SPOTIFY 2.0

Picchi Nicolò

2023/2024

UNIVERSITY OF PISA

Large Scale and Multi-Structured Databases

Sommario

Abstract

Spotify 2.0 is a JavaFX application designed to provide users with an enriched multimedia experience, allowing them to explore and share their favorite songs and playlists. To access the platform, users need an account. Existing users can log in by entering their username and password, while new users must complete a form containing fields for username, email, password, country, and a profile photo.

Upon successful login, users are directed to the homepage. Here, they can utilize the search bar and corresponding buttons to find songs by title or artist. The homepage also serves as a gateway to the user's library, where favorite songs are stored, playlists are managed, and the profile section is accessible for account information adjustments or deletions.

When a user clicks on a searched song, a dedicated song page displays comprehensive information about the song. Users can add the song to their library or create a new playlist on the spot and add the song to it.

The application incorporates a social aspect, allowing users to follow others, share playlists, and contribute songs to playlists shared by followed users. Each user maintains two personal lists: Library (housing favorite songs) and Playlists (containing user-created playlists). Users can add, delete, or share (limited to playlists) songs and playlists from their respective pages.

An Administrator, a special user role, possesses the ability to insert, modify, and delete songs from the system. However, Administrators cannot create playlists or maintain a list of favorite songs. They also have access to analytics, enabling them to study playlist statistics, identify the top ten favorite songs by country, and analyze genre distribution among listener playlists.

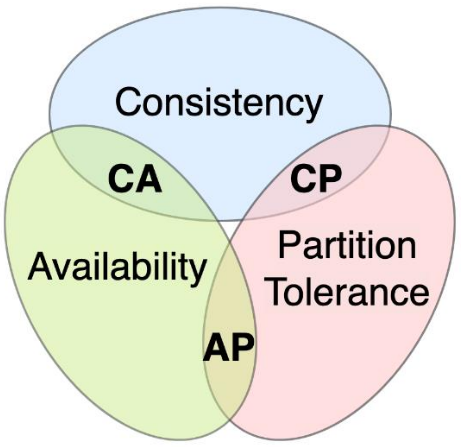
The **git server address** containing all the artifacts (code, database dump and executable files) is https://github.com/NicoUniPi/spotify2.

Feasibility study

Overview on the Cap Theorem

For our application, where read operations are anticipated to be frequent, ensuring high availability and low latency takes precedence, even in the face of potential network partitioning. In the context of the CAP theorem, our application aligns more with the AP side of the triangle, prioritizing Availability (A) and Partition Tolerance (P) over strict data Consistency (C).

To meet non-functional requirements, a conscious decision has been made to prioritize high availability of data. In the event of network errors, we accept that the content returned to the user might not always reflect the most up-to-date information, allowing for temporary display of data in an older version. This choice is directly linked to our adoption of the Eventual Consistency paradigm for our distributed system, as further detailed in the Distributed Database Design section.



Dataset analysis

Two primary datasets, namely "spotify_songs.json" and "amazon_music_songs.csv," were obtained from Kaggle.com. These datasets encompass songs featured in our application, along with their associated attributes. The original combined file size of both datasets was approximately 50 MB. To streamline the data for our application, certain attributes were removed, and others were reshaped to align with the application's requirements.

The user information, hereafter referred to as "listeners," was generated using randomuser.com. A total of 50,000 listeners were created, resulting in a dataset size of 38 MB.

To populate playlists and libraries, a Java script was employed, contributing to an uncompressed data size of 119 MB. This script facilitated the creation of playlists and libraries within the application, enriching the dataset with listener-specific song preferences and organizational structures.

Design

Actors of the system

The system involves two distinct categories of users:

- **Administrator:**
    - ***Responsability:*** Administrators are tasked with the overall management of the system.

    Their roles include tasks such as inserting, modifying, and deleting songs from the system. Additionally, administrators have access to analytical tools for studying playlist statistics, identifying top songs by country, and analyzing genre distribution among listener playlists.

- **Listener:**
    - ***Role:*** Listeners are the regular users of the system, utilizing it to search, share, and discover new songs. Their interactions encompass creating and managing playlists, exploring the song library, and engaging with social features such as following other users, sharing playlists, and contributing to shared playlists.

