UNIVERSITÀ DI PISA

Computer Engineering

Large-Scale and Multi-Structured Databases

Gameflows: A Social Network for Videogame Fans

TEAM MEMBERS (Group 1):

Fabrizio Lanzillo

Riccardo Sagramoni

Luca Tartaglia

Academic Year: 2021-2022

GitHub Release: https://github.com/RiccardoSagramoni/gameflows-social-network/releases

GitHub Repository: https://github.com/RiccardoSagramoni/gameflows-social-network

Table of Contents

Introduction and requirements

GameFlows is a **social networking application** that aims to put together people interested in the same videogames.

Users registered to the service can discover new videogames and follow the related community. Inside a community a user can interact with the other user by writing posts or reply to other users' posts.

Moreover, user can read posts written by special users, called *influencers*. Influencers are users who have proven to be engaging inside the community in respect to the other users. The level of "ability to engage" is defined the number of recent likes and comments on its posts. In this perspective, their posts can be filtered out from the others while browsing the post of a videogame community.

Functional requirements

In the application we can observe two types of actors: **users** and **admins**.

Actor: user

At the start of the application, a user can:

- Register a new account
- Login into the application with their account

After their login, they can:

- Logout
- Browse the videogame community
- Search a videogame community by name
- View a list of **followed** videogame communities
- View a list of **suggested videogames**
- View some statistics on the videogame communities

Inside a videogame community, they can:

- View the videogame community information
- Follow a videogame community
- Write a post
- Read the posts of the community

Inside a post, they can:

- Like the post
- Remove their like to the post
- Write a comment
- Read the comments
- Like a comment (or remove the like)
- Delete the posts or comments written by themselves.

Actor: admin

An **admin** can:

- manage the videogame communities (create or delete them)
- create other administrator accounts
- ban/unban users (a banned user is not allowed to use the social network in any way)
- view statistics
- submit an automated update of the influencers (they can configure how many).

Non-functional requirements

During the design of the application, the following requirements are taken in account:

- Fast responses
- High availability
- Fault-tolerance, in terms of single-point failure and data lost
- Usability

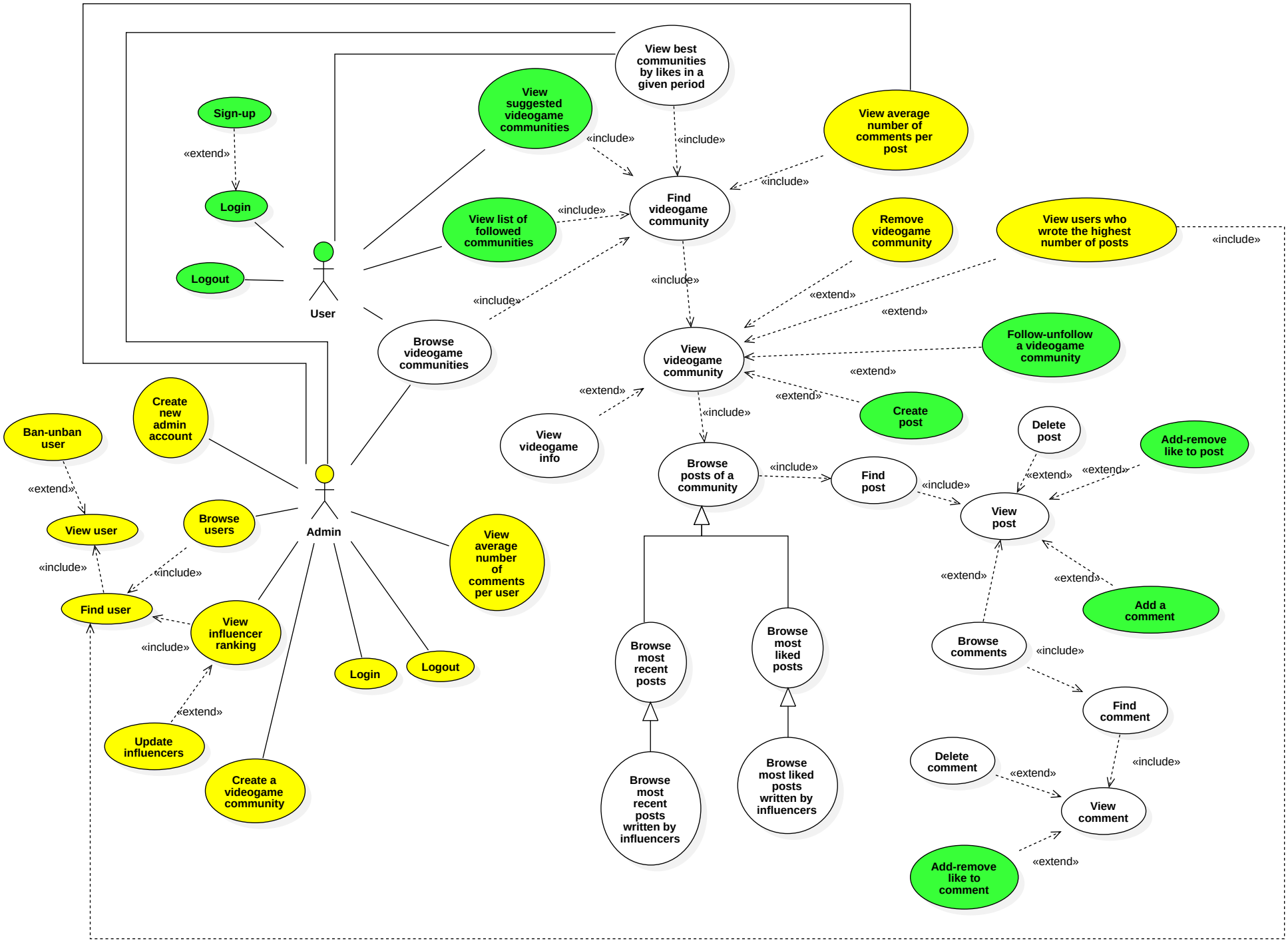
To satisfy these requirements, we focused on the "Availability – Partition Tolerance" features of the CAP triangle, i.e. the **Eventual Consistency**.

Use case

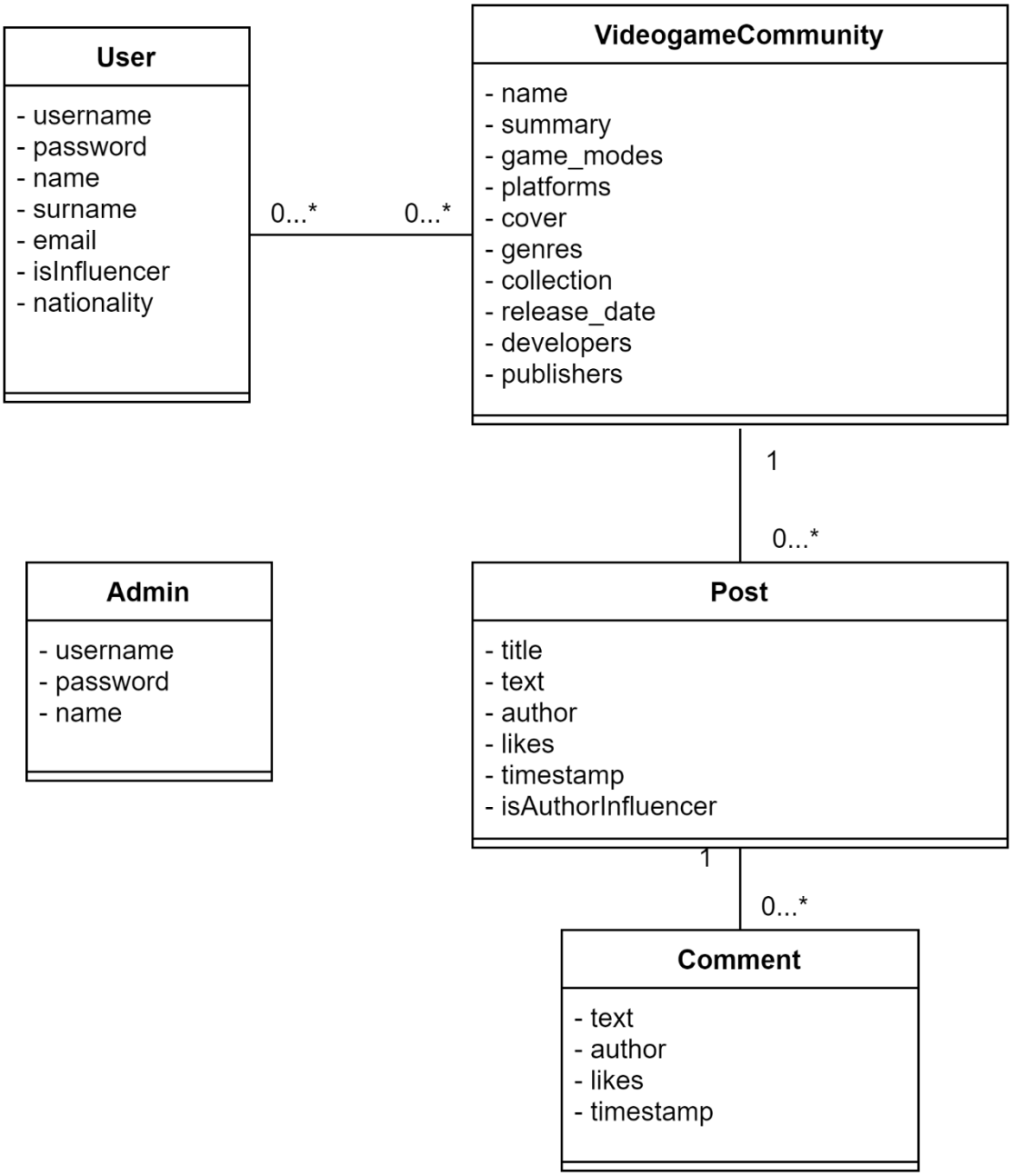
In the following page, we can observe a **Use Case diagram** derived from the functional requirements.

The use cases that belong exclusively to the admin are marked in yellow, whereas the use cases which belong exclusively to the user are marked in green.

White use cases are shared by both actors.


View best communities by likes in a given period
View suggested videogame communities
View average number of comments per post
Sign-up
Login
Logout
View list of followed communities
Find videogame community
Remove videogame community
View users who wrote the highest number of posts
User
Browse videogame communities
View videogame community
Follow-unfollow a videogame community
Create post
View videogame info
Browse posts of a community
Find post
Delete post
Add-remove like to post
View post
Add a comment
Browse comments
Find comment
Delete comment
View comment
Add-remove like to comment
Browse most recent posts
Browse most liked posts
Browse most recent posts written by influencers
Browse most liked posts written by influencers
Ban-unban user
Create new admin account
Admin
View user
Browse users
Find user
View influencer ranking
Update influencers
Create a videogame community
View average number of comments per user
Login
Logout
«include»
«extend»

Analysis class diagram



Dataset description

For our application, we built a dataset containing:

- Real videogame information, extracted from the IGDB.com database through the **IGDB API** (https://api-docs.igdb.com/)
- Randomly generated user, with their personal information, through **RandomUser API** (https://randomuser.me/)
- Real posts and comments, extracted from **Reddit** through the **PRAW API** (https://praw.readthedocs.io/en/stable/)

Then, we integrate the extracted data and obtained a final volume of about 60 MB.

For what concerns the requirements of variety and velocity/variability:

- ***Variety***: We built out dataset by integrating three different sources
- ***Velocity/Variability***: Our dataset doesn't lose importance after time, so no refresh mechanism from the external world is taken in account. The dataset will be only modified from the interaction with the users through the application.

