



UNIVERSITÀ DI PISA

University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Large-Scale and Multi-Structured Databases

UniMusic

Academic year 2020-2021

Lorenzo Bianchi, Valerio Giannini, Alessio Serra

Github: https://github.com/ValeGian/LSMDB_UniMusic

Index

- Introduction and Requirements	4
- Functional Requirements.....	4
- Non Functional Requirements.....	5
- Specifications	5
- Actors and Use Case Diagram.....	5
- Analysis Class Diagram.....	6
- Architectural Design	7
- Software Architecture.....	7
- Client Side.....	7
- Server Side.....	7
- Architecture Diagram.....	8
- Dataset Organization and Database Population.....	9
- Selected NoSQL Databases.....	10
- MongoDB Design and Implementation	11
- User Collection.....	11
- Song Collections.....	12
- CRUD Operations.....	12
- Create.....	12
- Read.....	13
- Update.....	16
- Delete.....	17
- Queries Analysis and MongoDB Connection String.....	18
- Analytics and Statistics.....	18
- Artists with Most Hit Songs.....	18
- Top Rated Album per Decade.....	19
- Top Favourite Genres.....	20
- Top Favourite Artist per Age Range.....	21
- Index Analysis.....	23
- Index “artist”.....	23
- Index “album”.....	24

- Index “title”.....	24
- Neo4j Design and Implementation.....	26
- Nodes.....	26
- Relations.....	26
- Queries Implementation.....	27
- CRUD operation.....	27
- Create.....	27
- Read.....	27
- Update.....	28
- Delete	28
- “On-graph” queries.....	30
- Suggested User.....	31
- Suggested Playlist.....	31
- Hot songs	32
- Neo4j Indexes	32
- Other Implementation Details.....	33
- Shards Configuration and Non-Functional Requirements.....	33
- Cross-Database Consistency Management.....	33
- Application Package Structure.....	35
- User Manual.....	36
- Login and Registration.....	36
- Home page.....	37
- Song page.....	39
- User page.....	39
- Playlist page.....	40
- Admin.....	41

Introduction and Requirements

UniMusic is an application whose main intent is to provide *songs discovery and managing service*; songs that can be found on the platform are associated with links to official sources on the internet where users can listen to them (e.g., *official YouTube/Spotify source*).

The application provides general information about each song it stores (e.g., its name, genre, artist) and offers to users the possibility to search and browse for them, possibly applying specific filters to reduce the searching scope. Users are then allowed to express their thoughts on songs through a “like” system and eventually add them to their *favourites*.

In order to simplify and enhance users’ experience, they can organize songs in *playlists* they create or *follow* other people’s playlists. They are also allowed to *follow* other users in order to get *suggestions* about songs and playlists they may like.

Functional Requirements

Anonymous Users:

- ❖ Anonymous Users can **register** a Standard User account on the platform.
- ❖ Anonymous Users can **log in** as Standard User or as Admin.

Standard Users:

- ❖ *Standard Users* can **search** for *Songs*:
 - by title.
 - by artist.
 - by album.
- ❖ *Standard Users* can **manage** *Playlists*:
 - *Standard Users* can **create** a new playlist.
 - *Standard Users* can **add** and **remove** songs to their playlists.
 - *Standard Users* can **update** the name to their playlists.
 - *Standard Users* can **delete** their playlists.
 - *Standard Users* can **follow/unfollow** other User’s playlists.
- ❖ *Standard Users* can **search** other Users by username and **follow/unfollow** them.
- ❖ *Standard Users* can **browse** suggested playlists.
- ❖ *Standard Users* can **browse** suggested songs.
- ❖ *Standard Users* can **browse** hot songs.
- ❖ *Standard Users* can **like** *Songs*.
- ❖ *Standard Users* can **add** songs to their “favorite”.
- ❖ *Standard Users* can **log out**.

Admins:

- ❖ *Admins* can do all the operation of a *Standard Users*.
- ❖ *Admins* can **delete** a song.
- ❖ *Admins* can **promote** a *Standard User* to *Admin*.
- ❖ *Admins* can **access** to the *statistics* of the platform.

Non-Functional Requirements

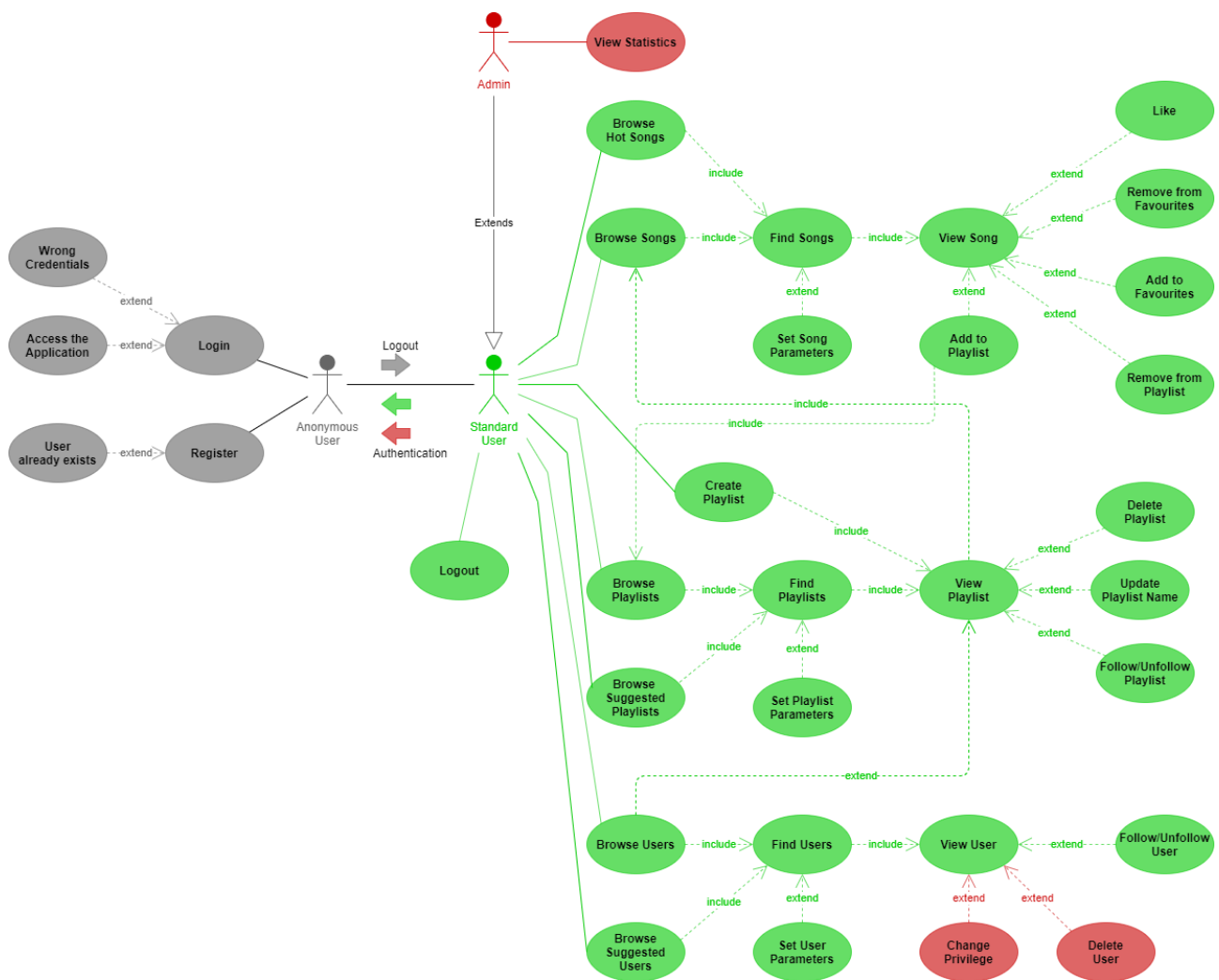
- ❖ The system needs to have *fast response times*, in order to make the application enjoyable for users.
- ❖ The content of the application should *be highly available* in any moment.
- ❖ The system needs to be *tolerant to data lost* and to *single point of failure*.
- ❖ The application needs to be *user friendly*, providing a *GUI*.

Specifications

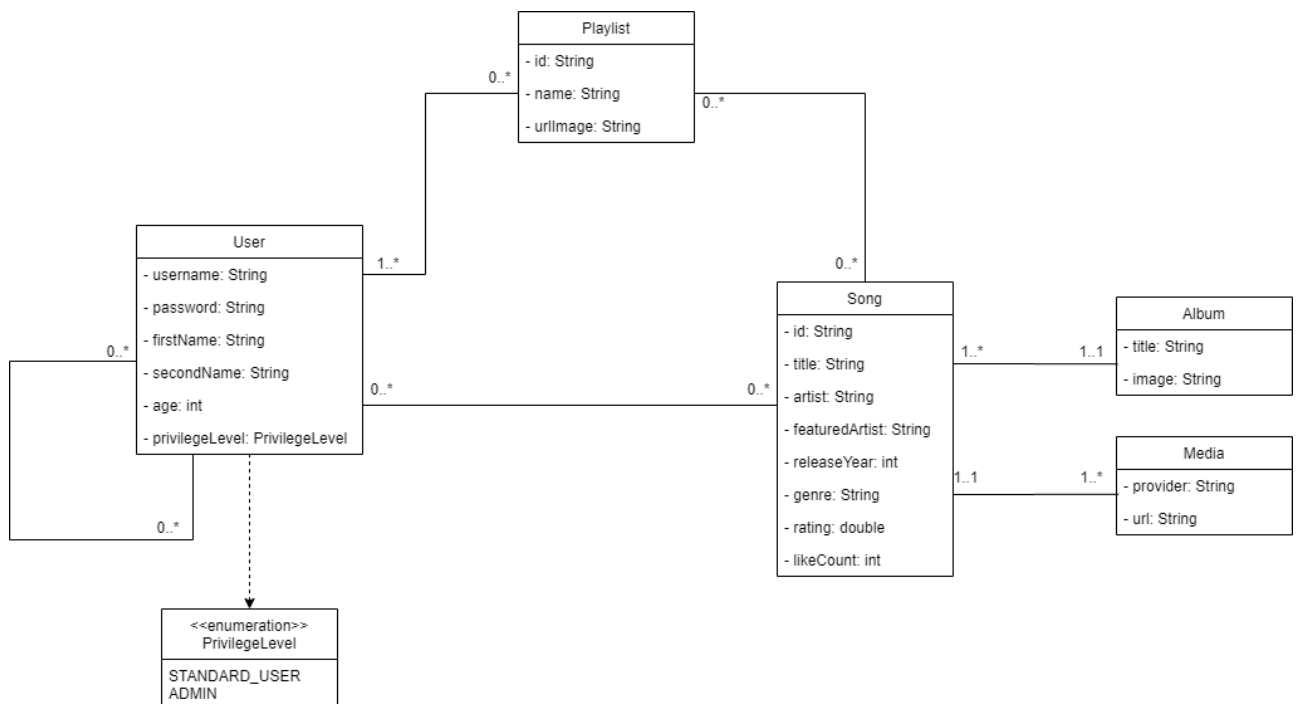
Actors and Use Case Diagram

Based on the software functional requirements, the application is meant to be used by 3 actors:

- ❖ **Anonymous Users:** can just log in or register through a *username/password system*.
- ❖ **Standard Users:** can benefit all the application's feature such as having a personalized own profile, follow other users, follow other users' playlists, visualize in their home page personalized content based on their followed or search songs filtering by title, artist or album.
Moreover, they can create new playlists to have always handy specific songs based on their mood.
- ❖ **Administrators:** they can access to all the functionalities of the application: they can delete or promote to admin a standard user, monitor all application's statics, such as total number of users, songs and playlists, and perform complex analytics to draw up rankings on artists, albums or genres.



Analysis Class Diagram



Architectural Design

Software Architecture

The application has been divided into *two-tiers* according to the *Client-Server* paradigm, exploiting **Java** as the core programming language.