Functional requirements

This section defines the services that the system provides to the users. To every functional requirement is assigned a unique key *FNC-nnn* where *nnn* is a sequence number identifying the functional requirement. The following table lists all the functional requirements:

Functional Requirement	Description
FNC-001	A listener shall be able to login into the system if he has an account, otherwise he shall be able to register.
FNC-002	Each username shall be associated with up to one account.
FNC-003	A listener shall be able to modify or delete his account.
FNC-004	A listener shall be able to search for a song by title using a search bar.
FNC-005	A listener shall be able to search for a song by artist using a search bar.
FNC-006	A listener shall be able to see all the information regarding a song, on a dedicated page, clicking on the corresponding title.
FNC-007	A listener shall be able to add and remove a song from the library list.
FNC-008	A listener shall be able to create or delete a new playlist.
FNC-009	A listener shall be able to add and remove a song from a playlist.
FNC-010	A listener shall be able to search for a listener by username using a search bar.
FNC-011	A listener shall be able to see all the information

	regarding another listener, on a dedicated page, clicking on the corresponding username.
FNC-012	A listener shall be able to follow or unfollow another listener.
FNC-013	A listener shall be able to share or remove from the sharing a playlist with his follower.
FNC-014	A follower shall be able to add new songs or remove added song to a playlist's follower.
FNC-015	A follower shall not be able to remove songs added by other listeners.
FNC-016	The administrator shall be able to insert, modify and delete songs from the system.
FNC-017	The administrator shall be able to see the top ten liked songs per country.
FNC-018	The administrator shall be able to see statistics about a listener's playlists.
FNC-019	The system shall be able to suggest songs based on the listener favorite genre.
FNC-020	The system shall be able to suggest friends with similar music tastes of the user.

Non-functional requirements

Non-functional requirements describe only attributes of the system or attributes of the system environment. To every non-functional requirement is assigned a unique key XX-nnn where XX identifies the type of requirement (example: AVA for AVAILABILITY) and nnn is a sequence number identifying the non-functional requirement.

Availability

Availability Requirement	**Description**
AVA-001	high availability, accepting data displayed temporarily in an old version.
AVA-002	tolerance to the loss of data, avoiding a single point of failure.

Scalability

Scalability Requirement	**Description**
SCA-001	The system shall work on a non-prefixed number of cluster nodes.

Reliability

Reliability Requirement	**Description**
REL-001	The system must give stable and reproducible results.

Latency

Latency Requirement	Description
LA-001	The application must have a low response time.

