



Università di Pisa
MSc in Computer Engineering
MSc in Artificial Intelligence and Data Engineering

Pixel Index

Final report for the Large Scale and Multi-Structured Databases' project

GitHub repository:

<https://github.com/lorenzoceccanti/ProjectLSMSDCMC>

Authors:

Ceccanti Lorenzo

Corsi Cristiano

Mulé Niccolò

Table of Contents

1. Introduction	5
1.1 Glossary	5
2. Dataset and Web Scraping	6
2.1 Raw Games Data.....	6
2.2 Reviews Scraping.....	6
2.3 User data generation	6
2.4 Final dataset	6
3. Application requirements	7
3.1 Functional requirements - Unregistered user:.....	7
3.2 Functional requirements - Registered user:	8
3.3 Functional Requirements - Moderator:	10
3.4 Non-functional Requirements	10
4. UML use case diagram	11
5. UML Class Diagram.....	12
6. Data modelling	13
6.1 Document database	13
6.2 Graph database	16
6.3 Discussion on the choices made during data modelling	17
7. Application architecture and implementation.....	18
7.1 Packages	18
7.2 Main classes.....	18
8. Statistics and queries	21
8.1 Mongo DB	21
8.1.1 Find Top Reviewers by Reviews Count.....	21
8.1.2 Get Reviews by Game ID	22
8.1.3 Get Games Advanced Search	25
8.1.4 Top Games by Positive Rating Ratio	27
8.1.5 Number of Registrations by Month	29
8.2 Neo4j.....	32
8.2.1 Suggesting new user to follow based on mutual games	33
8.2.2 Suggesting new games based on follower's libraries.....	33
8.2.3 Retrieving the most trending games based on number of adds to library in a specific year	34
9. Distributed databases design	36

9.1	Considerations on the CAP theorem	36
9.2	Consistency	36
9.2.1	Consistency Thread	36
9.2.2	Eventual consistency operations and redundancy updates.....	38
9.2.3	Strict consistency operations	39
9.2.4	MongoDB write and read concern	40
10.	Sharding organization	40
11.	Indexes and performance	41
11.1	Mongo DB	42
11.2	Neo4j.....	43
12.	Application user manual	45
12.1	Starting the application	45
12.2	Unregistered user manual	46
12.2.1	Searching and browsing games	46
12.2.2	Viewing game details.....	50
12.2.3	Browsing and viewing reviews excerpts	51
12.2.4	Viewing review details	52
12.2.5	Displaying the most active reviewers	53
12.2.6	Displaying the top ten rated games.....	54
12.2.7	Displaying the trending games chart.....	55
12.2.8	Login	55
12.2.9	Registration.....	56
12.2.10	Exiting the application	56
12.3	Registered user manual.....	56
12.3.1	Searching and browsing users	57
12.3.2	Editing the set of the followed users.....	58
12.3.3	Viewing library.....	58
12.3.4	Adding a game to library	59
12.3.5	Adding a game to wishlist	59
12.3.6	Removing a game from library	61
12.3.7	Removing a game from wishlist	61
12.3.8	Edit like preferences on a review	62
12.3.9	Adding a new review	62
12.3.10	Removing a review	63

12.3.11	Viewing suggested users and suggested games	63
12.4	Moderator manual.....	64
12.4.1	View reports and ban a user	64
12.4.2	Check registration stats	64

1. Introduction

PixelIndex is a video game focused application crafted with the following goals in mind:

- Helping a gamer to track the games he/she has played: each user has a personal library, which he/she can use as a registry of the game he/she owns.
- Offering the possibility to each gamer to have a personal wishlist in which he/she can put the games that he/she does not own yet but that arouse interest on the user.
- Favoring relationships among gamers: users can follow each other, can agree or disagree on other user's reviews by liking or disliking a review.
- Offering gamers the possibility to share pasts experiences by leaving reviews about the games they have played.
- Security in terms of harassment among users, banned words and fake reviews are guaranteed by the work carried out by the moderators of the platform. Each user can provide its contribution by reporting other suspicious users.
- Additionally, the platform ensures child safety by restricting access to games deemed too mature for their age through age-appropriate filtering based on their date of birth and adherence to PEGI game classifications.

Our application will also include some analytics on games and user interactions:

- Trending games: this feature will allow users to retrieve the games ranked by how many users have added the game to their library. This will give an insight of which games are popular within the gaming community.
- Most active reviewers: users will be able to retrieve a leaderboard of the most active users determined by the number of reviews posted.
- Top games of all time: users can discover the all-time favorite games through a ranking that compares the ratio of positive to negative reviews. Games with higher positive feedback will be ranked first.
- New user insights: in their dedicated area, moderators will be able to receive some metrics about the number of new users joining the platform, broken down by age group and time frame.

1.1 Glossary

Before delving into the details of our project, it's crucial to define some key terms for the sake of clarity:

1. **Library:** this term is used for the set of games that the user claims to own. This list is meant to be persistent over time.
2. **Wishlist:** the wishlist is a list of games that the user has interest to own in the future. This list is maintained persistently over time to keep track of the games that are yet to be acquired by the user because they are, for example, awaiting release.
3. **Followers:** the set of users that a certain user is followed by.
4. **Followings:** the users that a certain user follows.
5. **Review excerpt:** this is a brief snapshot of a larger review text, consisting in the first 50 characters to provide a quick overview of the user's opinion.
6. **Reaction:** whenever we use this term, we refer to the action that corresponds to a user liking or disliking another user's review

2. Dataset and Web Scraping

In order to build our application, we gathered data from two main sources:

- IGDB.com: an online videogames database
- Steam: a video game digital distribution platform and storefront

For data that couldn't be scraped from the web, like user details, we used Python libraries to generate it randomly.

2.1 Raw Games Data

Our first step in creating our dataset was to collect game data, which we did leveraging IGDB's developer API to gather and locally store all the relevant information. Given that IGDB operates on a relational model, multiple steps were necessary to restructure the dataset into a non-relational one, which involved querying different endpoints to resolve the actual values starting from foreign keys. Furthermore, additional filtering steps were performed to keep only the essential data and to store it in the desired format. Finally, IGDB's API also allowed us to retrieve the Steam ID¹ of each game. This was crucial, as it made the reviews retrieval process much easier.

2.2 Reviews Scraping

IGDB's dataset provided all the necessary game data we needed to populate our games collection. It didn't, however, contain data about user reviews. In order to accomplish this, we performed web scraping on Steam's storefront. For each game in the dataset, we queried the Steam's frontend page using the Steam ID contained in the dataset and parsed reviews using BeautifulSoup, a Python library for HTML and XML parsing. This allowed us to gather thousands of legitimate reviews from actual users. Additionally, from each review, we managed to extract the username of the author to give our dataset realistic user profiles, along with a boolean value indicating whether the review was positive (Recommended) or negative (Not Recommended), offering a brief user's opinion on the game.

2.3 User data generation

To enhance our user data beyond simple usernames, we needed additional details like personal information, social connections and interactions with other users' reviews. To generate user data, we used Faker – a Python library that allowed us to create realistic user data randomly. For user interactions, we implemented a simple random matching system. Essentially, we paired users with other users and assigned interactions with reviews in a randomized manner.

2.4 Final dataset

The final dataset consists in:

- Over 250.000 games.
- More than 295.000 users and roughly 570.000 reviews.

¹ Not all games indexed on IGDB are available on Steam's storefront.

- 8 million “User Follows User” relationships, 3.7 million “User Likes Review” relationships and 1.5 million “User Adds To Library Game” relationships.

3. Application requirements

The three main actors that have been considered in this application are:

- **Unregistered user:** a guest user that still not have a personal account.
- **Registered user:** the user registered to the platform which has provided his/her details in the registration phase: username, name, surname, email address, date of birth and his/her personal password.
- **Moderator:** the moderator is an enhanced role for a user which manages the platform and checks if everything is good. The moderator has all the function of a registered user and other special buttons in a dedicated section to remove other users’ review, to view the reports appended on other users, to ban a user, to add new games and to see the registration trends in a specific year.

3.1 Functional requirements- Unregistered user:

- An unregistered user can login to the platform via username and password
- An unregistered user can have the possibility to register to the platform providing his/her username, name, surname, date of birth, email address and personal password.
- An unregistered user can search for a game specifying a portion of the title.
- An unregistered user can perform an advanced search to search the games specifying the name, the company of a game, the platform of a game (PC, PlayStation, Xbox, ...), and the year in which the game has been released. The advanced search can be made specifying all this options or a subset of them.
- An unregistered user, after issuing the game search, can browse among the list of the games satisfying the provided query. The games are displayed on pages: each page displays 10 games.
The result of the search prompts the first page displaying only the game title and the release year for each game.
- An unregistered user, after viewing the first page, can press on «Previous page» or «Next page» to scroll the pages.
- An unregistered user can press on the specific game to view the game details. Such details will be displayed only on games without the PEGI age rating classification.
- An unregistered user, after viewing the game details, can see the excerpt of the first top 10 most relevant reviews (along with the author, the number of likes and the number of dislikes associated to the review) by pressing the button «Show top 10 most relevant reviews». The sorting criteria projects first the reviews with higher number of likes. The review’s excerpt will be displayed in pages of ten elements and the user can press «Previous page» or «Next page» to scroll the pages.
- An unregistered user, after browsing the reviews, can select a specific review by pressing on it. The complete text of the review, the date in which the review was posted, the author, the number of likes and dislikes will be displayed.
- An unregistered user can see the top ten reviewers of last month by pressing the button «Most active reviewers».

- An unregistered user can see the top ten rated games in terms of rating ratio² by pressing the button «Top rated games».
- An unregistered user can see, after specifying a year, the trending games chart of the selected year (in terms of how many users had added the game to their library) by pressing the button «Trending games chart».

3.2 Functional requirements - Registered user:

- A registered user can search for a game specifying a portion of the title.
- A registered user can perform an advanced search to search the games specifying the name, the company of a game, the platform of a game (PC, PlayStation, Xbox,..), and the year in which the game has been released. The advanced search can be made specifying all this options or a subset of them.
- A registered user, after issuing the game search, can browse among the list of the games satisfying the provided query. The games are displayed on pages: each page displays 10 games.
The result of the search prompts the first page displaying only the game title and the release year for each game.
- A registered user, after viewing the first page, can press on «Previous page» or «Next page» to scroll the pages.
- A registered user, after viewing the first page, can press on the specific game to view the game details. The users that do not match the age limit cannot act on the age registered games: they can't see neither the game details, nor make any operation on such restricted games.
- A registered user, after viewing the game details, can see the excerpt of the first top 10 most relevant reviews (along with the author, the number of likes and the number of dislikes associated to the review) by pressing the button «Show top 10 most relevant reviews». The sorting criteria projects first the reviews written by the user who's pressing the button (if there are any) and then the reviews with higher number of likes. The review's excerpt will be displayed in pages of ten elements and the user can press «Previous page» or «Next page» to scroll the pages.
- A registered user, after viewing the game details, can press the button «Add this game to library» to add the game in its personal library. If the game is already in library, this action will not have effect.
- A registered user, after viewing the game details, can press the button «Remove this game to library» to remove the game from the user's personal library. If the game is not in the library of the user, this action will not have effect.
- A registered user, after viewing the game details, can press the button «Add this game to wishlist» to add the game in its personal wishlist. If the game is already in wishlist, this action will not have effect.
- A registered user, after viewing the game details, can press the button «Remove this game to wishlist» to remove the game from the user's personal wishlist. If the game is not in the wishlist of the user, this action will not have effect.
- A registered user, after viewing the game details, can press the button «Add review» in order to add a new review to the game displayed within the page.

² The rating ratio is computed as:
$$\frac{\#reviews_{RECOMMENDED}}{\#reviews_{RECOMMENDED} + \#reviews_{NOT_RECOMMENDED}}$$

- A registered user, after browsing the reviews excerpts, can select a specific review by pressing on it. The complete text of the review, the date in which the review was posted, the author, the number of likes and dislikes will be displayed.
- A registered user, after selecting a specific review, can press on three buttons: «Add like», «Add dislike», «Remove review» in order to add a like, remove a like and remove a review respectively. Only the user that has wrote the review before can succeed in removing the displayed review.
- A registered user, after pressing the button «Search user» can search for another registered user (by username). The result of the search is a page of ten rows; in each row, it's possible to see the username, the number of user's followers and the number of users followed by the user shown.
- A registered user, after performing a query to search by username, can browse through the pages by pressing the buttons «Previous page» or «Next page».
- A registered user, after performing a query to search by username, can select a specific user and follow him/her.
- A registered user, after performing a query to search by username, can select a specific user and unfollow him/her.
- A registered user, after performing a query to search by username, can select a specific user and report him/her. If the user is already reported, the report action will be confirmed.
- A registered user can browse his/her personal game library by pressing the «View your library» button placed in the main menu. The library will display first a page with the last ten games added. In the library page only the game name, the year of release and the date in which the user added the game to the library are shown. Then, the registered user, if he/she wants, can browse the other pages by pressing the buttons «Previous page» or «Next page».
- A registered user, once he/she lands in his/her personal library page, can select a game to view the complete details of the game.
- A registered user can browse his/her personal game wishlist by pressing the «View your wishlist» button placed in the main menu. The wishlist will display first a page with then games. The first ten games displayed follow the alphabetical order. In the wishlist page only the game name and the year of release are shown. Then, the registered user, if he/she wants can browse and see the games sorted by name on other pages by pressing the buttons «Previous page» or «Next page».
- A registered user, he/she once lands in his/her personal wishlist page, can select a game to view the complete details of the game.
- A registered user can see the first ten suggested friends by pressing the button «Users you might follow». The suggestion is based on common games added to library.
- A registered user can see the top ten reviewers of last month by pressing the button «Most active reviewers».
- A registered user can see the top ten rated games in terms of rating ratio² by pressing the button «Top rated games».
- A registered user can see, after specifying a year, the trending games chart of the selected year (in terms of how many users had added the game to their library) by pressing the button «Trending games chart».
- A registered user can see, after pressing the button «Show suggested games», the top ten games suggested by the platform, based on games not yet in the user's library but owned by the people he/she follows.

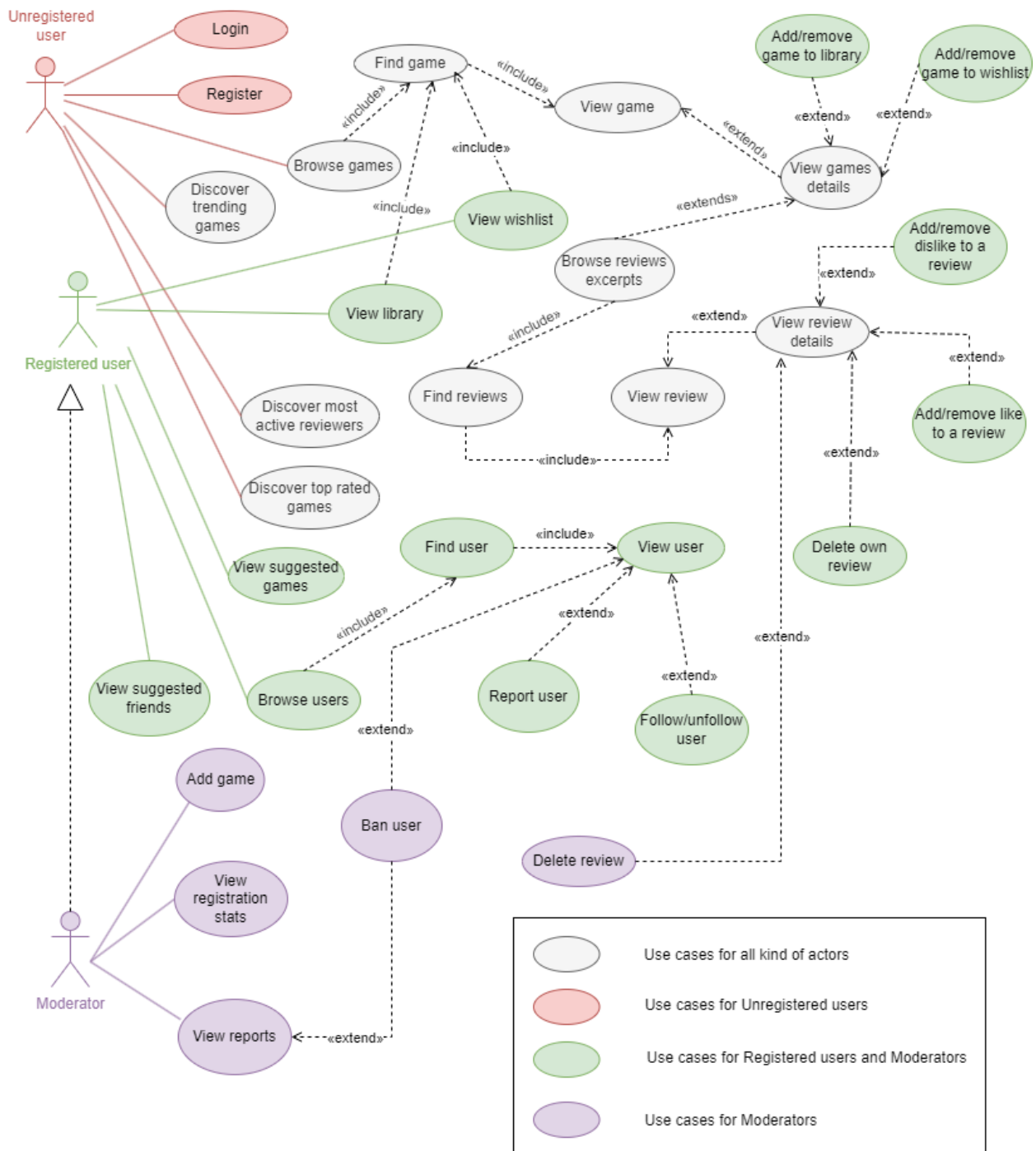
3.3 Functional Requirements – Moderator:

- The system must allow a moderator to perform all the actions that can be performed by a registered user.
- A moderator can press on the «Moderator area» to access the moderator dashboard.
- A moderator can see, after landing in the moderator dashboard, the reports issues on the users.
- A moderator, after seeing the reports, can select a user and ban it from the platform.
- A moderator, after landing in the moderator dashboard, can add a new game to the platform.
- A moderator can see, after landing in the moderator dashboard, the user's registration stats: after selecting a specific year, a table with the number of user's registration per month will appear (the users' count are computed by age range).