Database choices and motivation

We have chosen to adopt two different databases for our application:

- **MongoDB**: we have chosen a document database to exploit the support that provides in terms of storage, indexing and complex queries elaboration (critical for the analytic part of the application).
- **Neo4j**: we have chosen a graph database in order to efficiently represent the relationships between different entities in the application (e.g., the "likes" relationship between a user and a post they like) and to analyze the paths created by these relationships.

Design of MongoDB

This database contains five collections to store information about users, admin, videogames, posts and comments.

Collection: User

- 800 documents
- 151.4 KB storage size
- Average size of a document: 342 B


```
{
  "_id": {
    "$oid": "61dbf8a992a81a85adeaeced"
  },
  "username": "whitefish666",
  "email": "maurijn.wijdeven@example.com",
  "password": {
    "sha256": "e0c1a8d336d7447960f78655d91adf7d13075c7be3947b4c9df1cc0784e6805a",
    "salt": "XFYVyr8i"
  },
  "name": "Maurijn",
  "surname": "Wijdeven",
  "gender": "male",
  "date-of-birth": {
    "$date": "1954-09-18T00:00:00.000Z"
  },
  "nationality": "NL",
  "isInfluencer": false,
  "isBlocked": false
}
```


Each user has their information (username, email, gender, password eccetera) stored in a document of the User collection.

The fields isInfluencer and isBlocked are boolean tags, which are set by the administrators. Neither of them is mandatory (they could be null)

The password field is composed of a randomly generated string (a *salt*) and the SHA-256 hash of the password with the salt concatenated. This prevents us to store passwords in clear (which is a very bad practice) and makes our system more robust to rainbow table attacks.

Collection: Admin

- 1 document
- 20.5 KB storage size
- Average size of a document: 169 B


```
{
  "_id": {
    "$oid": "62069fa8d7a57e386f43b85c"
  },
  "username": "admin",
  "password": {
    "sha256": "f5d777e104f096c6bb2a809482fc609ce6c0bf84f5ed73c6ab920807874595c6",
    "salt": "PW63h6p1"
  },
  "name": "admin"
}
```


Each admin has their information (username, password and name) stored in the admin collection. The same considerations about the password for the user have been also made for the admin.

Collection: Videogame

- 132 documents
- 81.9 KB storage size
- Average size of a document: 894 B

Each videogame community has their technical information stored in a document of the Videogame collection.

The following fields are mandatory, whereas the others are optional (they could be missing):

- name
- summary

```
{
  "_id": {
    "$oid": "61dbfbbe92a81a85adeaf012"
  },
  "name": "The Sims 3",
  "summary": "In The Sims 3, you can create Sims with unique personalities, fulfill
their desires, and control their lives within a customizable living neighborhood. Unlock
all-new Karma Powers and unleash them on your Sims: help your Sim get lucky with the
power of “love connector”, bless them with the power of “age defiler” or \"instant
beauty\" or curse them with an \"instant enemy\". Use these powers wisely, because they
may have unexpected results! As you guide your Sims through life, you can complete
challenges to unlock additional items, new buildings and landmarks.",
  "game_modes": [
    "Single player"
  ],
  "platforms": [
    "Wii",
    "PC (Microsoft Windows)",
    "PlayStation 3",
    "Xbox 360",
    "Mac",
    "Nintendo DS",
    "Android",
    "Nintendo 3DS",
    "iOS"
  ],
  "cover": "//images.igdb.com/igdb/image/upload/t_thumb/co1zof.jpg",
  "genres": [
    "Role-playing (RPG)",
    "Simulator",
    "Strategy",
    "Adventure"
  ],
  "collection": "The Sims",
  "aggregated_rating": 83.1111111111111,
  "release_date": {
    "$date": "2009-06-01T22:00:00.000Z"
  },
  "developers": [
    "Maxis"
  ],
  "publishers": [
    "Electronic Arts"
  ]
}
```


Collection: Post

- ≈ 63,000 documents
- 19.5 MB storage size
- Average size of a document: 598 B

Each post stores its data in a document of the post collection.

There are two main redundancies:

- The `likes` field is a redundant counter of the :LIKES relationship in the Neo4j graph. This allows us to efficiently compute statistics on the amount of likes per post. Since it’s an

integer, its size is fixed so there won't be any necessary to reallocate the documents after a lot of updates.

- The `community` field is a redundancy of the videogame community the post belongs to. This redundant field contains a copy of the fields _id and name of a Videogame document, thus is quite small in terms of storage size. The redundancy allows us to efficiently retrieve the posts of a given videogame community


```
{
  "_id": {
    "$oid": "61dbfbd392a81a85adeaf096"
  },
  "title": "PSA: Spam Posts & New Flair Requirement",
  "likes": 22,
  "author": "bigdog847",
  "text": "In an attempt to counteract the *inordinate* amount of spam posts, flairs
are required on all posts for now. \n\nA new flair (\"Other\") has been added for all
those posts that don't fit pre-existing categories.\n\nSorrowful be the heart,
Penitent Ones in shambles.",
  "community": {
    "community_id": {
      "$oid": "61dbfbbe92a81a85adeaf04c"
    },
    "community_name": "Blasphemous"
  },
  "timestamp": {
    "$date": "2021-06-20T23:39:19Z"
  },
  isAuthorInfluencer: true
}
```


isAuthorInfluencer is an optional field

Collection: Comment

- ≈ 302,700 documents
- 59.7 MB of storage size
- Average size per document: 397 B

Each comment stores its data in a document of the post collection.

There are two main redundancies:

- The `likes` field is a redundant counter of the :LIKES relationship in the Neo4j graph. This allows us to efficiently compute statistics on the amount of likes per comment. Since it's an integer, its size is fixed so there won't be any necessary to reallocate the documents after a lot of updates.
- The `post` field is a redundancy of the post the comment is replying to. This redundant field contains a copy of:
    - The _id field of the post
    - The author field of the post
    - The _id field of the videogame community the post belongs to
    - The name field of the videogame community the post belongs to

  The redundancy allows us to efficiently retrieve the comments of a given post or videogame community and compute statistics on them

```
{
  "_id": {
    "$oid": "61dbfc0692a81a85adebe691"
  },
  "text": "Sorrowful be the heart, elder bro",
  "likes": 4,
  "author": "happypanda257",
  "post": {
    "post_id": {
      "$oid": "61dbfbd392a81a85adeaf095"
    },
    "author": "tinywolf599",
    "community_id": {
      "$oid": "61dbfbbe92a81a85adeaf04c"
    },
    "community_name": "Blasphemous"
  },
  "timestamp": {
    "$date": "2021-02-26T11:04:29.000Z"
  }
}
```


CRUD operations

CREATE

Insert user	`db.user.insertOne({ "username": "whitefish666", "email": "maurijn.wijdeven@example.com", "password": { "sha256": "e0c1a8d336d7447960f78655d91adf7d13075c7be3947b4c9df1cc0784e6805a", "salt": "XFYVyr8i" }, "name": "Maurijn", "surname": "Wijdeven", "gender": "male", "date-of-birth": { "$date": "1954-09-18T00:00:00.000Z" }, "nationality": "NL", "isInfluencer": true, "isBlocked": false })`
Insert admin	`db.admin.insertOne({ "username": "admin", "password": { "sha256": "f5d777e104f096c6bb2a809482fc609ce6c0bf84f5ed73c6ab920807874595c6", "salt": "PW63h6p1" }, "name": "admin" })`

Insert videogame community	db.videogame.insertOne({
	"name": "BioShock",
	"summary": "Set in the steampunk Objectivist underwater dystopia of Rapture in the 1960's, Bioshock is a first-person shooter wherein the player arms and genetically modifies themselves in order to survive against the city's resident great minds-turned-maddened anomalies and aid a revolutionary leader named Atlas in overthrowing the city's ruling despot, Andrew Ryan.",
	"game_modes": [
	"Single player"
	],
	"platforms": [
	"PC (Microsoft Windows)",
	"PlayStation 3",
	"Xbox 360",
	"Mac",
	"iOS"
	],
	"cover": "//images.igdb.com/igdb/image/upload/t_thumb/co2mli.jpg",
	"genres": [
	"Shooter",
	"Role-playing (RPG)"
	],
	"collection": "Bioshock",
	"aggregated_rating": 92.625,
	"release_date": {
	"$date": "2007-08-20T22:00:00.000Z"
	},
	"developers": [
	"2K Boston",
	"2K Australia"
	],
	"publishers": [
	"2K Games",
	"Feral Interactive"
	]
	}

Insert post	db.post.insertOne({
	"title": "PSA: Post Flairs & User Flairs Are Up!",
	"likes": 0,
	"author": "tinywolf599",
	"text": "text",
	"community": {
	"community_id": {
	"$oid": "61dbfbbe92a81a85adeaf04c"
	},
	"community_name": "Blasphemous",
	"community_genre": [
	"Platform",
	"Role-playing (RPG)",
	"Hack and slash/Beat 'em up",
	"Adventure",
	"Indie"

	] }, "timestamp": { "$date": "2021-02-25T04:01:54.000Z" }, isAuthorInfluencer: true })

Insert comment	db.comment.insertOne({
	"text": "Sorrowful be the heart, elder bro",
	"likes": 0,
	"author": "happypanda257",
	"post": {
	"post_id": {
	"$oid": "61dbfbd392a81a85adeaf095"
	},
	"author": "tinywolf599",
	"community_id": {
	"$oid": "61dbfbbe92a81a85adeaf04c"
	},
	"community_name": "Blasphemous"
	},
	"timestamp": {
	"$date": "2021-02-26T11:04:29.000Z"
	}
	}

READ

USER	
Get password for login	db.user.find({$and: [{username:"whitefish666"}, {$or: [{"isBlocked": {$not: {$exists: ""}}}, {"isBlocked": false}]}]}, {_id: 0, password: 1})
Find user by username	db.user.find({username: "whitefish666"})
Browse users	db.user.find({}).projection({"password": 0}).sort({"username": 1}).skip(0).limit(5)

Search user (regex)	db.user.find(
	{
	username: {
	$regex: 'white',
	$options: 'i'
	}
	}).projection({"password": 0}).sort({"username": 1}).skip(0).limit(5)

ADMIN	
Get password for login	db.admin.find({username:"whitefish666"}, {_id: 0, password: 1})
VIDEOGAME	
Get all videogames	db.videogame.find({}).sort({"name": 1})

Search a videogame by name	db.videogame.find(
	{
	name: {
	$regex: 'battle',
	$options: 'i'
	}
	}
	)

Find a videogame	db.videogame.find({_id: ObjectId('61dbfbbe92a81a85adeaf00f')})
POST	
Find a post	db.post.find({_id: ObjectId('61dbfbbe92a81a85adeaf00f')})
Browse posts by likes	db.post.find({"community.community_id": ObjectId('61dbfbbe92a81a85adeaf04c')}).sort({likes: -1})
Browse posts by timestamp	db.post.find({"community.community_id": ObjectId('61dbfbbe92a81a85adeaf04c')}).sort({timestamp: -1})
Browse influencers' posts by likes	db.post.find({"community.community_id": ObjectId('61dbfbbe92a81a85adeaf04c'), "isAuthorInfluencer": true}).sort({likes: -1})
Browse influencers' posts by timestamp	db.post.find({"community.community_id": ObjectId('61dbfbbe92a81a85adeaf04c'), "isAuthorInfluencer": true}).sort({timestamp: -1})
COMMENT	
Find a comment	db.comment.find({_id: ObjectId('61dbfc0692a81a85adebe691')})
Get all comments of a post	db.comment.find({"post.post_id": ObjectId('61dbfbde92a81a85adeb7fd5').sort({timestamp: -1})

