UNIVERSITÀ DI PISA

Large-scale and Multi-structured Databases
GameBase Application Report

Table of Contents

Introduction and Requirements

The application that was developed provides a service (named "*Gamebase*") whose core functionalities consist in collecting, organizing and presenting to its *users* information related to *videogames.*

The information stored for each game comprises its title, description, genre, release date, rating (which is influenced both by ratings external to the application and ratings of the users within the application) and so forth, and the users will be offered the opportunity to search and browse for games using different filters, such as by title and genre, and view detailed information on the desired games.

In order to enhance the user experience with the application, users will be allowed on the one hand to *rate* and *add* games to their favourite games list, and on the other to *follow* other users, allowing them to *view* their favourite games list, where suggestions will be presented to the users, both in terms of games to add to their favourites list and users to add to their followed list.

In addition, in order for the application to produce an income (or at least to ensure self-sustainability) on the one hand users will be offered a one-time payment option allowing them to perform more advanced queries on the dataset, whose results will be presented as *statistics*, a solution which should appeal to professional-oriented users such as data analysts, and on the other while viewing a game users will be presented, when applicable, with links to affiliated digital stores allowing them to purchase the game.

As a final feature, a dynamic scraping mechanism should be included in the application, allowing administrators to scrape videogames information from defined sources and adding such games to the existing dataset.

Functional Requirements

The following list summarizes the functional requirements of the application:

- The users of the application must be divided into three categories: the *Administrators*, which are allowed to perform any operation within the application, the *Analysts*, who are allowed to perform advanced queries on the *Games* and *Users* datasets whose results must be presented as statistics, and the *Standard Users*, who are allowed to use the rest of the functionalities of the application not requiring a higher level of privilege.
  Furthermore, access to the application is to be granted to registered users only through a login system based on the *username/password* model, through which the category of each user is also identified.
  Finally, a sign-up form allowing new users to register within the application as standard users must also be provided.

- The following functions must be offered to the *Administrators* of the application:
  - Delete a game from the *Games* dataset
  - Delete a user from the *Users* dataset
  - Update the *Games* dataset via a dynamic scraping mechanism
  - All other functions requiring a lower privilege level

- The following functions must be offered to the *Analysts* of the application:
  - View detailed statistics on the videogames of the *Games* dataset, such as the most viewed, most liked and best rated games, with the possibility of filtering results by parameters such as the game release year and genre.
  - View the most followed users within the application, who can be considered as *influencers*.

- The following functions must be offered to the application's standard users:
  - Browse the lists of most viewed, most liked and most recent games in the application, allowing the use of filters such as by game genre, etc…

- Browse a list of featured games for the user, representing suggestions for him to add to his own favourite games list
- Search for a game by title
- View the detailed information of a game, such as its title, description, release date, rating, etc...
- Follow links to affiliated digital stores allowing games to be purchased.
- Set or update his personal rating for a game
- Add/remove a game from their favourites list.
- Browse his favourite games list
- Browse his followed users list
- View the favourite games lists of the followed users
- Enter and update their personal information, including name, gender, age, favourite game genre, etc...
- Perform a one-time payment to upgrade their account to an analyst account.

Non-Functional Requirements

The non-functional requirements of the application are outlined below:

- A professional-grade quality of service (QoS) must be provided in terms of high availability, low latency, and tolerance to single points of failure, for which the application must be designed in order to attune to the *Availability* (A) and *Partition Tolerance* (P) vertices of the CAP triangle, and so to adopt the *Eventual Consistency* paradigm on its dataset, which must thus be replicated and sharded among different physical servers, allowing for load-balancing purposes and future-proofing a possible development towards a content distribution network (CDN) architecture.

- Since the success and profitability of the application strictly rely on the number of active users, in order to enhance the user experience on the one hand the videogames dataset should be composed of data of different natures, such as text, images, video, hyperlinks to related content, etc... (hypermedia), and on the other a modern structured graphical interface must be provided.

- In order to guarantee the *security* of the users' credentials, the users' passwords must be stored in an encrypted form within the database.

Working Hypotheses

The development of the application was also based on the following working hypotheses:

- The implementation details of how the payment relative to a user upgrading their account to an analyst account are processed are not taken into consideration.

- Existing regulations and concerns relative to the users' data privacy are not taken into account.

- Advanced aspects relative to data replication, such as load balancing and disaster recovery, are not taken into consideration.
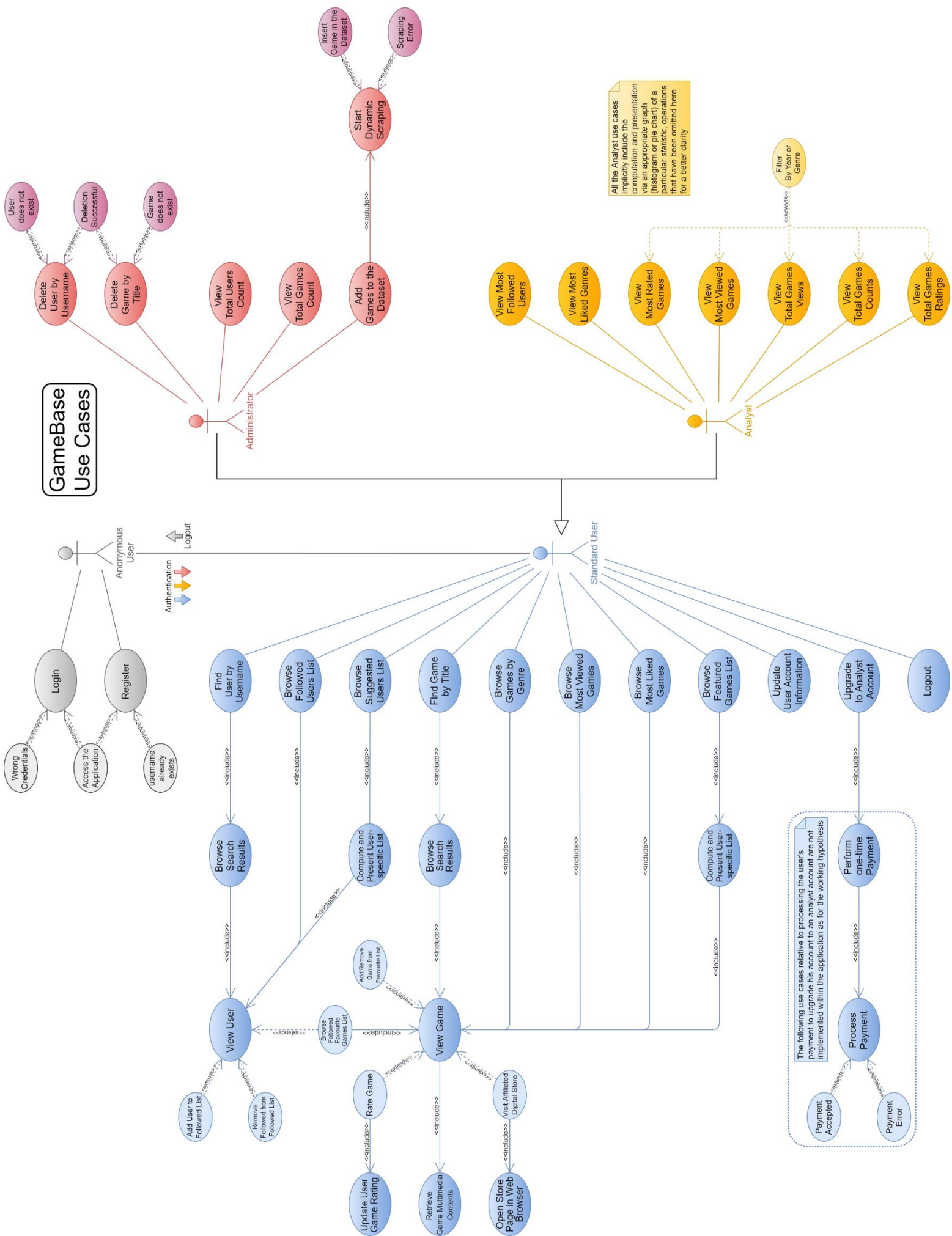
Specification

Actors and Use Cases Diagram

Based on the software functional requirements, the application is meant to be used by four actors:

- <u>Anonymous Users</u>, who are only allowed to log in or register within the application via a *username/password* system.

- <u>Standard Users</u>, who are allowed to interact with most of the application's features such as browse the list of most viewed, most liked and most recent games in the application, possibly by applying filters, browse a list of featured games, search for a game by title, view the detailed information on a game, follow links to affiliated digital stores, set or update the personal game rating, add/remove a game from their favourites list, browse their favourite games list, browse their followed users list, view the favourite games of followed users, update their personal information and make a one-time payment to upgrade their account to an analyst account.

- <u>Analysts</u>, who in addition to the features offered to standard users are also allowed to perform a set of advanced queries on the *games* and *users* datasets, whose results will be presented as statistics, such as viewing the most viewed, rated and favourite games, possibly filtering the results by parameters such as the game release year or genre, and viewing the most followed users within the application.

