

GamerList Application

Large-scale and Multi-structure Databases



UNIVERSITÀ DI PISA

Edoardo Cogotti, Francesco Pesciatini, Anna Fabbri

17/02/2022

Table of Contents:

Introduction and Requirements	2
Functional Requirements	2
Non functional requirements	3
Functioning hypothesis	3
Specifications	3
Actors and use case diagram	3
Class Diagram	5
Classes definition	6
Design	10
Software Architecture	10
Dataset Organization	12
Data Distribution	13
MongoDB	15
Document Structure	15
Queries Analysis	17
Aggregations	19
Shards and Indexes	20
Sharding	20
Queries Configuration	21
MongoDB Indexes	21
Neo4j	23
Nodes	23
Relationships	23
Queries Analysis	24
Graphic-centric Query Translations	26
Aggregations	26
Indexes	27
Additional Details	27
Eventual Consistency Management	27
Java Implementation	28

Github repository: <https://github.com/EdoardoCogotti/GamerListApplication>

Introduction and Requirements

The developed application provides a service named GamerList whose core functionalities consist in collecting information and reviews about video games. A user needs to register an account to access GamerList, filling a form. If a user already has an account, he/she can insert his credential to access to the section of service according to his/her role. Once logged the Normal User can navigate the application. The Normal User is able to manually search for other users and view their profile, as well as searching for games and view their information and reviews with the possibility to leave his/her review. Normal users can follow and be followed by their friends, write reviews on games to express their opinion about them and add and remove in their list (named GamerList) their favorite games from the specific page of the game. Each Normal User can exploit the GamerList service to discover new possible friends or new appealing video games exploiting the network system and can also discover the most popular video games of each genre. An Admin has a special account that allows him/her to moderate the service. An Admin can publish games, examine analytics of each Normal User to have summarized information about him/her, moderate his/her content reviews if inappropriate and in serious cases ban him/her from the service. For further information it is recommended to read the provided User Manual.

Functional Requirements

- Admin must be able to perform the following operations:
 - Create and Update any Game
 - Create and Delete any Normal User
 - Update and Delete any review of a given Normal User
 - See the profile of an user, including personal information, GamerList and Review
 - Examine percentile information of a given user
- Normal User must be able to perform the following operations:
 - Follow and unfollow another Normal Users
 - Search for a Game by name
 - Search for an User by name
 - Add or delete a Game from their GamerList
 - Write a review to a Game
 - Edit and Delete his/her review to a Game, if already reviewed
 - View the detailed information of a game, including its reviews
 - View the profile of another Normal User
 - Browse a list of user that the application suggest you to follow
 - Browse a list of games that the application suggest you to add in your

GamerList

- View own percentile according to the review score
- View the best games of a selected genre

Both must be able to login into the system if they have an account, otherwise they must be able to register into the system. Each email must be associated up to a single account and the username must be unique

Non functional requirements

- Application must be usable and have an user-friendly Graphical User Interface
- Security through a registration and login process using credentials to identify the user and his/her role, with password encryption through SHA-256 and salt.
- Application must be always available through replicas in MongoDB and Neo4J
- Low response time (at maximum order of seconds)

Functioning hypothesis

- Disaster recovery is not taken into consideration
- Legal issues relative to the users ' data privacy such as anonymization are not taken into consideration

