UNIVERSITÀ DI PISA

Master of Science in Artificial Intelligence and Data Engineering
Large-Scale and Multi-Structured Databases

MappIt

Project Documentation

Team Members:
Pacini Giacomo
Nello Enrico
Pierucci Matteo

Academic Year 2021-2022

Contents

1 Introduction

MappIt is an application that allows **users** to discover new **places** to visit and share their adventures and experiences, helping creators to promote their contents on the community.

1.1 Description

A user can **register** an account submitting **required information.**

After the registration process, the new user can browse through **posts** of other users, find new **places**, **follow** friends and **like** trip posts.

The application could retrieve points of interest **near** the user geospatial position, but it also suggests new places to visit based on user's interests. As a user could add places to its **favorites** and **visited** ones, another user can discover them through its followings.

A user can **post his own adventure**, submitting information related to the experience he has done, for example: a **description**, the visit **date**, the **category** in which the visit will be located and **link** to multimedia contents.

The application also **suggests posts and users** to follow to the logged user, based on popularity rankings. Some of the **sorting criteria** adopted are the number of posts published, the number of followers and the likes counter.

An **admin user** can access to the service providing superuser credentials. It can browse statistics and analytics as:

- the number of posts per year and category
- the most active users by published posts
- most popular posts based on likes
- most popular places by likes in posts and favorites counter

A super user can also perform **deletion** on users and **posts**.

2 Analysis

2.1 Functional requirements and use cases

2.1.1 Use case list

- An **unlogged** user can:
  - Register to the service
  - Login into the service
- A **logged** user can:
  - Logout
  - Change account password
  - Write a new post
  - Browse places
    - Find places
    - View place information
    - Add/remove one place to favorites
    - Add one place to visited ones
  - Browse posts
    - Find posts
    - View post information
    - Like/unlike one post
  - Browse users
    - Find users
    - View user information
    - Follow/unfollow a user
- An **admin** user can:
  - Browse analytics
    - Find analytics
    - View analytics
  - Delete posts
  - Delete users

2.1.2 UML use case diagram

In our application there are three main actors, which we mentioned in the previous requirements:

- **Unlogged user** can register to the service providing required information and login to the service
- **Logged user** has full access to the application functionalities. It can find other users and follow them, create new posts and like the ones belonging to other users, browse through places and adding them to favorite/visited ones.
- **Admin user** can access to logged user functionalities, moreover, can browse analytics ad delete posts or users.

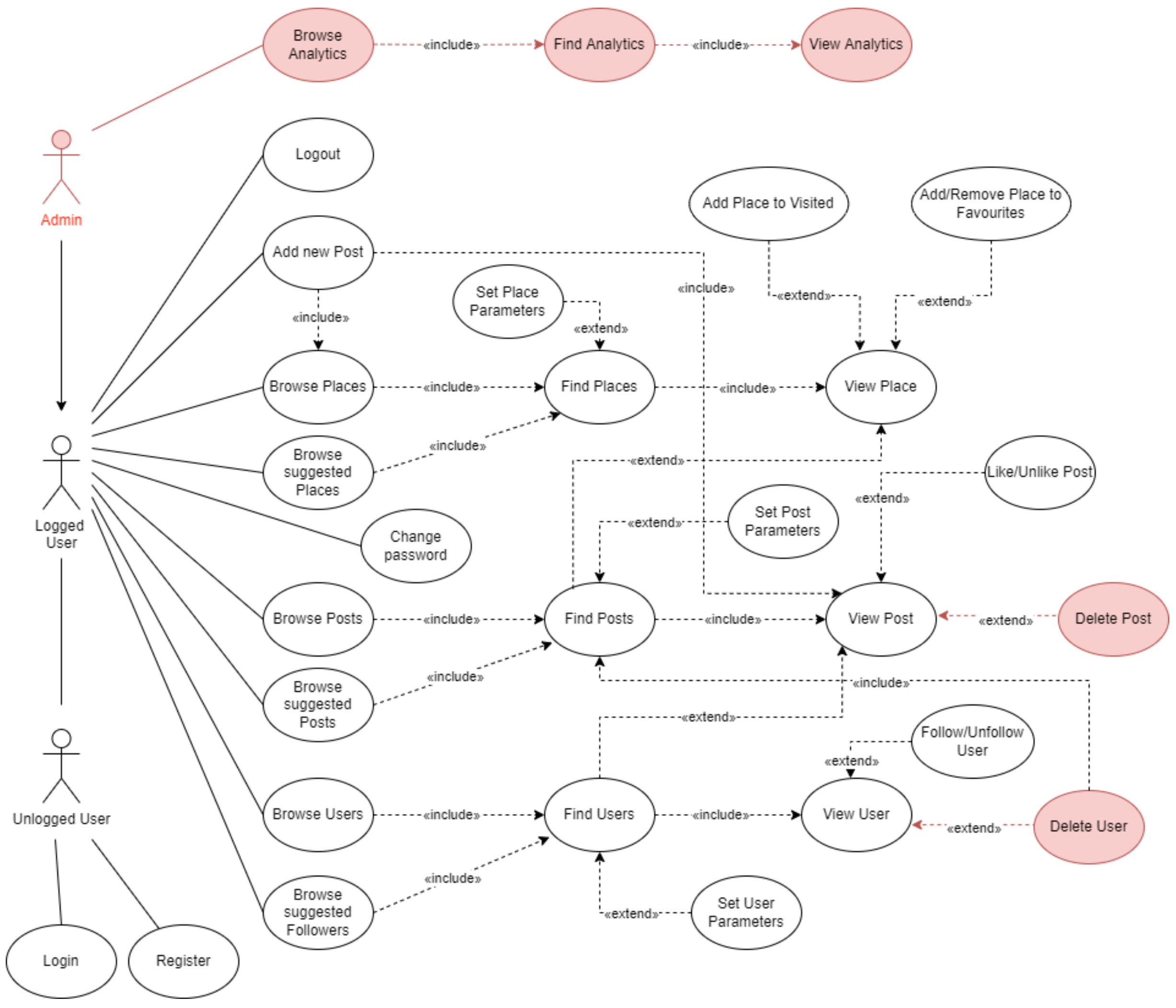


Figure 1 UML Use Case Diagram

2.2 Non-Functional Requirements

- The application should guarantee **high availability**

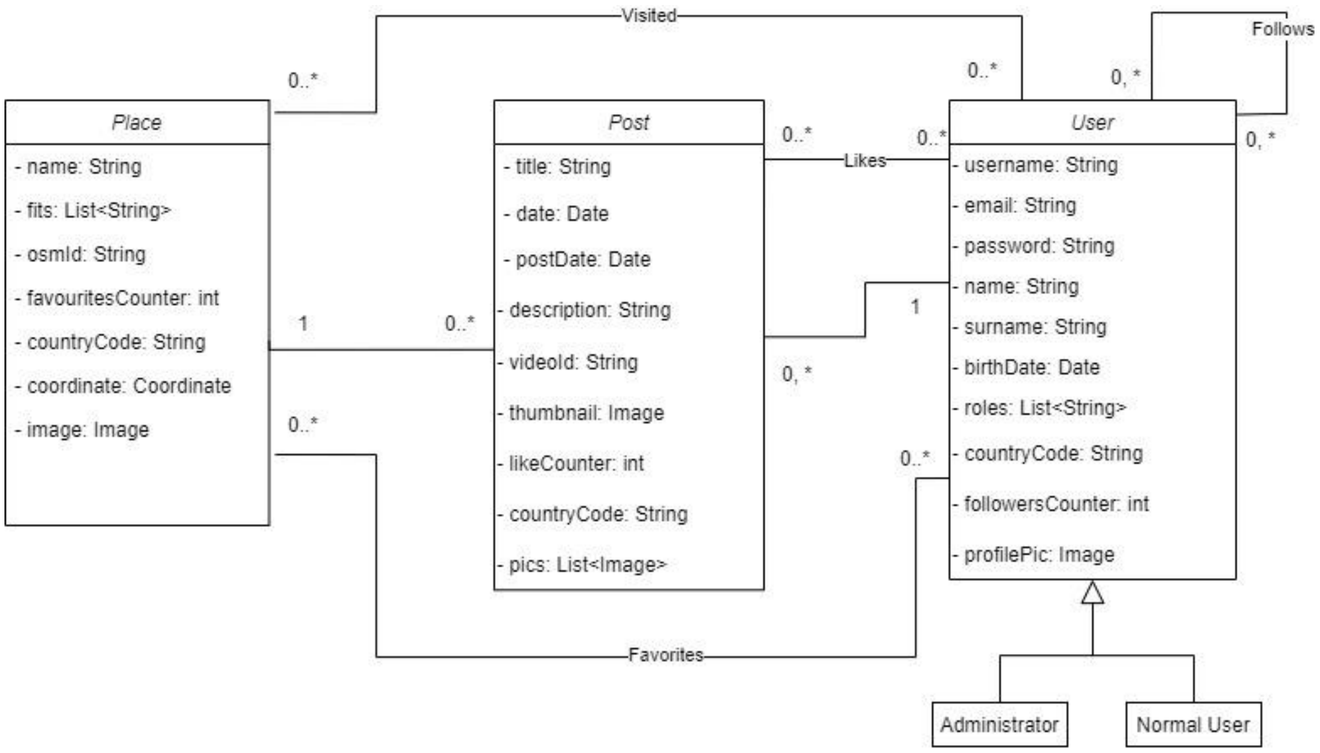
- The application must be **tolerant to data loss** and resilient to **single point of failure**
- **Response time** should be as **limited** as possible to guarantee a good user experience
- The application must be **portable**, implementing a RESTful API interface
- Sensitive information, like passwords, must be encrypted to ensure **Data Security**

To satisfy these requirements, we choose to focus on **Availability** and **Partition Tolerance** features of the CAP Triangle.

2.3 Entities

We defined the following main entities: **User**, **Place** and **Post**.

2.3.1 UML Class Diagram



1. A place could have zero or more posts published in it, while a post has only one associated place
2. A user is the author of zero or more posts, while a post has only one author
3. A user follows zero o more other users
4. A user adds zero or more places to favorites

3 Architectural Design

3.1 Software Architecture

MappIt is an application based on the *client-server* paradigm. In the next pages we discuss in detail the *server part* that we designed and implemented.

3.1.1 General Architecture

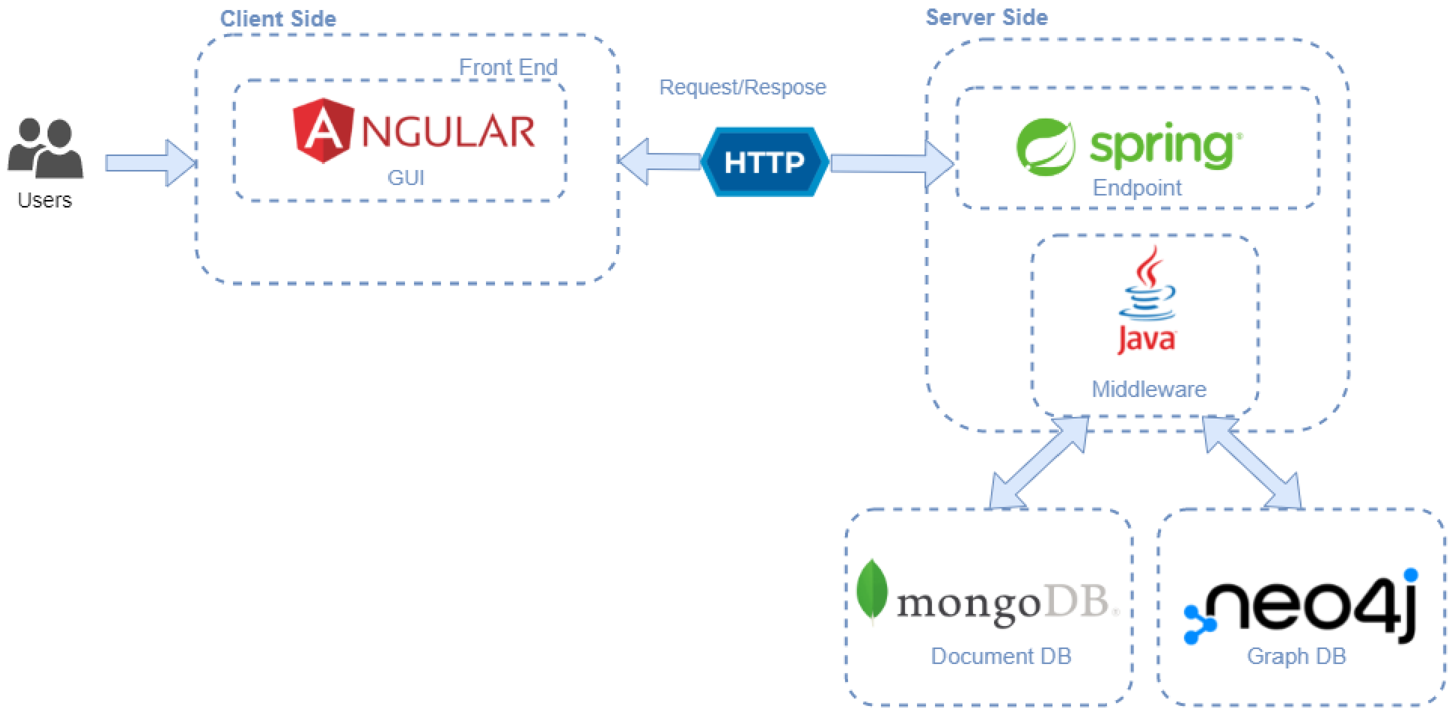


Figure 3 Software Architecture

3.2 Client Side

The Client side exploits Angular, that allows client to interact with the endpoints of the Server.

3.3 Server Side

The Server side exploits Spring, that allows to easily create endpoints. The middleware that interacts with the Databases is written in Java. In the following we explain our server endpoints.