- <u>Administrators</u>, who are allowed to perform any operation in the application, including those requiring a high level of privilege such as deleting a user or a game from the database and adding new games to the *games* dataset by enabling/disabling a dynamic scraping mechanism.

The full set of actors with their related use cases is shown in the diagram below:

GameBase Use Cases

Task 2+3: GameBase Application Report   FEDERICO BALESTRI, NICOLA BARSANTI, RICCARDO BERTINI, MIRCO QUINTAVALLA

All the Analyst use cases implicitly include the computation and presentation via an appropriate graph (histogram or pie chart) of a particular *statistic* operations that have been omitted here for a better clarity

The following use cases relative to processing the user's payment to upgrade his account to an analyst account are not implemented within the application as for the working hypothesis

4

Analysis Classes Diagram

Based on the software's requirements and the additional working hypotheses the following analysis classes were identified within the application:
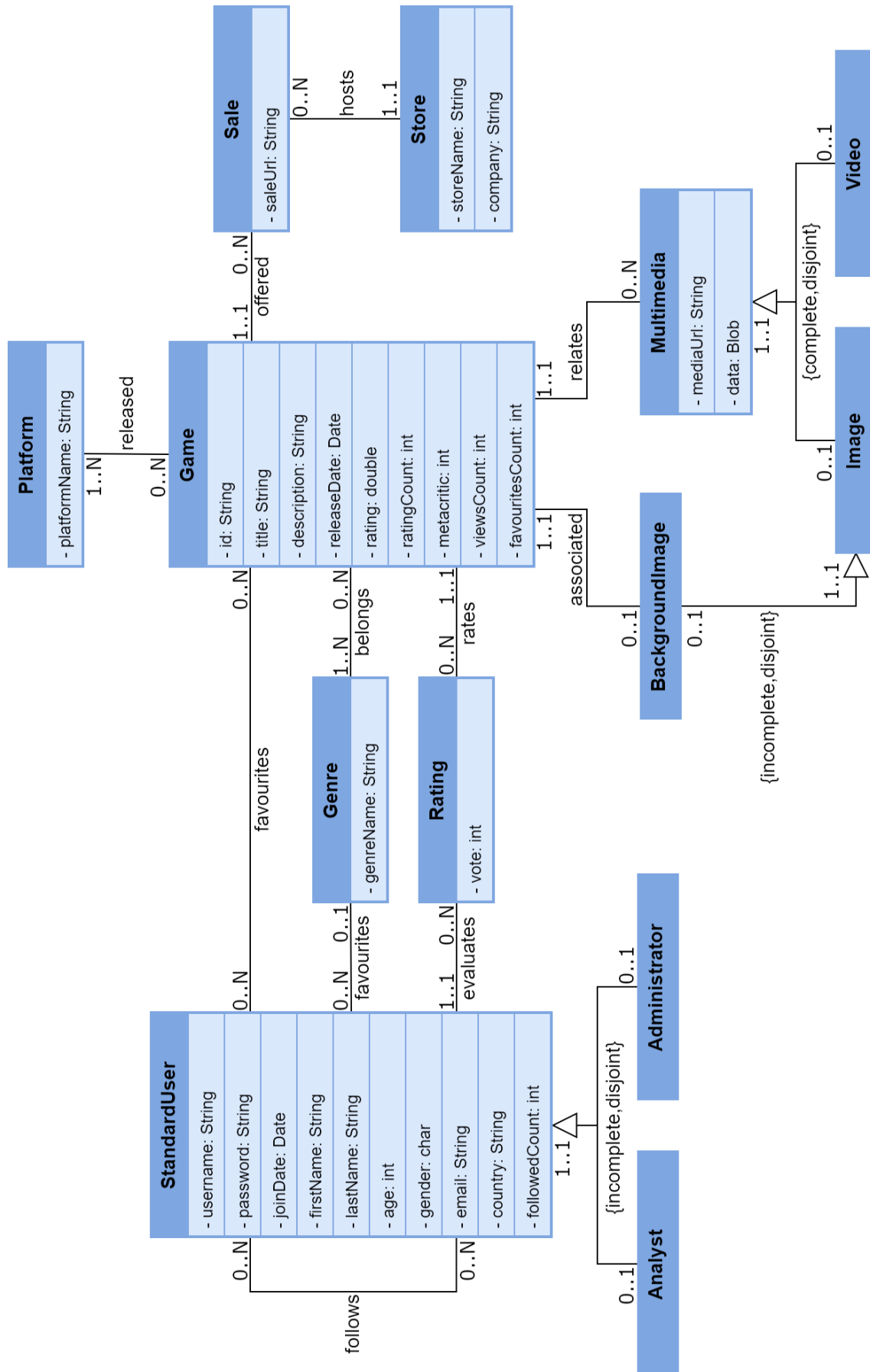
Classes Definitions

CLASS	DESCRIPTION
StandardUser	A user with no special privileges within the application, who is nevertheless allowed to interact with most of its functionalities
Analyst	A special user who is allowed to perform advanced queries on the *users* and *games* datasets, whose results will be presented as statistics
Administrator	A super user who is allowed to perform any operation in the application
Game	A videogame whose information is offered by the application
Genre	A videogame's genre
Rating	A user's evaluation of a game
Platform	A platform on which games are released
Multimedia	A multimedia content related to a game, which may consist in an image or a video clip
Image	An image related to a game
BackgroundImage	An image related to a game that can be used as a background in the application's graphical user interface
Video	A video clip related to a game
Store	An affiliated digital store where users can purchase games
Sale	A sale offer of a particular game in a specific store

Classes Attributes

StandardUser		
Attribute	**Type**	**Description**
username	String	A unique string identifying the user, which is also used by them to access the application
password	String	The password required for the user to access the application
joinDate	Date	The date the user registered within the application
firstName	String	The user's first name (optional)
lastName	String	The user's last name (optional)
age	int	The user's age (optional)
email	String	The user's email (optional)
gender	char	The user's gender (optional)
country	String	The user's country (optional)
followedCount	int	The number of the user's followers

Analyst		
Attribute	**Type**	**Description**
none (same as the superclass)		

Administrator		
Attribute	**Type**	**Description**
none (same as the superclass)		

Game		
Attribute	**Type**	**Description**
id	String	The game's unique identifier within the application
title	String	The game's title
description	String	The game's description
releaseDate	Date	The game's first release date
rating	double	The mean of all users' evaluations of the game, expressed as a decimal number between 1.0 and 5.0
ratingCount	int	The number of unique users' evaluations of the game
metacritic	int	The Metacritic® rating of the game, expressed as an integer between 0 and 100
viewsCount	int	The number of times the game was viewed by a user
favouritesCount	int	The number of users who have added the game to their favourites list

Genre		
Attribute	**Type**	**Description**
genreName	String	The name of the game's genre (e.g. Arcade, Puzzle, Shooting, ...)

Rating		
Attribute	**Type**	**Description**
vote	int	A user's evaluation of a specific game, expressed as an integer between 1 and 5

Platform		
Attribute	**Type**	**Description**
platformName	String	The name of the gaming platform (e.g. PC, XBOX, PS4, ...)

Multimedia		
Attribute	**Type**	**Description**
mediaUrl	String	The URL from which the multimedia content can be retrieved
data	Blob	The data representing the multimedia content

Image		
Attribute	**Type**	**Description**
none (same as the superclass)		

BackgroundImage		
Attribute	**Type**	**Description**
none (same as the superclass)		

Video		
Attribute	**Type**	**Description**
none (same as the superclass)		

Store		
Attribute	**Type**	**Description**
storeName	String	The affiliated digital store name (e.g. Steam, Xbox Store, …)
company	String	The store's owner company (e.g. Valve, Microsoft, Sony, …)

Sale		
Attribute	**Type**	**Description**
saleUrl	String	The URL of the page in a store where a specific game is sold

Architectural Design

Software Architecture

In terms of its architecture, the application has been divided into two tiers according to the *Client-Server* paradigm, where *Java 8* has been used as the core programming language:

Client Side

The client side is divided into:

- A *Front-end* module, consisting of a graphical user interface based on the *Swing* API, allowing users to interact with the application, and the access logic required to exchange data with the underlying middleware.
- A *Middleware* module, composed of three different modules and the logic required to bridge data between the modules and the front end.
  The main modules comprising the *Middleware* are:

    - A module handling the logic relative to the *games* dataset, which exchanges data with a MongoDB shared cluster on the server side through a *Mongo Connector.*

    - A module handling the logic relative to the *users* dataset, which exchanges data with a Neo4j database on the server side through a *Graph Connector*.

    - A module implementing a local persistent cache for the games' images through the use of a key-value database, which makes it possible to reduce the number of images that must be downloaded by the application at run-time, thereby improving its performance.

