UNIVERSITÀ DI PISA

Computer Engineering

Large-Scale and Multi-Structured Databases

MyPodcastDB

Project Documentation

TEAM MEMBERS:
Biagio Cornacchia
Gianluca Gemini
Matteo Abaterusso

Academic Year: 2021/2022

GitHub Release: https://github.com/itm-unipi/MyPodcastDB/releases
GitHub Repository: https://github.com/itm-unipi/MyPodcastDB

Contents

1 Introduction

MyPodcastDB is a social network that allows to keep in touch on podcasts, view rankings about the top rated podcasts, the most followed authors and so on. A **podcast** is an episodic series of digital audio or video files. For each podcast, the application provides a detailed page that contains information such as episodes, duration, release date and reviews.

An **user** can like and review a podcast or add it to his watchlist. He can follow an author and follow other users. These activities will be exploited to provide suggestions to the user which can help him to find out new authors and podcasts.

On the other side an **author** can create, update and delete a podcast. He can use the application as the user but he can't follow users, write reviews, like a podcast or add it in a watchlist. He can follow other authors.

Admins can manage users, authors, podcasts and reviews. They also have access to the usage analytics of the application.

Thus, the application is meant to be used by four **main actors**: unregistered users, users, authors and administrators.

1.1 Functional Requirements

The functional requirements that the application provides, divided by the four main actors, are shown below.

1. **Unregistered User**
    - Sign-up
    - Browse top rated podcasts
    - Browse most liked podcasts
    - Browse most followed authors
    - Search podcasts and authors
    - Browse reviews associated to a podcast
2. **User**
    - Login and Logout
    - Browse top rated podcasts
    - Browse most liked podcasts
    - Browse most followed authors
    - Browse suggested authors based on followed users
    - Browse suggested podcasts based on followed users' likes
    - Browse suggested podcasts based on the authors in the user's watchlist
    - Search podcasts, authors and users

- Browse watchlist
- Browse liked podcasts
- Add a podcast to watchlist
- Remove a podcast from watchlist
- Like/Unlike a podcast
- Follow/Unfollow an author or user
- View own profile's information and update them
- Browse own followed users and authors
- Delete own account
- View podcast's information
- Browse reviews associated to a specific podcast
- Write a review to a specific podcast
- Delete own reviews
- View author's profile information and podcasts
- Browse author's followed authors
- View other user's profile information
- Browse other user's watchlist, liked podcasts, followed authors and followed users

3. **Author**
    - Login and Logout
    - Browse top rated podcasts
    - Browse most liked podcasts
    - Browse most followed authors
    - Search podcasts and authors
    - Follow/Unfollow an author
    - View own profile's information and update them
    - Browse own podcasts
    - Add/Remove a podcast
    - Delete own account
    - View own podcast's information and update them
    - Add/Remove podcast's episode
    - View other author's profile information and podcasts
    - Browse other author's followed authors

- View podcast's information
- Browse reviews associated to a specific podcast
- View user's profile information and browse user's watchlist, liked podcasts, followed authors and followed users

4. **Admin**
    - Login and Logout
    - Browse top rated podcasts
    - Browse most liked podcasts
    - Browse most followed authors
    - Search podcasts, authors and users
    - View author's profile information and podcasts and delete his account
    - View a specific podcast's information, delete an associated episode or delete the entire podcast
    - View user's profile information, browse user's watchlist, liked podcasts, followed authors and followed users and delete his account
    - Update own account information
    - Delete own account
    - Add a new admin
    - View analytics such as average age for favourite category, podcasts with highest number of reviews, country with highest number of podcasts, top favourite categories, most numerous category and most appreciated category
    - Update statistics

1.2 Non-Functional Requirements

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

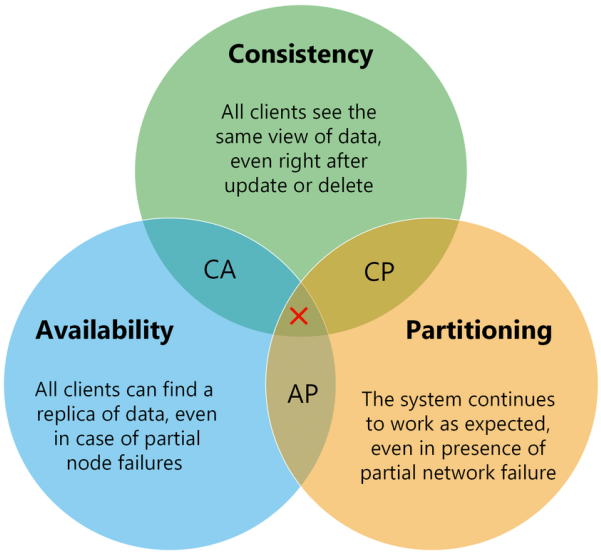
Product requirements:

- **Usability**, the application must be simple and user friendly
- **High availability**, the displayed data might not be always up to date
- **Low latency** in accessing the database to have a responsive application
- **Tolerance to the loss of data**, avoiding a single point of failure

Organizational requirement:

- When a user deletes his account, his **reviews** must be maintained

1.3 CAP Theorem



In order to satisfy the non-functional requirements it is reasonable to sacrifice consistency in favor of high-availability and partition tolerance. Thus, an **AP solution** is used.

2 Project Design

In this chapter will be described the project design choices made for the application such as use case, class diagram and the chosen databases.

2.1 Use case

Below it is possible to see the Use Case diagram extracted from the functional requirements.

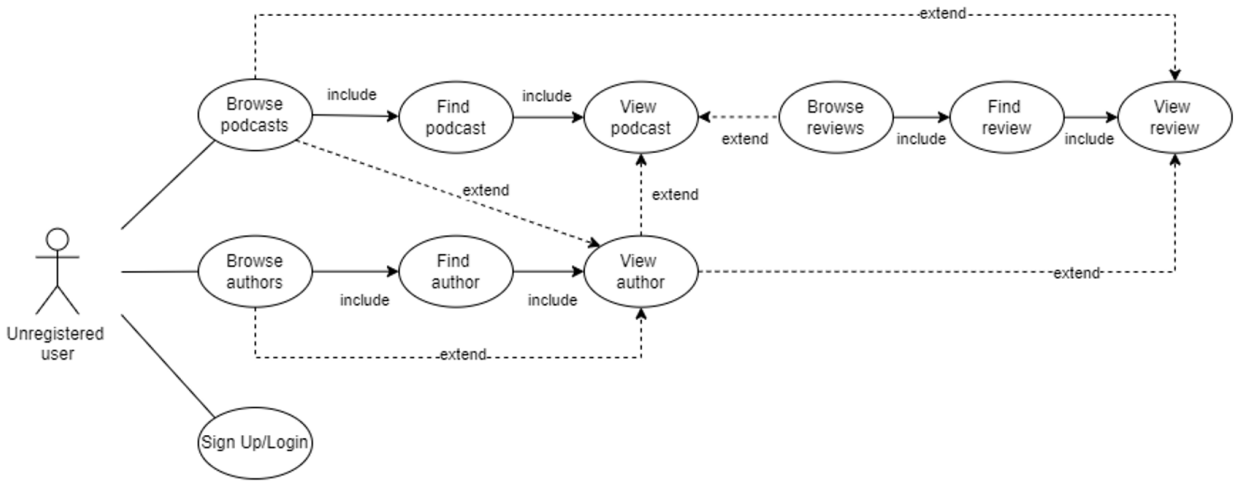


Figure 1: Unregistered user use case.

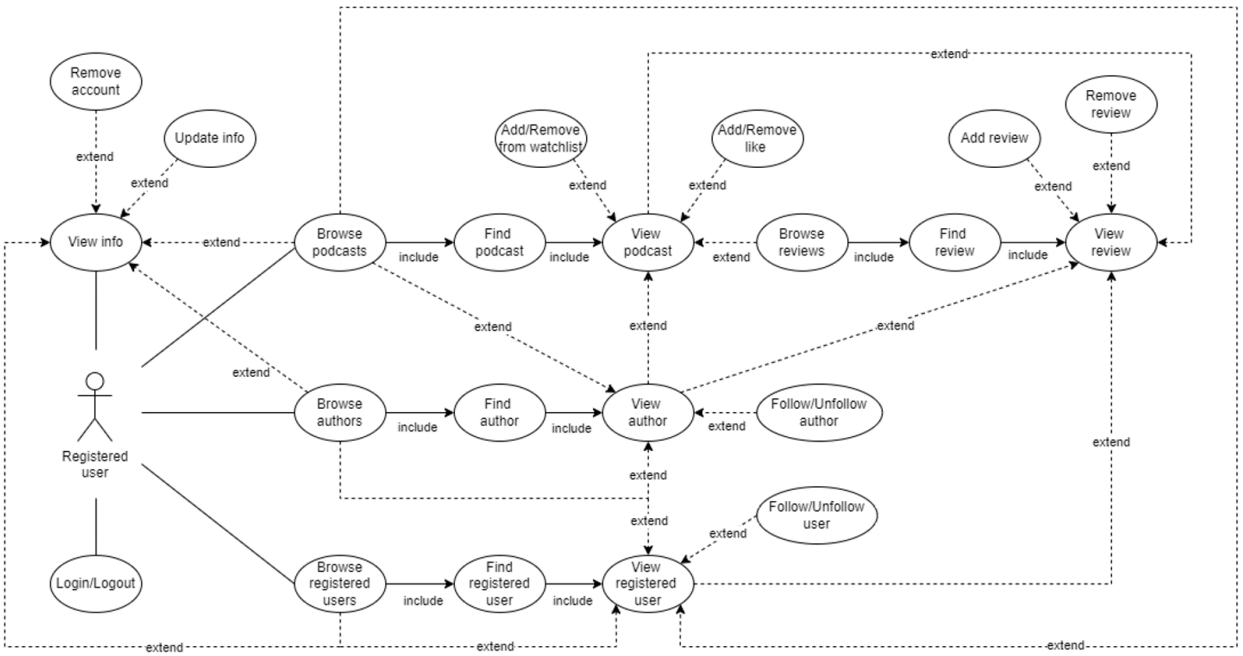


Figure 2: User use case.

