shortDescription,
          commonCategories
        ORDER BY commonCategories DESC
    SKIP $pageSize * ($pageNumber - 1)
    LIMIT $pageSize
""")
List<GameSuggestionDTO> getGamesRecommendation(String username, int pageSize, int pageNumber);
```

9.2.4 Tournaments recommendation

Domain-specific: Tournament recommendation based on the favorite games of the user and tournaments that the user's friends are participating in

Graph-centric: Starting from the user A vertex, search for all tournament nodes linked via a "PARTICIPATES" relationship and find all games linked to these tournaments via a "IS _RELATED_TO" relationship. All tournament summits linked to previously found games that have public visibility, are not yet started and are not full where user A does not participate already. Count the user vertices connected to user A through a "FOLLOWS" relationship that also have a "PARTICI-

PATES" relationship with the tournaments found. Return tournaments in descending order of the number of followers participating, and in case of tie give priority to tournaments that start earlier.

```
@Query("""
    MATCH (user:User {username: $username})-[:PARTICIPATES]->(userTournament:Tournament)-[:IS_RELATED_TO]->(game:Game)
    MATCH (game)<-[:IS_RELATED_TO]-(suggestedTournament:Tournament)
    WHERE NOT (user)-[:PARTICIPATES]->(suggestedTournament)
      AND suggestedTournament.startingTime > datetime()
      AND suggestedTournament.visibility = 'PUBLIC'
      AND NOT EXISTS {
        MATCH (suggestedTournament)<-[:PARTICIPATES]-(:User)
        WITH COUNT(*) AS participantCount
        WHERE participantCount >= suggestedTournament.maxParticipants
      }
    MATCH (user)-[:FOLLOWS]->(follower:User)-[:PARTICIPATES]->(suggestedTournament)
    WITH suggestedTournament, COUNT(DISTINCT follower) AS numParticipantFollowers
    MATCH (suggestedTournament)<-[:PARTICIPATES]-(participant:User)
    WITH suggestedTournament, numParticipantFollowers, COUNT(DISTINCT participant) AS numParticipants
    MATCH (suggestedTournament)-[:IS_RELATED_TO]->(relatedGame:Game)
    WITH suggestedTournament, numParticipantFollowers, numParticipants,
         relatedGame._id AS gameId,
         relatedGame.name AS gameName
    RETURN DISTINCT
      suggestedTournament._id AS id,
      suggestedTournament.name AS name,
      suggestedTournament.startingTime AS startingTime,
      suggestedTournament.maxParticipants AS maxParticipants,
      numParticipants,
      numParticipantFollowers,
      { id: gameId, name: gameName } AS game
    ORDER BY numParticipantFollowers DESC, suggestedTournament.startingTime ASC
    SKIP $pageSize * ($pageNumber - 1)
    LIMIT $pageSize
""")
List<TournamentSuggestionDTO> getTournamentsRecommendation(String username, int pageSize, int pageNumber);
```

9.3 Indexes

9.3.1 MongoDB indexes

This section explores potential indexes that could enhance the performance of read operations for each collection. Subsequently, we will evaluate each suggested index to determine whether its addition is justified.

Users Collection

- *username*: Adding an index on the username field is highly efficient because it optimizes the performance of operations like login, signup, and browsing users, which are among the most frequent actions performed on the database. Given the high frequency of these operations, creating an index on username provides a substantial

improvement in query performance, ensuring the system remains responsive and scalable as the database grows. The table illustrates the performance improvement achieved by using an index.

	Without index	**With index**
Returned	1	1
Execution Time (ms)	36	1
Keys examined	0	1
Documents examined	5000	1

Games Collection

- *name*: An index has been added to the name field in the Games collection to significantly enhance the performance of the browse game operation, which is the primary activity of the application. This operation frequently involves searching for games by their names, filtering, or sorting them based on user input. With the index in place, the database can efficiently locate and retrieve games without performing a full collection scan. This not only reduces query execution time but also ensures scalability as the number of games increases. By prioritizing this optimization, the application can deliver faster response times and a smoother user experience during its most frequently used feature.

	Without index	**With index**
Returned	1	1
Execution Time (ms)	1	0
Keys examined	0	1
Documents examined	2000	1

Reviews Collection

- *game.id*: Given the high frequency of queries that load reviews for a specific game or perform searches using *game.id*, we decided to add an index on the *game.id* field in the Reviews collection. This

index significantly enhances query performance by enabling faster and more efficient data retrieval. While maintaining the index adds some overhead during inserts, the benefits in read performance far outweigh the cost, especially in a read-heavy system. This ensures faster queries, better scalability, and an overall improved user experience. As demonstrated in the table below, adding the index significantly improves the performance of the Find reviews by *game.id* query, making it much more efficient and faster.

	Without index	**With index**
Returned	361	361
Execution Time (ms)	3321	6
Keys examined	0	361
Documents examined	674945	361

Threads Collection and Tournament Collection

- *game.id*: As with the Reviews collection, we have also decided to add an index on *game.id* in the Threads and Tournaments collections for the same reasons mentioned earlier. This index significantly enhances query performance by enabling efficient retrieval of threads and tournaments related to specific games, avoiding costly full collection scans. While maintaining the index incurs some overhead during inserts or updates, the benefits in terms of faster read operations, improved scalability, and a smoother user experience far outweigh the costs.

	Without index	**With index**
Returned	50	50
Execution Time (ms)	11	0
Keys examined	0	50
Documents examined	9666	50

Table 9.1: Performance for Thread collection

	Without index	With index
Returned	20	20
Execution Time (ms)	449	2
Keys examined	0	20
Documents examined	20006	20

Table 9.2: Performance for Tournament collection

9.3.2 Neo4j indexes

User entity

- *username*: In the User entity on Neo4j, we deemed it necessary to add an index on the username field since all queries rely on the findByUsername operation. This index ensures efficient and fast retrieval of user data, significantly improving query performance and supporting the high frequency of operations involving username searches.

Game entity and Tournament entity

- *id*: In the Tournaments and Games entities, we added indexes on the id field because all Neo4j queries that start computations from a specific node use the id property to locate it. These indexes ensure efficient and fast access to nodes, significantly improving query performance and scalability.