UNIVERSITÀ DI PISA

Department of Information Engineering

Artificial Intelligence and Data Engineering course

Large-Scale and Multi-Structure Database

SYP - Share Your Playlist

Application developed by

Lorenzo Massagli, Antonio Patimo, Fabio Buchignani

Academic Year 2021/2022

Contents

Abstract

SYP - Share Your Playlist is an application with the goal of connecting people together and share their playlists.

The application has been developed in multiple programming languages and frameworks: Angular (javascript framework) for the front-end, Java for the back-end and Python for web-scraping.

All the data have been stored in two different databases: Neo4j and MongoDB.

Neo4j has been used for the social part. The entities that it contains are users, comments, playlists and songs. The relationships managed by this databases are the follow, comment and like functions.

MongoDB has been used to create, store and retrieve all the information about users, songs and playlists entities.

Github project source.

Chapter 1

Introduction

SYP is an application with the goal of connecting together people interested to the music.
The application allows to register in the community in order to become a member and use all the functionalities, including the social ones. A user can create a playlist and share it, follow other users and their playlists, give a comment and a rating to the songs that he likes the most. Moreover, a user has the opportunity to search and browse for playlists, users and songs using their names and view detailed information about them.
Additional functionalities are given to the administrators that can extract some statistical information from users, playlists, songs and comments. Moreover, they can add, update and delete all the entities that he wants, to improve the application and remove inappropriate content.

Chapter 2

Application Design

In this chapter it will be described how the application is designed in terms of functional and non-functional requirements, classes, dataset, paradigm used and databases design.

2.1 Main Actors

The application interacts with an actor called "User". The user is a generalization of three types of users: Registered user, Unregistered user and Admin user.

- Unregistered user: An unregistered user can register to the application and use the "search" functionalities, like browse songs, but he can't use the social functions of the application;
- Registered user: A registered user can use all the basic functionalities of the application;
- Admin user: An admin user can use all the basic functionalities, view the statistical information and make important changes to the application.

2.2 Function and Non-Functional requirements

This section describes the functional and non-functional requirements that the application must provide.

2.2.1 Functional Requirements

The functional requirements are divided by user type and follows this hierarchy:
Admin user > Register user > Unregistered user.

2.2.1.1 Unregistered User

An unregistered user can register to the application and use the "search" functionalities:

- Register to the application;
- Look up for a user, playlist and song by name;
- Visualize information about a user, playlist and song;
- Visualize the most popular songs;
- Visualize the most followed playlists;
- Visualize the most followed users;
- Visualize the comments of a song.

2.2.1.2 Registered User

A registered user is allowed to use all the search functionalities as the unregistered users and moreover to:

- Login and Logout from the application;
- Follow/Unfollow a user;
- Like/Unlike a playlist;
- Create, modify and delete his playlists;
- Modify and Delete his account;
- Comment a song;

- Modify or delete his comments;
- See suggestions on playlists based on the users that he follows.

2.2.1.3 Admin User

An admin user is allowed to do all the registered user functionalities and moreover to:

- Change the privileges of a user;
- Add, delete and modify a song;
- Find the top k users that has created the highest number of playlists;
- Find the top k users that have added to their playlists the highest number of song of a specific artist;
- Find the users that follows at least k same playlists of the user provided in input;
- Find the k songs that has the highest number of comments;
- View how many songs a playlists contains on average;
- View in how many playlists a song is contained on average;
- View how many playlist a user creates on average;
- View how many followers has a playlist on average;
- View how many comments has a song on average;
- View how many followers has a user on average;
- View how many comments a user writes on average.

2.2.2 Non-Functional requirements

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

- The application must provide a great quality of service (QoS) in terms of high availability, low latency and tolerance to failure.
- The user's usernames must be unique.
- The application shall be user-friendly, since a user shall interact with an intuitive graphical interface.
- The code shall be readable and easy to maintain.

2.2.3 Use Cases diagram

Based on the functional requirements, in figure 2.1 is represented the use cases diagram.

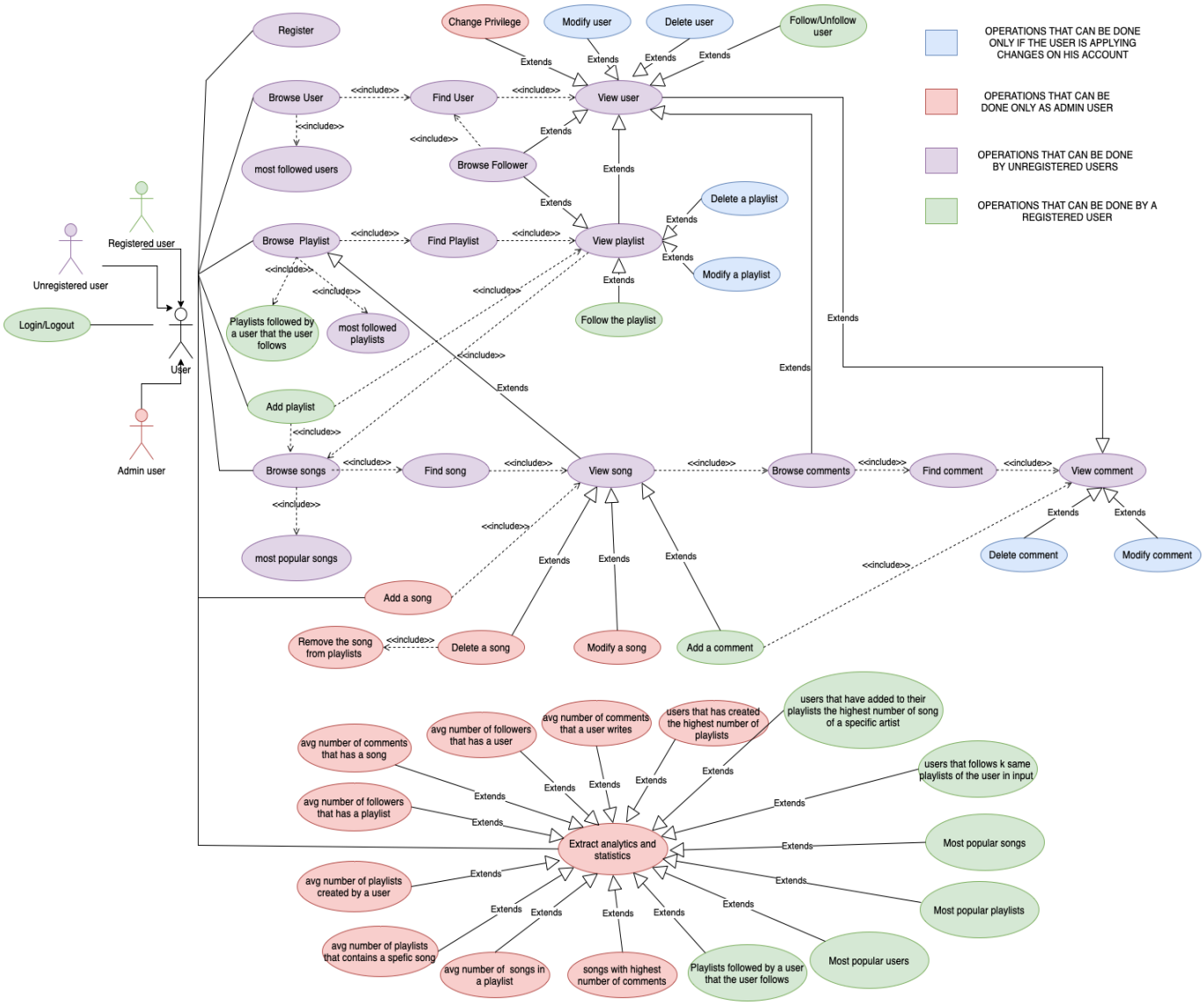

Figure 2.1: Use cases diagram

2.3 Class analysis

Based on the software requirements and actors the following analysis classes were identified in the application.