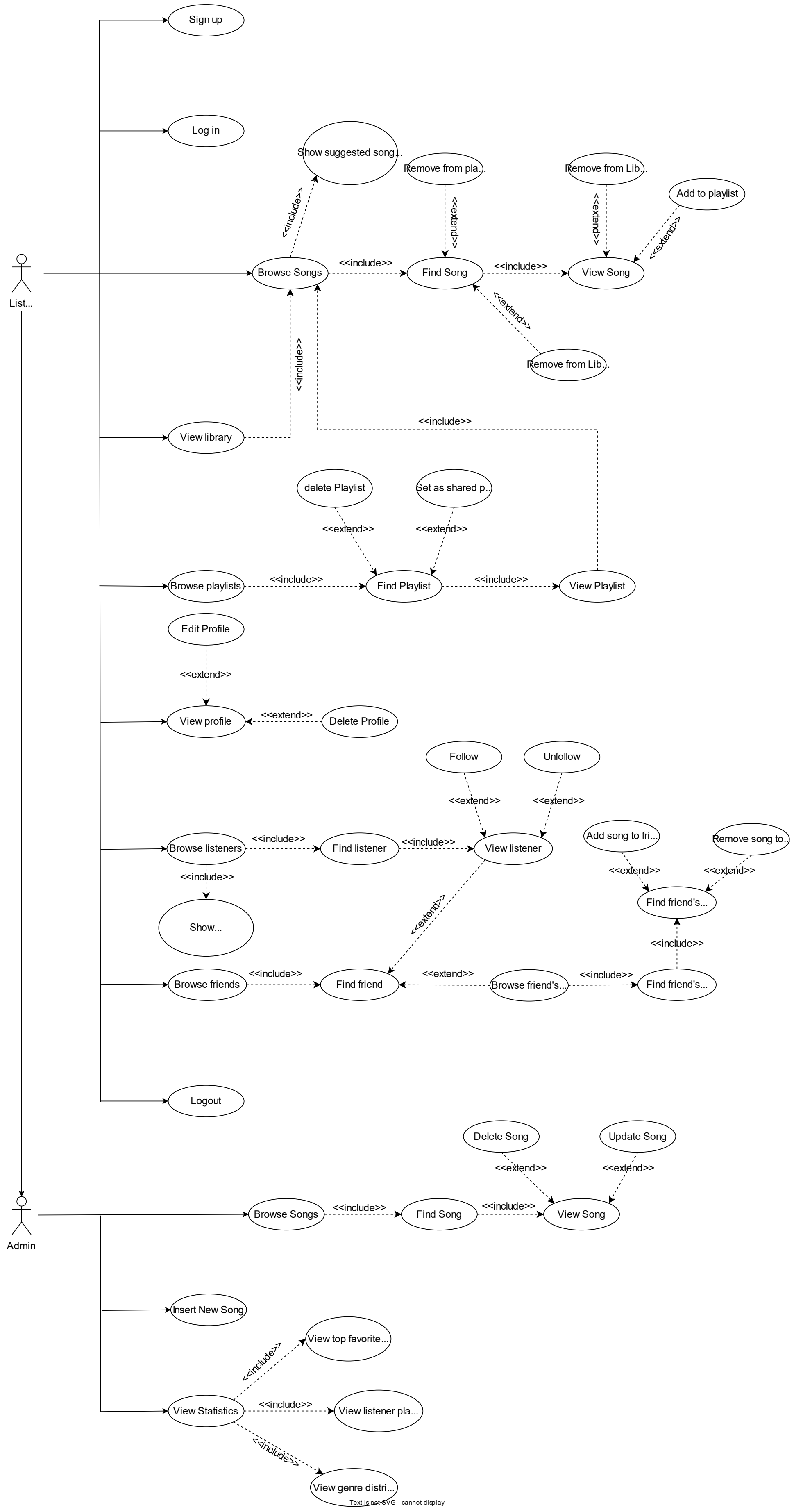
Maintainability

Maintainability Requirement	Description
MAN-001	The code shall be readable and easy to maintain.

Usability