Server Side

The server side is comprised of a cluster of 9 virtual machines made available to us The server side is comprised of a cluster of 9 virtual machines made available to us by the University of Pisa, which were used to host a sharded MongoDB cluster and a single instance of the Neo4j database, where the distribution of tasks for each machine is described in the table below:

IP Address	Task 1		Task 2		Task 3	
172.16.0.80	Data Server (mongod)	Replica Set (Shard 3)	Config Server (mongod)	Replica Set	Config Server (mongos)	
172.16.0.82	Data Server (mongod)		Config Server (mongod)		-	
172.16.0.83	Data Server (mongod)		Config Server (mongod)		-	
172.16.0.75	-		Data Server (mongod)	Replica Set (Shard 1)	-	
172.16.0.76	-		Data Server (mongod)		-	
172.16.0.78	Neo4j Instance		Data Server (mongod)		-	
172.16.1.20	-		-		Data Server (mongod)	Replica Set (Shard 2)
172.16.1.30	-		-		Data Server (mongod)	
172.16.1.42	-		-		Data Server (mongod)	

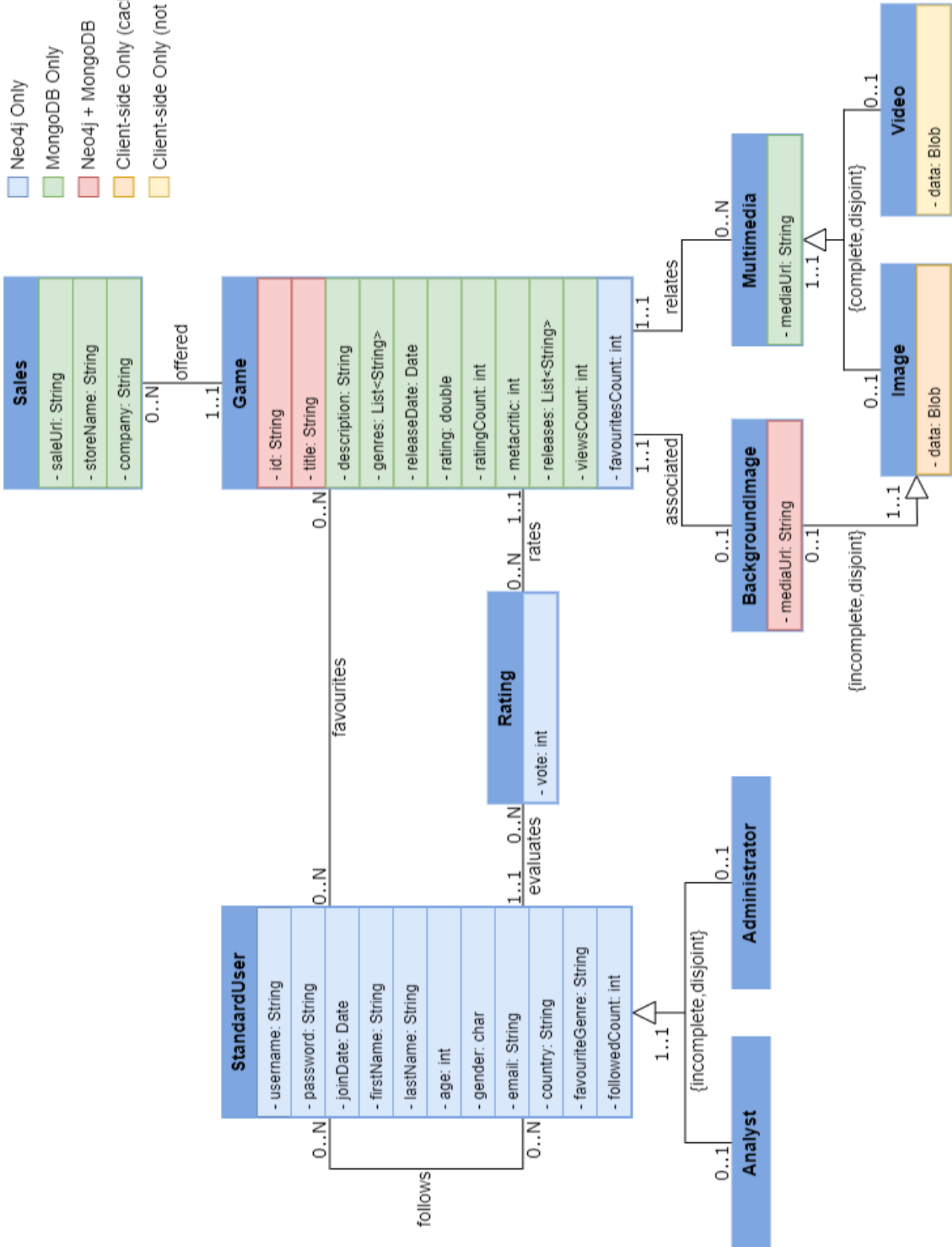The overall software architecture of the application is summarized in the following diagram:
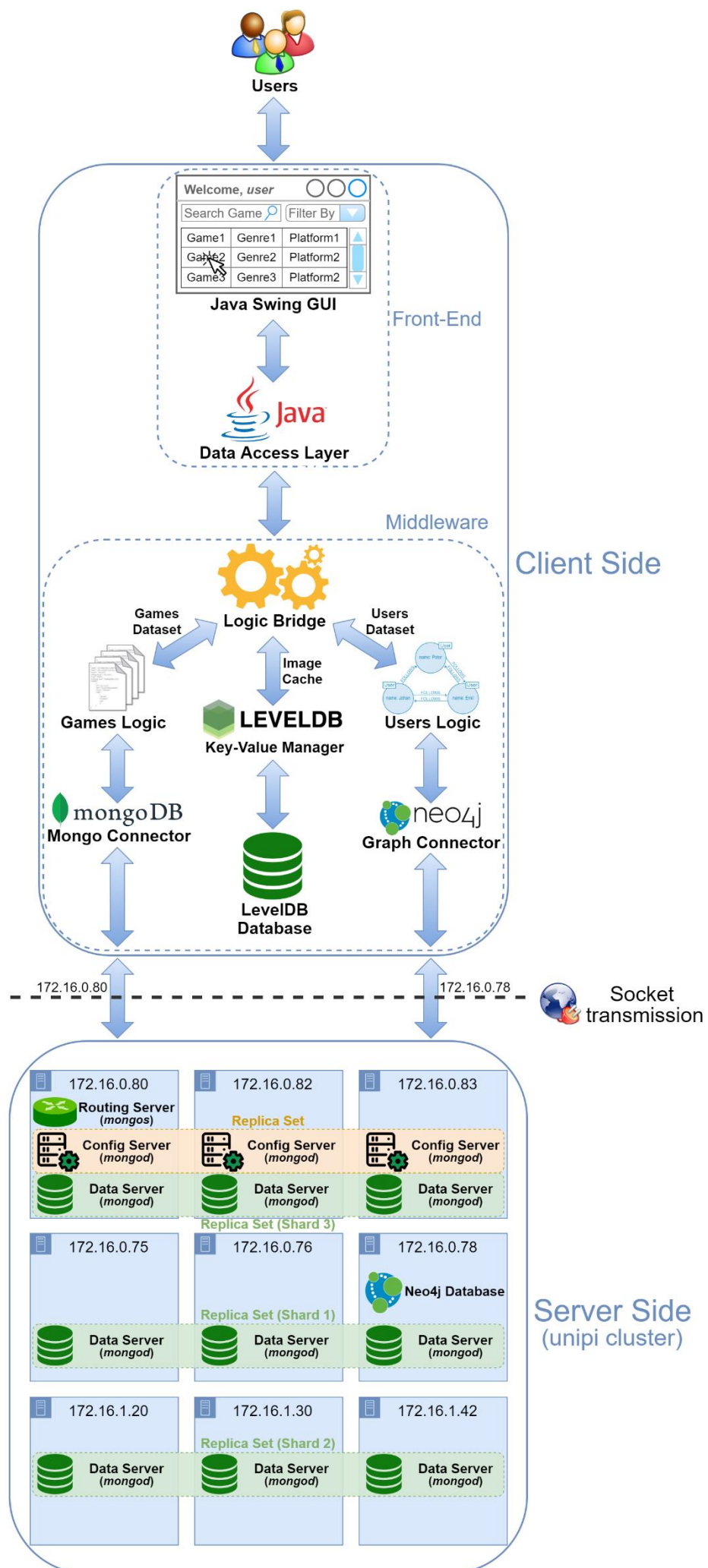
Application Dataset Organization