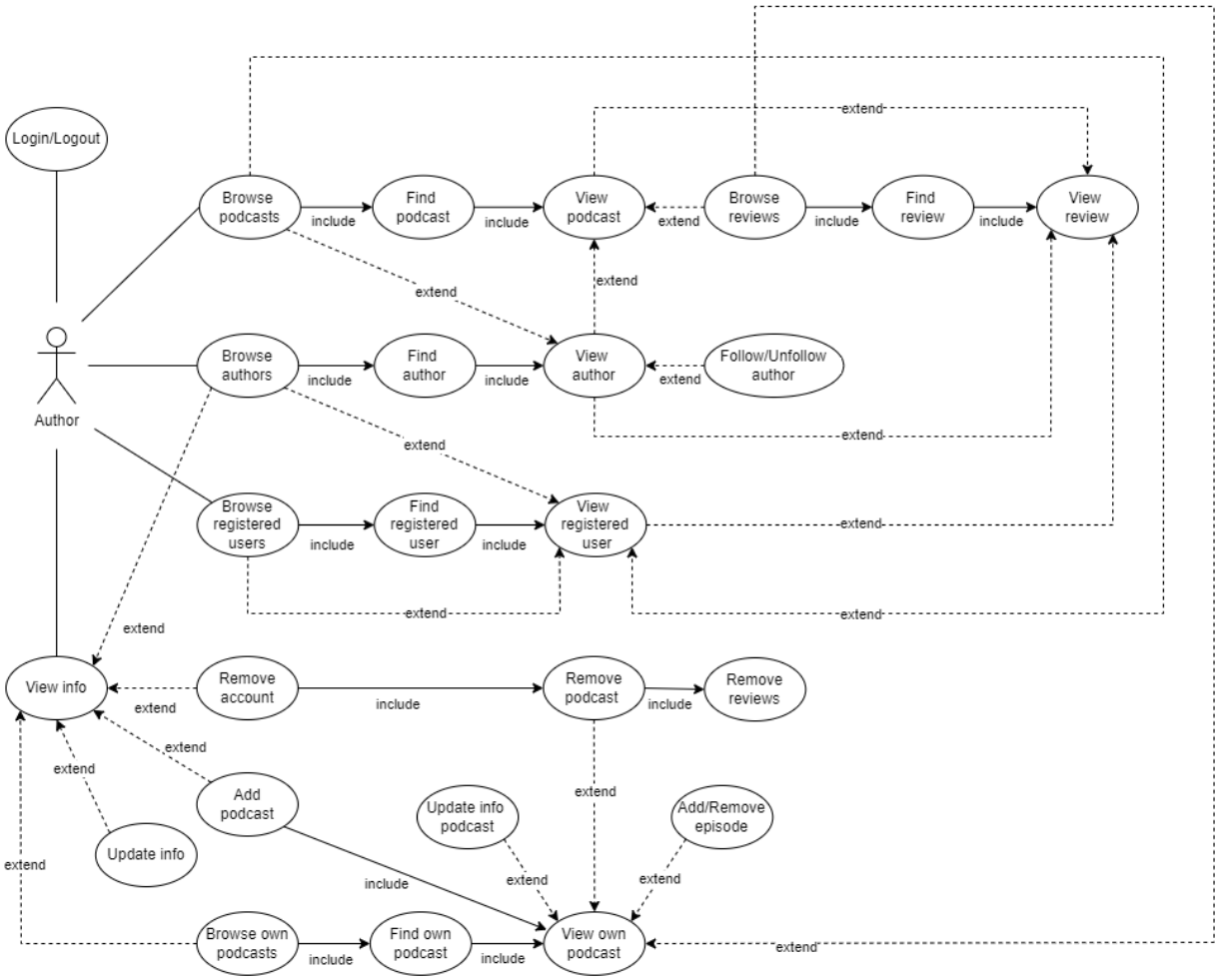


Figure 3: Author use case.

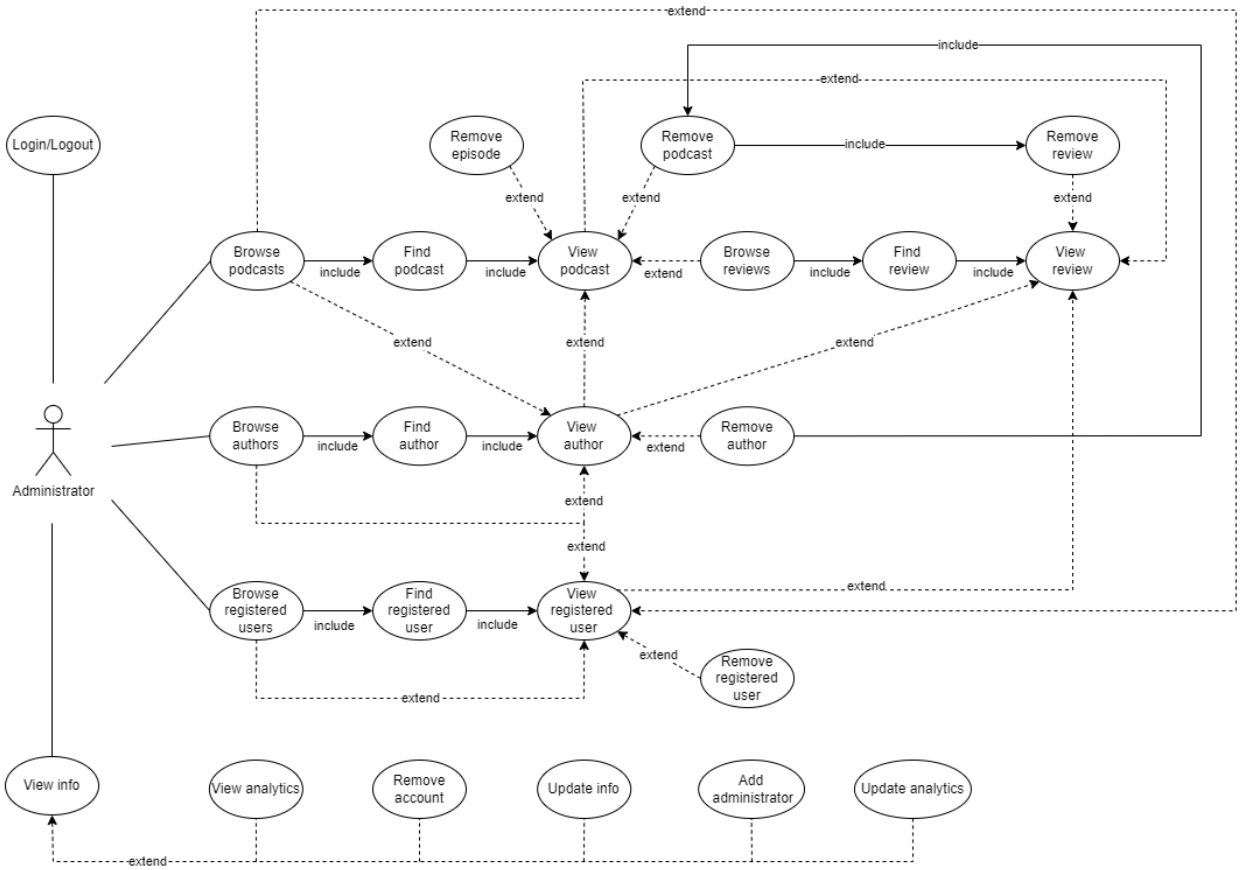


Figure 4: Administrator use case.

2.2 UML Class Diagram

Below it is possible to see the class diagram.

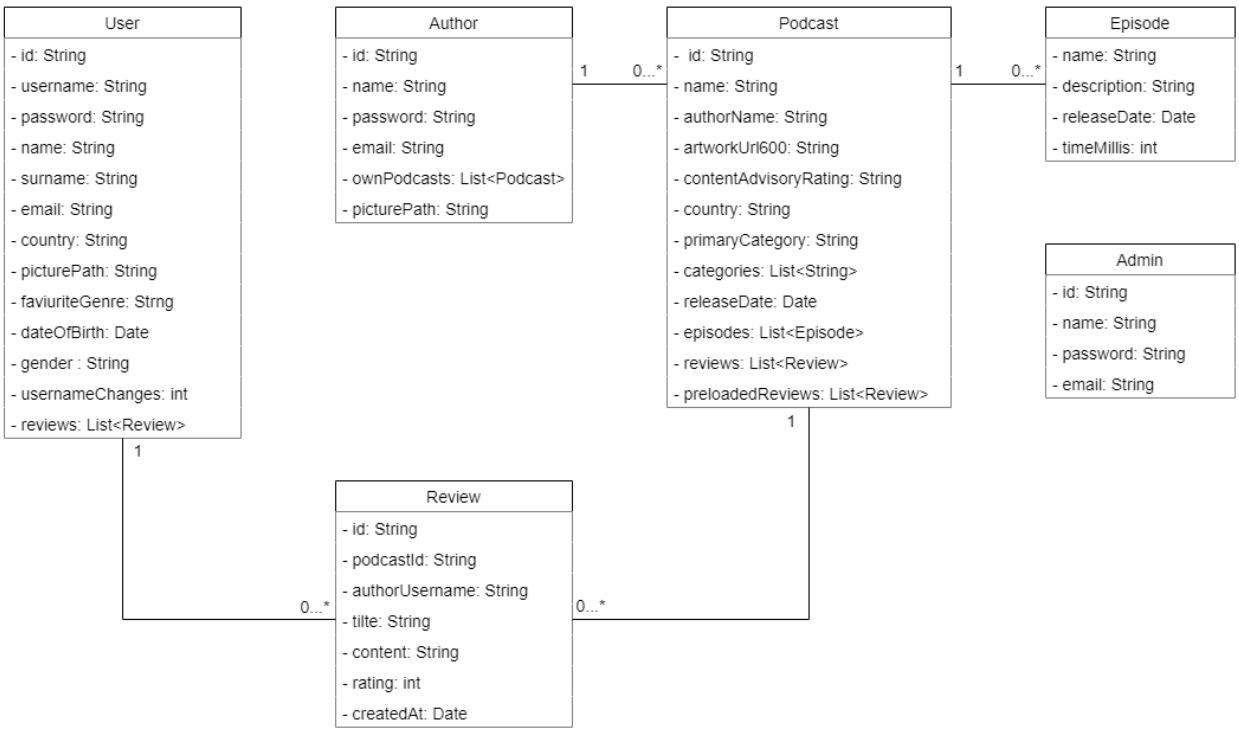


Figure 5: UML class diagram.

2.3 Database Choices

For the design of the project two types of databases have been exploited:

- **MongoDB** as document database in order to obtain low latency and high flexibility. The application can store for every entities a complex structure that contains all the information related to them. Finally, MongoDB allows to perform complex operations efficiently on these complex structures.
- **Neo4J** as graph database for representing relationships between the entities of the application such as follow, likes and so on. This kind of database allows to exploit these relations to extract suggestions about authors and podcasts.

2.4 Dataset Description

The dataset consists of real podcasts information and reviews taken from the following sources:

- https://www.kaggle.com/thoughtvector/podcastreviews (reviews in SQLite)
- https://itunes.apple.com (info podcasts in JSON)

Instead, all the users have been randomly generated through RandomUser API (https://ran-

domuser.me). The resulting size of JSON files containing the entities for MongoDB is:

- Users (116 MB)
- Authors (20.6 MB)
- Podcasts (414 MB)
- Reviews (168 MB)

Starting from these entities, the Neo4J relationships have been generated randomly.

The previous different sources have been used to build the dataset so this ensures the variety of the data.

3 MongoDB Design

3.1 Documents

In this section will be shown the structure of entities that compose the database.

3.1.1 Podcast Collection

This is an example of podcast document:

```
{
    "_id": {
      "$oid": "54eb342567c94dacfb2a3e50"
    },
    "podcastName": "Scaling Global",
    "authorName": "Slate Studios",
    "artworkUrl600": "https://preview-url.com/600x600bb.jpg",
    "contentAdvisoryRating": "Clean",
    "country": "Netherlands",
    "categories": [
      "Business"
    ],
    "primaryCategory": "Business",
    "releaseDate": {
      "$date": {
        "$numberLong": "1510635600000"
      }
    },
    "episodes": [...],
    "reviews": [...],
    "preloadedReviews": [...]
}
```

The **podcast** document has been designed in order to retrieve all its information in a single operation. In particular, each podcast consists of a **set of episodes** whose number is generally limited. Thus, for this one-to-many relationship it was decided to embed the episode in the podcast.