3.3.1 Endpoints

3.3.1.1 User Endpoints

1. URL	/api/register
Method	POST
Parameters	RegistrationUser object that contains the following fields required for the registration process: username, email, password, name, surname, birthDate
Description	This method registers a new user to the service

URL	/api/login
Method	POST
Parameters	JwtRequest object that contains username and password for the authentication
Description	This method authenticates a registered user to the service

URL	/api/users/find
Method	GET
Parameters	Username of the user we want to retrieve information
Description	This method retrieves information about a specific user

URL	/api/user
Method	GET
Parameters	None
Description	This method retrieves information of the current logged user

URL	/api/user/{id}
Method	GET
Parameters	Id of the User from which we want to retrieve information
Description	This method retrieves information of a specific user, given its id

URL	/api/user/{id}/posts/liked
Method	GET
Parameters	Id of the User from which we want to retrieve liked posts, number of posts to retrieve
Description	This method retrieves posts liked by a specific user

URL	/api/user/{id}
Method	DELETE
Parameters	Id of the user we want to delete
Description	This method deletes a specific user, given its id

URL	/api/users/{id}/followers
Method	GET
Parameters	Id of the user from which we want to retrieve the followers, number of followers to retrieve
Description	This method retrieves the followers of a specific user
URL	/api/users/{id}/followed
Method	GET
Parameters	Id of the user from which we want to retrieve the followings, number of followings to retrieve
Description	This method retrieves the followings of a specific user

URL	/api/users/most-active
Method	GET
Parameters	Activity and number of users to return
Description	This method returns the most active users, filtering by activity

URL	/api/user/posts/suggested
Method	GET
Parameters	The number of suggested posts to retrieve
Description	This method retrieves the suggested posts for the logged in user

URL	/api/user/posts/published

Method	GET
Parameters	Id of the user from which we want to retrieve published posts
Description	This method retrieves the published posts of a specific user

URL	/api/user/places/suggested
Method	GET
Parameters	None
Description	This method retrieves a list of suggested places for the current logged in user

URL	/api/user/places/favourites
Method	GET
Parameters	userId [Optional] of the user from which we want to retrieve favourite places
Description	This method retrieves a list of favourite places for a specific user, or the current logged in user

URL	/api/user/places/visited
Method	GET
Parameters	userId [Optional] of the user from which we want to retrieve visited places
Description	This method retrieves a list of visited places for a specific user, or the current logged in user

URL	/api/users/followers/suggested
Method	GET
Parameters	None
Description	This method retrieves users suggested for the logged in user

URL	/api/user/places/{placeId}/visit
Method	POST
Parameters	placeId of the place to add to visited
Description	This method adds the specified place to the visited places of the current logged in user

URL	/api/user/places/{placeId}/favourites
Method	POST
Parameters	placeId of the place to add to favourites
Description	This method adds the specified place to the favourites places of the current logged in user

URL	/api/user/places/{placeId}/favourites
Method	DELETE
Parameters	placeId of the place to remove from favourites
Description	This method removes the specified place from the favourites places of the current logged in user

URL	/api/user/follower/{id}
Method	POST
Parameters	Id of the user to follow

Description	This method adds the specified user to the followings of logged in user

URL	**/api/user/follower/{id}**
Method	DELETE
Parameters	Id of the user to unfollow
Description	This method removes the specified user from the followings of logged in user

URL	**/api/user/password**
Method	PUT
Parameters	New password to store and authorization token
Description	This method updates the password of the current logged user

3.3.1.2 Post Endpoints

URL	**/api/post**
Method	POST
Parameters	Post object with necessary information to public a post
Description	This method creates a post for the logged in user

URL	**/api/post/{id}**
Method	GET
Parameters	Id of the post from which we want to retrieve information
Description	This method retrieves the information of a specific post

URL	**/api/posts/most-popular**
Method	GET
Parameters	Time interval, activity filter and number of posts to retrieve
Description	This method retrieves most popular posts

URL	**/api/posts/per-year**
Method	GET
Parameters	Number of results to show
Description	This method retrieves the number of posts grouped by year and activity

URL	**/api/post/{id}**
Method	DELETE
Parameters	Id of the post we want to delete
Description	This method deletes a specific post

URL	**/api/posts/author/{author-id}**
Method	DELETE
Parameters	Id of the author associated to the posts we want to delete
Description	This method deletes all posts of a specific user

3.3.1.3 Place Endpoints

URL	**/api/place/{placeId}**
Method	GET
Parameters	Id of the place from which we want to retrieve information
Description	This method retrieves information of a specific place, given the id

URL	**/api/place/nearby**
Method	GET
Parameters	Latitude, longitude, radius [Optional], activityFilter [Optional available values: {“distance”, “popularity”}] and orderBy [Optional] method
Description	This method retrieves places near a given point, specifying a radius distance and an activity filter

URL	**/api/places/most-popular**
Method	GET
Parameters	activityFilter [Optional] to filter places, number of places to retrieve
Description	This method retrieves most popular places, it can also specify an activityFilter for the query

3.3.1.4 Activity Endpoint

URL	**/api/activities/all**
Method	GET
Parameters	None
Description	This method retrieves information about all the available activities

3.3.2 Roles
There are two possible roles inside the application: **standard users** and **admins**.

3.3.2.1 Users
The common users can perform all the queries listed above but the following:

- Deleting a User
- Deleting a Post
- Perform the analytic query that returns the most active users, given a specific Activity type (see Ch. 8.3.1 for details)
- Perform the analytic query that returns the number of posts per year and filtered by a specific activity type (see Ch. 8.3.1 for details)

3.3.2.2 Admin
Admins have some additional rights with respect of common users.
They are supposed to be the ones who oversee posts written by the users and who can gather useful information about the people’s trends through analytics.

An administrator could decide whether to delete a post or an account in case he spots someone who violates the applications’ terms/agreement.
The endpoints that require an **admin role** to be granted are:

- /user/{id} [DELETE]
- /post/{id} [DELETE]
- /users/most-active [GET]
- /posts/per-year [GET]

4 Design Choices and Implementation

4.1 MongoDB Design

This document database contains three collections, that store information about users, places, and posts.

4.1.1 User Collection

Information about every user is stored inside a document of the ***user*** collection. In the user document we use an array of post IDs to maintain the relationship between entities and we choose to add redundant details about posts inside each user document. In this way, when a list of published or liked post must be displayed, we could retrieve the main details about them with a **single access** to the database. Once a post is clicked, a query to the ***post*** collection retrieves all the remaining information.

Field	Description
_id	Corresponds to the ID generated by MongoDB
username	Contains the username of the user
password	Contains the password of the user
name	Contains the name of the user
surname	Contains the surname of the user
email	Contains the email address of the user
countryCode	Contains the country code of the user (e.g., IT)
birthDate	Optional field that contains the birth date of the user
role	Contains the privilege level of the user. Admin or User are the only two possible values
YTVideoId	Contains the ID of the YouTube associated video if any
FlickrAccountId	Contains the ID of the Flickr associated account if any
profilePic	Contains the link to the avatar of the user
followers	Redundant field that contains the total number of followers of the user
posts	Holds an array containing embedded documents of posts written by the user

4.1.2 Place Collection

A document inside ***place*** collection contains all information about that place.
We have an array of post IDs, and we choose to introduce a redundancy adding details about post. In this way, when a list of post of a certain place is shown, we could retrieve the main details about that post with a query that involves only one collection. Once a post is clicked, a query to the Post Collection retrieves all the remaining information.

Field	Description
_id	Corresponds to the ID generated by MongoDB
name	This one corresponds to the one that is present in the GeoJSON document, but it has a its own field just in case the OSM name will not be used

loc {
 - type
 - coordinates
 }
	This field contains the GeoJSON of the point that represents the place itself. Inside we have the actual coordinates of the representative point. This field is essential to perform searches by distance

fits	This is an attribute that indicates the suggested activities that could be done in that place. Drone, skiing, surfing, trekking, etc. are some examples of the possible values.
countryCode	Contains the country code of the place (e.g., IT)
image	Contains a link to the image chosen to represent the place, namely the thumbnail.
osmID	Contains the _id of the external document inside the placeOsmData collection, which is a document containing all the geojson feature obtained for that place by OpenStreetMap
posts	Holds an array containing embedded documents of posts which are in that place
favs	Redundant field that contains the total number of favorites from the users
totalLikes	Redundant field that contains the total number of likes from the posts that refer to this place

4.1.3 Post Collection

A document inside ***post*** collection contains all information about a post.
The query to retrieve all the like relationships of each post node could be **computational heavy**, so we decided to add the number of likes per post as a redundant attribute.

Field	Description
_id	Corresponds to the ID generated by MongoDB
author	Contains the _id of the user author of the post

authorUsername	Redundant value that contains the username of the author
title	Contains the title of the experience
place	Contains the _id of the place of the post
placeName	Redundant value that contains the name of the place the post refers to
countryCode	Contains the country code of the place on which the post refers to (e.g., IT)
date	Optional field that contains the date of the experience
postDate	Contains the date of the post
desc	Short description of the post
activity	Contains the _id of the kind of the activity related to this post
tags	Optional field that contains an array of Strings useful for categorization and indexing porpouses
YT_videoId	If the post embeds a video from YT, this field contains the video id of that video
FlickrPostId	If the post embeds a post from Flickr, this field contains the video id of that post
pics	If the post embeds an image from Flickr, this field contains the link to that image
likes	An aggregate redundancy value, that let us to fast obtain the total count of like of a post, avoiding everytime to retrieve it from GraphDB

4.1.4 Activity Collection

A document inside ***activity*** collection contains all information about a possible activity to specify inside a post.

Field	Description
_id	Corresponds to the ID generated by MongoDB
category	Contains the main topic for a post, e.g., *Sport, Photography,* etc.
activity	Contains the specific (one or more) sub-topic for a post, e.g., inside *Sport* category, *Surfing*

tags	Contains an array of keywords related to the activity

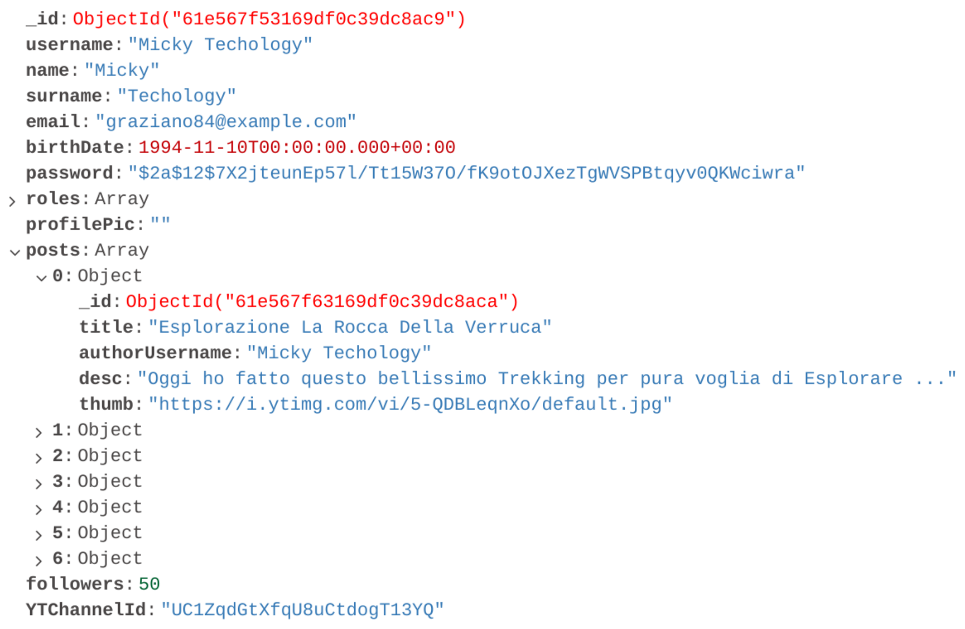
4.1.5 Document Embedding

We decided to exploit a data model that uses **embedded documents** to describe our one-to-many relationships between connected data.
We introduced the **posts** in both ***user*** collection and ***place*** collection, which is a document which contains partial information respectively about posts that are written by the user of the parent document and posts that refers to the place of the parent document.

Embedding connected data in a single document can **reduce the number of read operations** required to obtain data and our aim is to structure so that our application receives all the required information in a single read operation.

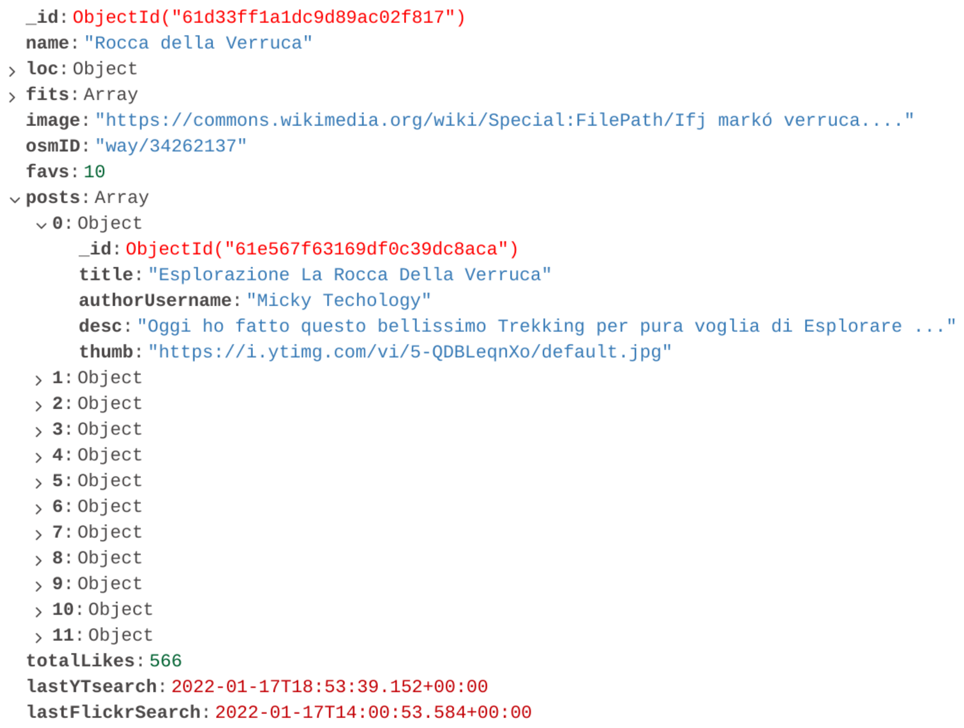
In both cases, the field **posts** will hold the following documents:

User Collection



```
  _id: ObjectId("61e567f53169df0c39dc8ac9")
  username: "Micky Techology"
  name: "Micky"
  surname: "Techology"
  email: "graziano84@example.com"
  birthDate: 1994-11-10T00:00:00.000+00:00
  password: "$2a$12$7X2jteunEp57l/Tt15W37O/fK9otOJXezTgWVSPBtqyv0QKWciwra"
> roles: Array
  profilePic: ""
v posts: Array
  v 0: Object
      _id: ObjectId("61e567f63169df0c39dc8aca")
      title: "Esplorazione La Rocca Della Verruca"
      authorUsername: "Micky Techology"
      desc: "Oggi ho fatto questo bellissimo Trekking per pura voglia di Esplorare ..."
      thumb: "https://i.ytimg.com/vi/5-QDBLeqnXo/default.jpg"
  > 1: Object
  > 2: Object
  > 3: Object
  > 4: Object
  > 5: Object
  > 6: Object
  followers: 50
  YTChannelId: "UC1ZqdGtXfqU8uCtdogT13YQ"
```

Place Collection



```
  _id: ObjectId("61d33ff1a1dc9d89ac02f817")
  name: "Rocca della Verruca"
> loc: Object
> fits: Array
  image: "https://commons.wikimedia.org/wiki/Special:FilePath/Ifj markó verruca...."
  osmID: "way/34262137"
  favs: 10
v posts: Array
  v 0: Object
       _id: ObjectId("61e567f63169df0c39dc8aca")
       title: "Esplorazione La Rocca Della Verruca"
       authorUsername: "Micky Techology"
       desc: "Oggi ho fatto questo bellissimo Trekking per pura voglia di Esplorare ..."
       thumb: "https://i.ytimg.com/vi/5-QDBLeqnXo/default.jpg"
  > 1: Object
  > 2: Object
  > 3: Object
  > 4: Object
  > 5: Object
  > 6: Object
  > 7: Object
  > 8: Object
  > 9: Object
  > 10: Object
  > 11: Object
  totalLikes: 566
  lastYTsearch: 2022-01-17T18:53:39.152+00:00
  lastFlickrSearch: 2022-01-17T14:00:53.584+00:00
```

The document embedding technique implies the problem of **document growth**; for that reason, we calculated the dimension of a single embedded document inside the array, thanks to the following query on ***Mongosh***:

```
db.user.aggregate([
  {
    "$unwind":"$posts"
  },
  {
    "$project": {
      "username": 1,
      "object_size": { $bsonSize: "$posts" }
    }
  },
  {
    "$group": {"_id":null, "avg":{"$avg":"$object_size"}}
  }
])
```

The query returned an average value of **253 bytes** for each embedded post document inside the array (the worst case corresponds to a post description of 75 characters, as we decided to truncate the description field inside the embedded posts documents).