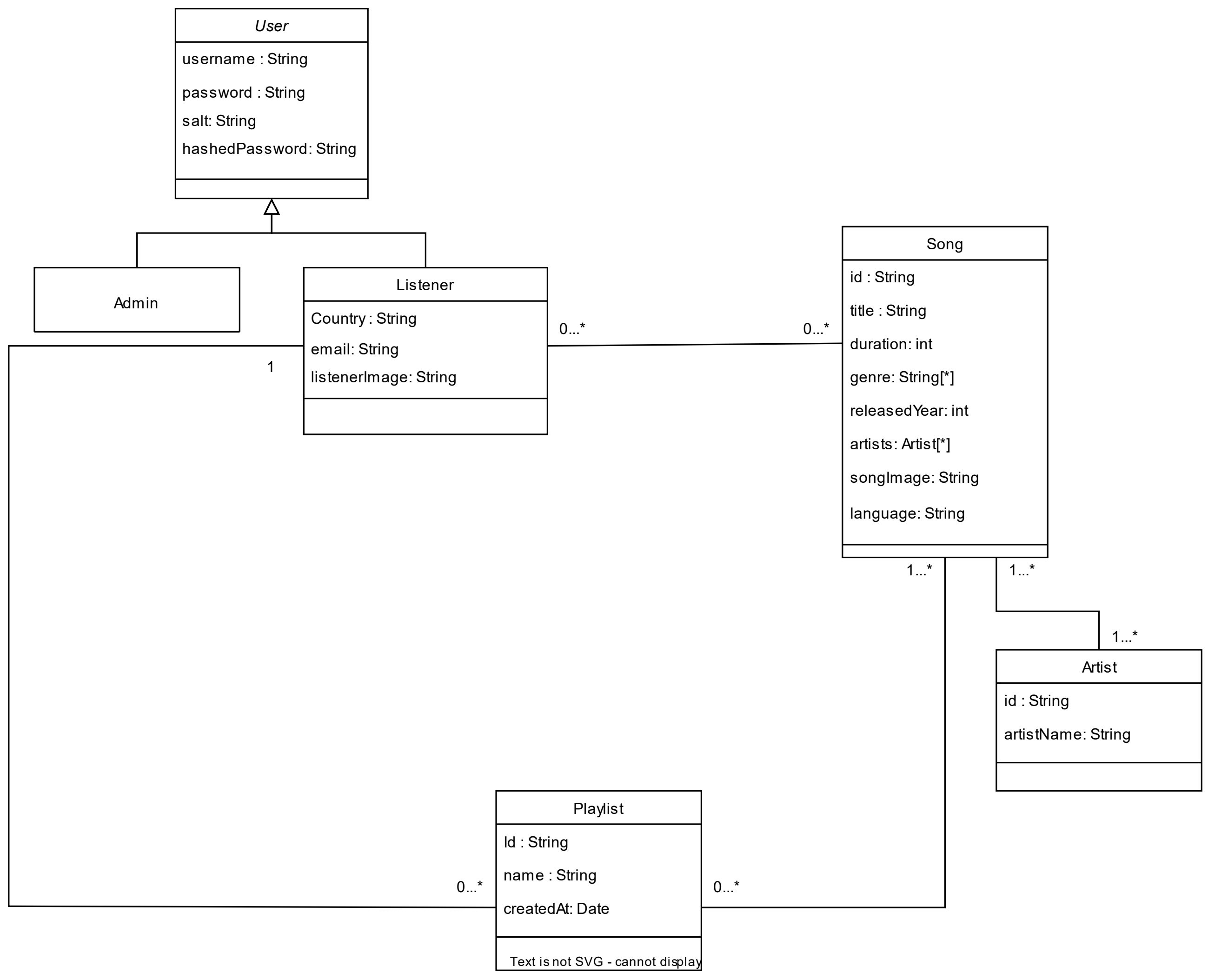
Usability Requirement	Description
USA-001	The application shall be user-friendly, as users shall interact with an intuitive graphical interface.