Client Side

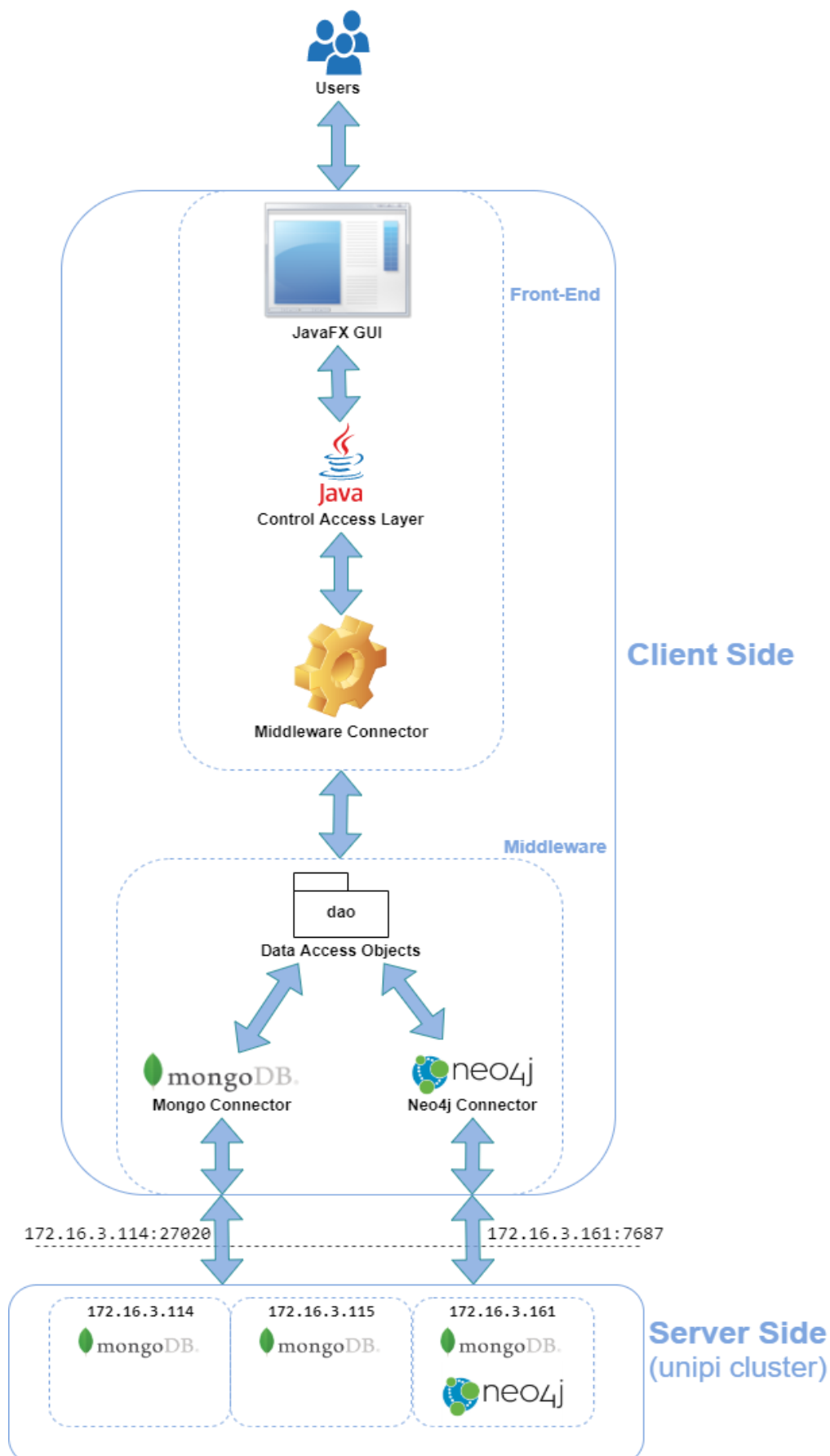
The client side is divided into:

- ❖ A *Front-End* module, in charge of
 - providing a **GUI** based on **JavaFX** for users to interact with the application.
 - communicating with the underlying middleware to retrieve information obtained by processing data stored on server side.
- ❖ A *Middleware* module, which communicates with the server side. It includes the logic used to communicate with the **MongoDB** cluster and the **Neo4j** database on the server, plus all the logic needed to handle and process data retrieved so that it can be used by the front end to present it to users.

Server Side

Server composed of 3 virtual machines which hosts a MongoDB cluster and a single instance of the Neo4j database.

Architecture Diagram



Dataset Organization and Database Population

Song information

The application dataset is mainly composed by songs' information, that are the core of our application.

They have been collected through multiple steps and multiple sources to respect the **variety constraint**.

Firstly, we have obtained almost all the information using services offered by GENIUS (www.genius.com), that permits to scrape their content through HTTP-based API (all steps can be found here www.docs.genius.com).

It returns a raw json document for each song requested; we then filter all information needed, in particular also 2 URLs and a URI that are used to scrape remaining information:

- ❖ using the Genius URL to the specific song page, we scrape the attribute “genre” that is not provided through API.
- ❖ using the Spotify URI, we have requested, through Spotify API, the song's popularity.
- ❖ using the YouTube URL to the official video of the song we scrape the number of Like and Dislike.

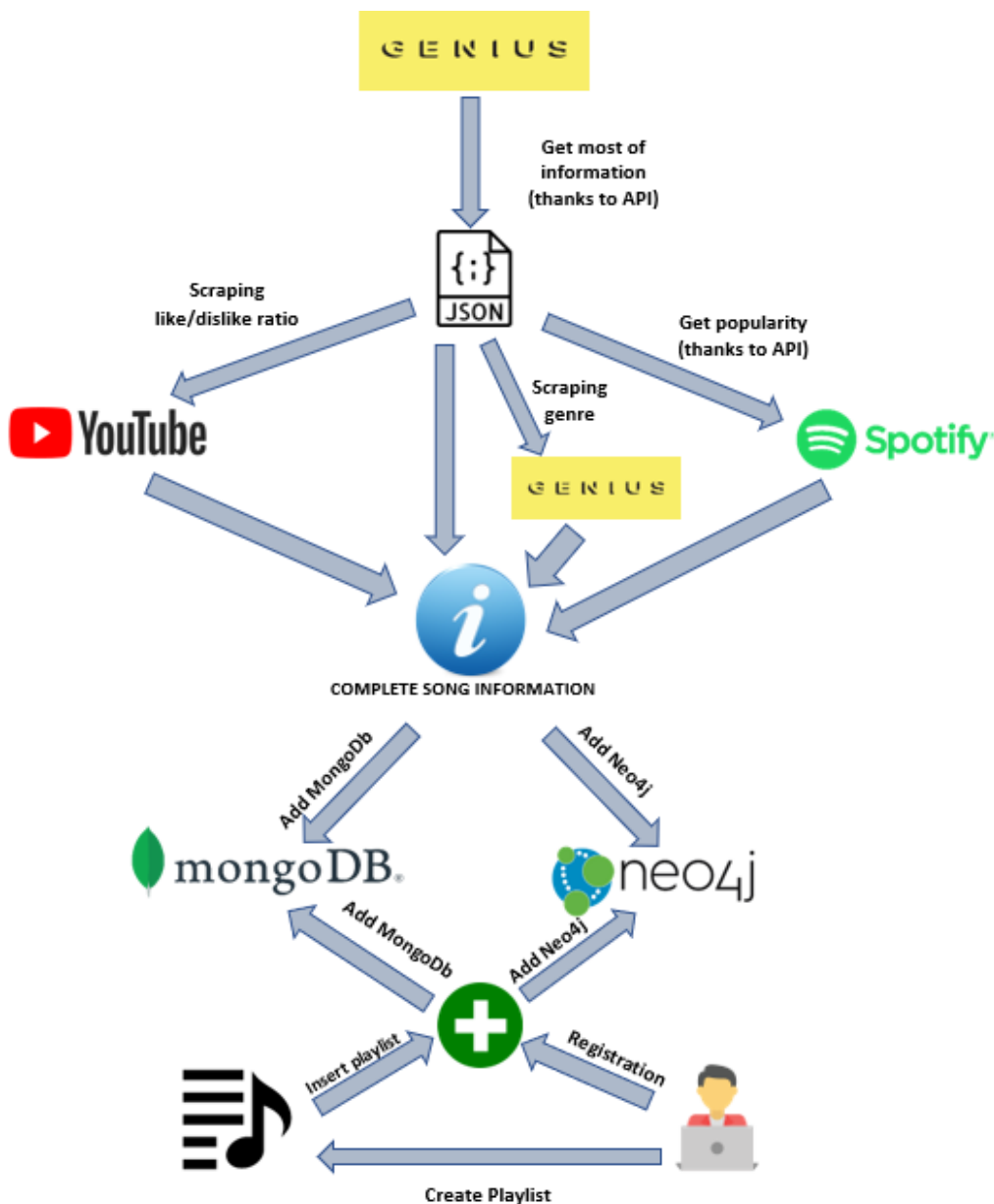
The two last information have been used to create our “popularity” attribute as an aggregation of Spotify popularity and YouTube Like/Dislike ratio using this formula:

$0.7 * SpotifyPopularity + YouTubeRatio * 0.3$.

User and playlist information

Since our application is also a kind of social network, it is important for it to contain a satisfying number of users and playlists created by users. For this reason, we populate our databases using a script that randomly creates a high number of users, playlists, “follow” and “like” relations among them.

Here we have a summary of how the two databases are populated:



Selected NoSQL Databases

We decided to exploit a *Document Database* with **MongoDB** to store informations about entities composing our application dataset and make use of *document embedding* in order to *simplify* and *speed up* the execution of some statistical queries.

We then made the choice of exploiting a *Graph Database* with **Neo4j** to represent the “*network*” part of our application, using it also for the purpose of giving *suggestions* based on the network’s connections.

MongoDB Design and Implementation

Information about songs, users and playlists are stored in MongoDB in two different collections.

User Collection

The document of a user contains all the information about the account and an **array of embedded documents** describing the playlists created by the user; it has been decided to embed the *playlist document* inside the *user collection* due to the *one to many* relationship between those two entities and to improve the **performances** of some of the analytics.

In each playlist are stored information about songs that have been added to it by the user; such information is a **subset** of the totality stored for each song, which can be found in the corresponding **song document**.

This redundancy was chosen to improve the **read performances**, so that we can give a simple preview of all the songs contained inside specific playlists without immediately retrieving all its information with a second read on the song collection.

```
1 {
2   "_id" : "mario1990",
3   "password" : "123456",
4   "firstName" : "Mario",
5   "lastName" : "Rossi",
6   "age" : 30,
7   "country" : "Italy",
8   "privilegeLevel" : "standard_user",
9   "createdPlaylists" : [
10    {
11      "playlistId" : "5fd39867d58de181c8c8775a",
12      "name" : "Favourites",
13      "isFavourite" : true,
14      "songs" : [
15        {
16          "songId" : "5fd24df2dec1f62bba89f985",
17          "title" : "X Gon' Give It to Ya by W55P0MX",
18          "artist" : "DMX",
19          "urlImage" : "https://images.genius.com/3b2d54d2cb438cdc62f8f8ce14d4fa.488x488x1.jpg",
20          "genre" : "rap"
21        },
22        {...}
23      ]
24    },
25    {
26      "playlistId" : "5fd398f51059072884c8f9ac",
27      "name" : "Playlist rock di Mario",
28      "songs" : [
29        {...}
30      ]
31    },
32    {...}
33  ]
34 }
```

Such collection is also used for *login* purposes (e.g. checking if the password is correct).

Song Collection

The document of a song contains all information obtained by the scraping.

```
1  {
2    "_id" : "5fd24d5bdec1f62bba09f8ec",
3    "title" : "Two Words byKanyeWest (Ft. TheBoys Choir of Harlem, Freeway & YasiinBey)",
4    "album" : {
5      "title" : "Get Well Soon... by Kanye West",
6      "image" : "https://images.genius.com/cbe707220b3ae519878db1967dc015cb.602x602x1.png"
7    },
8    "artist" : "Kanye West",
9    "genre" : "rap",
10   "featuredArtists" : [
11     "The Boys Choir of Harlem",
12     "Freeway",
13     "Yasiin Bey"
14   ],
15   "releaseYear" : 2004,
16   "rating" : 29.44980694980695,
17   "likeCount" : 0,
18   "media" : [
19     {
20       "provider" : "youtube",
21       "url" : "http://www.youtube.com/watch?v=tkF08x6j0l8"
22     },
23     {
24       "provider" : "spotify",
25       "url" : "https://open.spotify.com/track/6GI114Ru81Ryc09NcXyU6y"
26     },
27     {
28       "provider" : "genius",
29       "url" : "https://genius.com/Kanye-west-two-words-lyrics"
30     }
31   ]
32 }
```

CRUD operations

Create

The functions used to create Users, Songs and Playlists are these:

```
670 @ private void createUserDocument(User user) throws MongoException {
671     Document userDoc = user.toBsonDocument();
672
673     MongoCollection<Document> userColl = MongoDriver.getInstance().getCollection(Collections.USERS);
674     userColl.insertOne(userDoc);
675 }
676
677 void addPlaylistToUserDocument(User user, Playlist playlist) throws MongoException, IllegalArgumentException{
678
679     if(user == null || playlist == null) throw new IllegalArgumentException();
680
681     Document playlistDoc = playlist.toBsonDocument();
682
683     MongoCollection<Document> userColl = MongoDriver.getInstance().getCollection(Collections.USERS);
684     userColl.updateOne(
685         eq( fieldName: "_id", user.getUsername()),
686         addToSet( fieldName: "createdPlaylists", playlistDoc)
687     );
688 }
689
690 private void createSongDocument(Song song) throws MongoException{
691
692     MongoCollection<Document> songCollection = MongoDriver.getInstance().getCollection(Collections.SONGS);
693
694     Document songDocument = song.toBsonDocument();
695
696     songCollection.insertOne(songDocument);
697 }
698 }
```

Read

The functions used to get the Users, Songs and Playlists using the id are these:

```
127 @Override
128 public User getUserByUsername(String username) throws ActionNotCompletedException{
129     User user = null;
130
131     try (MongoCursor<Document> cursor =
132         MongoDriver.getInstance().getCollection(Collections.USERS).find(eq( fieldName: "_id", username)).iterator()) {
133         if (cursor.hasNext()) {
134             user = new User(cursor.next());
135         }
136     } catch (MongoException mEx) {
137         logger.warn(mEx.getMessage());
138         throw new ActionNotCompletedException(mEx);
139     }
140     return user;
141 }
142
143 public Song getSongById(String songID) {
144
145     MongoCollection<Document> songCollection = MongoDriver.getInstance().getCollection(Collections.SONGS);
146     Song songToReturn = null;
147
148     try (MongoCursor<Document> cursor = songCollection.find(eq( fieldName: "_id", songID)).iterator())
149     {
150         if(cursor.hasNext())
151         {
152             String jsonSong = cursor.next().toJson();
153             songToReturn = new Song(jsonSong);
154         }
155     } catch (MongoException mongoEx) {
156         logger.error(mongoEx.getMessage());
157     }
158     return songToReturn;
159 }
```

```

77 public Playlist getPlaylist(String playlistID) throws ActionNotCompletedException{
78     MongoCollection<Document> usersCollection = MongoDriver.getInstance().getCollection(Collections.USERS);
79     Playlist playlist = null;
80
81
82     Bson match = match(eq( fieldName: "createdPlaylists.playlistId", playlistID));
83     Bson unwind = unwind( fieldName: "$createdPlaylists");
84     Bson project = project(fields(include( fieldName: "_id", "createdPlaylists")));
85
86     try (MongoCursor<Document> cursor = usersCollection.aggregate(Arrays.asList(match, unwind, match, project)).iterator()) {
87         if(cursor.hasNext()) {
88             Document result = cursor.next();
89             playlist = new Playlist(result.get( key: "createdPlaylists", Document.class), result.getString( key: "_id"));
90         }
91     } catch (MongoException mongoEx) {
92         logger.error(mongoEx.getMessage());
93         throw new ActionNotCompletedException(mongoEx);
94     }
95     return playlist;
96 }