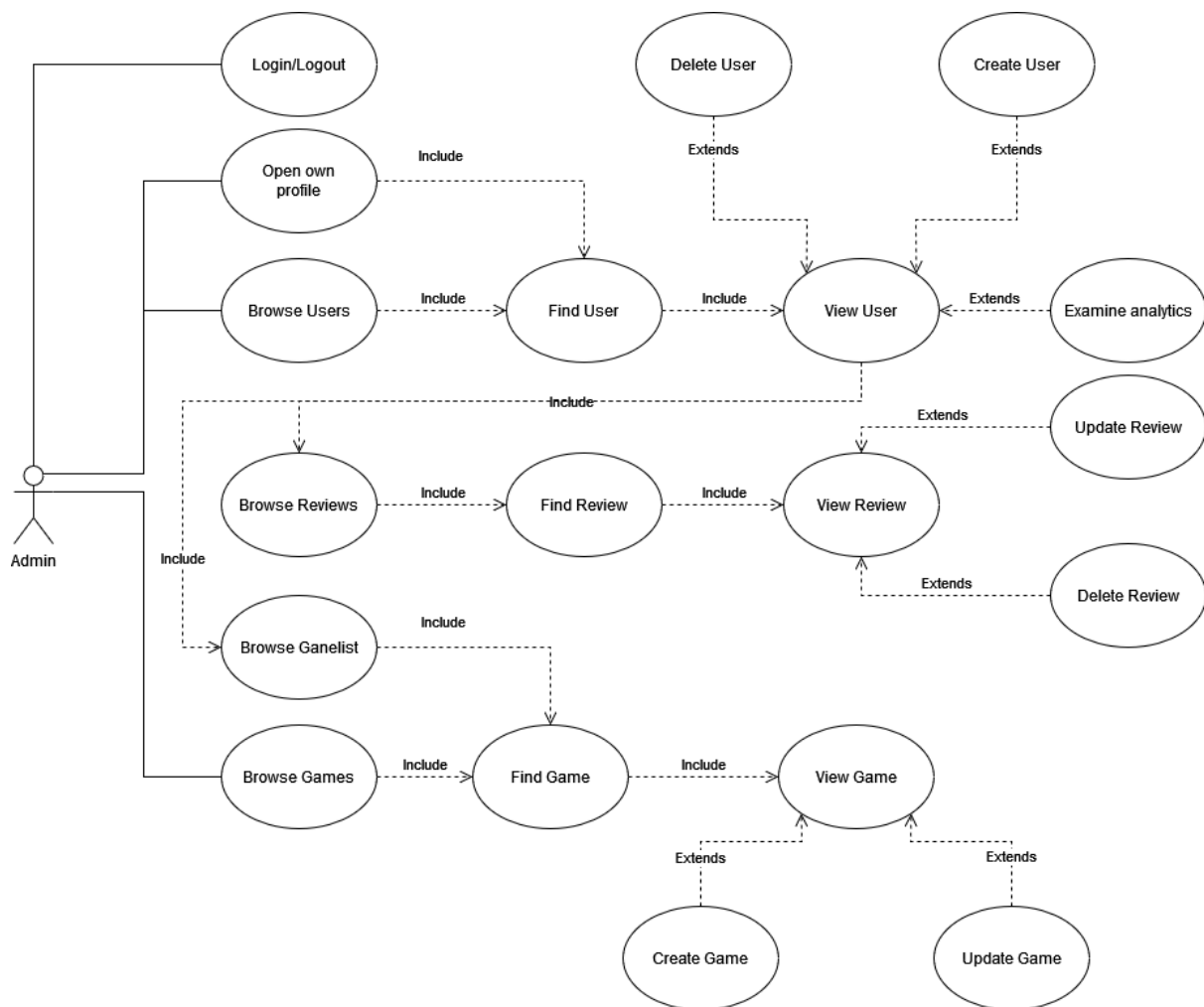
Specifications

Actors and use case diagram

The Gamerlist application is meant to be used by three kinds of actors that are the following:

- Unregistered users: they must be able to sign up to the service with a chosen unique username, password and others fields, or log in to the system if they are already registered;
- Registered users: they must be able to search, browse and view all games, browse and read all reviews and write their own positive or negative reviews about any game. Registered users are also able to add any game to their list of played games, or remove them from it. They can also browse the profiles of other registered users, follow or unfollow them. Users can also see basic statistics from their usage of the service, and get suggestions based on their activity;
- Admins: they must be able to search and browse all games, all users and their personal information (reviews included); since their job is to keep the platform a friendly place, admins must be able to edit and occasionally remove

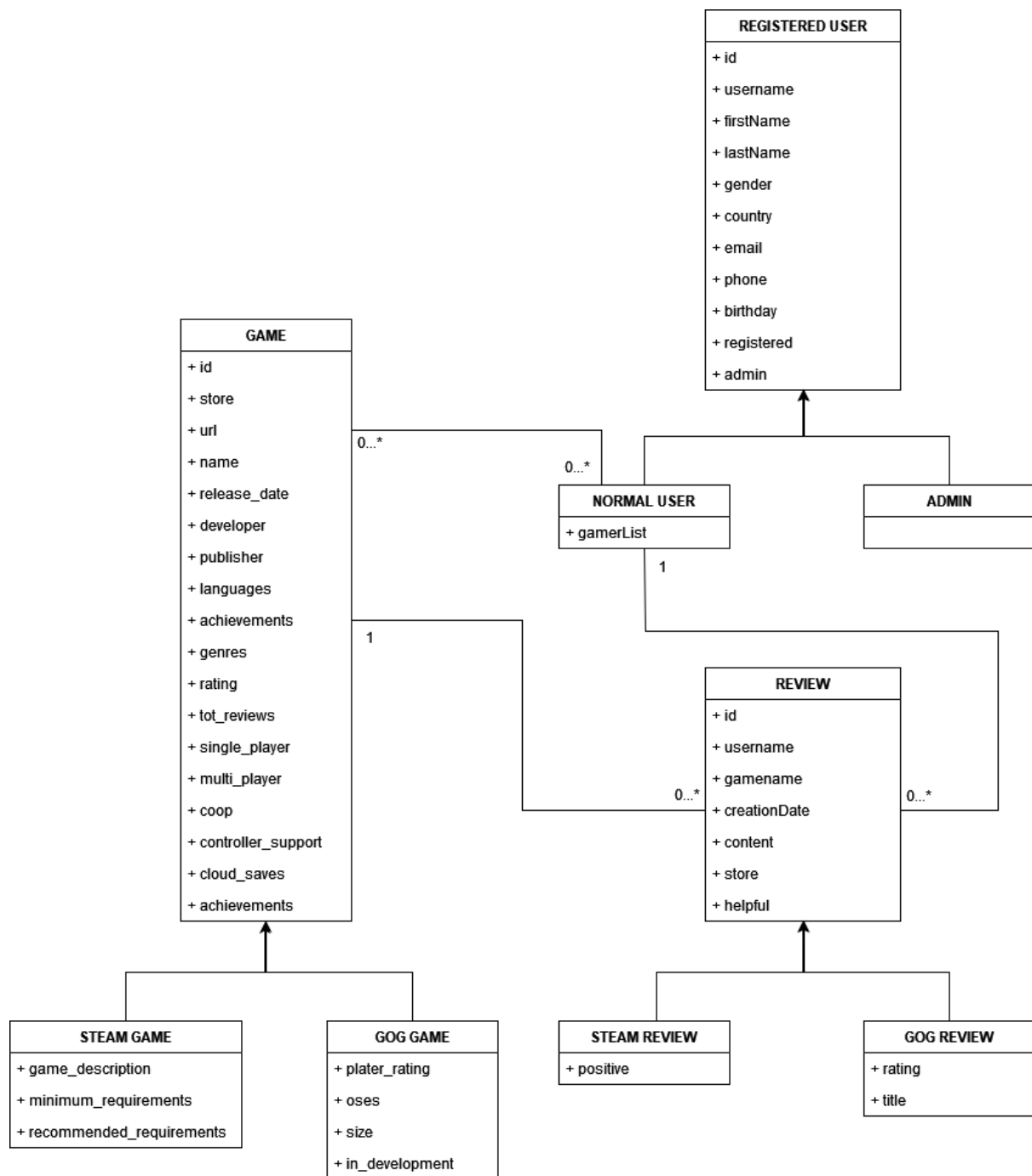
The admin use case diagram is the following one:



Admin performs different actions with respect to normal Users because they have special accounts only for management purposes.