As previously mentioned, the data of the application is divided into two separate datasets:
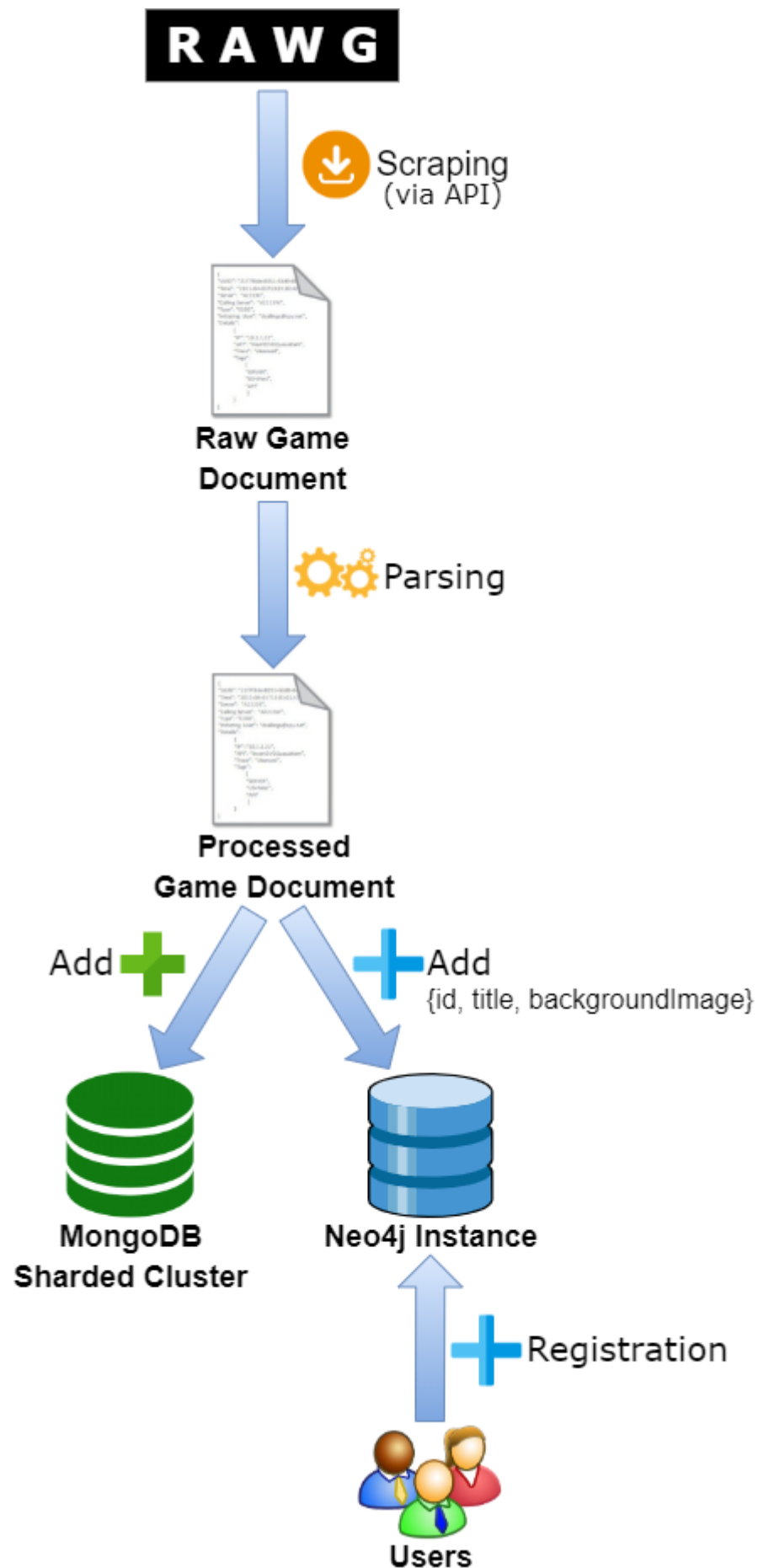
- A *Games* dataset, consisting in all the information on videogames offered by the application. This dataset has been built (and can be updated by Administrators via dynamic scraping) from a partition of the dataset offered by the RAWG service, which allows its contents to be scraped directly in JSON format through a set of HTTP-based [API](API).
  It should be noted, however, that the structure of the raw scraped data does not exactly reflect the one envisaged in our application in terms of both data organization and relevance, making it necessary for the raw data to undergo a "normalization" process before being added to the *Games* dataset.

- The *Users* dataset, containing all the information on the user registered within the application, which is populated as the users register and interact with the application's features.

While the two datasets were envisioned to be hosted in a MongoDB sharded cluster and in a Neo4j database respectively, following the analysis of the use cases and the data flow in the application, in order to prevent accessing different databases to perform a same query, which would have also complicated the cross-consistency management, the two datasets have been distributed between the two databases as depicted in the following "normalized" class diagram:

UML Class Diagram — GameBase Application

Legend:
- Neo4j Only
- MongoDB Only
- Neo4j + MongoDB
- Client-side Only (cached into key-value)
- Client-side Only (not cached)

Sales
- saleUrl: String
- storeName: String
- company: String

Game
- id: String
- title: String
- description: String
- genres: List<String>
- releaseDate: Date
- rating: double
- ratingCount: int
- metacritic: int
- releases: List<String>
- viewsCount: int
- favouritesCount: int

Multimedia
- mediaUrl: String

Video
- data: Blob

Image
- data: Blob

BackgroundImage
- mediaUrl: String

Rating
- vote: int

StandardUser
- username: String
- password: String
- joinDate: Date
- firstName: String
- lastName: String
- age: int
- gender: char
- email: String
- country: String
- favouriteGenre: String
- followedCount: int

Administrator

Analyst

Relationships:
- Sales — offered — Game (0..N / 1..1)
- Game — relates — Multimedia (1..1 / 0..N)
- Multimedia {complete,disjoint} → Video (1..1 / 0..1), Image (0..1)
- Game — associated — BackgroundImage (1..1 / 0..1)
- BackgroundImage {incomplete,disjoint} → Image (0..1 / 1..1)
- Image / Video {incomplete,disjoint} (1..1)
- StandardUser — favourites — Game (0..N / 0..N)
- Rating — rates — Game (0..N / 1..1)
- StandardUser — evaluates — Rating (1..1 / 0..N)
- StandardUser — follows — StandardUser (0..N / 0..N)
- StandardUser {incomplete,disjoint} → Administrator (0..1), Analyst (0..1) (1..1)

A summary of how the two databases of the application are populated is shown below:

MongoDB Design and Implementation

Documents Organization and Structure

The subset of the *Games* dataset stored in the MongoDB sharded cluster (p.12) has been organized in a single collection, with documents representing games and attuning to the following general structure:

```
{
"id": 3328,
"title": "The Witcher 3: The Wild Hunt",
"description": "The third game in a series, it holds nothing back from the player…",
"genre": "RPG",
"subGenres": ["Adventure","Action"],
"releaseDate": "Jun 18, 3915 12:00:00 AM",
"backgroundImage": " https://media.rawg.io/media/games/088/088b41ca3f9d22163e43be07acf42304.jpg",
"rating": 4.6746,
"ratingCount": 2591,
"metacritic": 93,
"releases": ["PC", "Xbox One", "PlayStation 4", "Nintendo Switch"],
"viewsCount": 510,
"favouritesCount": 11,
"sales":
  [
   {
    "store": "Steam", "company": "Valve",
    "saleUrl": " https://store.steampowered.com/app/292030/The_Witcher_3_Wild_Hunt/"
   },
   {
    "store": "XboxOne", "company": "Microsoft",
    "saleUrl": " https://www.microsoft.com/en-us/p/the-witcher-3-wild-hunt/br765873cqjd"
   },
   {
    "store": "PlayStation", "company": "Sony Entertainment",
    "saleUrl": " https://store.playstation.com/en-us/product/UP4497-CUSA00527_00-0000000000000002"
   }
  ],
"multimedia":
  {
   "images":
   [
    "https://media.rawg.io/media/games/088/088b41ca3f9d22163e43be07acf42304.jpg",
    "https://media.rawg.io/media/screenshots/1ac/1ac19f31974314855ad7be266adeb500.jpg",
    "https://media.rawg.io/media/screenshots/6a0/6a08afca95261a2fe221ea9e01d28762.jpg",
    "https://media.rawg.io/media/screenshots/cdd/cdd31b6b4a687425a87b5ce231ac89d7.jpg"
   ],
   "videos":
   [
    "https://media.rawg.io/media/stories-320/310/3109cc6c6168db8a03acc45f7edf5109.mp4",
    "https://media.rawg.io/media/stories-640/619/6197079f588ae10e3fe87edad3ee2d43.mp4",
    "https://media.rawg.io/media/stories/a25/a257f298e1de01ef2a077da02a5c8ee8.mp4"
   ]
  }
}
```

Where it should be noted that:

- The list of Genres of a game has been separated into two fields (*genre*, *subGenres*) representing respectively the main genre and an array of subgenres for the game, and this was decided in order to improve the read operations on such fields, since doing so allows the atomic *genre* field to be used directly in an aggregation without the additional pipeline step involved in separating the array into a set of documents each containing a single element (*unwind* operator), which as we examined in practice has a great impact on the read performance.

- The *Sale* and *Store* classes described in the original analysis class diagram have been merged as shown in the normalized class diagram (p. 12) in a single *Sales* class, which again was chosen to improve the read performance, allowing the information relating to a certain offer to be retrieved with a single read operation, at the cost of an added small redundancy within the database (the *store* and *company* fields, which can be repeated across different documents).