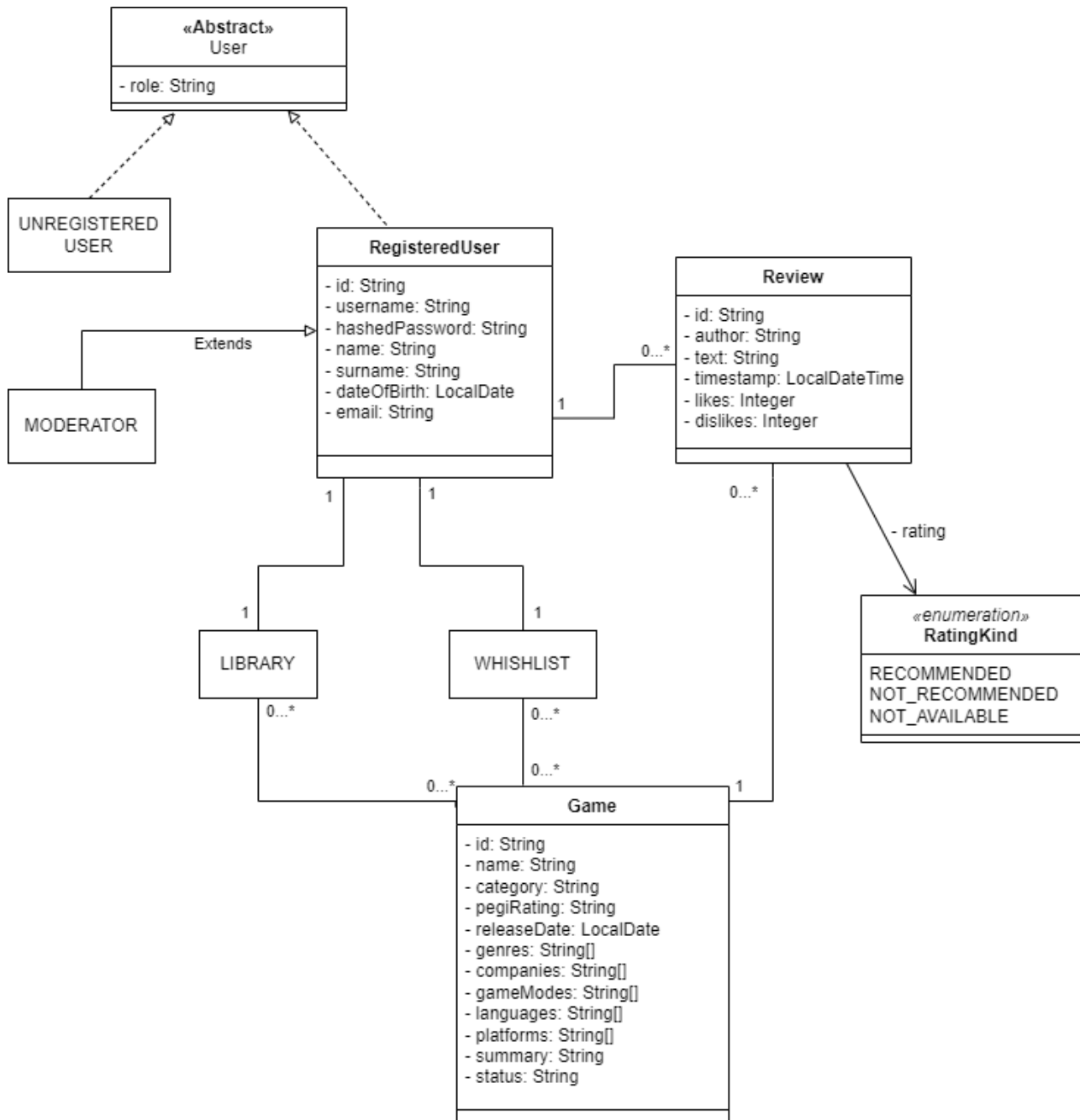
3.4 Non-functional Requirements

- The system must check the uniqueness of usernames.
- The system must hash users' passwords using secure hash functions and salting.
- The system must be developed using an OOP language.
- The system must be usable through a user-friendly Command Line Interface.
- The system must ensure high availability and must be able to recover from network partitions.

4. UML use case diagram



5. UML Class Diagram



6. Data modelling

6.1 Document database

The entities involved in document databases are:

- User
- Game
- Review

This is how we have mapped the entities into Document DB collections:

Collection **users**: In this collection we inserted all the details of a user. Inside this collection there are specific details for the registered users (name, surname, date of birth, email, registration date and so on). The **role** is a String attribute that takes one of the following values: "user", "moderator". The **followers** and **following** fields are numerical fields that count how many other users follow the considered account and how many are the user that the considered account follows, respectively. Notice that these two fields are updated in eventual consistency. More details will be provided in the next chapters. The field **reportedBy** is an embedded array containing the list of usernames that have reported the considered user. We have supposed that a user could receive at most 50 reports, so there's no risk that the embedded array **reportedBy** grows up indefinitely. Such a big number of reports will certainly not go unnoticed by a moderator intervention. **isBanned** is a Boolean value that indicates whether the user has been banned or not. We keep the banned users in the collection to avoid that the banned user can sign up again with the same username.

```
1. "users":{
2.   _id,
3.   username,
4.   hashedPassword,
5.   dateOfBirth,
6.   email,
7.   name,
8.   surname,
9.   role,
10.  followers,
11.  following,
12.  registrationDate,
13.  reportedBy: [
14.    username1,
15.    username2,
16.    ...,
17.    username50
18.  ],
19.  isBanned
20. }
```

Collection **games**: In this collection we have all the details of a game along with the companies that have participated in the development of that game and the list of genres that are associated to a game. Let's see one by one the fields of a game document:

_id: The unique index automatically assigned by MongoDB identifying a specific game.

name: The title of the game.

category: Examples values of categories are **main_game**, **port**, **dlc** and so on.

first_release_date: The date in which the game was released (without taking into account the following updates) expressed as a BSON ISODate.

pegiRating: This attribute expresses the minimum recommended age to play a specific videogame. Basing on game contents, themes, and potential inappropriate materials an age rating authority like PEGI (for Europe) gives the minimum age advised to play that game.

game_modes: This is an array of strings. Each string inside this array could be something like **singleplayer**, **multiplayer**, ... and so on.

genres: This is an array of strings which specifies the game genres. A genre is something like **action**, **FPS**, **RPG**, **fantasy**, **horror** and so on.

platforms: The platform field includes the list of all the platforms (PC, portable devices, consoles) on which the game can be played.

language_supports: This array contains as strings all the languages in which the game has been translated to.

companies: This is an array of strings which reports which companies have been involved in the development of the game.

status: This field, if present in the document, is a String field that expresses the release status of the game. Possible values are **alpha**, **beta**, **cancelled**, **delisted**, **early_access**, **offline**, **released**, **rumored**.

summary: A detailed description of the game.

consistent: this is a Boolean flag that is used to inform which games are consistent across our distributed database deployment. More details are provided later on.

```
1. "games":{
2.   _id,
3.   name,
4.   category,
5.   first_release_date,
6.   pegiRating,
7.   game_modes:[
8.     game_modes1,
9.     game_modes2,
10.    ...,
11.    game_modesN,
12.  ],
13.   genres: [
14.     genre1,
15.     genre2,
16.     ...,
17.     genreN
18.  ],
19.   platforms: [
20.     platform1,
21.     platform2,
22.     ...,
23.     platformN
24.  ],
25.   language_supports:[
26.     language1,
27.     language2,
28.     ...,
```

```

29.     languageN
30. ],
31. companies:[
32.     company1,
33.     company2,
34.     ...,
35.     companyN
36. ]
37. status,
38. summary,
39. consistent,
40. }

```

Collection **reviews**: The collection of the reviews is the collection of the documents containing the details of a game review. Each document contains the following fields: **_id** is the identifier of the review automatically set by MongoDB; **review** is a String attribute that contains the complete text of the review; **author** is the username of the user that have posted the review; **recommended** is a Boolean attribute that expresses the user's opinion on the game (if he/she recommends the game or not). If a document lacks the recommend field, when we'll visualize the review from the side of the application the review's recommendation will appear as "NOT AVAILABLE". This eventuality occurs even when the user has a neutral opinion about the game he/she is reviewing. The **postedDate** attribute is a BSON field of type ISODate corresponding to the date in which the review's author has published the review for the game whose id is specified in **gameId** field. The **gameName** and the **gameReleaseYear** field are two redundant fields of the collection games that we have included in order to explore only just the reviews collection while we make the query in MongoDB to display the top rated games. The **likes** and **dislikes** fields are two integer fields that contain the count of the users that have liked or disliked the review respectively. These fields are updated in eventual consistency. More details will be provided in the next chapters.

```

1. "reviews":{
2.   _id,
3.   reviews,
4.   author,
5.   recommended,
6.   postedDate,
7.   gameId,
8.   gameName,
9.   gameReleaseYear,
10.  likes,
11.  dislikes
12. }

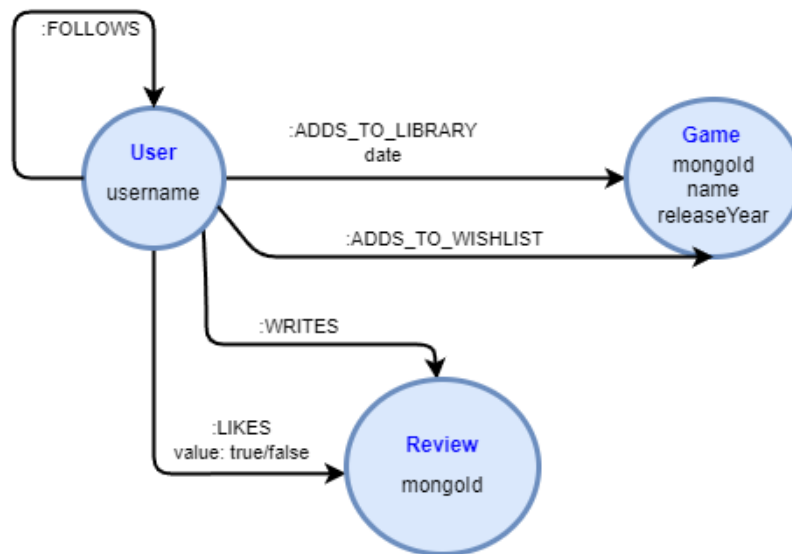
```

6.2 Graph database

The entities involved in the graph database are the following:

- User
- Game
- Review
- Library
- Wishlist

This is a snapshot of the graph structure:



In the Graph database there are three possible types of nodes:

- **User:** The node with label User contains just the **username**. This is because we want to suggest new potential users to follow based on the games that both the users have added to their library.
- **Game:** The node with label Game contains the **mongold**: this field expresses a identification reference for a game both in the Document and Graph databases. Since the retrieval of a user's personal library and user's personal wishlist is done in Neo4j, when the user decides to view the game details from library or from the wishlist such identifier, previously stored in the application's memory, will be used as a filter in the games collections. The game node contains also **name** and **releaseYear** properties: this because when we show the library or the wishlist to the user, we want to display such information. Notice that these are only minimal information of a game and they are necessary in order not to perform multiple requests to different databases when performing the use-cases "View library" and "View wishlist".
- **Review:** The node with label Review contains just the **mongold** as property. This field is used in a similar way with respect to the Game node: in particular, after a user has viewed a review, the review's **mongold** is stored into the application's memory ready for a possible like or dislike interaction by the user on the review. In such case, we need that field to retrieve the specific review node in the Graph database. This happens also when the user actor performs the "Remove own review" use case.

As concerns the relationships, in the Graph database we've designed there are five different relationships:

- **:FOLLOWS:** The :FOLLOWS relationship can link two nodes of type User. If the user with username *a* starts to follow the user with username *b*, a new oriented relationship from *a* to *b* will be created. The following relationship among users is not necessarily mutual.
- **:LIKES:** The :LIKES relationship connects two nodes of type User and Review, allowing a user to maintain a like or dislike vote on another user's review until the vote is removed. This relationship has associated as property a boolean value for expressing whether the user's vote on a review consists in a like or in a dislike.
- **:ADDS_TO_LIBRARY:** The ADDS_TO_LIBRARY edge is how we decided to map the many to many relationship between games and a user's library. The date property attached to the ADDS_TO_LIBRARY relationship corresponds to the date in which the user added the game into his/her personal library.
- **:ADDS_TO_WISHLIST:** The ADDS_TO_WISHLIST edge is how we decided to map the many to many relationship between games and a user's wishlist.

6.3 Discussion on the choices made during data modelling

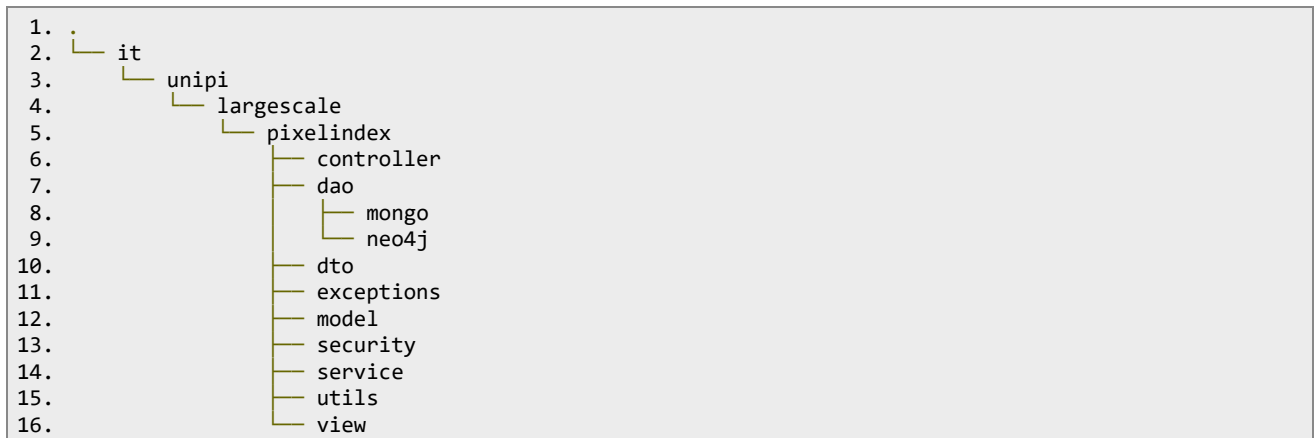
- We have decided to put a unique index on the field username of the users collection in the document database because we want to impose that a specific username can be possessed by one and only one registered user. Consequently, if a user is banned from the platform, we assume that any new user cannot redeem any username involved in a banning procedure.
- During the design phase, we deeply thought whether to choose a Key-Value database for implementing the wishlist. We have decided to discard the usage of a Key-Value database because, contrary to the "cart example" for the E-commerce application we have seen during the course, the removal of a game from a user's wishlist is not so frequent as adding it. Also, if we consider the start of multiple application instances (each one with a different user that performs the login), the Key-Value database would need to store in memory all the wishlists of all the users connected to the platform, thus resulting in a quick exhaustion of the RAM memory and consequently in frequent swap-in and swap-out operation that will break down the rapidity of in-memory read and write operations.

7. Application architecture and implementation

Our application adopts a monolithic architecture. This means that our CLI app is a standalone client that directly interacts with the database infrastructure. This chapter offers an overview of our project's structure, detailing the packages included and their respective responsibilities.