Use Cases Diagram



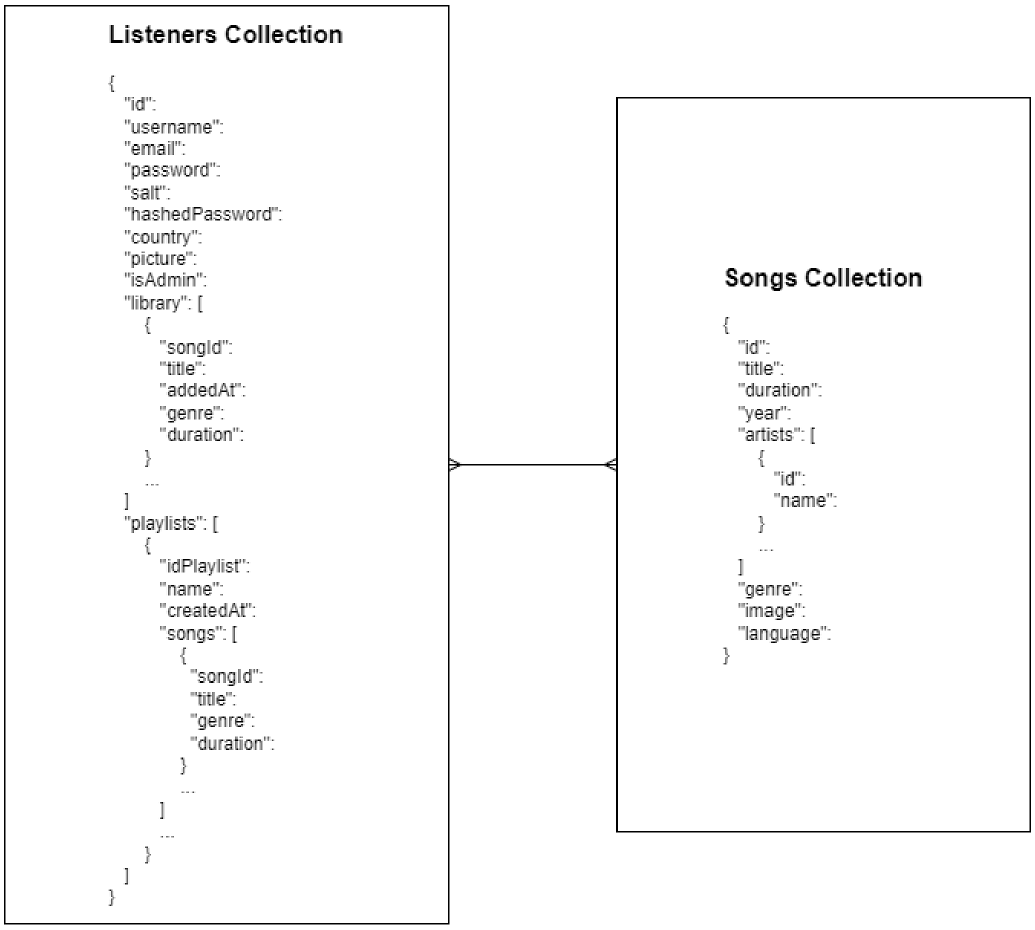
Sign up
Log in
Show suggested song...
Remove from pla...
Remove from Lib...
Add to playlist
<<include>>
<<extend>>
<<extend>>
<<extend>>
List...
Browse Songs
<<include>>
Find Song
<<include>>
View Song
<<extend>>
Remove from Lib...
<<include>>
<<include>>
View library
delete Playlist
Set as shared p...
<<extend>>
<<extend>>
Browse playlists
<<include>>
Find Playlist
<<include>>
View Playlist
Edit Profile
<<extend>>
View profile
<<extend>>
Delete Profile
Follow
Unfollow
<<extend>>
<<extend>>
Browse listeners
<<include>>
Find listener
<<include>>
View listener
Add song to fri...
Remove song to...
<<extend>>
<<extend>>
<<include>>
Find friend's...
Show...
<<extend>>
<<include>>
Browse friends
<<include>>
Find friend
<<extend>>
Browse friend's...
<<include>>
Find friend's...
Logout
Delete Song
Update Song
<<extend>>
<<extend>>
Admin
Browse Songs
<<include>>
Find Song
<<include>>
View Song
Insert New Song
View top favorite...
<<include>>
View Statistics
<<include>>
View listener pla...
<<include>>
View genre distri...
Text is not SVG - cannot display