```

To retrieve all the playlists created by a user there is the function *getAllPlaylist*:

```

440 public List<Playlist> getAllPlaylist(User user) throws ActionNotCompletedException, IllegalArgumentException {
441     if(user == null) throw new IllegalArgumentException();
442
443     MongoCollection<Document> usersCollection = MongoDriver.getInstance().getCollection(Collections.USERS);
444     List<Playlist> playlists = new ArrayList<>();
445
446     Bson match = match(eq( fieldName: "_id", user.getUsername()));
447     Bson unwind = unwind( fieldName: "$createdPlaylists");
448
449     try (MongoCursor<Document> cursor = usersCollection.aggregate(Arrays.asList(match, unwind)).iterator()) {
450         while(cursor.hasNext()) {
451             Document result = cursor.next();
452             playlists.add(new Playlist(result.get( key: "createdPlaylists", Document.class), user.getUsername()));
453         }
454     } catch (MongoException mongoEx) {
455         logger.error(mongoEx.getMessage());
456         throw new ActionNotCompletedException(mongoEx);
457     }
458     return playlists;
459 }

```

A user can search other users using the search bar thanks to the function *getUserByPartialUsername*:

```

149 List<User> getUserByPartialUsername(String partialUsername, int limitResult) throws ActionNotCompletedException, IllegalArgumentException {
150     if(limitResult <= 0) throw new IllegalArgumentException();
151
152     MongoCollection<Document> userCollection = MongoDriver.getInstance().getCollection(Collections.USERS);
153     List<User> usersToReturn = new ArrayList<>();
154
155     Bson match = match(regex( fieldName: "_id", pattern: "(?i)^" + partialUsername + ".*"));
156
157     try (MongoCursor<Document> cursor = userCollection.aggregate(Arrays.asList(match, limit(limitResult))).iterator()) {
158         while(cursor.hasNext()) {
159             usersToReturn.add(new User(cursor.next()));
160         }
161     } catch (MongoException mongoEx) {
162         logger.error(mongoEx.getMessage());
163         throw new ActionNotCompletedException(mongoEx);
164     }
165     return usersToReturn;
166 }

```


The function to retrieve all the songs of a playlist is the following:

```
283 public List<Song> getAllSongs(Playlist playlist) throws ActionNotCompletedException{
284     MongoCollection<Document> usersCollection = MongoDriver.getInstance().getCollection(Collections.USERS);
285     List<Song> songs = new ArrayList<>();
286
287     Bson match = match(eq( fieldName: "createdPlaylists.playlistId", playlist.getID()));
288     Bson unwind1 = unwind( fieldName: "$createdPlaylists");
289     Bson unwind2 = unwind( fieldName: "$createdPlaylists.songs");
290
291     try (MongoCursor<Document> cursor = usersCollection.aggregate(Arrays.asList(match, unwind1, match, unwind2)).iterator()) {
292         while(cursor.hasNext()) {
293             Document result = cursor.next().get( key: "createdPlaylists", Document.class).get( key: "songs", Document.class);
294             Song song = new Song();
295             song.setID(result.getString( key: "songID"));
296             song.setTitle(result.getString( key: "title"));
297             song.setArtist(result.getString( key: "artist"));
298             song.setAlbum(new Album( title: null, result.getString( key: "urlImage")));
299             song.setGenre(result.getString( key: "genre"));
300             songs.add(song);
301         }
302     } catch (MongoException mongoEx) {
303         logger.error(mongoEx.getMessage());
304         throw new ActionNotCompletedException(mongoEx);
305     }
306     return songs;
307 }
```

In the same way a user can use the search bar to search for songs by title, artist or album, specifying the correct parameter for attributeField using the *filterSong* function:

```
329 public List<Song> filterSong(String partialInput, int maxNumber, String attributeField) throws ActionNotCompletedException {
330
331     if(attributeField == null || maxNumber <= 0)
332         throw new IllegalArgumentException();
333
334     MongoCollection<Document> songCollection = MongoDriver.getInstance().getCollection(Collections.SONGS);
335     List<Song> songsToReturn = new ArrayList<>();
336
337     Bson match = match(regex(attributeField, pattern: "(?i)^(?i) + partialInput + ".*"));
338     Bson sortLike = sort(descending( ...fieldNames: "likeCount"));
339     try (MongoCursor<Document> cursor = songCollection.aggregate(Arrays.asList(match, sortLike, limit(maxNumber))).iterator()) {
340         while(cursor.hasNext()) {
341             String jsonSong = cursor.next().toJson();
342             songsToReturn.add(new Song(jsonSong));
343         }
344     } catch (MongoException mongoEx) {
345         logger.error(mongoEx.getMessage());
346         throw new ActionNotCompletedException(mongoEx);
347     }
348     return songsToReturn;
349 }
```

checkUserPassword is used when there is the check of the credentials inserted in the login form:

```
324 public boolean checkUserPassword(String username, String password) {
325     try (MongoCursor<Document> cursor = MongoDriver.getInstance().getCollection(Collections.USERS)
326         .find(eq( fieldName: "_id", username)).iterator()) {
327         if (cursor.hasNext())
328             if(password.equals(cursor.next().get("password").toString()))
329                 return true;
330     } catch (MongoException mEx) {
331         return false;
332     }
333     return false;
334 }
```

Update

The function *addSong* is used to add a song to a playlist specified by its id:

```
121 public void addSong(Playlist playlist, Song song) throws ActionNotCompletedException{
122     try {
123         MongoClientCollection<Document> usersCollection = MongoClient.getInstance().getCollection(Collections.USERS);
124
125         Document songDocument = new Document("songId", song.getID())
126             .append("title", song.getTitle())
127             .append("artist", song.getArtist())
128             .append("genre", song.getGenre());
129         if (song.getAlbum().getImage() != null)
130             songDocument.append("urlImage", song.getAlbum().getImage());
131
132         Bson find = eq( fieldName: "createdPlaylists.playlistId", playlist.getID());
133         Bson query = push( fieldName: "createdPlaylists.$.songs", songDocument);
134         usersCollection.updateOne(find, query);
135         logger.info("Added song " + song.getID() + " to playlist " + playlist.getID());
136     } catch (MongoException mongoEx) {
137         logger.error(mongoEx.getMessage());
138         throw new ActionNotCompletedException(mongoEx);
139     }
140 }
141 }
```

When a user puts a *like* to a song, the document of the song needs to be updated to maintain the **redundancy** *\$likeCount*, the function used in this situation is the following (there is an analogue function when a like is removed):

```
408 public void incrementLikeCount(Song song) throws ActionNotCompletedException{
409
410     if(song == null || song.getID() == null)
411         throw new IllegalArgumentException();
412
413     MongoClientCollection<Document> songCollection = MongoClient.getInstance().getCollection(Collections.SONGS);
414     try{
415         songCollection.updateOne(eq( fieldName: "_id", song.getID()), inc( fieldName: "likeCount", number: 1));
416         song.setLikeCount(song.getLikeCount()+1);
417     } catch (MongoException mongoEx) {
418         logger.error(mongoEx.getMessage());
419         throw new ActionNotCompletedException(mongoEx);
420     }
421 }
422 }
```

An admin can update the privilege level of a user:

```
425 public void updateUserPrivilegeLevel(User user, PrivilegeLevel newPrivLevel) throws ActionNotCompletedException, IllegalArgumentException {
426     if(user == null || newPrivLevel == null)
427         throw new IllegalArgumentException();
428
429     user.setPrivilegeLevel(newPrivLevel);
430     try {
431         updateUserDocument(user);
432         logger.info("Updated privilege level of user <" +user.getUsername()+ "> : new level <" +user.getPrivilegeLevel()+ ">");
433     } catch (MongoException mEx) {
434         logger.warn(mEx.getMessage());
435         throw new ActionNotCompletedException(mEx);
436     }
437 }
438 }
```



```

685 @ private void updateUserDocument(User user) throws MongoException {
686     MongoCollection<Document> userColl = MongoDriver.getInstance().getCollection(Collections.USERS);
687     userColl.updateOne(
688         eq( fieldName: "_id", user.getUsername()),
689         combine(
690             set("firstName", user.getFirstName()),
691             set("lastName", user.getLastName()),
692             set("age", user.getAge()),
693             set("privilegeLevel", user.getPrivilegeLevel().toString())
694         ));
695 }

```

Delete

An admin can delete a user:

```

697 @Override
698 public void deleteUserDocument(User user) throws MongoException {
699     MongoCollection<Document> userColl = MongoDriver.getInstance().getCollection(Collections.USERS);
700     userColl.deleteOne(eq( fieldName: "_id", user.getUsername()));
701 }
702

```

A user can delete its own playlists using the function:

```

321 @ public void deletePlaylistDocument(Playlist playlist) throws MongoException{
322     try {
323         Bson find = eq( fieldName: "createdPlaylists.playlistId", playlist.getID());
324         MongoCollection<Document> usersCollection = MongoDriver.getInstance().getCollection(Collections.USERS);
325         usersCollection.updateOne(find, pull( fieldName: "createdPlaylists", eq( fieldName: "playlistId", playlist.getID())));
326     }
327     catch (MongoException mongoEx) {
328         throw mongoEx;
329     }
330 }

```

Queries Analysis and MongoDB Connection String

Write operations results *less frequent* and generally simple (at most one document created or updated at a time) while, on the opposite, **Read operations** are more frequently performed and involve a higher number of documents.

Furthermore, the system has *response time* and *availability* requirements, therefore it is tuned as follow:

- ❖ **Write Concern:** 1. This allows the system to be as fast as possible during write operations, introducing the need to implement some kind of **Eventual Consistency paradigm**.
- ❖ **Read Preferences:** nearest. Read operations are performed on the node with the lowest network latency to have the fastest response.

To set those options, the following connection string is used in the MongoDB Java Driver:

`mongodb://172.16.3.115:27020,172.16.3.114:27020,172.16.3.161:27020/?w=1&readPreference=nearest`

Analytics and Statistics

In the following the four pipelines aggregations used to extract interesting information from data are described.

Artists with Most Hit Songs

Description: Select artists which made the highest number of “hit songs”.

A song is a “hit” if it received more than “*hitLimit*” likes.

Mongo java driver:

```
337 of public List<Pair<String, Integer>> findArtistsWithMostNumberOfHit(int hitLimit, int maxNumber) throws ActionNotCompletedException {
338
339     if(maxNumber <= 0)
340         throw new IllegalArgumentException();
341
342     MongoCollection<Document> songCollection = MongoDriver.getInstance().getCollection(Collections.SONGS);
343
344     List<Pair<String, Integer>> artistRank = new ArrayList<>();
345
346     Bson match = match(gt( fieldName: "LikeCount", hitLimit));
347     Bson group = group( id: "$artist", sum( fieldName: "numberOfHits", expression: 1));
348     Bson sortHits = sort(descending( fieldName: "numberOfHits"));
349     Bson limit = limit(maxNumber);
350     Bson project = project(fields(excludeId(), computed( fieldName: "artist", expression: "$_id", include( fieldName: "numberOfHits"))));
351     try (MongoCursor<Document> cursor = songCollection.aggregate(Arrays.asList(match, group, sortHits, limit, project)).iterator()) {
352         while(cursor.hasNext()) {
353             Document artist = cursor.next();
354             artistRank.add(new Pair<>(artist.getString( key: "artist"), artist.getInteger( key: "numberOfHits")));
355         }
356     } catch (MongoException mongoEx) {
357         logger.error(mongoEx.getMessage());
358         throw new ActionNotCompletedException(mongoEx);
359     }
360     return artistRank;
361 }
```

Mongo with “*hitLimit*” = 0, “*maxNumber*” = 5:

```
1 db.songs.aggregate([
2   {
3     $match:
4     {
5       "LikeCount":{ $gte: 0}
6     }
7   },
8   {
9     $group:
10    {
11      _id: "$artist", numberOfHits: { $sum: 1}
12    }
13  },
14  { $sort: { "numberOfHits": -1}},
15  { $limit: 5},
16  {
17    $project:
18    {
19      _id:0,
20      artist: "$_id",
21      numberOfHits:1
22    }
23  }
24 ])
```

Top Rated Album per Decade

Description: For every decade, select the album with the highest average of rating.