7.1 Packages

The codebase is organized into the following packages structure:



- **Controllers:** This package contains the logic to manage user interactions and commands.
- **DAO:** This package includes all queries and logic for database read and write operations. It is divided into two sub-packages, each dedicated to a specific database:
 - **Mongo**
 - **Neo4J**
- **DTO:** This package contains all the necessary objects for data transfer to and from the user interface.
- **Exceptions:** a collection of custom exceptions.
- **Model:** This package contains the classes mapping the entities of our application
- **Security:** package that contains some security-relevant data classes, such as the one to hash users' passwords.
- **Service:** Here we find the implementation of the application's functional requirements.
- **Utils:** Miscellaneous utility functions.
- **View:** Classes responsible for the visualization of the data within our CLI interface.

7.2 Main classes

DTO		
Class	Package	Description
AuthUserDTO	-	Contains the data of the authenticated user
GameLibraryElementDTO	-	Contains the subset of information of a game to display when the user opens his/her library
GamePreviewDTO	-	Contains the subset of data of a game to display when the user searches for it, alongside with the game's PEGI

		rating (if available) to prevent under-age users to expand the game details.
GameRatingDTO	-	Contains the subset of data of a game to display when it appears in the "top rated games" section, alongside with its positive review ratio
MostActiveReviewerDTO	-	Contains the necessary data of a user that appears in the "most active reviewers" section
RegistrationStatsDTO	-	Contains, for a specific month, the amount of registered users grouped by age group
ReviewPageDTO	-	Contains the number of total count of reviews for a specific game and a list of ReviewPreviewDTO for that specific page
ReviewPreviewDTO	-	Contains a brief preview of the review of a game, consisting in the excerpt, the author's opinion and the author's username
TrendingGamesDTO	-	Contains the subset of data of a game to display when it appears in the "trending games" section, alongside with the number of "adds to library" relationships
UserLoginDTO	-	Contains the data necessary to perform the login procedure
UserRegistrationDTO	-	Contains the data necessary to perform the registration procedure
UserReportsDTO	-	Contains a subset of data of a user in order to display it in the "View Reports" section, alongside with the number of reports that user has
UserSearchDTO	-	Contains a subset of data of a user to be shown when that user is displayed in the "search user" section

DAO		
Class	Package	Description
BaseMongoDAO	mongo	DAO class that reads an .env file and opens a connection to MongoDB. Other MongoDAOs will extend this class
GameMongoDAO	mongo	DAO Class responsible of managing the games collection on Mongo
RegisteredUserMongoDAO	mongo	DAO Class responsible of managing the users collection on Mongo
ReviewMongoDAO	mongo	DAO Class responsible of managing the reviews collection on Mongo
StatisticsMongoDAO	mongo	DAO Class responsible of querying MongoDB to perform statistics and analytics
BaseNeo4jDAO	neo4j	DAO class that reads an .env file and opens a connection to Neo4J. Other Neo4jDAOs will extend this class
GameNeo4jDAO	neo4j	DAO Class responsible of managing game nodes on Neo4J

LibraryNeo4jDAO	neo4j	DAO Class responsible of managing "adds to library" relationships on Neo4j
RegisteredUserNeo4jDAO	neo4j	DAO Class responsible of managing user nodes on Neo4j
ReviewNeo4jDAO	neo4j	DAO Class responsible of managing review nodes on Neo4j
SuggestionsNeo4jDAO	neo4j	DAO Class responsible of performing Graph-based queries, such as game or follow suggestions
WishlistNeo4jDAO	neo4j	DAO Class responsible of managing the "adds to wishlist" relationships on Neo4j

Security		
Class	Package	Description
Crypto	-	Class that contains utility functions to perform simple security operations, such as generating random bytes for salting and hashing user passwords

Service		
Class	Package	Description
GameServicesImpl	impl	Implementation of the GameService interface, containing the implementation for game-related functional requirements.
LibraryServiceImpl	impl	Implementation of the LibraryService interface, containing the implementation for library-related functional requirements.
ModeratorServiceImpl	impl	Implementation of the ModeratorService interface, containing the implementation for moderator-related functional requirements.
RegUserServiceImpl	impl	Implementation of the RegUserService interface, containing the implementation for registered user related functional requirements.
ReviewServiceImpl	impl	Implementation of the ReviewService interface, containing the implementation for reviews-related functional requirements.
StatisticsServiceImpl	impl	Implementation of the StatisticsService interface, containing the implementation for statistics-related (analytics) functional requirements.
SuggestionsServiceImpl	impl	Implementation of the SuggestionsService interface, containing the implementation for suggestion-related functional requirements.
WishlistServiceImpl	impl	Implementation of the WishlistService interface, containing the

		implementation for wishlist-related functional requirements.
ServiceLocator	-	Initializes single instances of each service and provides methods to retrieve them

Other packages, including controllers, views, models, and utils, are not detailed here as they primarily handle user interactions, data modeling, and utility functions of lesser relevance.

8. Statistics and queries

In the following paragraphs we are going to dive into the main queries of our application, both for Mongo DB and Neo4j.

8.1 Mongo DB

For each query we are going to provide a summary of the query behavior, both JavaScript and Java version of the query.

8.1.1 Find Top Reviewers by Reviews Count

This query aims to provide the top n users with the highest number of reviews posted in the last month. This aggregation is particularly useful for analysing user engagement over specific periods, identifying top contributors in terms of reviews within the defined date range.

JS:

```

1. db.reviews.aggregate(
2.   {
3.     $match: {
4.       postedDate: {
5.         $gte: "<firstDayLastMonth>",
6.         $lte: "<lastDayLastMonth>",
7.       },
8.     },
9.   },
10.  {
11.    $group: {
12.      _id: "$author",
13.      count: { $sum: 1 },
14.    },
15.  },
16.  {
17.    $sort: {
18.      count: -1,
19.    },
20.  },
21.  {
22.    $limit: "<n>",
23.  }
24. );

```

JAVA:

```

1. try (MongoClient mongoClient = beginConnection(false)) {
2.     MongoDBDatabase database = mongoClient.getDatabase("pixelindex");
3.     MongoCollection<Document> collection = database.getCollection("reviews");
4.
5.     // For the sake of testing, we set the month to November 2023

```

```

6.         ZonedDateTime now = ZonedDateTime.of(2023, 11, 1, 0, 0, 0, 0, ZoneId.of("UTC"));
7.         ZonedDateTime firstDayLastMonth =
8.
now.with(TemporalAdjusters.firstDayOfMonth()).toLocalDate().atStartOfDay(ZoneId.of("UTC"));
9.         ZonedDateTime lastDayLastMonth =
10.
now.with(TemporalAdjusters.lastDayOfMonth()).toLocalDate().atStartOfDay(ZoneId.of("UTC")).plusDays(1).minusSeconds(1);
11.
12.         Bson matchStage = match(
13.             and(gte("postedDate", firstDayLastMonth.toInstant()),
14.                 lte("postedDate", lastDayLastMonth.toInstant())));
15.         Bson groupStage = group("$author", sum("count", 1));
16.         Bson sortStage = sort(new Document("count", -1));
17.         Bson limitStage = limit(n);
18.
19.         AggregateIterable<Document> result = collection.aggregate(Arrays.asList(matchStage,
groupStage, sortStage, limitStage));
20.         ArrayList<MostActiveUserDTO> userDTOs = new ArrayList<>();
21.         int count = 1;
22.         for (Document user : result) {
23.             MostActiveUserDTO userDTO = new MostActiveUserDTO();
24.             userDTO.setRank(count);
25.             userDTO.setUsername(user.getString("_id"));
26.             userDTO.setNumOfReviews(user.getInteger("count", 0));
27.             userDTOs.add(userDTO);
28.             count++;
29.         }
30.         return userDTOs;
31.     } catch (Exception e) {
32.         throw new DAOException(e);
33.     }

```

Steps:

1. **\$match Stage:** Filters documents by **postedDate**, only including those within a specific date range (from the first to the last day of the previous month).
2. **\$group Stage:** Groups the documents by the **author** field and counts the number of documents in each group. This stage aggregates documents per author, resulting in a count of reviews that each author has made within the specified time frame.
3. **\$sort Stage:** Sorts the groups based on the count in descending order. This means authors with the most documents appear first. (*enables pagination*)
4. **\$limit Stage:** Limits the number of documents in the output to **n**, which is a variable representing the desired number of top records. (*enables pagination*)

8.1.2 Get Reviews by Game ID

This aggregation is designed to fetch a page of reviews for a specific game, sorted by the number of likes, while also including a count of the total number of reviews; if the current user has written a review for this game, his/her review will be shown as the first one. If the text of the review is longer than 50 characters, then it will be truncated to improve the readability.

JS:

```

1. db.reviews.aggregate(
2.     {
3.         $match: { gameId: { $oid: "<gameId>" } },
4.     },
5.     {

```

```

6.    $facet: {
7.        totalCount: [{ $count: "count" }],
8.        data: [
9.            {
10.                $addFields: {
11.                    byUser: {
12.                        $cond: {
13.                            if: { $eq: ["$author", "<username>"] },
14.                            then: 1,
15.                            else: 0,
16.                        },
17.                    },
18.                },
19.            },
20.            { $sort: { byUser: -1, likes: -1 } },
21.            {
22.                $project: {
23.                    review: {
24.                        $cond: {
25.                            if: { $gt: [{ $strLenCP: "$review" }, 50] },
26.                            then: {
27.                                $concat: [{ $substrCP: ["$review", 0, 50] }, "..."],
28.                            },
29.                            else: "$review",
30.                        },
31.                    },
32.                    author: 1,
33.                    recommended: 1,
34.                    postedDate: 1,
35.                },
36.            },
37.            { $skip: 10 * "<page>" },
38.            { $limit: 10 },
39.        ],
40.    },
41. },
42. { $unwind: "$totalCount" },
43. {
44.     $addFields: {
45.         totalCount: "$totalCount.count",
46.     },
47. },
48. {
49.     $replaceRoot: {
50.         newRoot: {
51.             $mergeObjects: [
52.                 "$$ROOT",
53.                 {
54.                     data: "$data",
55.                     totalCount: "$totalCount",
56.                 },
57.             ],
58.         },
59.     },
60. },
61. {
62.     $project: {
63.         data: 1,
64.         totalCount: 1,
65.     },
66. }
67. );

```

JAVA:

```

1. try (MongoClient mongoClient = beginConnection(false)) {
2.     MongoDBDatabase database = mongoClient.getDatabase("pixelindex");
3.     MongoCollection<Document> collection = database.getCollection("reviews");

```

```

4.
5.     ArrayList<Document> result = collection.aggregate(
6.         Arrays.asList(
7.             new Document("$match", new Document("gameId", new ObjectId(gameId))),
8.             new Document("$facet",
9.                 new Document("totalCount", Arrays.asList(new Document("$count", "count")))
10.                .append("data", Arrays.asList(
11.                    new Document("$addFields",
12.                        new Document("byUser",
13.                            new Document("$cond",
14.                                new Document("if",
15.                                    new Document("$eq", Arrays.asList("$author", username)))
16.                                .append("then", 1L)
17.                                .append("else", 0L))),
18.                    new Document("$sort", new Document("byUser", -1L) .append("likes", -1L)),
19.                    new Document("$project",
20.                        new Document("review",
21.                            new Document("$cond",
22.                                new Document("if",
23.                                    new Document("$gt", Arrays.asList(new
Document("$strLenCP", "$review"), 50L)))
24.                                .append("then",
25.                                    new Document("$concat", Arrays.asList(new
Document("$substrCP", Arrays.asList("$review", 0L, 50L)), "..."))
26.                                    .append("else", "$review"))))
27.                                .append("author", 1L)
28.                                .append("recommended", 1L)
29.                                .append("postedDate", 1L)
30.                                .append("likes", 1L)
31.                                .append("dislikes", 1L)),
32.                    new Document("$skip", 10L * page),
33.                    new Document("$limit", 10L))),
34.                new Document("$unwind", "$totalCount"),
35.                new Document("$addFields", new Document("totalCount", "$totalCount.count")),
36.                new Document("$replaceRoot",
37.                    new Document("newRoot",
38.                        new Document("$mergeObjects", Arrays.asList("$ROOT",
39.                            new Document("data", "$data")
40.                            .append("totalCount", "$totalCount")))),
41.                    new Document("$project",
42.                        new Document("data", 1L)
43.                        .append("totalCount", 1L))))).into(new ArrayList<>());
44.
45.     ReviewPageDTO reviewPage = new ReviewPageDTO();
46.     // check if the result is empty
47.     if (result.isEmpty()) {
48.         reviewPage.setTotalReviewsCount(0);
49.         reviewPage.setReviews(new ArrayList<>());
50.         return reviewPage;
51.     } else {
52.         reviewPage.setTotalReviewsCount(result.get(0).getInteger("totalCount"));
53.
54.         List<ReviewPreviewDTO> reviews = new ArrayList<>();
55.         for (Document res : result.get(0).getList("data", Document.class)) {
56.             ReviewPreviewDTO review = reviewPreviewFromQueryResult(res);
57.             reviews.add(review);
58.         }
59.         reviewPage.setReviews(reviews);
60.
61.         return reviewPage;
62.     }
63.
64. } catch (Exception e) {
65.     throw new DAOException("Error while retrieving reviews by game id" + e);
66. }

```

Steps:

1. **\$match Stage:** Filters the reviews to include only those where the **gameId** matches a specific ObjectId. This stage narrows down the documents to those related to a particular game.

2. **\$facet Stage:** Performs a multi-faceted aggregation to produce two parallel sets of results: **totalCount** and **data**.
 - **totalCount:** Uses **\$count** to calculate the total number of reviews for that specific game.
 - **data:** A series of operations to prepare and format the actual review data:
 - **\$addFields:** Adds a new field, **byUser**, which is a flag indicating whether the review was written by a specific user (matching **username**). This is determined by a condition that checks if the **author** field matches the given username, setting **byUser** to 1 if true, and 0 otherwise.
 - **\$sort:** Sorts the documents first by **byUser** in descending order (prioritizing the review of the current user, if exists) and then by **likes** in descending order (highlighting popular reviews).
 - **\$project:** Transforms the document to include only the desired fields. It conditionally shortens the **review** text to the first 50 characters plus an ellipsis if the review is longer than 50 characters, and includes **author**, **recommended**, and **postedDate** fields as is.
 - **\$skip** and **\$limit:** Implements pagination by skipping a calculated number of documents (**10 * page**) and then limiting the results to 10 documents. This allows for batch retrieval of the data, useful for displaying results page by page.
3. **\$unwind Stage:** Deconstructs **totalCount** (always a single-element array), generated by the **\$facet** stage, so it can be accessed as embedded document.
4. **\$addFields Stage:** Takes the unwound **totalCount** value and promotes it to a top-level field in the documents. This allows for easy access to the total count of matching documents alongside the paginated data.
5. **\$replaceRoot Stage:** Uses **\$mergeObjects** to combine the root document (**\$\$ROOT**) with the **data** and **totalCount** fields into a new root document. This effectively flattens the structure, making **data** and **totalCount** top-level fields.
6. **\$project Stage:** Finally, this stage explicitly includes only the **data** and **totalCount** fields in the output documents. This stage shapes the final output, focusing on the prepared review data and the total count of matching documents.

8.1.3 Get Games Advanced Search

This query allows to perform an advanced search in the game collection, exploiting various attributes, like the game name, the release year, the companies, and the platforms. The games that have not reached a consistent state are hidden. Moreover, a higher priority is given to games in the *"main_game"* category.