We then calculated the average size of a user document and a place document, with the ***posts* field empty**, as stated in the following query:

```
db.user.aggregate([
    {
      "$match":
      {
        "posts.0": {$exists: false}
      }
    },
    {
      "$project": {
        "username": 1,
        "object_size": { $bsonSize: "$$ROOT" }
      }
    },
    {
      "$group":
      {
        "_id":null, "average":{"$avg":"$object_size"}
      }
    }
  ])
```

The query returned an average value of **361.5 bytes** for each User document with zero embedded posts. The maximum BSON document size imposed by MongoDB is **16 megabytes**, so the maximum amount for the embedded documents can be found with through the ratio:

$$Max\ N_{embedded\ docs} = \left\lfloor \frac{Document\ Size_{max} - Document\ Size_{empty\ embedded\ docs\ array}}{Embedded\ Document\ Size_{avg}} \right\rfloor$$

$$= \left\lfloor \frac{16\ 000\ 000\ B - 361.5\ B}{253\ B} \right\rfloor = 63\ 239$$

The same approach could be used to retrieve the maximum number of embedded post document inside a Place document, just changing the Document Size with empty posts array to **265.15 bytes**.
In conclusion, the upper threshold for the number of embedded post document will be respectively:

- **63 239** for User collection
- **63 240** for Place collection

This result is very unlikely to reach for a single user, while it may be for a very popular place considering a global scale application.
As a possible solution for the latter case, we thought about splitting our place document into different ones.

```
{ _id: ObjectId("61d3400ca1dc9d89ac02f85f"),
    name: 'Bagni di Nerone',
    (...)

    posts:
     [ { _id: ObjectId("61e57e23fb6caad3dd24f16f"),
         title: 'I bagni di Nerone - Pisa',
         authorUsername: 'Fruizione del patrimonio Il caso di Pisa',
         desc: 'I bagni di Nerone - Pisa',
         thumb: 'https://i.ytimg.com/vi/UInHbrfIjHM/default.jpg' },
        (...) ],

    (...)}
```

Recalling the place document structure above, we could change the structure slightly, including a **count** property:

```
{ _id: ObjectId("61d3400ca1dc9d89ac02f85f"),
    name: 'Bagni di Nerone',
    (...)

    posts:
     [ { _id: ObjectId("61e57e23fb6caad3dd24f16f"),
         title: 'I bagni di Nerone - Pisa',
         authorUsername: 'Fruizione del patrimonio Il caso di Pisa',
         desc: 'I bagni di Nerone - Pisa',
         thumb: 'https://i.ytimg.com/vi/UInHbrfIjHM/default.jpg'},
       (...) ],
    count: 5,
    (...)}
```

The idea behind this is, when we perform an **insertion** of a new post, we increment the count field every time, and we would include to the update operation a filter which might look like this:

```
 {count : { $lt : 60000} }
```

If the filter condition is not met, we insert the embedded post document into a new place document ($upsert: true).

4.2 Neo4j

4.2.1 Nodes

The nodes used in our Graph DB are the ones associated with the main entities of our application: *User*, *Place* and *Post*. We decide to maintain only the necessary attributes of the entities to make light and fast queries. In the following we explain in detail each node and its properties.

4.2.2 Place Entity

A node of the entity ***place*** contains two attributes, namely the id of the document inside the mongo collection and place name.

Field	Description
id	Corresponds to the ID of the document describing the place inside the MongoDB collection
name	Contains the name of the place

Entity Constraint

To avoid duplicate MongoDB id values for different places we introduced the following constraint on the place entity (even though this situation should never happen, as we handled it during the MongoDB population stage for the places).

```
CREATE CONSTRAINT place_id_is_unique IF NOT EXISTS
FOR (n:Place)
REQUIRE n.id IS UNIQUE
```

4.2.3 User Entity

A node of the entity ***user*** contains two attributes, namely

Field	Description
id	Corresponds to the ID of the document describing the user inside the MongoDB collection

username	Contains the username

Entity Constraint
To avoid duplicate MongoDB id values for different users we introduced the following constraint on the User entity.

```
CREATE CONSTRAINT user_id_is_unique IF NOT EXISTS
FOR (u:User)
REQUIRE u.id IS UNIQUE
```

4.2.4 Post Entity

A node of the entity ***post*** contains the following five attributes. Even though the post title, description and thumbnail are redundant attributes that were already present in Mongo, but we decided to insert them also here to be able to gather core information about the post when querying Neo4j (i.e. while suggesting new post to check out), avoiding the interaction between different databases.
Additional information about the post like, the author and the location are stored as relationship between post entity and user and place entities respectively.

Field	**Description**
id	Corresponds to the ID of the document describing the post inside the MongoDB collection
title	Contains the title of the post
description	Contains a glimps of the post's description (75 characters)
thumbnail	Contains the link to the post thumbnail

Entity Constraint
To avoid duplicate MongoDB id values for different posts we introduced the following constraint on the Post entity.

```
CREATE CONSTRAINT post_id_is_unique IF NOT EXISTS
FOR (p:Post)
REQUIRE p.id IS UNIQUE
```

4.2.5 Relationships between nodes

- **USER -AUTHOR-> POST:** this relationship is added every time a user public a new post.
- **USER -LIKES-> PLACE:** this relationship is added when a user clicks the like button on a place page, it also has a timestamp attribute that represent when the like was dropped.
- **USER -FOLLOWS-> USER:** this relationship is added when a user follows another user, it also has a timestamp attribute that represent when the follow relationship was created.
- **USER -VISITED-> PLACE:** this relationship is added when a user adds a place to the visited ones. There could be many relationships between a user and a place node if the user visits multiple times that specific place. A timestamp attribute allows to identify when the user visited the place.
- **USER -FAVOURITES-> PLACE:** this relationship is added when a user adds a place to the favorite ones, as we stated that a user could mark a place as visited without the need to post his/her experience. A timestamp attribute specifies when that relationship was added.

- **POST -LOCATION-> PLACE:** this relationship is added between a post and the place associated when that post is published by a user.

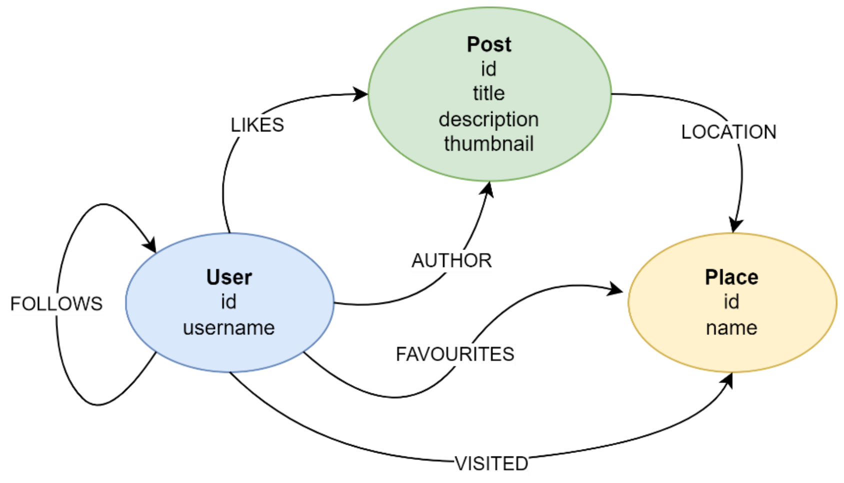


Figure 4 Nodes and relationships scheme

5 Dataset Population

We chose to use **Python** scripts to populate our main datasets: places, users and posts.
These scripts, organized by entity and data source inside ***dataPopulation*** directory, exploit different sources API to collect information.
The main idea to maintain a consistency with real world data, is to run periodically a routine that update the datasets from the same sources.

5.1 Places

The source used for the Place dataset population is **OSM** (Open Street Map). We exploit Overpass API, which provides information like place name, place type, coordinates etc.
We justify the low **variety** assuming that all the places are basically the same of the ones exposed by other sources. Moreover, also the **velocity** is very low, because our application mainly copes with points of interests, that are rarely added.

The classes in charge of managing the ***place*** collection are in **dataPopulation/places** package and are the following:

- *place.py*
- *placeFactory.py*
- *osmPlaceFactory.py*

The **Place** class is a bean class used to generate and handle the objects of our place collection.

5.1.1 OsmPlaceFactory

It is the class that loads the OSM point of interests from Overpass API and parses them to Place objects.

The **OsmPlaceFactory** class exposes two public methods:

- `search_and_parse()` which receives an area parameter that let the search be done in each area (by some cities names or by country)
- `search_and_parse_by_ids()` which gives the possibility to load from OSM the places characterized by a given id (it accepts nodes, ways, and relations).

To query the OSM for entities which respect some given properties, we exploit Overpass API.

5.1.2 Overpass API & OQL (Overpass Query Language)

The Overpass API is a read-only API that serves up custom selected parts of the Open Street map data.
It acts as a database over the web: the client sends a query to the API and gets back the data set that corresponds to the query. A client can specify several search criteria like e.g., location, type of objects, tag properties, proximity, or combinations of them.

Overpass QL (short for "Overpass Query Language") is a procedural, imperative programming language for querying the Overpass API. In our case we prepared the following query, which aim is to retrieve all the *historical nodes* and ways in *Italy* with some exceptions for the value of the attribute historic:

```
[out:json];
area["ISO3166-1:alpha2"="IT"]->.searchArea;
(
  node["historic"]["historic"!~"cannon|charcoal_pile|boundaray_stone|city_gate|creamery|farm|gallows|highwater_mark|milestone|optical_telegraph|pa|railway_car|rune_stone|vehicle|wayside_cross|wayside_shrine|yes"]["name"](area.searchArea);
  way["historic"]["historic"!~"cannon|charcoal_pile|boundaray_stone|city_gate|creamery|farm|gallows|highwater_mark|milestone|optical_telegraph|pa|railway_car|rune_stone|vehicle|wayside_cross|wayside_shrine|yes"]["name"](area.searchArea);
);
out body center meta;
>;
out skel qt;
out center meta;
```

The class OsmPlaceFactory has a private method `__load_query_string` which makes the Overpass query parameterizable. In fact it is possible to specify the search area and the set of historic values to be excluded.
There is also the possibility to include the OSM entities of kind relation in the result of the query.

To handle the OSM result set we exploit the python library ***overpy***, which let easily manage the OSM responses received from an Overpass query.

In the place object we store the country code of the place, which is obtained from the specified search area, the coordinates of the center of each entity, the name attribute from OSM, the id and an image of the place which is retrieved from the Wikipedia page associated to the entity, when exists.

The class *WikiImageFactory* is responsible of querying the Wikipedia API and obtain an image URI.
After that the place object is generated, the factory checks for a place in the database collection Place for a document that is equal or like the one just generated. In case it does not exists, the new place is stored in persistence.

5.2 Posts

To create the Post dataset, we used two different sources to increase the **variety**: YouTube for videos and Flickr for images. Both platforms provide APIs to collect information like title, author, description, publication date, etc.
As the dataset **velocity** is quite low, we assume that a routine runs periodically to update Post dataset with new videos and images from the same sources. (See chapter 5.5)

Also for this entity we use a Post bean class to use the entity inside the data population application, which also implements a method `determine_activity()` that iterates on the information of the Post (title, description, tags) and tries to find activity tags from the ones specified in **activities.json**

5.2.1 YouTube DATA API

The posts generation from YT are managed by the package YTposts, which includes:

- **YTClient** class, which handles the requests to YouTube Data API and the API keys rotations
- **YTPost** class which extends the bean class Post with an attribute YTvideoId
- **YTPostFactory** class which is responsible of the process of search and parse of YT videos from given places.

YTPostFactory exposes a method `posts_in_given_place()` which receives the information of a place and, from those:

1. performs a query to the YT data list method specifying:
    1. the given place name
    2. the given place coordinates
2. for each video result, checks if a post with the given YT video id already exists.
    1. If it exists, skips to the next video
3. For each video result, performs a query to YT for video details, then:
    1. Parses the post details from the YT video
        - Calls the UserFactory method `get_author_obj_from_YTchannel()` which returns the author User object
    2. Stores the post in the databases (both Post collection in Mongo and Neo4j)
    3. Adds the determined activity of the current post to the fits array of the given place
4. At the end of the result set, updates the YTlastSearch field of the given place

Note that the country code of the place received as parameter for the search, will be also the country code for the generated posts.

5.2.2 Flickr API

Like for YT Data API, the posts generation from Flickr are managed by the package FlickrPosts, which includes:

- **FlickrClient** class, which handles the requests to Flickr APIs
- **FlickrPost** class which extends the bean class Post with an attribute FlickrPostId
- **FlickrPostFactory** class which is responsible of the process of search and parse of Flickr images from given places.

FlickrPostFactory exposes a method `posts_in_given_place` which receives the information of a place and, from those:

1. performs a query to the Flickr list method specifying:
    1. the given place name
    2. the given place coordinates
2. for each Flickr result, checks if a post with the given YT video id already exists.
    1. If it exists, skips to the next result
3. For each Flickr result, performs a query to Flickr for post's details, then:
    1. Parses the post details from the Flickr post
        - Calls the UserFactory method `get_author_obj_from_flickr_account_id` which returns the author User object
    2. Stores the post in the databases (both Post collection in Mongo and Neo4j)
    3. Adds the determined activity of the current post to the fits array of the given place
4. At the end of the result set, updates the FlickrLastSearch field of the given place

Note that the country code of the place received as parameter for the search, will be also the country code for the generated posts.

5.3 Users

Also this entity is handled by a dedicated package, the users package. The class of the package users are:

- **User** class: the bean class that handles the User entity in the data population application
- **UserFactory** class: the class responsible for accessing the User entity in the database, creating new users and retrieving users from given information

To create the User dataset, we don't use any API or web scraping, instead we simulate real users aggregating information generated by the library ***Faker***, while some attributes like the username or the real name of the user are scraped from posts sources when possible.

5.3.1 Faker

Faker is a Python library that generates fake data, useful for testing and developing. After the instantiation of the faker generator, it can generate data by accessing properties named after the type of the data. We insert and aggregate information from Faker to fill all the fields of the User document.
The attributes of the user obtained by Faker are:

- Email
- Username (when not specified)
- Birthdates

5.3.2 UserFactory Package

UserFactory is responsible for creating the users on our platform. This module interacts with the classes that populate Post collection. When a post is retrieved from **YouTube API** it is also returned an id associated with the given YouTube channel, if a user with this id doesn't exist, it is created using Faker to fill the new user information. In this way the User collection in populated using real world association between posts and users.