UPDATE

USER	
Block user	db.user.updateOne({username: "whitefish666"}, {$set: {isBlocked: true})
Unblock user	db.user.updateOne({username: "whitefish666"}, {$set: {isBlocked: false})
POST	
Like a post	db.post.updateOne({ _id: ObjectId('61dbfbd392a81a85adeaf095')}, { $inc: { likes: 1} })
Remove like from a post	db.post.updateOne({ _id: ObjectId('61dbfbd392a81a85adeaf095')}, { $inc: { likes: -1} })
COMMENT	
Like a post	db.comment.updateOne({ _id: ObjectId('61dbfc0692a81a85adebe691')}, { $inc: { likes: 1} })

Remove like from a post	db.comment.updateOne({ _id: ObjectId(‘61dbfc0692a81a85adebe691’)}, { $inc: { likes: -1} })

DELETE

VIDEOGAME	

Delete a videogame	db.videogame.deleteOne(
	{
	"_id": ObjectId('61dff9b95dd4f4998d7c453d')
	}
	);

POST	

Delete a post	db.post.deleteOne(
	{
	"_id": ObjectId('61dbfbd392a81a85adeaf095')
	}
	);

Delete all posts of a videogame community	db.post.deleteMany(
	{
	community.community_id:
	ObjectId('61dbfbbe92a81a85adeaf04c')
	}
	);

COMMENT	

Delete a comment	db.comment.deleteOne(
	{
	"_id": ObjectId('61dbfc0692a81a85adebe691')
	}
	);

Delete all comments of a post	db.comment.deleteMany(
	{
	post.post_id: ObjectId(‘61dbfbd392a81a85adeaf095’)
	}
	);

Delete all comments of a videogame community	db.comment.deleteMany(
	{
	post.community_id:
	ObjectId('61dbfbbe92a81a85adeaf04c')
	}
	);

Aggregations

Post collection: best users by number of posts in a given videogame community

```
db.post.aggregate([
	{
		$match: {
			'community.community_id':##id
		}
	},
	{
		$group: {
			_id: '$author',
			num_post: {
				$count: {}
			}
		}
	},
	{
		$sort: {
			num_post: -1
		}
	},
	{
		$limit: ##limit
	}
])
```

- **Description:** Select the users who wrote more posts in a videogame community
- **Parameters:** videogame community id (##id); how many users to return (##limit).
- **Java method:** PostService.bestUsersByNumberOfPosts

Post collection: best videogame community by number of likes obtained in a given time period

- **Description:** Select the videogame communities whose posts, written in a given time period, received (as a whole) the highest number of likes.
- **Parameters:** start of time period (##from); end of time period (##to); how many videogame communities to return (##limit).
- **Java method:** PostService.bestVideogameCommunities

```
db.post.aggregate([{$match: {
  timestamp: {
    $gt: ##from,
    $lt: ##to
  }
}
}, {
    $group: {
        _id: {
            id: '$community.community_id',
            name: '$community.community_name'
        },
        sum_likes: {
            $sum: '$likes'
        }
    }
}, {
    $sort: {
        sum_likes: -1
    }
}, {
    $limit: ##limit
}])
```

We group by both community id and community name so that we can return the name of the videogame without having to make a read access to the videogame collection.

Comment collection: average number of comments per post

```
db.comment.aggregate([{$group: {
 _id: {
  post_id: '$post.post_id',
  community_id: '$post.community_id',
  community_name: '$post.community_name'
 },
 comments_per_post: {
  $count: {}
 }
}}, {$group: {
 _id: {
  community_id: '$_id.community_id',
  community_name: '$_id.community_name'
 },
 avg_comments_per_post: {
  $avg: '$comments_per_post'
 }
}}, {$sort: {
 avg_comments_per_post: -1
}},
{$skip: ##skip},
{$limit: ##limit}
])
```

- **Description:** Compute the average number of comments per post in every videogame community.
- **Parameters:** none
- **Java method:** CommentService.averageNumberOfCommentsPerPost

Comment collection: average number of comments per each post of a user

```
db.comment.aggregate([{$group: {
 _id: {
  post: '$post.post_id',
  author: '$post.author'
 },
 comments_per_user: {
  $count: {}
 }
}}, {$group: {
 _id: {
  author: '$_id.author'
 },
 avg_comments_per_user: {
  $avg: '$comments_per_user'
 }
}}, {$sort: {
 avg_comments_per_user: -1
}},
{$skip: ##skip},
{$limit: ##limit}
])
```

- **Description:** Compute the average number of comments per each post of a user
- **Parameters:** none
- **Java method:** CommentService.averageNumberOfCommentsPerUser

Sharding

In order to improve the availability of our document database, we designed a sharding method of our data.

- For the *user* and *admin* collection, we have chosen the field **username** as the *sharding key*, since the most frequent CRUD operations related to these two collections have a match filter on this field.
- For the *videogame*, *post* and *comment* collections, we have chosen the field containing the **id of the videogame community** as *sharding key* (i.e., _id for videogame, community.community_id for post and post.community_id for comment). In fact, thanks to the structure of our application (as shown in the use cases diagram), the user will always read/write only the posts and comments of the videogame community he/she is currently viewing. By segmenting the data by the id of the videogame community, we will put together data frequently accessed at the same time.

In order to more evenly distribute the CRUD operations in the cluster, we propose to adopt a **hashing strategy** for the sharding.

This is very useful since the sharding keys of videogame, post and comment are a monotonically increasing field (it's an ObjectId), thus it doesn't fit with a ranged strategy nor a list one.

On the other hand, the username field is chosen by the user, so no assumptions on its distribution can be made.

A hashing strategy will allow the data to be distributed more equally in the nodes of the cluster.

Design of Neo4j

We decided to represent in the graph only the nodes and properties necessary for the queries executed on the Neo4j database. In particular, every node stores a MongoDB key (so that is possible to get references between both databases) and some relationship stores the creation timestamp (so that we can exploit it in the query that analyze the graph to find the influencers).

Nodes

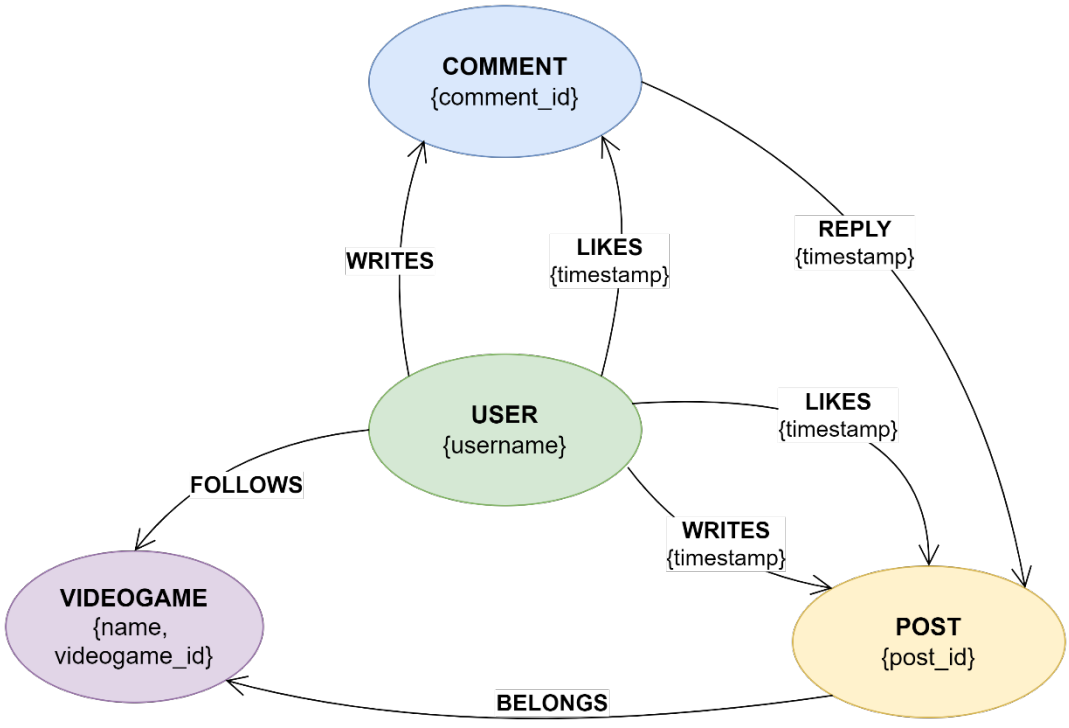
Volume: ≈ 370,000 nodes

- **User:** represent a user of the application. It has the property *username*.
- **Videogame**: represent a videogame community. It has a property *videogame_id*, which contains the MongoDB ObjectId of the corresponding document in the Videogame collection, and a property *name*, which stores the name of the videogame.
- **Post**: represent a post. It has a property *post_id*, which contains the MongoDB ObjectId of the corresponding document in the Post collection
- **Comment**: represent a comment. It has a property *comment_id*, which contains the MongoDB ObjectId of the corresponding document in the Comment collection

Relationships

Volume: ≈ 10,000,000 relationships

- **FOLLOWS** (User → Videogame): this relationship is added when a user follows a videogame community.
- **BELONGS** (Post → Videogame): this relationship is added when a post is created, in order to link it to its videogame community.
- **WRITES** (User → Post): this relationship is added when a user writes a post. It has a property *timestamp*, which stores the timestamp when the relationship was created.
- **LIKES** (User → Post): this relationship is added when a user likes a post. It has a property *timestamp*, which stores the timestamp when the relationship was created.
- **REPLY** (Comment → Post): this relationship is added when a comment is created, in order to link it to its post. It has a property *timestamp*, which stores the timestamp when the relationship was created.



- **WRITES** (User → Comment): this relationship is added when a user writes a comment.
- **LIKES** (User → Comment): this relationship is added when a user likes a comment. It has a property *timestamp*, which stores the timestamp when the relationship was created.