JS:

```
1. db.games.aggregate([
2.   {
3.     $match: {
4.       consistent: true,
5.       $text: { $search: "<name>" },
6.       first_release_date: { $gte: "<startDate>", $lt: "<endDate>" },
7.       companies: { $regex: "<company>", $options: "i" },
8.       platforms: { $regex: "<platform>", $options: "i" },
9.     },
10.  },
11.  { $addFields: { textScore: { $meta: "textScore" } } },
12.  {
13.    $addFields: {
14.      isMainGame: { $cond: [{ $eq: ["$category", "main_game"] }, 1, 0] },
15.    },
16.  },
17.])
```

```

16.   },
17.   { $sort: { textScore: -1, isMainGame: -1 } },
18.   { $skip: 10 * page },
19.   { $limit: 10 },
20.   {
21.     $project: {
22.       _id: 1,
23.       name: 1,
24.       first_release_date: 1,
25.       pegiRating: 1,
26.     },
27.   },
28. ]];

```

JAVA:

```

1. try (MongoClient mongoClient = beginConnection(false)) {
2.     MongoDB database = mongoClient.getDatabase("pixelindex");
3.     MongoCollection<Document> collection = database.getCollection("games");
4.
5.     List<Bson> aggregationPipeline = new ArrayList<>();
6.
7.     // Filter only documents with consistent equal to true
8.     Document matchCriteria = new Document("consistent", true);
9.     // Text search for name
10.    if (name != null && !name.isEmpty()) {
11.        matchCriteria.append("$text", new Document("$search", name));
12.    }
13.    if (releaseYear != null) {
14.        Date startDate = new GregorianCalendar(releaseYear, Calendar.JANUARY, 1).getTime();
15.        Date endDate = new GregorianCalendar(releaseYear + 1, Calendar.JANUARY, 1).getTime();
16.        matchCriteria.append("first_release_date", new Document("$gte", startDate).append("$lt",
endDate));
17.    }
18.    if (company != null && !company.isEmpty()) {
19.        matchCriteria.append("companies", new Document("$regex", company).append("$options", "i"));
20.    }
21.    if (platform != null && !platform.isEmpty()) {
22.        matchCriteria.append("platforms", new Document("$regex", platform).append("$options",
"i"));
23.    }
24.
25.    aggregationPipeline.add(Aggregates.match(matchCriteria));
26.
27.    // Add textScore for sorting
28.    aggregationPipeline.add(Aggregates.addFields(new Field<>("textScore", new Document("$meta",
"textScore"))));
29.
30.    // Prioritizing main_games and textScore
31.    aggregationPipeline.add(
32.        Aggregates.addFields(new Field<>("isMainGame",
33.            new Document("$cond", Arrays.asList(new Document("$eq", Arrays.asList("$category",
"main_game")), 1, 0)))
34.    );
35.
36.    // Sorting stage modified to textScore and main games
37.    aggregationPipeline.add(Aggregates.sort(Sorts.orderBy(Sorts.descending("textScore",
"isMainGame"))));
38.
39.    // Pagination
40.    aggregationPipeline.add(Aggregates.skip(10 * page));
41.    aggregationPipeline.add(Aggregates.limit(10));
42.
43.    // Projection
44.    aggregationPipeline.add(Aggregates.project(new Document("_id", 1L)
45.        .append("name", 1L)
46.        .append("first_release_date", 1L)
47.        .append("pegiRating", 1L)));
48.

```

```

49. // Aggregation
50. ArrayList<Document> results = collection.aggregate(aggregationPipeline).into(new
ArrayList<>());
51.
52. for (Document result : results) {
53.     GamePreviewDTO game = gamePreviewFromQueryResult(result);
54.     games.add(game);
55. }
56. } catch (Exception e) {
57.     throw new DAOException("Error retrieving games: " + e.getMessage());
58. }

```

Steps:

1. **\$match**: Filters documents to include those that are consistent (**consistent: true**), optionally matches the game name using text search, filters by release year (between **startDate** and **endDate**), and optionally matches the company and platform using case-insensitive regular expressions.
2. **\$addFields (textScore)**: Adds a field **textScore** to each document, which is the relevance score from the text search on the **name** field. This is used later for sorting.
3. **\$addFields (isMainGame)**: Adds a field **isMainGame**, which is set to **1** if the game's category is "main_game" and to **0** otherwise. This field is used to prioritize main games in the sorting stage.
4. **\$sort**: Sorts the documents first by **textScore** in descending order (to prioritize matches that are more relevant to the text search), then by **isMainGame** in descending order (to prioritize main games).
5. **\$skip**: Skips several documents based on the **page** variable, enabling pagination. The number to skip is calculated as **10 * page**.
6. **\$limit**: Limits the number of documents returned to 10, implementing pagination.
7. **\$project**: Specifies the fields to include in the final output (**_id**, **name**, **first_release_date**, **pegiRating**), shaping the documents to contain only relevant information for the frontend or application layer.

8.1.4 Top Games by Positive Rating Ratio

This aggregation pipeline is designed to analyse review data for games, identifying highly recommended games based on a positive rating ratio, and focusing on those with a substantial number of reviews to ensure the reliability of the rating metric.

JS:

```

1. db.reviews.aggregate([
2.   {
3.     $group: {
4.       _id: "$gameId",
5.       gameName: { $first: "$gameName" },
6.       gameReleaseYear: { $first: "$gameReleaseYear" },
7.       positiveReviews: {
8.         $sum: {

```



```

26.                                     new Document("$add",
Arrays.asList("$positiveReviews", "$negativeReviews")))), 100)))
27.     )),
28.     sort(descending("positiveRatingRatio")),
29.     limit(n)
30. );
31.
32. AggregateIterable<Document> results = collection.aggregate(aggregationPipeline);
33. int rank = 1;
34. for (Document doc : results) {
35.     GameRatingDTO dto = new GameRatingDTO();
36.     dto.setRank(rank);
37.     dto.setName(doc.getString("gameName"));
38.     if (doc.get("gameReleaseYear") != null) {
39.         dto.setReleaseYear(doc.getInteger("gameReleaseYear"));
40.     }
41.     dto.setPositiveRatingRatio(doc.getDouble("positiveRatingRatio"));
42.     gameRatingDTOs.add(dto);
43.     rank++;
44. }
45. } catch (Exception e) {
46.     throw new DAOException("Error retrieving games: " + e.getMessage());
47. }
48.

```

Steps:

1. **\$group**: Aggregates documents by **gameId**, compiling the first occurrence of **gameName** and **gameReleaseYear** for each group, and counts **positiveReviews**, **negativeReviews**, and **totalReviews**. Positive and negative reviews are counted based on the **recommended** field's value.
2. **\$match**: Filters the aggregated results to include only those games with a **totalReviews** count of 15 or more. This step ensures that only games with a significant number of reviews are considered in further calculations and output.
3. **\$project**: Transforms the aggregated data to include the game's name, release year, and calculates the **positiveRatingRatio**. The positive rating ratio is calculated as the percentage of positive reviews out of the total reviews (excluding the neutral ones), aiming to provide a metric for comparing games based on user recommendations. Some checks are implemented in order to avoid division by zero.
4. **\$sort**: Orders the resulting documents by **positiveRatingRatio** in descending order, prioritizing games with a higher ratio of positive reviews.
5. **\$limit**: Restricts the output to **n** documents, where **n** is a variable indicating the number of top results to return.

8.1.5 Number of Registrations by Month

This query aims to categorize user registrations by month and *age group* for a specific year, then processes and organizes this data into a structured format suitable for reporting.

JS:

```

1. db.users.aggregate([
2.   {
3.     $match: { $expr: { $eq: [{ $year: "$registrationDate" }, year] } },
4.   },
5.   {

```

```

6.     $project: {
7.         month: { $month: "$registrationDate" },
8.         age: { $subtract: [2024, { $year: "$dateOfBirth" }] },
9.     },
10. },
11. {
12.     $group: {
13.         _id: {
14.             month: "$month",
15.             ageGroup: {
16.                 $switch: {
17.                     branches: [
18.                         { case: { $lt: ["$age", 18] }, then: "< 18 y.o." },
19.                         {
20.                             case: { $and: [{ $gte: ["$age", 18] }, { $lte: ["$age", 30] }], },
21.                             then: "18-30 y.o.",
22.                         },
23.                         {
24.                             case: { $and: [{ $gt: ["$age", 30] }, { $lte: ["$age", 50] }], },
25.                             then: "30-50 y.o.",
26.                         },
27.                         { case: { $gt: ["$age", 50] }, then: "50+ y.o." },
28.                     ],
29.                     default: "Other",
30.                 },
31.             },
32.         },
33.         count: { $sum: 1 },
34.     },
35. },
36. {
37.     $group: {
38.         _id: { month: "$_id.month" },
39.         properties: {
40.             $push: {
41.                 ageGroup: "$_id.ageGroup",
42.                 count: "$count",
43.             },
44.         },
45.     },
46. },
47. {
48.     $project: {
49.         month: "$_id.month",
50.         properties: 1,
51.         _id: 0,
52.     },
53. },
54. { $sort: { month: 1 } },
55. ]);

```

JAVA:

```

1. try (MongoClient mongoClient = beginConnection(false)) {
2.     MongoDBDatabase database = mongoClient.getDatabase("pixelindex");
3.     MongoCollection<Document> collection = database.getCollection("users");
4.
5.     AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
6.         new Document("$match",
7.             new Document("$expr",
8.                 new Document("$eq", Arrays.asList(new Document("$year", "$registrationDate"),
9. year))))),
10.         new Document("$project",
11.             new Document("month",
12.                 new Document("$month", "$registrationDate"))
13.                 .append("age",
14.                     new Document("$subtract", Arrays.asList(2024L,
15.                         new Document("$year", "$dateOfBirth")))),
16.             new Document("$group",
17.                 new Document("_id",

```

```

17.         new Document("month", "$month")
18.         .append("ageGroup",
19.             new Document("$switch",
20.                 new Document("branches", Arrays.asList(new Document("case",
21.                     new Document("$lt", Arrays.asList("$age", 18L)))
22.                     .append("then", "< 18 y.o."),
23.                     new Document("case",
24.                         new Document("$and", Arrays.asList(new Document("$gte",
Arrays.asList("$age", 18L)),
25.                             new Document("$lte", Arrays.asList("$age", 30L))))))
26.                     .append("then", "18-30 y.o."),
27.                     new Document("case",
28.                         new Document("$and", Arrays.asList(new Document("$gt",
Arrays.asList("$age", 30L)),
29.                             new Document("$lte", Arrays.asList("$age", 50L))))))
30.                     .append("then", "30-50 y.o."),
31.                     new Document("case",
32.                         new Document("$gt", Arrays.asList("$age", 50L)))
33.                         .append("then", "50+ y.o.")))
34.                     .append("default", "Other")))))
35.         .append("count",
36.             new Document("$sum", 1L))),
37.     new Document("$group",
38.         new Document("_id",
39.             new Document("month", "$_id.month"))
40.         .append("properties",
41.             new Document("$push",
42.                 new Document("ageGroup", "$_id.ageGroup")
43.                 .append("count", "$count")))),
44.     new Document("$project",
45.         new Document("month", "$_id.month")
46.         .append("properties", 1L)
47.         .append("_id", 0L)),
48.     new Document("$sort",
49.         new Document("month", 1L))));
50.
51.     HashMap<Integer, List<Document>> hashMap = new HashMap<>();
52.     ArrayList<RegistrationStatsDTO> registrationStatsDTOs = new ArrayList<>();
53.     List<Document> properties = new ArrayList<>();
54.     for (Document doc : result) {
55.         hashMap.put(doc.getInteger("month"), doc.getList("properties", Document.class));
56.     }
57.
58.     for (Map.Entry<Integer, List<Document>> entry : hashMap.entrySet()) {
59.         RegistrationStatsDTO registrationStatsDTO = new RegistrationStatsDTO();
60.         registrationStatsDTO.setMonth(entry.getKey());
61.
62.         HashMap<String, Long> hashMap1 = new HashMap<>();
63.         for (int i = 0; i < entry.getValue().size(); i++) {
64.             hashMap1.put(entry.getValue().get(i).getString("ageGroup"),
entry.getValue().get(i).getLong("count"));
65.         }
66.         registrationStatsDTO.setHashMap(hashMap1);
67.         registrationStatsDTOs.add(registrationStatsDTO);
68.     }
69.     return registrationStatsDTOs;
70. } catch (Exception e) {
71.     e.printStackTrace();
72.     throw new DAOException(e);
73. }

```

Steps:

1. **\$match Stage:** Filters documents to include only those whose registration year matches the specified year. It uses **\$expr** to allow for the use of aggregation expressions within the match stage, comparing the year extracted from **registrationDate** with the provided year.

2. **\$project Stage:** Projects (or transforms) each document to include the month of registration extracted from **registrationDate** and calculates the age by subtracting the birth year from a fixed year (2024, which seems to be intended as the current year or a static reference point).
3. **\$group Stage 1:** Groups the documents first by month and age group. The age group is determined using a **\$switch** statement that categorizes the **age** into predefined ranges ("*< 18 y.o.*", "*18-30 y.o.*", "*30-50 y.o.*", "*50+ y.o.*", and "*Other*"). It counts the number of users within each age group for each month.
4. **\$group Stage 2:** Groups the results from the previous stage by month only, pushing each age group and its count into an array named **properties** for each month. This effectively prepares a summary of counts per age group for each month.
5. **\$project Stage:** Transforms the documents to format the output, including the month and the **properties** array while excluding the **_id** field.
6. **\$sort Stage:** Sorts the results by month in ascending order to ensure the data is returned in chronological order from January to December.

8.2 Neo4j

In this section we have highlighted the typical on-graph queries:

This table maps graph-centric queries to domain-specific queries:

Graph-Centric Query	Domain-Specific Query
How many incoming <code>:FOLLOWS</code> edges are directed to vertex U?	How many users follow the user with username \$u?
How many outgoing <code>:FOLLOWS</code> edges depart from vertex U?	How many users is the user with username \$u following?
Given a vertex U, which are the Game vertexes connected to U via an <code>:ADDS_TO_LIBRARY</code> relationship?	Which are the games that are part of the library of the user with username \$u?
Given a vertex U, which are the Game vertexes connected to U via an <code>:ADDS_TO_WISHLIST</code> relationship?	Which are the games that are part of the wishlist of the user with username \$u?
Which are the User vertexes connected to other User vertexes by the means of an <code>:ADDS_TO_LIBRARY</code> relationship and not via a direct <code>:FOLLOWS</code> relationships?	Which are the new users suggested to other users basing on mutual games?
Which Games vertexes are connected to a User vertex by the means of another User vertex and	Which are the new games suggested to other users based on follows relationships?

not through a direct :ADDS_TO_LIBRARY relationship?	
-----------------------------------------------------	--

8.2.1 Suggesting new user to follow based on mutual games

This query considers a specific user identified by **\$username** and returns the list of the first ten users sorted considering the descending number of mutual games. First of all, we take the username **\$username** and passing through relationships of type **:ADDS_TO_LIBRARY** we reach the potential new followers (whose alias is **u2**). Then we put the condition written in line 2 in order to check that the new candidate follow is not the same user from which we start the search. Then we check that a **:FOLLOW** relationship between **u1** and **u2**: does not already exists, in this way we exclude the users **u2** already followed by **u1** and this is the key point to suggest only new users.

Cypher:

```
1. MATCH (u1:User {username:$username})-[:ADDS_TO_LIBRARY]->(g: Game)<-[:ADDS_TO_LIBRARY]-(u2: User)
2. WHERE u1.username <> u2.username AND NOT ((u1)-[:FOLLOWS]->(u2))
3. RETURN u2.username AS username, COUNT(g) AS NumberOfMutualGames
4. ORDER BY NumberOfMutualGames DESC
5. LIMIT 10
```

Java:

```
1. public ArrayList<String> getSuggestedUsers(String username) throws DAOException {
2.     ArrayList<String> suggestedUsers;
3.     try (Driver neoDriver = BaseNeo4jDAO.beginConnection());
4.         Session session = neoDriver.session()) {
5.         suggestedUsers = session.executeRead(tx -> {
6.             Result result = tx.run("MATCH (u1:User {username:$username})-[:ADDS_TO_LIBRARY]->(g:
Game)" +
7.                                     "<-[:ADDS_TO_LIBRARY]-(u2: User) " +
8.                                     "WHERE u1.username <> u2.username AND NOT ((u1)-[:FOLLOWS]->(u2)) " +
9.                                     "RETURN u2.username AS username, COUNT(g) AS NumberOfMutualGames " +
10.                                    "ORDER BY NumberOfMutualGames DESC " +
11.                                    "LIMIT 10",
12.                                     parameters("username", username));
13.             ArrayList<String> users = new ArrayList<>();
14.             while (result.hasNext()) {
15.                 Record record = result.next();
16.                 users.add(record.get("username").asString());
17.             }
18.             return users;
19.         });
20.     } catch (Exception ex) {
21.         throw new DAOException(ex);
22.     }
23.     return suggestedUsers;
24. }
```

8.2.2 Suggesting new games based on follower's libraries

This query considers a specific **\$username** and returns the list of the first ten games sorted considering the descending order of connections between the considered user and the other user he/she follows. First of all, we begin from the node with alias **u1**, which is the user to which we have to suggest the new games, then we proceed through the **:FOLLOWS** edge to reach the user followed by **u1** (with alias **u2**). We proceed from **u2** by traversing the **:ADDS_TO_LIBRARY** edge to

access the game nodes, denoted as **g**. In compliance with the condition specified in line 3, **g** are the games within the library of users followed by **u1**, but that are not yet in **u1**'s own library.