An example of an embedded episode's structure is the following:

```
{
    "episodeName": "Greener Pastures",
    "episodeDescription": "Hear Greiner USA President, David..",
    "episodeReleaseDate": {
        "$date": {
          "$numberLong": "1510635600000"
        }
    },
    "episodeTimeMillis": 1450000
}
```

On the other hand, the number of **reviews** can be unlimited and can grow over time. Therefore, putting the entire reviews into podcasts can cause problems of memory reallocation or problems related to the document size limit. So it was decided to use linking documents so that all reviews are in a dedicated collection. Therefore, the *id* and the *rating* associated to them were embedded in the *review* field of podcast's document. In this way it is possible to calculate the **average rating** of the podcast without making any additional queries.

An example of embedded review's structure is the following:

```
{
    "reviewId": {
        "$oid": "000000000000000000080116"
    },
    "rating": 5
}
```

Finally, the reviews are shown in their page in groups of ten. For this reason it was decided to embed the ten most recent reviews in the *preloaded reviews* field of the podcast document. In this way when the reviews page is loaded, there will be no need for additional queries. If the user requests additional reviews, they will be taken from the reviews collection through the *ids* embedded in reviews field of the podcast document.

So, the field preloaded reviews contains document like the following one:


```
{
    "_id": {
        "$oid": "000000000000000000080116"
    },
    "title": "Good Sernons",
    "content": "I'm a regular listener.  I only wish that pastor...",
    "rating": 5,
    "createdAt": {
        "$date": {
          "$numberLong": "1570533812000"
        }
    },
    "authorUsername": "beautifulzebra19914"
}
```


3.1.2 Review Collection

This is an example of review document:


```
{
    "_id": {
        "$oid": "000000000000000000000007"
    },
    "podcastId": {
        "$oid": "17aab2d100b3da0db0d22377"
    },
    "title": "Love this podcast",
    "content": "These guys are brilliant. I love listening...",
    "rating": 5,
    "createdAt": {
        "$date": {
            "$numberLong": "1575315120000"
        }
    },
    "authorUsername": "smallmeercat795128"
}
```

3.1.3 User Collection

This is an example of user document:


```
{
    "_id": {
        "$oid": "62aa5542b1f6097394a56ce9"
    },
    "username": "ticklishcat734546",
    "password": "scruffy",
    "name": "Sônia",
    "surname": "Pinto",
    "email": "ticklishcat734546@example.com",
    "country": "Brazil",
    "favouriteGenre": "Astronomy",
    "age": 28,
    "gender": "female",
    "picturePath": "/img/users/user21.png",
    "dateOfBirth": {
        "$date": {
            "$numberLong": "730598400000"
        }
    },
    "usernameChanges": 0,
    "reviews": [...]
}
```


When an **user** visits a podcast's review page, the application must be able to know whether the user has issued a review. Therefore, for each review created by the user, a document is inserted within the *reviews* field of the user's document containing the *id* associated to that review and the *id* of the corresponding podcast. Otherwise, it would be necessary to query the review collection using two foreign keys (the *podcast id* and *username*) but this looks like a relational approach. The choice to embed incomplete reviews is due to avoid the overgrowth of the document.

An example of the document in the reviews field is the following:


```
{
    "reviewId": {
        "$oid": "000000000000000000000001"
    },
    "podcastId": {
        "$oid": "c9cd8c0d5a9f1752ee1982c7"
    }
}
```

3.1.4 Author collection

This is an example of user document:

```
{
    "_id": {
        "$oid": "000000000000000000000000"
    },
    "name": "Michael Colosi",
    "email": "michaelcolosi@example.com",
    "password": "cfcd208495d565ef66e7dff9f98764da",
    "podcasts": [...],
    "picturePath": "/img/authors/author16.png"
}
```

The **author** document has been designed in order to retrieve all its information in a single operation. In particular, in the author page the application shows all the podcasts associated to that author. Thus, for this one-to-many relationship it was decided to embed only the necessary information to show a preview of the podcast.

An example of the podcast embedded in the author's document is the following:

```
{
    "podcastId": {
        "$oid": "6a70d8d5ffcc27889ba41086"
    },
    "podcastName": "Salon and Spa Marketing Toolkit",
    "podcastReleaseDate": {
        "$date": {
            "$numberLong": "1326631620000"
        }
    },
    "category": "Business",
    "artworkUrl600": "https://preview-url.com/600x600bb.jpg"
}
```

3.1.5 Admin Collection

This is an example of admin document:

```
{
    "_id": {
        "$oid": "62a0bf4e5958159cb9d7d080"
    },
    "name": "admin",
    "password": "admin",
    "email": "admin@example.com"
}
```

3.1.6 Query Collection

The application requires to load a set of **general statistics** used to suggest new authors and podcasts. The calculation of these statistics is too **time-consuming**, resulting in an application not responsive enough. So a good compromise is to give to the admin the task to perform the statistics update on demand, storing the results in a special collection called *query*. In this way the **precomputed** statistics will be retrieved directly from the collection.

This is an example of query document:

```
{
    "_id": {
        "$oid": "62a36ca9e4f327d91cbe219c"
    },
    "queryName": "MostLikedPodcast",
    "lastUpdate": {
        "$date": {
            "$numberLong": "1655376817142"
        }
    },
    "results": [...]
}
```

In this case the *results* field contains a set of podcasts ordered by the number of likes. The document's structure of the podcasts is the following:

```
{
    "podcastId": {
      "$oid": "4c59e540405a9272a10b81b9"
    },
    "podcastName": "Eduardo Caminos' Podcast",
    "podcastArtwork": "https://preview-url.com/600x600bb.jpg",
    "numLikes": 185
}
```

3.2 Indexes Analysis

In this section will be analyzed the indexes chosen for each collection in Mongo.

3.2.1 Podcast

The critical operations made on the podcast's collection are the search by *podcast name* and the update of podcast's *author name*. So the chosen indexes are the following:

Attribute	Index Type	Properties
podcastName	Single	Text
authorName	Single	1

The performance gains obtained adopting these two indexes are shown below:

	AuthorName
Without Index	
executionTimeMillis	933
totalKeysExamined	0
totalDocsExamined	39294
nReturned	39
With Index	
executionTimeMillis	7
totalKeysExamined	39
totalDocsExamined	39
nReturned	39

3.2.2 Review

The critical operations made on review's collection are delete by *podcast id* and update by *author name*. Thus, the chosen indexes are the following:

Attribute	**Index Type**	**Properties**
podcastId	Single	1
authorName	Single	1

The differences obtained using the indexes can be seen in the following table:

	PodcastId	**AuthorName**
Without Index		
executionTimeMillis	437	388
totalKeysExamined	0	0
totalDocsExamined	333501	333501
nReturned	33	1
With Index		
executionTimeMillis	8	1

totalKeysExamined	33	1
totalDocsExamined	33	1
nReturned	33	1

3.2.3 User

The *username* and *email* have to be unique. Moreover, the username is used to perform the search and all the CRUD operations. So, the chosen indexes are the following:

Attribute	**Index Type**	**Properties**
username	Single	Unique, 1
username	Single	Text
email	Single	Unique, 1

The performance gains obtained adopting the two unique indexes are shown below:

	Username	**Email**
Without Index		
executionTimeMillis	466	222
totalKeysExamined	0	0
totalDocsExamined	250000	250000
nReturned	1	1
With Index		
executionTimeMillis	1	1
totalKeysExamined	1	1
totalDocsExamined	1	1
nReturned	1	1

3.2.4 Author

The *author name* and *email* have to be unique. Moreover, the author name is used to perform the search and all the CRUD operations. So, the chosen indexes are the following:

Attribute	**Index Type**	**Properties**
name	Single	Unique, 1

name	Single	Text
email	Single	Unique, 1

The performance gains obtained adopting the two unique indexes are shown below:

	Name	**Email**
Without Index		
executionTimeMillis	45	38
totalKeysExamined	0	0
totalDocsExamined	36303	36303
nReturned	1	1
With Index		
executionTimeMillis	1	1
totalKeysExamined	1	1
totalDocsExamined	1	1
nReturned	1	1

3.2.5 Admin

The *admin name* and *email* have to be unique. This implies to have the following indexes:

Attribute	**Index Type**	**Properties**
name	Single	Unique, 1
email	Single	Unique, 1

3.2.6 Query

The *query name* has to be unique. This implies to have the following index:

Attribute	**Index Type**	**Properties**
queryName	Single	Unique, 1

3.3 Clustering

The available machines for hosting MongoDB are three. The configuration adopted for the replica set is the following:

```
rsconf = {
    _id: "mypodcastdb",
    members: [
        {
            _id: 0,
            host: "172.16.4.226:27020",
            priority: 1
        }, {
            _id: 1,
            host: "172.16.4.227:27020",
            priority: 2
        }, {
            _id: 2,
            host: "172.16.4.228:27020",
            priority: 5
        }
    ]
};
```

With this configuration, the VM 172.16.4.228 has the highest priority so it will be elected as primary. In order to implement a **ready-heavy AP application**, the *write concern* and *read preference* configurations are the following:

```
mongodb://admin:password@172.16.4.226:27020,172.16.4.227:27020,
172.16.4.228:27020/?replicaSet=mypodcastdb&w=1&readPreference=nearest
&retryWrites=true&ssl=false
```

The choice of set write concern to *1* and read preference to *nearest* is due to the fact that read operations are more frequent than the write operations. In this way it is possible to ensure **fast responses** and **eventual consistency** requirement.

3.4 Sharding Proposal

To guarantee **availability** and **fast responses**, the sharding proposal for the application uses the following fields as sharding keys:

- The podcast id for the *podcast* collection
- The review id for the *review* collection
- The username for the *user* collection
- The author name for the *author* collection
- The admin name for the *admin* collection
- The query name for the *query* collection

These fields have been chosen because they are the most used ones in the CRUD operations.

Regarding the partition method, it has been decided to adopt an **hashing strategy** in order to distribute in an homogeneous way the documents.

4 Neo4J Design

4.1 Nodes

The main nodes that compose Neo4J are the following:

- **Podcast**, which represents a podcast. It contains the *podcast id* (that corresponds to the MongoDB's *_id*), the *podcast name* and the *artwork url* that are the properties needed to represent the podcast's preview on the GUI.
- **User**, which represents the registered user of the application. It contains the *username* and the *picture path* that are the properties needed to represent the user's preview on the GUI.
- **Author**, which represents an author. It contains the *author name* and the *picture path* that are the properties needed to represent the author's preview on the GUI.
- **Category**, which represents one of the category associated to a podcast. It contains just the *category name* property.