Analysis Classes Diagram



Data Model

Mongo



Listeners Collection

```
{
  "id":
  "username":
  "email":
  "password":
  "salt":
  "hashedPassword":
  "country":
  "picture":
  "isAdmin":
  "library": [
     {
       "songId":
       "title":
       "addedAt":
       "genre":
       "duration":
     }
     ...
  ]
  "playlists": [
     {
       "idPlaylist":
       "name":
       "createdAt":
       "songs": [
          {
            "songId":
            "title":
            "genre":
            "duration":
          }
          ...
       ]
       ...
     }
  ]
}
```

Songs Collection

```
{
  "id":
  "title":
  "duration":
  "year":
  "artists": [
     {
       "id":
       "name":
     }
     ...
  ]
  "genre":
  "image":
  "language":
}
```

In the MongoDB data model for our application, playlists and the library are embedded within the listeners collection. This design decision is driven by the following considerations:

1. **Simplicity:**

- Embedding playlists within the listener collection simplifies queries for retrieving listener data and their associated playlists. This design allows for the retrieval of a listener and their playlists in a single database query, eliminating the need for complex "joining" queries.

2. **Atomic Updates:**

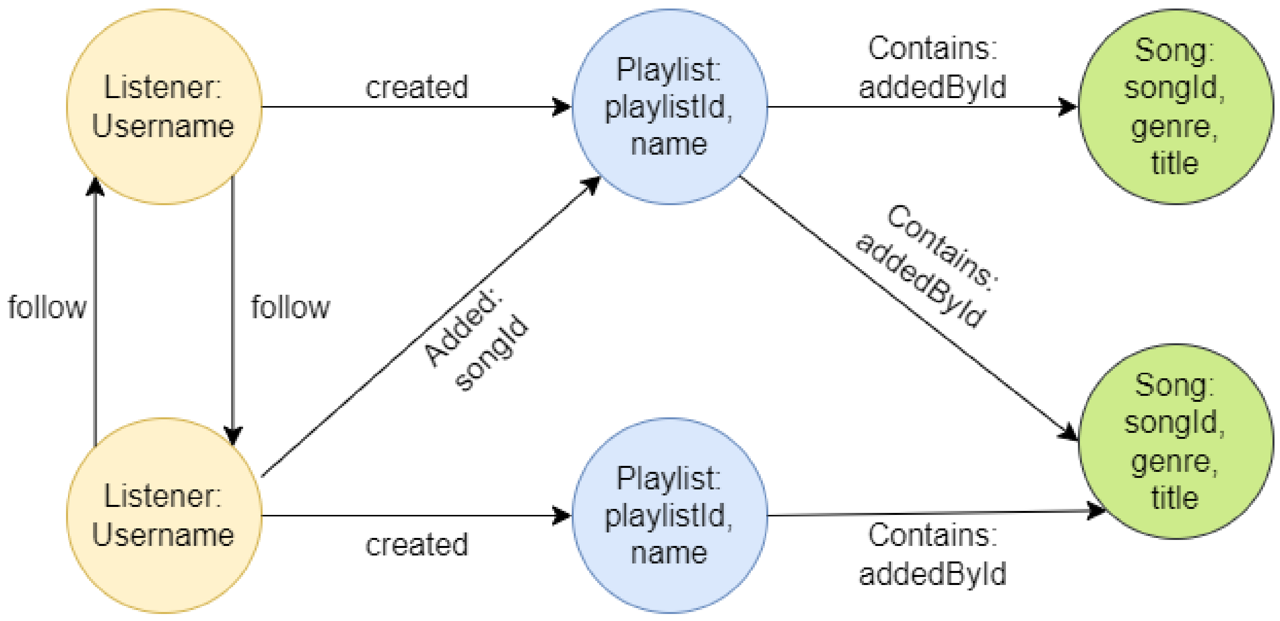
- By embedding playlists, updates to playlist data can be performed atomically within a single document. This approach ensures consistency in data updates, as the entire document, including playlists, can be modified in a single operation.

3. **Limits:**

- Anticipating that playlists won't experience significant growth, storing them as individual documents within the listener collection aligns with MongoDB's document-oriented nature. This design choice is suitable when individual documents are not expected to grow substantially.

The same rationale is applied to the inclusion of the library within the listeners collection, ensuring a cohesive and streamlined data model that aligns with the expected usage patterns and characteristics of the application.

Neo4j



In our Neo4j data model, the relationships are defined as follows:

1. **Listener - [:FOLLOWS] -> Listener:**

- Represents a user following another user.

2. **Listener - [:CREATED] -> Playlist:**

- Represents a listener sharing a playlist with other listeners who follow them.

3. **Listener - [:ADDED] -> Playlist:**
    - The songId attribute specifies the id of the added song.