Cypher:

```
1. MATCH (u1:User {username:$username})-[f:FOLLOWS]->(u2:User)
2. MATCH (u2:User)-[r:ADDS_TO_LIBRARY]->(g:Game)
3. WHERE NOT EXISTS ((u1)-[:ADDS_TO_LIBRARY]->(g))
4. RETURN g.name AS name, COUNT(f) AS connectionsNumber
5. ORDER BY connectionsNumber DESC
6. LIMIT 10
```

Java:

```
1. public List<GameSuggestionDTO> getSuggestedGames(String username) throws DAOException {
2.     ArrayList<GameSuggestionDTO> suggestedGames;
3.     try (Driver neoDriver = BaseNeo4jDAO.beginConnection());
4.         Session session = neoDriver.session() {
5.         suggestedGames = session.executeRead(tx -> {
6.             Result result = tx.run("MATCH (u1:User {username:$username})-[f:FOLLOWS]->(u2:User)" +
7.                                     "MATCH (u2:User)-[r:ADDS_TO_LIBRARY]->(g:Game)" +
8.                                     "WHERE NOT EXISTS ((u1)-[:ADDS_TO_LIBRARY]->(g))" +
9.                                     "RETURN g.name AS name, COUNT(f) AS connectionsNumber" +
10.                                    "ORDER BY connectionsNumber DESC" +
11.                                    "LIMIT 10",
12.                                     parameters("username", username));
13.             ArrayList<GameSuggestionDTO> games = new ArrayList<>();
14.             while (result.hasNext()) {
15.                 Record record = result.next();
16.                 GameSuggestionDTO gameSuggestionDTO = new GameSuggestionDTO();
17.                 gameSuggestionDTO.setGameName(record.get("name").asString());
18.                 gameSuggestionDTO.setConnectionsNumber(record.get("connectionsNumber").asInt());
19.                 games.add(gameSuggestionDTO);
20.             }
21.             return games;
22.         });
23.     } catch (Exception ex) {
24.         ex.printStackTrace();
25.         throw new DAOException(ex);
26.     }
27.     return suggestedGames;
28. }
```

8.2.3 Retrieving the most trending games based on number of adds to library in a specific year

This query, given a specific year, builds the top **n** rank of the most trending games. We refer to trending game with a game that is added into different user's library more than the others. The Cypher query matches the games added to user's library in the selected year: the year is selected by the side of the application and refers to the year in which the game has been added to the library. With the condition at line 3 we actually express the concept of trend. The games are sorted considering the descending count of "adds to library". The query returns the game name along with the number of players that added that game in the specified year. The results are sorted in descending order by number of players.

Cypher:

```
1. MATCH (g:Game)<-[r:ADDS_TO_LIBRARY]-(u:User)
2. WHERE r.date >= DATE($startDate) AND r.date <= DATE($endDate)
3. WITH g, COUNT(r) AS count
4. ORDER BY count DESC
5. LIMIT $n
6. RETURN g.name AS name, count
```

Java:

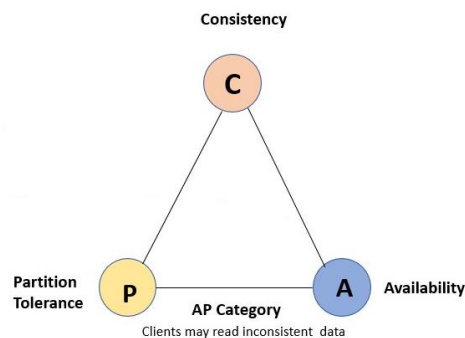
```
1. public List<TrendingGamesDTO> getMostAddedToLibraryGames(Integer year, Integer limit) throws
   DAOException {
2.     String startDate = year + "-01-01";
3.     String endDate = year + "-12-31";
4.
5.     try (Driver neoDriver = beginConnection()) {
6.         String query = ""
7.             MATCH (g:Game)<-[r:ADDS_TO_LIBRARY]-(u:User)
8.             WHERE r.date >= DATE($startDate) AND r.date <= DATE($endDate)
9.             WITH g, COUNT(r) AS count
10.            ORDER BY count DESC
11.            LIMIT $limit
12.            RETURN g.name AS name, count
13.            "";
14.        Value parameters = parameters("startDate", startDate, "endDate", endDate, "limit", limit);
15.
16.        AtomicInteger rank = new AtomicInteger(0);
17.        try (Session session = neoDriver.session()) {
18.            return session.executeRead(tx -> tx.run(query, parameters)
19.                .list(record -> {
20.                    TrendingGamesDTO game = new TrendingGamesDTO();
21.                    game.setRank(rank.incrementAndGet());
22.                    game.setGameName(record.get("name").asString());
23.                    game.setCount(record.get("count").asInt());
24.                    return game;
25.                }));
26.        }
27.    } catch (Exception ex) {
28.        throw new DAOException(ex);
29.    }
30. }
```

9. Distributed databases design

This chapter provides a detailed description of the rationale and decisions made during the design phase of distributed databases. In particular we focus on some consideration regarding the CAP theorem, and we discuss redundancy, eventual consistency, and MongoDB's write/read concerns.

9.1 Considerations on the CAP theorem

Since our application is a social network and browsing gaming system, we should prioritize service availability rather than consistency. So, we relax the consistency vertex of the CAP triangle because in such kind of application it's very import to reduce the response time perceived by the final user while he/she is scrolling games, reviews, adding likes and so on.

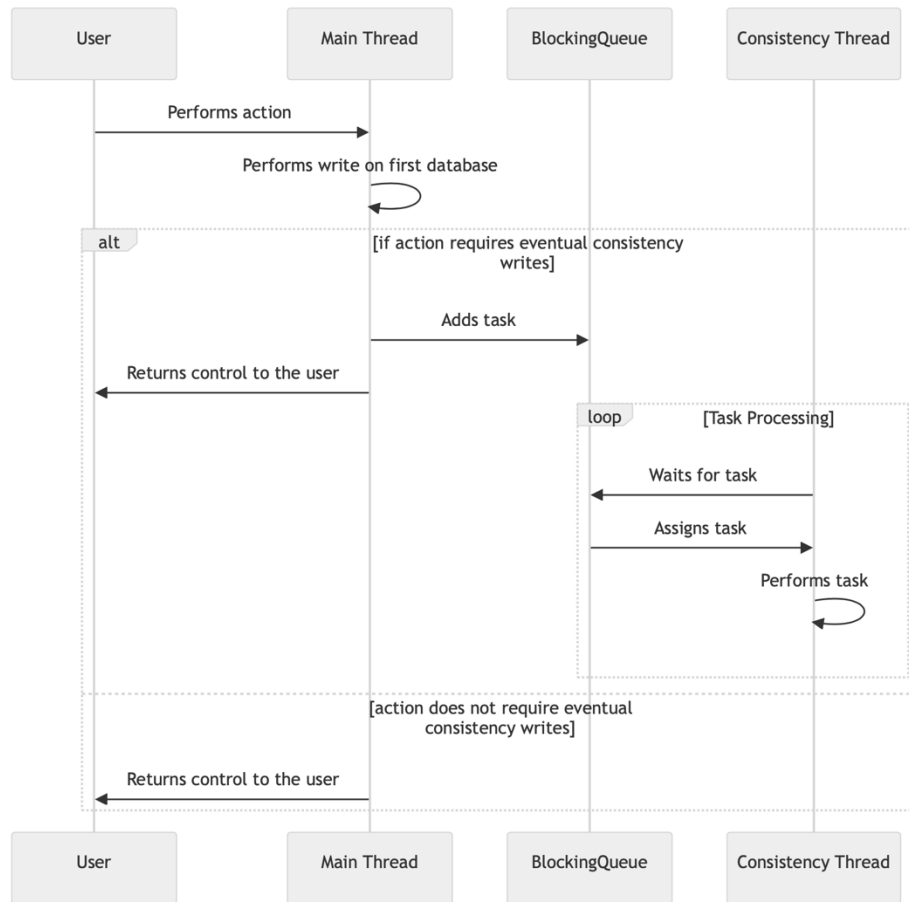


The design of the distributed databases was oriented by us choosing the AP intersection of the CAP triangle. The consistency among the databases will be handled eventually by the below described *Consistency Thread*. Moreover, some tests were performed in order to check if our application is tolerant to network partitions. We simulated a network partition by making one of the Mongo replicas unreachable for the others and verified that our application continues to work normally and seamlessly despite of that.

9.2 Consistency

9.2.1 Consistency Thread

Given that our application is monolithic, we adopted a straightforward and transparent approach to achieve eventual consistency through the use of a secondary thread, which we will refer to as "Consistency Thread". The consistency thread loops over a shared BlockingQueue instance, which the main thread will populate with tasks to guarantee eventual consistency. These tasks, essentially lambda functions, contain the necessary function calls to update the consistency on the second database. The consistency simply pops the first task on the queue and executes it.



This implementation allows us to return the control to the user immediately after making sure that at least one database is consistent, giving the idea of a responsive and highly available application. Below is the implementation of our consistency thread in *Java*. Our class extends the *Thread* class and overrides its *run* method. The Consistency Thread will also keep a *BlockingQueue* of *Runnable* instances as private field, from which it will take tasks (lambda functions) to execute. Additionally, the class provides methods for task queueing and thread termination.

```

1. public class ConsistencyThread extends Thread{
2.     private boolean running = true;
3.     private final BlockingQueue<Runnable> taskQueue;
4.     private final int DELAY = 5000;
5.     public ConsistencyThread (BlockingQueue<Runnable> taskQueue) {
6.         this.taskQueue = taskQueue;
7.     }
8.
9.     @Override
10.    public void run(){
11.        try{
12.            while(running || !taskQueue.isEmpty()){
13.                Runnable task = taskQueue.take();
14.                task.run();
15.                Thread.sleep(DELAY);
16.            }
17.        }catch(InterruptedException ex)
18.        {
19.            Thread.currentThread().interrupt();
20.            System.out.println("The consistency thread has been interrupted");
21.        }
22.    }
  
```

```

23.
24.     public void addTask(Runnable task) {
25.         taskQueue.add(task);
26.     }
27.     public void stopThread() {
28.         this.running = false;
29.     }
30. }

```

An example of usage can be found when a user follows another user and the followers count must be updated. This snippet is taken from `RegUserServiceImpl.java`.

```

1. String outcome = registeredUserNeo.followUser(usernameSrc, usernameDst);
2. consistencyThread.addTask(() -> {
3.     try{
4.         Map<String, Integer> folCount = registeredUserNeo.getFollowsCount(usernameSrc,
usernameDst);
5.         registeredUserMongo.updateFollowers(usernameSrc, usernameDst, folCount.get("followingSrc"),
folCount.get("followerDst"));
6.     }catch(DAOException e){
7.         System.out.println("Consistency update failed");
8.     }
9. });

```

9.2.2 Eventual consistency operations and redundancy updates

In this paragraph we will provide the details of where and why the eventual consistency operations are necessary:

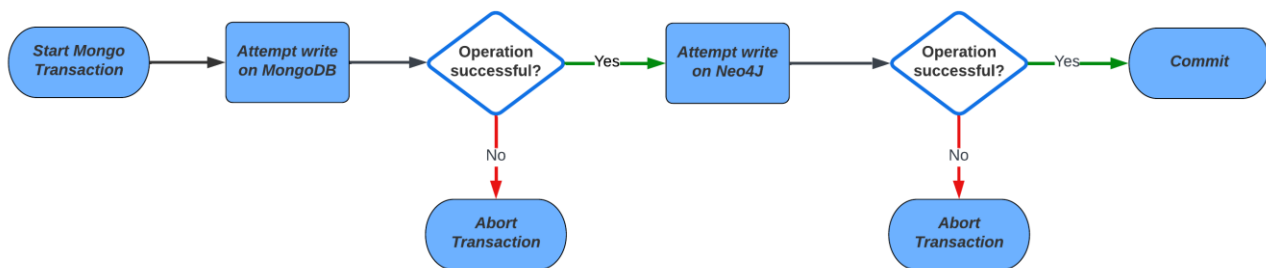
1. **Moderator adds a new game:** this operation requires adding the game to both Mongo and Neo4J databases. To ensure consistency, the game is first inserted into Mongo, followed by Neo4J via the Consistency Thread. The update of the **consistent** flag in the games collection in Mongo follows the Neo4J insertion. This approach enables us to exclude games that have not yet achieved consistency from search results. Games are shown to the user in the search section only if they are consistent across the databases. This ensures that features like "Add To Library" and "Add To Wishlist" function correctly, avoiding errors that could arise if a game's node is not yet present in Neo4J. However, should Neo4J be unavailable, a synchronization feature allows moderators to queue games for reinsertion to Neo4J, addressing potential consistency flag issues.
2. **Synchronize games:** a moderator can use this feature in order to retrieve all the games marked with a "false" consistency flag - possibly due to a network error on Neo4J - and queue them up for insertion to Neo4J again. This feature is, again, implemented using eventual consistency mechanisms described above.
3. **Follow a user:** when a user follows another user, which is an operation that involves a write operation on Neo4J, it's also necessary to update the redundant *followers* and *following* fields in the **users** collection on Mongo. This redundancy avoids the need to query Neo4J for a user's follower count. Please note that in this case, updating the followers and following count involves counting all **follows** edges, rather than simply incrementing the counters. This is a direct consequence of our application's monolithic architecture, where queues to handle tasks for eventual consistency cannot be considered persistent. Should the database be unavailable when a user attempts a consistency-related write operation, there's a risk of losing this update permanently, especially if the user exits their client unexpectedly. Our approach, despite being more demanding, ensures reliability by allowing any subsequent user interaction to correct any previously missed consistency updates.

4. **Like or dislike a review:** similarly to the previous point, but for the redundant *dislikes* and *likes* fields in the reviews collection.
5. **Delete a review:** reviews are first removed from Mongo, then from Neo4J using eventual consistency mechanisms.
6. **Ban a user:** similarly to the previous point, we use eventual consistency mechanisms to clean up the banned user's node and relationships on Neo4J after a moderator bans them.

9.2.3 Strict consistency operations

Although most operations that involve both databases within our system are handled in eventual consistency, there is one exception. Specifically, for the registration use case and exclusively in this case, we have chosen to enforce strict consistency. This is because when a user signs up to the platform, we want to make sure that both databases are consistent. Our goal is to guarantee that all features are fully accessible to the user immediately after login. Having inconsistent data on Neo4J after user registration might cause some features not to be available to the user (i.e. all the features that require Neo4J-based queries). For this reason, in this specific case, we decided to give priority consistency over availability. Considering this is a one-time operation for each user, we think that this minor delay will not affect the user's perception of the app's responsiveness.

Strict consistency was achieved leveraging Mongo transactions.



The implementation of this operation can be found in the `RegUserServiceImpl.java` class file.

```

1. try (MongoClient mongoClient = BaseMongoDAO.beginConnection(true)) {
2.     try (ClientSession clientSession = mongoClient.startSession()) {
3.         clientSession.startTransaction();
4.         try {
5.             // User registration, collection users MongoDB
6.             registeredUser = registeredUserMongo.register(mongoClient, registeringUser,
clientSession);
7.             // Adding node to Neo4J
8.             registeredUserNeo.register(userRegistrationDTO.getUsername());
9.             clientSession.commitTransaction();
10.        } catch (DAOException ex) {
11.            clientSession.abortTransaction();
12.            throw new ConnectionException(ex);
13.        }
14.    }
15. }

```

The registration DAO performs the insertion using the `clientSession` object created by the service class, which essentially is responsible of orchestrating the strict consistency update across the two databases.

```

1. public RegisteredUser register(MongoClient mc, RegisteredUser u, ClientSession clientSession)
throws DAOException {
2.     MongoDBase db = mc.getDatabase("pixelindex");
3.     MongoCollection<Document> usersCollection = db.getCollection("users");

```

```

4.
5.     Document doc = new Document("username", u.getUsername())
6.         .append("hashedPassword", u.getHashedPassword())
7.         .append("dateOfBirth", u.getDateOfBirth())
8.         .append("email", u.getEmail())
9.         .append("name", u.getName())
10.        .append("surname", u.getSurname())
11.        .append("role", u.getRole());
12.    try {
13.        usersCollection.insertOne(clientSession, doc);
14.    } catch (MongoWriteException ex) {
15.        throw new DAOException("Already registered user [" + u.getUsername() + "]");
16.    } catch (MongoSocketException e) {
17.        throw new DAOException("Error in connecting to MongoDB");
18.    }
19.    return u;
20. }
21.