5.3.2.1 UserFactory get_author_obj_from_YTchannel

As we explained above UserFactory module creates users from YouTube posts information. Which means that there is a call to a UserFactory function inside Post module, it checks if a user with the channel id retrieved from YouTube API exists:

- If **it exists**, the function returns the user object associated with the given channel id,
- If **it doesn't exist**, the function creates a user through Faker properties, binds the new user to the given channel id, then returns the new user object.

5.3.2.2 UserFactory get_author_obj_from_flickr_account_id

As we did for YouTube posts, UserFactory module also creates users from Flickr posts information.

Specifically, there is a call to a UserFactory function inside Post module, it checks if a user with the Flickr id retrieved from Flickr API exists:

- If **it exists**, the function returns the user object associated with the given Flickr id,
- If **it doesn't exist**, the function creates a user through Faker properties, binds the new user to the given Flickr id, then returns the new user object.

The country code of the generated user will be the one specified in the UserFactory method call or, if not specified, a default country code.
In the case a user has done more than one post, the country code of the user will be the country code of the first post done by that user.

5.3.2.3 UserFactory generate_social_relations

This method receives as parameter the ID of the user for which it will generate social relations.
The relations that will be generated are:

- LIKE
- FOLLOWS
- FAVOURITES
- VISITED

The number of relations generated is a random number obtained using a **.env** configuration variable SEED.
This method by default is called every time that a new user entity is generated.

5.4 Command Prompt

The Data Population application exposes a command line interface that let the main features accessible.
The main methods available are:

- ***generate posts:*** the available parameters for this command are:
  - ***YouTube*** specify this parameter to make the application perform a query over YT Data API
  - ***Flickr*** specify this parameter to make the application perform a query over Flickr API
  - ***- -num*** use this parameter to specify the maximum number of places on which the search must be performed
- ***update-redundancies:*** this method forces the run of the procedure of redundancies updating
- ***generate places:*** this method executes the `search_and_parse` method of `OsmPlaceFactory` using the area specified as parameter

Note: the command "**generate posts**" by default loads 10 places for each post source, ordered by ascending value of LastSearch.

5.5 Scheduled Procedures

The commands available from the Command Prompt are directly executable also from the shell as execution parameters of the data population script like in the following example:

```
python dataPopulation.py update-redundancies
```

By using this technique, we can run the procedures implemented in Python as shell commands.
This allowed us to use crontab for the scheduling of procedures that should be runned periodically.
The logs, stdout and stderr of the script execution are stored in a folder "/root/scheduled-procedures-logs".

5.5.1 Redundancies Updates

This procedure is responsible to update the redundancies counters that we inserted in the documents of some entities in Mongo like the field “likes” in the Post documents, the field “followers” in the User documents or the fields “favourites” and “totalLikes” in the Place documents.

This redundancies update is handled in the class RedundanciesUpdater of the package redundaciesUpdater.
The procedure to update these redundancies is to retrieve the number of relations of each kind for each redundant attribute and for each node. Then, it is sufficient to set this value in the Mongo documents.
The field “totalLikes” of each place is instead calculated using an aggregation on the “likes” attribute of each post for a given place.

The update of these redundancies is needed only when a user, all the posts of a user, or a post are deleted from the application, because the consistency in this case is demanded to this procedure, to prevent too much load on the server for the entities’ deletion.

This procedure is scheduled weekly.

5.5.2 Search for new posts from posts sources

This procedure is run by scheduling in crontab the command:

```
python dataPopulation.py generate posts youtube flickr –all-places
```

And it is scheduled weekly.

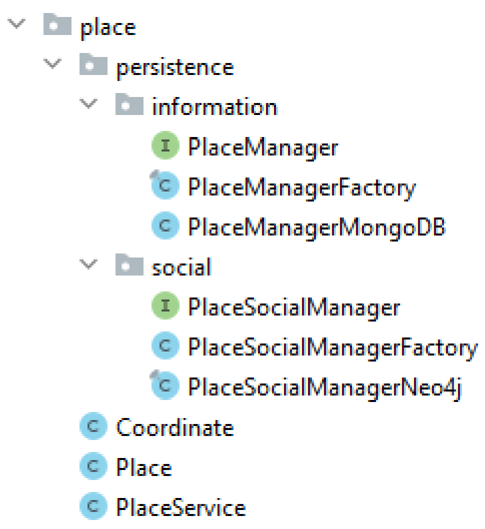
This procedure loads all the places present in MongoDB and for each place calls both the method `YTPostFactory.posts_in_given_place` and `FlickrPostFactory.posts_in_given_place`.
At the end of both run for all the places, logs the number of new posts found.

6 Backend

We developed the application backend using Java. Specifically, we exploited Spring Framework that allowed us to create endpoints and to handle the incoming requests from the client side through a set of APIs.
In addition to Java Spring, we also defined the OpenAPI Specification (the industry standard for RESTful API design) using the Swagger service.

6.1 Package Organization

As in standard conventions, we structured the package organization following the Hybrid criteria.
We divided the structure by type (entity), and inside the entity packages we divided by layer.



```
place
    persistence
        information
            PlaceManager
            PlaceManagerFactory
            PlaceManagerMongoDB
        social
            PlaceSocialManager
            PlaceSocialManagerFactory
            PlaceSocialManagerNeo4j
    Coordinate
    Place
    PlaceService
```

For instance, considering the “entity-package” *place*, we decided to separate the classes which deals with databases inside an initial sub-package called *persistence*, and to split it in turn with the *information* sub-package and *social* sub-package. Starting from the outer package we find the Bean classes and the only class that operate as an interface between the client side and the server side, PlaceService.

Going through the structure, inside *information* we find classes that represent the middleware interacting with MongoDB for data handling, whereas inside *social* the ones interacting with Neo4J.

We followed this organization criteria for all the other packages.

6.2 Database connection handling

6.2.1 Mongo DB connection handling

We use a MongoClient object to manage connections to Mongo Database. This class is designed to be thread safe and shared among threads, in this way we can create only one MongoClient instance for a given database and use it across our application.
MongoClient represents a pool of connections, we use one instance of this class even with multiple threads and we don’t need to explicitly close the connection with the database. The only resource we need to clean up is the cursor, we close it when we don’t need it anymore.

6.2.2 Neo4j Connection Handling

Neo4j client applications require a Driver Object which, from a data access perspective, forms the backbone of the application. It is through this object that all Neo4j interaction is carried out, and it should therefore be made available to all parts of the application that require data access.

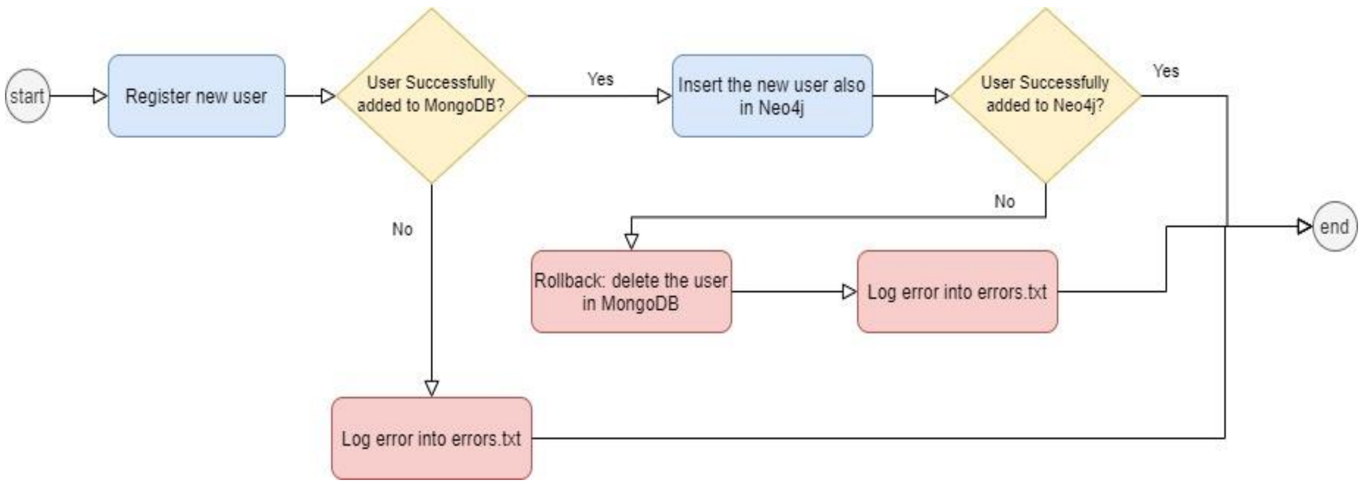
In our application, following the best practices on connections lifecycles, we constructed a Driver Object on startup and destroyed it on exit, as closing a Driver Object immediately shuts down any connections previously opened.

We embedded the Driver Object inside a Neo4jConnection class inside *persistence* package.
That connection class implements the *AutoCloseable* interface, which means that the close() method is called automatically whenever an instance needs to free the resources.
This construction ensures prompt release, avoiding resource exhaustion exceptions and errors that may otherwise occur.

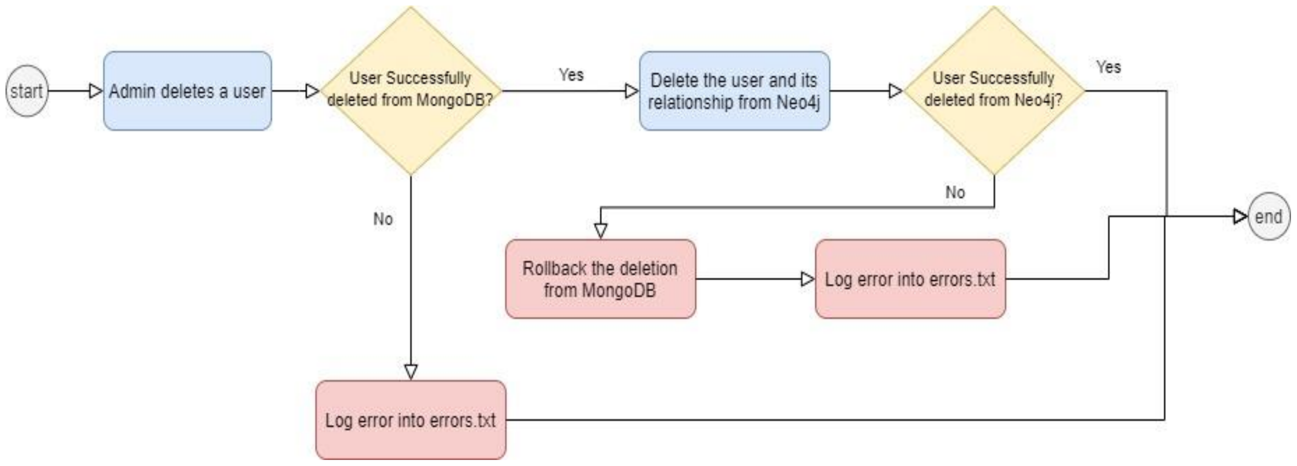
6.3 Cross-Database Consistency Management

In this section we analyze queries that requires to be handled as they involve both dbs.
In addition to perform an automatic attempt of consistency recovery, we decided to log errors into an errors.txt file, allowing administrators to manually check and enforce consistency and restored the nominal state.
The only operations requiring a cross-database consistency management are the following:

1. **Registration of a new user**


start
Register new user
User Successfully added to MongoDB?
Yes
Insert the new user also in Neo4j
User Successfully added to Neo4j?
Yes
No
end
No
Rollback: delete the user in MongoDB
Log error into errors.txt
Log error into errors.txt

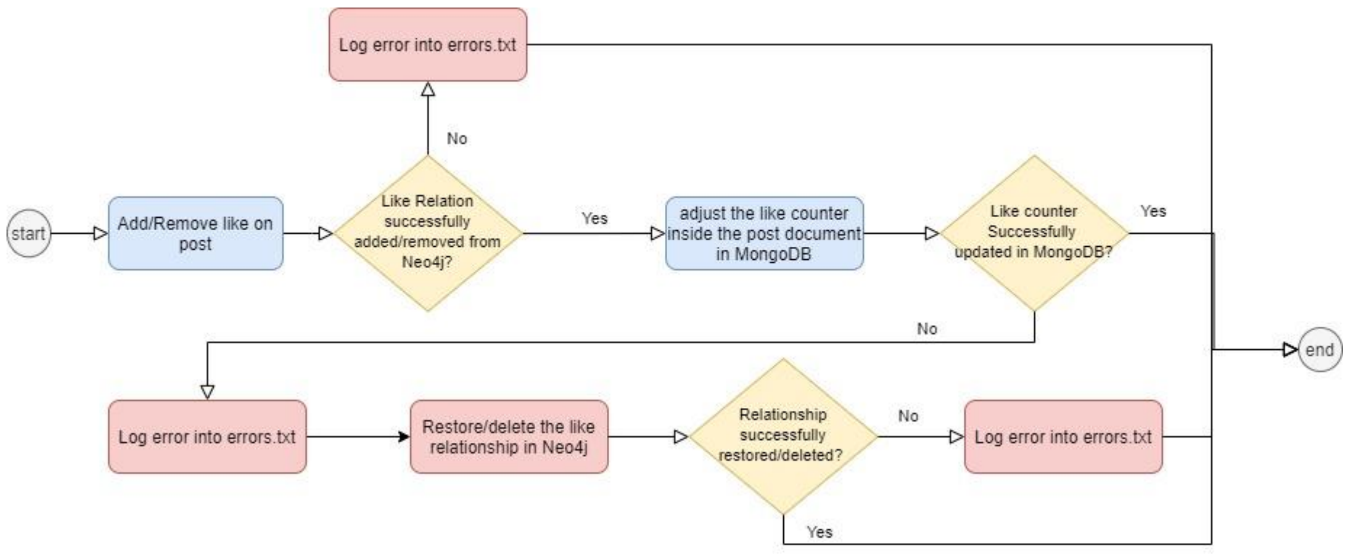
2. **Deletion of a user**


start
Admin deletes a user
User Successfully deleted from MongoDB?
Yes
Delete the user and its relationship from Neo4j
User Successfully deleted from Neo4j?
Yes
No
end
No
Rollback the deletion from MongoDB
Log error into errors.txt
Log error into errors.txt

3. **Creation of new post**
   Similar flow chart as n.1, instead we insert the newly created post inside Neo4j.

4. **Deletion of a post**
   Similar flow chart as n.2, instead we delete the post from Neo4j.

5. **Add/Remove like on post**


start
Add/Remove like on post
Like Relation successfully added/removed from Neo4j?
No
Log error into errors.txt
Yes
adjust the like counter inside the post document in MongoDB
Like counter Successfully updated in MongoDB?
Yes
end
No
Log error into errors.txt
Restore/delete the like relationship in Neo4j
Relationship successfully restored/deleted?
No
Log error into errors.txt
Yes

6. **Mark/Unmark a place as favorite**
   Similar flow chart as n.5, instead we update the **favorite counter** inside the place document in MongoDB.