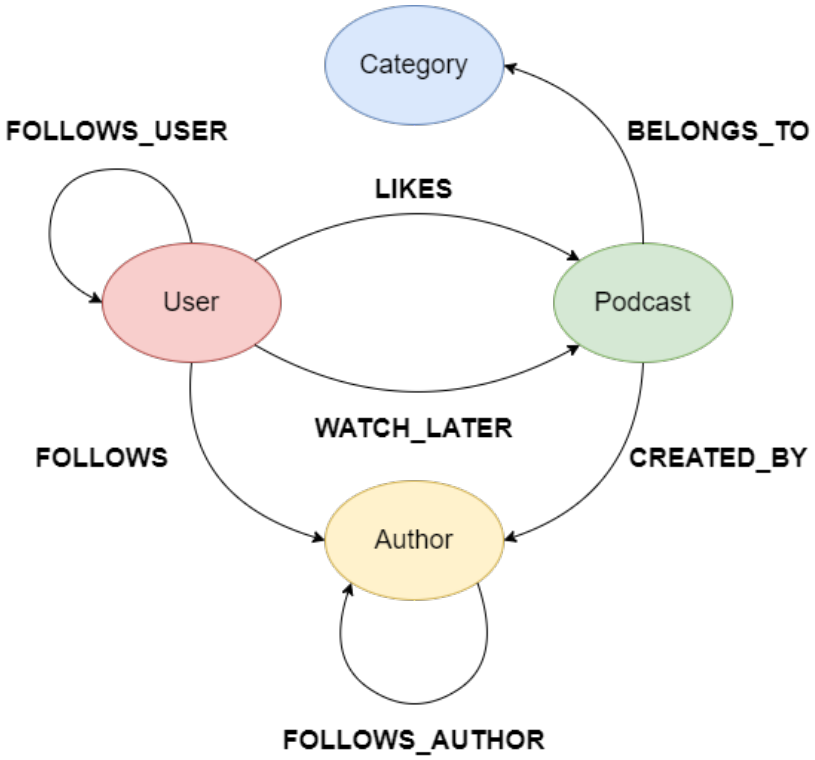
The nodes have been organized in such a way that a result from any query can always be represented as list of summarized entities on the GUI. If the user then wants to obtain additional information about a specific entity, that information will be taken from MongoDB by clicking on the corresponding preview.

4.2 Relations

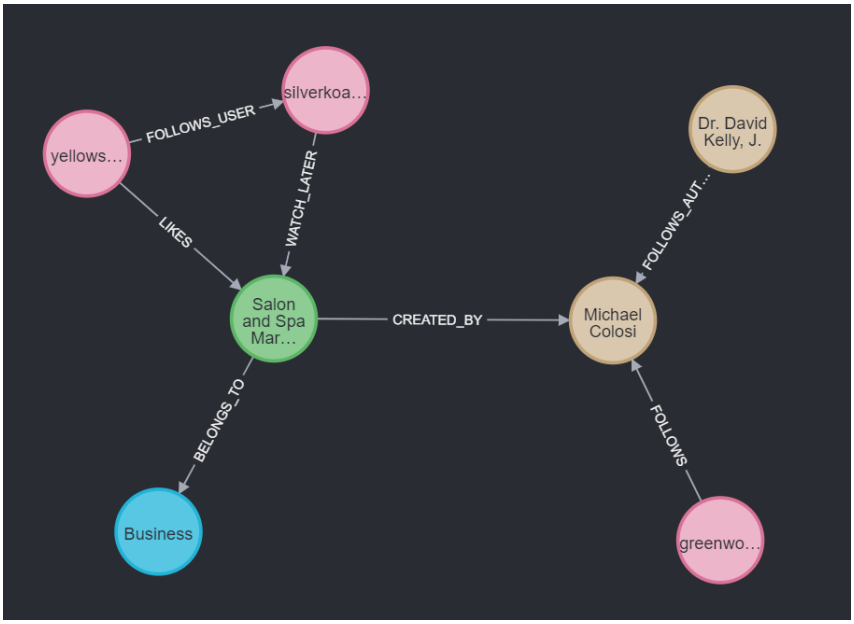
Seven different kinds of relationships between nodes are used within the database:

- **Podcast** → **CREATED_BY** → **Author**, which represents a belonging relationship between a podcast and his author and is created/removed when an author creates/removes a podcast. This relationship has no attributes.
- **Podcast** → **BELONGS_TO** → **Category**, which represents a belonging relationship between a podcast and one of its categories and is created/removed when an author creates/updates/removes a podcast. This relationship has no attributes.
- **User** → **LIKES** → **Podcast**, which represents a user that likes a podcast. This relationship has no attributes.
- **User** → **WATCH_LATER** → **Podcast**, which represents a user that adds to his watchlater list a podcast. This relationship has no attributes.
- **User** → **FOLLOWS** → **Author**, which represents a user that follows an author. This relationship has no attributes.
- **User** → **FOLLOWS_USER** → **User**, which represents a user that follows a user. This relationship has no attributes.
- **Author** → **FOLLOWS_AUTHOR** → **Author**, which represents an author that follows another author. This relationship has no attributes.

Each of these relationships has been designed in order to obtain **general suggestions** (i.e most liked podcasts) and **more specific ones** (i.e suggest podcasts based on user's watchlist).



A snapshot of the graph from Neo4J is shown below:



4.3 Indexes Analysis

The indexes chosen for Neo4J are:

- *Podcast id* for the podcast node
- *Username* for the user node
- *Author name* for the author node

- *Category name* for the category node

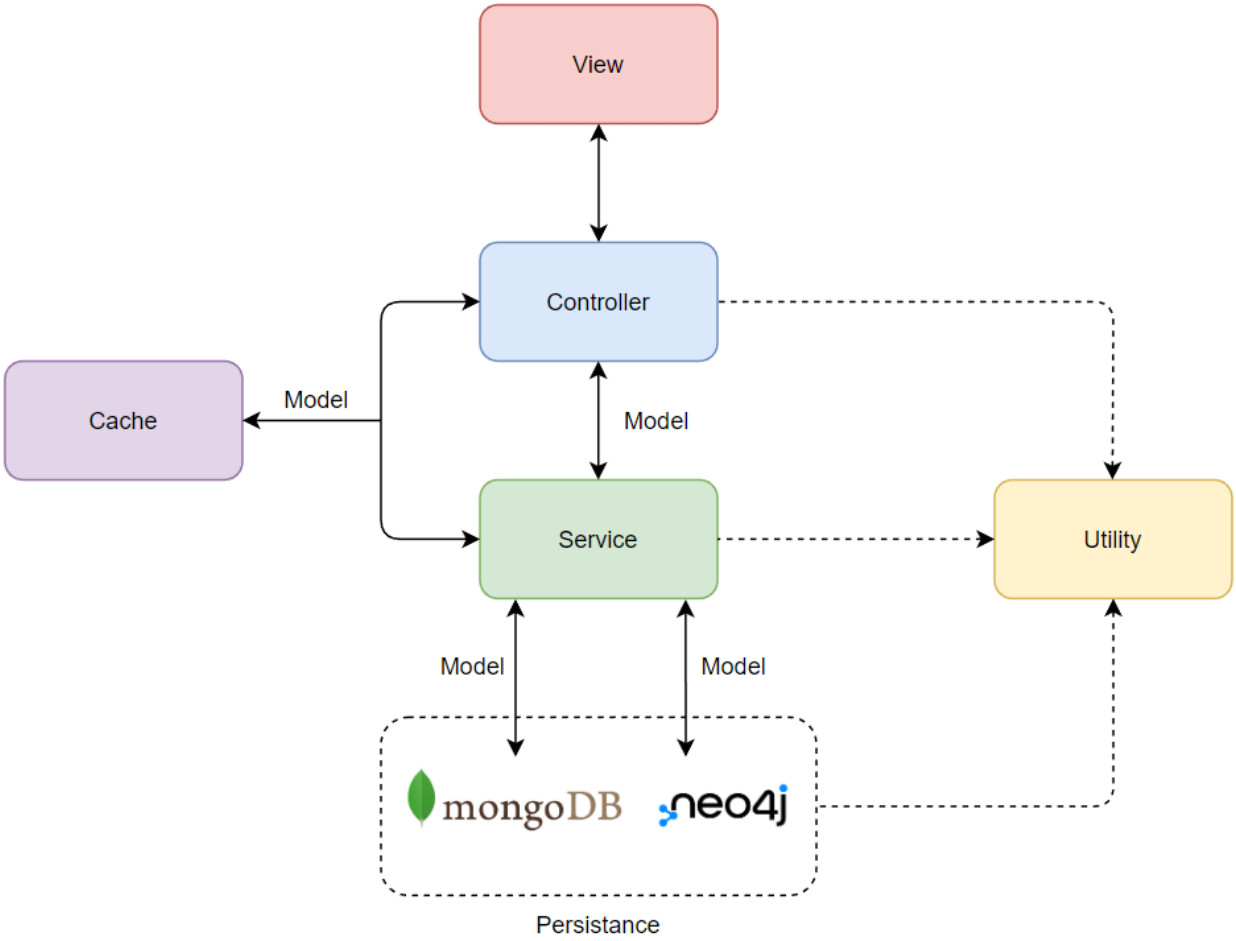
These are the performance gains obtained for a find operation:

	Author	**Podcast**	**User**	**Category**
Without Index				
executionTime	212ms	191ms	315ms	52ms
With Index				
executionTime	42ms	49ms	56ms	45ms

5 Implementation

5.1 Package Architecture

The chosen package architecture is a **layered architecture**. In particular, it has been used a *model-view-controller* pattern with a layer *service* that abstracts the communication with the databases.



More specifically, the packages are:

- **Model**: contains an abstraction for all the entities such as podcast, author, etc.
- **View**: contains the GUI used by the user to interact with the application.
- **Controller**: takes care of the user's requests and shows the results updating the views.
- **Service**: each service implements an operation issued by the controller. In particular, one operation is composed by a series of find, add, delete or update requests on the persistence layer. It also manages the connections with both databases.
- **Persistence**: is composed by two sub-packages, **mongo** and **neo4j**. It contains all the methods used to communicate with the databases.

- **Cache**: consists of hash tables used to store in memory the application's status in order to avoid unnecessary requests to the databases.
- **Utility**: contains useful operations for the application such as logging, configuration loading, etc.

5.2 Modules

In this section will be briefly described the modules that compose the application.

5.2.1 it.unipi.dii.lsmsdb.myPodcastDB

Name	Description
Main	Main class of the application. It initializes the image cache and the stage manager. Then, it launches the instance MyPodcastDB class.
MyPodcastDB	During its creation, it initializes the logger and the config manager. At runtime it manages objects' session.

5.2.2 it.unipi.dii.lsmsdb.myPodcastDB.model

Name	Description
Admin	Bean class that represents an admin of the application.
Author	Bean class that represents an author of the application.
Episode	Bean class that represents an podcast's episode of the application.
Podcast	Bean class that represents a podcast of the application.
Review	Bean class that represents a review of the application.
User	Bean class that represents a registered user of the application.

5.2.3 it.unipi.dii.lsmsdb.myPodcastDB.view

Name	Description
DialogManager	Utility class used to create pop-up messages inside the application.
StageManager	Class that manages the loading and the communication between the different views.
ViewNavigator	It is an enumerate that contains all the possible views of the application. For each of them, it returns the fxml file name.