```

9.2.4 MongoDB write and read concern

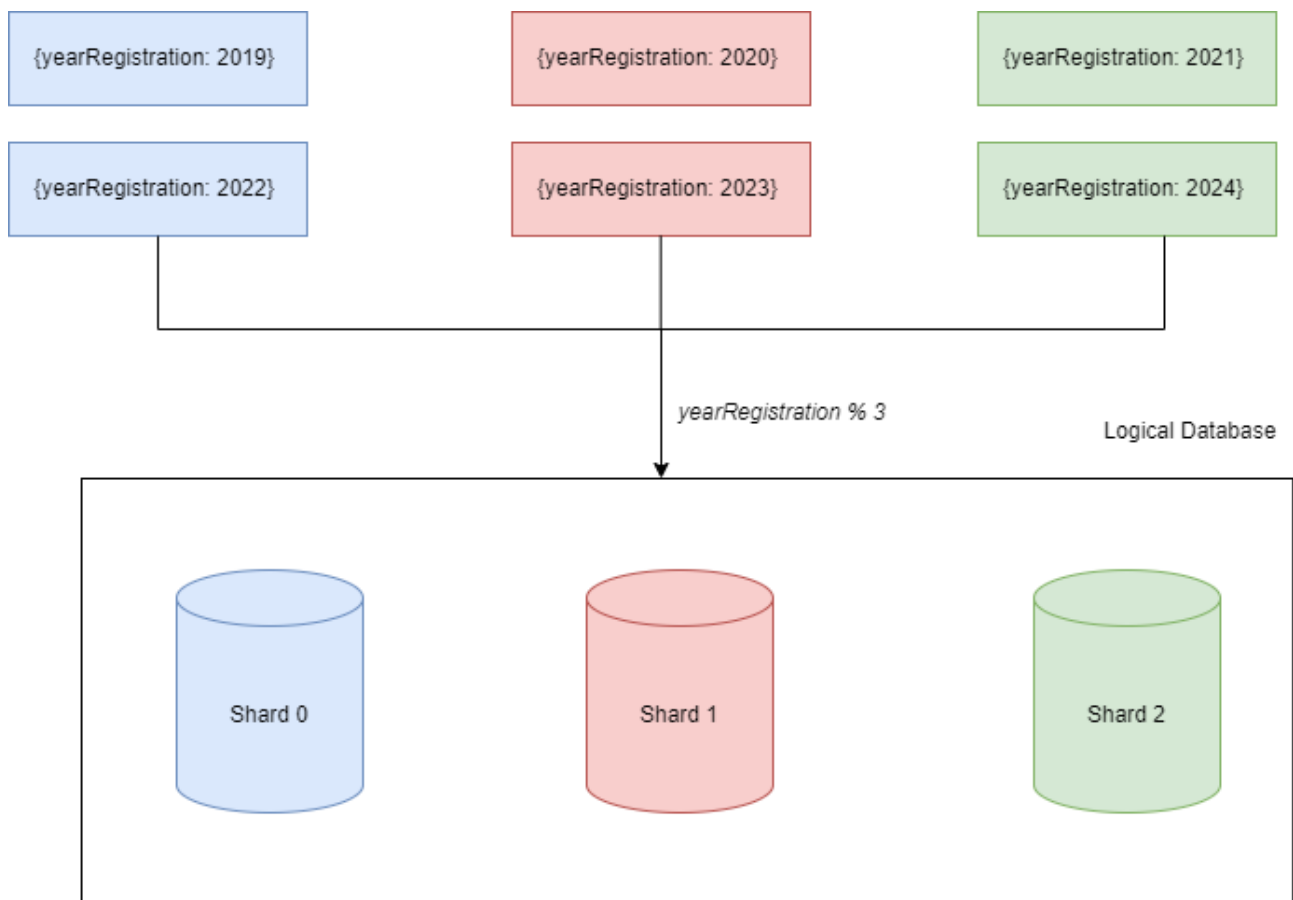
Regarding MongoDB's write concern, we chose a setting of $W=2$. This setting ensures that write operations are done on at least 2 replicas (primary included) before the applications regains control. Doing so, we can consider Mongo to be consistent. For some operations however, a write concern of $W=1$ might be sufficient, considering that Neo4J can serve as a first replica with consistent data. This would allow consistency update writes to be slightly faster. However, since consistency updates in MongoDB are managed by a secondary thread which is completely transparent to the user, we did not consider making this distinction as it doesn't affect the user's experience on the app. In contexts where performance is a critical factor, this change could instead be applied.

For read concern, in almost all our queries, it's not strictly necessary that the user always sees the most recently updated data. Being our application highly interactive, we decided to prioritize lower response times and availability. Consequently, we set Mongo's read concern to *nearest*, directing reads on the nearest Mongo replica to the user. The only exception to this rule is during the login phase, where we enforce a read concern of *primary preferred*. This makes sure that reads operation are directed towards the primary replica (if available), which contains their most recent write operations, such as registration. This is crucial to avoid login failures that could occur if a user's login attempt is directed to a replica that has not yet synchronized with the latest registration data, causing a self-induced denial of service for the final user.

10. Sharding organization

In order to decide whether or not to implement sharding in the design of our distributed database, we made a cost-benefit analysis. A possible analytics in MongoDB that would fit well with shards is the aggregation in which, given a specific year, we want to return for each month and for each age group the number of the users that joined the PixelIndex platform (§ [Chapter 8.1.5](#)). If during the design of distributed databases we were interested in load-balancing, an appropriate sharding key for performing the analytic described above would be:

```
shardingKey = year(registrationDate) % N
```

In this way, supposing for example that $N = 3$, the users registered in 2020 would refer to shard with id n.1, the users registered in 2021 would refer to shard with id n.2 and so on. Doing so, the load would be efficiently balanced while performing the above described analytics, but the costs to pay are too high. In particular:

- Since the **users** collection contains both the user's credential to perform the authentication and the registration date in which the user joined the platform, we would need to create a new ad-hoc collection just to perform the sharding. It would be no-sense to create sharded data starting from the complete user collection, otherwise we would need to scan all the shards to perform the login use-case. The new collection would contain: the username, the date of birth and the registration date. This will consume a higher amount of space in the persistent storage.
- Since we suppose that this analytics is not so frequently executed, we have decided to renounce in implementing any sharding strategy even because on the other aggregations we need to scan the complete collection and there would be the risk to scan all the servers hosting the shards.

11. Indexes and performance

To improve the queries of our application, we implemented these indexes:

11.1 Mongo DB

The performance stats were obtained by using the “Explain” functionality available in MongoDB Compass.

Collection	Field	Type	Where used	Performances with indexes	Performances without indexes
Games	name	text	[1]	executionTimeMillis: 16 totalKeysExamined: 569 totalDocsExamined: 500	executionTimeMills: 220 totalKeysExamined: 0 totalDocsExamined: 259290
Games	first_release_date	Ascending and Sparse	[1]	executionTimeMillis: 54 totalKeysExamined: 10652 totalDocsExamined: 10652	executionTimeMillis: 279 totalKeysExamined: 0 totalDocsExamined: 259290
Games	consistent	Ascending with partialFilterExpression {“consistent”:false}	CRUD operations	executionTimeMillis: 0 totalKeysExamined: 1 totalDocsExamined: 1	executionTimeMillis: 192 totalKeysExamined: 0 totalDocsExamined: 259290
Reviews	gameId ³	Ascending	CRUD operations, [2]	executionTimeMillis: 0 totalKeysExamined: 9 totalDocsExamined: 9	executionTimeMillis: 520 totalKeysExamined: 0 totalDocsExamined: 571006
Reviews	author ⁴	Ascending	CRUD operations	executionTimeMillis: 0 totalKeysExamined: 3 totalDocsExamined: 3	executionTimeMillis: 570 totalKeysExamined: 0 totalDocsExamined: 571006
Users	username	Ascending and Unique	CRUD operations	executionTimeMillis: 0 totalKeysExamined: 1 totalDocsExamined: 1	executionTimeMillis: 180 totalKeysExamined: 0 totalDocsExamined: 295684
Users	username	text	CRUD operations	executionTimeMillis: 1 totalKeysExamined: 11 totalDocsExamined: 11	executionTimeMillis: 213 totalKeysExamined: 0 totalDocsExamined: 295684

Comments on the most relevant indexes:

- **Games Name index:** The advanced search functionality allows users to retrieve games by name, even with partial name inputs. For this purpose, we used a text index on the game name. This precludes the possibility to directly compare the performance of this query with

³ This index was tested using a game that has a number of reviews equal to the rounded up average amount of reviews by game in our database.

⁴ This index was tested using a user that has a number of reviews equal to the rounded up average amount of reviews by user in our database.

and without index, because the usage of the text index is mandatory with the **\$text** and **\$search** operators used in the query. Therefore, we compared it with a query using the **\$regex** operator on the game's name. The regex-based query cannot take advantage of indexes, as it must perform a full collection scan to identify matching patterns, leading to much higher query times. Moreover, the regex-based query, which scans the whole collection, will take a longer time as the collection grows, differently from our query. Note that we didn't use a compound text index on all the text fields to keep searches specific to their fields. This ensures that a search query that targets the "company" or "platform" fields doesn't pull results from the "name" field and vice versa.

- **Games First Release Date Index:** This index significantly improves query execution times when the user decides to filter games by release year. It's necessary to note that the benefits of this and the previous index are lost if the user also provides a "company" or "platform" parameter, as they perform regex matching and therefore need to scan the whole collection. This only happens when a user wants to perform a more advanced and detailed search, which is a rarer event. This index is designed to be sparse as most games don't have this field, leading to a more compact index size.
- **Games Consistent Index:** This index is useful to quickly retrieve inconsistent games during the **synchronize games** query. We employed a `partialFilterExpression` to only index documents that have the consistent flag set to false, leading to minimal index size.
- **Users username index:** The same strategy applied to game name searches is also utilized for searching users by username. Similarly to the game search, a regex-based query was used for performance comparison.

11.2 Neo4j

To improve the read and write operations in Neo4j, the following indexes are defined:

Name	Label	Index type	Performances without index	Performances with index
index_adds_to_library_date	:ADDS_TO_LIBRARY (date)	RANGE	Response time: 7130 ms	Response time: 335 ms
			Page hits: ~5M	Page hits: 920k
constraint_user_username	User (mongoid)	CONSTRAINT	Response time: > 10000ms	Response time: 687 ms
				Page hits: 26 ms
constraint_game_mongoid	Game(mongoid)	CONSTRAINT	Response time [4]: 2420 ms Response time [5]: 290 ms	Response time [4]: 234 ms Response time [5]: 52 ms
			Page hits [4]: ~1.6M Page hits [5]: ~1.6M	Page hits [4]: 33 Page hits [5]: 13
constraint_review_mongoid	Review (mongoid)	CONSTRAINT	Response time: 1740 ms	Response time: 555 ms
			Page hits: ~4.3M	Page hits: 88

- The index `index_adds_to_library_date` is crucial: without indexing the property "date" on the `ADDS_TO_LIBRARY` relationship, the analytic query that displays the trending game chart of a specific year [\[3\]](#) would be very slow with respect to the same analytic performed without index.
- The index `constraint_user_username` is there for two reasons: we have to guarantee the uniqueness of a node for each username and the index is necessary when we have to perform the creation/updating/deletion of a `:FOLLOWS` relationship.
- The index `constraints_game_mongoid` is there mainly to speed up the time that the user has to wait when he/she adds a specific game to his/her library or his/her wishlist. [4]: Executions stats for the query that triggers the insertion of a game into a library. [5]: Executions stats for the query that triggers the insertion of a game into a wishlist.
- The index `constraint_review_mongoid` is there to guarantee better performances during the process in which the user edits his/her like preference on a review.

12. Application user manual

Our application is offered to the user through an intuitive Command Line Interface. Before starting the application, is worth to notice two aspects:

12.1 Starting the application

- The IP addresses and ports of the DBMSs are not hardcoded. This is to ensure a higher level of security. The application will be able to interact with the databases only if it finds a *well-formatted* .env file in the Java Maven project root.

~\ProjectLSMSDCMC\PixelIndexApp\PixelIndex				
Mode	LastWriteTime		Length	Name
da---1	15/02/2024	12:28		.idea
da---1	21/01/2024	15:50		src
d---1	14/02/2024	17:32		target
-a---1	15/02/2024	11:57	679	.env
-a---1	12/02/2024	00:58	564	.gitignore
-a---1	12/02/2024	12:06	127	debug.bat
-a---1	11/02/2024	23:44	58	execute.bat
-a---1	12/02/2024	12:07	3196	pom.xml

- This is what we mean by *well-formatted* .env file. Below you can see the expected structure for such configuration file:

```
1. ENVIRONMENT=PRODUCTION
2. MONGO_REPLICA_1_PROD=10.1.1.24
3. MONGO_REPLICA_2_PROD=10.1.1.23
4. MONGO_REPLICA_3_PROD=10.1.1.25
5. MONGO_REPLICA_1_PORT_PROD=27017
6. MONGO_REPLICA_2_PORT_PROD=27017
7. MONGO_REPLICA_3_PORT_PROD=27017
8. NEO_ADDRESS_PROD=10.1.1.24
9. NEO_PORT_PROD=7687
10. NEO_USER_PROD=neo4j
11. NEO_PASS_PROD=hvq6S3ELkEYCc4u
12. MONGO_REPLICA_1_TEST=127.0.0.1
13. MONGO_REPLICA_2_TEST=127.0.0.1
14. MONGO_REPLICA_3_TEST=127.0.0.1
15. MONGO_REPLICA_1_PORT_TEST=27018
16. MONGO_REPLICA_2_PORT_TEST=27019
17. MONGO_REPLICA_3_PORT_TEST=27020
18. MONGO_USER_TEST=myUserAdmin
19. MONGO_PASS_TEST=largescaleunipi
20. NEO_ADDRESS_TEST=127.0.0.1
21. NEO_PORT_TEST=7687
22. NEO_USER_TEST=neo4j
23. NEO_PASS_TEST=largescaleunipi
```

Let's focus on the first line. Our application expects the **ENVIRONMENT** variable to be set to **TESTING** or to **PRODUCTION**. If such variable is set to **TESTING**, then the application will use the IP addresses, ports, usernames and passwords that are specified from line 12 to line 23. Instead, if the **ENVIRONMENT** variable is set by the user to **PRODUCTION**, then the application will use the parameters from line 2 to line 11. In particular by looking at lines 2-11, it's possible to retrieve the credential we've used to connect to the University's VMs.

- Since our dataset has some non-ASCII printable characters (which however are part of the Unicode encoding) like emojis, accented letters and so on we need to switch encoding for

the terminal that we'll host our application. To do so, we prepared an *execute.bat* file in this way:

```
1. chcp 65001
2. java -jar .\target\PixelIndex-1.0-SNAPSHOT.jar
```

By launching the Java Virtual Machine through this batch file, we make sure that Unicode characters are well-prompted in the terminal even on Windows platform.

- After launching the application, the main menu will appear:



The “unregistered user” actor can use the up and down arrows on the keyboard to scroll among the options; by doing so the cursor will move up or down. Once the cursor is pointing the option, the choice can be confirmed by pressing the Enter key on the keyboard.

12.2 Unregistered user manual

12.2.1 Searching and browsing games



All the type of actors in our application can select the option “Search games” in order to start browsing the games. After pressing the “Search games” button, the application will block waiting for the user input: the user must specify at least a portion of the game title (simple search) or can specify also multiple criteria to match games (advanced search). The expected order of the

parameters by the application is also showed after the user has pressed the “Search games” button. Examples of searches are the following:



Fig: Example of simple search



Fig: Example of advanced search

After the confirmation of the search parameters, the application will allow the user to scroll among the list of games that are matching the query. These are the results of the first and second query respectively:



Fig: Results of the simple search

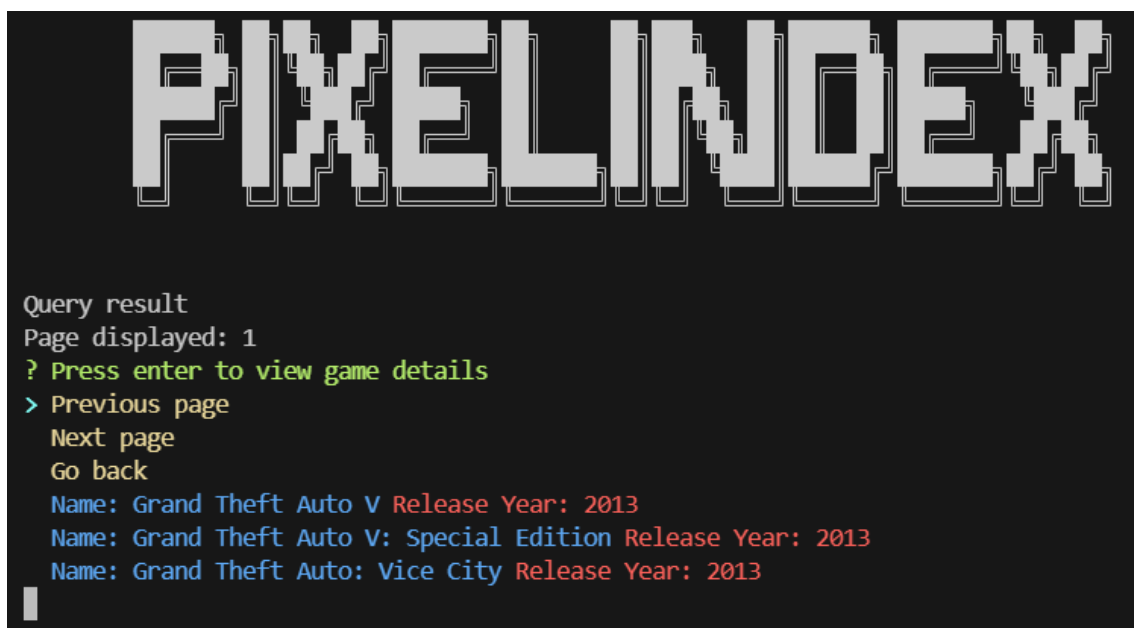


Fig: Results of the advanced search

Only the most relevant information of the game are displayed: the title and the year in which the game has been released.