Queries Analysis

From the analysis of the use cases the following read and write queries involving the MongoDB database were identified, along with their expected frequency and cost:

Read Operations		
Operation	**Expected Frequency**	**Cost**
Retrieve information on a Game	High	Low (1 read)
Retrieve the max document ID	Low	Low (1 read)
Retrieve the total number of games	Low	Low (1 read)
Retrieve a list of game previews (ordered by views, rating or year)	High	Average (multiple reads)
Retrieve the total number of games satisfying a filter (releaseDate, genre or both)	Average	High (Aggregation)
Retrieve the total number of views of games satisfying a filter (year, genre or both)	Average	High (Aggregation)
Retrieve the total number of ratings of games satisfying a filter (year, genre or both)	Average	High (Aggregation)
Retrieve the most viewed game per year or per genre	Average	Very High (Complex Aggregation)
Retrieve the highest rated game per year or per genre	Average	Very High (Complex Aggregation)

Write Operations		
Operation	**Expected Frequency**	**Cost**
Update a game's *viewCount*	High	Low (1 attribute write)
Update a game's *rating* and *ratingCount*	Average	Low (2 attributes write)
Add/Delete a game from the database	Low	Average (document add/remove)

From which it can be concluded that the queries are predominantly *read intensive* (especially the ones relative to the games' statistics, which require aggregations), while write operations for their expected frequency and cost carry a much lesser impact on the overall performance.

Queries Implementations

A selection of implementations of the queries involving the MongoDB database is presented below:

Operation	Cypher Implementation
Retrieve the total number of game ratings per genre	```db.games.aggregate(
[
{ $group: { _id:"$genres", count:{$sum:"$ratingCount"}} },
{ $sort: { _id:-1} }
]
)```
Retrieve the total number of game ratings per year and genre	```db.games.aggregate(
[
{ $group:
{
_id: {year:{$year:"$releaseDate"}, genres:"$genres"},
count: {$sum:"$ratingCount"}
}
},
{ $sort: { _id:-1} }
]
)```
Retrieve the total number of games per year and genre	```db.games.aggregate(
[
{ $group:
{
_id: {year:{$year:"$releaseDate"}, genres:"$genres"},
count: {$sum:1}
}
},
{ $sort: { _id:-1} }
]
)```
Retrieve the total number of game views per year and genre	```db.games.aggregate(
[
{ $group:
{
_id: {year:{$year:"$releaseDate"}, genres:"$genres"},
count: {$sum:"$viewsCount"}
}
},
{ $sort: { _id:-1} }
]
)```

Shards Configuration

Shard Key Selection

As possible candidates for the sharding key the _id, *genre*, and *releaseDate* fields were initially considered, even if, due to the presence of queries consisting in aggregations involving the games' genres and release dates, in order to achieve a better load balancing and parallelization of the request the document _id field was finally selected as sharding key, where the resulting distribution of the documents across the three shards is depicted in the pie chart below:
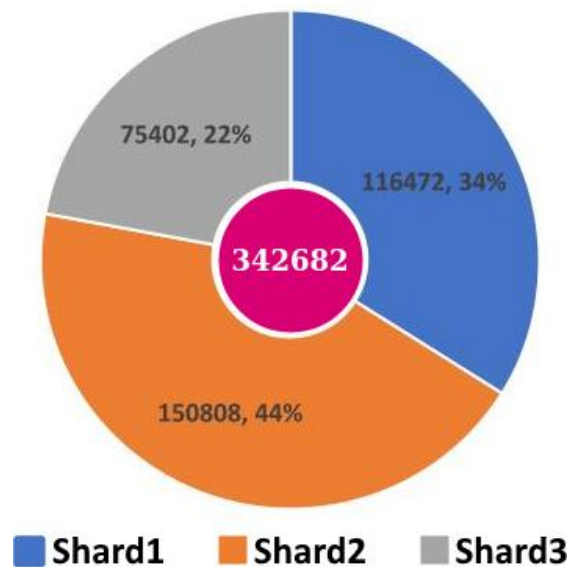
Where having a lower number of documents in Shard3 with respect to the other two shards in our case is a desirable situation, since the virtual machines hosting such shard (172.16.0.80, 172.16.0.82, 172.16.0.83) also host the replica-set of the configuration server and the *mongos* instance, thus achieving overall an even better load balancing.

Queries Configuration

Following our query analysis and the non-functional requirement of low latency in the user experience, the read and write operations on the shards were configured to return after reading or writing into a single member of the replica-set, thus attuning to the *eventual consistency* paradigm.

Indexes Definitions

In order to enhance the read operations on the database, the following indexes were introduced:

	1°	2°	3°	4°	5°	6°
Index Fields	_id	releaseDate	genre	viewsCount	rating	releaseDate, genre
Index Type	Hashed Index (default)	Simple Index	Simple Index	Simple Index	Simple Index	Composite Index

Sharding Latency Impact Analysis

As a confirmation of the performance benefits obtained by dividing the MongoDB dataset into multiple shards, we also carried out an analysis of the latencies of the queries involving aggregations (which are relative to computing statistics) using three, two and no sharding, where the results expressed as average latency for each query are outlined in the graph below:
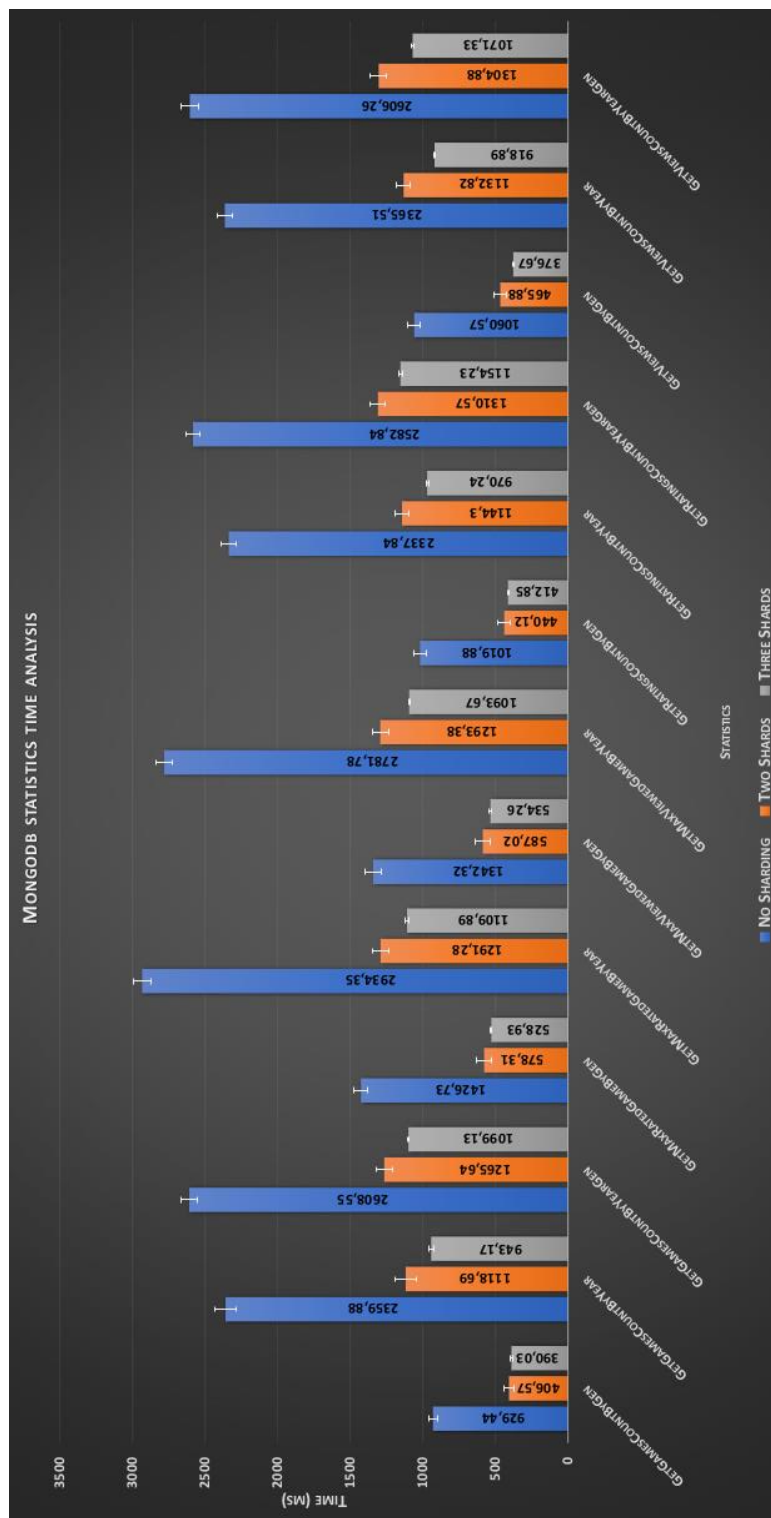
From the analysis we can conclude on the one hand that the latency improvements as the number of shards is varied differ depending on the operation, with some impacted more than others, while on the other that there are diminishing returns in the improvements by increasing the number of shards, since as we can observe, passing from no shards to two shards allows in general to halve the latencies, while going from two to three shards only carries modest latency improvements.