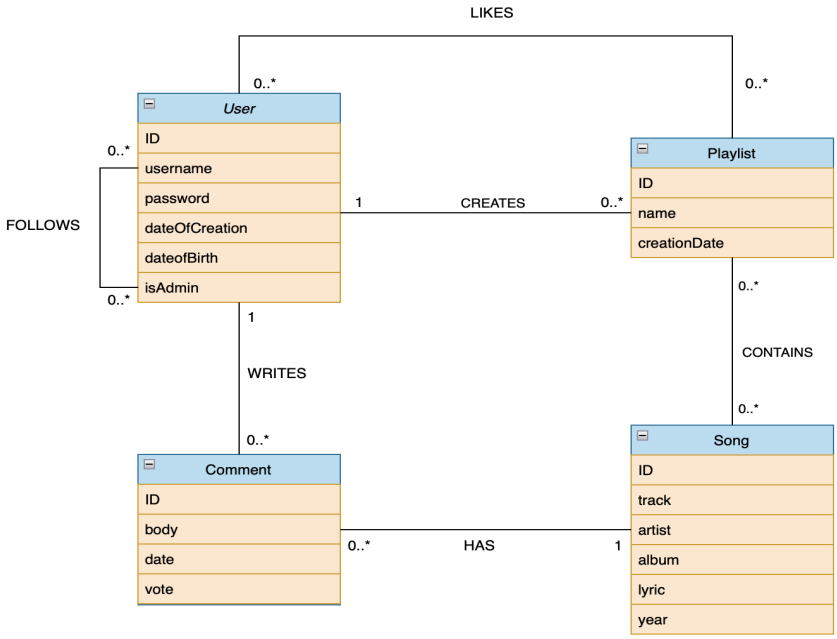

Figure 2.2: Class diagram

2.3.1 Classes definitions

User class		
A user is able to use all the basic function of the application and interact with other users. A user is an Admin user if the attribute isAdmin is true. An admin user can use all the functionalities.		
attribute	type	description
id	String	Unique user identifier.
username	String	A unique string identifying the user, which is also used by them to access the application.
password	String	The password used by a user to access the application.
dateOfCreation	Date	Represents user's account creation date.
dateOfBirth	Date	Represents user's birth date.
isAdmin	Boolean	Represents if a user is an admin.

Playlist class		
A playlist is a set of songs that is created by a user and shared among the application. It can be also liked by a user.		
attribute	type	description
id	String	Unique playlist identifier.
name	String	Name of the playlist.
creationDate	Date	Represents playlist's creation date.

Song class		
A song whose information is offered by the application. It can be added to a playlist, get a comment and a rate.		
attribute	type	description
id	String	Unique song identifier.
track	String	Song's name.
artist	String	Song's artist name.
year	String	Song's release year.
lyric	String	Song's text.
album	String	Song's album name.

Comment class		
A comment is a review refereed to a song, wrote by a user. It has a body and a vote.		
attribute	type	description
id	Long	Unique comment identifier.
body	String	Body of the comment, it contains the words wrote by a user about the song.
date	Date	Date when the comment has been wrote.
vote	Int	Vote associated to the comment given to the song.

2.4 Dataset

In order to simulate a real large scale database, we had to combine the information of a Kaggle dataset with information scraped on Genius site and random generated.

We have used the spotify playlist dataset to get the playlists created by the users and the song's names and artist. We have integrated those information performing scraping operation on the songs in the Genius site obtaining song's album, lyric, release year and comments. We also have used the comment's usernames to associate the user ids in the playlist dataset to a real username.

In addition, we have randomly generated the missing information like the user's birth date, account creation date, password and the comment's vote. The scraping has been performed with python programming language. We decided to use only a little part of the spotify database (~50mb) adding to it the scraping part and the data generated to have a simulate a large scale database (~220MB mongoDB and ~80MB neo4J).

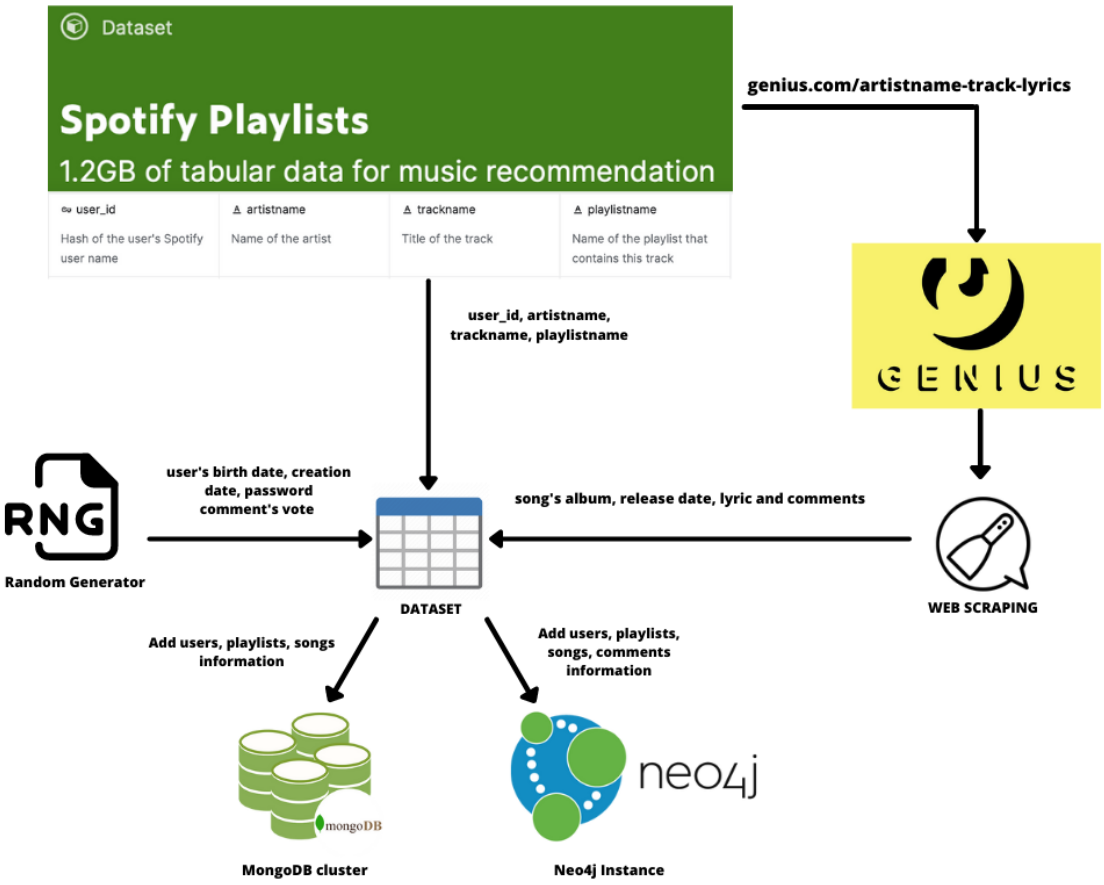

Figure 2.3: Dataset and Databases build process

2.5 Software Architecture

In terms of application architecture, it has been built in according to Client-Server paradigm.

2.5.1 Client Side

The client side uses a front-end module consisting in a graphical user interface built with Angular, allowing users to interact with the application and all its functionalities.

2.5.2 Server Side

The server side is composed with three virtual machines made available to us by the University of Pisa, which were used to host a MongoDB cluster and a single instance of the Neo4j database (hosted on the VM 172.16.4.60). According to the CAP theorem, we decided to design our application's databases with the AP approach (Availability - Partition tolerance). When a partition occurs, other nodes remains available but might return an older version of data. We needed to setup a replica set with write concern and read preference. The server side is also composed by the Java Spring part that functions as backend of the application and interacts with the databases using the mongoDB and neo4J connectors.

2.5.2.1 MongoDB replica set

In order to ensure the availability and partition tolerance of the system a cluster as been set up to build a replica set. The replica set is composed by a primary server, which takes the client requests, and two secondary servers, which copies that primary's data.

Replica Configuration The configuration is shown in the figure 2.4.
The VM 172.16.4.62 has the highest priority and is elected as primary server unless it's not available. In case the primary server goes down, a new one will be elected as primary server. Since we have assigned a priority on each replicas, we can control the behaviour of the election algorithm to predict which of the two secondaries will be promoted. In our case, when the primary is not available, the VM 172.16.4.61 will be marked as the new primary, as shown in the figure 2.5.