If the search query returns more than 7 games, the user can scroll down with the arrows to view the remaining 3 games of the page. The user can then check if there are other games to display by pressing the "Next page" button; indeed, the page counter will display the new page number and

the next page will be displayed. If there are not more games to display, a message *****Empty***** will appear.

```
PIXELINDEX

Query result
Page displayed: 1
? Press enter to view game details
  Name: ABC Animal Adventures
  Name: ABC Coloring Town Release Year: 2016
  Name: ABC Flappy
  Name: ABC Follow Me: Animals Release Year: 2022
  Name: ABC Follow Me: Animals - Extended Edition
  Name: ABC Follow Me: Food Festival Release Year: 2022
  Name: ABC Hunt Release Year: 2017
  Name: ABC Match with Me Release Year: 2022
  Name: ABC Match with Me: Extended Edition
> Name: ABC Memory Match Release Year: 2018
```

Fig: CLI aspect when the user presses the “Arrow down” key to display the remaining three games of the page: the navigation buttons will be replaced by games.

```
PIXELINDEX

Query result
Page displayed: 2
*** Empty ***
? Press enter to view game details
> Previous page
  Next page
  Go back
```

Fig: CLI aspect when there are not more games to display

```
PIXELINDEX PIXELINDEX

Query result
Page displayed: 3
? Press enter to view game details
> Previous page
  Next page
  Go back
Name: ABC Sports Indy Racing Release Year: 1997
Name: ABC Sports Presents: The Palm Spring Open Release Year: 1991
Name: ABC Super Solitaire
Name: ABC Wide World of Sports Boxing Release Year: 1991
Name: ABC World
Name: ABC z Reksiem
Name: ABC's & 123's Release Year: 2001

Query result
Page displayed: 2
? Press enter to view game details
> Previous page
  Next page
  Go back
Name: ABC Monday Night Football Release Year: 1989
Name: ABC Monday Night Football '98 Release Year: 1997
Name: ABC Mysteriez: Hidden Letters
Name: ABC Nanpure Word-a-Pix
Name: ABC Paint Release Year: 2018
Name: ABC Preschool car truck and engine dot puzzles
Name: ABC Search With Me: Extended Edition Release Year: 2022
```

Fig: CLI aspect during page scrolling

At this point, pressing the button "Go back" the user will be redirected to the main menu. Otherwise, if the user points the cursor to a game name and then presses Enter, he/she will be redirected to the page showing the details of a game. If the user is not logged to the application and the game is age-restricted, a warning message will appear and will remind the user to authenticate.

```
PIXELINDEX

[Warning]: This game is age-restricted. Please login
Query result
Page displayed: 1
? Press enter to view game details
  Previous page
  Next page
  Go back
> Name: Grand Theft Auto V Release Year: 2013
  Name: Grand Theft Auto V: Special Edition Release Year: 2013
  Name: Grand Theft Auto: iFruit Release Year: 2013
  Name: Grand Theft Auto Online: Beach Bum Release Year: 2013
  Name: Grand Theft Auto: Vice City Release Year: 2013
```

12.2.2 Viewing game details

This is how the Command Line interface looks like in the game details page:

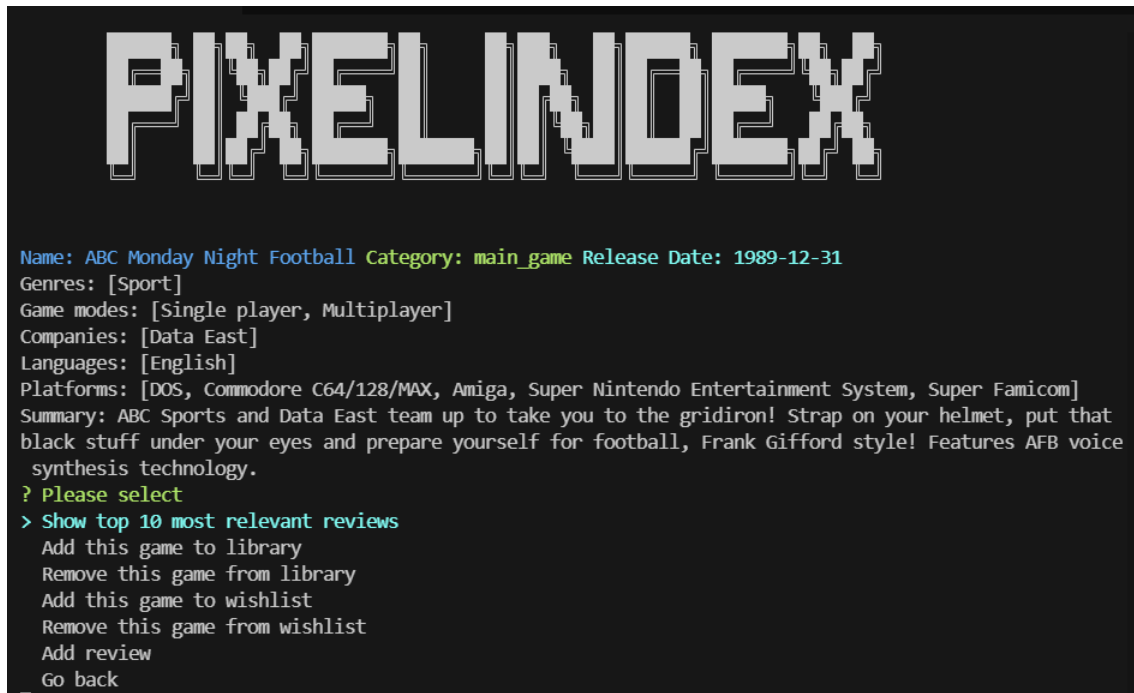


Fig: Game details page

At this point all the details of the game are displayed. In this page, the unregistered user has two options: 1) seeing the review excerpts, 2) going back to the result searches. Other functionalities are not available to unregistered users. If pressing the buttons offering such functionalities an error message will be displayed:

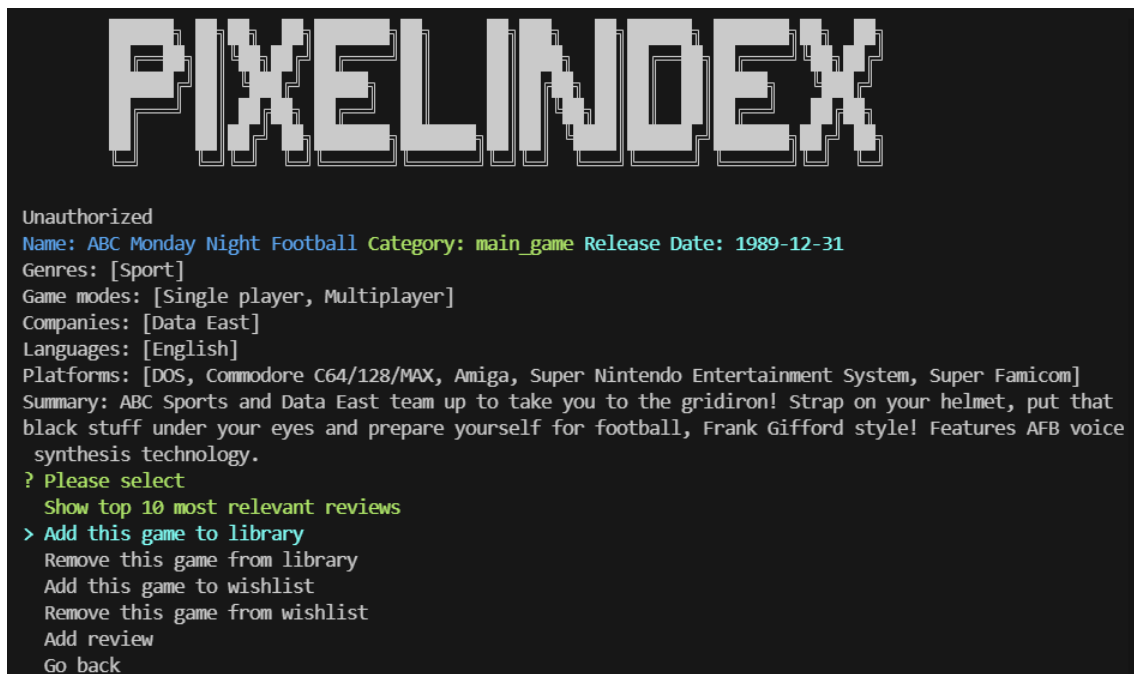


Fig: CLI showing the error message to an unregistered user

12.2.3 Browsing and viewing reviews excerpts

By pressing the button "Show top 10 most relevant reviews" the reviews excerpts will be displayed. As for the games, the user can scroll the excerpts by pressing the arrows keys, and the "Previous page" and "Next page" buttons.

```
PIXELINDEX

Name: 3D Puzzle: Japan Category: main_game Release Date: 2022-06-12
Genres: [Simulator, Strategy, Adventure, Indie]
Game modes: [Single player]
Languages: [Russian, Chinese (Simplified), Italian, Finnish, Spanish (Spain), Hungarian, Polish, French, Dutch, Danish, Korean, Portuguese (Brazil), Japanese, Swedish, Thai, Portuguese (Portugal), Arabic, Chinese (Traditional), Vietnamese, German, Ukrainian, English, Turkish, Czech, Norwegian]
Platforms: [PC (Microsoft Windows), Mac]
Summary: Collect a 3D puzzle, transferring things to the right places to create a beautiful house.
? Please select
> Show top 10 most relevant reviews
  Add this game to library
  Remove this game from library
  Add this game to wishlist
  Remove this game from wishlist
  Add review
  Go back
```

```
PIXELINDEX

Query result
Page displayed: 1 of 2
? Press enter to view review details
> Previous page
  Next page
  Go back
  Author: FurretTails Rating: RECOMMENDED Excerpt: graphics and controls aren't bad. enjoyable while ... Likes: 4 Dislikes: 0
  Author: Wispful Rating: RECOMMENDED Excerpt: Gives me dementia and loyalty to Imperial Japan Likes: 3 Dislikes: 1
  Author: Nytica Rating: NOT_AVAILABLE Excerpt: only 26 objects to put away... ridiculous Likes: 3 Dislikes: 0
  Author: Rolie27 Rating: RECOMMENDED Excerpt: Barry, 63 is already playing 3D-Puzzle - Japan, wi... Likes: 3 Dislikes: 0
  Author: Fourthrow Rating: RECOMMENDED Excerpt: Walking, grabbing and placing all with elegance. T... Likes: 2 Dislikes: 2
  Author: Tomi Rating: NOT_AVAILABLE Excerpt: Nice objects and not bad city, but the game mechan... Likes: 2 Dislikes: 0
  Author: Man Rating: RECOMMENDED Excerpt: Brought this game when it was on saleVery quick an... Likes: 2 Dislikes: 2
```

Fig: CLI displaying reviews excerpts

12.2.4 Viewing review details

To expand the review details, the user can move the cursor on the review excerpt and press Enter. The complete text of the review will appear.

```
PIXELINDEX

By: Rolie27 Rating: RECOMMENDED
Posted date: 2023-07-09T02:00
Likes: 3 Dislikes: 0
«Barry, 63 is already playing 3D-Puzzle - Japan, will you do it too?»
? Add a reaction if you want
> Go back
  Add like
  Add dislike
  Remove review
```

Fig: CLI displaying the review details

As unregistered user, at this point the user can only press "Go back" to return to the review's excerpts.

12.2.5 Displaying the most active reviewers

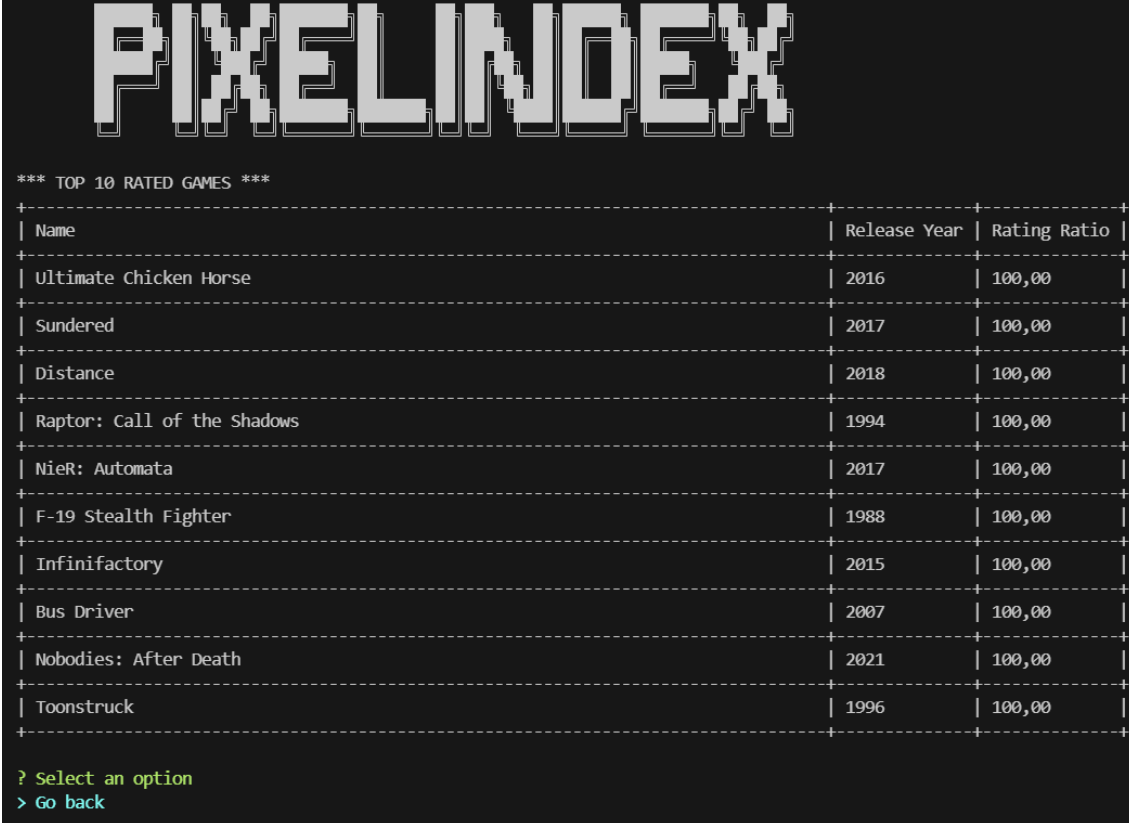
```
*** TOP 10 REVIEWERS OF LAST MONTH ***
+-----+-----+
| Username | Num of Reviews |
+-----+-----+
| DM       | 418             |
+-----+-----+
| ANNOYING CROW | 119           |
+-----+-----+
| LXROMAGE  | 117             |
+-----+-----+
| King di Bonita | 112          |
+-----+-----+
| Chang Liu | 108             |
+-----+-----+
| Bolo      | 105             |
+-----+-----+
| caio      | 104             |
+-----+-----+
| しのひ刀乱 | 95              |
+-----+-----+
| Mike      | 90              |
+-----+-----+
| Kane      | 89              |
+-----+-----+

? Select an option
> Go back
█
```

To see the top 10 reviewers of last month, the user has to select “Most active reviewers” in the main menu

12.2.6 Displaying the top ten rated games

By pressing the button “Top rated games” in the main menu, an ASCII-art table with the top 10 rated games of all time will appear displaying for each row the name of the game, the release year and the rating ratio (computed as explained in Chapter 3).



```
PIXELINDEX

*** TOP 10 RATED GAMES ***

+-----+-----+-----+
| Name                                     | Release Year | Rating Ratio |
+-----+-----+-----+
| Ultimate Chicken Horse                 | 2016         | 100,00       |
+-----+-----+-----+
| Sundered                               | 2017         | 100,00       |
+-----+-----+-----+
| Distance                               | 2018         | 100,00       |
+-----+-----+-----+
| Raptor: Call of the Shadows             | 1994         | 100,00       |
+-----+-----+-----+
| NieR: Automata                         | 2017         | 100,00       |
+-----+-----+-----+
| F-19 Stealth Fighter                   | 1988         | 100,00       |
+-----+-----+-----+
| Infinifactory                          | 2015         | 100,00       |
+-----+-----+-----+
| Bus Driver                             | 2007         | 100,00       |
+-----+-----+-----+
| Nobodies: After Death                  | 2021         | 100,00       |
+-----+-----+-----+
| Toonstruck                             | 1996         | 100,00       |
+-----+-----+-----+

? Select an option
> Go back
```

Name	Release Year	Rating Ratio
Ultimate Chicken Horse	2016	100,00
Sundered	2017	100,00
Distance	2018	100,00
Raptor: Call of the Shadows	1994	100,00
NieR: Automata	2017	100,00
F-19 Stealth Fighter	1988	100,00
Infinifactory	2015	100,00
Bus Driver	2007	100,00
Nobodies: After Death	2021	100,00
Toonstruck	1996	100,00

12.2.7 Displaying the trending games chart

After pressing the button “Trending games chart”, a form in which the user can insert the year will appear. Then, after the user confirms the year, this ASCII-art table will show, for each game, how many users have added that game into their library in the specified year.

```
PIXELINDEX

Which year?
2021
*** TRENDING GAMES CHART ***

+-----+-----+
| Game Name                                | Players count |
+-----+-----+
| Grand Theft Auto V                       | 4402          |
+-----+-----+
| The Witcher 3: Wild Hunt                 | 3156          |
+-----+-----+
| Portal 2                                | 1547          |
+-----+-----+
| The Elder Scrolls V: Skyrim              | 1336          |
+-----+-----+
| God of War                              | 1260          |
+-----+-----+
| Red Dead Redemption 2                   | 993           |
+-----+-----+
| BioShock Infinite                      | 730           |
+-----+-----+
| Grand Theft Auto: San Andreas            | 679           |
+-----+-----+
| Assassin's Creed II                     | 658           |
+-----+-----+
| Half-Life 2                             | 625           |
+-----+-----+

? Make your choice
> Go back
█
```

12.2.8 Login

To perform the login, the user has to press the “Login” button. The application will ask to put first the username and then personal password ⁵to authenticate. Notice that the echo of the typed password is disabled.

```
PIXELINDEX

Username?
test01
Password?
```

⁵ A possible test account could be: username: “test01”, password: “test”

12.2.9 Registration

To register to the platform, the user must have to select the "Register" button. One by one, the user must specify the fields in the correct order:

A screenshot of a terminal window showing the registration process for the PIXELINDEX application. The title 'PIXELINDEX' is displayed in a large, stylized, pixelated font at the top. Below it, a series of prompts are shown, each followed by the user's input. The prompts and inputs are: 'username?' with input 'test06', 'Name?' with input 'Test', 'Surname?' with input 'Six', 'Email address?' with input 'test@six.it', 'Date of birth? YYYY-MM-DD' with input '2006-06-06', and 'Choose your password:' with a single character input. The terminal has a black background with white text.

```
PIXELINDEX
username?
test06
Name?
Test
Surname?
Six
Email address?
test@six.it
Date of birth? YYYY-MM-DD
2006-06-06
Choose your password:
█
```

12.2.10 Exiting the application

By pressing the button "Exit", the application will be closed.