CRUD operations

CREATE

Description	Cypher implementation
Create a User node	`MERGE (u: User {username: $username})`
Create a Videogame node	`MERGE (v: Videogame {videogame_id: $id, name: $name})`

Create a Post node	`MATCH (u:User {username: $user})`
	`MATCH (v:Videogame {videogame_id: $videogame})`
	`MERGE (v)<-[:BELONGS]-(p:Post {post_id: $post})<-`
	`[:WRITES {timestamp: datetime()}]-(u)`

Create a Comment node	`MATCH (u:User {username: $author})`
	`MATCH (p:Post {post_id: $post})`
	`MERGE (p)<-[r:REPLY]-(c: Comment {comment_id: $id})<-`
	`[:WRITES]-(u)`
	`ON CREATE SET r.timestamp = datetime()`

Create a FOLLOWS relationship	`MATCH (v: Videogame {videogame_id: $videogame})`
	`MATCH (u:User {username: $user})`
	`MERGE (u)-[:FOLLOWS]->(v)`

Create a LIKES relationship (User-Post)	`MATCH (p:Post {post_id: $id})`
	`MATCH (u:User {username: $user})`
	`MERGE (u)-[r:LIKES]->(p)`
	`ON CREATE SET r.timestamp = datetime()`

Create a LIKES relationship (User-Comment)	`MATCH (c:Comment {comment_id:$id})`
	`MATCH (u:User {username: $user})`
	`MERGE (u)-[r:LIKES]->(c)`
	`ON CREATE SET r.timestamp = datetime()"`

READ

Description	Cypher implementation
Find videogames followed by a user	`MATCH (:User {username: $username})-[:FOLLOWS]->(v:Videogame)` `RETURN v as videogame`

Get all the posts liked by a user in a given videogame community	`MATCH (u: User {username: $username})-[:LIKES]->(p:Post)-`
	`[BELONGS]->(v:Videogame {videogame_id: $videogame})`
	`RETURN p.post_id as post`

Get all the comments liked by a user in a given post	`MATCH (u: User {username: $username})-[:LIKES]->(c:Comment)-`
	`[:REPLY]->(p:Post {post_id: $post})`
	`RETURN c.comment_id as comment`

DELETE

Description	Cypher implementation

Delete a Videogame node (cascade effect on related posts and comments)	`MATCH (v:Videogame)<-[:BELONGS]-(:Post)<-[:REPLY]-(c:Comment)`
	`where v.videogame_id = $id detach delete c;`
	`MATCH (v:Videogame)<-[:BELONGS]-(p:Post)`
	`where v.videogame_id = $id detach delete p;`
	`MATCH (v:Videogame)`
	`where v.videogame_id = $id detach delete v;`

Delete a Post node (cascade effect on related comments)	`MATCH (p:Post)<-[r:REPLY]-(c:Comment)`
	`WHERE p.post_id = $post_id`
	`DETACH DELETE c;`
	`MATCH (p: Post {post_id: $id})`
	`DETACH DELETE p;`

Delete a Comment node	`MATCH (c:Comment{comment_id: $id})` `DETACH DELETE c;`
Delete a FOLLOWS relationship	`MATCH (u:User {username: $user})-[f:FOLLOWS]->(v:Videogame {videogame_id: $videogame})` `DELETE f`
Delete a LIKES relationship (User-Post)	`MATCH (u:User {username:$author})-[r:LIKES]->(p:Post {post_id: $id})` `DELETE r`
Delete a LIKES relationship (User-Comment)	`MATCH (u:User {username: $author})-[r:LIKES]->(c:Comment {comment_id: $id})` `DELETE r`

On-graph queries

Suggest new videogame communities to a user

Domain centric	Graph centric
Find what videogame communities are followed by the highest number of users, who share at least one followed videogame with the given user	Starting from a User node, count the paths like **(u:User)→(v:Videogame)→ (u2:User)→(v2:Videogame)** for each Videogame node **v2** in the graph, provided that **u≠u2** and there is no FOLLOWS relationship between **u** and **v2**

```
MATCH (target_user: User)-[tg:FOLLOWS]->(his_videogame:Videogame)
WHERE target_user.username = $username
WITH COLLECT(his_videogame.name) AS game_not_to_suggest

MATCH (his_videogame:Videogame)<-[ou:FOLLOWS]-(other_user: User)-[ og: FOLLOWS ]->(
other_videogame: Videogame)
WHERE (his_videogame.name IN game_not_to_suggest
AND NOT other_videogame.name IN game_not_to_suggest)
```

```
RETURN DISTINCT other_videogame AS VIDEOGAME, count(og) AS number_of_following
ORDER BY number_of_following DESC
LIMIT $limit
```

Java method: VideogameCommunityServiceImpl.neo4jSuggestedVideogames

Identify influencer users

Domain centric	Graph centric
Count the number of likes and comments received by the posts of each user during a chosen time period	For each User node in the database, find all the Post node connected by a :WRITES relationship. For each of those Post nodes, count the number of incoming arches :LIKES and :REPLY with timestamp property between two fixed extremes. Finally, sum the partial counts.

```
MATCH (influencer: User)-[write:WRITES]->(posts:Post)
MATCH (user_who_liked: User)-[liked:LIKES]->(posts:Post)
WHERE date(liked.timestamp) >= date($from) AND date(liked.timestamp) <= date($to)

OPTIONAL MATCH (all_comments: Comment)-[commented:REPLY]->(posts:Post)
WHERE date(commented.timestamp) >= date($from) AND date(commented.timestamp) <=
date($to)

RETURN DISTINCT
        influencer.username AS INFLUENCER,
        (count(DISTINCT user_who_liked) + count(DISTINCT all_comments) ) AS GRADE
ORDER BY GRADE DESC
LIMIT $limit
```

Java method: AdminServiceImpl.neo4jInfluencerQuery

Consistency between databases

The non-functional requirements of our application don't consider **strict consistency**. Thus, we can accept some temporary inconsistencies in our database, in order to improve the latency of server's responses (*eventual consistency*)

Several operations require to write in both MongoDB and Neo4j databases, e.g. the creation of a new entity document/node or putting a like to a post or comment, so we must analyze if and when eventual consistency is acceptable.

Create/delete a videogame, post or comment

For VideogameCommunity, Post and Comment entities, the workflow is the following:

- Insert/delete on **MongoDB**
- Insert/delete on **Neo4j**

If the second operation fails, it will leave the MongoDB database in an inconsistent state.

However, we can accept these inconsistencies. In fact, if the second operation fails, the user is notified and can try again to insert a new entity with the same properties, without causing a collision with the already existing document in the database (the key for these collections is automatically generated by MongoDB). Moreover, the user will still be able to view the inconsistent post and eventually to delete it.

So, we **don't require to execute a rollback in case of failure of the second operation.**

Add/remove a like

Since the *LIKES* relationship is represented in Neo4j and "duplicated" in MongoDB as redundancy, the workflow is the following:

- Insert/delete on **Neo4j**
- Insert/delete on **MongoDB**

If the second operation fails, it will leave the Neo4j database in an inconsistent state.

However, **we can accommodate these inconsistencies** because they aren't critical information of the system. In this way, we will gain better availability and less latency.

Create a user

The workflow is the following:

- Insert document in **MongoDB**
- Create node in **Neo4j**

If the second operation fails, it will leave the MongoDB database in an inconsistent state.

We can't tolerate this inconsistency for the User entity. Since the username is a second key for the document in the User collection, any following attempt to create the same user will fail. This is unacceptable because it will prevent the user from using the application.

Thus, **we must execute a rollback in MongoDB if the operation in Neo4j fails.**

Final notes

In order to reduce the probability of causing an inconsistent state, **we must check if both databases are currently reachable before executing the two operations.**

Index Analysis

We have evaluated different indexes to improve read performances on the databases.

MongoDB: Videogame collection

The read queries on the videogame collection are the search and the browse operations. The first one involves a **filter** on the **name** field, whereas the latter sort involves a **sorting** on the **name** field.

Thus, it sounds reasonable to introduce an index on the name field.

SEARCH

```
db.videogame.find(
	{
		name: {
			$regex: 'battle',
			$options: 'i'
		}
	}
)
```

Base case:

```
executionTimeMillis: 17
totalKeysExamined: 0
totalDocsExamined: 132
```

Index:

```
executionTimeMillis: 1,
totalKeysExamined: 132,
totalDocsExamined: 3,
```

BROWSE

```
db.videogame.find({}).sort({"name": 1})
```

Base case:

```
executionTimeMillis: 94
totalKeysExamined: 0
totalDocsExamined: 132
```

Index:

```
executionTimeMillis: 1
totalKeysExamined: 132
totalDocsExamined: 132
```

As we can see, the index radically improves the performance of our query, thus it will be introduced in our database.

MongoDB: User collection

The read queries on the user collection involve a filter or a sort on the **username** field. Moreover, the data model of the application requires that the username field must be **unique**. Thus, we evaluated the opportunity to introduce a unique index on the username field.

LOGIN

```
db.user.find({$and: [{username:"whitefish666"},
{$or: [{"isBlocked": {$not: {$exists: ""}}}, {"isBlocked": false}]}]})
```

Base case:

```
executionTimeMillis: 27
totalKeysExamined: 0
totalDocsExamined: 800
```

Index:

```
executionTimeMillis: 0
totalKeysExamined: 1
totalDocsExamined: 1
```

BROWSE

```
db.user.find({}).projection({"password": 0}).sort({"username": 1})
```

Base case:

```
executionTimeMillis: 11
totalKeysExamined: 0
totalDocsExamined: 800
```

Index:

```
executionTimeMillis: 4
totalKeysExamined: 800
totalDocsExamined: 800
```

As we can see, the index radically improves the performance of our query, thus it will be introduced in our database.

MongoDB: Admin collection

The application requires that the **username** field must be unique, thus we will introduce a unique index on the username field.

Since the collection is very small and writes are very rare, there is no need (nor way) to test its performance.

MongoDB: CRUD operations on Post collection

The heaviest CRUD query on the post collection is the BROWSE, which get all the posts of a given community and return them sorted by timestamp or likes and (in some cases) filtered by the isAuthorInfluencer field.

The other CRUD query involves the _id field of the post (which is already indexed) or only the videogamecommunity id (e.g. delete by videogamecommunity id).

Since the likes field is subjected to change frequently, whereas the videogamecommunity id, the timestamp and the isAuthorInfluencer field are fixed at creation set, it sounds reasonable to discuss the introduction an index on these three fields (or on a subset).

BROWSE (likes, no influencer)

```
db.post.find({"community.community_id":
ObjectId('61dbfbbe92a81a85adeaf04c')}).sort(likes: -1)
```

Base case:

```
executionTimeMillis: 63
totalKeysExamined: 0
totalDocsExamined: 62970
```

Index on community.community_id:

```
executionTimeMillis: 3
totalKeysExamined: 500
totalDocsExamined: 500
```

BROWSE (timestamp, no influencer)

```
db.post.find({"community.community_id":
ObjectId('61dbfbbe92a81a85adeaf04c')}).sort(timestamp: -1)
```

Base case:

```
executionTimeMillis: 117,
totalKeysExamined: 62970
totalDocsExamined: 62970
```

Index on community.community_id (1):

```
executionTimeMillis: 3
totalKeysExamined: 500
totalDocsExamined: 500
```

Index on community.community_id (1) + timestamp (-1)

```
executionTimeMillis: 1
totalKeysExamined: 500
totalDocsExamined: 500
```

BROWSE (likes, influencer)

```
db.post.find({"community.community_id": ObjectId('61dbfbbe92a81a85adeaf04c'),
“isAuthorInfluencer”: true}).sort(likes: -1)
```

Base case:

```
executionTimeMillis: 126
totalKeysExamined: 62970
totalDocsExamined: 62970
```

Index on community.community_id (1):

```
executionTimeMillis: 5,
totalKeysExamined: 500,
totalDocsExamined: 500,
```

Index on community.community_id (1) + isAuthorInfluencer(1)

```
executionTimeMillis: 0,
totalKeysExamined: 129,
totalDocsExamined: 129,
```

BROWSE (timestamp, influencer)

```
db.post.find({"community.community_id": ObjectId('61dbfbbe92a81a85adeaf04c'),
"isAuthorInfluencer": true}).sort({timestamp: -1})
```

Base case:

```
executionTimeMillis: 126
totalKeysExamined: 62970
totalDocsExamined: 62970
```

Index on community.community_id (1):

```
executionTimeMillis: 1
totalKeysExamined: 500
totalDocsExamined: 500
```

Index on community.community_id (1) + timestamp (-1)

```
executionTimeMillis:
1
totalKeysExamined:
500
totalDocsExamined: 500
```

Index on community.community_id (1) + isAuthorInfluencer (1)

```
executionTimeMillis: 0
totalKeysExamined: 129
totalDocsExamined: 129
```

Index on community.community_id (1) + isAuthorInfluencer (1) + timestamp (-1)

```
executionTimeMillis: 0
totalKeysExamined: 129
totalDocsExamined: 129
```

For this query, the introduction of an index on the field community.community_id and an index on the fields **community.community_id + isAuthorInfluencer** yield the best tradeoff between performance for read operations and on write operations (each index must be kept updated!). Since the two indexes share the first field, we can use only the second one and achieve the same performance.