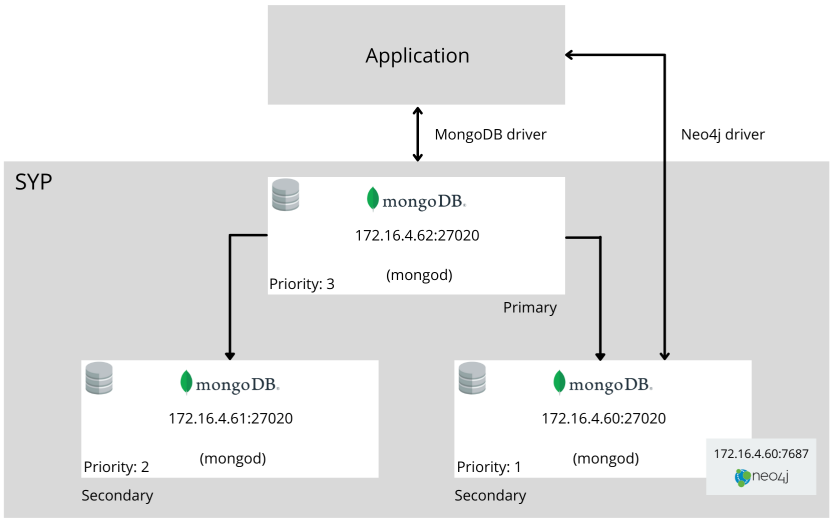

Figure 2.4: Server side architecture

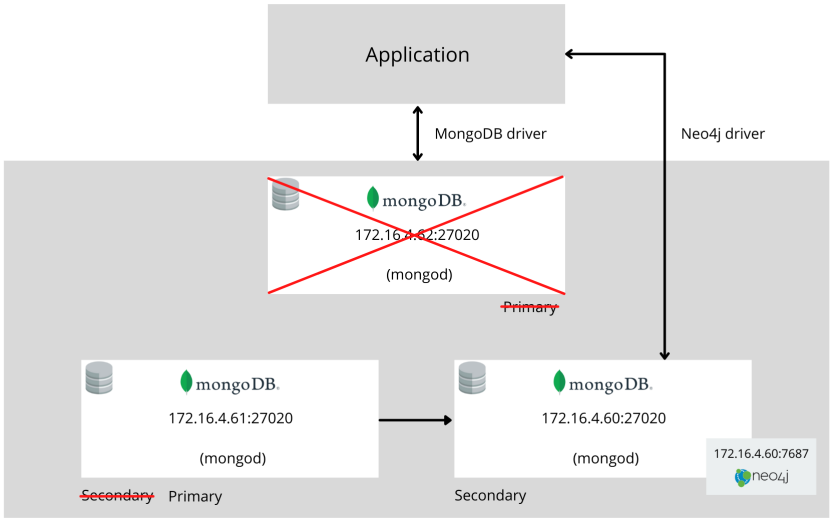

Figure 2.5: New primary after primary server down.

Write Concern We decided to setup the parameter $w = 2$ for the write concern to ensure the eventual consistency of the application. This choice is due to the fact that the application is read-heavy then we decided to reduce the latency of the reads ensuring the consistency on at least two databases instead of three. When a write occurs the application waits that two writes happens with success before returing to the user, the third could be done later.

Read Preference We decided to setup the parameter $readPreference = nearest$. That means that the application reads from the member of the replica set with the least network latency to have the fastest response. This choice, like the Write Concern, has been done to reduce the application's latency and ensure high availability.

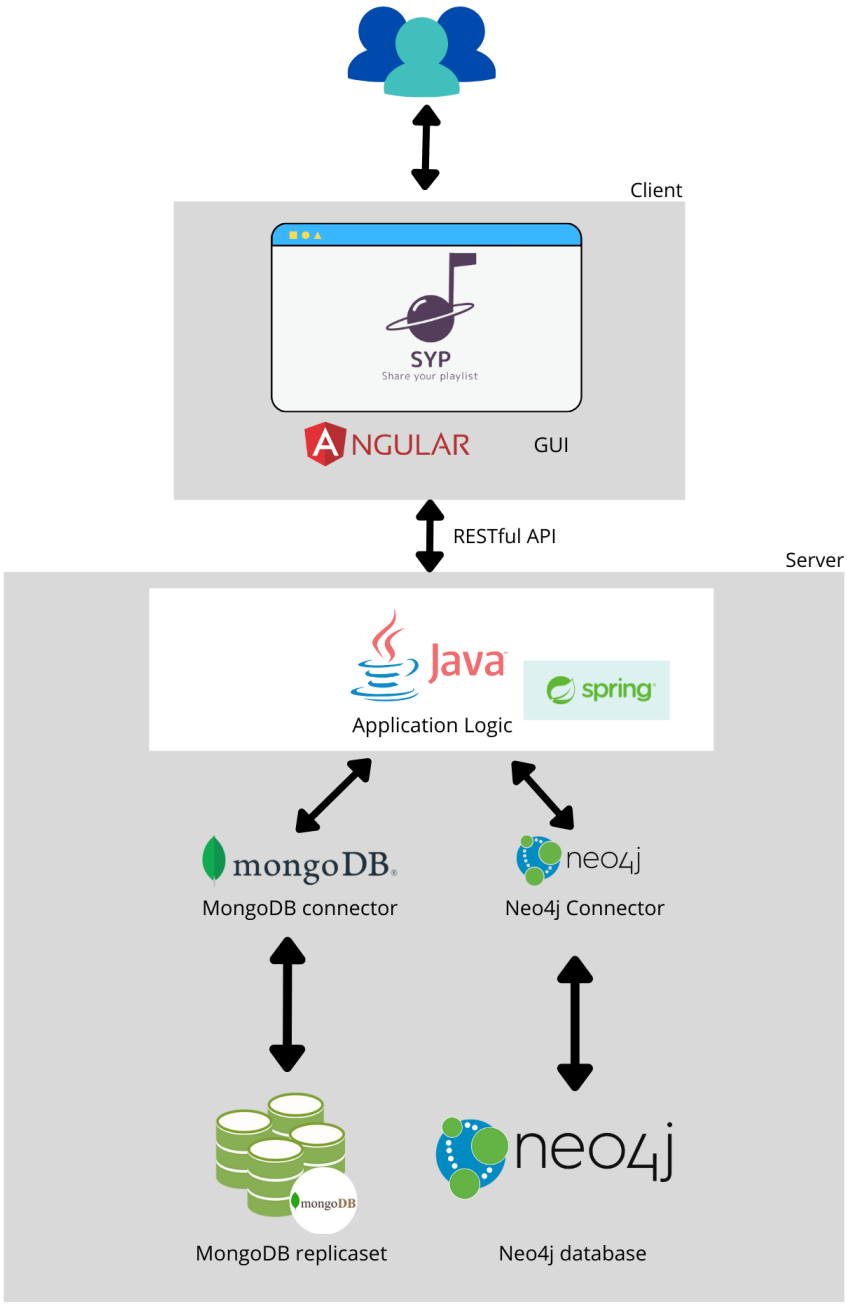

Figure 2.6: Software and Hardware architecture.

2.6 MongoDB and Neo4j Design

In this section will be described how MongoDB and Neo4j are organized in terms of collections and documents for MongoDB, nodes and relationships for Neo4j.

2.6.1 MongoDB organization and structure

MongoDB cluster stores three different collections:

- User collection: ~13.000 user documents;
- Playlist collection: ~65.000 playlist documents;
- Song collection: ~180.000 song documents.