5.2.4 it.unipi.dii.lsmsdb.myPodcastDB.controller

Name	Description
ActorPreviewController	Controller associated to ActorPreview.fxml. It is used to show a preview of an actor (author or user) in the user page.
AddAdminController	Controller associated to AddAdmin.fxml. It is used to create a new admin.
AddPodcastController	Controller associated to AddPodcast.fxml. It is used to create a new podcast.
AdminDashboardController	Controller associated to AdminDashboard.fxml. It provides all the main functionalities used by the administrators.
AuthorPreviewController	Controller associated to AuthorPreview.fxml. It is used to show a preview of the author in the homepage and author's profile.
AuthorProfileController	Controller associated to AuthorProfile.fxml. It provides all the main functionalities used by the authors.
AuthorReducedPodcastController	Controller associated to AuthorReducedPodcast.fxml. It is used to show a preview of the podcast in the author's profile.
AuthorSearchPreviewController	Controller associated to AuthorSearchPreview.fxml. It is used to show a preview of the author in the search page.
AuthorSettingsController	Controller associated to AuthorSettings.fxml. It is used to update the author's personal information.
BarChartController	Controller associated to BarChart.fxml. It allows to create a bar chart with custom data in the admin dashboard.
EpisodeController	Controller associated to Episode.fxml. It is used to show an episode on the podcast page.
EpisodeEditController	Controller associated to EpisodeEdit.fxml. It is used to update a podcast's episode.
HomePageController	Controller associated to HomePage.fxml. It is used to manage the homepage of the application.

LoginController	Controller associated to Login.fxml. It is used to manage the login of the different actors of the application.
PieChartController	Controller associated to PieChart.fxml. It allows to create a pie chart with custom data in the admin dashboard.
PodcastPageController	Controller associated to PodcastPage.fxml. It is used to show a podcast of the application.
PodcastPreviewController	Controller associated to PodcastPreview.fxml. It is used to show a preview of a podcast in the author's profile and in the homepage.
PodcastPreviewInUserPageController	Controller associated to PodcastPreviewInUserPage.fxml. It is used to show a preview of a podcast in the user page.
PodcastUpdateController	Controller associated to PodcastUpdate.fxml. It is used to update the podcast's information.
ReviewController	Controller associated to Review.fxml. It is used to show a review in the review page.
ReviewPageController	Controller associated to ReviewPage.fxml. It is used to show all reviews of a podcast in the application.
SearchController	Controller associated to Search.fxml. It is used to manage the search page of the application.
SignUpController	Controller associated to SignUp.fxml. It is used to manage the registration of the different actors of the application.
TableController	Controller associated to Table.fxml. It allows to create a table with custom data in the admin dashboard.
TotalStatisticController	Controller associated to TotalStatistic.fxml. It allows to create a table that show all results of a specific statistic in the admin dashboard.
UserPageController	Controller associated to UserPage.fxml. It provides all the main functionalities used by the users.
UserSearchPreviewController	Controller associated to UserSearchPreview.fxml. It is used to show a preview of a user in the search page.

5.2.5 it.unipi.dii.lsmsdb.myPodcastDB.service

Name	Description
AdminDashboardService	Class that contains all the operations that an administrator can request from the admin dashboard.
AuthorProfileService	Class that contains all the operations that an actor can request from the author's profile.
HomePageService	Class that contains all the operations needed in the homepage.
LoginService	Class that contains all the operations that allows the application to handle the login functionalities.
PodcastPageService	Class that contains all the operations that an actor can request from the podcast page.
ReviewPageService	Class that contains all the operations that an actor can request from the review page.
SearchService	Class that contains all the operations that allows an actor to search podcasts, users and authors.
SignUpService	Class that contains all the operations that allows the application to handle the sign up functionalities.
UserPageService	Class that contains all the operations that an actor can request from the user page.

5.2.6 it.unipi.dii.lsmsdb.myPodcastDB.peristence.mongo

Name	Description
AdminMongo	Class that contains all the CRUD operations available for the admin collection.
AuthorMongo	Class that contains all the CRUD operations available for the author collection.
MongoManager	Class that contains the methods to manage the mongo connection.
PodcastMongo	Class that contains all the CRUD and aggregations available for the podcast collection.
QueryMongo	Class that contains all the CRUD operations available for the query collection.

ReviewMongo	Class that contains all the CRUD operations available for the review collection.
UserMongo	Class that contains all the CRUD and aggregations available for the user collection.

5.2.7 it.unipi.dii.lsmsdb.myPodcastDB.peristence.neo4j

Name	Description
AuthorNeo4j	Class that contains all the CRUD and graph queries available for the author nodes and their relationships.
Neo4jManager	Class that contains the methods to manage the neo4j connection.
PodcastNeo4j	Class that contains all the CRUD and graph queries available for the podcast nodes and their relationships.
UserNeo4j	Class that contains all the CRUD and graph queries available for the user nodes and their relationships.

5.2.8 it.unipi.dii.lsmsdb.myPodcastDB.cache

Name	Description
FollowedAuthorCache	Class that implements a cache containing the followed authors of the author/user that logged in the application.
FollowedUserCache	Class that implements a cache containing the followed users of the user that logged in the application.
ImageCache	Class that contains the least recently used images loaded in the application.
LikedPodcastCache	Class that implements a cache containing the liked podcasts by the user that logged in the application.
WatchlistCache	Class that implements a cache containing the podcasts that the logged user has in the watchlist.

5.2.9 it.unipi.dii.lsmsdb.myPodcastDB.utility

Name	Description
ConfigManager	Class that provides methods that allows to get the settings loaded from the configuration XML file.
JsonDecode	Utility class that allows to decode JSON documents.
Logger	Utility class that allows to send messages on the standard output and on a log file simultaneously, in order to log and track the status of the application.
XMLconfig	Class that represents the configuration XML's structure.

5.3 Aggregations

In this section will be presented all the aggregations used to get **suggestions** and **statistics** in the application. They will be shown both in *Mongo Query Language* and *Java*.

5.3.1 Show countries with highest number of podcasts

```
db.podcast.aggregate([
    {
        $group: {
            _id: { country: "$country" },
            totalPodcasts: { $sum: 1 }
        }
    }, {
        $sort: { totalPodcasts: -1 }
    }, {
        $limit : 1
    }, {
        $project: {
            _id: 0,
            country: "$_id.country",
            totalPodcasts: "$totalPodcasts"
        }
    }
])
```

```java
public List<Pair<String, Integer>> showCountriesWithHighestNumberOfPodcasts(int lim) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson group = group( id: "$country", sum( fieldName: "totalPodcasts",  expression: 1));
        Bson projectionsFields = fields(
                excludeId(),
                computed( fieldName: "country",  expression: "$_id"),
                computed( fieldName: "totalPodcasts",  expression: "$sum"),
                include( ...fieldNames: "totalPodcasts")
        );
        Bson projection = project(projectionsFields);
        Bson sort = sort(descending( ...fieldNames: "totalPodcasts"));
        Bson limit = limit(lim);

        List<Pair<String, Integer>> countries = new ArrayList<>();
        for (Document doc :manager.getCollection("podcast").aggregate(Arrays.asList(group, sort, projection, limit)))
            countries.add(new Pair<>(doc.getString( key: "country"), doc.getInteger( key: "totalPodcasts")));

        if (countries.isEmpty())
            return null;

        return countries;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.3.2 Show podcasts with highest average rating

```
db.podcast.aggregate([
    {
        $project : {
         _id : 0,
         name : "$podcastName",
         artwork : "$artworkUrl600",
         meanRating : { $avg : "$reviews.rating" },
         numberOfReviews : { $size : "$reviews" }
        }
    }, {
        $match : { numberOfReviews : { $gt : 50 } }
    }, {
        $sort : { meanRating : -1, numberOfReviews : -1 }
    }, {
        $limit : 25
    }
])
```

```java
public List<Pair<Podcast, Float>> showPodcastsWithHighestAverageRating(int limit) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson project = project(fields(
                computed( fieldName: "name",  expression: "$podcastName"),
                computed( fieldName: "artwork",  expression: "$artworkUrl600"),
                computed( fieldName: "meanRating", computed( fieldName: "$avg",  expression: "$reviews.rating")),
                computed( fieldName: "numberOfReviews", computed( fieldName: "$size",  expression: "$reviews"))
        ));
        Bson match = match(gt( fieldName: "numberOfReviews",  value: 50));
        Bson sort = sort(Sorts.descending( ...fieldNames: "meanRating", "numberOfReviews"));
        Bson lim = limit(limit);

        List<Pair<Podcast, Float>> results = new ArrayList<>();
        for (Document result : manager.getCollection("podcast").aggregate(Arrays.asList(project, match, sort, lim))) {
            String id = result.getObjectId( key: "_id").toString();
            String name = result.getString( key: "name");
            String artwork = result.getString( key: "artwork");
            double meanRating = result.getDouble( key: "meanRating");
            Podcast podcast = new Podcast(id, name, artwork);
            Pair<Podcast, Float> record = new Pair<>(podcast, (float)meanRating);
            results.add(record);
        }

        if (!results.isEmpty())
            return results;
        else
            return null;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.3.3 Show podcasts with highest average rating per country

```
db.podcast.aggregate([
    {
        $project : {
            name : "$podcastName",
         artwork : "$artworkUrl600",
         country : "$country",
         meanRating : { $avg : "$reviews.rating" },
         numberOfReviews : { $size : "$reviews" }
        }
    }, {
        $match : { numberOfReviews : { $gt : 50 } }
    }, {
        $sort : { meanRating : -1, numberOfReviews : -1 }
    },{
        $group : {
            _id : "$country",
            podcastId : { $first : "$_id" },
            podcastName : { $first : "$name" },
            artwork : { $first : "$artwork" },
            meanRating : { $first : "$meanRating" }
        }
    }
])
```

```java
public List<Triplet<Podcast, String, Float>> showPodcastsWithHighestAverageRatingPerCountry(int limit) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson project = project(fields(
                computed( fieldName: "name",  expression: "$podcastName"),
                computed( fieldName: "artwork",  expression: "$artworkUrl600"),
                include( ...fieldNames: "country"),
                computed( fieldName: "meanRating", computed( fieldName: "$avg",  expression: "$reviews.rating")),
                computed( fieldName: "numberOfReviews", computed( fieldName: "$size",  expression: "$reviews")))
        );
        Bson match = match(gt( fieldName: "numberOfReviews",  value: 50));
        Bson sort = sort(Sorts.descending( ...fieldNames: "meanRating", "numberOfReviews"));
        Bson group = group( id: "$country",
                Accumulators.first( fieldName: "podcastId",  expression: "$_id"),
                Accumulators.first( fieldName: "podcastName",  expression: "$name"),
                Accumulators.first( fieldName: "artwork",  expression: "$artwork"),
                Accumulators.first( fieldName: "meanRating",  expression: "$meanRating")
        );
        Bson lim = limit(limit);

        List<Triplet<Podcast, String, Float>> results = new ArrayList<>();
        for (Document result : manager.getCollection("podcast").aggregate(Arrays.asList(project, match, sort, group, lim))) {
            String id = result.getObjectId( key: "podcastId").toString();
            String name = result.getString( key: "podcastName");
            String artwork = result.getString( key: "artwork");
            String country = result.getString( key: "_id");
            Podcast podcast = new Podcast(id, name, artwork);
            double meanRating = result.getDouble( key: "meanRating");
            Triplet<Podcast, String, Float> record = new Triplet<>(podcast, country, (float)meanRating);
            results.add(record);
        }

        if (!results.isEmpty())
            return results;
        else
            return null;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.3.4 Show podcasts with highest number of reviews