12.3 Registered user manual

Once the user has performed authentication, the dedicated menu for the registered user will appear:

A screenshot of a terminal window showing the main menu for a registered user in the PIXELINDEX application. The title 'PIXELINDEX' is displayed in a large, stylized, pixelated font at the top. Below it, the text 'Welcome test01' is shown. A prompt '? Make your choice' is followed by a list of menu options: '> Search games', 'Search users', 'View your library', 'View your wishlist', 'Users you might follow', 'Most active reviewers', 'Top rated games', 'Trending games chart', 'Show suggested games', and 'Exit app'. The terminal has a black background with white text.

```
PIXELINDEX
Welcome test01
? Make your choice
> Search games
Search users
View your library
View your wishlist
Users you might follow
Most active reviewers
Top rated games
Trending games chart
Show suggested games
Exit app
█
```


12.3.1 Searching and browsing users

To search new users, the user must press the button "Search users". After that, the user has to insert the username of the user he/she wants to search (or a portion of that).



After pressing Enter key the first usernames matching the search will be displayed:



Even in this case usernames are showed in pages. To scroll the pages the user must press "Previous page" / "Next page" buttons. One the cursor points to a specific username, the user can press Enter to navigate possible options.

12.3.2 Editing the set of the followed users

By using the "Edit your follow" button, the user can follow or unfollow the other users.

```
Query result
Page displayed: 1
? Select an user: username: Tev followers: 48 following: 26
User selected: Tev
? Make your choice
> Edit your follow
  Report
  Go back
█
```

A message will appear on the screen describing the operation just performed; "Operation [Created]" if the action corresponds to a following operation or "Operation [Deleted]" if the action corresponded to the unfollow of a user:

```
[Operation]: created
Query result
Page displayed: 1
? Select an user:
  Previous page
  Next page
  Go back
> username: Tev followers: 48 following: 26
  username: Stev followers: 34 following: 26
  username: Tevi followers: 1 following: 29
  username: Stevo followers: 41 following: 22
  username: Tevon followers: 36 following: 25
  username: $teve followers: 28 following: 28
  username: Steve followers: 20 following: 29
```

```
[Operation]: deleted
Query result
Page displayed: 1
? Select an user:
  Previous page
  Next page
  Go back
> username: Tev followers: 49 following: 26
  username: Stev followers: 34 following: 26
  username: Tevi followers: 1 following: 29
  username: Stevo followers: 41 following: 22
  username: Tevon followers: 36 following: 25
  username: $teve followers: 28 following: 28
  username: Steve followers: 20 following: 29
```

The update of the follower count is not immediate since they are updated eventually.

12.3.3 Viewing library

To view the library, from the main menu a user must select the option "View your library". After pressing that button the library will be displayed:

```
PIXELINDEX

Your library:
Page displayed: 1
? Press enter to view game details
> Previous page
  Next page
  Go back
  Name: Aeroplane Blaster Release Year: 2021 Added Date: 2024-02-13
  Name: Fortnite Release Year: 2017 Added Date: 2024-02-13
```

Pressing on the game title, the user will be redirected to the game details and there he/she has the possibility to edit their library.

12.3.4 Adding a game to library

To add a game to library, the user must first search the game, then select it and then press on the button “Add this game to library”:

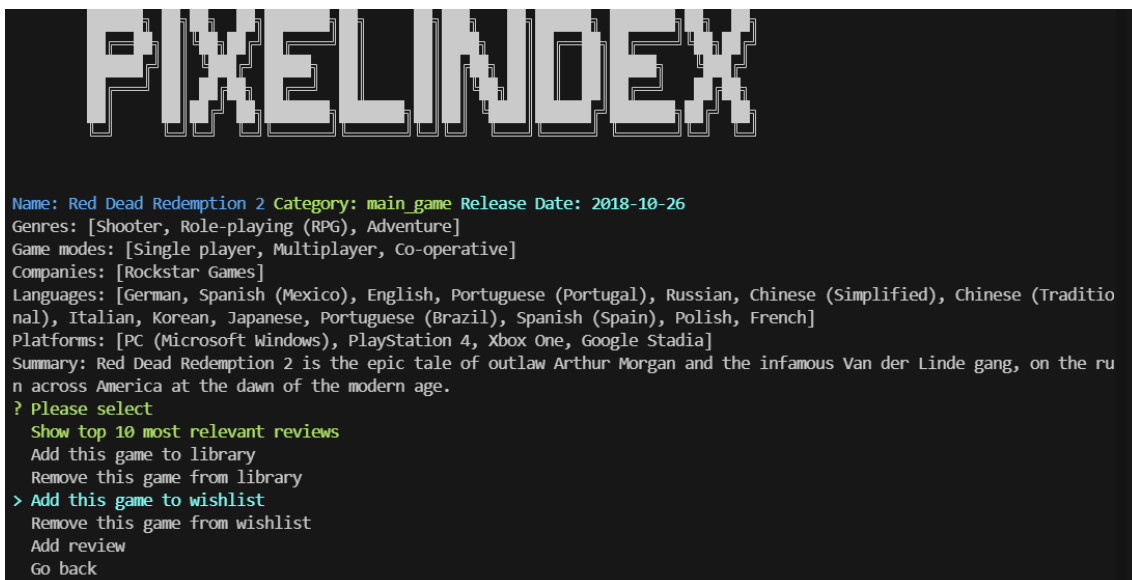


After pressing Enter, a confirmation message will appear, and the user will find the just added game into his library:

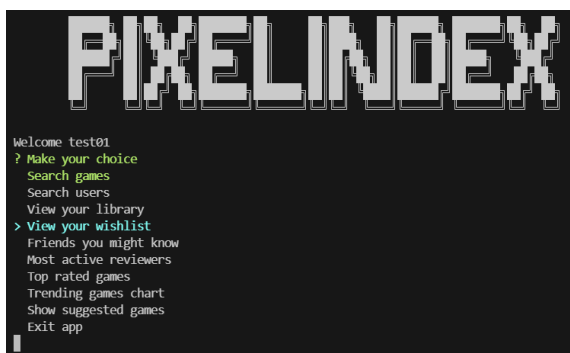


12.3.5 Adding a game to wishlist

In order to use the “Add to wishlist feature”, the user must use the corresponding option on a game details page.



A message on the screen will confirm that adding of the game the wishlist.



12.3.6 Removing a game from library

To remove a game from the library, the user has first to access his/her personal. Then he/she scrolls across the games and selects the game to remove. After that, he/she presses the button "Remove this game from library". After pressing the button, a confirmation message will appear:



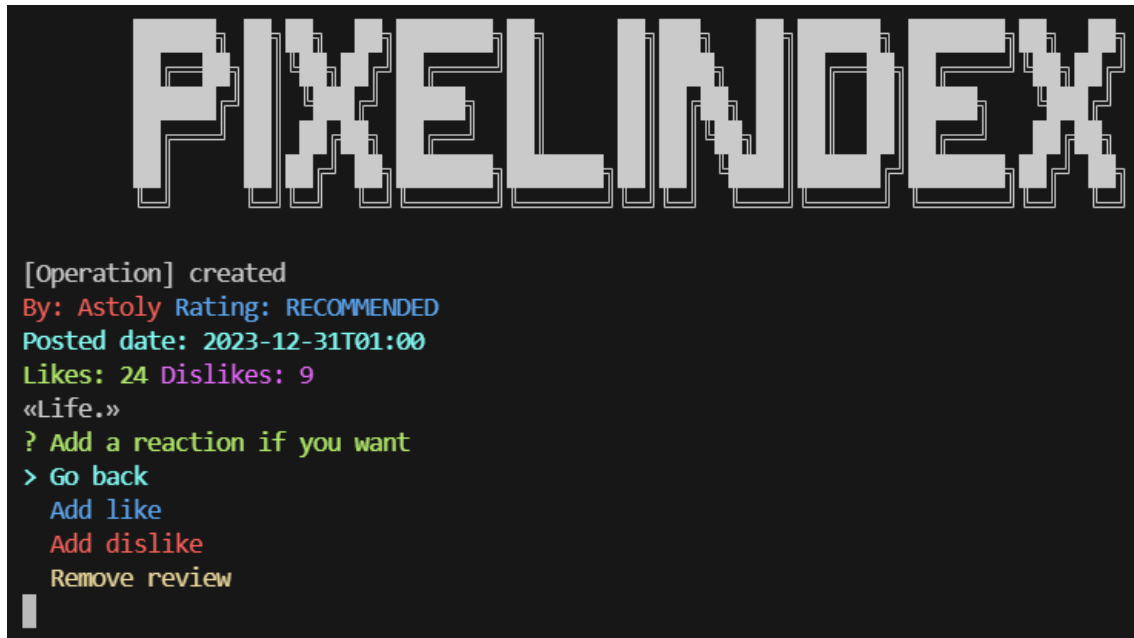
12.3.7 Removing a game from wishlist

To remove a game from the library, the user has first to access his/her personal. Then he/she scrolls across the games and selects the game to remove. After that, he/she presses the button "Remove this game from wishlist". After pressing the button, a confirmation message will appear:



12.3.8 Edit like preferences on a review

To put/remove like/dislike on a review the user will first search under the game's reviews: after searching for a game, the except it's displayed. The user will select the review on which adding the reaction and he/she will press "Add like" or "Add dislike" based on his/her preference.



A confirmation message will be displayed by the application. The possible confirmation messages are [Operation] created; [Operation] deleted or [Operation] updated. The first message is displayed after the adding of a new reaction, the second message is displayed when the user undoes the reaction he/she has previously put, the last one is displayed if the voting is changed from like to dislike or viceversa.

12.3.9 Adding a new review

To add a new review, the user must first reach the game details page, then he/she must click the button "Add review". A new dialogue will open: the application will first ask the overall opinion on the game, then the text of the review.



After pressing enter the confirmation message Review successfully added will appear.

12.3.10 Removing a review

In order to remove a review, the user must first browse the game's reviews and then he/she has the possibility to remove the review by clicking the "Remove Review" button.

```
Query result
Page displayed: 1 of 3
? Press enter to view review details
Previous page
Next page
Go back
> Author: test02 Rating: RECOMMENDED Excerpt: Good game. Likes: 0 Dislikes: 0
Author: crackfiend_charlie Rating: RECOMMENDED Excerpt: I will leave the cat he
Author: zorbarek Rating: RECOMMENDED Excerpt: Life. Likes: 26 Dislikes: 8
Author: Talonus Rating: NOT_AVAILABLE Excerpt: L game i cant play bc my steam i
Author: Astoly Rating: RECOMMENDED Excerpt: Life. Likes: 24 Dislikes: 10
Author: Sasquatch Genius 1980 Rating: NOT_AVAILABLE Excerpt: Great game besides
Author: cjbrow2011 Rating: NOT_AVAILABLE Excerpt: Recreated 9/11Good game Likes:
```

```
By: test02 Rating: RECOMMENDED
Posted date: 2024-02-15T20:21:09.705
Likes: 0 Dislikes: 0
«Good game.»
? Add a reaction if you want
Go back
Add like
Add dislike
> Remove review
```

12.3.11 Viewing suggested users and suggested games

To see the user follow suggestions, the user just needs to press the button "*Users you might follow*" on the main menu. A list of the ten suggested users will appear. Similarly can be done for the suggested games.

```
Users you might follow:
Ludwig Richter
CallMeShadow
Merlin
KraysS
cutest girl
Moyano
Molochniy
EGOIST
Coper
Laman
? Make your choice
> Go back
```

```
Games you might like:
Game name: Dead by Daylight Connections count: 1
Game name: Fallout: New Vegas Connections count: 1
Game name: Goosebumps: Dead of Night Connections count: 1
Game name: Tales of Zestiria: Additional Chapter - Alisha's Story Connections count: 1
Game name: Blades of Time: Dismal Swamp Connections count: 1
Game name: Wild West and Wizards Connections count: 1
Game name: Epic Conquest 2 Connections count: 1
Game name: The Good Life: Behind the Secret of Rainy Woods Connections count: 1
Game name: Succubus Hunter Connections count: 1
Game name: River City Girls Zero Connections count: 1

? Make your choice
> Go back
```

12.4 Moderator manual

To login as moderator, the moderator has to put special credentials⁶. After that, a dedicated menu will appear when pressing the button "Moderator area".

```
*** Special Area ***
? Make your choice
> View reports
  Add game
  Synchronize games
  User Registration Stats
  Go back
```

12.4.1 View reports and ban a user

To view the reports, the moderator simply presses on the button "View reports". A page with user's usernames and number of reports will appear. Then, to ban a specific user, after scrolling the usernames the moderator must select the user and use Enter to ban him/her.

```
Most reported users:
? Press enter to ban the user
  Go back
> User: test02 Number of reports: 1
```

12.4.2 Check registration stats

To view registration stats, the moderator will press the button "User Registration Stats". An ASCII-art table (after inserting the year to monitor) will be displayed showing how many users have registered to PixelIndex by each month.

```
? Make your choice User Registration Stats
Which year?
2023
*** REGISTRATION STATS: PIXELINDEX ***
+-----+-----+-----+-----+
| Month | < 18 y.o. | 18-30 y.o. | 30-50 y.o. | 50+ y.o. |
+-----+-----+-----+-----+
| 1      | 11         | 925        | 1436        | 1570      |
+-----+-----+-----+-----+
| 2      | 12         | 805        | 1311        | 1287      |
+-----+-----+-----+-----+
| 3      | 9          | 810        | 1371        | 1469      |
+-----+-----+-----+-----+
| 4      | 15         | 778        | 1265        | 1310      |
+-----+-----+-----+-----+
| 5      | 12         | 675        | 1131        | 1249      |
+-----+-----+-----+-----+
| 6      | 16         | 593        | 1034        | 1061      |
+-----+-----+-----+-----+
| 7      | 12         | 535        | 912         | 1004      |
+-----+-----+-----+-----+
| 8      | 15         | 476        | 719         | 777       |
+-----+-----+-----+-----+
| 9      | 7          | 95         | 136         | 158       |
+-----+-----+-----+-----+
| 10     | 12         | 53         | 61          | 73        |
+-----+-----+-----+-----+
| 11     | 12         | 42         | 52          | 79        |
+-----+-----+-----+-----+
| 12     | 12         | 36         | 79          | 62        |
+-----+-----+-----+-----+
? Select an option
> Go back
```

⁶ We suppose to have already a moderator account registered with such credentials: username **mod001**, password: **pippo**.