The user collection contains the user's documents which each one stores information about a user.

```
{
   "_id":{"$oid":"61cc6cbb3bd8fb88c6628b8b"},
   "username":"DeeCee",
   "createdPlaylists":[
      {
         "_id":{"$oid":"61cc6cb53bd8fb88c65ec87c"},
         "name":"HARD ROCK 2010"
      }
   ],
   "dateOfBirth":{"$date":"1952-08-29T00:00:00.000Z"},
   "dateOfCreation":{"$date":"2013-01-28T00:00:00.000Z"},
   "isAdmin":false,
   "password":"IbIBOgsxjypTyyvT"
}
```

Listing 2.1: Example of a user document in the users collection

The playlist collection contains the playlist's documents which each one stores information about a playlist.

```
{
   "_id":{"$oid":"61cc6cb53bd8fb88c65ec87c"},
   "creationDate":{"$date":"2014-05-03T00:00:00.000Z"},
   "creator":{
      "_id":{"$oid":"61cc6cbb3bd8fb88c6628b8b"},
      "username":"DeeCee"
   },
   "name":"HARD ROCK 2010",
   "songs":[{
         "_id":{"$oid":"61cc6cb83bd8fb88c65fc8aa"},
        "artist":"Crosby Stills Nash",
        "track":"Helplessly Hoping"}]
}
```

Listing 2.2: Example of a playlist document in the playlist collection

The song collection contains the song's documents which each one stores information about a song.

```
{
"_id":{"$oid":"61cc6cb83bd8fb88c65fc8ab"},
"track":"Chris",
"artist":"C418",
"album":"Minecraft - Volume Alpha",
"lyric":"This song is an instrumental",
"year":"2011",
"playlists":[{
      "_id":{"$oid":"61cc6cb53bd8fb88c65ec87d"},
      "name":"C418"
   }]
}
```

Listing 2.3: Example of a song document in the song collection

We decided to embed playlist in user documents, user and songs in playlist document and playlists in song document because we need to retrieve these information when a user look for one of these entities and we don't want to do joins every time and increase the application latency.

2.6.2 Neo4j organization and structure

Neo4j stores four different type of nodes:

- Users nodes: Representing the users registered within the application, having as attributes id and username which are the same as MongoDB collection;
- Playlists nodes: Representing a set of songs. Playlists have as attributes id and name which are the same as MongoDB collection;
- Songs nodes: Representing a song. Songs have as attributes id, track and artist which are the same as MongoDB collection;
- Comment nodes: Representing all the comments written by users. Comments have as attribute id, body, vote and date.

Relationships:

- User ->: FOLLOWS -> User, which represent a register user that follows another register user in the application. This relationship is created when a user follows another user. This relationship has no attributes.
- User ->: LIKES -> Playlist, which represent a register user that likes a playlist in the application. This relationship is created when a user give a like to a playlist. This relationship has no attributes.
- User ->: WRITES -> Comment, which represent a register user that writes a comment in the application. This relationship is created when a user write a comment. This relationship has no attributes.
- Comment ->: RELATED -> Song, which represent a comment related to a song. This relationship is created when a user write a comment on a specific song. This relationship has no attributes.

In the Neo4j database are stored ~300.000 nodes and ~1.000.000 relationships.

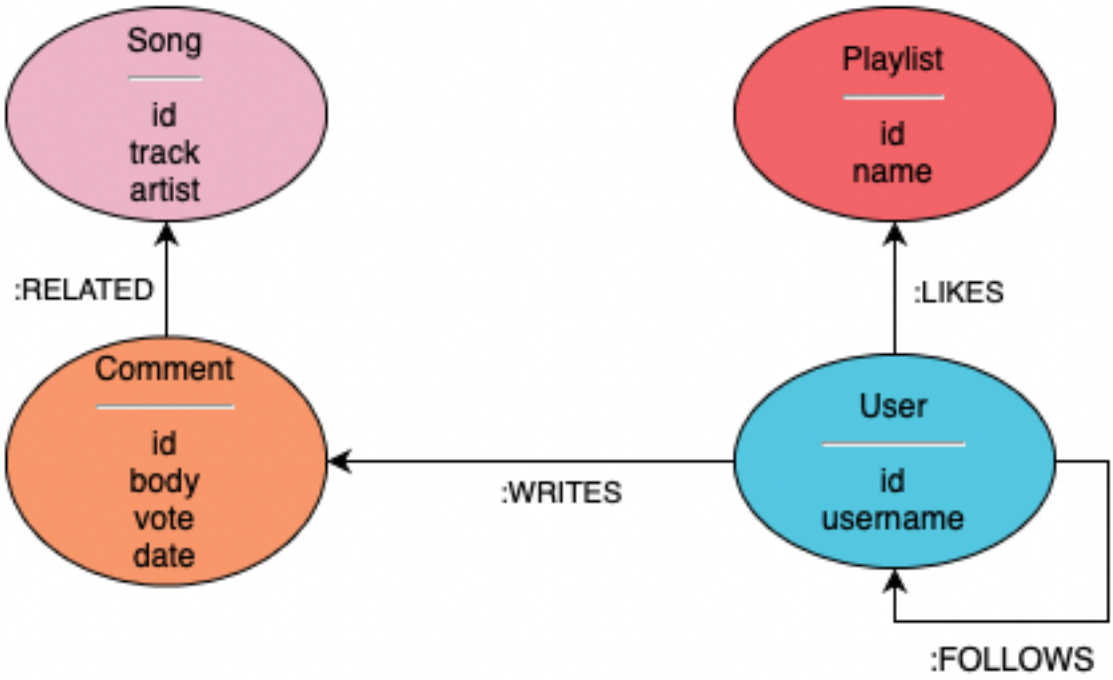

Figure 2.7: GraphDB scheme

Chapter 3

Application implementation

In this chapter a summary of the software architecture will be made and will be described the implementation details of the application.

3.1 Software architecture

As described in the section 2.5, the application architecture is structured according to the client-server paradigm. The client has been divided in two layers: a front-end layer, that contains the presentation logic and a back-end layer, which manages the connection with the server. The server is composed of three virtual machines hosted by the University of Pisa, on which a cluster of three MongoDB instances has been deployed. A single instance of Neo4j ha been deployed on one of the virtual machines.

3.1.1 Client-side implementation

The client of the application has been developed using Angular framework and development platform, in order to create a single page application. Angular is a component-based framework where every part of the application is a component. Each component consist of:

- An HTML template that declares what renders on the page;
- A Typescript class that defines the behavior of the component;
- A CSS selector that defines the component style.

Angular implements the dependency injection design pattern and it is used for injecting in the components the services used to retrieve data from the server. These data are retrieved using REST API, an

application programming interface that permits the exchange of the data from the client to the server over the HTTP protocol in a JSON format. The angular project structure has been divided in two main parts:

- The main page of the application;
- The details pages of the entities of the application.

Here there is the project structure of the Angular application:

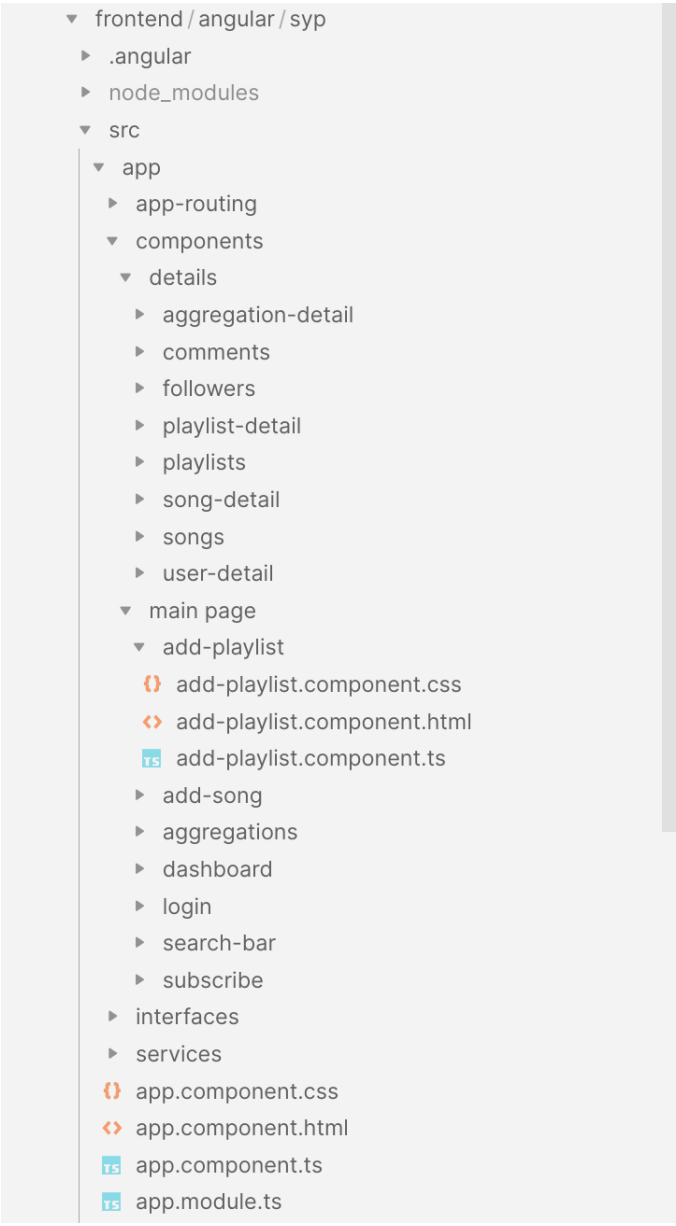

Figure 3.1: Angular project structure

3.1.2 Server-side implementation

The server-side of the application has been developed using Java and Spring framework. It builds a Java Web API that can be exploited by the front-end, communicating thus with the http protocol and exchanging information through JSON documents. The application is divided in different packages, organized according to the suggested structure given in Spring documentation:

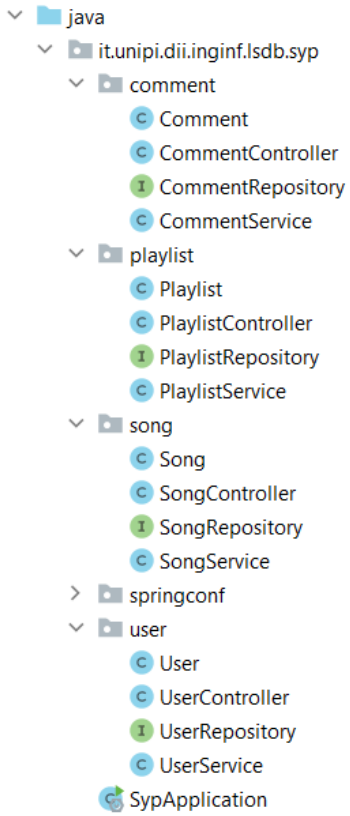

Figure 3.2: Package structure

The main class, SypBackendApplication, is located in the root package. The springConf package contains a class used to configure some aspects of Spring. Then there is one package for each entity of the application, that contains the entity Java representation, the controller that defines the operations on that entity, and the service which implements the business logic. The connection with MongoDB is handled by the Spring class MongoTemplate, injected in the Service Class, while the connection with Neo4J is handled through the Repository interface injected in the Service Class as well.

```java
@Node("Playlist")
@Document(collection="playlists")
public class Playlist {
    @Id
    @org.springframework.data.neo4j.core.schema.Id
    @Property("id")
    private String identifier;
    private String name;
    private Date creationDate;
    private User creator;
    private List<Song> songs;

    private Integer numberOfFollowers;

    //constructor, getters and setters
```

Figure 3.3: Playlist class

Each entity class contains all the information needed for that entity. Each entity has been enriched with annotations in such a way that Spring Data MongoDB and Spring Data Neo4J drivers can map instances automatically. These classes are automatically serialized in JSON objects to send them to the front-end. Each attribute could be null (not all the information related to an object is always needed, or always available), but the Spring default serializer has been configured to consider only non-null attributes when sending these objects to the front-end.

Chapter 4

Queries Analysis and Implementation

In this chapter will be described which query have been identified from the use cases diagram and how have been implemented.

4.1 MongoDB queries

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

Read operations		
Operation	Expected Frequency	Cost
Find a playlist, song, user	High	Average (Multiple reads)
Retrieve information on a playlist, song, user	High	Low (1 read)
View how many songs has a playlist on average	Average	High (Aggregation)
View in how many playlists a song is contained on average	Average	High (Aggregation)
View how many playlist are created by a user on average	Average	High (Aggregation)
View the top k users that has created the highest number of playlists	Average	High (Aggregation)
Find the top k users that have added to their playlists the highest number of songs of a specific artist	Average	High (Aggregation)
Find the top k most popular songs	High	High (Aggregation)

Write operations		
Operation	Expected Frequency	Cost
Insert a playlist	High	Average (add document)
Add a song	Low	Average (add document)
Add a user	High	Average (add document)
Delete a playlist	Average	Average (remove a document)
Delete a song	Low	Average (remove a document)
Delete a user	Low	Average (remove a document)
Modify a playlist	High	Average (Find playlist and attribute write)
Modify a song	Low	Average (Find song and attribute write)
Modify a user	Low	Average (Find user and attribute write)

As we can see from the table, the queries are predominantly read intensive, while write operations for their expected frequency and cost, carry a much lesser impact on the overall performance.

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

Operation	MongoDB Implementation
Read a playlist	`db.playlists.find({ "_id" : { "$oid" : "61cc6cb53bd8fb88c65f5014"} })`

Retrieve how many songs has a playlist in average	``` db.playlists.aggregate([{ "$project" : { "_id" : 1, "name" : 1, "songs" : { "$ifNull" : ["$songs", []]}}}, { "$project" : { "_id" : 1, "name" : 1, "numberOfSongs" : { "$size" : ["$songs"]}}}, { "$group" : { "_id" : null, "avgNumberOfSongs" : { "$avg" : "$numberOfSongs"}}}]) ```
Retrieve how many playlist are created by a user in average	``` db.users.aggregate([{ "$project" : { "_id" : 1, "username" : 1, "createdPlaylists" : { "$ifNull" : ["$createdPlaylists", []]}}}, { "$project" : { "_id" : 1, "username" : 1, "numberOfPlaylists" : { "$size" : ["$createdPlaylists"]}}}, { "$group" : { "_id" : null, "avgNumberOfPlaylistsCreated" : { "$avg" : "$numberOfPlaylists"}}}]) ```

Find the top k users that has created the highest number of playlists	`db.users.aggregate([` `{ "$match" : { "createdPlaylists" : { "$exists" : true}}},` `{ "$project" : { "_id" : 1, "username" : 1,` `"numberOfPlaylists" : { "$size" :` `["$createdPlaylists"]}}},` `{ "$match" : { "numberOfPlaylists" : { "$gt" : 0}}},` `{ "$sort" : { "numberOfPlaylists" : -1}},` `{ "$limit" : 5}` `])`
Find the top k users that have added to them playlist the highest number of songs of a specific artist	`db.playlists.aggregate([` `{ "$match" : { "songs" : { "$exists" : true}}},` `{ "$project" : { "_id" : 1, "creator" : 1, "songs" : {` `"$filter" : { "input" : "$songs", "as" : "songs", "cond"` `: { "$eq" : ["$$songs.artist", "C418"]}}}}},` `{ "$project" : { "_id" : 1, "creator" : 1, "numberOfSongs"` `: { "$size" : ["$songs"]}}},` `{ "$group" : { "_id" : "$creator._id", "username" : {` `"$first" : "$creator.username"}, "numberOfSongs" : {` `"$sum" : "$numberOfSongs"}}},` `{ "$sort" : { "numberOfSongs" : -1}},` `{ "$limit" : 5}` `])`

Find the top k most popular songs	`db.songs.aggregate([` `{ "$match" : { "playlists" : { "$exists" : true}}},` `{ "$project" : { "_id" : 1, "track" : 1,` `"numberOfPlaylists" : { "$size" : ["$playlists"]}}},` `{ "$match" : { "numberOfPlaylists" : { "$gt" : 0}}},` `{ "$sort" : { "numberOfPlaylists" : -1}},` `{ "$limit" : 5}` `])`