Class Diagram

In the following diagram are displayed the main entities present in the application including their attributes and the relationships between them.



Classes definition

CLASS	DESCRIPTION
User	A user able to access to part of the functionalities of the application according to its role
Game	A game whose information is provided by the application, included its reviews
Review	A review of a game, consisting in a textual review and other information according to its type

Class Attribute - User

FIELD NAME	TYPE	DESCRIPTION
Id	int	Unique identifier inside the application
Username	String	Name chosen by the user in the signup phase. Unique inside the application.
firstName	String	First name of the user
lastName	String	Last name of the user
gender	String	Gender in which the user identifies himself/herself
country	String	Country where the user lives in
email	String	Email address specified by the user in the signup phase
phone	String	Phone specified by the user in the signup phase
birthday	LocalDate	Date field with the date of birth of the User
registered	LocalDate	Date field with the date of the registration of the User
admin	boolean	flag to indicate if the user is admin (true) or not (false)
gamerlist	List<GamerListElement>	List of favorite games of the User

Class Attribute - GamerlistElement

FIELD NAME	TYPE	DESCRIPTION
Name	String	Name of the game
Publisher	String	Publisher of the game
Developer	String	Developer of the game
FriendsCount	int	How many following people have the same game in their Gamerlist

Class Attribute - Review

FIELD NAME	TYPE	DESCRIPTION
Username	String	Username of the user that write the review
Gamename	String	name of the reviewed game
CreationDate	LocalDate	Date field with the date of when the review was written
Content	String	Review content
Store	String	Platform on which was written the review. It can be Store, GOG or Gamerlist
Helpful	int	How many people find the review useful
Rating (GOG)	int	Review score between 0 and 5
Title (GOG)	String	Title of the review
Positive (STEAM)	boolean	Flag to indicate if the overall review is positive (true) or not (false)

Class Attribute - Game

FIELD NAME	TYPE	DESCRIPTION
Id	ObjectId	Unique identifier inside the application
Store	String	Store on the game. It can be GOG or Steam
Url	String	Link to the main page of the game in its store
Name	String	Name of the game
Release_date	Date	Date field with the date of release of the game
Developer	String	Name of the developer
Publisher	String	Name of the publisher
Languages	List<String>	List of languages you can choose in the game
Achievements	int	Number of achievements in the game
Genres	List<String>	List of genres of the game
Rating	String	PEGI rating of the game
Tot_reviews	int	Total number of reviews in the game
SinglePlayer	boolean	Flag to indicate if the game is single player or not
MultiPlayer	boolean	Flag to indicate if the game is multi player or not
Coop	boolean	Flag to indicate if the game is coop or not
ControllerSupport	boolean	Flag to indicate if the game support the use of a controller or not
CloudSaves	boolean	Flag to indicate if the game support saves in cloud or not
Achievement	boolean	Flag to indicate if the game ha achievements or not
Reviews	List<ReviewCom pact>	List of reviews of the game in a summarized form
PlayerRating(GOG)	double	Average score of the game between 0 and 5 considering all the game reviews
Oses (GOG)	List<String>	List of supported oses for Windows, Linux and MacOS
Size (GOG)	String	Size of the game in terms of memory

inDevelopment(GOG)	boolean	Flag to indicate if the game was already released (false) or not (true)
gameDescription (STEAM)	String	Summarized description of the game
minimumRequirements (STEAM)	String	Requirement that a computer must satisfy to run the game
recommendedRequirements (STEAM)	String	Requirement that a computer must satisfy to run the game to get a better experience

Class Attribute - Review Compact

FIELD NAME	TYPE	DESCRIPTION
Platform	String	Platform on which was written the review. It can be Store, GOG or Gamerlist
Name	String	Username of the user that write the review
CreationDate	LocalDate	Date field with the date of when the review was written
Helpful	int	How many people find the review useful
Rating (GOG)	int	Review score between 0 and 5
Positive (STEAM)	boolean	Flag to indicate if the overall review is positive (true) or not (false)