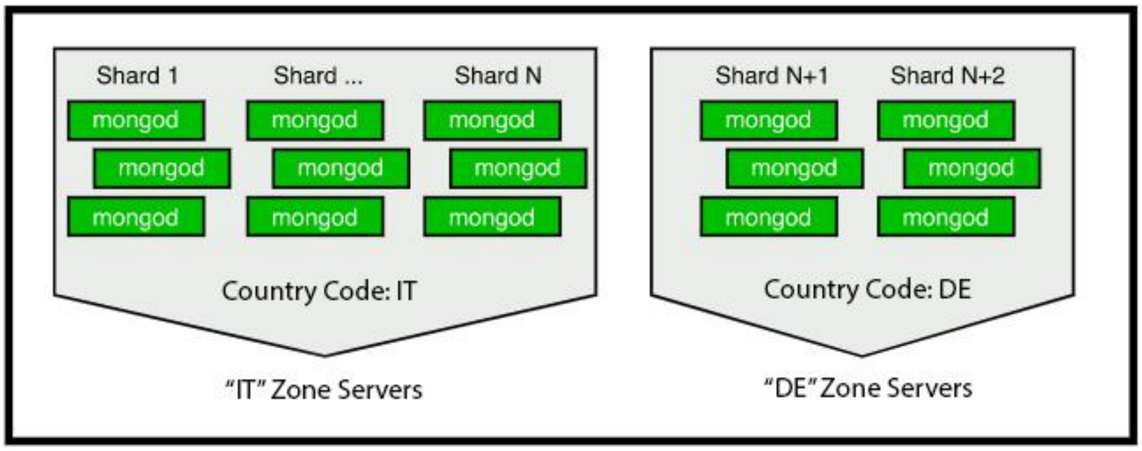
7. **Follow/unfollow a user**
   Similar flow chart as n.5, instead we update the **followers counter** inside the user document in MongoDB.

6.4 Sharding: a possible implementation

To satisfy the non-functional requirements, as the availability of the service, we propose a possible sharding method. We can consider the **country code** as a shard key for the following collections: user, place, and post.

For this reason, we added the field **countryCode** to them so that we can choose a partitioning method based on **lists**. In this way data is divided by country on the cluster and every server could handle in a better way incoming requests for resources of a specific country.

The following diagram illustrates a possible sharded cluster that uses geographic based zones to manage and satisfy our data segmentation requirements:



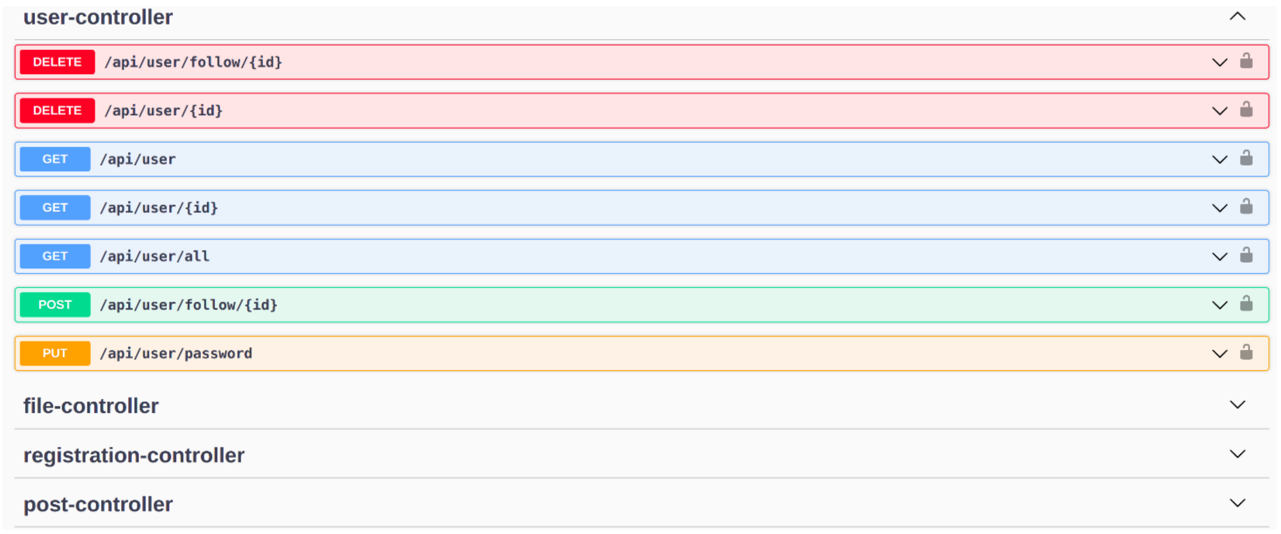
The result is that requests are well-distributed, so the response latency to the client side is reduced leading to a better user experience.

7 Frontend

7.1 Swagger

In order to develop our APIs with the OpenAPI Specification we decided to exploit the open source tool sets Swagger. It is an Interface Description Language for describing RESTful APIs expressed using JSON, that helped us to generate an Open API document based on the code itself.

This embeds the API description in the source code of a project, and it has let us construct a bottom-up API development. In fact, starting from our Java code and Spring
Web server controllers, it reflects the exact implementation adding the extra GUI for an easier navigation through the endpoints.


user-controller

DELETE /api/user/follow/{id}

DELETE /api/user/{id}

GET /api/user

GET /api/user/{id}

GET /api/user/all

POST /api/user/follow/{id}

PUT /api/user/password

file-controller

registration-controller

post-controller

This is how it renders the endpoints with relative methods:

The Lock icon on the right side allow to identify users through exploiting JWT Authentication mechanism and enables the methods in case of success.
In addition to that, we can generate our API specification in Json format, following the OpenAPI 3.0 Standard, just browsing the following address:

http://localhost:8080/api-docs

This feature helps maintain the portability of our application.

8 Queries analysis

8.1 Operation Frequency Table

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:

8.1.1 MongoDB Frequency Table

Read Operations

Operation	Expected Frequency	Cost
Retrieve information on a user	Low	Low (1 read)
Retrieve information on a post	High	Low (1 read)
Retrieve information on a place	High	Low (1 read)
Retrieve nearby places suitable for a certain activity	High	Average (multiple reads)
Retrieve the most popular places filtering by activity	Average	High (Aggregation)
Retrieve the most popular posts of a given period	Average	High (Aggregation)
Retrieve the most active users	Low	High (Aggregation)

Write Operations

Operation	Expected Frequency	Cost
Register a new user into the database	Average	Average (document insertion)
Update of the user password	Low	Low (1 attribute write)
Update likes counter of a post	Average	Low (1 attribute write)
Update favourite counter of a place	Average	Low (1 attribute write)
Update followers counter of a user	Average	Low (1 attribute write)
New post submission	High	Low (1 read)
Delete a post	Low	Average
Delete a user	Low	High (document deletion)

8.1.2 Neo4j Frequency Table

Read Operations

Operation	Expected Frequency	Cost
Get visited places by the user	Average	Low (1 read)
Get favourite places of a user	Average	Low (1 read)
Get the suggested places for a user	High	High (complex aggregation)
Get the suggested posts based on user likes	High	High (complex aggregation)
Get the suggested user to follow (co-followers)	Average	High (complex aggregation)

Write Operations

Operation	Expected Frequency	Cost

Register a new user into the database	Average	Low (1 node creation)
Add new like to a post	High	Low (1 relationship creation)
Mark a new place as favourite	Average	Low (1 relationship creation)
Mark a new place as visited	Average	Low (1 relationship creation)
Start following a new user	Average	Low (1 relationship creation)
New post submission	High	Low (1 node creation)
Delete a user	Low	High (deletion of multiple relationships, remove *1 + n* nodes)

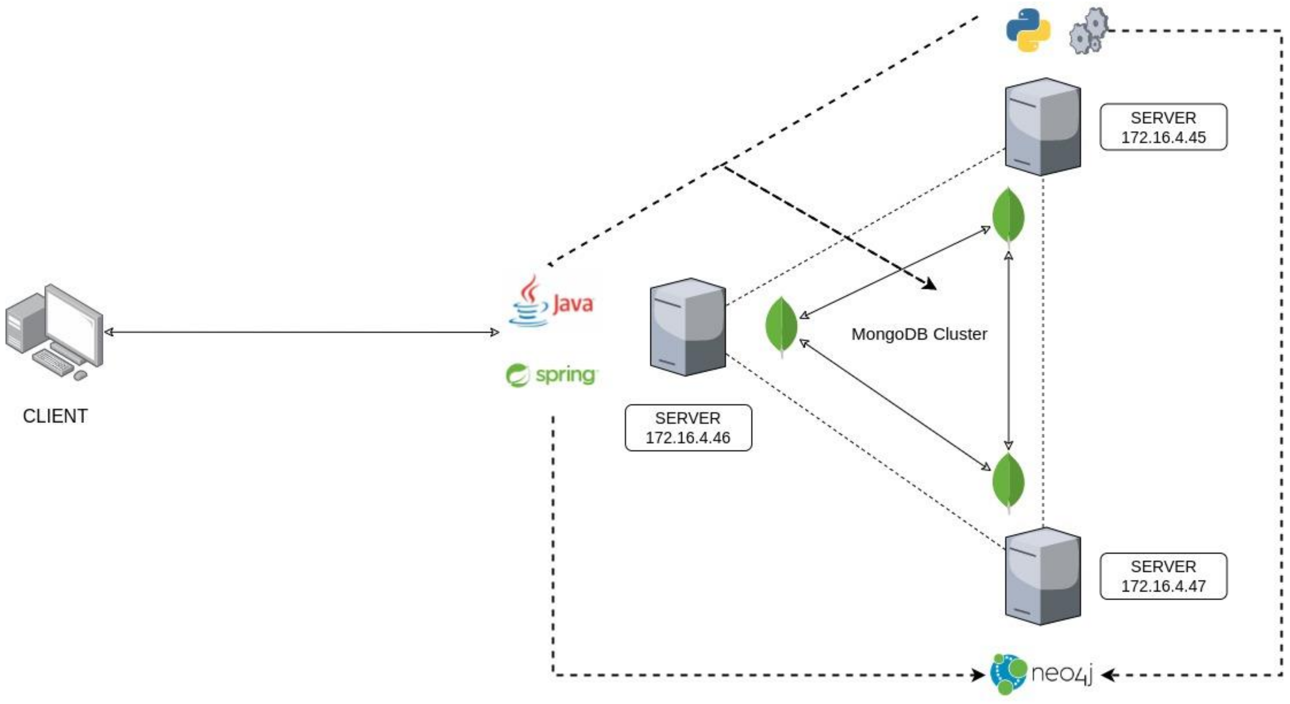
As extracted from the tables above, write operations are less frequent than the read operations. The latter are more frequent and are performed on higher number of documents. In addition, the requirements of the system are availability and fast response time, for these reasons we set:

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

The first one is to allow the system to perform the write operation as quick as possible, the writes affect only the primary node and replicas will be updated later (eventual consistency paradigm).

The nearest read preferences specifies that the read operations are performed on the node with the fastest response, that is, the one with the least network latency. In fact, the driver reads from a member whose network latency falls within the acceptable latency window. So, we do not consider whether a member is a primary or secondary when routing read operations: primaries and secondaries are treated equivalently.
We chose this mode against **primary preferred** read preference to minimize the effect of network latency on read operations, in this way we decrease the response time of the application at the expenses of consistency, meeting non-functional requirements.

8.2 Cluster Organization



As shown in the picture above, we decided to organize our cluster in this way:

- Server 172.16.4.47:
  - Mongo instance
  - Neo4j server
- Server 172.16.4.46:
  - Mongo instance
  - Java server
- Server 172.16.4.45:
  - Mongo instance
  - Python scripts scheduled

The priority for the Mongo server was driven by the logic to ask first to server 172.16.4.45 (as it has the lowest medium payload, python runs only periodically every given period), then ask to server 172.16.4.46 (it also runs Java) and last priority to server 172.16.4.47 (because it has also to handle Neo4j queries).
We then assigned the following priorities into the *rsconfig parameter:*

members:
{_id: 0, host: "172.16.4.47:27020", priority=1},
{_id: 1, host: "172.16.4.46:27020", priority=5},
{_id: 2, host: "172.16.4.45:27020", priority=10}

8.3 Indexes

8.3.1 Place index

We consider the introduction of indexes in the place collection. This collection is particularly suitable for index introduction because a small number of write operations are performed on it, this means slow updates are avoided.

2dsphere Index

To support queries on geospatial data in our application, we decided to exploit the ***$near*** operator, provided by MongoDB.
This operator takes a GeoJSON point for which a geospatial query returns the documents from nearest to farthest, but requires a geospatial index to work, called **2dsphere.**
Following the standard convention, we have stored our location data as GeoJSON objects to calculate geometry over an Earth-like sphere and exploting ***$near*** operator.
To specify GeoJSON data, we have used an embedded document with:

- a field named *type* that specifies the GeoJSON object type
- a field named *coordinates* that specifies the object's coordinates.

To create the 2d index, we used the db.collection.createIndex() method, specifying the location field as the key and the string literal "2d" as the index type:

```
db.collection.createIndex( { loc: "2d" } )
```

We have done the following considerations about adding other indexes.
We consider a text index on the **name** field, useful when we perform queries to search places by name. The place name is rarely updated, so this index doesn't affect nor write operations on place collection, nor the request latency.

Base case:

```
db.place.find(
    {$and:[
        {name:{$regex:/.*chiesa.*/i}},
        {favs:{$gte:4}},
        {totalLikes:{$gte:500}}
        ]
    }).hint({_id:1}).explain("executionStats")
```

```
{
nReturned: 2,
executionTimeMillis: 8,
totalKeysExamined: 1344,
totalDocsExamined: 1344,
}
```

Text index on name:

```
db.place.find(
    {$and:[
        {$text: {$search: "chiesa"}},
        {favs:{$gte:4}},
        {totalLikes:{$gte:500}}
        ]
    }).explain("executionStats")
```

```
{
    nReturned: 2,
    executionTimeMillis: 1,
    totalKeysExamined: 88,
    totalDocsExamined: 176
}
```

Final Place Index Configuration

We decided to adopt the text index on place name, because as the result shows, the number of total documents and keys examined significantly decrease.

	1°	2°	3°
Indexed Fields	**_id**	**loc**	**name**
Indexes Type	Default Index	Geospatial Index (2dsphere)	Text Index

8.3.2 Post Indexes

We consider the introduction of an index in the post collection. Write operations are performed hugely on this collection, so we need to introduce and index that doesn't slow down the post creation and updating.

Base case:

```
db.post.find(
    {$and:[
        {postDate:{$gte:ISODate("2020-05-01")}},
        {likes:{$gte:1}}
        ]
    }).hint({_id:1}).explain("executionStats")
```

```
{
nReturned: 1937,
executionTimeMillis: 123,
totalKeysExamined: 8508,
totalDocsExamined: 8508
}
```

One possible approach could be the creation of an index based on **postDate**, that is never updated by the user. On the other hand, the creation of a post, that include a write operation will become slower.