This is the implementation of some of the queries in java:

```java
public Double getAverageSongsContained() {
    ProjectionOperation projectEmptyArrayInPlaylistsWithoutSongs = project("_id", "name").and("songs")
            .applyCondition(ifNull("songs").then(new ArrayList<>()));
    ProjectionOperation getNumberOfSongs = project("_id", "name").and("songs").size().as("numberOfSongs");
    GroupOperation groupAllAndGetAverageSongs = group().avg("numberOfSongs").as("avgNumberOfSongs");

    Aggregation aggregation = newAggregation(projectEmptyArrayInPlaylistsWithoutSongs,
                                             getNumberOfSongs,
                                             groupAllAndGetAverageSongs);

    try{
        AggregationResults<Document> result = mongoTemplate.aggregate(
                aggregation, "playlists", Document.class);

        Document document = result.getUniqueMappedResult();

        return document.getDouble("avgNumberOfSongs");
    } catch (Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }
}
```

Figure 4.1: Avg songs query in Java

```java
public List<User> getTopCreators(int numberToReturn) {
    MatchOperation filterUsersWithoutAttributeCreatedPlaylists = match(new Criteria("createdPlaylists").exists(true));
    ProjectionOperation getNumberOfCreatedPlaylists = project("_id", "username").and("createdPlaylists").size().as("numberOfPlaylists");
    MatchOperation matchUsersWithAtLeastOneCreatedPlaylist = match(new Criteria("numberOfPlaylists").gt(0));
    SortOperation sortByNumberOfCreatedPlaylists = sort(Sort.by(Sort.Direction.DESC, "numberOfPlaylists"));

    Aggregation aggregation = newAggregation(filterUsersWithoutAttributeCreatedPlaylists,
                                             getNumberOfCreatedPlaylists,
                                             matchUsersWithAtLeastOneCreatedPlaylist,
                                             sortByNumberOfCreatedPlaylists,
                                             limit(numberToReturn));

    try{
        AggregationResults<User> result = mongoTemplate.aggregate(
                aggregation, "users", User.class);
        return result.getMappedResults();
    } catch (Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }
}
```

Figure 4.2: Top creators query in Java

```java
public List<User> getUsersWithMostSongsOfASpecificArtist(int numberToReturn, String artist) {
    MatchOperation filterPlaylistsWithoutAttributeSongs = match(new Criteria("songs").exists(true));
    ProjectionOperation filterArraySongsBasedOnArtist = project("_id", "creator")
            .and(filter("songs").as("songs").by(
            ComparisonOperators.valueOf("songs.artist").equalToValue(artist))).as("songs");
    ProjectionOperation getNumberOfSongsPerPlaylist = project("_id", "creator").and("songs").size().as("numberOfSongs");
    GroupOperation groupByCreator = group("creator._id").first("creator.username").as("username").sum("numberOfSongs").as("numberOfSongs");
    SortOperation sortByNumberOfSongs = sort(Sort.by(Sort.Direction.DESC, "numberOfSongs"));

    Aggregation aggregation = newAggregation(filterPlaylistsWithoutAttributeSongs,
                                             filterArraySongsBasedOnArtist,
                                             getNumberOfSongsPerPlaylist,
                                             groupByCreator,
                                             sortByNumberOfSongs,
                                             limit(numberToReturn));

    try{
        AggregationResults<User> result = mongoTemplate.aggregate(
                aggregation, "playlists", User.class);

        return result.getMappedResults();
    } catch(Exception e) {
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }
}
```

Figure 4.3: Users that have added to them playlist the highest number of songs of a specific artist query in Java

4.2 Neo4j queries

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

Read operations		
Operation	Expected Frequency	Cost
Show all the comments related to a song	High	Average (Multiple reads)
Show all the followers of a user	Average	Average (Multiple reads)
Show all the playlists liked by a user	Average	Average (Multiple reads)
Show all the comments that a user have wrote	Low	Average (Multiple reads)
Retrieve how many followers has a playlist on average	Average	Average
Retrieve how many comments has a song on average	Average	Average
Retrieve how many followers has a user on average	Average	Average
Retrieve how many comments a user writes on average	Average	Average
Find the most k followed users	High	High
Find the most k followed playlists	High	High
Find the users that follows at least k same playlists of the user provided in input	Average	High
Find the k songs that has the highest number of comments	Average	High
Find suggested playlists for a user based on the user that he follows	High	High

Write operations		
Operation	Expected Frequency	Cost
Insert a playlist	High	Average (create 1 node)
Add a song	Low	Average (create 1 node)
Add a user	High	Average (create 1 node)
Add a comment related to a song	High	High (create a new node and two relationships)
Delete a playlist	Low	Average (delete multiple relationships, remove 1 node)

Delete a song	Low	Average (delete multiple relationships, remove 1 node)
Delete a user	Low	Average (delete multiple relationships, remove 1 node)
Delete a comment	Low	Average (delete multiple relationships, remove 1 node)
Modify a playlist	Low	Low (multiple attributes modifications)
Modify a song	Low	Low (multiple attributes modifications)
Modify a user	Low	Low (multiple attributes modifications)
Modify a comment	Low	Low (multiple attributes modifications)
Follow/Unfollow another user	Average	Low (create/delete 1 relationship)
Like/Unlike a playlist	Average	Low (create/delete 1 relationship)

As we can see from the table, the queries are predominantly read intensive, while write operations for their expected frequency and cost, carry a much lesser impact on the overall performance.

Queries implementation A selection of implementations of the queries involving the Neo4J database is presented below:

Operation	MongoDB Implementation
Insert an user	`CREATE (n:User {id: $id, name: $name})`
Follow another user	`MATCH (n:User {id = $followerId}), (n2:User {id = $followedId}) CREATE (n)-[:FOLLOWS]->(n2)`
Unfollow another user	`MATCH (n:User {id: $followerId})-[r:FOLLOWS]->(n2:User {id: $followedId}) DELETE r`
Show all the comments related to a song and relative users	`MATCH (s:Song {id:$id})<-[:RELATED]-(c:Comment)<-[w:WRITE]-(u:User) RETURN c,u,w`
Find suggested playlists for a user	`MATCH (u1:User {identifier : $id})-[:FOLLOWS]->(u2:User)-[:LIKES]->(p2:Playlist) WHERE NOT (u1)-[:LIKES]->(p2) RETURN p2, u2`

Find the users that follows at least k same playlists of the user provided in input	``` MATCH path = ((n:User {name: $username})-[:LIKES]->(p)<-[:LIKES]-(users)) WHERE NOT (n.id = users.id) WITH DISTINCT users as possibleUsers, count(path) as numberOfSharedPlaylist WHERE numberOfSharedPlaylist >= $numberOfPlaylist RETURN possibleUsers ORDER BY numberOfSharedPlaylist DESC ```
Find the k songs that has the highest number of comments	``` MATCH (song)<-[:RELATED]-(comment) RETURN song, COUNT(comment) as numberOfComments ORDER BY numberOfComments DESC LIMIT $number ```
Retrieve how many followers has a playlist on average	``` MATCH (n:Playlist) WITH COUNT(n) as NumberOfPlaylists MATCH ()-[f:LIKES]->() RETURN 1.0*COUNT(f)/NumberOfPlaylists ```

4.3 CRUD operations implementation

In this section will be described the implementation of the CRUD operations for the databases.