Design

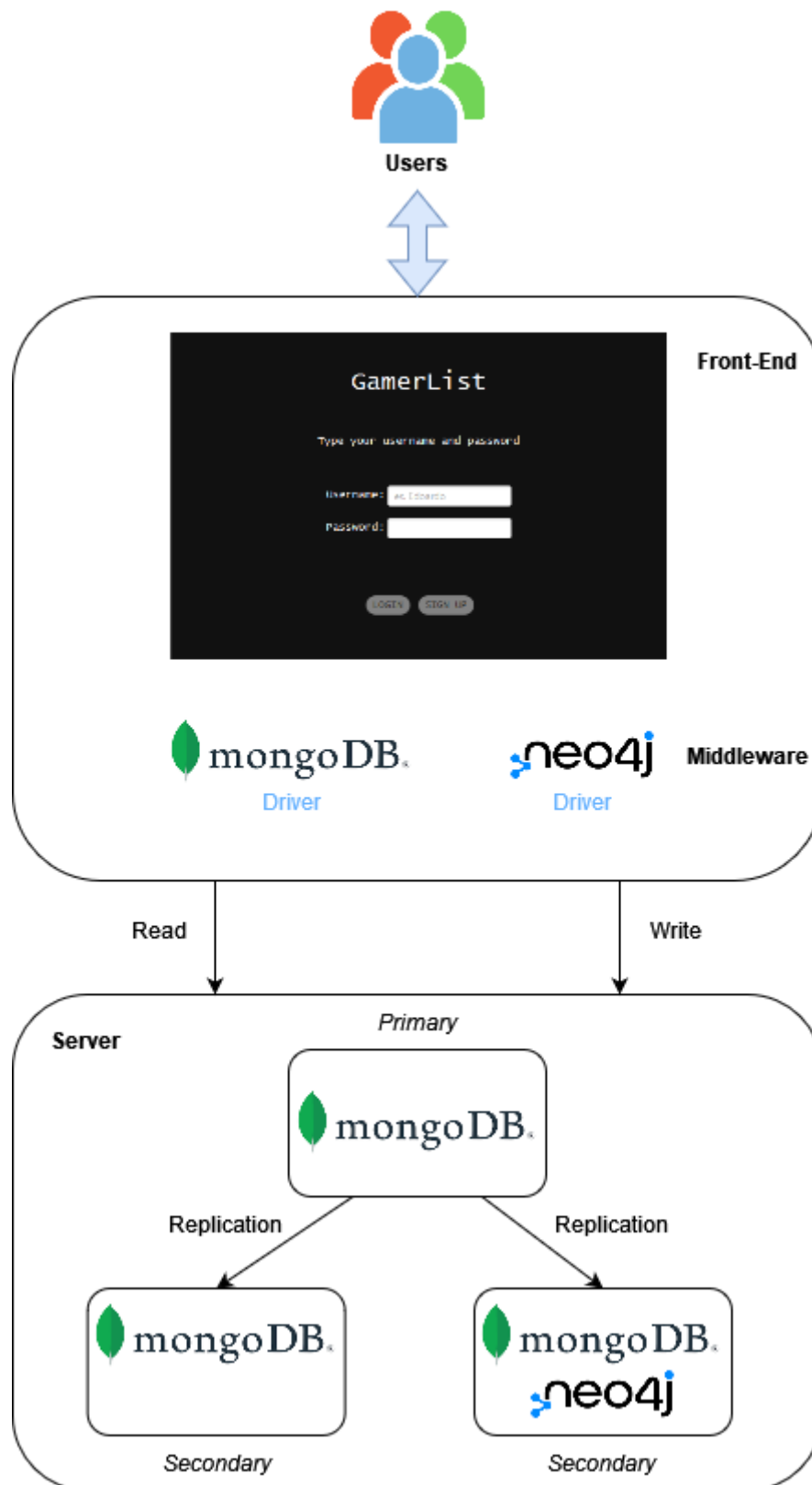
Software Architecture

In order to develop our application, we chose Java 11 as our programming language using a two-tier approach to develop a client-server application. The client provides to the user the interface to interact with the databases. On the server side there is a document database (a MongoDB replica set) and a graph database (Neo4J). In the details:

- On the Client side we have two modules:
 - Front-end: using the JavaFX framework and the visual tool Scene Builder to create the FXML document we develop the Graphical User Interface that is shown to the user allowing him/her to interact with the application functionalities. It's mainly composed by Controllers and fxml files (see the package structure)

- Middleware: in this module is possible to establish a communication between application and the database to send read or write requests to the document database using a MongoDriver and to the graph database using Neo4jDriver. The Front-end (in general Controller classes) handle this information.
- On the Server side data is stored in a MongoDB and a Neo4j database. The document database stores general information about users, games and reviews, while the graph database handles the relationship between user and between user and game. In detail the server side is a cluster of three virtual machines provided by the University of Pisa and the application accesses them using the openVPN tool. The cluster is composed of a single instance of Neo4j on just a virtual machine and MongoDB replicas. The structure is the following:

DATABASE	IP ADDRESS
MongoDB (primary)	172.16.4.77
MongoDB (secondary)	172.16.4.76
MongoDB (secondary) + Neo4j	172.16.4.75



Dataset Organization

The application uses a pseudo-real dataset.

- Games: the data regarding games was obtained by merging two different

Kaggle datasets representing information about games sold on two different websites, Steam and GOG; these services are real, known and used websites which offer to their large user bases a very wide range of video games, on sale in digital form.

Both these datasets had to endure a “pre-processing” process, in order to extract from them all the data that was deemed relevant to use inside the application without having to deal with the unnecessary details and unnecessary difference in data format. This allowed us to build an initial wide and realistic dataset of games;

- Reviews: all the data regarding reviews was obtained by using a dump on Zenodo.org of all the reviews made on Steam on each of the games that appeared in the previously cited Games dataset, in order to keep realism and consistency; Kaggle GOG games dataset already contained Reviews and they were extracted from it.
- Users: all the usernames were taken from the reviews’ dataset after a username censorship process, hereby assuring consistency within the three datasets: all the other data, regarding the anagraphics of the Users of the system, was generated automatically by a Python script using the RandomUser API.

Realism and consistency of the dataset is assured by the fact that all games were taken from real life game websites and reviews are written by real users - this gives also a perspective on which users have played and liked which games, as well as which genres were the most popular for each user and in general.

Data Distribution

In Gamerlist the information was splitted between MongoDB and Neo4j. In this application the Graph Database serves two main purposes: first of all, representing all the connections within the different elements of the platform - whether it be within different users or between the different kinds of connections between games and users - second of all, ensuring a quick access to all the relevant data.

In order to ensure a good performance, data is distributed in order to keep in the graph database only the essential information required to effectively represent the structures and relevant aspects of the relationships, and in order to be able to execute graph-specific queries and aggregations without request information to the document database. The connections between users and the information about the favorite genre are stored only in Neo4j.

On the other hand, the document database keeps almost the entirety of the application data.