```
db.podcast.aggregate([
    {
        $project : {
            _id : 0,
            name : "$podcastName",
            artwork : "$artworkUrl600",
            numReviews : { $size : "$reviews" }
        }
    }, {
        $sort : { numReviews : -1 }
    }, {
        $limit : 25
    }
])
```

```java
public List<Pair<Podcast, Integer>> showPodcastsWithHighestNumberOfReviews(int limit) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson project = project(fields(
                computed( fieldName: "name",  expression: "$podcastName"),
                computed( fieldName: "artwork",  expression: "$artworkUrl600"),
                computed( fieldName: "numReviews", computed( fieldName: "$size",  expression: "$reviews")))
        );
        Bson sort = sort(Sorts.descending( ...fieldNames: "numReviews"));
        Bson lim = limit(limit);

        List<Pair<Podcast, Integer>> results = new ArrayList<>();
        for (Document result : manager.getCollection("podcast").aggregate(Arrays.asList(project, sort, lim))) {
            String id = result.getObjectId( key: "_id").toString();
            String name = result.getString( key: "name");
            String artwork = result.getString( key: "artwork");
            int numReviews = result.getInteger( key: "numReviews");
            Podcast podcast = new Podcast(id, name, artwork);
            Pair<Podcast, Integer> record = new Pair<>(podcast, numReviews);
            results.add(record);
        }

        if (!results.isEmpty())
            return results;
        else
            return null;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.3.5 Show average age of users per favourite category

```
db.user.aggregate([
    {
        $group: {
            _id: { favouriteGenre: "$favouriteGenre" },
            avgAge: { $avg: "$age" }
        }
    }, {
        $project: {
            _id: 0,
            favouriteGenre: "$_id.favouriteGenre",
            avgAge: "$avgAge"
        }
    }
])
```

```java
public List<Entry<String, Float>> showAverageAgeOfUsersPerFavouriteCategory(int lim) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson group = group( id: "$favouriteGenre", avg( fieldName: "avg",  expression: "$age"));
        Bson projectionsFields = fields(
                excludeId(),
                computed( fieldName: "FavouriteCategory",  expression: "$_id"),
                computed( fieldName: "avgAge",  expression: "$avg")
        );
        Bson projection = project(projectionsFields);
        Bson limit = limit(lim);

        List<Entry<String, Float>> avgList = new ArrayList<>();

        for (Document doc : manager.getCollection("user").aggregate(Arrays.asList(group, projection, limit))) {
            double avg = doc.getDouble( key: "avgAge");
            Entry<String, Float> test = new AbstractMap.SimpleEntry<~>(doc.getString( key: "FavouriteCategory"), (float) avg);
            avgList.add(test);
        }

        if (avgList.isEmpty())
            return null;

        return avgList;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.3.6 Show average age of users per country

```
db.user.aggregate([
    {
        $group: {
            _id: { country: "$country" },
            avgAge: { $avg: "$age" }
        }
    }, {
        $project: {
            _id: 0,
            Country: "$_id.country",
            avgAge: "$avgAge"
        }
    }
])
```

```java
public List<Entry<String, Float>> showAverageAgeOfUsersPerCountry(int lim) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson group = group( id: "$country", avg( fieldName: "avg",  expression: "$age"));
        Bson projectionsFields = fields(
                excludeId(),
                computed( fieldName: "Country",  expression: "$_id"),
                computed( fieldName: "avgAge",  expression: "$avg")
        );
        Bson projection = project(projectionsFields);
        Bson limit = limit(lim);

        List<Entry<String, Float>> avgList = new ArrayList<>();

        for (Document doc : manager.getCollection("user").aggregate(Arrays.asList(group, projection, limit))) {
            double avg = doc.getDouble( key: "avgAge");
            Entry<String, Float> test = new AbstractMap.SimpleEntry<~>(doc.getString( key: "Country"), (float) avg);
            avgList.add(test);
        }

        if (avgList.isEmpty())
            return null;

        return avgList;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.3.7 Show number of users per country

```
db.user.aggregate([
    {
        $group: {
            _id: { country: "$country" },
            num: { $sum: 1 }
        }
    }, {
        $sort : { num : -1 }
    }, {
        $project: {
            _id: 0,
            country: "$_id.country",
            users: "$num"
        }
    }, {
        $limit : 25
    }
])
```

```java
public List<Entry<String, Integer>> showNumberOfUsersPerCountry(int limit) {
    MongoManager manager = MongoManager.getInstance();

    try {
        Bson group = group( id: "$country", sum( fieldName: "num",  expression: 1));
        Bson sort = sort(descending( ...fieldNames: "num"));
        Bson project = project(fields(
                excludeId(),
                computed( fieldName: "country",  expression: "$_id"),
                computed( fieldName: "users",  expression: "$num")
        ));
        Bson limitRes = limit(limit);

        List<Document> results = manager.getCollection("user") MongoCollection<Document>
                .aggregate(Arrays.asList(group, sort, project, limitRes)) AggregateIterable<Document>
                .into(new ArrayList<>());
        if(results.isEmpty())
            return null;

        List<Entry<String, Integer>> countries = new ArrayList<>();
        for (Document result : results){
            Entry<String, Integer> country = new AbstractMap.SimpleEntry<>(
                    result.getString( key: "country"),
                    result.getInteger( key: "users")
            );
            countries.add(country);
        }

        return countries;

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }

}
```

5.3.8 Show favourite categories for gender

```
db.user.aggregate([
    {
        $match: { gender: { $eq: "other" } }
    }, {
        $group: {
            _id: "$favouriteGenre",
            num: { $sum: 1 }
        }
    }, {
        $sort: { num: -1 }
    }, {
        $project: { category: "$_id" }
    }, {
        $limit: 10
    }
])
```

```
public List<String> showFavouriteCategoryForGender(String gender, int limit) {

    MongoManager manager = MongoManager.getInstance();

    try {
        Bson match = match(eq( fieldName: "gender", gender));
        Bson group = group( id: "$favouriteGenre", sum( fieldName: "num",  expression: 1));
        Bson sort = sort(descending( ...fieldNames: "num"));
        Bson project = project(fields(excludeId(), computed( fieldName: "category",  expression: "$_id")));
        Bson limitRes = limit(limit);

        List<Document> results = manager.getCollection("user") MongoCollection<Document>
                .aggregate(Arrays.asList(match, group, sort, project, limitRes)) AggregateIterable<Document>
                .into(new ArrayList<>());
        if(results.isEmpty())
            return null;
        List<String> categories = new ArrayList<>();
        for (Document category : results)
            categories.add(category.getString( key: "category"));
        return categories;

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.4 Graph Queries

In this section will be presented all the graph queries used to get **suggestions** and **statistics** in the application. They will be shown both in *Cypher Query Language* and *Java*.

5.4.1 Show suggested authors followed by followed user

```
MATCH (u:User { username: $username})-[:FOLLOWS_USER]-(:User)-
    [:FOLLOWS]->(a1:Author)
WHERE NOT EXISTS {
    MATCH (u)-[:FOLLOWS]->(a1)
}
RETURN DISTINCT a1
SKIP $skip
LIMIT $limit
```

```
public List<Author> showSuggestedAuthorsFollowedByFollowedUser(String username, int limit, int skip) {
    Neo4jManager manager = Neo4jManager.getInstance();

    try {
        String query = "MATCH (u:User { username: $username })-[:FOLLOWS_USER]-(:User)-[:FOLLOWS]->(a1:Author) " +
                "WHERE NOT EXISTS " +
                "{ MATCH (u)-[:FOLLOWS]->(a1) } " +
                "RETURN DISTINCT a1 " +
                "SKIP $skip " +
                "LIMIT $limit";
        List<Record> result = manager.read(query, parameters( ...keysAndValues: "username", username, "limit", limit, "skip", skip));

        if (result.isEmpty())
            return null;

        List<Author> authors = new ArrayList<>();
        for (Record record : result) {
            String name = record.get(0).get("name").asString();
            String picturePath = record.get(0).get("picturePath").asString();
            Author author = new Author( id: "", name, picturePath);
            authors.add(author);
        }

        return authors;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.4.2 Show most followed author

```
MATCH (a:Author)<-[f:FOLLOWS]-()
WITH a.name AS name, a.picturePath AS picturePath,
    COUNT(f) AS followers
RETURN name, picturePath, followers
ORDER BY followers DESC
LIMIT $limit
```

```java
public List<Pair<Author, Integer>> showMostFollowedAuthor(int limit) {
    Neo4jManager manager = Neo4jManager.getInstance();

    List<Record> result = null;
    try {
        String query =  "MATCH (a:Author)<-[f:FOLLOWS]-() " +
                        "WITH a.name AS name, a.picturePath AS picturePath, COUNT(f) AS followers " +
                        "RETURN name, picturePath, followers " +
                        "ORDER BY followers DESC " +
                        "LIMIT $limit";
        Value params = parameters( ...keysAndValues: "limit", limit);
        result = manager.read(query, params);
    } catch (Exception e) {
        e.printStackTrace();
    }

    if (result == null)
        return null;

    List<Pair<Author, Integer>> authors = new ArrayList<>();
    for (Record record : result) {
        String name = record.get("name").asString();
        String picturePath = record.get("picturePath").asString();
        int followers = record.get("followers").asInt();
        Author author = new Author( id: "", name, picturePath);
        authors.add(new Pair<>(author, followers));
    }

    return authors;
}
```

5.4.3 Show most liked podcasts

```
MATCH (p:Podcast)-[l:LIKES]-()
WITH p.name as PodcastName, p.podcastId as PodcastId, p.artworkUrl600
    as Artwork, COUNT(l) as Likes
RETURN PodcastName, PodcastId, Artwork, Likes
ORDER BY Likes DESC
LIMIT $limit
```

```java
public List<Entry<Podcast, Integer>> showMostLikedPodcasts(int limit) {
    Neo4jManager manager = Neo4jManager.getInstance();

    try {
        String query = "MATCH (p:Podcast)-[l:LIKES]-() " +
                "WITH p.name as PodcastName, p.podcastId as PodcastId, p.artworkUrl600 as Artwork, " +
                "COUNT(l) as Likes " +
                "RETURN PodcastName, PodcastId, Artwork, Likes " +
                "ORDER BY Likes DESC " +
                "LIMIT $limit";
        List<Record> result = manager.read(query, parameters( ...keysAndValues: "limit", limit));

        if (result.isEmpty())
            return null;

        List<Entry<Podcast, Integer>> mostLikedPodcasts = new ArrayList<>();
        for (Record record: result) {
            String podcastId = record.get("PodcastId").asString();
            String podcastName = record.get("PodcastName").asString();
            String artworkUrl600 = record.get("Artwork").asString();
            int likes = record.get("Likes").asInt();

            Podcast podcast = new Podcast(podcastId, podcastName, artworkUrl600);
            Entry<Podcast, Integer> newPodcast = new AbstractMap.SimpleEntry<>(podcast, likes);
            mostLikedPodcasts.add(newPodcast);
        }

        return mostLikedPodcasts;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.4.4 Show most numerous categories