Date index on postDate:

```
db.post.find(
    {$and:[
        {postDate:{$gte:ISODate("2020-05-01")}},
        {likes:{$gte:1}}
        ]
    }).hint({postDate:1}).explain("executionStats")
```

```
{
nReturned: 1937,
executionTimeMillis: 5,
totalKeysExamined: 2247,
totalDocsExamined: 2247,
}
```

As we can see the number of keys and documents examined is significantly decreased, and the execution time, so we decided to adopt this ascending index.

We consider also to add a text index on post **title**, to support the queries that use a text search on that attribute. These queries are often performed, and the post collection is the one with the higher number of documents in it.

Base case search:

```
db.post.find(
    {$and:[
        {title:{$regex: /.*esplorazione.*/i}},
        {likes:{$gte:5}}
        ]
    }).hint({_id:1}).explain("executionStats")
```

```
{
    nReturned: 1,
    executionTimeMillis: 24,
    totalKeysExamined: 8508,
    totalDocsExamined: 8508
}
```

Text index on title:

```
db.post.find(
    {$and:[
        {$text:{$search: "esplorazione"}},
        {likes:{$gte:5}}
        ]
    }).explain("executionStats")
```

```
{
    nReturned: 1,
     executionTimeMillis: 1,
     totalKeysExamined: 1,
     totalDocsExamined: 2
}
```

As we can see from the results, the execution time is significantly reduced, like the number of keys and documents examined. So we decide to introduce also this index.

Final Post Index Configuration

	1°	2°	3°
Indexed Fields	**_id**	**postDate**	**title**
Indexes Type	Default Index	Ascending	Text

8.3.3 User Indexes

One possible index on user collection could be the **followers** field, but it is not suitable because this field is updated every time a user follows/unfollows another one, so the index needs to be updated very frequently slowing down the write operations performance.
Another possibility is to add an index on **birthDate**, this field is written only when the user is created, that means only this write operation is slowed down by the index introduction, but we rarely execute queries with a filter on that attribute.
Username field is often used in queries to retrieve Users, so an index on this attribute could lead to a faster response.
Considering that the username field is usually accessed in read operations and never in writes (as a user cannot change its username), we could evaluate the performance improvement of a text index on that attribute.

Base case:

```
db.user.find(
    {$and:[
        {username:{$regex:/.*micky.*/i}},
        {followers:{$gte:5}}
        ]
    }).hint({_id:1}).explain("executionStats")
```

```
{
    nReturned: 1,
    executionTimeMillis: 12,
    totalKeysExamined: 2232,
    totalDocsExamined: 2232,
}
```

Index on username:

```
db.user.find(
    {$and:[
        {$text:{$search:"micky"}},
        {followers:{$gte:5}}
        ]
    }).explain("executionStats")
```

```
{
    nReturned: 1,
    executionTimeMillis: 1,
    totalKeysExamined: 1,
    totalDocsExamined: 2
}
```

As a result, the number of docs and keys examined drastically reduced, and also the execution time was shorter.

Final User Index Configuration

	1°	2°
Indexed Fields	**_id**	**username**
Indexes Type	Default Index	text

8.4 Analytics

8.4.1 MongoDB analytics

To extract interesting and useful information from data we chose the following pipelines aggregation.

8.4.1.1 Most popular places

Description: this query selects the most appreciated places in terms of likes received by posts located in them, and in case of score tied, by the number of favorites.
Mandatory parameter: none
Optional parameters: activity name and maximum number of places to return
Java location: PlaceService.getPopularPlaces

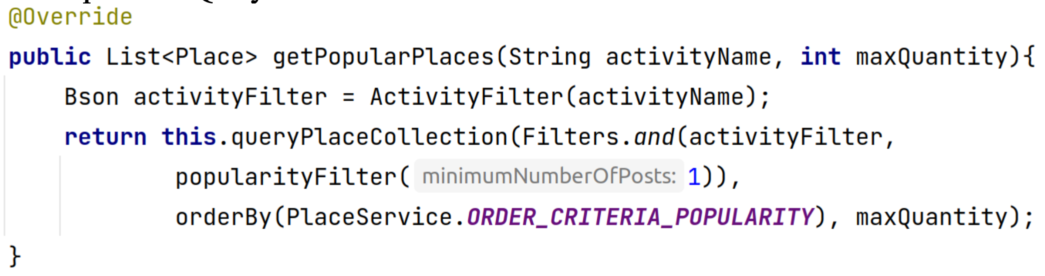
```
db.place.aggregate([
    {$match:
      {fits:{$in:["activityName"]}}
    },
    {
      $addFields:{"placeId":{$toString:"$_id"}}
    },
    {$lookup:
      {
        from:"post",
        localField:"placeId",
        foreignField:"place",
        as:"posts"
      }
    },
    {
      $unwind:"$posts"
    },
    {$group:
      {
        _id:"$_id",
        placename:{$first:"$name"},
        favs:{$first:"$favs"},
        count:{$sum:"$posts.likes"}
      }
    },
    {$sort:{count:-1, favs:-1}},
    {$limit: "howManyResults"},
  ])
```

Same aggregation with a redundant field in Place documents that stores **total number of likes** received by posts in that specific place.

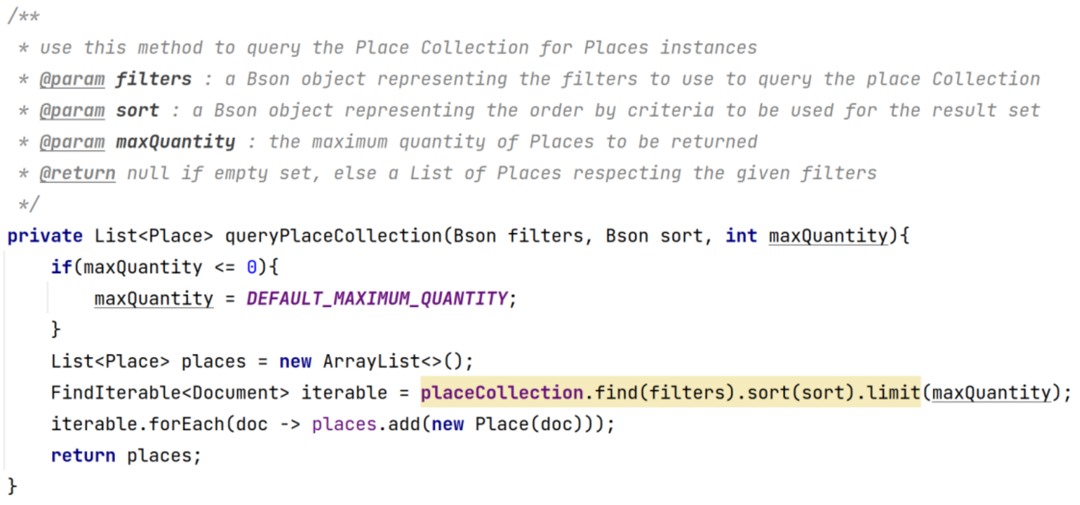
```
db.place.aggregate([
    {$match:
      {fits:{$in:["activityName"]}}
    },
    {$sort:{totalLikes:-1, favs:-1}},
    {$project: {name:1, totalLikes:1, favs:1}},
    {$limit: "howManyResults"}
  ])
```

We decided to add the redundant attribute “totalLikes” because this analytic query can be performed much faster using this redundancy, as we do not have to access another collection using a $lookup and we do not have to perform a $group operation. The query becomes as easy as doing a $match and a $sort.

Java Equivalent Query



```java
@Override
public List<Place> getPopularPlaces(String activityName, int maxQuantity){
    Bson activityFilter = ActivityFilter(activityName);
    return this.queryPlaceCollection(Filters.and(activityFilter,
            popularityFilter( minimumNumberOfPosts: 1)),
            orderBy(PlaceService.ORDER_CRITERIA_POPULARITY), maxQuantity);
}
```



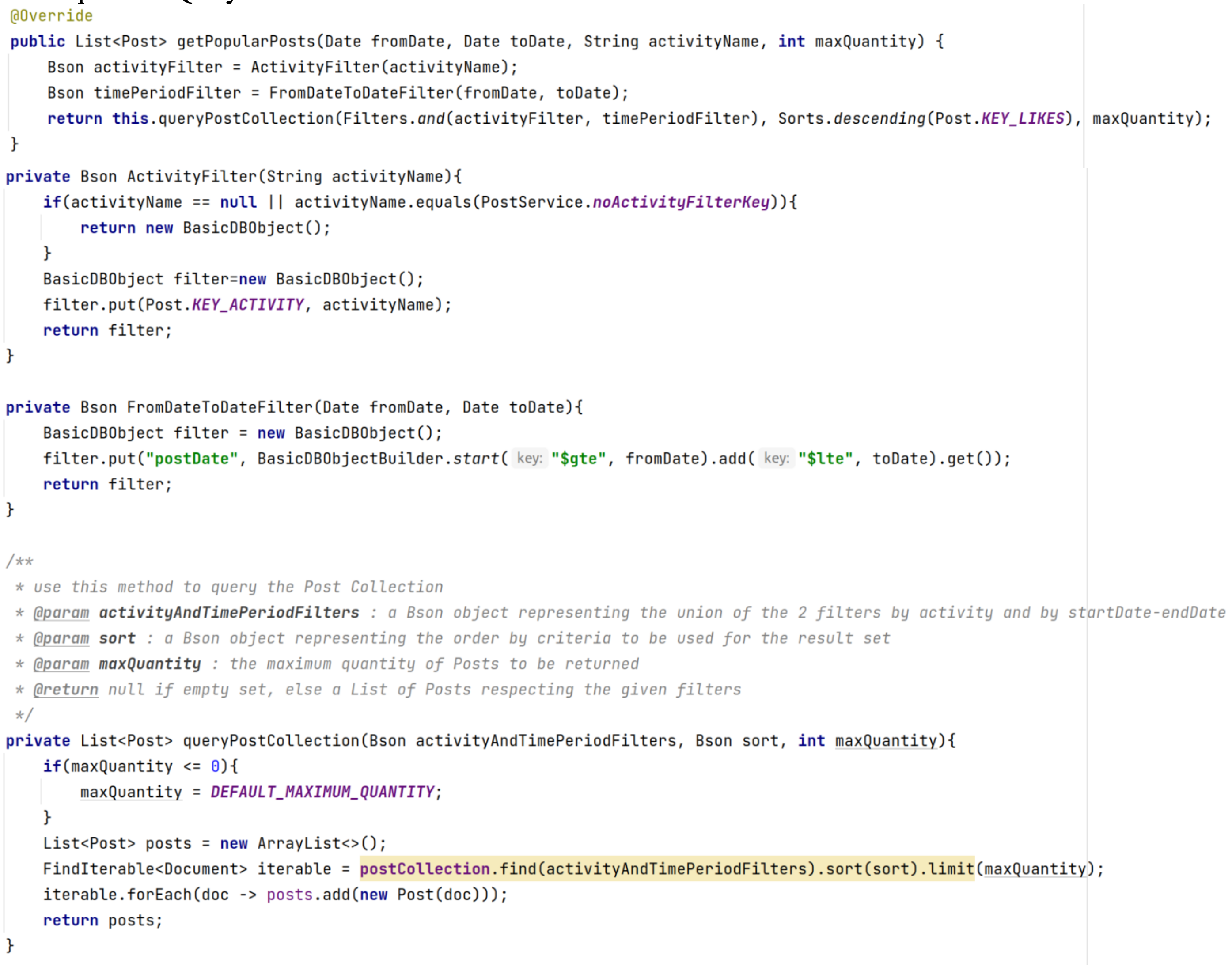
```java
/**
 * use this method to query the Place Collection for Places instances
 * @param filters : a Bson object representing the filters to use to query the place Collection
 * @param sort : a Bson object representing the order by criteria to be used for the result set
 * @param maxQuantity : the maximum quantity of Places to be returned
 * @return null if empty set, else a List of Places respecting the given filters
 */
private List<Place> queryPlaceCollection(Bson filters, Bson sort, int maxQuantity){
    if(maxQuantity <= 0){
        maxQuantity = DEFAULT_MAXIMUM_QUANTITY;
    }
    List<Place> places = new ArrayList<>();
    FindIterable<Document> iterable = placeCollection.find(filters).sort(sort).limit(maxQuantity);
    iterable.forEach(doc -> places.add(new Place(doc)));
    return places;
}
```

8.4.1.2 Most popular posts of a given period

Description: this query selects the most appreciated posts, in terms of likes received, in a period between two dates and filtering by an activity.
Mandatory parameter: fromDate, toDate
Optional parameters: activity name and maximum number of posts to return
Java location: PostService.getPopularPosts

```
db.post.aggregate([
  {$match:
    {activity:{$in:["activityName"]}},
  },
  {$match:
    {postDate:{$gte:"fromDate", $lt: "toDate" }}
  },
  {$sort:{likes:-1}},
  {$project:
    {
      _id:0,
      title:1,
      authorUsername:1,
      placeName:1,
      desc:1,
      postDate:1
    }
  },
  {$limit: "howManyResults"}
])
```