USER
ID
Username
FirstName
LastName
Gender
Country
Email
Phone
Birthday
Registered
Admin
GamerList
Favourite Genre

GAME
Id
Store
URL
Name
Release Date
Developer
Publisher
Languages
Achievements
Genres
Rating
Tot Reviews
Single player
Multiplayer
Coop
Controller Support
Cloud Saves
Achievement
Reviews
Player Rating
Oses
Size
In Development
Description
Minimum Requirements
Recommended requirements

REVIEW
Username
Game name
Creation date
Content
Store
Helpful
Rating
Title
Positive

LEGENDA
Only on MONGO
on Both Databases
only on NEO4J

MongoDB

Document Structure

We use three collections in our MongoDB database: Games, Reviews and Users

Games

A typical Steam game documents is made like this:

```
1. _id:61fcfc0b1a245e0bb490a8b3
2. url:"https://store.steampowered.com/app/10/CounterStrike/"
3. store:"Steam"
4. name:"Counter-Strike"
5. release_date:2000-11-01T00:00:00.000+00:00
6. developer:"Valve"
7. publisher:"Valve,Valve"
8. game_details:Object
  1. single_player:false
  2. multi_player:false
  3. coop:false
  4. controller_support:false
  5. cloud_saves:false
  6. achievement:false
9. languages:Array
  1. 0:"English"
  2. 1:"French"
  3. 2:"German"
  4. 3:"Italian"
10. achievements:0
11. genres:Array
  1. 0:"Action"
12. rating:" Mature Content Description The developers
describe the content like ..."
13. game_description:" About This Game Play the world's number
1 online action game. Engage ..."
14. minimum_requirements:""
15. recommended_requirements:""
16. reviews:Array
17. tot_reviews:4499
```

The attribute “game_details” is an embedded document that contains all the boolean properties that both GOG and Steam games had in common.

A game extracted from the GOG store will miss the “game_description”, “minimum_requirements” and “recommended_requirements” fields, and instead will have this kinds of fields:

```
1. player_rating:3.6
```



```
2. oses:Array
3. size:"2.5 GB"
4. in_development:false
```

The array “**reviews**” instead contains a compact version of all the review relative to the game of the object:

```
1. creation_date:2010-02-14T00:00:00.000+00:00
2. helpful:47
3. positive:true
4. name:"md5crypt"
```

This is done to simplify queries we were supposed to perform in the Gamerlist application. In this way the games collections contains the needed information about its reviews without having to join games collection and review collection.

Each review element in the array is a compact version to avoid inserting the entire review in the collection that would easily overshoot the maximum size of the document. So we add and update only the fields that are strictly necessary for the queries. Creation_date, helpful and positive fields are used in the queries, while the “**name**” field allows to identify the review (this because each user can write only 1 review about a certain game)

Reviews

Each review document of Steam will contain this fields:

```
1. _id:61fcfc021a245e0bb4909913
2. game_name:"Counter-Strike"
3. username:"md5crypt"
4. creation_date:2010-02-14T00:00:00.000+00:00
5. store:"Steam"
6. helpful:47
7. positive:true
8. content:"This will be more of a 'my experience with this game' type of review..."
```

Here specific for the platform we only have the “**positive**” property, where instead a GOG review will have this attributes:

```
1. rating:4
2. title:"Live With It!"
```

Users

Unlike the other collections, all documents in “Users” are homogeneous, and here we

have a example the way in which the objects are formatted:

```
1. _id:62077fc625efc9145eb45ad6
2. username:"zzzz_tim"
3. cell:"0792-538-960"
4. state:"Bedfordshire"
5. city:"Hereford"
6. dob:1961-06-23T15:19:48.319+00:00
7. email:"lauren.long@example.com"
8. first_name:"Lauren"
9. last_name:"Long"
10. gender:"female"
11. pwd:"unique"
12. reg:2010-09-19T23:30:14.697+00:00
13. salt:"vwKZn07k"
14. sha256:"381baf0daf776513cdd78bf967046f6b36c6367282ec843"
15. role:false
16. gamerlist:Array
```

The property “gamerlist” is an array of objects, where each instance represents a game that the user has played. Each embedded document is a compacted version of it’s relative game object, and is formatted in this way:

```
1. name:"Warhammer 40,000: Gladius"
2. developer:"Proxy Studios"
3. publisher:"Slitherine Ltd."
```

Queries Analysis

The application uses the following queries to interact with the MongoDB database:

Query	Read or Write	Cost	Frequency
Add Review	Write	Low(add document and update relative game)	Medium
Delete Review	Write	Low (remove document)	Low
Update Review	Write	Low(update document and update relative game)	Medium
Get informations of a Review	Read	Low (read all fields of a review)	Low
Get a list of	Read	Medium (multiple reads)	High

review relative to a Game			
Get a list of reviews written by a User	Read	Medium (multiple reads)	Low
Get positive ranking of a User	Read	High (aggregation)	Low
Add Game	Write	Low (add document)	Medium
Delete Game	Write	Medium (delete game and all it's reviews)	Low
Update Game	Write	Low (update document)	Low
Get a list of games by a substring of the name	Read	Medium (multiple reads)	Medium
Get top 3 Games by genre by how much they are liked	Read	Very High (complex aggregation)	Low
Add User	Write	Low (add document)	Medium
Delete User	Write	Low (remove document)	Low
Update User	Write	Low (update document)	Low
Get User from its username	Read	Low (read all fields of a user)	High
Get a list of users from substring of the username	Read	Medium (multiple reads)	Low

The main assumption for this table is that considering the two kinds of users (Admin and Normal User) we have a low number of admins with respect to the numbers of the normal user. In addition to this, admin actions are about edge cases such as impolite users (reviews editing or user deletion) or arrival/errata corrige of games (game insert and update). So all the number of queries that are relative to the management of the platform (example: adding new games) will be a lower of the ones done by regular users that use the application (example: finding a game by a part of

its name)

Aggregations

The main aggregations where implemented in this way:

Query	Implementation
Get positive ranking of a User	<pre>db.reviews.aggregate([{ \$match: { creation_date: { \$gte: "one_year_ago_dates" } } }, { \$group: { _id: "\$username", helpful: { \$avg: "\$helpful" } } }, { \$sort: { helpful: 1 } }])</pre>
Get percentage of negative reviews of a user with a negative feedback (less than 10 reviews)	<pre>db.reviews.aggregate([{ \$match: { \$and: { creation_date: { \$gte: "five_years_ago_dates" }, username: "my_username", \$or: { \$and: { "positive": { \$exist: true }, "positive": false }, \$and: { "rating": { \$exist: true }, "rating": { \$lt: 3 } } } } } }, { \$group: { _id: "\$username", totReviews: { \$sum: 1 }, helpful: { \$push: "\$helpful" } } }, { \$unwind : "\$helpful" }, { \$match: { helpful: { \$lte: 10 } } }, { \$group: { _id: { username, "\$username" },</pre>