- Insert operation:

```
//@Transactional
Playlist savePlaylist(Playlist newPlaylist){
    Playlist savedPlaylist = null;
    try {
        savedPlaylist = mongoTemplate.insert(newPlaylist);
    } catch (Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }

    try{
        playlistRepository.insert(savedPlaylist.getIdentifier(), savedPlaylist.getName());
    } catch (Exception e){
        e.printStackTrace();
        Query findPlaylistById = new Query(Criteria.where("_id").is(savedPlaylist.getIdentifier()));
        mongoTemplate.remove(findPlaylistById, Playlist.class);
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }

    //manage redundancy
    Playlist embeddedPlaylistInfo = new Playlist(savedPlaylist.getIdentifier(), savedPlaylist.getName(),
                                                 null, null, null, null);

    //update creator's createdPlaylists array
    Query findCreator = new Query(Criteria.where("_id").is(savedPlaylist.getCreator().getIdentifier()));
    Update updateCreator = new Update().push("createdPlaylists", embeddedPlaylistInfo);
    mongoTemplate.updateFirst(findCreator, updateCreator, User.class);

    //update contained songs' playlists array
    List<String> songsIdentifiers = getIdentifiersFromPlaylist(savedPlaylist);

    if(songsIdentifiers != null){
        Query findSongs = new Query(Criteria.where("_id").in(songsIdentifiers));
        Update updateSongs = new Update().push("playlists", embeddedPlaylistInfo);
        mongoTemplate.updateMulti(findSongs, updateSongs, Song.class);
    }


    return savedPlaylist;
}
```

Figure 4.4: Insert operation for the playlists

- Read operation:

```
List<Playlist> getPlaylistsByRegex(String regex){
    try{
        Query findPlaylistsByRegex = new Query(Criteria.where("name").regex("^" + regex, "i"));
        return mongoTemplate.find(findPlaylistsByRegex, Playlist.class);
    } catch (Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }
}

Playlist getPlaylistById(String id){
    try{
        Query findPlaylistById = new Query(Criteria.where("_id").is(id));
        return mongoTemplate.findOne(findPlaylistById, Playlist.class);
    } catch(Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }
}
```

Figure 4.5: Read operation for the playlists

- Update operation:

```
//@Transactional
public Playlist updatePlaylist(Playlist oldPlaylist, Playlist newPlaylist) {
    Playlist savedPlaylist = null;
    try{
        savedPlaylist = mongoTemplate.save(newPlaylist);
    } catch (Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }

    //if name changed, redundancy must be managed
    if (!oldPlaylist.getName().equals(newPlaylist.getName())){
        try{
            playlistRepository.updateName(newPlaylist.getIdentifier(), newPlaylist.getName());
        } catch (Exception e){
            e.printStackTrace();
            mongoTemplate.save(oldPlaylist);
            throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
        }
        updateName(newPlaylist);
    }

    //manage redundancy on songs
    List<String> removedSongsIdentifiers = getSongsOnlyInFirstPlaylist(oldPlaylist, newPlaylist);

    List<String> insertedSongsIdentifiers = getSongsOnlyInFirstPlaylist(newPlaylist, oldPlaylist);

    if(removedSongsIdentifiers != null){
        Playlist embeddedOldPlaylistInfo = new Playlist(oldPlaylist.getIdentifier(), oldPlaylist.getName(),
                                                        null, null, null, null);

        Query findSongs = new Query(Criteria.where("_id").in(removedSongsIdentifiers));
        Update updateSongs = new Update().pull("playlists", embeddedOldPlaylistInfo);
        mongoTemplate.updateMulti(findSongs, updateSongs, Song.class);
    }

    if(insertedSongsIdentifiers != null){
        Playlist embeddedNewPlaylistInfo = new Playlist(newPlaylist.getIdentifier(), newPlaylist.getName(),
                                                        null, null, null, null);

        Query findSongs = new Query(Criteria.where("_id").in(insertedSongsIdentifiers));
        Update updateSongs = new Update().push("playlists", embeddedNewPlaylistInfo);
        mongoTemplate.updateMulti(findSongs, updateSongs, Song.class);
    }

    return savedPlaylist;
}
```

Figure 4.6: Update operation for the playlists

- Delete operation:

```
//@Transactional
public void deletePlaylist(String id) {
    Playlist deletedPlaylist = null;
    try{
        Query findPlaylistById = new Query(Criteria.where("_id").is(id));
        deletedPlaylist = mongoTemplate.findAndRemove(findPlaylistById, Playlist.class);
    } catch (Exception e){
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }

    //remove node from graph database
    try{
        playlistRepository.deletePlaylistByIdentifier(deletedPlaylist.getIdentifier());
    } catch (Exception e){
        e.printStackTrace();
        mongoTemplate.insert(deletedPlaylist);
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE);
    }

    //remove redundancy
    Playlist embeddedPlaylistInfo = new Playlist(deletedPlaylist.getIdentifier(), deletedPlaylist.getName(),
                                                 null, null, null, null);

    //eliminate createdPlaylists array entry regarding deleted playlist
    Query findCreator = new Query(Criteria.where("_id").is(deletedPlaylist.getCreator().getIdentifier()));
    Update updateCreator = new Update().pull("createdPlaylists", embeddedPlaylistInfo);
    mongoTemplate.updateFirst(findCreator, updateCreator, User.class);

    List<String> songsIdentifiers = getIdentifiersFromPlaylist(deletedPlaylist);

    if(songsIdentifiers != null){
        //eliminate playlists array entry regarding deleted playlist
        Query findSongs = new Query(Criteria.where("_id").in(songsIdentifiers));
        Update updateSongs = new Update().pull("playlists", embeddedPlaylistInfo);
        mongoTemplate.updateMulti(findSongs, updateSongs, Song.class);
    }
}
```

Figure 4.7: Delete operation for the playlists

4.4 Indexes Definition

4.4.1 MongoDB indexes

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. We analyzed the query's performance before and after defining the indexes, using the explain() function offered by MongoDB Compass. It's important to remember that MongoDB creates automatically the indexes for the document ids then we hadn't to create them manually for queries that filters by id.

Index Name	Index Type	Collection	Attributes
find-playlist-by-name	Single	Playlists	name
find-song-by-track	Single	Songs	track
find-user-by-username	Single	Users	username

4.4.1.1 Find Playlist

```
db.playlists.find(
   {name:"HARD ROCK 2010"}
)
```

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	65582	90
True	1	1	1	10

4.4.1.2 Find Song

```
db.songs.find(
   {track:"Writing On the Wall"}
)
```

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	180961	259
True	1	1	1	1

4.4.1.3 Find User

```
db.users.find(
   {username:"mriebs"}
)
```

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	13447	74
True	1	1	1	0

Chapter 5

Other Application Details

In this chapter will be described the cross-database consistency management, the sharding proposal structure and the Project Object Model (POM) file.

5.1 Cross-Database Consistency Management

As the application uses two database, we need to ensure consistency between them in the insert, update and delete operations. The Neo4j database only contains copies of the information in the MongoDB database, so we must first modify them in the MongoDB and then replicate it in the Neo4j database.