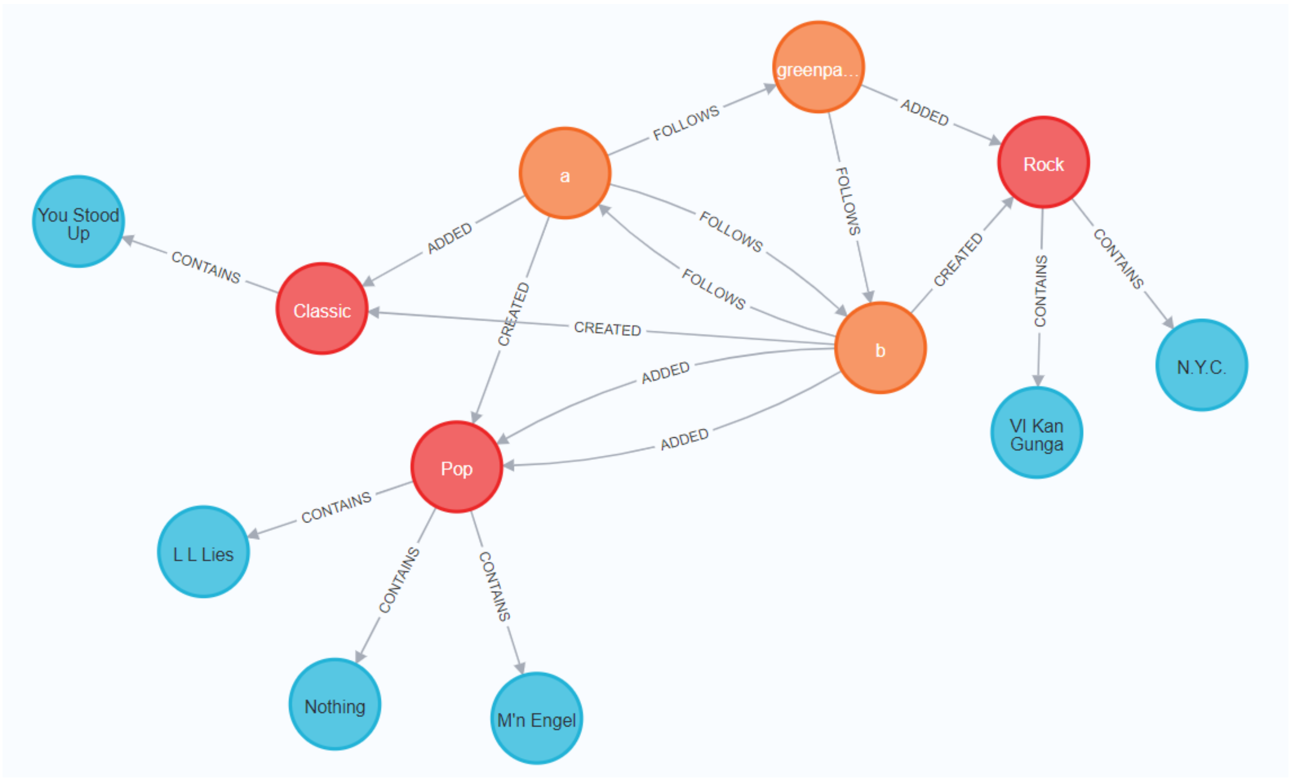
- Signifies a listener adding a song to a friend's playlist.

4. **Playlist - [:CONTAINS] -> Song:**

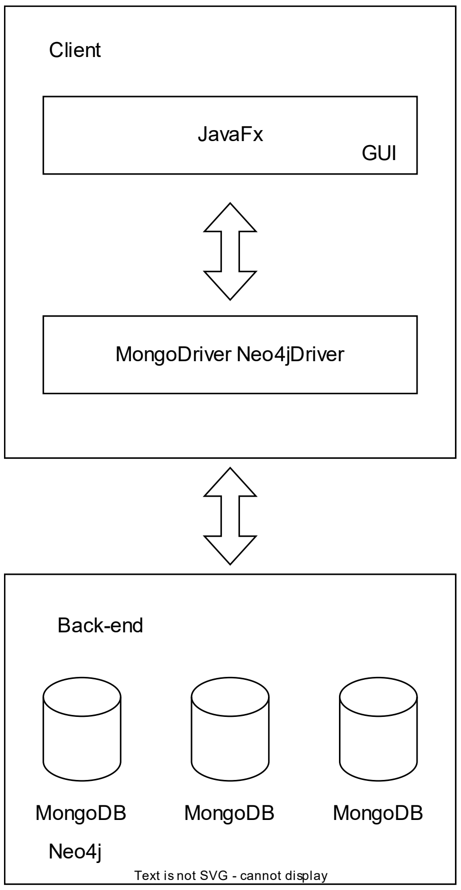
- Indicates that a playlist contains a specific song.
    - The addedById attribute specifies the listener who added the song to the playlist.

These relationships capture the social and music-sharing aspects of our application, providing a foundation for building meaningful connections and interactions within the Neo4j graph database.