Mongo java driver:

```
public List<Pair<Integer, Pair<Album, Double>>> findTopRatedAlbumPerDecade() throws ActionNotCompletedException {  
    MongoClient<Document> songCollection = MongoDriver.getInstance().getCollection(Collections.SONGS);  
    List<Pair<Integer, Pair<Album, Double>>> topAlbums = new ArrayList<>();  
  
    Document computeExpression = Document.parse("{ $multiply: [{ $floor: { $divide: [ \"$releaseYear\", 10 ] }, 10 ] }");  
  
    Bson match = match(exists( fieldName: "releaseYear"));  
    Bson project = project(fields(excludeId(),  
                                include( fieldName: "releaseYear",  
                                include( fieldName: "album",  
                                include( fieldName: "rating",  
                                computed( fieldName: "decade", computeExpression)));  
  
    Bson group = Document.parse("{ $group: { _id: { title: \"$album.title\", url: \"$album.image\", decade: \"$decade\" }, avgRating: { $avg: \"$rating\" } } }");  
    Bson sortRate = sort(ascending( fieldName: "avgRating"));  
  
    Bson group2 = Document.parse("{ $group: { " +  
        "_id: \"$ _id.decade\", " +  
        "topAlbumTitle: { $last: \"$ _id.title\" }, " +  
        "topAlbumUrl: { $last: \"$ _id.url\" }, " +  
        "avgRating: { $last: \"$avgRating\" } " +  
        " } }");  
    Bson sortId = sort(ascending( fieldName: "_id"));  
    Bson project2 = project(fields(excludeId(),  
                                computed( fieldName: "decade", expression: "$ _id"),  
                                include( fieldName: "topAlbumTitle"),  
                                include( fieldName: "avgRating"),  
                                include( fieldName: "topAlbumUrl")));  
  
    try (MongoCursor<Document> cursor = songCollection.aggregate(Arrays.asList(match, project, group, sortRate, group2, sortId, project2)).iterator()) {  
        while(cursor.hasNext()) {  
            Document record = cursor.next();  
  
            Album albumToAdd = new Album(); albumToAdd.setTitle(record.getString( key: "topAlbumTitle")); albumToAdd.setImage(record.getString( key: "topAlbumUrl"));  
  
            int decade = record.getDouble( key: "decade").intValue();  
  
            double avgRating = record.getDouble( key: "avgRating");  
  
            Pair<Integer, Pair<Album, Double>> resultToAdd = new Pair<>(decade, new Pair<>(albumToAdd, avgRating));  
            topAlbums.add(resultToAdd);  
        }  
    } catch (MongoException mongoEx) {  
        logger.error(mongoEx.getMessage());  
        throw new ActionNotCompletedException(mongoEx);  
    }  
    return topAlbums;  
}
```

Mongo:

```
1 db.songs.aggregate([  
2   {  
3     $match: {releaseYear: {$exists: true}}  
4   },  
5   {  
6     $project: { _id: 0, releaseYear: 1, album: 1, rating: 1, decade: { $multiply: [{ $floor: { $divide: [ "$releaseYear", 10 ] }, 10 ] } }  
7   },  
8   { $group:  
9     {  
10      _id: {album: "$album.title", decade: "$decade"}, avgRating: { $avg: "$rating"  
11    }  
12  },  
13  { $sort: { "avgRating": 1 } },  
14  {  
15    $group:  
16    {  
17      _id: "$ _id.decade",  
18      topAlbum: { $last: "$ _id.album",  
19      avgRating: { $last: "$avgRating"  
20    }  
21  },  
22  { $sort: { "_id": 1 } },  
23  {  
24    $project:  
25    {  
26      _id: 0,  
27      decade: "$ _id",  
28      topAlbum: 1,  
29      avgRating: 1  
30    }  
31  }  
32 ]  
33 ])
```

Top Favourite Genres

Description: Select the top favourite genres of users. A genre is the favourite of a user if it is the most present genre in his playlists.

Mongo java driver:

```
531 @Override
532 public List<Pair<String, Integer>> getFavouriteGenres(int numGenres) throws ActionNotCompletedException, IllegalArgumentException{
533     if(numGenres <= 0) throw new IllegalArgumentException();
534
535     MongoClient usersCollection = MongoClient.getInstance().getCollection(Collections.USERS);
536     List<Pair<String, Integer>> result = new ArrayList<>();
537     Bson unwind1 = unwind( "fieldName": "$createdPlaylists");
538     Bson unwind2 = unwind( "fieldName": "$createdPlaylists.songs");
539     Bson group = Document.parse("{ $group: { " +
540         "    _id: \"\"$createdPlaylists.songs.genre\"\", " +
541         "    totalSongs: { $sum: 1 } " +
542         "}}");
543     Bson sort = sort(descending( "fieldName": "totalSongs"));
544     Bson limit = limit(numGenres);
545     try (MongoCursor<Document> cursor = usersCollection.aggregate(Arrays.asList(unwind1, unwind2, group, sort, limit)).iterator()) {
546         while(cursor.hasNext()) {
547             Document genre = cursor.next();
548             result.add(new Pair<>(genre.getString( keys: "_id"), genre.getInteger( keys: "totalSongs")));
549         }
550     } catch (MongoException mEx) {
551         throw new ActionNotCompletedException(mEx);
552     }
553     return result;
554 }
```

Mongo with “maxGenres = 3”:

```
1 db.users.aggregate(
2     {$unwind: "$createdPlaylists",
3     {$unwind: "$createdPlaylists.songs",
4     {
5         $group:
6         {
7             _id: "$createdPlaylists.songs.genre",
8             totalSongs: { $sum: 1}
9         }
10    },
11    {$sort: {totalSongs: -1}},
12    {$limit: 3}
13 );
```

Top Favourite Artist per Age Range

Description: Select the top artist per age range.

Mongo java driver:

```
public List<Pair<Integer, Pair<String, String>>> getFavouriteArtistPerAgeRange() throws ActionNotCompletedException {
    MongoClient userCollection = MongoClient.getInstance().getCollection(Collections.USERS);
    List<Pair<Integer, Pair<String, String>>> topArtists = new ArrayList<>();
    Document computeDecade = Document.parse("{ $multiply: [{ $floor: { $divide: [ \"$age\", 10 ] } }, 10 ] }");
    Bson match = match(exists( fieldName: "age"));
    Bson projectDecade = project(fields(
        excludeId(),
        include( fieldName: "age"),
        include( fieldName: "createdPlayLists"),
        computed( fieldName: "decade", computeDecade)
    ));
    Bson unwind1 = unwind( fieldName: "$createdPlayLists");
    Bson unwind2 = unwind( fieldName: "$createdPlayLists.songs");
    Bson groupArtists = Document.parse(
        "{ $group: " +
        "  { " +
        "    _id: { decade: \"$decade\", artist: \"$createdPlayLists.songs.artist\" }, " +
        "    presences: { $sum: 1 } " +
        "  } " +
        "}"
    );
    Bson sortPresences = sort(descending( fieldName: "presences"));
    Bson groupDecades = Document.parse(
        "{ $group: " +
        "  { " +
        "    _id: \"$_id.decade\", " +
        "    favouriteArtist: { $first: \"$_id.artist\" }, " +
        "    artistPresences: { $first: \"$presences\" }, " +
        "    overallPresences: { $sum: \"$presences\" } " +
        "  } " +
        "}"
    );
    Bson finalProject = project(fields(
        excludeId(),
        include( fieldName: "favouriteArtist"),
        computed( fieldName: "decade", expression: "$_id"),
        include( fieldName: "artistPresences"),
        include( fieldName: "overallPresences")
    ));
    Bson sortDecades = sort(ascending( fieldName: "decade"));
    try (MongoCursor<Document> cursor = userCollection.aggregate(Arrays.asList(match, projectDecade, unwind1, unwind2,
                                                                              groupArtists, sortPresences, groupDecades,
                                                                              finalProject, sortDecades)).iterator()) {
        while (cursor.hasNext()) {
            Document record = cursor.next();
            int decade = record.getDouble( key: "decade").intValue();
            String artist = record.getString( key: "favouriteArtist");
            String presences = String.valueOf(record.getInteger( key: "artistPresences"));
            String overallPresences = String.valueOf(record.getInteger( key: "overallPresences"));
            topArtists.add(new Pair<>(decade, new Pair<>(artist, presences + " " + overallPresences)));
        }
    } catch (MongoException mongoEx) {
        logger.error(mongoEx.getMessage());
        throw new ActionNotCompletedException(mongoEx);
    }
    return topArtists;
}
```

Mongo:

```
1 db.users.aggregate(  
2   { $match: { age: { $exists: true } } },  
3   { $project:  
4     {  
5       _id: 0,  
6       age: 1,  
7       createdPlaylists: 1,  
8       decade: {  
9         $multiply: [{ $floor: { $divide: [ "$age", 10 ] } }, 10]  
10      }  
11    }  
12  },  
13  { $unwind: { path: "$createdPlaylists" } },  
14  { $unwind: { path: "$createdPlaylists.songs" } },  
15  { $group:  
16    {  
17      _id: { decade: "$decade", artist: "$createdPlaylists.songs.artist" },  
18      presences: { $sum: 1 }  
19    }  
20  },  
21  { $sort: { presences: -1 } },  
22  { $group:  
23    {  
24      _id: "$_id.decade",  
25      favouriteArtist: { $first: "$_id.artist" },  
26      artistPresences: { $first: "$presences" },  
27      overallPresences: { $sum: "$presences" }  
28    }  
29  },  
30  { $project:  
31    {  
32      _id: 0,  
33      $artistPresences: 1,  
34      $overallPresences: 1,  
35      favouriteArtist: 1,  
36      decade: "$_id"  
37    }  
38  },  
39  { $sort: { decade: 1 } }  
40 );  
41
```

Indexes Analysis

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

Index Type	Collection	Index Name	Index Field
Hashed Index (default)	songs	<i>id</i>	<i>_id</i>
Simple index	songs	title	title
Simple index	songs	artist	artist
Simple index	songs	album	album.title
Hashed Index (default)	users	<i>id</i>	<i>_id</i>
Simple index	users	playlist	createdPlaylist.playlistId

They are useful to improve the main queries that are executed more frequently:

- *title*, *artist* and *album* whenever the search bar is used (*filterSong*).
- *playlist* whenever is displayed the page of a playlist (*getPlaylist*).

Index “artist”

Below is shown the improvement with the equality match between a **partial artist’s name** and a **regular expression**.

As we can see were examined all the documents (124561), while with the index just the right documents that match the regular expression (6212) are examined.

Consequently, the **execution time** decrease very much: from 82 to 15 milliseconds.

```
@workgroup-10@ProfileLARGESCALE33:~
$ mongo --quiet --eval 'use music; db.songs.find({artist:/^A./}).explain("executionStats")'
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "music.songs",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "artist" : {
        "$regex" : "^A."
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "artist" : {
          "$regex" : "^A."
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 6212,
    "executionTimeMillis" : 82,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 124561
  }
}
```

Without artist Index

```
use music;
db.songs.createIndex({artist:1});
db.songs.find({artist:/^A./}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "music.songs",
    "indexFilterSet" : true,
    "parsedQuery" : {
      "artist" : {
        "$regex" : "^A."
      }
    },
    "winningPlan" : {
      "stage" : "INDEXSCAN",
      "filter" : {
        "artist" : {
          "$regex" : "^A."
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 6212,
    "executionTimeMillis" : 15,
    "totalKeysExamined" : 6213,
    "totalDocsExamined" : 6212
  }
}
```

With artist Index

Index “album”

Below is shown the improvement with the equality match between a **partial album’s title** and a **regular expression**.

As we can see we examined all the documents (124561), while with the index just the right document that match the regular expression (6188) are examined.

Consequently, the **execution time** decrease very much: from 86 to 16 milliseconds.

```
workgroup-10@ProfileLARGESCALE33: ~
lsmdb:PRIMARY> db.songs.find({"album.title":{"regex":"A.*"}}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "UniMusic.songs",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "album.title" : {
        "$regex" : "A.*"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "album.title" : {
          "$regex" : "A.*"
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 6188,
    "executionTimeMillis" : 86,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 124561,
  }
}
```

Without album Index

```
},
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 6188,
  "executionTimeMillis" : 16,
  "totalKeysExamined" : 6189,
  "totalDocsExamined" : 6188,
  "executionStages" : {

```

With album Index

Index “title”

Below is shown the improvement with the equality match between a **partial song’s title** and a **regular expression**.

As we can see we examined all the documents (124561), while with the index just the right document that match the regular expression (6619) are examined.

Consequently, the **execution time** decreased very much: from 81 to 18 milliseconds.

```
workgroup-10@ProfileLARGESCALE33: ~
lsmdb:PRIMARY> db.songs.find({"title":{"regex":"A.*"}}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "UniMusic.songs",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "title" : {
        "$regex" : "A.*"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "title" : {
          "$regex" : "A.*"
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 6619,
    "executionTimeMillis" : 81,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 124561,
  }
}
```

Without title Index


```

    },
    "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 6619,
      "executionTimeMillis" : 18,
      "totalKeysExamined" : 6620,
      "totalDocsExamined" : 6619,
      "executionStages" : {

```

With title Index

Index “playlist”

Below is shown the improvement when we **retrieve a playlist** specifying the ID.

As we can see were examined all the documents (13745), while with the index just the correct one.

Consequently, the **execution time** decrease very much: from 20 to 0 milliseconds.

```

@workgroup-10@ProfileLARGESCALE33 ~
lsmdb:PRIMARY> db.users.find({"createdPlaylists.playlistId":"5fecb8eff159fcb415ed1db").explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "UniMusic.users",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "createdPlaylists.playlistId" : {
        "$eq" : "5fecb8eff159fcb415ed1db"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "createdPlaylists.playlistId" : {
          "$eq" : "5fecb8eff159fcb415ed1db"
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 20,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 13745,

```

Without playlist Index

```

    },
    "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 1,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 1,
      "totalDocsExamined" : 1,
      "executionStages" : {

```

With playlist Index

Neo4j Design and Implementation

Nodes

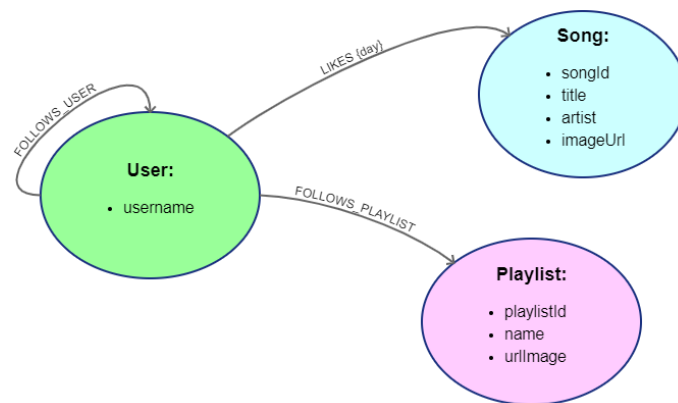
There is a node for each **main entity** of our application, in particular (:Song), (:User) and (:Playlist). They just contain **essential properties** needed to display a **preview** of the entity.