```
MATCH (c:Category)<-[b:BELONGS_TO]-()
WITH c.name as name, COUNT(b) AS belonging
RETURN name, belonging
ORDER BY belonging DESC
LIMIT $limit
```

```java
public List<Entry<String, Integer>> showMostNumerousCategories(int limit) {
    Neo4jManager manager = Neo4jManager.getInstance();

    List<Record> result = null;
    try {
        String query =  "MATCH (c:Category)<-[b:BELONGS_TO]-() " +
                        "WITH c.name as name, COUNT(b) AS belonging " +
                        "RETURN name, belonging " +
                        "ORDER BY belonging DESC " +
                        "LIMIT $limit";
        Value params = parameters( ...keysAndValues: "limit", limit);
        result = manager.read(query, params);
    } catch (Exception e) {
        e.printStackTrace();
    }

    if (result == null)
        return null;

    List<Entry<String, Integer>> categories = new ArrayList<>();
    for (Record record : result) {
        String name = record.get("name").asString();
        int value = record.get("belonging").asInt();
        Entry<String, Integer> category = new AbstractMap.SimpleEntry<>(name, value);
        categories.add(category);
    }

    return categories;
}
```

5.4.5 Show most appreciated categories

```
MATCH (c:Category)<-[:BELONGS_TO]-(p:Podcast)<-[l:LIKES]-(:User)
WITH c, count(l) as likes
RETURN c.name as name, likes
ORDER BY likes desc
LIMIT $limit
```

```java
public List<Entry<String, Integer>> showMostAppreciatedCategories(int limit) {
    Neo4jManager manager = Neo4jManager.getInstance();
    String query = " MATCH (c:Category)<-[:BELONGS_TO]-(p:Podcast)<-[l:LIKES]-(:User)" + "\n" +
            "WITH c, count(l) as likes" + "\n" +
            "RETURN c.name as name, likes" + "\n" +
            "ORDER BY likes desc" + "\n" +
            "LIMIT $limit";
    Value params = parameters( ...keysAndValues: "limit", limit);
    List<Record> result = null;

    try {
        result = manager.read(query, params);

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }

    if(result == null || !result.iterator().hasNext())
        return null;

    List<Entry<String, Integer>> categories = new ArrayList<>();
    for(Record record : result){
        String categoryName = record.get("name").asString();
        int likes = record.get("likes").asInt();
        Entry<String, Integer> category = new AbstractMap.SimpleEntry<>(categoryName, likes);
        categories.add(category);
    }

    return categories;
}
```

5.4.6 Show suggested podcasts liked by followed users

```
MATCH (source:User{username: $username})-[:FOLLOWS_USER]->(u:User)-
    [:LIKES]->(p:Podcast), (p)<-[l:LIKES]-(:User)
WHERE NOT EXISTS {
    MATCH (source)-[:LIKES]->(p)
} AND NOT EXISTS {
    MATCH (source)-[:WATCH_LATER]->(p)
}
WITH p.name as name, p.podcastId as id, p.artworkUrl600 as artwork,
    COUNT(l) as likes
RETURN DISTINCT name, id, artwork, likes
ORDER BY likes desc
SKIP $skip
LIMIT $limit
```

```java
public List<Podcast> showSuggestedPodcastsLikedByFollowedUsers(User user, int limit, int skip) {
    Neo4jManager manager = Neo4jManager.getInstance();
    String query = "MATCH (source:User{username: $username})-[:FOLLOWS_USER]->(u:User)-[:LIKES]->(p:Podcast)," + "\n" +
            "(p)<-[l:LIKES]-(:User)" + "\n" +
            "WHERE NOT EXISTS { match (source)-[:LIKES]->(p) }" + "\n" +
            "and NOT EXISTS { match (source)-[:WATCH_LATER]->(p) }" + "\n" +
            "WITH p.name as name, p.podcastId as id, p.artworkUrl600 as artwork, count(l) as likes" + "\n" +
            "RETURN DISTINCT name, id, artwork, likes" + "\n" +
            "ORDER BY likes desc" + "\n" +
            "SKIP $skip" + "\n" +
            "LIMIT $limit" ;
    Value params = parameters( ...keysAndValues: "username", user.getUsername(), "limit", limit, "skip", skip);
    List<Record> result = null;

    try {
        result = manager.read(query, params);

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }

    if(result == null || !result.iterator().hasNext())
        return null;

    List<Podcast> podcasts = new ArrayList<>();
    for(Record record : result){
        String podcastName = record.get("name").asString();
        String podcastId = record.get("id").asString();
        String artworkUrl600 = record.get("artwork").asString();

        Podcast podcast = new Podcast(podcastId, podcastName, artworkUrl600);
        podcasts.add(podcast);
    }

    return podcasts;
}
```

5.4.7 Show suggested podcasts based on category of podcast user liked

```
MATCH (u:User {username: $username})-[:LIKES]->(liked:Podcast)
MATCH (liked)-[:BELONGS_TO]->(:Category)<-[:BELONGS_TO]-(p:Podcast)
WHERE NOT EXISTS {
    MATCH (u)-[:LIKES]->(p)
}
WITH p.name as name, p.podcastId as pid, p.artworkUrl600 as artwork
RETURN DISTINCT name, pid, artwork
SKIP $skip
LIMIT $limit
```

```java
public List<Podcast> showSuggestedPodcastsBasedOnCategoryOfPodcastsUserLiked(String username, int limit, int skip) {
    Neo4jManager manager = Neo4jManager.getInstance();

    List<Record> result = null;
    try {
        String query =  "MATCH (u:User {username: $username})-[:LIKES]->(liked:Podcast) " +
                        "MATCH (liked)-[:BELONGS_TO]->(:Category)<-[:BELONGS_TO]-(p:Podcast) " +
                        "WHERE NOT EXISTS {MATCH (u)-[:LIKES]->(p)} " +
                        "WITH p.name as name, p.podcastId as pid, p.artworkUrl600 as artwork " +
                        "RETURN DISTINCT name, pid, artwork " +
                        "SKIP $skip " +
                        "LIMIT $limit";
        Value params = parameters( ...keysAndValues: "username", username, "limit", limit, "skip", skip);
        result = manager.read(query, params);
    } catch (Exception e) {
        e.printStackTrace();
    }

    if (result == null)
        return null;

    List<Podcast> podcasts = new ArrayList<>();
    for (Record record : result) {
        String id = record.get("pid").asString();
        String name = record.get("name").asString();
        String artworkUrl600 = record.get("artwork").asString();

        Podcast podcast = new Podcast(id, name, artworkUrl600);
        podcasts.add(podcast);
    }

    return podcasts;
}
```

5.4.8 Show suggested podcasts based on authors of podcasts in watchlist

```
MATCH (s:User{username: $username})-[w:WATCH_LATER]->(p1:Podcast)-
    [c1:CREATED_BY]->(a:Author), (a)<-[c2:CREATED_BY]-(p2:Podcast)
WHERE NOT EXISTS {
    MATCH (s)-[:WATCH_LATER]->(p2)
}
WITH p2.name as name, p2.podcastId as id, p2.artworkUrl600 as artwork
RETURN DISTINCT name, id, artwork
SKIP $skip
LIMIT $limit
```

```java
public List<Podcast> showSuggestedPodcastsBasedOnAuthorsOfPodcastsInWatchlist(User user, int limit, int skip) {
    Neo4jManager manager = Neo4jManager.getInstance();
    String query = "MATCH (s:User{username: $username})-[w:WATCH_LATER]->(p1:Podcast)-[c1:CREATED_BY]->(a:Author)," + "\n" +
            "(a)<-[c2:CREATED_BY]-(p2:Podcast)" + "\n" +
            "WHERE NOT EXISTS { match (s)-[:WATCH_LATER]->(p2) }" + "\n" +
            "WITH p2.name as name, p2.podcastId as id, p2.artworkUrl600 as artwork" + "\n" +
            "RETURN DISTINCT name, id, artwork" + "\n" +
            "SKIP $skip" + "\n" +
            "LIMIT $limit";

    Value params = parameters( ...keysAndValues: "username", user.getUsername(), "limit", limit, "skip", skip);
    List<Record> result = null;

    try {
        result = manager.read(query, params);

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }

    if(result == null || !result.iterator().hasNext())
        return null;

    List<Podcast> podcasts = new ArrayList<>();
    for(Record record : result){
        String podcastName = record.get("name").asString();
        String podcastId = record.get("id").asString();
        String artworkUrl600 = record.get("artwork").asString();

        Podcast podcast = new Podcast(podcastId, podcastName, artworkUrl600);
        podcasts.add(podcast);
    }

    return podcasts;
}
```

5.4.9 Show suggested users by followed authors

```
MATCH (u1:User {username: $username})-[:FOLLOWS]->(:Author)<-
    [:FOLLOWS]-(u2:User)
WHERE NOT EXISTS {
    MATCH (u1)-[:FOLLOWS_USER]->(u2)
}
RETURN DISTINCT u2
LIMIT $limit
```

```java
public List<User> showSuggestedUsersByFollowedAuthors(String username, int limit) {
    Neo4jManager manager = Neo4jManager.getInstance();

    try {
        String query = "MATCH (u1:User {username: $username })-[:FOLLOWS]->(:Author)<-[:FOLLOWS]-(u2:User) " +
                "WHERE NOT EXISTS " +
                "{ MATCH (u1)-[:FOLLOWS_USER]->(u2) } " +
                "RETURN DISTINCT u2 " +
                "LIMIT $limit";
        List<Record> result = manager.read(query, parameters( ...keysAndValues: "username", username, "limit", limit));

        if (result.isEmpty())
            return null;

        List<User> suggestedUsers = new ArrayList<>();
        for (Record record: result) {
            String suggestedUsername = record.get(0).get("username").asString();
            String picturePath = record.get(0).get("picturePath").asString();

            User user = new User(suggestedUsername, picturePath);
            suggestedUsers.add(user);
        }

        return suggestedUsers;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

5.4.10 Show suggested users by liked podcasts

```
MATCH (u1:User {username: $username})-[:LIKES]->(:Podcast)<-
    [:LIKES]-(u2)
WHERE NOT EXISTS {
    (u1)-[:FOLLOWS_USER]->(u2)
}
RETURN DISTINCT u2
LIMIT $limit
```

```java
public List<User> showSuggestedUsersByLikedPodcasts(String username, int limit) {
    Neo4jManager manager = Neo4jManager.getInstance();

    List<Record> result = null;
    try {
        String query =  "MATCH (u1:User {username: $username})-[:LIKES]->(:Podcast)<-[:LIKES]-(u2) " +
                        "WHERE NOT EXISTS { (u1)-[:FOLLOWS_USER]->(u2) } " +
                        "RETURN DISTINCT u2 " +
                        "LIMIT $limit";
        Value params = parameters( ...keysAndValues: "username", username, "limit", limit);
        result = manager.read(query, params);
    } catch (Exception e) {
        e.printStackTrace();
    }

    if (result == null)
        return null;

    List<User> users = new ArrayList<>();
    for (Record record : result) {
        String suggestedUsername = record.get(0).get("username").asString();
        String picturePath = record.get(0).get("picturePath").asString();

        User user = new User(suggestedUsername, picturePath);
        users.add(user);
    }

    return users;
}
```

5.5 Insights

In this section will be given additional information about some implementation choices.