Documents Java Representation

The documents representing games are represented in the Java application through two different classes: a *Game* class, containing all the information stored in MongoDB relative to a game, and a *PreviewGame* class, which instead contains just the basic information relative to a game (namely its *id*, *title* and *backgroundImage*) that is used in context where loading the entire information about a game is not necessary (such as when searching for a game), allowing to reduce the load on the database and so to improve the responsiveness of the application.

Game.java

```java
public class Game
{
private Integer id;
private String  title;
private String  description;
private String  genre;
private Date    releaseDate;
private String  backgroundImage;
private Double  rating;
private Integer ratingCount;
private Integer metacritic;
private Integer viewsCount;
private Integer favouritesCount;

private ArrayList<String> subgenres;
private ArrayList<String> releases;
private ArrayList<PlatformInfo> sales;
private Multimedia multimedia;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

PreviewGame.java

```java
public class PreviewGame
{
private final Integer id;
private final String  title;
private final String  backgroundImage;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

Additional information on the implementation of the two classes, as well as the implementation of all the other classes involved in the interaction with the MongoDB database can be found in the application project attached to this report.

Neo4j Design and Implementation

Graph Design

Nodes

Two main types of nodes are used within the database:

- The **Users Nodes**, representing the users registered within the application, having as attributes the ones specified in their account information as per the *StandardUser* class attributes (pg. 6), and where the level of privilege is identified by the node's *label* (*:user*, *:analyst*, *:administrator*).

- The **Games Nodes**, representing all the games stored within the application with a small subset of their information consisting of the game *_id*, *title* and *backgroundImage* as per the relative documents in the MongoDB database, other than their *favouriteCount*. These nodes represent the portion of the *games* dataset that has been replicated into the Neo4j database, and are identified via the *:game* label.

Relationships

Three different kinds of relationships between nodes are used within the database:

- **User → FOLLOW → User**, which represents a user following another in the application, and is created/removed when a User adds/removes another User from his followed users list. This relationship has no attributes.

- **User → ADD → Game**, which represents a user favouriting a game of the application, and is created/removed when a User adds/removes a game from his favourite games list. This relationship has no attributes.

- **User → RATE → Game**, which represents a user's evaluation of a game, expressed as an integer between 1 and 5, that is stored in the *vote* attribute of the relationship. It should be noted that this relationship cannot be deleted, but only created or updated with a different vote by the user.

An example of the types of nodes and relationships used in the graph database is depicted below:
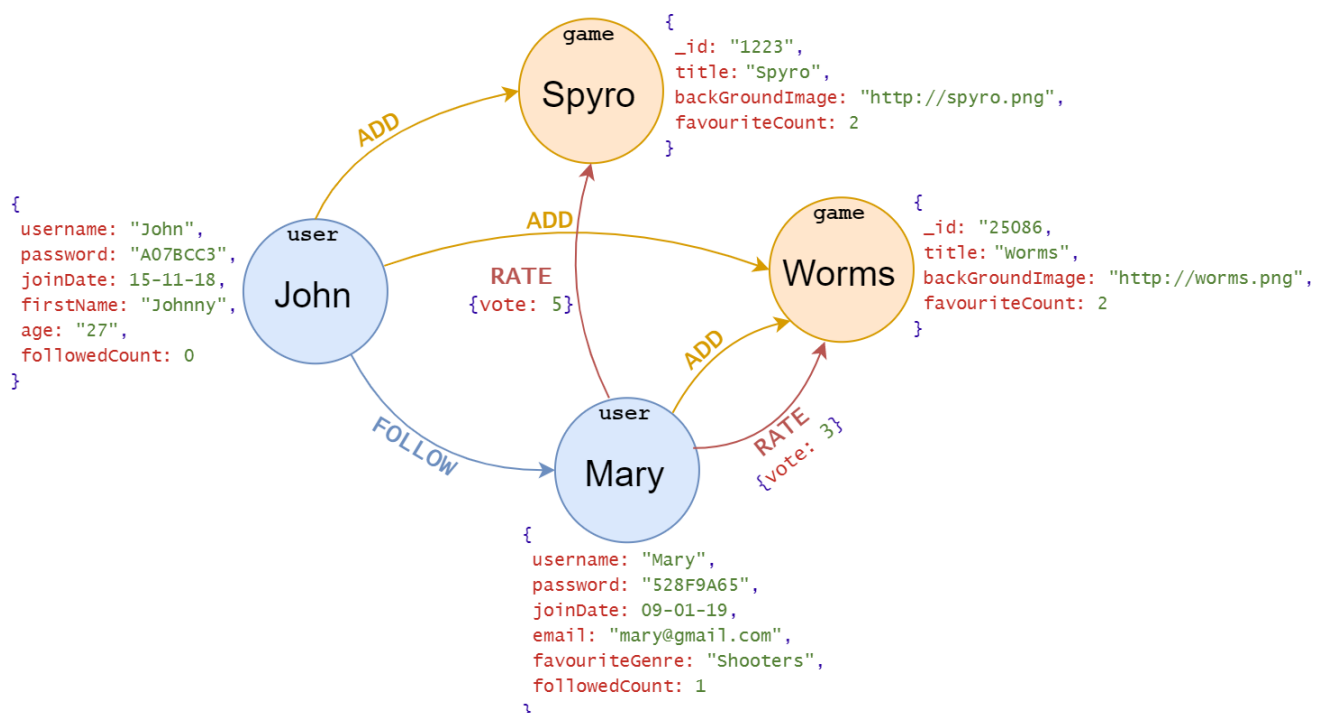
Queries Analysis

From the analysis of the use cases the following read and write queries involving the Neo4j database were identified, along with their expected frequency and cost:

Read Operations		
Operation	**Expected Frequency**	**Cost**
Validate a user login attempt	High	Low (1 read)
Retrieve the information related to a user on login	High	High (multiple reads and adjacencies)
Search for users by username via a mask	Average	Low (indexed read)
Retrieve a list of suggested users for the user	Low	Average (multiple reads and adjacencies)
Retrieve a user favourite games list	Average	Average (multiple adjacencies)
Retrieve a list of featured games for the user	Average	Very High (complex aggregation)
Retrieve the most followed users in the application	Low	Average (multiple reads)
Retrieve the most favourite games in the application	Low	Average (multiple reads)
Retrieve general information on the users registered within the application	Low	High (multiple reads)

Write Operations		
Operation	**Expected Frequency**	**Cost**
Register a new user into the database	Average	Low (create 1 node)
Update a user's account information	Low	Low (multiple attributes modifications)
Delete a user from the database	Low	Med (update multiple nodes, remove 1 node)
Upgrade a user from StandardUser to Analyst	Low	Low (modify 1 label)
Follow/Unfollow another user	Average	Low (create/delete 1 relationship)
Add/Remove a game from the database	Low	Low (create/remove 1 node)
Add/Remove a game from a user favourites list	Average	Low (create/delete 1 relationship)
Set/Update a user's game rating	Average	Low (create or update 1 relationship)

From which it can be concluded, similar to the MongoDB database, that the queries are predominantly *read intensive*, while write operations, even if greater in number and frequency with respect to the document database, still carry a limited impact on performance.

Queries Implementations

A selection of implementations in the Cypher language of the queries involving the Neo4j database is presented below:

Operation	Cypher Implementation

Retrieve the information related to a user on login	```MATCH (n{username:$username})```
	```RETURN labels(n) n.username, n.joinDate, n.firstName,```
	``` n.lastName, n.age, n.email, n.gender, n.country,```
	``` n.favouriteGenre, n.followedCount```
	```ORDER BY n.followedCount DESC LIMIT $limit```

Register a new user into the database	```CREATE (n:user{username:$username, password:$password,```
	``` joinDate:$joinDate, firstName:$firstName,```
	``` lastName:$lastName, age:$age, email:$email,```
	``` gender:$gender, country:$country,```
	``` favouriteGenre:$favGenre, followedCount:0})```

Follow another user	```MATCH (n{username:$currUser}),(p{username:$followUser})```
	```CREATE (n)-[:FOLLOWS]->(p)```
	```SET p.followedCount = p.followedCount + 1```

Search for users by username via a mask	```MATCH (n:user)```
	```WHERE n.username CONTAINS $mask```
	``` AND NOT n.username = $currentUser```
	```RETURN n.username, n.joinDate, n.firstName, n.lastName,```
	``` n.age, n.email, n.gender, n.country,```
	``` n.favouriteGenre, n.followedCount```
	```ORDER BY n.followedCount DESC LIMIT $limit```