	<pre> totReviews: "\$totReviews", negReviews: { \$sum: 1 } }]) </pre>
Get top 3 Games by genre by how much they are liked	<pre> db.games.aggregate ([{ \$match: { \$in: { genres: "genre" }, { \$project : { name: 1, reviews.rating: 1, reviews.positive: 1, reviews.creation_date: 1, genres: 1, _id: 0 } } }, { \$unwind : "\$reviews" }, { \$match: { \$and: { reviews.creation_date: { \$gte: "one_year_ago_dates" }, \$or: { \$and: { "reviews.positive": { \$exist: true }, "reviews.positive": true }, \$and: { "reviews.rating": { \$exist: true }, "reviews.rating": { \$gte: 3 } } } } } }, { \$group: { _id: { "name": "\$name" }, totalPositiveReview: { \$sum: 1 } } }, { \$sort: { totalPositiveReview: -1 } }, { \$limit : 3 }]) </pre>

Shards and Indexes

Sharding

The main question is the following: is there any meaningful way to divide our dataset between different shards that would improve the access performance of our database? If we analyze our 3 collections we can see of this only the "User" can be a good candidate for being divided in different shards: by clustering the users by their state (that would represent the sharding key) the login and logout operations would be much more efficient to do, this because each user would have their data relatively close to them. Assuming that the user won't check other users accounts too often

(especially of different states), “User” alone would be a perfect candidate for Sharding. The main issue is that the other 2 collections (“Games” and “Reviews”) are not as easy to split. In both of them we have a distribution that follows the pareto principle, also known as 80-20 law. In the Gamerlist application we have 20% of the games that are really popular all over the world, and all the other 80% of the games that are not as famous. The “Review” situation has the same problem: some reviews are about popular games, some reviews not. But this case would be even worse, because we adopted a pagination structure like the result pages in Google and the user will probably check only the first pages of review, leaving the others unread or less read. With this in mind we can say that there is no meaningful way to split the database in independent shards that could improve the access performance for each collection

Queries Configuration

Following the query analysis and considering the non-functional requirements (especially the low response time requirement) of our application in order to have a better user experience both read and write requests on the replicat-set are configured to return after a single reading from one server. It's not mandatory to read the latest updates and is less important than low response time. For the same reasons the control to our application is returned after writing on one single server. We have implemented an eventual consistency paradigm: we set the nearest read and write in the configuration to minimize the operation time requesting the nearest replica.

MongoDB Indexes

In order to speed up the read operations on MongoDB we created two indexes to speed up the execution of our application

- In the Review collection we created an index structure according to the creation date of the reviews called `creationDateIndex`. It's very important to speed up the analytic queries. In many cases we are interested in aggregate considering only the reviews written in the last year. This match filter is a range query and `creationDateIndex` can speed up the execution.
- In the Review collection we have also created an index structure according to the game name called `gameNameIndex`. It's used in the main scene of our application, the Game Information Page. In this page all the reviews of a certain game are retrieved and so it's important to have a special data structure to access specifically to this block of reviews. The Review collection contains almost 4 millions of documents and accessing only a small subsection of it in a page so critical in terms of access frequency is fundamental.

We used the MongoDB Compass Explain Plan section to evaluate the query performance executed on the database. In the following pictures it is shown the

improvement obtained from each of these queries using this feature to compare the performance. On the top pictures we have the result obtained without using the index to query our document database, while on the bottom we have the result of the same query obtained using the index:

```
db.reviews.find({game_name:"Counter-Strike"}).explain("executionStats")
```

Query Performance Summary

Documents Returned: **12341**

Index Keys Examined: **0**

Documents Examined: **3690619**

Actual Query Execution Time (ms): **1960**

Sorted in Memory: **no**

⚠ No index available for this query.

Query Performance Summary

Documents Returned: **12341**


Index Keys Examined: **12341**

Documents Examined: **12341**

Actual Query Execution Time (ms): **14**

Sorted in Memory: **no**

Query used the following index:

game_name 

We can notice a dramatic improvement of the query.

```
db.reviews.find({creation_date:{$eq:ISODate('2021-02-05')}}).explain("executionStats")
```

Query Performance Summary

Documents Returned: **339272**

Index Keys Examined: **0**


Documents Examined: **3690619**

Actual Query Execution Time (ms): **2045**

Sorted in Memory: **no**

⚠ No index available for this query.

Query Performance Summary

Documents Returned: 339272	Actual Query Execution Time (ms): 876
Index Keys Examined: 339272	Sorted in Memory: no
Documents Examined: 339272	Query used the following index: creation_date 

In this case the index performance is less dramatic because the possible value of the creation date field is high. However in the considered range query we want to consider only the reviews of the last year and using the index MongoDB can access directly to the latest reviews instead of simply scanning all documents.

Neo4j

Nodes

The application's Graph database has two types of nodes.

- Users: the User nodes all the application's registered normal users, keeping track of their unique usernames and of their favorite game genre as attributes;
- Games: the Game nodes keep track of all the available games in the application. For each game node, the graph database stores an unique name and an array of the related genres.