Java Equivalent Query



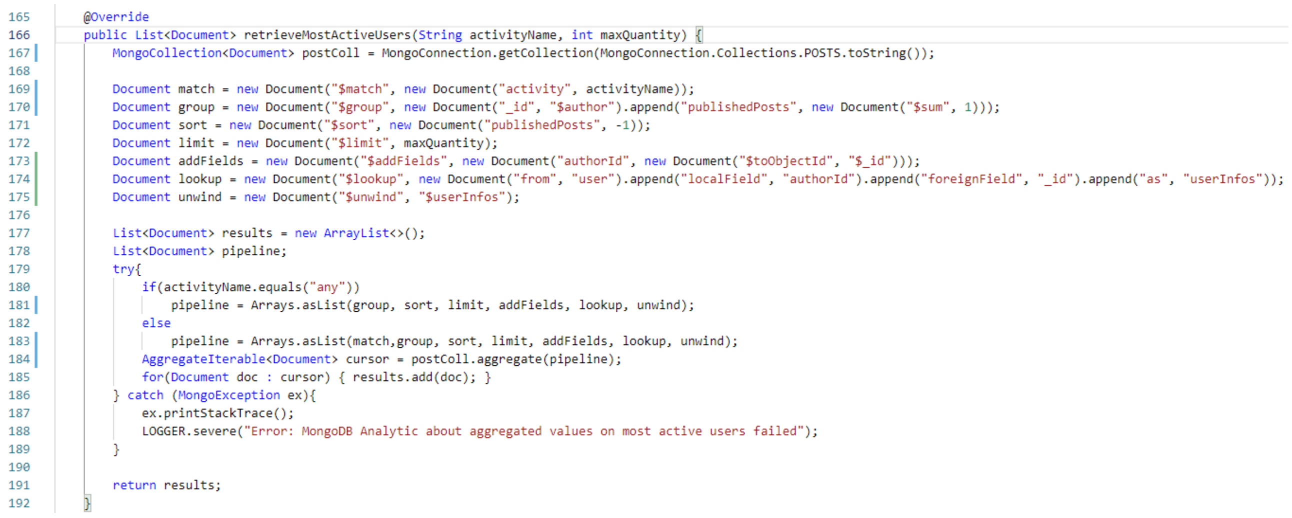
```java
@Override
public List<Post> getPopularPosts(Date fromDate, Date toDate, String activityName, int maxQuantity) {
    Bson activityFilter = ActivityFilter(activityName);
    Bson timePeriodFilter = FromDateToDateFilter(fromDate, toDate);
    return this.queryPostCollection(Filters.and(activityFilter, timePeriodFilter), Sorts.descending(Post.KEY_LIKES), maxQuantity);
}
private Bson ActivityFilter(String activityName){
    if(activityName == null || activityName.equals(PostService.noActivityFilterKey)){
        return new BasicDBObject();
    }
    BasicDBObject filter=new BasicDBObject();
    filter.put(Post.KEY_ACTIVITY, activityName);
    return filter;
}

private Bson FromDateToDateFilter(Date fromDate, Date toDate){
    BasicDBObject filter = new BasicDBObject();
    filter.put("postDate", BasicDBObjectBuilder.start( key: "$gte", fromDate).add( key: "$lte", toDate).get());
    return filter;
}

/**
 * use this method to query the Post Collection
 * @param activityAndTimePeriodFilters : a Bson object representing the union of the 2 filters by activity and by startDate-endDate
 * @param sort : a Bson object representing the order by criteria to be used for the result set
 * @param maxQuantity : the maximum quantity of Posts to be returned
 * @return null if empty set, else a List of Posts respecting the given filters
 */
private List<Post> queryPostCollection(Bson activityAndTimePeriodFilters, Bson sort, int maxQuantity){
    if(maxQuantity <= 0){
        maxQuantity = DEFAULT_MAXIMUM_QUANTITY;
    }
    List<Post> posts = new ArrayList<>();
    FindIterable<Document> iterable = postCollection.find(activityAndTimePeriodFilters).sort(sort).limit(maxQuantity);
    iterable.forEach(doc -> posts.add(new Post(doc)));
    return posts;
}
```

8.4.1.3 Most active users

Description: this query selects the most active users for a given activity, in terms of posts published.
Mandatory parameter: none
Optional parameters: activity type and maximum number of users to return
Java location: UserService.mostActiveUsersByActivity

```
db.post.aggregate([
  {$match:{activity:"activityName"}},
  {$group:{_id:"$author", publishedPosts:{$sum:1}}},
  {$sort:{publishedPosts:-1}},
  {$limit: "howManyResults"},
  {$addFields:{"authorId" : {$toObjectId : "$_id"}}},
{$lookup: {from: "user", localField:"authorId", foreignField:"_id",
as:"userInfos"}},
{$unwind:"$userInfos"}
])
```

Java Equivalent Query



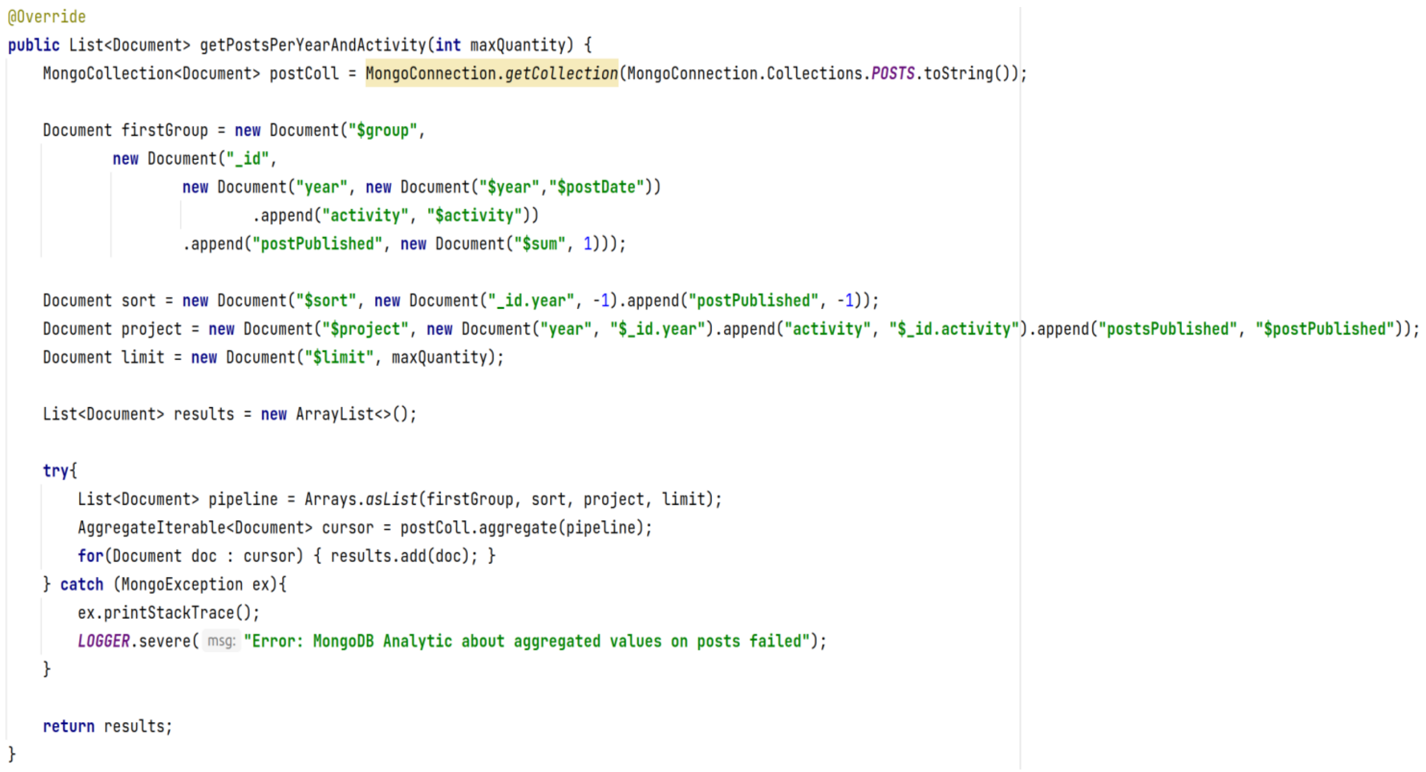
```java
@Override
public List<Document> retrieveMostActiveUsers(String activityName, int maxQuantity) {
    MongoCollection<Document> postColl = MongoConnection.getCollection(MongoConnection.Collections.POSTS.toString());

    Document match = new Document("$match", new Document("activity", activityName));
    Document group = new Document("$group", new Document("_id", "$author").append("publishedPosts", new Document("$sum", 1)));
    Document sort = new Document("$sort", new Document("publishedPosts", -1));
    Document limit = new Document("$limit", maxQuantity);
    Document addFields = new Document("$addFields", new Document("authorId", new Document("$toObjectId", "$_id")));
    Document lookup = new Document("$lookup", new Document("from", "user").append("localField", "authorId").append("foreignField", "_id").append("as", "userInfos"));
    Document unwind = new Document("$unwind", "$userInfos");

    List<Document> results = new ArrayList<>();
    List<Document> pipeline;
    try{
        if(activityName.equals("any"))
            pipeline = Arrays.asList(group, sort, limit, addFields, lookup, unwind);
        else
            pipeline = Arrays.asList(match,group, sort, limit, addFields, lookup, unwind);
        AggregateIterable<Document> cursor = postColl.aggregate(pipeline);
        for(Document doc : cursor) { results.add(doc); }
    } catch (MongoException ex){
        ex.printStackTrace();
        LOGGER.severe("Error: MongoDB Analytic about aggregated values on most active users failed");
    }

    return results;
}
```

8.4.1.4 Number of posts per year and activity

Description: this query retrieves the number of posts, filtered by year and activity
Mandatory parameter: none
Optional parameters: none
Java location: PostService.getPostsPerYearAndActivity

```
db.post.aggregate([
  {$group:
    {
      _id: {year:{$year:"$postDate"}, activity:"$activity"},
      postPublished: {$sum:1}
    }
  },
  {
    $sort: {"_id.year":-1, postPublished:-1}
  },
  {$project:
    {
      _id:0,
      "year": "$_id.year",
      "activity": "$_id.activity",
      "postsPublished": "$postPublished"
    }
  }
])
```

Java Equivalent Query



```java
@Override
public List<Document> getPostsPerYearAndActivity(int maxQuantity) {
    MongoCollection<Document> postColl = MongoConnection.getCollection(MongoConnection.Collections.POSTS.toString());

    Document firstGroup = new Document("$group",
            new Document("_id",
                    new Document("year", new Document("$year","$postDate"))
                            .append("activity", "$activity"))
                    .append("postPublished", new Document("$sum", 1)));

    Document sort = new Document("$sort", new Document("_id.year", -1).append("postPublished", -1));
    Document project = new Document("$project", new Document("year", "$_id.year").append("activity", "$_id.activity").append("postsPublished", "$postPublished"));
    Document limit = new Document("$limit", maxQuantity);

    List<Document> results = new ArrayList<>();

    try{
        List<Document> pipeline = Arrays.asList(firstGroup, sort, project, limit);
        AggregateIterable<Document> cursor = postColl.aggregate(pipeline);
        for(Document doc : cursor) { results.add(doc); }
    } catch (MongoException ex){
        ex.printStackTrace();
        LOGGER.severe( msg: "Error: MongoDB Analytic about aggregated values on posts failed");
    }

    return results;
}
```

8.4.2 Neo4j analytics

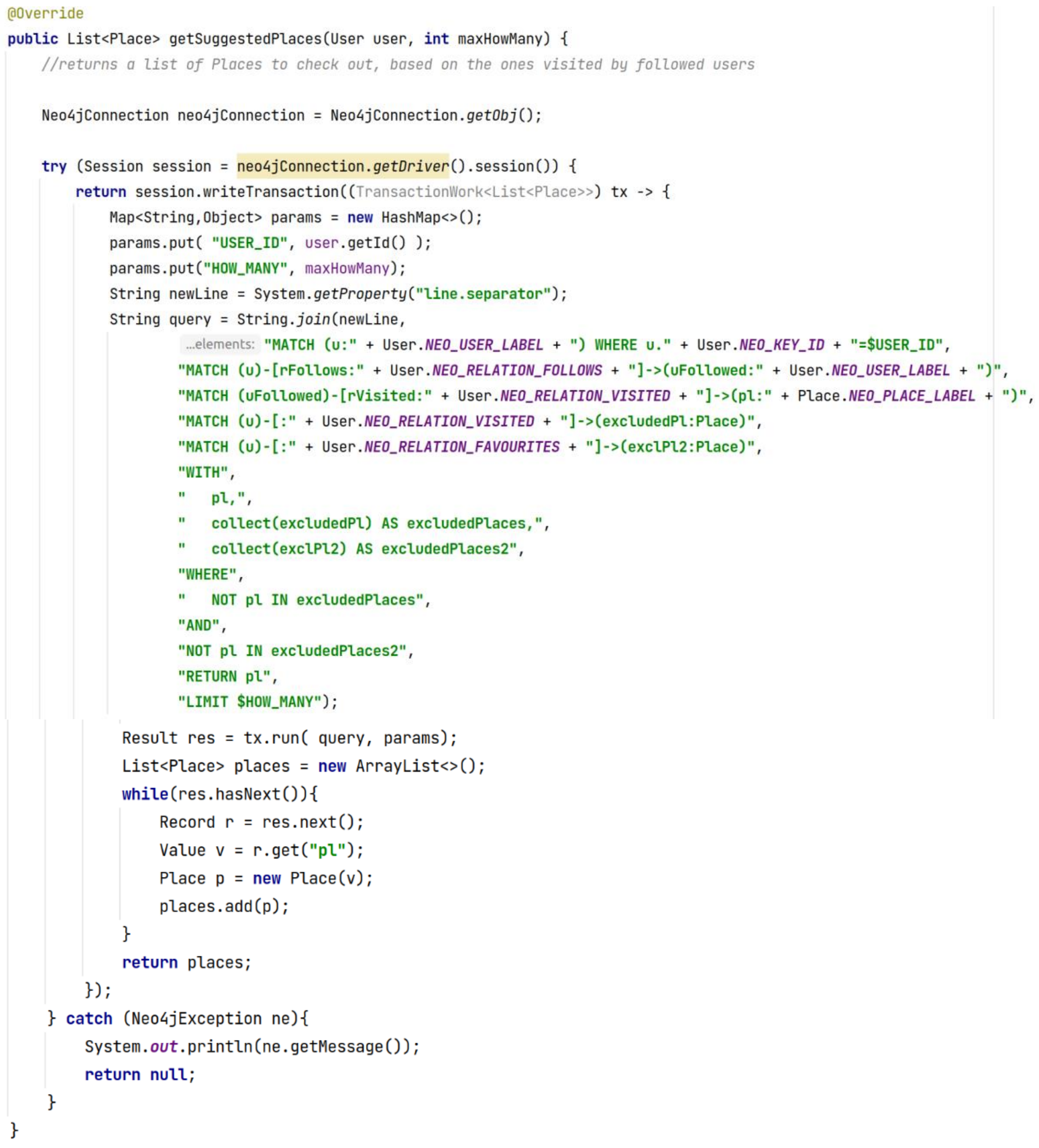
To extract interesting and useful information from Neo4j data we chose the following cypher queries.