Retrieve a list of suggested users for the user	```MATCH (p{username:$username}),(n:user)```
	```WHERE NOT (p)-->(n) AND NOT p.username = n.username```
	```RETURN n.username, n.joinDate, n.firstName, n.lastName,```
	``` n.age, n.email, n.gender, n.country,```
	``` n.favouriteGenre, n.followedCount```
	```ORDER BY n.followedCount DESC LIMIT $limit```

Add a game to a user favourites list	```MATCH (n{username:$username}),(g:game{_id:$gameID})```
	```CREATE (n)-[:ADD]->(g)```
	```SET g.favouriteCount = g.favouriteCount + 1```

Retrieve a list of featured games for the user	```CALL```
	```{```
	```MATCH (p{username:$username})-[:FOLLOWS]->(n)-[:ADD]->(g:game)```
	```WHERE NOT (p)-[:ADD]->(g)```
	```RETURN g ORDER BY g.favouriteCount DESC LIMIT $limit```
	```UNION```
	```MATCH (g:game)```
	```WHERE NOT ({username:$username})-[:ADD]->(g)```
	```RETURN g ORDER BY g.favouriteCount DESC LIMIT $limit```
	```}```
	```RETURN g._id, g.title, g.favouriteCount LIMIT $limit```

Implementation Details

The following represent miscellaneous details of the implementation of the Neo4j database and the GraphConnector module:

- In order to enhance the read operations on the database, the following indexes were introduced:

Index Name	Index Type	Index Labels	Index Attributes
usersIndex	BTREE (simple index)	:user	username
analystsIndex	BTREE (simple index)	:analyst	username
administratorsIndex	BTREE (simple index)	:administrator	username
gamesIndex	BTREE (simple index)	:game	_id

- To guarantee the security on their credentials, the users' passwords are stored in the database in an encrypted form using the SHA-256 hashing algorithm.

- All the user-related logic, from login management, to storing user information (including account information, followed users list, suggested users list, favourite games list and featured games list), to privilege level controls and so on have been embedded within the GraphConnector module, which depending on the type of operation also incorporates some automatic retrial and error correction mechanisms, that are thoroughly explained in the *GraphInterface.java* interface file implemented by the *GraphConnector* module.

- The *users* dataset was manually populated with 47 stub users having different privilege levels, each with its user attributes, followed users and favourite games.

- A view of the final dataset stored in the Neo4j database is depicted below:
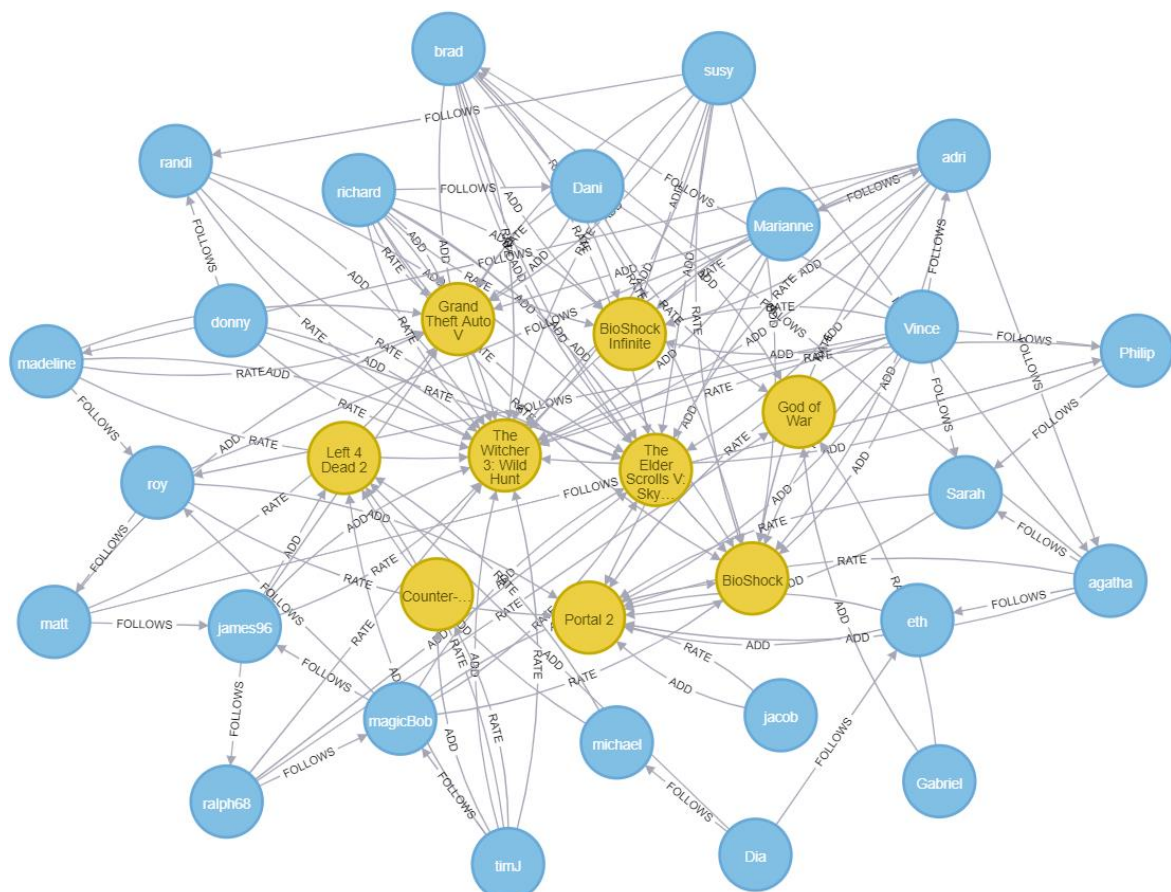
Nodes Java Representation

The users and games nodes are represented in the Java application through two different classes: the *User* class (along with the *Analyst* and *Administrator* subclasses), containing all the user account information, and the *GraphGame* class, containing all the information related to a game in the database.

User.java

```java
public class User
{
  //Final Attributes
  private final String username;        //The user's username
  private final String password;        //The user's password
  private final LocalDate joinDate;     //The user registration date (see note 2)

  //Variable Attributes
  private String firstName;             //The user's first name
  private String lastName;              //The user's last name
  private Long age;                     //The user's age (see note 2)
  private String email;                 //The user's email
  private Character gender;             //The user's gender (M or F)
  private String country;              //The user's country
  private String favouriteGenre;        //The user's favourite videogame genre
  private Long followedCount;          //The number of the user's followers

  //NOTES:
  // 1) Wrapper classes were used to allow for a better handling of null values
  // 2) The use of selected classes is dictated due to current implementation of the Neo4j
  //      driver (as the LocalDate and the use of the Long class, since Integer/int is not
  //      natively supported)

  /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Analyst.java

```java
public class Analyst extends User
{
  //Same Attributes as the superclass

  /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Administrator.java

```java
public class Administrator extends User
{
  //Same Attributes as the superclass

  /* Constructors, Getters and Setters (omitted for brevity) */
}
```

GraphGame.java

```java
public class GraphGame
{
  //Final Attributes
  public final String _id;              //The game unique identifier
  public final String title;            //The game title
  public final String previewImage;     //The game preview image
  public final Long favouriteCount;     //The game favouriteCount

  //Variable Attributes
  Long vote;                            //The game vote (when applicable)

  /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Additional information on the implementation of the two classes, as well as the implementation of all the other classes involved in the interaction with the Neo4j database, can be found in the application project attached to this report.