Relations

USER - FOLLOW_USER -> USER: this relation is added whenever a user adds another to his followed user list. It is useful to retrieve suggested playlist and suggested user.

USER - FOLLOW_PLAYLIST -> PLAYLIST: this relation is added whenever a user saves another user's playlist to his playlist's list. It is useful to retrieve suggested playlist and to have always handy in the user's profile followed playlists. It is important to note that it is just a link to the playlist, so if the author modify or delete it, this action will affect the playlist saved on our profile.

USER - LIKES -> SONG: this relation is added whenever a user likes a song. It is useful to retrieve hot song (that also use the relation attribute "day") and the second layer of suggested user.



Two screenshots of the Neo4j database are shown below:



Queries Implementation

CRUD Operations

The following tables contain a collection of all the **CRUD operations** done within the application, along with **their Cypher implantation**.

Create

Operation	Cypher Implementation
Create a user node	MERGE (a:User {username: \$username})
Create song node	CREATE (p:Song {songId: \$songId, title: \$title, artist: \$artist, imageUrl: \$imageUrl})
Create playlist node	CREATE (p:Playlist {playlistId: \$playlistId, name: \$name, urlImage: \$urlImage})
Add a FOLLOW_USER relation	MATCH (following:User { username: \$following }) MATCH (followed:User { username: \$followed }) WHERE following <> followed MERGE (following)-[:FOLLOWS_USER]->(followed)
Add a LIKES relation	MATCH (u:User { username: \$username }) MATCH (s:Song { songId: \$songId }) MERGE (u)-[:LIKES {day: date()}]->(s)
Add a FOLLOW_PLAYLIST relation	MATCH (following:User { username: \$following }) MATCH (followed:Playlist { playlistId: \$followed }) MERGE (following)-[:FOLLOWS_PLAYLIST]->(followed)

Read

Operation	Cypher Implementation
Get total number of users	MATCH (:User) RETURN COUNT(*) AS NUM
Get total number of songs	MATCH (:Song) RETURN COUNT(*) AS NUM
Get total number of playlists	MATCH (:Playlist) RETURN COUNT(*) AS NUM

Operation	Cypher Implementation
Get all the playlist followed by a user	MATCH (:User {username: \$username})-[:FOLLOWS_PLAYLIST]->(playlist:Playlist) RETURN playlist
Get all the user followed by a user	MATCH (:User {username: \$username})-[:FOLLOWS_USER]->(followedUser:User) RETURN followedUser
Get all the follower of a user	MATCH (followingUser:User)-[:FOLLOWS_USER]->(:User {username: \$username}) RETURN followingUser
Check if a user follows a specific playlist	MATCH (:User { username: \$username })-[f:FOLLOWS_PLAYLIST]-> (:Playlist { playlistId: \$playlistId }) RETURN count(*) AS Times
Check if a user is followed by a certain user	MATCH (:User { username: \$following })-[f:FOLLOWS_USER]-> (:User { username: \$followed }) RETURN count(*) AS Times
Check if a user likes a specific song	MATCH (:User { username: \$username })-[l:LIKES]->(:Song { songId: \$songId }) RETURN count(*) AS Times

Update

Operation	Cypher Implementation
Update the name of a Playlist	MATCH (:Playlist { playlistId: \$playlistId }) SET name = \$newName

Delete

Operation	Cypher Implementation
Remove a FOLLOW_USER relation	MATCH (:User { username: \$username1 })-[f:FOLLOWS_USER]-> (:User { username: \$username2 }) DELETE f
Remove a FOLLOW_PLAYLIST relation	MATCH (:User { username: \$username })-[f:FOLLOWS_PLAYLIST]-> (:Playlist { playlistId: \$playlistId }) DELETE f

Remove a LIKE relation	MATCH (:User { username: \$username })-[I:LIKES]->(:Song { songId: \$songId }) DELETE I
Delete a user node	MATCH (a:User {username: \$username}) DETACH DELETE a
Delete playlist node	MATCH (p:Playlist { playlistId: \$playlistId }) DETACH DELETE p

“On-graph” queries

Suggested User

Using Neo4j we are able to find **two levels** of suggested user.

Graph-Centric Query	Domain-specific Query
Considering only vertices that takes only 1 hop from vertex “User A” using edge “FOLLOW_USER”, select all vertices that take 1 hop from them using edge “FOLLOW_USER”.	Who are the users that are followed by user that “User A” follows?
Find the vertices “User X” that reach the highest number of vertices Song with the edge “LIKES”, where the same vertex Song is reached by “User A” with the edge “LIKES”.	Who are the users that have the highest number of likes in common with “User A”?

```
@Override
public List<User> getSuggestedUsers(User user, int limit) throws ActionNotCompletedException, IllegalArgumentException {
    if(limit <= 0 || user == null) throw new IllegalArgumentException();

    List<User> list = new ArrayList<>();
    try( Session session = Neo4jDriver.getInstance().getDriver().session()) {
        //First layer Suggestion
        list = session.readTransaction((TransactionWork<List<User>>) tx -> {
            Result result = tx.run(
                "MATCH (me:User {username: $me})-[:FOLLOWS_USER]->(followed:User)"
                + "-[:FOLLOWS_USER]->(suggested:User) WHERE NOT (me)-[:FOLLOWS_USER]->(suggested) "
                + "AND me <> suggested RETURN suggested, count(*) AS Strength "
                + "ORDER BY Strength DESC LIMIT $limit",
                parameters("me", user.getUsername(), "limit", limit)
            );
            ArrayList<User> firstLayerUsers = new ArrayList<>();
            while ((result.hasNext())){
                Record r = result.next();
                firstLayerUsers.add(new User(r.get("suggested").get("username").asString()));
            }
            return firstLayerUsers;
        });

        final int firstSuggestionSize = list.size();
        if(firstSuggestionSize < limit) {
            //Second layer suggestion
            List<User> secondLayerSuggestion = session.readTransaction((TransactionWork<List<User>>) tx -> {
                Result result = tx.run(
                    "MATCH (me:User {username: $username})-[:LIKES]->(<-[:LIKES]->(suggested:User) "
                    + "WHERE NOT (me)-[:FOLLOWS_USER]->(suggested) "
                    + "AND NOT (me)-[:FOLLOWS_USER]->()-[:FOLLOWS_USER]->(suggested) AND me <> suggested "
                    + "RETURN suggested, count(*) AS Strength ORDER BY Strength DESC LIMIT $limit",
                    parameters("username", user.getUsername(), "limit", limit - firstSuggestionSize)
                );
                ArrayList<User> secondLayerUsers = new ArrayList<>();
                while ((result.hasNext())){
                    Record r = result.next();
                    secondLayerUsers.add(new User(r.get("suggested").get("username").asString()));
                }
                return secondLayerUsers;
            });
            list.addAll(secondLayerSuggestion);
        }
    } catch (Neo4jException n4jEx) {
        throw new ActionNotCompletedException(n4jEx);
    }
    return list;
}
```

Suggested Playlist

Using neo4j we are able to find two levels of **suggested** Playlist.

Graph-Centric Query	Domain-specific Query
Considering only vertices that takes only 1 hop from vertex "User A" using edge "FOLLOW_USER", select all vertices that take 1 hop from them using edge "FOLLOW_PLAYLIST".	Which are the Playlists that are followed by user that "User A" follows?
Considering only vertices that are selected by first layer of User suggestion, select all vertices that take 1 hop from them using edge "FOLLOW_PLAYLIST".	Which are the Playlists that are followed by User that are followed by User that "User A" follows?

```

@Override
public List<Playlist> getSuggestedPlaylists(User user, int limit) throws ActionNotCompletedException{
    if (user == null || limit <= 0)
        return new ArrayList<>();
    List<Playlist> firstList = new ArrayList<Playlist>();
    List<Playlist> secondList = new ArrayList<Playlist>();
    try( Session session = Neo4jDriver.getInstance().getDriver().session()) {
        firstList = session.readTransaction((TransactionWork<List<Playlist>>) tx -> {
            //first level suggestions
            Result result = tx.run(
                "MATCH (me:User {username: $me})-[:FOLLOWS_USER]->(followed:User)"
                + "-[:FOLLOWS_PLAYLIST]->(suggested:Playlist) WHERE NOT (me)-[:FOLLOWS_PLAYLIST]->(suggested)"
                + "RETURN suggested, count(*) AS Strength"
                + "ORDER BY Strength DESC LIMIT $limit",
                parameters("me", user.getUsername(), "limit", limit)
            );
            ArrayList<Playlist> playlists = new ArrayList<Playlist>();
            while ((result.hasNext())){
                Record r = result.next();
                playlists.add(new Playlist(r.get("suggested")));
            }
            return playlists;
        });

        //second level suggestions
        final int firstSuggestionsSize = firstList.size();

        if (firstList.size() < limit) {
            secondList = session.readTransaction((TransactionWork<List<Playlist>>) tx2 -> {
                //first level suggestions
                Result result = tx2.run(
                    "MATCH (me:User {username: $me})-[:FOLLOWS_USER]->(followed:User)\n" +
                    "-[:FOLLOWS_USER]->(suggestedUser:User)-[:FOLLOWS_PLAYLIST]->(suggestedPlaylist) \n" +
                    "WHERE NOT (me)-[:FOLLOWS_USER]->(suggestedUser) \n" +
                    "AND me <> suggestedUser \n" +
                    "AND NOT (me)-[:FOLLOWS_PLAYLIST]->(suggestedPlaylist)\n" +
                    "AND NOT (followed)-[:FOLLOWS_PLAYLIST]->(suggestedPlaylist)\n" +
                    "RETURN suggestedPlaylist, count(*) AS Strength \n" +
                    "ORDER BY Strength DESC LIMIT $limit",
                    parameters("me", user.getUsername(), "limit", limit - firstSuggestionsSize)
                );
                ArrayList<Playlist> playlists = new ArrayList<Playlist>();
                while ((result.hasNext())) {
                    Record r = result.next();
                    playlists.add(new Playlist(r.get("suggestedPlaylist")));
                }
                return playlists;
            });
        }
    } catch (Neo4jException n4jEx) {
        logger.error(n4jEx.getMessage());
        throw new ActionNotCompletedException(n4jEx);
    }
    firstList.addAll(secondList);
    return firstList;
}

```

Hot songs

Using neo4j we are able to find songs that received the **highest number of likes in the last month**.

Graph-Centric Query	Domain-specific Query
Which are the Songs with the highest number of incoming edge "LIKES" that has the property "day" contained in the last 30 days?	Which are the songs that received the highest number of likes in the last month?

```
@Override
public List<Song> getHotSongs(int limit) throws ActionNotCompletedException {

    List<Song> hotSongs = new ArrayList<>();

    try ( Session session = Neo4jDriver.getInstance().getDriver().session() )
    {
        session.writeTransaction((TransactionWork<List<Song>>) tx -> {

            LocalDate lastMonth = LocalDate.now().minusDays(30);

            String query = "MATCH (s:Song)->[l:LIKES]-(u:User) " +
                "WHERE l.day > date($lastMonth) " +
                "WITH s, COUNT(*) as num ORDER BY num DESC " +
                "RETURN s.songId as songId, s.title as title, s.artist as artist, s.imageUrl as imageUrl " +
                "LIMIT $limit";

            Result result = tx.run(query, parameters( ...keysAndValues: "lastMonth", lastMonth.toString(), "limit", limit));
            while(result.hasNext()) {
                hotSongs.add(new Song(result.next()));
            }
            return hotSongs;
        });
    } catch (Neo4jException neoEx) {
        logger.error(neoEx.getMessage());
        throw new ActionNotCompletedException(neoEx);
    }
    return hotSongs;
}
```

Neo4j Indexes

In order to **enhance the read operations** on the database, the following indexes were introduced to find the **starting point** of a **graph traversal**:

Index Type	Index Name	Index Label	Index Attributes
BTREE (simple index)	User_index	:User	username
BTREE (simple index)	Playlist_index	:Playlist	playlistId

```
neo4j@neo4j> show indexes;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | name           | state | populationPercent | uniqueness | type   | entityType | labelsOrTypes | properties | indexProvider |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2 | "Playlist_index" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"     | ["Playlist"]  | ["playlistId"] | "native-btree-1.0" |
| 1 | "User_index"     | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"     | ["User"]      | ["username"]  | "native-btree-1.0" |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```


Other Implementation Details

Shards Configuration and Non-Functional Requirements

According to the *Non-Functional Requirements*, the system must provide *high availability*, *fast response times* and be *tolerant to data lost* and *single point of failure*.

To obtain such results, we decided to orientate our application on the **A** (Availability) **P** (Partition Tolerance) edge of the CAP triangle, therefore having the need to adopt the *Eventual Consistency* paradigm on our dataset. This choice leads to make the content of the application highly available, even if an error occur on the physical network, at the cost of returning to the user data that is not always accurate.

The possibility to implement a sharding for the of the database would allow to further reduce the latency/response time for server interactions, improving user experience.

In that regard, we thought to partition the *document database* in shards hosted each one on a different erver. Such **sharding**, together with the already implemented **data replication**, would allow for **load-balancing** purposes, achieving the characteristics required by the **Non-Functional Requirements**.

To implement such sharding, we would select two different **sharding keys** and **partitioning method**, one for each collection of our document database (*User Collection* and *Song Collection*):

- ❖ For the **User Collection** we decided to choose the attribute “*country*” as a **sharding key** and as a **partitioning method** we choose a *List*, to map each country to a specific server in a way to further reduce the response time such mapping should be done in a way to balance the load on servers.
- ❖ For the **Song Collection** we decided to choose the attribute “*_id*” as a **sharding key** and as a **partitioning method** we choose a *Hashing*, eventually applying *Consistent Hashing* to relocate data when a server goes down or a new server is added.

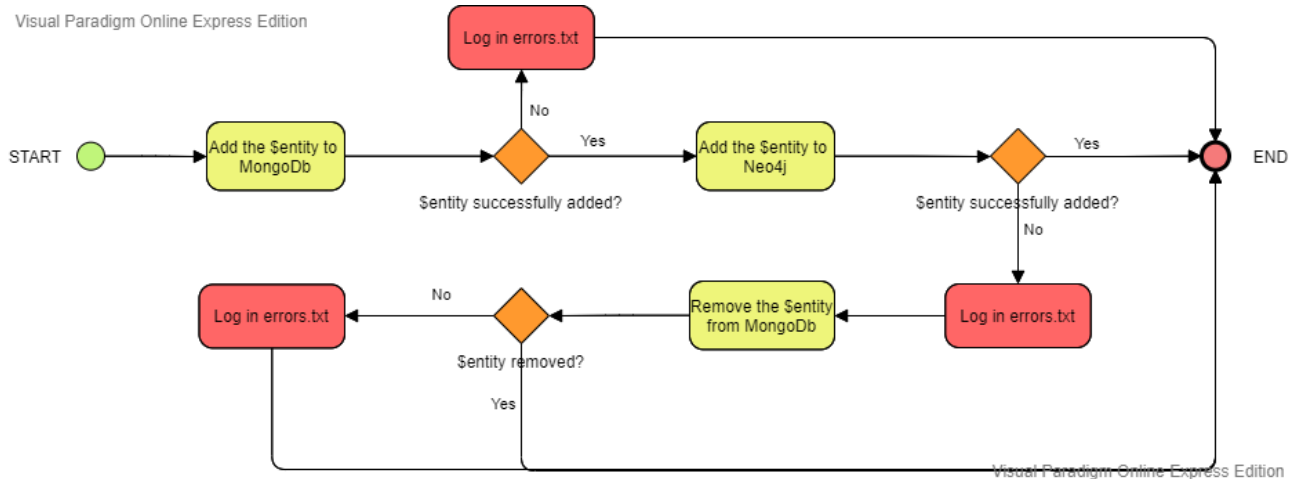
Cross-Database Consistency Management

The operations that require a cross-database consistency management are the Add/Remove a Song/User/Playlist and Add/Remove a like to a song:

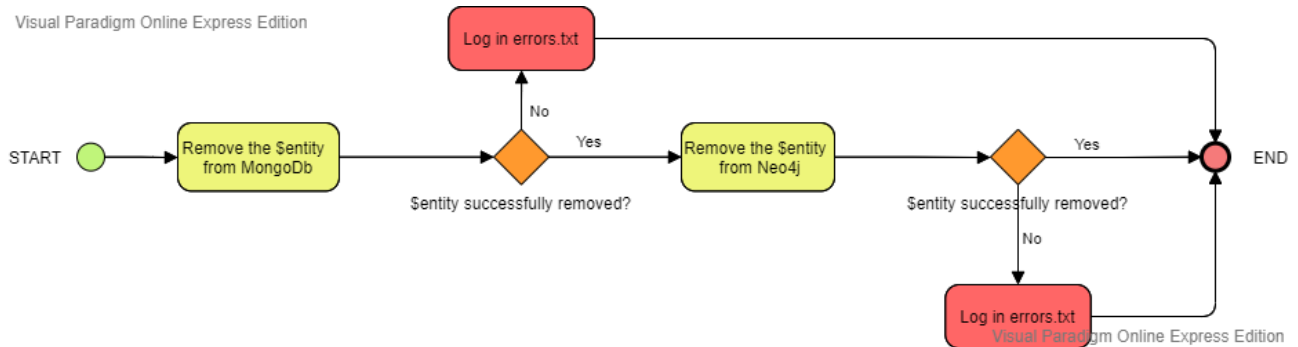
- ❖ **Considering the add \$entity**, if an error occurs with the first write operation, we just log the error in errors.txt file, while if an error occurs with the second write operation, we also try to delete the preceding write operation, beyond log the error.