MongoDB: aggregations on Post collection

The Post collection also have two aggregations:

1. **Best users by number of posts in a given community**, which match on the community id.
2. **Best videogame communities by amount of likes in a given time period**, which match on the timestamp field.

According to MongoDB documentation "*Aggregation Pipeline Optimization*", only the **MATCH** stage can be optimized through the introduction of an index. Thus, it seems reasonable to discuss the introduction of two indexes:

- An index on the community.community_id field (already introduced for the CRUDs)
- An index on the timestamp field

Best users by number of posts in a given community

Base case:

```
executionTimeMillis: 54,
totalKeysExamined: 0,
totalDocsExamined: 62970
```

Index on community.community_id (1) + isAuthorInfluencer (1):

```
executionTimeMillis: 2
totalKeysExamined: 500
totalDocsExamined: 500
```

The results clearly show a performance improvement after the introduction of the index.

Best videogame communities by amount of likes in a given time period

Base case:

```
executionTimeMillis: 203
totalKeysExamined: 0
totalDocsExamined: 62970
```

Index on timestamp (-1):

```
executionTimeMillis: 141,
totalKeysExamined: 28357,
totalDocsExamined: 28357,
```

The results clearly show a performance improvement after the introduction of the index.

MongoDB: CRUD operations on Comment

The only CRUD query that doesn't match on the _id is the one which view the comments of a post.

GET COMMENTS OF A POST

```
db.comment.find({"post.post_id":
ObjectId('61dbfbd392a81a85adeaf099')}).sort({timestamp: -1})
```

Base case:

```
executionTimeMillis: 203
totalKeysExamined: 0
totalDocsExamined: 302657
```

Index on post.post_id (1):

```
executionTimeMillis: 1
totalKeysExamined: 571
totalDocsExamined: 571
```

The data clearly shows that the usage of an index improves the performance of the reads

MongoDB: aggregations on Comment collection

According to MongoDB documentation "*Aggregation Pipeline Optimization*", only the **MATCH** stage can be optimized through the introduction of an index. Since none of the two aggregations on the Comment collection have a match stage, no index will be introduced.

Summary for MongoDB

INDEX'S NAME	FIELDS
User	
username	username (1) [UNIQUE]
Admin	
username	username (1) [UNIQUE]
Videogame	
name	name (1)
Post	
community.community_id_1_isAuthorInfluencer_1	community.community_id (1); isAuthorInfluencer (1)
timestamp_-1	timestamp (-1)
Comment	
post.post_id_1	post.post_id (1)

Neo4j Indexes

Indexes on **User.username, Videogame.videogame_id, Post.post_id** and **Comment.comment_id** will be built, since both CRUDs and analytics on the Neo4j graph use these properties to find the nodes inside the graph database.

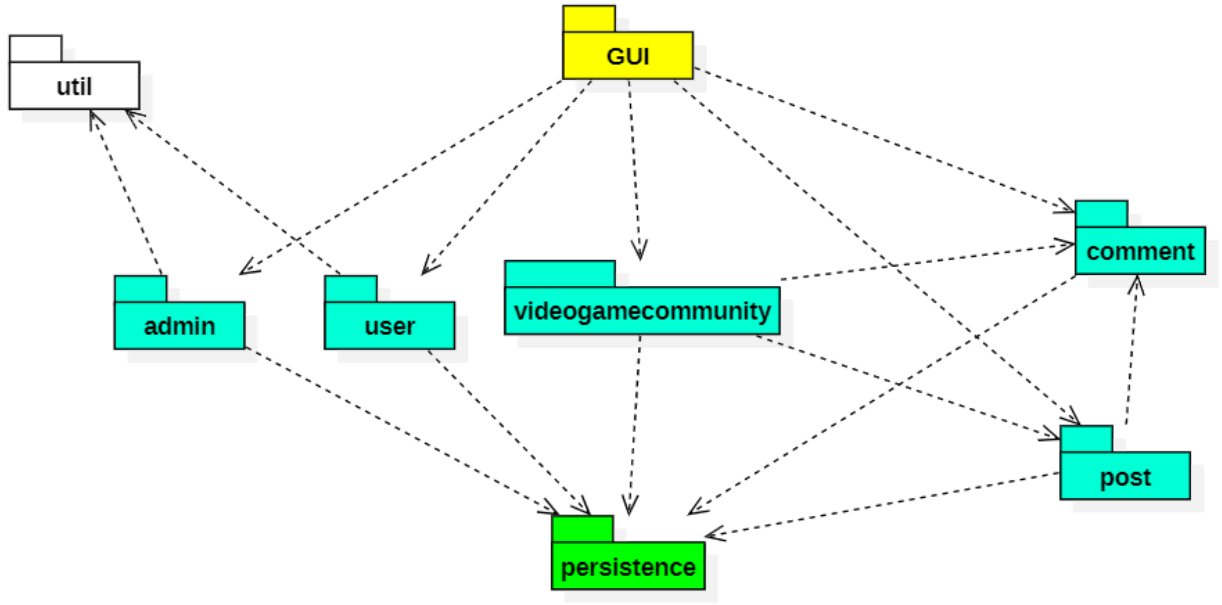
Architecture of the application

Client side

Our application has been developed in **Java 11**, with the support of the following external libraries or frameworks:

- **MongoDB Java Sync driver** [4.4.1]: necessary to use MongoDB
- **Neo4j Java driver** [4.4.2]: necessary to use Neo4j
- **Log4j2** [2.17.0]: console and file logging
- **Junit** [5.8.2]: tests
- **Mockito** [4.3.1]: writing mocks for unit tests
- **Apache Commons Codec** [1.15]: support to SHA-256 hash computation
- **JavaFX** [17.0.2]: graphical user interface

Packages



We have chosen to structure our packages **by feature**, since it was a more intuitive organization for our type of application.

- **GUI**: it generates the graphical user interface of Gameflows and handles user's requests by calling the methods provided by the middleware packages (the cyan one).
- **admin**: contains all the java classes that are in charge of handling all the data revolving the admin entity.
- **user**: contains all the java classes that are in charge of handling all the data revolving the user entity.
- **videogamecommunity**: contains all the java classes that are in charge of handling all the data revolving the entity "videogame community".
- **post**: contains all the java classes that are in charge of handling all the data revolving the post entity.
- **comment**: contains all the java classes that are in charge of handling all the data revolving the comment entity.

- **util**: offers to admin and user packages the functionalities to handle the passwords
- **persistence**: handles the connection to the MongoDB cluster and the Neo4j server.

Server side

MongoDB

In order to meet the requirements of fault-tolerance (keep the service available even if a server goes down and avoid a single point of failure), a cluster has been set up.

REMOTE CLUSTER			
IP address	**Port**	**MongoDB version**	**Role**
172.16.4.42	27020	5.0	PRIMARY
172.16.4.43	27020	5.0	SECONDARY (1)
172.16.4.44	27020	5.0	SECONDARY (2)
LOCAL CLUSTER			
IP address	**Port**	**MongoDB version**	**Role**
localhost	27018	5.0	PRIMARY
localhost	27019	5.0	SECONDARY (1)
localhost	27020	5.0	SECONDARY (2)

Replica set configuration of remote cluster

The replica set was created with the following configuration:

```
rsconf = {
	_id: "lsmdb",
	members: [
		{_id: 0, host: "172.16.4.42:27020", priority: 100},
		{_id: 1, host: "172.16.4.43:27020", priority: 50},
		{_id: 2, host: "172.16.4.44:27020", priority: 1}
	]
};
```

Replica set configuration of local cluster

The replica set was created with the following configuration:

```
rsconf = {
	_id: "lsmdb",
	members: [
		{_id: 0, host: "localhost:27018", priority: 100},
		{_id: 1, host: "localhost:27019", priority: 50},
		{_id: 2, host: "localhost:27020", priority: 1}
	]
};
```

Write concern and read preference

In our application, read operations are more frequent and heavier than the write operations. In addition to that, our non-functional requirements are *availability* and *fast response time*.

For these reasons we decided to set:

- **Write concern: 1**
- **Read preference: nearest**

This configuration will guarantee fast responses both in write and reads operations, while still meeting the *eventual consistency* requirement.

Neo4j

A single neo4j server has been deployed for our application. In order to reduce both the overloading of the server and the chances of a server failure (that would make the graph database unreachable), we decided to put the server on the host with less priority to become MongoDB primary, i.e. host **172.16.4.44** on the remote cluster on the port **7687**.

For what concerns the local cluster, the Neo4j server is reachable from the port 7687 through a bolt connection.

Implementation in Java

Modules

it.unipi.dii.inginf.lsdb.gameflows

Name	Visibility	Type	Description
Main	Public	Class	Main class of the application. It launches the JavaFX Application instance

it.unipi.dii.inginf.lsdb.gameflows.admin

Name	Visibility	Type	Description
Admin	Public	Class	Bean class that represents an admin of the social network
AdminService	Public	Interface	Interface which offers a set of methods which allows the application to fulfill functionalities requested by an Admin
AdminServiceFactory	Public	Class	Factory which correctly allocate AdminService objects (singleton)
AdminServiceImpl	Package	Class	Implementation of AdminService

it.unipi.dii.inginf.lsdb.gameflows.comment

Name	Visibility	Type	Description
Comment	Public	Class	Bean class that represents a comment in the social network
CommentService	Public	Interface	Interface which offers methods to handle the comment entity in the databases
CommentServiceFactory	Public	Class	Factory which correctly allocate CommentService objects (singleton)
CommentServiceImpl	Package	Class	Implementation of CommentService
InfoPost	Public	Class	Redundant information of the post to which a comment reply (extracted from MongoDB)
LikedCommentsCache	Public	Class	Class which implements a cache system for the comments liked by the user under a specified post. The class handles both the CRUD queries on the remote databases and the CRUD on the corresponding local copy.
ResultAverageCommentPerPost	Public	Class	Class that stores the result of the aggregation which computes the

			average number of comments per post
ResultAverageCommentPerUser	Public	Class	Class that stores the result of the aggregation which computes the average number of comments per user

it.unipi.dii.inginf.lsdb.gameflows.gui

it.unipi.dii.inginf.lsdb.gameflows.gui.controller

Name	Visibility	Type	Description
AdminHomeController	Public	Class	Controller class of Admin Home page. It stores all the main functionalities used by the administrator
AdminUserInfoPageController	Public	Class	Controller class of user information page. It shows to the admin all the information about the users.
CreateNewAdminController	Public	Class	Controller class of Create Admin Page. It allows the admin to create a new admi profile.
CreateNewVideogameCommunityController	Public	Class	Controller class of create videogame page. It allows the admin to create a new videogame community
DeleteCommunityAlertController	Public	Class	Controller class of delete community alert page. Let the admin choose if delete a videogame community or not.
ItemCommentController	Public	Class	Controller class of Item Comment. It is used to create comments items.
ItemInfluencerUserController	Public	Class	Controller class of Item Influencer user. It is used to show the Influencer users with
ItemPostController	Public	Class	Controller class of item Post. It is used to create post items.
ItemUserController	Public	Class	Controller class of User item. It is used to create user items.