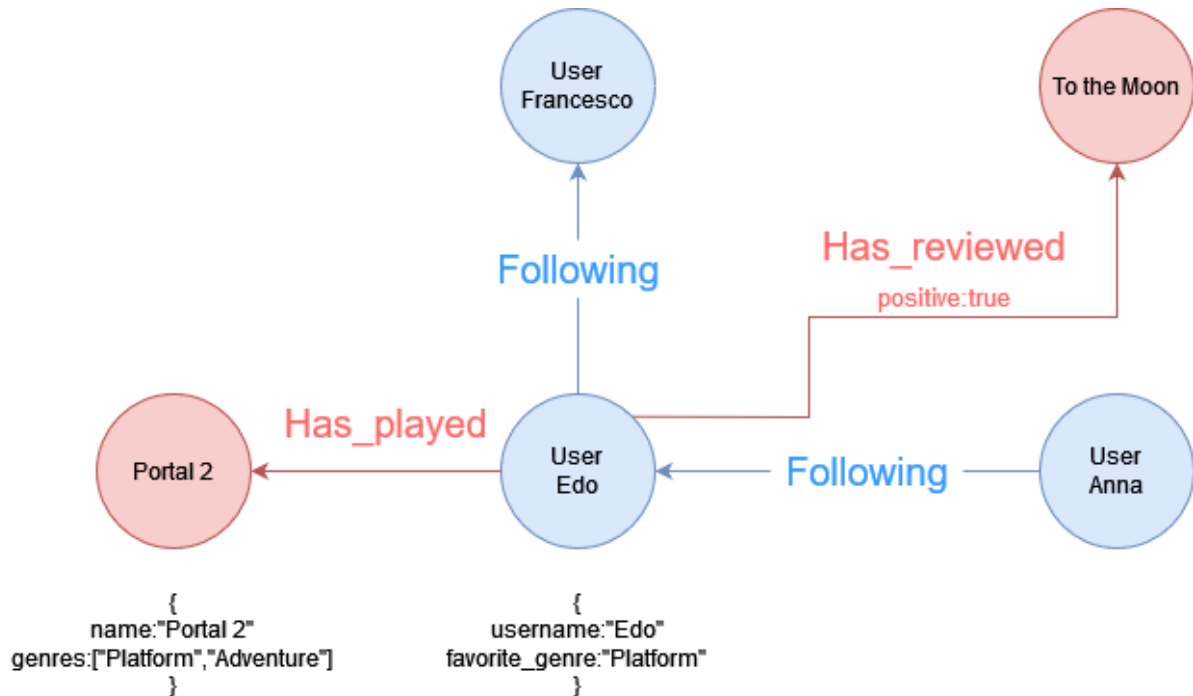
It's important to underline that in the graph database there are only Normal Users and not Admin. The graph database is used for suggestion analytic and operations about gamerlist, friends and reviews that the Admin doesn't perform.

Relationships

In the database, we have three types of relationships.

- User -> FOLLOWING -> User: this relationship represents the connections between users. This relationship has no attributes and it is created when an User follows another User;
- User -> HAS_PLAYED -> Game: this relationship represents the connections between an User and its list of played games within the application's pool of available games. This relationship has no attributes and it is created when an User adds a game to their list of played games;

- User -> HAS_REVIEWED -> Game: this relationship represents the connection between an User and a game they have reviewed. This relationship has a boolean attribute, “positive”, which indicates whether that review is positive or negative.



Queries Analysis

Operation	R/W	Frequency	Cost
Add User	W	AVERAGE	LOW (1 node added)
Add Game	W	AVERAGE	LOW (1 node added)
Delete User	W	LOW	AVERAGE (1 node removed, relationships removed)
Delete Game	W	LOW	AVERAGE (1 node removed, relationships removed)
Update User	W	LOW	LOW (update field)
Update game	W	LOW	LOW (update field)

Retrieve information about an User	R	AVERAGE	LOW (1 read)
Retrieve information about a Game	R	AVERAGE	LOW (1 read)
Follow/Unfollow User	W	AVERAGE	LOW (1 relationship added/removed)
Add/Remove Game from list	W	AVERAGE	LOW (1 relationship added/removed)
Add review	W	AVERAGE	LOW (1 relationship added)
Remove review	W	LOW	LOW (1 relationship removed)
Edit review	W	LOW	LOW (update field)
Check presence of follow	R	AVERAGE	LOW (1 read)
Check presence of review of game from user	R	AVERAGE	LOW (1 read)
Check presence of game in gamer list of user	R	AVERAGE	LOW (1 read)
Retrieve list of followed users	R	AVERAGE	AVERAGE (multiple adjacencies)
Retrieve list of followed users' followed users	R	AVERAGE	Average (multiple adjacencies)
Retrieve list of reviewed games for user	R	AVERAGE	AVERAGE (multiple adjacencies)
Retrieve list of played games for user	R	AVERAGE	AVERAGE (multiple adjacencies)
Retrieve list of reviews for a game	R	AVERAGE	AVERAGE (multiple adjacencies)

Retrieve number of users who played a game	R	AVERAGE	AVERAGE (multiple adjacencies)
Update most played genre by user	W	AVERAGE	AVERAGE (multiple adjacencies)
Suggest “followed by followed” users with the same most played genre game	R	AVERAGE	HIGH (complex aggregation)
Recommend not yet played games with more positive reviews of your followed users	R	AVERAGE	HIGH (complex aggregation)
For a specific game, indicate how many followed users play it	R	AVERAGE	AVERAGE (multiple adjacencies)

Graphic-centric Query Translations

Domain - specific queries	Graph - centric queries
For a specific game G1, indicate how many users U2 followed by user U1 play it	For a vertex U1, count all the nodes U2 that both have an incoming edge starting from U1 and an outgoing edge incident on G1.
Retrieve games G1 not yet played by user U1, with more positive reviews by users U2 followed by user U1	For a vertex U1, retrieve all the nodes U2 that have an incoming edge starting from U1. For each node U2, retrieve nodes G1 that have an outgoing edge incident on U2, and that do not have an outgoing edge incident on U1.
Retrieve users U3 followed by users U2 followed by users U1 with the same favorite genre as U1	For a vertex U1, retrieve all nodes U2 that have an incoming edge starting from U1. For each node U2 retrieve nodes U3 that have an incoming edge starting from U2.