8.4.2.1 Suggested places by followings

Graphic-centric query	Domain query
Considering U as all the User vertices that have an incoming edge “FOLLOWS” from a specific User vertex, select Place vertices that have an incoming edge “VISITED” from U vertices. Then count the incoming “VISITED” edges for each of those places.	What are the most visited places, between the ones visited by the followings of a specified user?

```cypher
MATCH (u:User{username:$username})-[f:FOLLOWS]->(followings:User)-[v:VISITED]
->(p:Place)
WITH p.id AS id, p.name AS place, count(v) AS visitTimes
ORDER BY visitTimes DESC
LIMIT $howManyResults
RETURN id, place, visitTimes
```

Java Equivalent Query



```java
@Override
public List<Place> getSuggestedPlaces(User user, int maxHowMany) {
    //returns a list of Places to check out, based on the ones visited by followed users

    Neo4jConnection neo4jConnection = Neo4jConnection.getObj();

    try (Session session = neo4jConnection.getDriver().session()) {
        return session.writeTransaction((TransactionWork<List<Place>>) tx -> {
            Map<String,Object> params = new HashMap<>();
            params.put( "USER_ID", user.getId() );
            params.put("HOW_MANY", maxHowMany);
            String newLine = System.getProperty("line.separator");
            String query = String.join(newLine,
                    ...elements: "MATCH (u:" + User.NEO_USER_LABEL + ") WHERE u." + User.NEO_KEY_ID + "=$USER_ID",
                    "MATCH (u)-[rFollows:" + User.NEO_RELATION_FOLLOWS + "]->(uFollowed:" + User.NEO_USER_LABEL + ")",
                    "MATCH (uFollowed)-[rVisited:" + User.NEO_RELATION_VISITED + "]->(pl:" + Place.NEO_PLACE_LABEL + ")",
                    "MATCH (u)-[:" + User.NEO_RELATION_VISITED + "]->(excludedPl:Place)",
                    "MATCH (u)-[:" + User.NEO_RELATION_FAVOURITES + "]->(exclPl2:Place)",
                    "WITH",
                    "   pl,",
                    "   collect(excludedPl) AS excludedPlaces,",
                    "   collect(exclPl2) AS excludedPlaces2",
                    "WHERE",
                    "   NOT pl IN excludedPlaces",
                    "AND",
                    "NOT pl IN excludedPlaces2",
                    "RETURN pl",
                    "LIMIT $HOW_MANY");
            Result res = tx.run( query, params);
            List<Place> places = new ArrayList<>();
            while(res.hasNext()){
                Record r = res.next();
                Value v = r.get("pl");
                Place p = new Place(v);
                places.add(p);
            }
            return places;
        });
    } catch (Neo4jException ne){
        System.out.println(ne.getMessage());
        return null;
    }
}
```

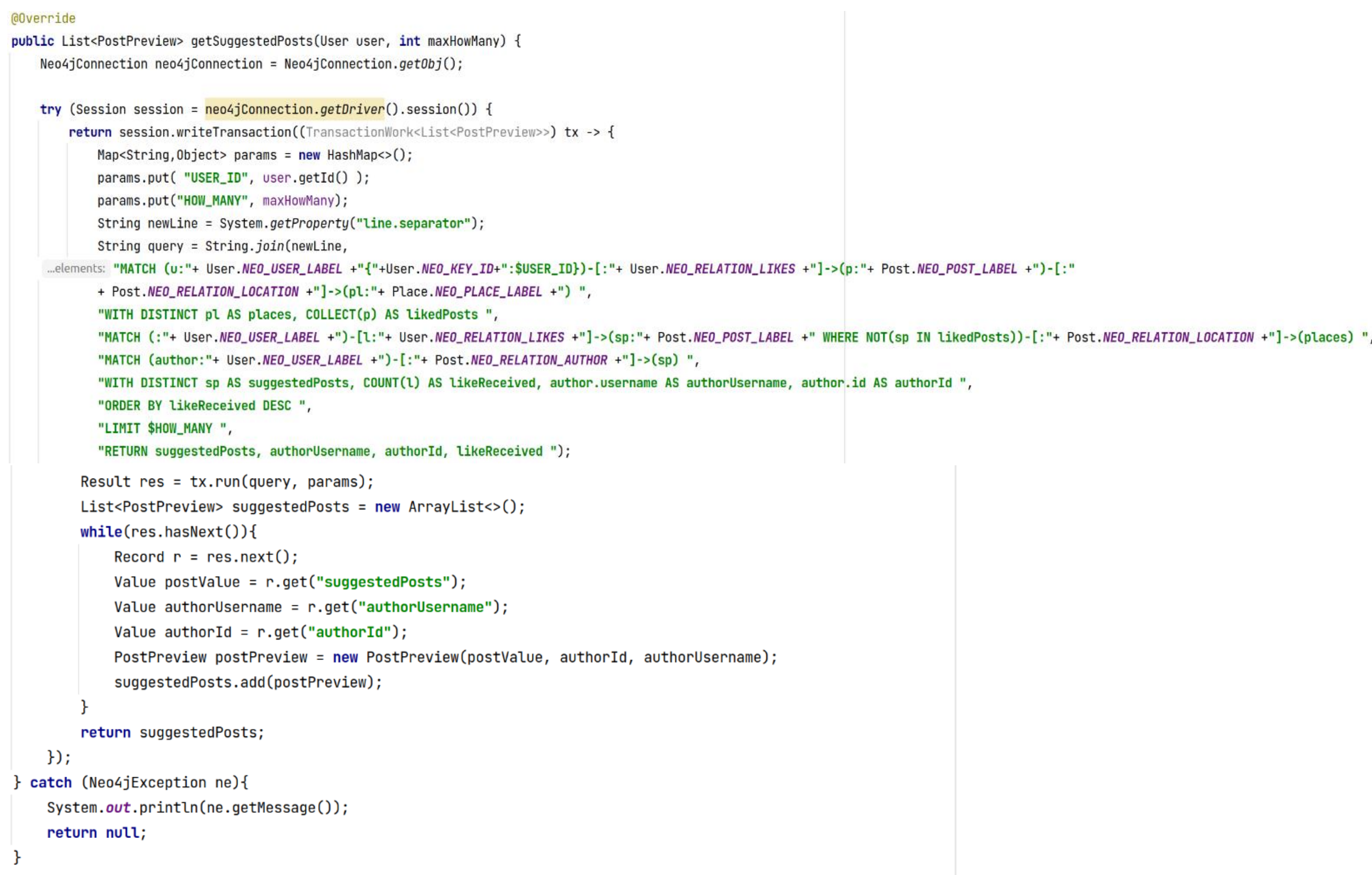
8.4.2.2 Suggested posts based on user likes

Graphic-centric query	Domain query
Considering P as the Post vertices with an incoming edge “LIKES” from a specific User vertex, select PL as the Place vertices that have an incoming edge “LOCATION” from P. Then considering the Post vertices with an outgoing edge “LOCATION” from PL, count the incoming “LIKES” edges and sort Posts by this value.	Makes suggestions about new posts in the same places to check, basing on users’ liked posts and ordering by number of likes

```
MATCH (u:User{username:$username})-[:LIKES]->(p:Post)-[:LOCATION]->(pl:Place)
WITH DISTINCT pl AS places, COLLECT(p) AS likedPosts
```

```
MATCH (:User)-[l:LIKES]->(sp:Post WHERE NOT(sp IN likedPosts))-[:LOCATION]->(places)
WITH DISTINCT sp AS suggestedPosts, COUNT(l) AS likeReceived
ORDER BY likeReceived DESC
RETURN suggestedPosts.id, likeReceived, suggestedPosts.title
```

Java Equivalent Query



```
@Override
public List<PostPreview> getSuggestedPosts(User user, int maxHowMany) {
    Neo4jConnection neo4jConnection = Neo4jConnection.getObj();

    try (Session session = neo4jConnection.getDriver().session()) {
        return session.writeTransaction((TransactionWork<List<PostPreview>>) tx -> {
            Map<String,Object> params = new HashMap<>();
            params.put( "USER_ID", user.getId() );
            params.put("HOW_MANY", maxHowMany);
            String newLine = System.getProperty("line.separator");
            String query = String.join(newLine,
    ...elements: "MATCH (u:"+ User.NEO_USER_LABEL +"{"+User.NEO_KEY_ID+":$USER_ID})-[:"+ User.NEO_RELATION_LIKES +"]->(p:"+ Post.NEO_POST_LABEL +")-[:"
            + Post.NEO_RELATION_LOCATION +"]->(pl:"+ Place.NEO_PLACE_LABEL +") ",
            "WITH DISTINCT pl AS places, COLLECT(p) AS likedPosts ",
            "MATCH (:"+ User.NEO_USER_LABEL +")-[l:"+ User.NEO_RELATION_LIKES +"]->(sp:"+ Post.NEO_POST_LABEL +" WHERE NOT(sp IN likedPosts))-[:"+ Post.NEO_RELATION_LOCATION +"]->(places) ",
            "MATCH (author:"+ User.NEO_USER_LABEL +")-[:"+ Post.NEO_RELATION_AUTHOR +"]->(sp) ",
            "WITH DISTINCT sp AS suggestedPosts, COUNT(l) AS likeReceived, author.username AS authorUsername, author.id AS authorId ",
            "ORDER BY likeReceived DESC ",
            "LIMIT $HOW_MANY ",
            "RETURN suggestedPosts, authorUsername, authorId, likeReceived ");
        Result res = tx.run(query, params);
        List<PostPreview> suggestedPosts = new ArrayList<>();
        while(res.hasNext()){
            Record r = res.next();
            Value postValue = r.get("suggestedPosts");
            Value authorUsername = r.get("authorUsername");
            Value authorId = r.get("authorId");
            PostPreview postPreview = new PostPreview(postValue, authorId, authorUsername);
            suggestedPosts.add(postPreview);
        }
        return suggestedPosts;
    });
} catch (Neo4jException ne){
    System.out.println(ne.getMessage());
    return null;
}
```

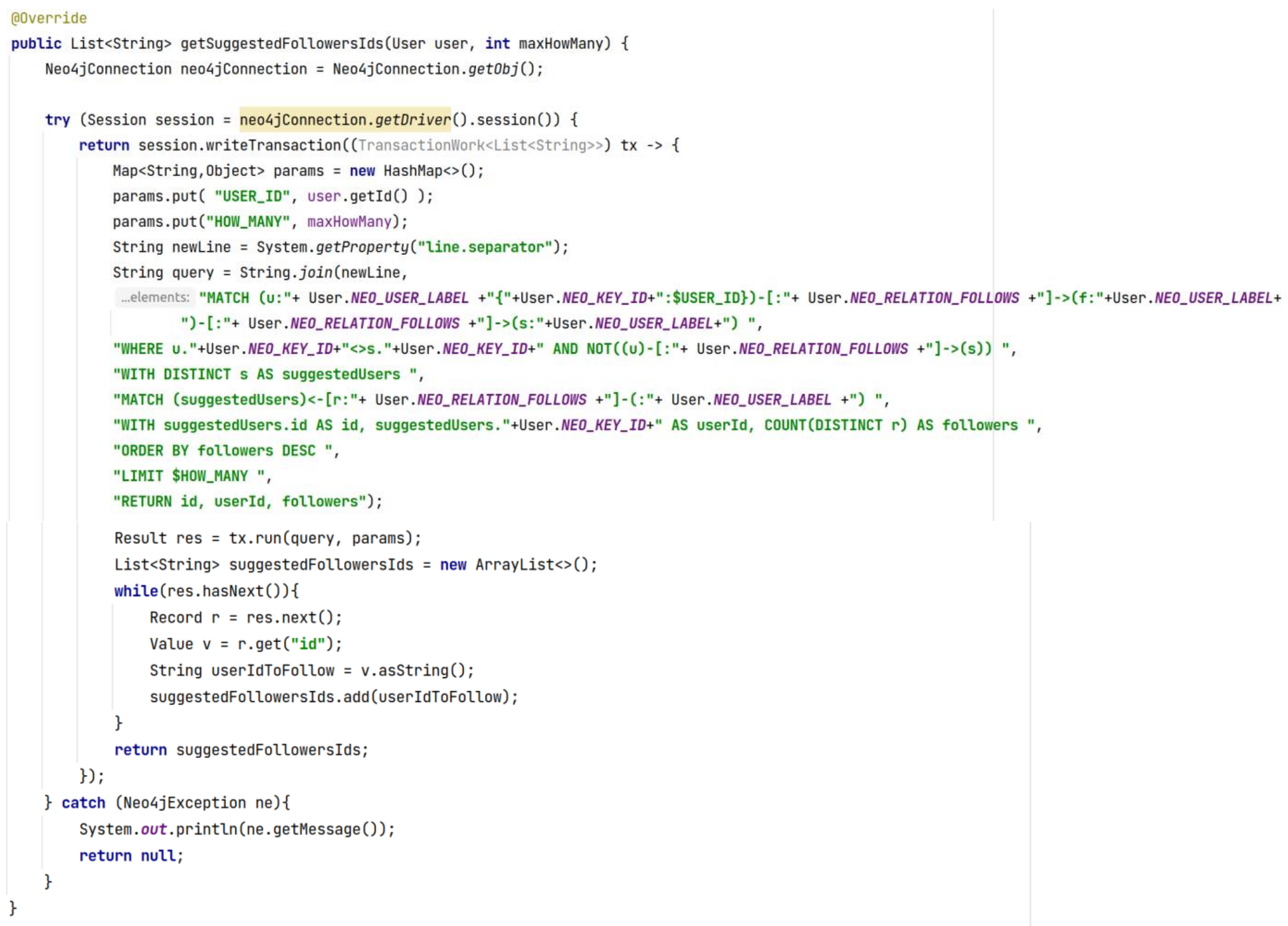
8.4.2.3 Suggested users to follow based on followings

Graphic-centric query	Domain query
Considering U as the Users with an incoming edge “FOLLOWS” from a given User, select U2 that are the Users with an incoming edge “FOLLOWS” starting from U and that are not already followed by the specified User. Then count incoming edges in U2 nodes and sort these Users by this value.	Who are the Users with the highest number of followers, between the ones followed by the followings of a certain User and that the latter doesn’t follow?

```
MATCH (u:User{username:$username})-[:FOLLOWS]->(f:User)-[:FOLLOWS]->(s:User)
WHERE u.username<>s.username AND NOT((u)-[:FOLLOWS]->(s))
WITH DISTINCT s AS suggestedUsers
MATCH (suggestedUsers)<-[r:FOLLOWS]-(:User)
WITH suggestedUsers.id AS id, suggestedUsers.username AS username, COUNT(DISTINCT r) AS
followers
ORDER BY followers DESC
```

```
LIMIT 20
RETURN id, username, followers
```

Java Equivalent Query



```java
@Override
public List<String> getSuggestedFollowersIds(User user, int maxHowMany) {
    Neo4jConnection neo4jConnection = Neo4jConnection.getObj();

    try (Session session = neo4jConnection.getDriver().session()) {
        return session.writeTransaction((TransactionWork<List<String>>) tx -> {
            Map<String,Object> params = new HashMap<>();
            params.put( "USER_ID", user.getId() );
            params.put("HOW_MANY", maxHowMany);
            String newLine = System.getProperty("line.separator");
            String query = String.join(newLine,
             ...elements: "MATCH (u:"+ User.NEO_USER_LABEL +"{"+User.NEO_KEY_ID+":$USER_ID})-[:"+ User.NEO_RELATION_FOLLOWS +"]->(f:"+User.NEO_USER_LABEL+
                    ")-[:"+ User.NEO_RELATION_FOLLOWS +"]->(s:"+User.NEO_USER_LABEL+") ",
            "WHERE u."+User.NEO_KEY_ID+"<>s."+User.NEO_KEY_ID+" AND NOT((u)-[:"+ User.NEO_RELATION_FOLLOWS +"]->(s)) ",
            "WITH DISTINCT s AS suggestedUsers ",
            "MATCH (suggestedUsers)<-[r:"+ User.NEO_RELATION_FOLLOWS +"]-(:"+ User.NEO_USER_LABEL +") ",
            "WITH suggestedUsers.id AS id, suggestedUsers."+User.NEO_KEY_ID+" AS userId, COUNT(DISTINCT r) AS followers ",
            "ORDER BY followers DESC ",
            "LIMIT $HOW_MANY ",
            "RETURN id, userId, followers");
            Result res = tx.run(query, params);
            List<String> suggestedFollowersIds = new ArrayList<>();
            while(res.hasNext()){
                Record r = res.next();
                Value v = r.get("id");
                String userIdToFollow = v.asString();
                suggestedFollowersIds.add(userIdToFollow);
            }
            return suggestedFollowersIds;
        });
    } catch (Neo4jException ne){
        System.out.println(ne.getMessage());
        return null;
    }
}
```