ItemUserWithStatsController	Public	Class	Controller class of User with stats item. It is used to create User items with specific statistics.
ItemVideogameController	Public	Class	Controller class of Videogame item. It is used to create videogame items.
ItemVideogameWithStatsController	Public	Class	Controller class of Videgame with stats item. It is used to create videogames items with specific statistics.
LoginController	Public	Class	Controller class of login page. It is possible to make the login as a user or an admin. It is also possible to register a new user
UserCommunityPageController	Public	Class	Controller class of a videogame community page. It stores all the methods used on the videogame communities
UserCreateCommentPageController	Public	Class	Controller class of a create comment page. Allow a user to create a new comment
UserCreatePostPageController	Public	Class	Controller class of a create post page. Allow a user to create a new post
UserHomeController	Public	Class	Controller class of user home page. It stores all the main functionalities used by a user.
UserPostPageController	Public	Class	Controller class of post page. It shows a specific post with its comments and allow to make operations on it.
UserSignUpPageController	Public	Class	Controller class of sign-up page. It allows users registration
UserVideogameInfoPageController	Public	Class	Controller class of videogame info page. It shows all information stored about a specific videogame community.

ViewBestUsersByNumberOfPostsController	Public	Class	Controller class of best user by post page. It allows to research a list of the best users in a specific date interval.
ViewInfluencerRankingController	Public	Class	Controller class of influencer ranking page. It allows to research a list of influencers and also give the influencer badge to users

it.unipi.dii.inginf.lsdb.gameflows.gui.controller.listener

Name	Visibility	Type	Description
PostListener	Public	Interface	Interface which offers the method to check if a post item is clicked
UserListener	Public	Interface	Interface which offers the method to check if a user item is clicked
VidegameListener	Public	Interface	Interface which offers the method to check if a videogame item is clicked

it.unipi.dii.inginf.lsdb.gameflows.gui.model

Name	Visibility	Type	Description
AdminResearchMode	Public	Enum	Enum which represents admin research modes (for users and videogames)
AdminResearchTypeMode	Public	Enum	Enum which represents admin research types (users or communities)
CommunityPageResearchMode	Public	Enum	Enum which represent user research mode (for posts and comments)
HomeResearchMode	Public	Enum	Enum which represent user research mode (for videogames)
LoginMode	Public	Enum	Enum which represent the login mode (user or admin)

it.unipi.dii.inginf.lsdb.gameflows.persistence

Name	Visibility	Type	Description
DatabaseConfiguration	Public	Interface	Interface which offers methods to retrieve the configuration options written in the `dbconfig.properties` file
DatabaseConfigurationImpl	Package	Class	Implementation of DatabaseConfiguration
GameflowsCollection	Public	Enum	Enum which represents the collections in the MongoDB database
MongoConnection	Public	Interface	Interface which offers methods to correctly set up the connection instances to a MongoDB server or cluster. Extends *AutoClosable*
MongoConnectionImpl	Package	Class	Implementation of MongoConnection
Neo4jConnection	Public	Interface	Interface which offers methods to correctly set up the connection instances to the Neo4j server. Extends *AutoClosable*
Neo4jConnectionImpl	Package	Class	Implementation of Neo4jConnection
PersistenceFactory	Public	Class	Factory which correctly allocate instances of DatabaseConfiguration (*singleton*), MongoConnection and Neo4jConnection

it.unipi.dii.inginf.lsdb.gameflows.post

Name	Visibility	Type	Description
InfoVideogameCommunity	Public	Class	Class that stores the redundant information of the videogame community, retrieved from the documents of the Post collection in MongoDB
LikedPostsCache	Public	Class	Class which implements a cache system for the posts liked by the user in a given videogame community. The class handles both the CRUD queries on the remote databases and the CRUD on the corresponding local copy.
Post	Public	Class	Bean class that represents a post in the social network
PostFilter	Public	Enum	Select the sorting field for browsing the posts in a videogame community: • PostFilter.date will order posts by timestamp

			• PostFilter.like will order posts by number of likes
PostService	Public	Interface	Interface which offers methods to handle the post entity in the databases
PostServiceFactory	Public	Class	Factory which correctly allocate PostService objects (singleton)
PostServiceImpl	Package	Class	Implementation of PostService
ResultBestUserByPostAggregation	Public	Class	Contains a result row of the "best users by number of posts" aggregation
ResultBestVideogameCommunityAggregation	Public	Class	Contains a result row of the "best videogame communities by number of likes" aggregation

it.unipi.dii.inginf.lsdb.gameflows.user

Name	Visibility	Type	Description
User	Public	Class	Bean class that represents a user of the social network
UserService	Public	Interface	Interface which offers a set of methods which allows the application to handle the login functionalities and the user entity in the database
UserServiceFactory	Public	Class	Factory which correctly allocate UserService objects (singleton)
UserServiceImpl	Package	Class	Implementation of UserService

it.unipi.dii.inginf.lsdb.gameflows.util

Name	Visibility	Type	Description
Password	Public	Class	Represent a password of the gameflows database. A Password object is composed by the SHA-256 hash of the real password and a randomly generated salt (in order to prevent rainbow table attacks)

it.unipi.dii.inginf.lsdb.gameflows.videogamecommunity

Name	Visibility	Type	Description
FollowedVideogameCache	Public	Class	Singleton class which implements a cache system for the followed communities of the user. The class handles both the CRUD queries on from the remote

			databases and the CRUD on the corresponding local copy. In this way, we will reduce the number of requests to the databases.
FollowedVideogames	Package	Class	Maintains a local copy of the followed videogame communities by the user of the application. It offers methods to keep the local copy updated.
VideogameCommunity	Public	Class	Bean class that represents a videogame community of the social network
VideogameCommunityService	Public	Interface	Interface which offers methods to handle the videogame community entity in the databases
VideogameCommunityServiceFactory	Public	Class	Factory which correctly allocate VideogameCommunityService objects (singleton)
VideogameCommunityServiceImpl	Package	Class	Implementation of VideogameCommunityService

Relevant code snapshots

Factory of “entity service” classes

```
public class UserServiceFactory {

    private static UserService service = null;

    public static UserService getService () {
        if (service == null) {
            service = new UserServiceImpl();
        }
        return service;
    }
}
```

The “entity service” classes (those who offer methods to handle the entities inside the databases, e.g. UserService) are designed as **singleton** classes, i.e. only one instance of those classes is allowed to exist inside the application. Since the actual implementations of the service classes have package-level visibility, this is achieved by using **factories**.

The picture on the left shows the factory for UserSevice. The schema is basically the same for the other classes.

Handling of MongoDB or Neo4j connection

```
@Override
public ObjectId insertPost(@NotNull Post post) {
    try (MongoConnection mongoConnection = PersistenceFactory.getMongoConnection();
         Neo4jConnection neo4jConnection = PersistenceFactory.getNeo4jConnection()
    ) {
        LOGGER.info("insertPost() | Insert post");
        return insertPost(mongoConnection, neo4jConnection, post);

    } catch (Exception e) {
        LOGGER.error("insertPost() | " +
            "Unable to close MongoConnection or Neo4jConnection instance " +
            "due to error: " + e);
    }
    return null;

}
```

MongoConnection and Neo4jConnection are Autoclosable objects. We can exploit this feature by creating the objects inside a try-with-resources, which will automatically call the close() method when it will exit from the try-catch blocks.

Cache mechanisms

In order to optimize the performance of the application, we introduced some sort of caching mechanisms for the followed videogame communities and the liked posts and comments by the user.

In fact, the list of the followed videogame communities and the list of what posts has been liked by the user are generated by executing Neo4j queries. Since the Neo4j database is not distributed in our implementation and this information are modified only by the user that is using the application, we decided to store a local copy of the lists. Every time the list is modified by the user, we execute an update query on the remote database, and we modify accordingly the local copy. In this way, we will avoid executing a read operation every time a user follows a new community or likes a post.

Videogame cache

The cache of the videogame is handled by the classes FollowedVideogameCache and FollowedVideogames (package videogamecommunity).

FollowedVideogameCache offers the methods to follow/unfollow a videogame community. It generates a local copy of the remote state if it doesn't exist and then proceeds to update both the

```java
public class FollowedVideogameCache {
    // Logger
    private static final Logger LOGGER = LogManager.getLogger(FollowedVideogameCache.class);

    // Instance of the singleton object
    private static FollowedVideogameCache instance = null;
    // User who follows the listed communities
    private static User user = null;

    // List of followed communities
    private final FollowedVideogames followedVideogames;

    /**
     * Return a singleton instance of the class
     * @param currentUser current logged user
     * @return instance of FollowedVideogameCache
     */
    public static FollowedVideogameCache getInstance (@NotNull User currentUser) {
        // Check if instance for the current user has already been allocated
        if (instance == null ||
            user == null ||
            !currentUser.getUsername().equals(user.getUsername())
        ){
            LOGGER.info("getInstance() | Generate FollowedVideogameCache instance");
            user = currentUser;
            instance = new FollowedVideogameCache(currentUser);
        }

        return instance;
    }

    /**
     * Private constructor: it fetches data from remote databases
     * @param user current user
     */
    @TestOnly
    FollowedVideogameCache (@NotNull User user) {
        followedVideogames = VideogameCommunityServiceFactory
                .getImplementation()
                .viewFollowedVideogames(user);
    }
```

local copy and the remote database every time a user follows/unfollows a community. Moreover, it queries the local copy in order to check if a given videogame is followed by the user.

```
public boolean followVideogameCommunity (@NotNull VideogameCommunity videogameCommunity,
                                         boolean follow)
{
    LOGGER.info("followVideogameCommunity()");

    // Update the remote databases first (original copy)
    boolean ret = VideogameCommunityServiceFactory
            .getImplementation()
            .followVideogameCommunity(
                    videogameCommunity,
                    user,
                    follow
            );

    // If the database query was successful, update the local copy
    if (ret) {
        if (follow) {
            followedVideogames.add(videogameCommunity.getId(), videogameCommunity.getName());
        }
        else {
            followedVideogames.remove(videogameCommunity.getId());
        }
    }

    // Return the result of the remote query
    return ret;
}
```

FollowedVideogames stores the local copy of the list of the followed communities both as a *map* (so that we can quickly check if a certain videogame is followed) and as a *list* (so that we can efficiently print all the followed communities).

```java
class FollowedVideogames {

    // Two data structure used to organize the videogame communities
    // both sequentially (list) and by id (map)
    private final Map<ObjectId, String> map = new HashMap<>();
    private final List<VideogameCommunity> list = new ArrayList<>();

    /**
     * Check if a videogame community is followed by the user
     * @param videogameId id of the videogame
     * @return true if the user follows the community, false otherwise
     */
    boolean isVideogameFollowed (@NotNull ObjectId videogameId) {
        return (map.get(videogameId) != null);
    }

    /**
     * Get the followed videogame communities as a List.
     * Only the id and name field are set, the others are null.
     * @return the list of followed communities
     */
    List<VideogameCommunity> getList() {
        return list;
    }
```

Liked posts cache

The cache for the list of the liked posts is handled by the class LikedPostsCache, which creates a **HashSet** with the ids of the liked posts by the user in a given videogame community. The HashSet allows us to efficiently check if a post is liked, by its id (with complexity `O(1)`).

```java
public class LikedPostsCache {
    // Logger
    private static final Logger LOGGER = LogManager.getLogger(LikedPostsCache.class);

    // Set containing the posts liked by the user
    private final Set<ObjectId> set;

    /**
     * Instantiate the local copy.
     * @param username user's username
     * @param videogame id of the videogame community
     */
    public LikedPostsCache (String username, ObjectId videogame) {
        set = PostServiceFactory
                .getImplementation()
                .getLikedPostsOfVideogameCommunity(username, videogame);

        LOGGER.info("LikedPostsCache instantiated for user " + username + ", videogame " + videogame);
    }
```

The class manages to consistently update both the remote databases and the local copy.

```java
public boolean likePost (@NotNull ObjectId postId, @NotNull String user, boolean like) {

    LOGGER.info("likePost() | postId " + postId + ", user " + user + ", like " + like);
    // Update remote database
    boolean ret = PostServiceFactory
            .getImplementation()
            .likePost(postId, user, like);

    // If successful, update the local copy
    if (ret) {
        if (like) {
            set.add(postId);
        }
        else {
            set.remove(postId);
        }
    }

    return ret;
}
```

Liked comments cache

It has the same behavior for the liked posts, but it's handled by the LikedCommentsCache class.

```java
public class LikedCommentsCache {
    // Logger
    private static final Logger LOGGER = LogManager.getLogger(LikedCommentsCache.class);

    // Set containing the posts liked by the user
    private final Set<ObjectId> set;

    /**
     * Instantiate the local copy.
     * @param username user's username
     * @param post id of the post
     */
    public LikedCommentsCache(String username, ObjectId post) {
        set = CommentServiceFactory
                .getImplementation()
                .getLikedCommentsOfPost(username, post);

        LOGGER.info("LikedCommentsCache instantiated for user " + username + ", post " + post);
    }
```

Logging

In order to monitor the application's behavior, we introduced the **Log4j** framework in our code. This allows us to print meaningful logs both on console and on file in a standardized and flexible way. Moreover, the framework allows to declare the level of severity of the log.