Other Implementation Details

Cross-Database Consistency Management

Given how the two datasets and the application queries have been distributed between the two databases, the only operations requiring a cross-database consistency management are the *Update Game User Rating*, *Add Game* and *Remove Game* operations, where, in all cases, in addition to attempting automatic consistency recovery mechanisms, errors are logged within an *errors.txt* file, allowing administrators to manually check and enforce consistency once the database(s) where problem(s) have occurred are restored to the nominal operating state.
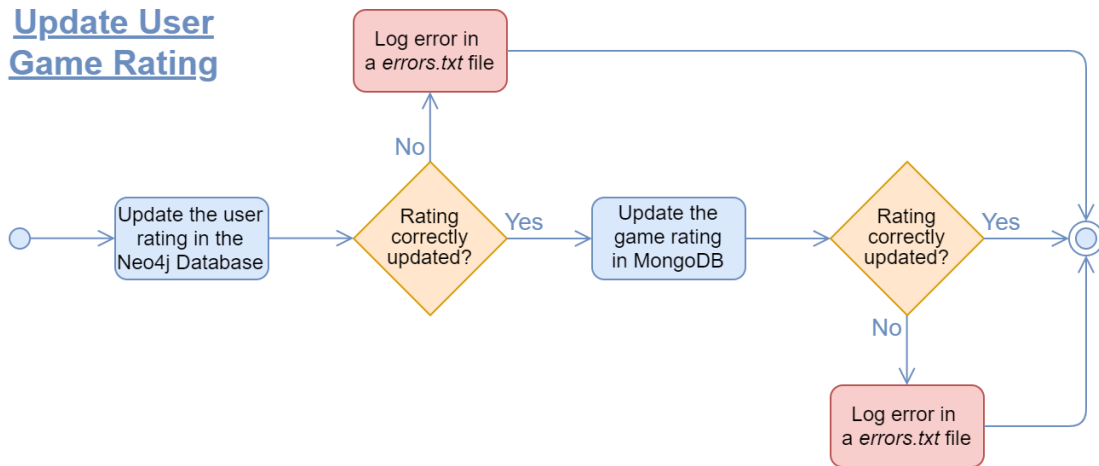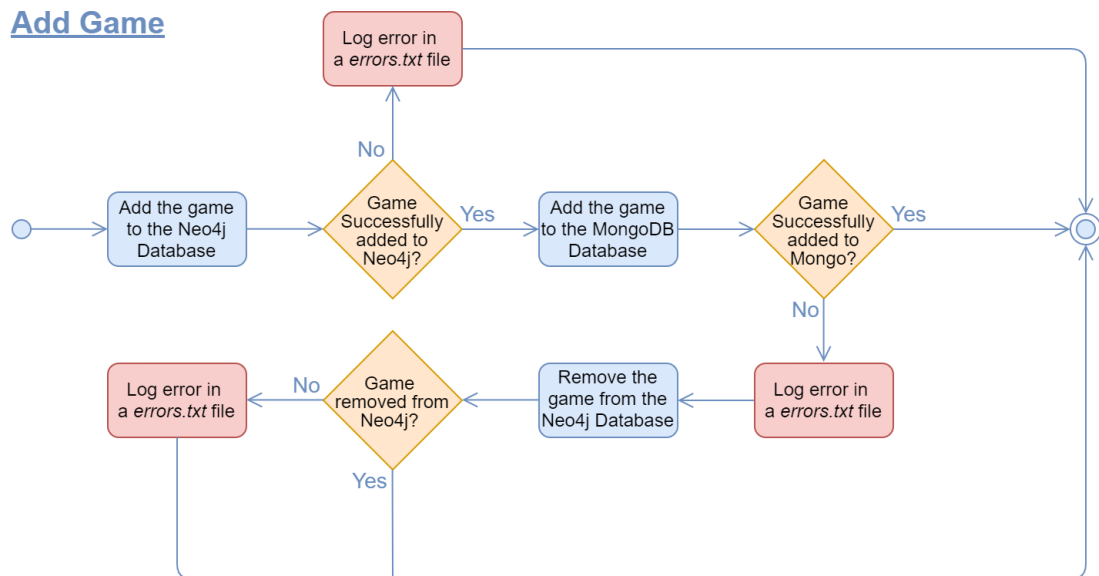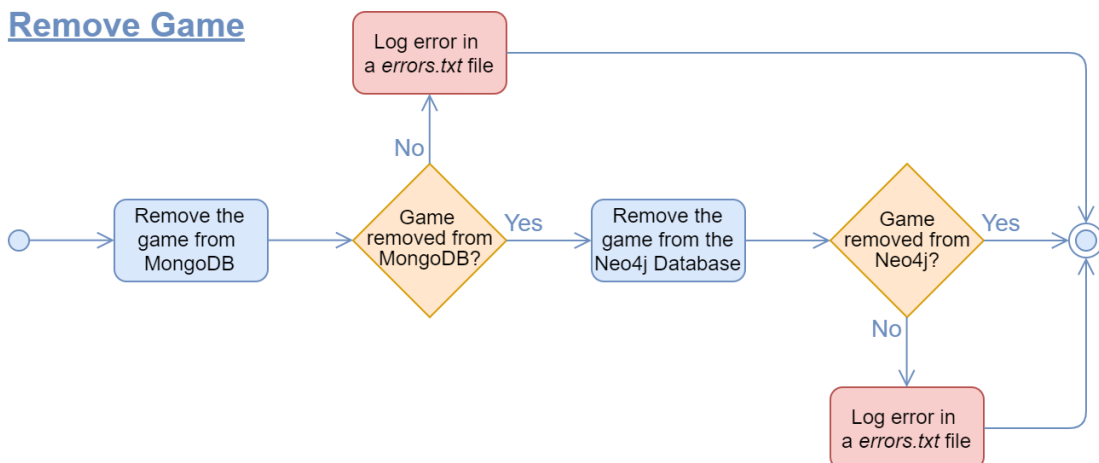
It should also be noted, when a user is deleted by an administrator, that, while the *favouriteCount* of the games in its favourite list are decremented, the *rating* and *ratingCount* of the games the user voted are not updated, which is a desidered behavior since the values of such attributes are determined by factors external to the application in the first place, being they obtained via dynamic scraping.

Key-Value Database Implementation

As previously mentioned in the software architecture (Pg. 11), in order to reduce the bulk of image data to be downloaded by the application at run time, a key-value database was chosen to be employed on the client side.

Among the different open-source possibilities a Java port of Google's *LevelDB* database was adopted, where a single bucket was used with the only key-value pairing consisting of an image's URL (key) and its related data (value), allowing the application to retrieve the image data from the internet only when not already present is such cache, which as we have confirmed allows the responsiveness of the application to be drastically improved.

Application Project Source Organization

The final implementation of the Java application project consists of over 13,000 lines of code divided among 38 classes, interfaces and enumerations, whose names and package organization are outlined below:

- ∨ ⊞ > graphicInterface
  - > ▣ > AreaChartPanel.java
  - > ▣ > BarChartPanel.java
  - > ▣ BufferedGameRenderer.java
  - > ▣ > ButtonColumn.java
  - > ▣ > GraphicInterface.java
  - > ▣ > ImageRenderer.java
  - > ▣ > PieChartPanel.java
  - > ▣ > VideoPlayerPanel.java

- ∨ ⊞ > logic.data
  - > ▣ Administrator.java
  - > ▣ Analyst.java
  - > ▣ BufferedGame.java
  - > ▣ > Game.java
  - > ▣ > GraphGame.java
  - > ▣ > Multimedia.java
  - > ▣ PlatformInfo.java
  - > ▣ PreviewGame.java
  - > ▣ Statistics.java
  - > ▣ StatusObject.java
  - > ▣ > User.java
  - > ▣ UserInfo.java
  - > ▣ UserStats.java

- ∨ ⊞ > logic
  - > ▣ ImgCache.java
  - > ▣ > LogicBridge.java
  - > ▣ > StatusCode.java
  - > ▣ UserType.java

- ∨ ⊞ > logic.graphConnector
  - > ▣ > GraphConnector.java
  - > ▣ > GraphInterface.java

- ∨ ⊞ > logic.mongoConnection
  - > ▣ CacheThread.java
  - > ▣ CachingPolicy.java
  - > ▣ > DataNavigator.java
  - > ▣ > MongoConnection.java
  - > ▣ MongoStatistics.java
  - > ▣ > NavElem.java
  - > ▣ > NavType.java

- ∨ ⊞ > scraping
  - > ▣ > HttpClient.java
  - > ▣ ScrapingThread.java
  - > ▣ util.java
  - > ▣ > WebScraping.java

Project Object Model (POM) File

The entire development cycle of the Java application relied on the support of the *Maven* project management tool, where the *Project Object Model* (POM) file that was used is shown below:

pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>Gamebase</groupId>
  <artifactId>Gamebase</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Gamebase</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.target>1.8</maven.compiler.target> <maven.compiler.source>1.8</maven.compiler.source>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId> <artifactId>junit</artifactId> <version>3.8.1</version> <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId> <artifactId>commons-math3</artifactId> <version>3.0</version>
    </dependency>
    <dependency>
      <groupId>org.jsoup</groupId> <artifactId>jsoup</artifactId> <version>1.11.3</version>
    </dependency>
    <dependency>
      <groupId>org.apache.httpcomponents</groupId> <artifactId>httpclient</artifactId> <version>4.5.10</version>
    </dependency>
    <dependency>
      <groupId>org.json</groupId> <artifactId>json</artifactId> <version>20190722</version>
    </dependency>
    <dependency>
      <groupId>org.jfree</groupId> <artifactId>jfreechart</artifactId> <version>1.5.0</version>
    </dependency>
    <dependency>
      <groupId>com.googlecode.json-simple</groupId> <artifactId>json-simple</artifactId> <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>org.mongodb</groupId> <artifactId>mongo-java-driver</artifactId> <version>3.12.1</version>
    </dependency>
    <dependency>
      <groupId>com.google.code.gson</groupId> <artifactId>gson</artifactId> <version>2.8.6</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId> <artifactId>log4j-api</artifactId> <version>2.9.0</version>
    </dependency>
    <dependency>
      <groupId>ch.qos.logback</groupId> <artifactId>logback-classic</artifactId> <version>1.2.3</version>
    </dependency>
    <dependency>
      <groupId>org.neo4j.driver</groupId> <artifactId>neo4j-java-driver</artifactId> <version>4.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.iq80.leveldb</groupId> <artifactId>leveldb-api</artifactId> <version>0.9</version>
    </dependency>
    <dependency>
      <groupId>org.iq80.leveldb</groupId> <artifactId>leveldb</artifactId> <version>0.9</version>
    </dependency>
  </dependencies>


  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId> <version>3.6.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```