- ❖ **Considering the remove \$entity**, we first remove the document from MongoDB, but if an error occurs in Neo4j we do not add again the document in Mongo, but we just log the error.
- ❖ **Considering the adding/remove like to a song**, we first remove the edge in Neo4j, but if an error occurs the second write operation in MongoDB we do not add again the edge in Neo4j, but we just log the error.

Administrators oversee manually check and **enforce consistency** where problem(s) have occurred to restore the consistency.

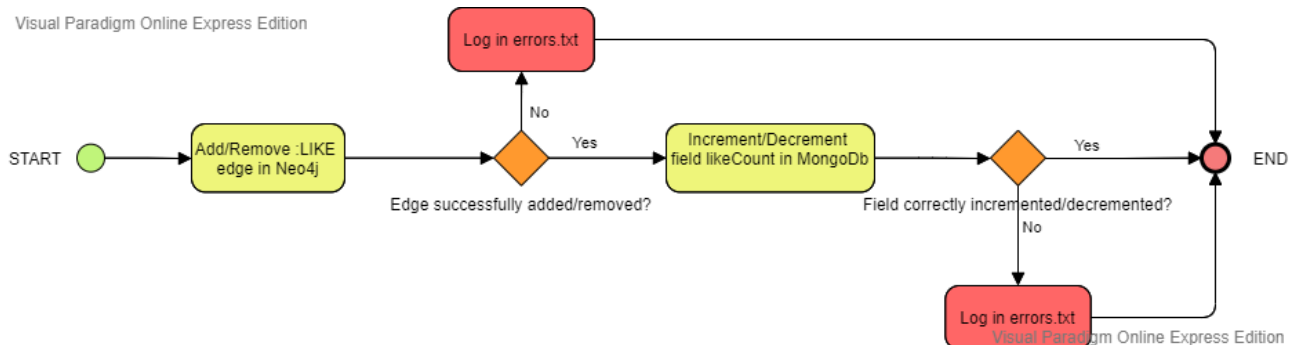
ADD AN \$ENTITY (Song, User or Playlist)



REMOVE AN \$ENTITY (Song, User or Playlist)



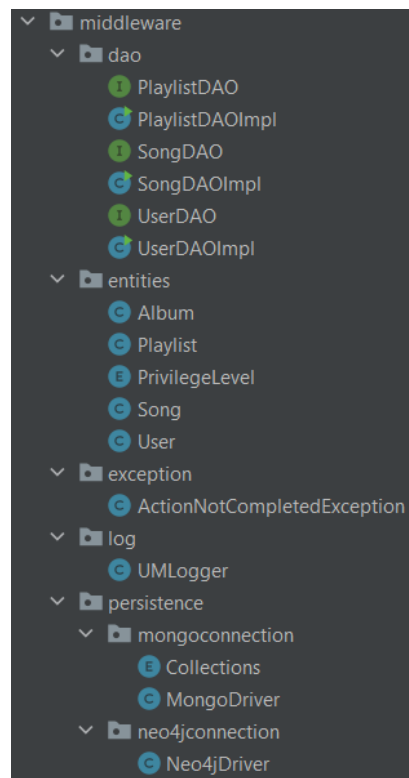
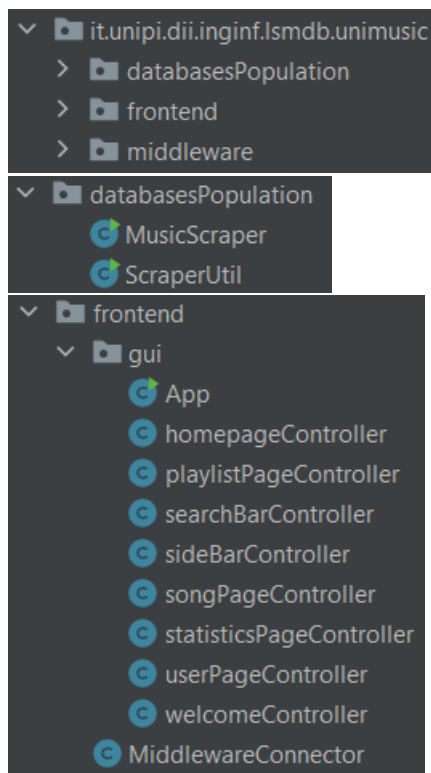
ADD/REMOVE A SONG'S LIKE



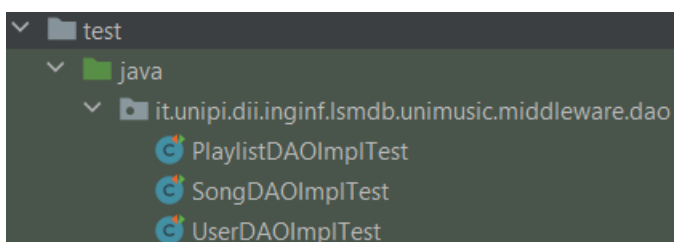
Application Package Structure

The final implementation of the Java application project consists of 3 main packages:

- **databasePopulation**: contains files used for the *Scraping*.
- **frontend**: contains all the files used to manage the *Graphical User Interfaces* and also the *MiddlewareConnector* file used to give an access interface to the underlying middleware logic.
- **middleware** contains:
 - o the logic needed to connect to used databases (**persistence** package).
 - o the logic needed to exploit such connection to perform operations required from the *frontend*. In particular, it contains all the *Model Objects (entities)*, the *Data Access Object Interfaces* and their *Concrete Implementations (dao)*.
 - o a *Logger* used to classify and maintain log information.



We also have done some **test unit**, that can be found in the following package:

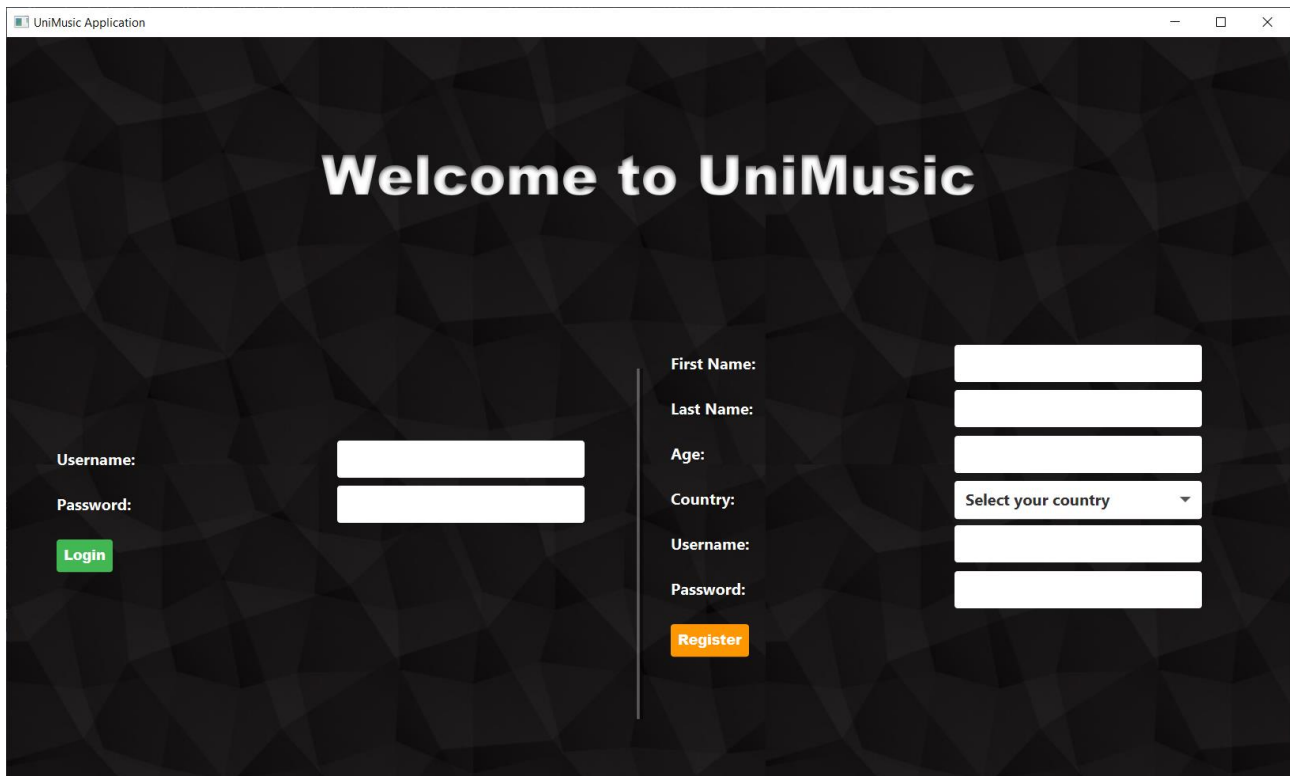


User Manual

Login and registration

Once the application is started, the first screen shown to the user contains the forms to **log in** and to **register a new account**.

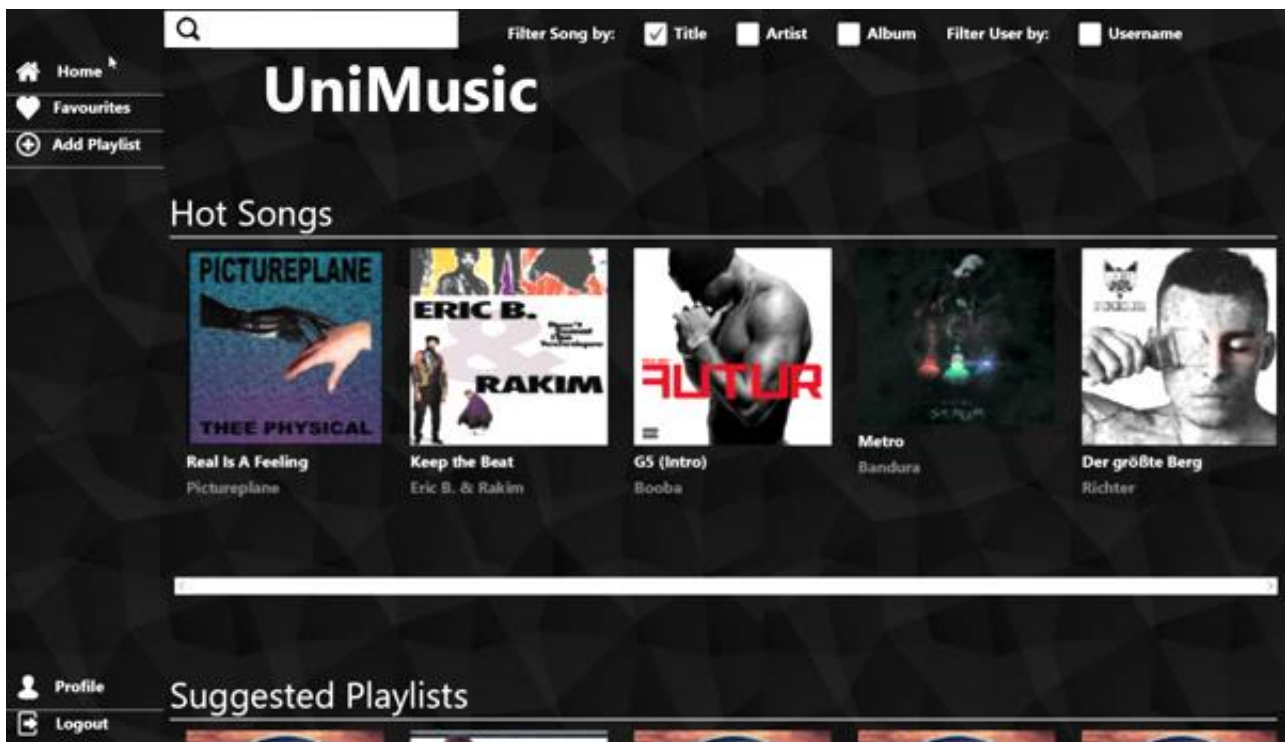
If a user accesses the application for the first time, he needs to register a new standard user account using the form on the right. After that, he can insert his credential on the left to log in.

The screenshot shows a web application window titled "UniMusic Application". The background is dark with a geometric pattern. In the center, the text "Welcome to UniMusic" is displayed in a large, white, sans-serif font. On the left side, there is a login form with two white input fields labeled "Username:" and "Password:". Below these fields is a green button with the text "Login" in white. On the right side, there is a registration form with five white input fields. The labels for these fields are "First Name:", "Last Name:", "Age:", "Country:", and "Username:". The "Country:" field is a dropdown menu with the text "Select your country" and a downward arrow. Below the registration fields is an orange button with the text "Register" in white.

Pic. 1 Log in and registration page

Home page

For a standard user, the homepage is the following:



Pic. 2 Homepage

Side bar

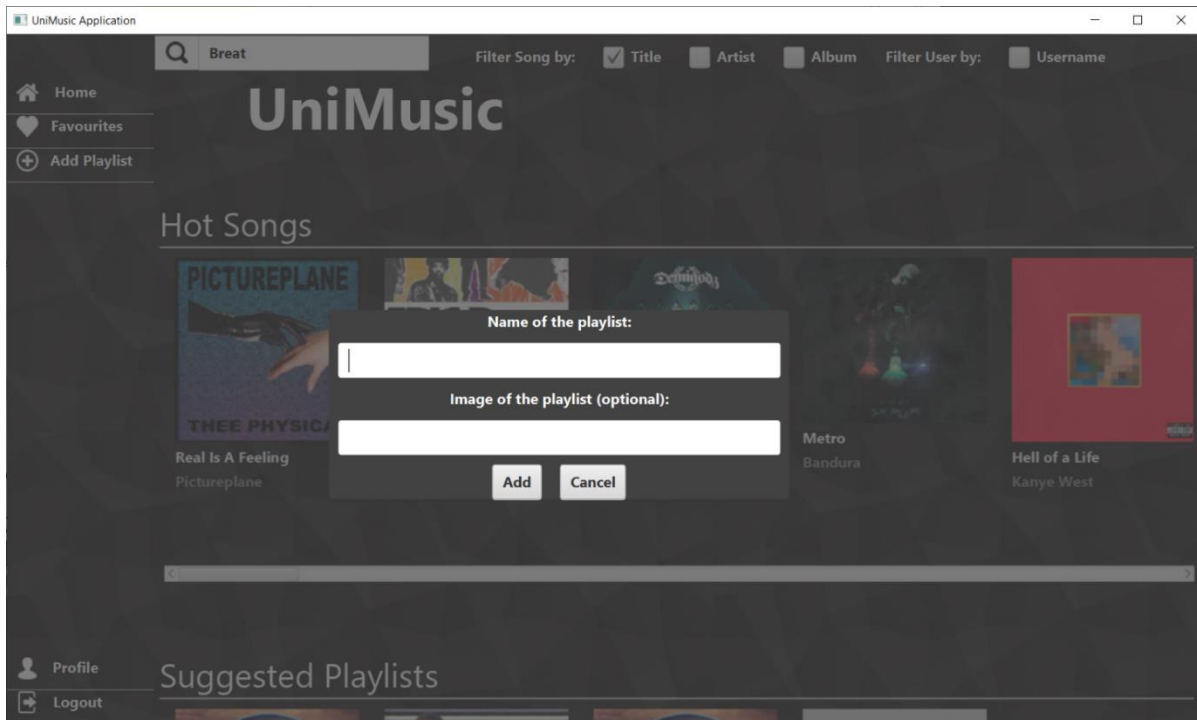
On the left there is a **side bar** (equal for every screen of the application), used by the user to:

- ❖ visualize its homepage (Home).
- ❖ visualize its playlist of favourite songs (Favourites).