Declare an instance of the Log4j logger for a class

```
// Logger
private static final Logger LOGGER = LogManager.getLogger(LikedCommentsCache.class);
```

Print a log (severity INFO)

```
LOGGER.info("LikedCommentsCache instantiated for user " + username + ", post " + post);
```

Implementation of MongoDB aggregations

Post collection: best users by number of posts in a given videogame community

```java
@Override
public List<ResultBestUserByPostAggregation> bestUsersByNumberOfPosts (
        @NotNull MongoConnection connection, @NotNull ObjectId videogameCommunityId, int limit
){
    // Get post collection
    MongoCollection<Document> posts = connection.getCollection(GameflowsCollection.post);

    // STAGE 1: match by videogame id
    Bson matchStage = match(eq( fieldName: "community.community_id", videogameCommunityId));
    // STAGE 2: group by author
    Bson groupStage = group( id: "$author", sum( fieldName: "num_post",  expression: 1));
    // STAGE 3: sort by num_post
    Bson sortStage = sort(descending( ...fieldNames: "num_post"));
    // STAGE 4: limit
    Bson limitStage = limit(limit);

    try {
        List<Document> postList =
            posts.aggregate(Arrays.asList(matchStage, groupStage, sortStage, limitStage)) AggregateIterable<Document>
                .into(new ArrayList<>());

        List<ResultBestUserByPostAggregation> resultList = new ArrayList<>();
        for (Document doc : postList) {
            resultList.add(
                new ResultBestUserByPostAggregation(
                    doc.getString( key: "_id"),
                    doc.getInteger( key: "num_post")
                )
            );
        }

        LOGGER.info("bestUsersByNumberOfPosts() | " +
            "Aggregation returned " + resultList.size() + " documents");

        return resultList;

    } catch (MongoException ex) {
        LOGGER.error("bestUsersByNumberOfPosts() | " +
            "Unable to execute the aggregation due to error: " + ex);
        return null;
    }
}
```

Post collection: best videogame community by number of likes in a given time period

```
public List<ResultBestVideogameCommunityAggregation> bestVideogameCommunities(
        @NotNull MongoConnection connection, @NotNull Date fromDate,
        @NotNull Date toDate, int limit)
{
    // Get post collection
    MongoCollection<Document> posts = connection.getCollection(GameflowsCollection.post);

    // STAGE 1: match timestamp
    Bson matchStage = match(
            and(
                    gte( fieldName: "timestamp", fromDate),
                    lte( fieldName: "timestamp", toDate)
            )
    );

    // STAGE 2: group by videogame community
    Bson groupStage = new Document("$group",
            new Document("_id",
                    new Document("id", "$community.community_id")
                            .append("name", "$community.community_name")
            )
            .append("sum_likes", new Document("$sum", "$likes")));

    // STAGE 3: sort by likes
    Bson sortStage = sort(descending( ...fieldNames: "sum_likes"));

    // STAGE 4: limit
    Bson limitStage = limit(limit);

    try {
        // Execute aggregation
        List<Document> postList =
                posts.aggregate(Arrays.asList(matchStage, groupStage, sortStage, limitStage)) AggregateIterable<Document>
                        .into(new ArrayList<>());

        // Fetch results
        List<ResultBestVideogameCommunityAggregation> resultList = new ArrayList<>();
        for (Document doc : postList) {
            resultList.add(
                    new ResultBestVideogameCommunityAggregation(
                            doc.getEmbedded(List.of("_id", "id"), ObjectId.class),
                            doc.getEmbedded(List.of("_id", "name"), String.class),
                            doc.getInteger( key: "sum_likes")
                    )
            );
        }

        LOGGER.info("bestVideogameCommunities() | " +
                "Aggregation returned " + resultList.size() + " documents");

        return resultList;

    } catch (MongoException ex) {
        LOGGER.error("bestVideogameCommunities() | " +
                "Unable to execute the aggregation due to error: " + ex);
        return null;
    }

}
```

Comment collection: average number of comments per post

```
public List<ResultAverageCommentPerPost> averageNumberOfCommentsPerPost(
        MongoConnection connection, int skip, int limit
){
    if(connection == null){
        LOGGER.fatal("averageNumberOfCommentsPerPost() | " +
                "MongoDB connection cannot be null!");
        throw new IllegalArgumentException("MongoDB connection cannot be null!");
    }

    // Comment collection
    MongoCollection<Document> comments =
            connection.getCollection(GameflowsCollection.comment);

    //stage 1: group (sum/count of post in a community)
    Bson group_sum = new Document("$group",
            new Document("_id",
                new Document("post_id","$post.post_id")
                    .append("community_id","$post.community_id") Document
                    .append("community_name", "$post.community_name"))
                    .append("comments_per_post", new Document("$count", new Document()))
    );

    //stage 2: group (avg comments per post of a community)
    Bson group_avg = new Document("$group",
            new Document("_id", new Document("community_id","$_id.community_id")
                    .append("community_name", "$_id.community_name"))
                    .append("avg_comments_per_post", new Document("$avg","$comments_per_post")));

    //stage 3: sort (desc by avg)
    Bson sort_avg = sort(descending( ...fieldNames: "avg_comments_per_post"));

    try {
        // Aggregation:  average number of comments per post in a community (disc sorted)
        List<Document> documentList =
                comments.aggregate(
                        Arrays.asList(
                                group_sum,
                                group_avg,
                                sort_avg,
                                skip(skip),
                                limit(limit)
                        )
                ).into(new ArrayList<>());

        // Convert results and return
        List<ResultAverageCommentPerPost> resultList = new ArrayList<>();

        for (Document doc : documentList) {
            resultList.add(
                new ResultAverageCommentPerPost(
                    doc.getEmbedded(List.of("_id", "community_id"), ObjectId.class),
                    doc.getEmbedded(List.of("_id", "community_name"), String.class),
                    doc.getDouble( key: "avg_comments_per_post")
                )
            );
        }

        return resultList;

    }catch (Exception e){
        LOGGER.error("averageNumberOfCommentsPerPost() | " +
                "Error while retrieving comments, message: " + e);
        return null;
    }
}
```

Comment collection: average number of comments per post of an user

```
public List<ResultAverageCommentPerUser> averageNumberOfCommentsPerUser (
                    MongoConnection connection, int skip, int limit
) {
    if(connection == null){
        LOGGER.fatal("averageNumberOfCommentsPerUser() | MongoDB connection cannot be null!");
        throw new IllegalArgumentException("MongoDB connection cannot be null!");
    }

    // Get comments collection
    MongoCollection<Document> comments = connection.getCollection(GameflowsCollection.comment);

    // Stage 1: group (count of post in a community)
    Bson groupSum = new Document("$group",
            new Document("_id", new Document("post_id","$post.post_id")
                    .append("author","$post.author"))
                    .append("comments_per_author", new Document("$count", new Document())));

    // Stage 2: group (avg comments per post of a community)
    Bson groupAvg = new Document("$group",
            new Document("_id", new Document("author","$_id.author"))
                    .append("avg_comments_per_author", new Document("$avg","$comments_per_author")));

    // Stage 3: sort (desc by avg)
    Bson groupSort = sort(descending( ...fieldNames: "avg_comments_per_author"));

    try {
        // Aggregation: AVG number of comments per post for each author (disc sorted)
        List<Document> documentList =
                comments.aggregate(
                        Arrays.asList(
                                groupSum,
                                groupAvg,
                                groupSort,
                                skip(skip),
                                limit(limit)
                        )
                ).into(new ArrayList<>());

        // Convert MongoDB result to Java objects
        List<ResultAverageCommentPerUser> resultList = new ArrayList<>();

        for (Document doc : documentList) {
            resultList.add(
                new ResultAverageCommentPerUser(
                    doc.getEmbedded(List.of("_id", "author"), String.class),
                    doc.getDouble( key: "avg_comments_per_author")
                )
            );
        }

        return resultList;


    } catch (Exception e){
        LOGGER.error("avgCommentsPerAuthorPosts() | Connection to MongoDB failed: " + e);
        return null;
    }
}
```

Implementation of Neo4j on-graph queries

Suggest new videogame communities to a user (VideogameCommunityServiceImpl)

```
@Override
public List<VideogameCommunity> viewSuggestedVideogameCommunities (Neo4jConnection connection,
                                                                   User user, int limit)
{
    // Check arguments
    if (connection == null) {
        LOGGER.fatal("viewSuggestedVideogameCommunities() | Neo4jConnection parameter cannot be null");
        throw new IllegalArgumentException("Neo4jConnection cannot be null");
    }

    ArrayList<VideogameCommunity> list = new ArrayList<>();

    // Get a new neo4j session
    try (Session session = connection.getSession()) {
        session.readTransaction(
                tx -> {
                    // Read from neo4j db
                    Result result = neo4jSuggestedVideogames (tx, user.getUsername(), limit);

                    // Iterate the obtained data and build a list of videogames to return
                    while (result.hasNext()) {
                        Record record = result.next();
                        list.add(
                            new VideogameCommunity(
                                new ObjectId(record.get("VIDEOGAME").get("videogame_id").asString()),
                                record.get("VIDEOGAME").get("name").asString()
                            )
                        );
                    }

                    return 1;
                }
        );

        LOGGER.info("viewSuggestedVideogameCommunities() | " +
                "Returned " + list.size() + " videogame communities");

        return list;

    } catch (Neo4jException ex) {
        LOGGER.error("viewSuggestedVideogameCommunities() | " +
                "Neo4j transaction failed due to errors: " + ex);
        return null;
    }
}
```

```
    private static Result neo4jSuggestedVideogames (@NotNull Transaction tx,
                                                    @NotNull String username,
                                                    int limit)
    {
        return tx.run( s: "MATCH (target_user: User)-[tg:FOLLOWS]->(his_videogame:Videogame) " +
                        "WHERE target_user.username = $username " +
                        "WITH COLLECT(his_videogame.name) AS game_not_to_suggest " +
                        "MATCH (his_videogame:Videogame)<-[ou:FOLLOWS]-(other_user:User)-[og:FOLLOWS]->" +
                        "(other_videogame:Videogame) " +
                        "WHERE (NOT other_videogame.name IN game_not_to_suggest) " +
                        "RETURN DISTINCT other_videogame AS VIDEOGAME, count(og) AS number_of_following " +
                        "ORDER BY number_of_following DESC " +
                        "LIMIT $limit",
                parameters( ...keysAndValues: "username", username, "limit", limit));
    }
```