Aggregations

Operation	CYPHER
Suggest “followed by followed” users	MATCH (a:User)-[:FOLLOWING*2]->(user)

with the same most played genre game	WHERE a.username = \$user AND NOT user.username = a.username AND user.favorite_genre = a.favorite_genre AND NOT (a)-[:FOLLOWING]->(user) RETURN DISTINCT user.username as username LIMIT 5
Recommend not yet played games with more positive reviews of your followed users	MATCH (a:User)-[:FOLLOWING]->(user)-[r:HAS_PLAYED]->(game) WHERE a.username = \$user AND a.favorite_genre IN game.genres AND r.positive = true AND NOT (a)-[:HAS_PLAYED]-(game) RETURN game.name as name, count(*) as occurrences ORDER BY occurrences DESC LIMIT 3
For a specific game, indicate how many followed users play it	MATCH (a:User)-[:FOLLOWING]-(user)-[:HAS_PLAYED]-(game) WHERE a.username = \$user AND game.name = \$game AND (a)-[:HAS_PLAYED]-(game) RETURN game.name as name, count(*) as occurrences

Indexes

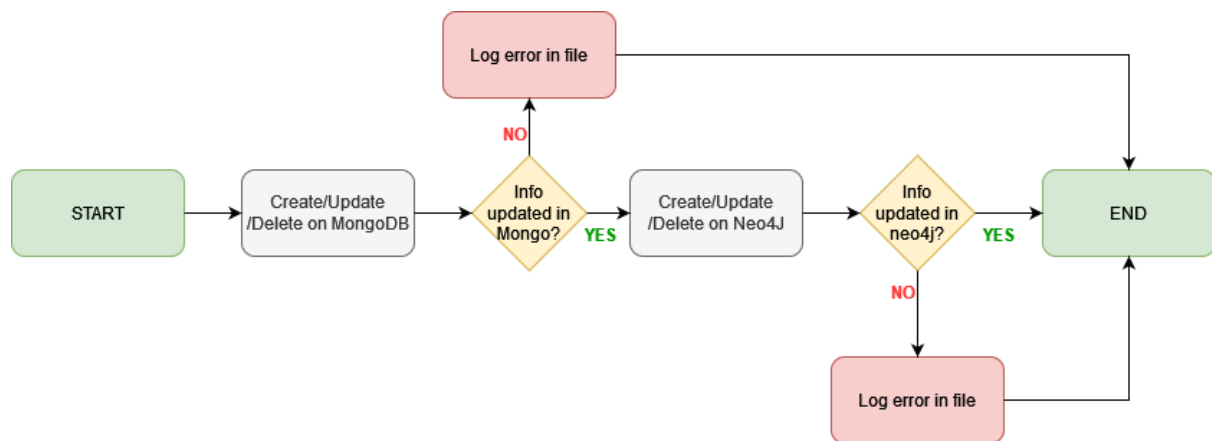
All kinds of operations performed on our graph database and all relationships (Following, Has_reviewd, has_played) are related and have as starting node an User node. Since that, we decided to introduce an index of the username in order to speed up the starting node search in a graph with a large number of User nodes.

```
CREATE INDEX usernameIndex FOR (n:User) ON (u.username)
```

Additional Details

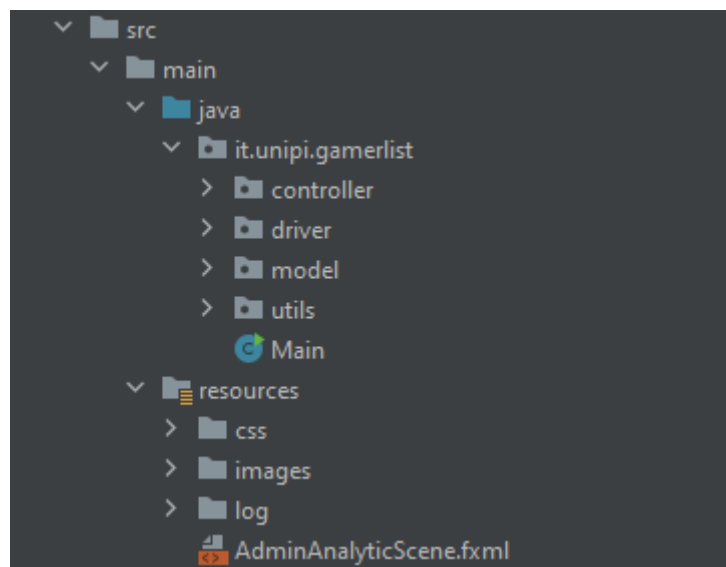
Eventual Consistency Management

Some fields in our application are present both in MongoDB and Neo4 and we want to avoid consistency errors when we create/update/delete them (write requests in general). It's a problem of cross-database consistency. In our application we apply a write request before on MongoDB, we return the control to the application and finally we apply that request on Neo4j. After the MongoDB action, but before the Neo4j action the databases are not consistent, but they will be at a certain point: we accept eventual consistency between them. Write actions that request both databases are written in a log file in order to keep information about them.



Java Implementation

To build our application we exploited Java 11. The structure of our project is the following:



Classes in the GamerList application are organized in packages. The main ones have the following naming convention:

- **controller**: this package includes all the controller classes. In JavaFX this kind of class describes logic and behavior of the events to link with the Graphical User Interface described in FXML and the model classes.
- **model**: this package includes all the classes that represent the application main entities (Game, Review, User). They include main fields and methods of a given entity and static methods to make requests about that entity to the databases. Additional classes are used to represent their complex fields (GamerListElement is used in User class and ReviewCompact in Game class).
- **driver**: this package includes the Database drivers for MongoDB and Neo4j (singleton pattern is adopted)

- `utils`: this package includes additional classes such as the `Session` class and the `UtilityMenu` class that handles the common menu between the different scenes

All these packages and the `Main` class are included in the package `it.unipi.gamerlist`. In order to build our Java application we adopted the Maven project management tool.