5.5.1 Connection Management

The class that handles the connection to the mongo database is shown below:

```java
public class MongoManager implements AutoCloseable {
    // Singleton
    static MongoManager mongoManager;

    private MongoClient mongoClient;
    private MongoDatabase database;

    public MongoManager() {...}

    public boolean openConnection() {...}
    public boolean closeConnection() {...}

    public MongoCollection<Document> getCollection(String collection) {...}
    public static MongoManager getInstance() {...}

    @Override
    public void close() throws Exception {...}
}
```

The class provides two methods to handle the **opening** and the **closing** of the connection.

As mentioned earlier, in the service layer there are methods that group all the operations required to perform a given operation on the databases. Therefore, it has been chosen to open the connection at the beginning of each service, perform all the required operations, handle any errors through rollback operations, and close the connection at the end.

An example of a service is presented below:

```
public void searchAsUser(String searchText, List<Podcast> podcastsMatch, List<Author> authorsMatch, ...) {
    Logger.info("Searching as Registered User");
    MongoManager.getInstance().openConnection();

    // Searching for podcasts
    if (filters.getValue0()) {
        List<Podcast> podcasts = podcastMongoManager.searchPodcast(searchText, limit, skip: 0);
        if (podcasts != null)
            podcastsMatch.addAll(podcasts);
    }

    // Searching for authors
    if (filters.getValue1()) {
        List<Author> authors = authorMongoManager.searchAuthor(searchText, limit, skip: 0);
        if (authors != null)
            authorsMatch.addAll(authors);
    }

    // Searching for users
    if (filters.getValue2()) {
        List<User> users = userMongoManager.searchUser(searchText, limit, skip: 0);
        if (users != null)
            usersMatch.addAll(users);
    }

    MongoManager.getInstance().closeConnection();
}
```

The system is set up so that it is not allowed to create a new connection if one is already active and it is not allowed to close a connection if there is not already an active one.

Finally, the mongo manager implements *AutoCloseable*. So, if when the application terminates there is an active connection, the close method of mongo manager will close it.

The class Neo4J Manager follows the same concepts.

5.5.2 Cache

In order to avoid unnecessary requests to the databases, a **cache layer** has been adopted. In particular, this has been done to store locally the relationships between the different entities for example likes, watch later, follows, etc. Since the watch later and follows relations cannot grow too much, they are fully loaded at the login. Instead, the like relations can grow indefinitely, so they will be stored into the cache gradually during the execution of the application.

```
public class WatchlistCache {
    // Singleton
    private static WatchlistCache watchlistCache;

    private Hashtable<String, Podcast> htPodcast;

    // Cache parameters
    private static int limit = 100;
    private static int size;

    public WatchlistCache() {...}

    public static List<Podcast> getAllPodcastsInWatchlist() {...}
    public static Podcast getPodcast(String podcastId) {...}
    public static boolean addPodcast(Podcast podcast) {...}
    public static void addPodcastList(List<Podcast> podcasts) {...}
    public static void removePodcast(String podcastId) {...}
    public static void clearPodcasts() {...}

    public static WatchlistCache getInstance() {...}
}
```

A different thing is for the **image cache**. The application loads several images, in particular from external URLs. This operation requires some times to be completed. Moreover, browsing the application can happen that the same images are requested several times. So, it has been decided to store the images in a dedicated cache.

Since a single image can exceed *100KB* of size, the cache may become too big. To overcome this, a **least recently used strategy** has been implemented to limit the maximum number of images that can reside in memory.

5.5.3 Config Manager

A useful functionality is to use a **configuration file** that allows to change parameters such as server IPs that can change over time. So, all these parameters have been gathered in the file *config.xml* and it has been created the config manager class that makes accessible them in every modules of the application.

This is an example of the configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xmlConfig>
    <mongoDBConfigType>local</mongoDBConfigType>
    <mongoDBLocalConfig>
        <mongoDBIp>localhost</mongoDBIp>
        <mongoDBPort>27017</mongoDBPort>
        <mongoDBName>myPodcastDB</mongoDBName>
        <mongoDBUser>admin</mongoDBUser>
        <mongoDBPassword>password</mongoDBPassword>
    </mongoDBLocalConfig>
    <neo4JConfig>
        <neo4JIp>localhost</neo4JIp>
        <neo4JPort>7687</neo4JPort>
        <neo4JName>myPodcastDB</neo4JName>
        <neo4JUser>neo4j</neo4JUser>
        <neo4JPassword>password</neo4JPassword>
    </neo4JConfig>
    <logMode>verbose</logMode>
    <imageCacheSize>100</imageCacheSize>
</xmlConfig>
```

5.5.4 Logger

This utility class allows to **send messages** on the *standard output* and on a *log file* simultaneously, in order to log and track the status of the application. In particular, three different types of messages are available:

- **info**: it's identified by the "!" character that precedes the message
- **success**: it's identified by the "+" character that precedes the message
- **error**: it's identified by the "-" character that precedes the message

The messages are provided together with additional information such as the *name of the source view* and the *timestamp*.

Three different behaviors are defined for the logger class:

- **verbose**: all types of messages are shown
- **noInfo**: the info messages are hidden
- **deploy**: only the error messages are shown

The behavior can be set in the configuration file.

An example of usage of the logger is the following:

```
Logger.success("Operation completed!");
```

The resulting message is:

```
[+] [ 2022-06-17T19:22:03.96 ] {Search.fxml} Operation completed!
```

5.5.5 Session Management

It has been implemented a **session system** to store the type and the information related to the actor that logged into the application. These information are stored in the *MyPodcastDB* class, instantiated using a *singleton* that made it reachable from every module of the application.

The possible actor types available are the following:

- Admin
- Author
- User
- Unregistered User

This is an example of actor recognition in the code:

```
String sessionType = MyPodcastDB.getInstance().getSessionType();
if (sessionType.equals("Unregistered")) {
    this.authorName.setDisable(true);
}
```

5.6 Consistency between databases

In this section will be shown all the operations that involve both databases and for which will be necessary to grant the **consistency**.

5.6.1 Add User

For the creation of a new user, the workflow is the following:

1. Add **user** in user collection on *Mongo*
2. Add **user** node on *Neo4J*

So, it is necessary that the shared fields between the databases are consistent. If the add on **Neo4J fails**, the user document added previously on Mongo must be **removed**.

5.6.2 Update User

For the update of a user, the workflow is the following:

1. Update **user** document on *Mongo*
2. Update **user** node on *Neo4J*, if needed
3. Update **review** document on *Mongo* if the username has been modified
4. Update **preloaded reviews** in **podcast**'s document if one or more of them have been written by the user who is updating his profile and if he is changing his username

It is necessary to ensure the consistency between the username and picture path fields either for Mongo and Neo4J. Moreover, the username field of the reviews in the review collection has to be consistent with the new username of the user, if changed. The same is for the reviews embedded in the podcast document.

5.6.3 Delete User

For the delete of a user, the workflow is the following:

1. Delete **user** document on *Mongo*
2. Update the **username of reviews** in review collection in *Removed Account*
3. Update **preloaded reviews** in **podcast**'s document if one or more of them have been written by the user which is being removed
4. Delete **user** node on *Neo4J*

It is necessary to grant that if the delete on **Neo4J fails** or one of the updates fails, the user document and the modified entities must be **restored**.

5.6.4 Add Author

For the creation of a new author, the workflow is the following:

1. Add **author** in author collection on *Mongo*
2. Add **author** node on *Neo4J*

So, it is necessary that the shared fields between the databases are consistent. If the add on **Neo4J fails**, the author document added previously on Mongo must be **removed**.

5.6.5 Update Author

For the update of an author, the workflow is the following:

1. Update **author** document on *Mongo*
2. Update **author** node on *Neo4J*
3. Update **podcasts** in podcast collection if the author name is changed

In this case it is necessary to ensure the consistency between the name and picture path fields for both databases. Moreover, the author name field of the podcasts in the podcast document has to be consistent with the new author name, if changed.

5.6.6 Delete Author

For the delete of an author, the workflow is the following:

1. Delete **author** document on *Mongo*
2. Delete **author** node on *Neo4J*
3. Delete **author's podcast** nodes on *Neo4J*
4. Delete **author's podcast** documents in the podcast collection
5. Delete **reviews embedded in user**
6. Delete **reviews** in the review collection

It is necessary to grant that if the **Neo4J deletes fail**, the Mongo delete will be **restored** and **vice versa**.

This operation is the most complex one because it requires to delete all the author's podcasts and all the reviews associated to them; so if *point 5* is completed successfully, no rollback will be performed anymore. In *point 6* will be deleted the author's podcast reviews using **delete many** by podcast id. If a delete fails, **another delete attempt** will be made when all other deletes are completed. If the second attempt also fails, the id of the reviews that were not deleted will be **logged**, so that the admin can delete them later. Another solution might be to store all the reviews in memory before deleting them in order to perform an eventual rollback. As said before, the number of reviews can be very large so this is not a good solution.

5.6.7 Add Podcast

For the creation of a new podcast, the workflow is the following:

1. Add **podcast** document on *Mongo*
2. Add **podcast embedded in author** on *Mongo*
3. Add **podcast** node on *Neo4J*

So, it is necessary that the shared fields between the databases are consistent. If the add on **Neo4J fails**, the podcast document added previously on Mongo must be **removed**.

5.6.8 Update Podcast

For the update of a podcast, the workflow is the following:

1. Update **podcast** document on *Mongo*
2. Update **podcast embedded in author** on *Mongo*, if needed
3. Update **podcast** node on *Neo4J*, if needed

In this case it is necessary to ensure the consistency between the podcast name and artwork url fields for both databases. Moreover, podcast name, release date, artwork url and primary category fields embedded in author document have to be consistent with the new information, if changed.

5.6.9 Delete Podcast

For the delete of a podcast, the workflow is the following:

1. Delete **podcast** document on *Mongo*
2. Delete **podcast embedded in author** on *Mongo*
3. Delete **podcast** node on *Neo4J*
4. Delete **reviews embedded in users**
5. Delete **podcast's reviews**

It is necessary to grant that if the delete on **Neo4J fails** or one of the **Mongo delete fails**, the podcast document and the modified entities must be **restored**. Even in this case, as for the delete of an author, the last operation is the reviews delete in order to **simplify** the **rollback operation**.

6 Unit Tests

The test have been done for all the classes of **persistence layer** in order to verify the correctness of the operations made on the two different databases.

6.1 Tests on MongoDB

These are the classes used to test the operations on MongoDB.

Name	Description
AdminMongoTest	Class that tests the CRUD operations on the admin collection.
AuthorMongoTest	Class that tests the CRUD operations on the author collection.
PodcastMongoTest	Class that tests the CRUD and aggregations on the podcast collection.
ReviewMongoTest	Class that tests the CRUD operations on the review collection.
UserMongoTest	Class that tests the CRUD and aggregations on the user collection.

6.2 Tests on Neo4J

These are the classes used to test the operations on Neo4J.

Name	Description
AuthorNeo4jTest	Class that tests the CRUD and graph queries on the author nodes and relationships.
PodcastNeo4jTest	Class that tests the CRUD and graph queries on the podcasts nodes and relationships.
UserNeo4jTest	Class that tests the CRUD and graph queries on the user nodes and relationships.