Identify influencer users (AdminServiceImpl)

```
@Override
public List<String> viewInfluencerRanking(@NotNull Neo4jConnection connection,
                                          @NotNull Date fromDate, @NotNull Date toDate, int limit)
{
    // Open a neo4j session
    try (Session session = connection.getSession()) {
        List<String> list = new ArrayList<>();

        // Execute the read transaction
        session.readTransaction(
                tx -> {
                    Result result = neo4jInfluencerQuery(tx, fromDate, toDate, limit);

                    // Parse the results
                    while (result.hasNext()) {
                        Record record = result.next();
                        list.add(record.get("INFLUENCER").asString());
                    }
                    return 1;
                }
        );

        LOGGER.info("viewInfluencerRanking() | Successfully computed new list of " +
                list.size() + " influencers");

        return list;

    } catch (Neo4jException ex) {
        LOGGER.error("viewInfluencerRanking() | Neo4j query failed: " + ex);
        return null;
    }
}
```

```java
private static Result neo4jInfluencerQuery (@NotNull Transaction tx,
                                            @NotNull Date fromDate,
                                            @NotNull Date toDate,
                                            int limit)
{
    LocalDate localFromDate = fromDate.toInstant() Instant
            .atZone(ZoneId.systemDefault()) ZonedDateTime
            .toLocalDate();

    LocalDate localToDate = toDate.toInstant() Instant
            .atZone(ZoneId.systemDefault()) ZonedDateTime
            .toLocalDate();

    return tx.run( s: "MATCH (influencer: User)-[write:WRITES]->(posts:Post)\n" +
            "MATCH (user_who_liked: User)-[liked:LIKES]->(posts:Post)\n" +
            "WHERE date(liked.timestamp) >= $from AND date(liked.timestamp) <= $to\n" +
            "OPTIONAL MATCH (all_comments: Comment)-[commented:REPLY]->(posts:Post)\n" +
            "WHERE date(commented.timestamp) >= $from " +
            "AND date(commented.timestamp) <= $to\n" +
            "RETURN DISTINCT \n" +
            "        influencer.username AS INFLUENCER, \n" +
            "        (count(DISTINCT user_who_liked) + count(DISTINCT all_comments) ) AS GRADE\n" +
            "ORDER BY GRADE DESC\n" +
            "LIMIT $limit",
            parameters( ...keysAndValues: "limit", limit, "from", localFromDate, "to", localToDate));
}
```

Tests

In order to test the different modules of our application, we used the Junit and Mockito frameworks.

it.unipi.dii.inginf.lsdb.gameflows.admin

- **AdminMockup**: generate an Admin object with "default" values
- **AdminServiceImplTest:** test AdminServiceImpl class

`TEST_createAdminAccount`	Test the creation of an admin account in both MongoDB and Neo4j
`TEST_login`	Test login by using arbitrary username and password (already existing in database)
`GIVEN_account_created_WHEN_login_with_correct_password_THEN_login_return_true`	Create an admin account and test if login works
`GIVEN_account_created_WHEN_login_with_wrong_password_THEN_login_return_false`	Create an admin account and test if login fails when using the wrong password
`GIVEN_account_doesnt_exist_WHEN_login_THEN_login_return_false`	Test if login fails when try to login with an account that doesn't exist
`TEST_block_user`	Test blockUser on an existing user account
`TEST_viewInfluencers`	Test if viewInfluencers works
`TEST_updateInfluencers`	Test if updateInfluencers works by using a list of already existing users in the database

it.unipi.dii.inginf.lsdb.gameflows.comment

- **CommentMockup**: generate a Comment object with "default" values
- **InfoPostMockup:** generate a InfoPost object with "default" values
- **CommentServiceImplTest**: test CommentServiceImpl class

`TEST_addComment`	Add a mockup comment to the databases
`TEST_find`	Insert a mockup comment in the databases and find it by ObjectId
`TEST_browse`	Test the browsing of an existing post's comments
`TEST_likeComment`	Insert a mockup comment in the databases and test if likeComment works
`GIVEN_comment_already_liked_THEN_like_fails`	Test if "like comment" fails when the comment is already liked by the user
`TEST_dislikeComment`	Insert a mockup comment in the databases, execute the "like comment" and test if "dislike comment" (likeComment with field *like* false) works
`TEST_deleteCommentById`	Insert a mockup comment and test if its deletion works
`TEST_deleteCommentsByPostId_MongoDB`	Test if the deletion of comments of an existing post works
`TEST_deleteCommentsByCommunityId`	Test if the deletion of comments of an exiting videogame community works

TEST_averageNumberOfCommentsPerPost	Test if aggregation “average number of comments per post” works
TEST_averageNumberOfCommentsPerUser	Test if aggregation “average number of comments per user” works
TEST_getLikedCommentsOfPost	Test the methods that returns the ids of the liked comments in an existing post by an existing user

it.unipi.dii.inginf.lsdb.gameflows.persistence

- **DatabaseConfigurationMockup**: create a DatabaseConfiguration object with “default values”
- **DatabaseConfigurationImplTest**: test DatabaseConfigurationImpl

TEST	Test if toString method works. If toString() works, it means the class was able to successfully read the configuration options from file

- **MongoConnectionImplTest**: test MongoConnectionImpl

TEST_getCollection	Test that getCollection method doesnàt return null for the different collections
TEST_verifyConnectivity	Test verifyConnectivity, i.e. if application can reach the database when it’s up

- **Neo4jConnectionImplTest**: test Neo4jConnectionImpl

WHEN_Neo4jConnection_is_created_THEN_application_can_connect_to_neo4j	Test if application can connect to neo4j server
WHEN_getSession_called_THEN_session_is_open	Test if neo4j session is open when call getSession

- **PersistenceFactoryTest**: test PersitenceFactory

WHEN_getDatabaseConfiguration_invoked_twice_THEN_same_instance_returned	Test if DatabaseConfiguration is a singleton instance

it.unipi.dii.inginf.lsdb.gameflows.post

- **InfoVideogameCommunityMockup**: create an InfoVideogameCommunity object with “default” values
- **PostMockup**: create a Post object with “default” values
- **PostServiceImplTest**: test PostServiceImpl

TEST_insertPost	Insert a mockup post in the databases
TEST_browsePostByLike	Test the browsing by like of an existing videogame community’s posts
TEST_browsePostByDate	Test the browsing by like of an existing videogame community’s posts
TEST_browsePostInfluencer	Test the browsing of an existing videogame community’s posts, written by influencer users

TEST_find	Insert a new post and test if the user can find it by ObjectId
TEST_likePost	Insert a new post and test if the user can like it
TEST_dislikePost	Insert a new post, like it and test if the user can remove the like
GIVEN_post_is_liked_THEN_like_post_fails	Insert a new post, like it and test if a second “like post” operation fails
GIVEN_post_is_not_liked_THEN_dislike_post_fails	Insert a new post and test if a “dislike post” operation fails when there is no LIKES relationship between user and post
TEST_getLikedPostsOfVideogameCommunity	Test if user can successfully retrieve the ids of the post they liked, inside an existing videogame community
GIVEN_post_exists_THEN_deletePostById_returns_true	Insert a new post and test if the application can successfully delete it
GIVEN_post_not_exist_THEN_deletePostById_returns_false	Test if the deletion of a post fails when the post doesn’t exist
TEST_deletePostByVideogameCommunity_MongoDB	Test if the application can successfully delete all the posts of a given community inside the MongoDB database
TEST_bestUsersByNumberOfPosts	Test if the “get best users by number of written posts” aggregation works
TEST_bestVideogameCommunities	Test if “get best videogame communities by number of likes” aggregation works

it.unipi.dii.inginf.lsdb.gameflows.user

- **UserMockup**: create a User object with “default” values
- **UserServiceImplTest**: test UserServiceImpl

TEST_login	Test if login works for an already existing user
TEST_insertUser	Test if the application can successfully insert a new user
WHEN_insert_same_user_twice_THEN_return_false	Test if the method insertUser fails when trying to insert the same user twice (the username must be unique!)
GIVEN_user_is_blocked_THEN_isUserBlocked_returns_true	Insert a new user (while mocking the connection to neo4j), block him/her and test if isUserBlocked return true
TEST_browse	Test if the application can browse the users
TEST_find	Test if the application can find a user by username

it.unipi.dii.inginf.lsdb.gameflows.util

- **PasswordTest**: test Password

WHEN_using_the_correct_password_THEN_checkPassword_is_true	Test if checkPassword returns true when the tested password is correct
WHEN_using_different_passwords_THEN_checkPassword_is_false	Test if checkPassword returns false when the tested password is wrong
WHEN_create_two_instances_from_same_password_THEN_salts_are_not_equals	Test that the salts are not equals for two different Password instances, even if the cleartext password is the same

WHEN_create_two_instances_from_same_password_THEN_hash_are_not_equals	Test that the hash values are not equals for two different Password instances, even if the cleartext password is the same

it.unipi.dii.inginf.lsdb.gameflows.videogame

- **VideogameCommunityMockup:** create a VideogameCommunity object with "default" values
- **VideogameCommunityServiceImplTest**: test VideogameCommunityService

WHEN_MongoConnection_is_null_THEN_insertVideogameCommunity_throws_IllegalArgumentException	Check if the method insertVideogameCommunity throws an IllegalArgumentException if the MongoConnection parameter is null
WHEN_Neo4jConnection_is_null_THEN_insertVideogameCommunity_throws_IllegalArgumentException	Check if the method insertVideogameCommunity throws an IllegalArgumentException if the Neo4jConnection parameter is null
WHEN_MongoConnection_is_null_THEN_deleteVideogameCommunity_throws_IllegalArgumentException	Check if the method deleteVideogameCommunity throws an IllegalArgumentException if the MongoConnection parameter is null
WHEN_Neo4jConnection_is_null_THEN_deleteVideogameCommunity_throws_IllegalArgumentException	Check if the method deleteVideogameCommunity throws an IllegalArgumentException if the Neo4jConnection parameter is null
WHEN_MongoConnection_is_null_THEN_view_throws_IllegalArgumentException	Check if the method view throws an IllegalArgumentException if the MongoConnection parameter is null
WHEN_MongoConnection_is_null_THEN_search_throws_IllegalArgumentException	Check if the method search throws an IllegalArgumentException if the MongoConnection parameter is null
TEST_search	Test if the search of a videogame community works
TEST_browse	Test if the browsing of videogame communities works
TEST_find	Test if the application can find a videogame community by its ObjectId
TEST_insertVideogameCommunity	Test the insertion of a videogame community
TEST_deleteVideogameCommunity	Insert a videogame community and test its deletion is successful
WHEN_delete_not_existing_videogame_THEN_deleteVideogameCommunity_returns_false	Test if the deletion of a videogame community fails when the videogame community doesn't exist
TEST_followVideogameCommunity	Test if a user can successfully follow and unfollow a videogame community
TEST_viewFollowedVideogames	Test the viewing of the followed videogame communities
TEST_viewSuggestedVideogameCommunities	Test the viewing of the suggested videogame communities