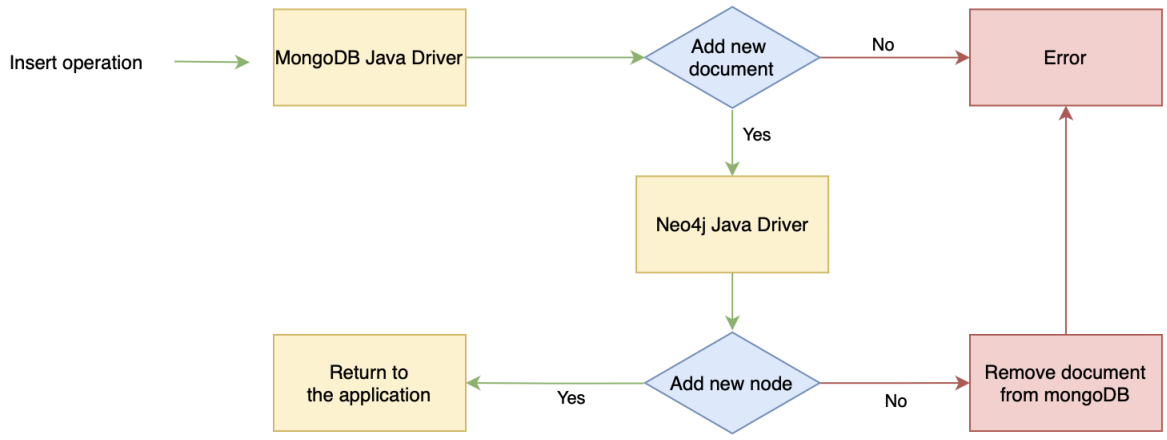

Figure 5.1: Insert operation cross-database consistency management

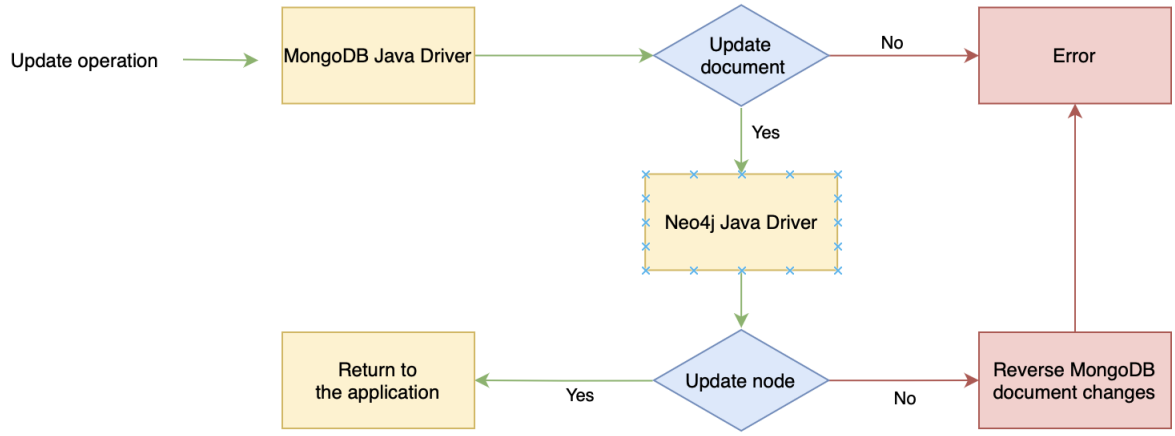

Figure 5.2: Update operation cross-database consistency management

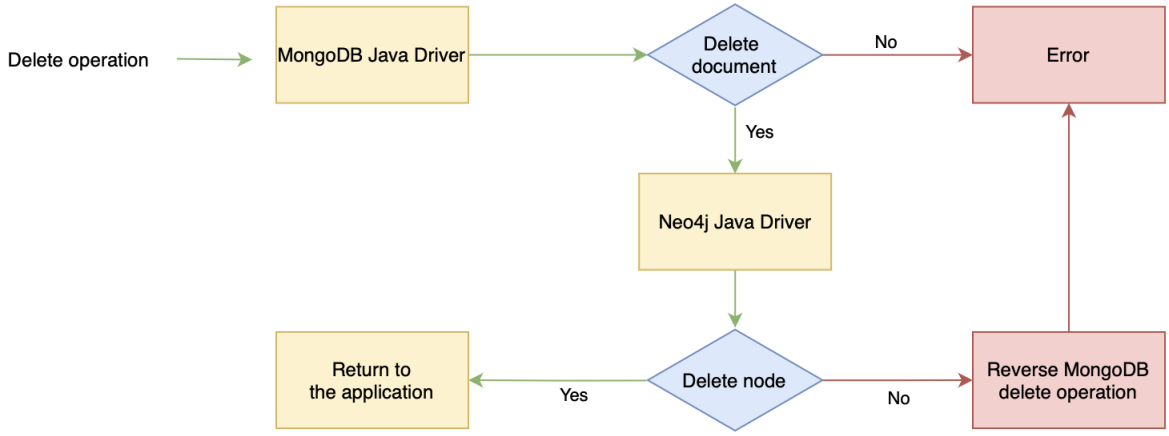

Figure 5.3: Delete operation cross-database consistency management

5.2 Sharding Proposal

A MongoDB sharded cluster consists of the following components:

- shard: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.
- mongos: The mongos acts as a query router, providing an interface between client applications and the sharded cluster.

- config server: Config server store metadata and configuration settings for the cluster.

The structure of our sharding proposal is shown below.

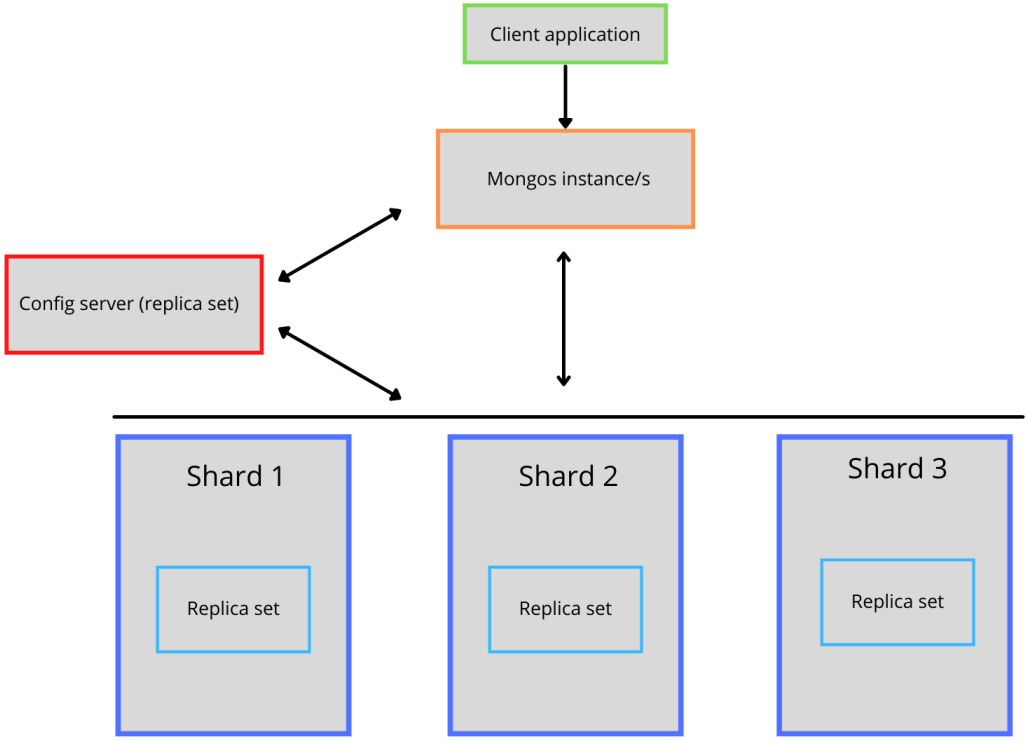

Figure 5.4: Sharding proposal

The mongos instance is in charge of receiving the read or write requests coming from the client application and route them to the shards. The config server is used to state and organize all data and components within the sharded cluster. We thought to use three shards in accord to the number of VM given to us and use a replica set in each one to ensure high availability and partition tolerance. We can decide to improve more the availability of the sharding system developing the config server as replica set to avoid all the issues about the single point of failure. We can also improve read and write workloads scaling horizontally across the cluster by adding more shards. To implement sharding we have select a shard key for each collection and a partitioning method.

5.2.1 Sharding key

A shard key is one or more fields, that exist in all documents in a collection, that is used to separate documents. We decided to use the following shard keys:

- Playlist collection: For this collection we decided to use the id field as shard key because it's the only atomic attribute in the collection and exists in all documents.
- User collection: For this collection we decided to use the username field as shard key because it's an atomic attribute in the collection and exists in all documents.
- Song collection: For this collection we decided to use the id field as shard key because it's the only atomic attribute in the collection and exists in all documents.

5.2.2 Partition algorithm

There are three main categories of partition algorithms, based on: range, partition or list. We propose the Hashed Strategy as a strategy. In this way the field chosen as the sharding key will be mapped through a hash function and the data will be balanced across the shards. An important observation is that a consistent hashing may be also used.

5.3 Project Object Model

The Java part of the application has been managed through Maven. The following is the POM used for the project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.2</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>it.unipi.dii.inginf.lsdb</groupId>
  <artifactId>syp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

```
    <name>syp</name>
    < description >SYP project for the course of Large scale and multistructered
        databases</ description >
    < properties >
        <java.version>11</java.version>
    </ properties >
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            < artifactId >spring-boot- starter -data-mongodb</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            < artifactId >spring-boot- starter -data-neo4j</ artifactId >
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            < artifactId >spring-boot- starter -web</ artifactId >
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            < artifactId >spring-boot- starter - test </ artifactId >
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                < artifactId >spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</ project >
```