- ❖ add a new playlist (Add Playlist).
- ❖ access to his own profile (Profile).
- ❖ log out and return to the login page (Logout).

Body

At the centre of the page there are shown:

- ❖ **hot songs**, i.e. the songs which received more like in the last month.
- ❖ **suggested playlists**, i.e. the playlist followed by our followed user or by the user followed by them.
- ❖ **suggested users**, i.e. users followed by our followed users and users which like the same songs as us.

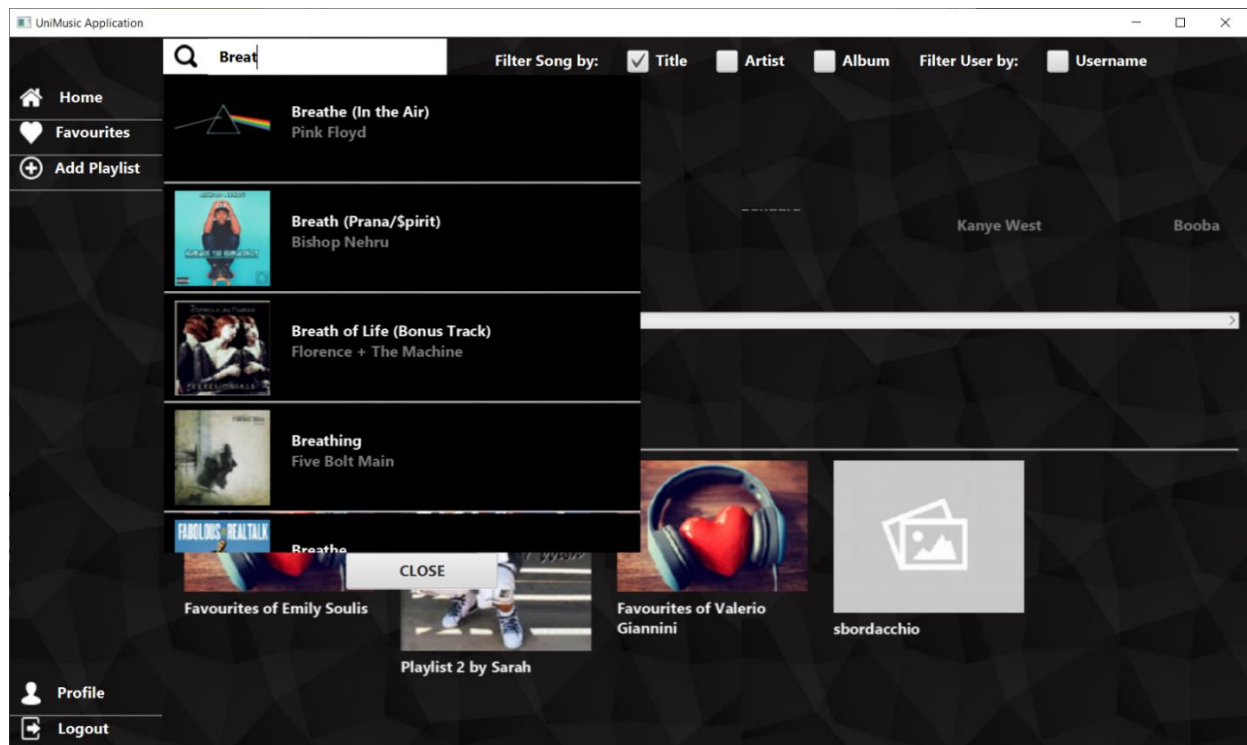


Pic. 3 Form to create a new Playlist

Search bar

At the top of the screen there is a search bar (also the same for all screens) where the user can search for **songs or users**.

The type of search is differentiated by the use of the **checkboxes**, the user has to insert the title of the song, the name of the artist or the name of the album to find songs depending on which box is checked between Title, Artist or Album, otherwise he can search for other users ticking the checkbox Username.

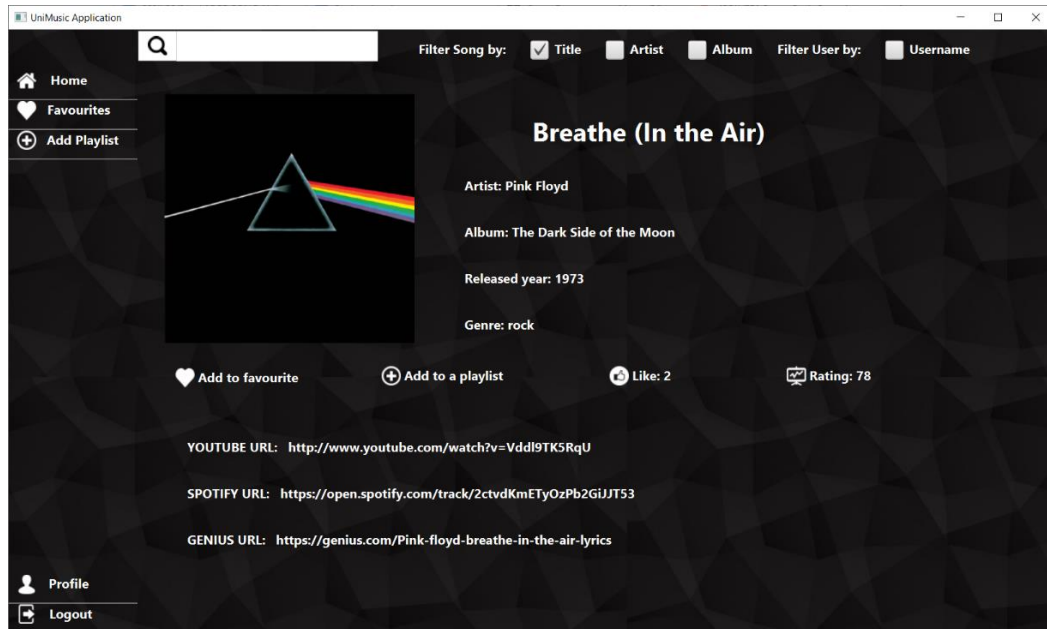


Pic. 4 Use of the search bar

Song page

The page of a song contains his information, buttons to add the song to favourites, to add song to a playlist and to like the song.

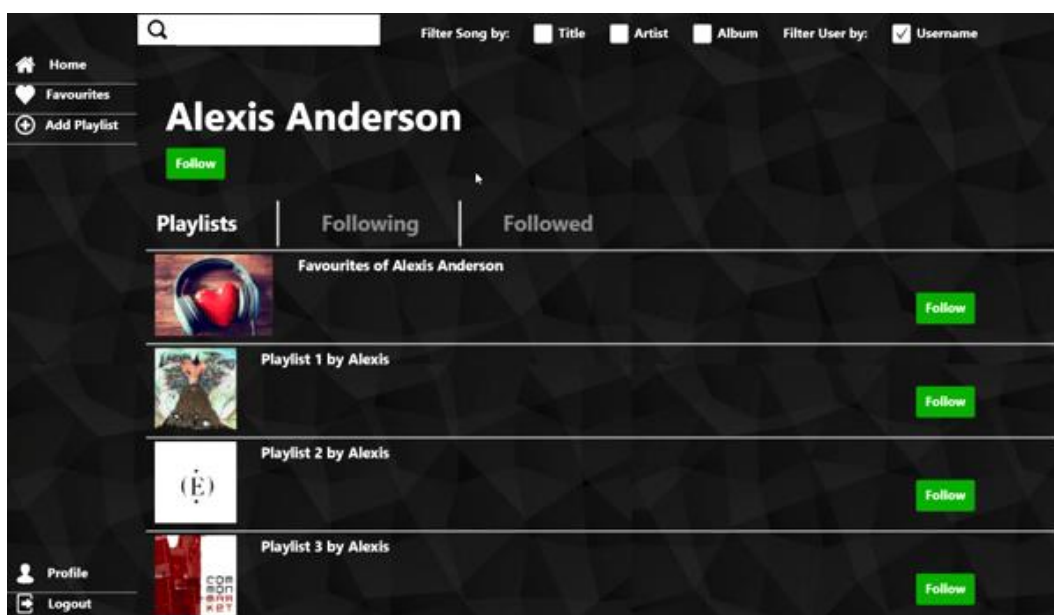
It is also shown a rating, based on the popularity on Spotify and the number of like and dislike on YouTube. Below there are the links to YouTube, Spotify and Genius, and by clicking on them the user can listen the song or view its video on his browser.



Pic. 5 Song page

User Page

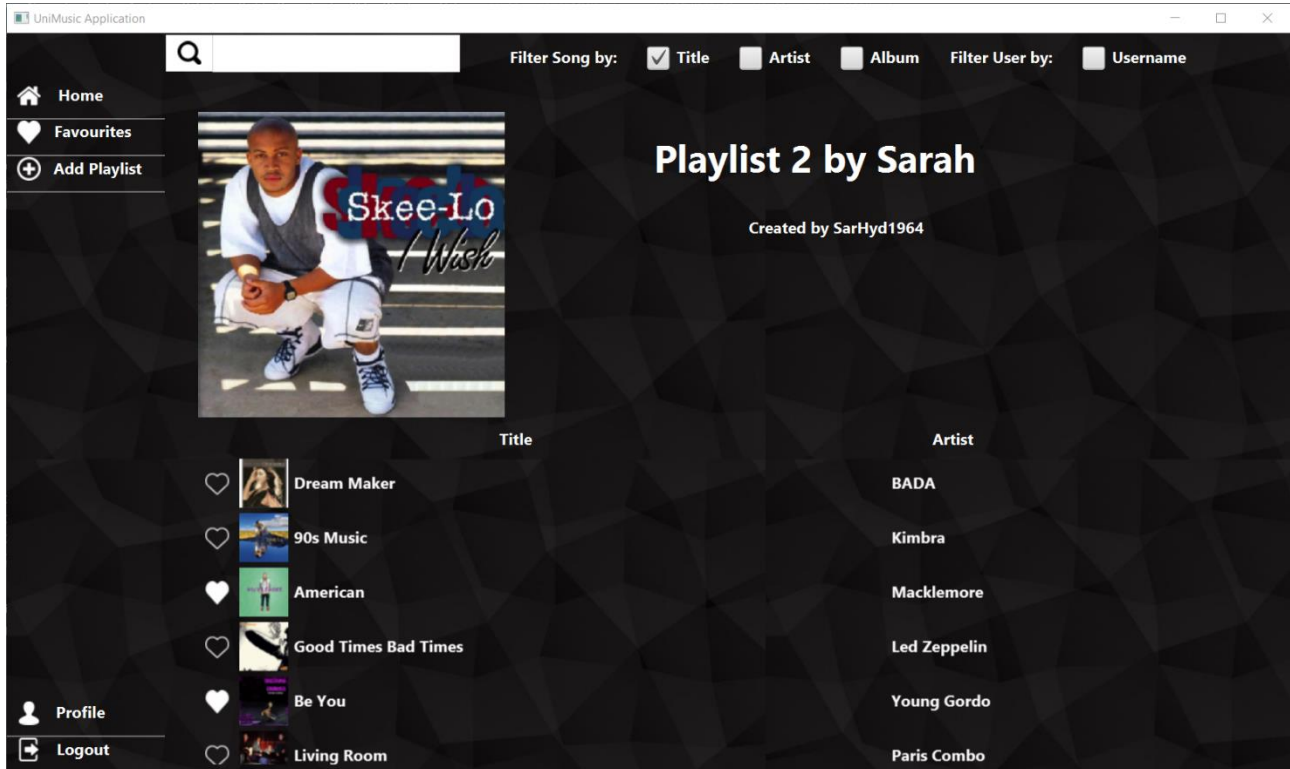
The page of a user gives the possibility to follow him, to see his playlists and follow them, and to see and follow the follower and the followed user of that profile. If the user shown correspond to the logged user, he has the possibility to delete his own profile.



Pic. 6 User page

Playlist Page

The playlist page shows the information of the playlist and all the songs it contains, and it gives the possibility to like them and to reach the correspondent song page. If the playlist is created by the logged user, he has the possibility to delete it.

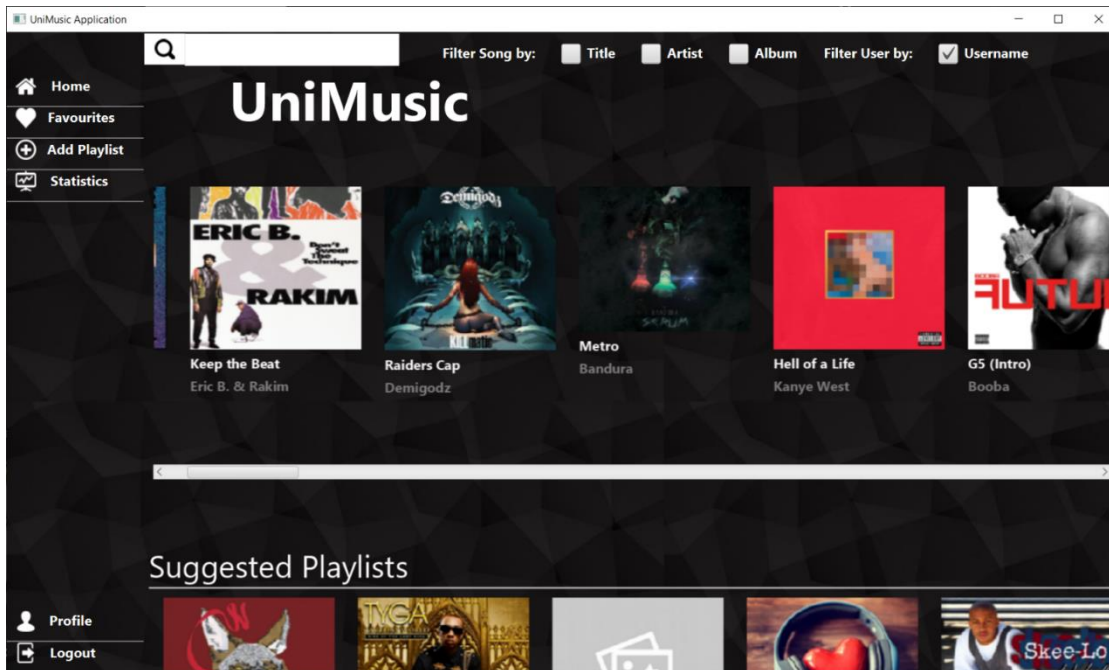


Pic. 7 Playlist page

Admin

If the credentials inserted in the login form are associated to an Administrator account, the user can access to some extra functionalities of the application.

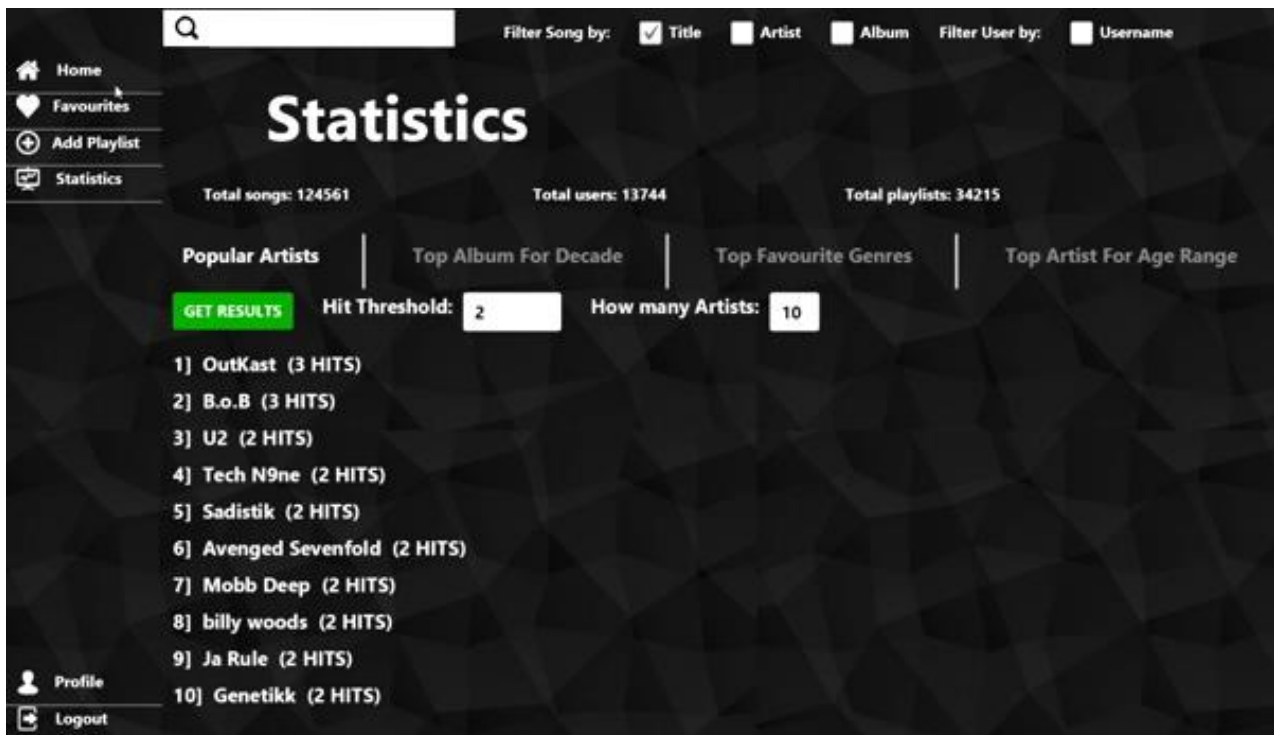
The sidebar of an administrator gives the possibility to access to the statistics of the application



Pic. 8 Homepage for an Administrator

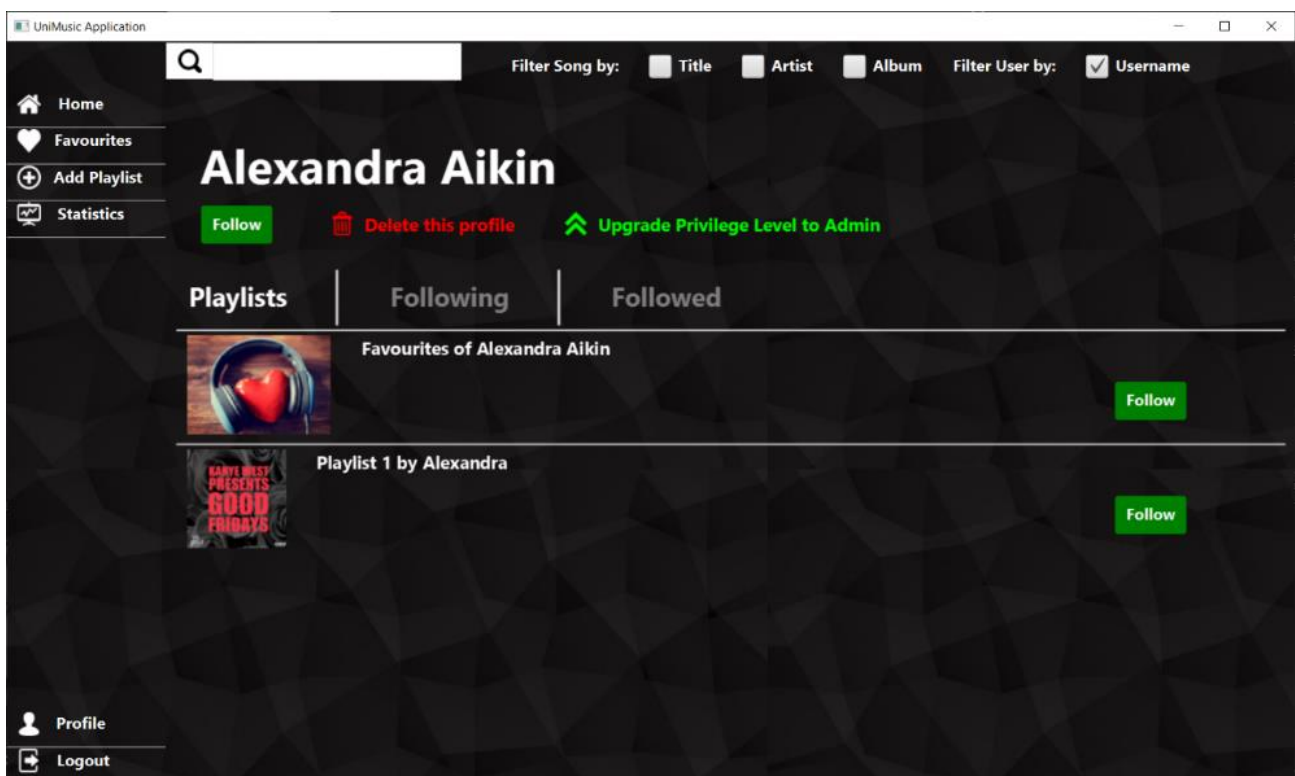
From the statistics page the admin can:

- Check the number of users, songs and playlists saved in the database.
- View the artists which made the higher number of hits (songs that received more than Hit Threshold likes, which is a customizable parameter for the admin).
- View the most rated albums for every decades.
- View the most present genres in playlists.
- View the favourite artists for every age range.



Pic. 9 Statistics page

If an admin goes to a user page, he has the possibility to delete his account and to update his privilege level to the Administrator level:



Pic. 10 User page for an Administrator