Snapshot of the graph

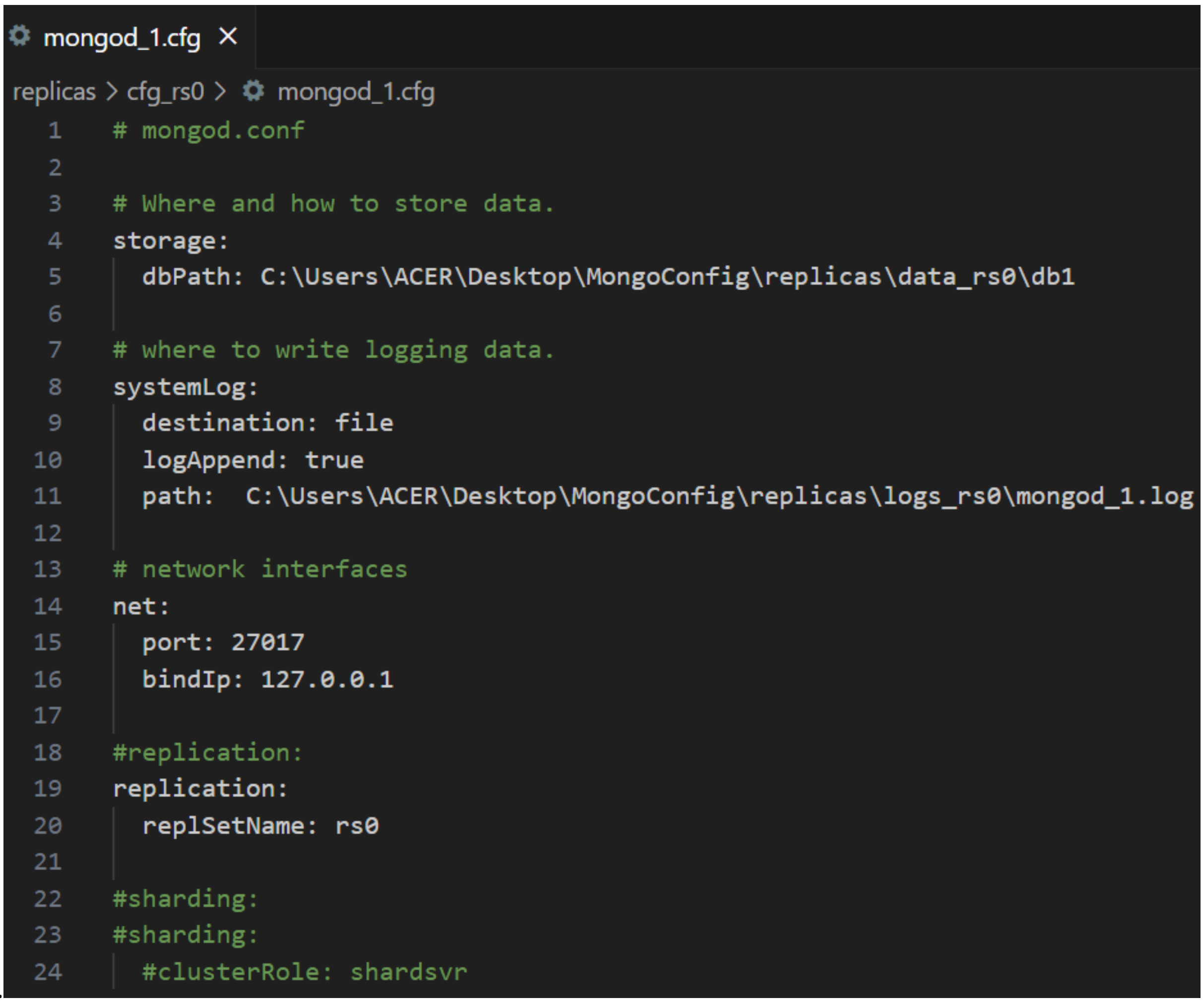


Architecture



Replica configuration

During the initial architectural design phase and subsequent implementation, a replica set comprising three data-bearing nodes (with no arbiters) was established. The configuration file for each replica is identical, differing only in port numbers



mongod_1.cfg

replicas > cfg_rs0 > mongod_1.cfg

```
# mongod.conf

# Where and how to store data.
storage:
  dbPath: C:\Users\ACER\Desktop\MongoConfig\replicas\data_rs0\db1

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path:  C:\Users\ACER\Desktop\MongoConfig\replicas\logs_rs0\mongod_1.log

# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1

#replication:
replication:
  replSetName: rs0

#sharding:
#sharding:
  #clusterRole: shardsvr
```

.

Primary Election Strategy:

The highest priority was assigned to localhost:27017, ensuring that, unless issues arise, the replica on this machine will be elected as the primary.

Use Cases and Write/Read Options:

Consideration was given to typical use cases for listeners, and the following write and read options were chosen:

Query	Read	Write
Sign-up	2	1
Sign-In	2	0
Search a song	1 each character pressed	0
Get song information	1	0
Create a new playlist	0	1
Add a song to playlist	0	1
Search a listener	1 each character pressed	0

Add a song to the library	0	1
View the library	1	0

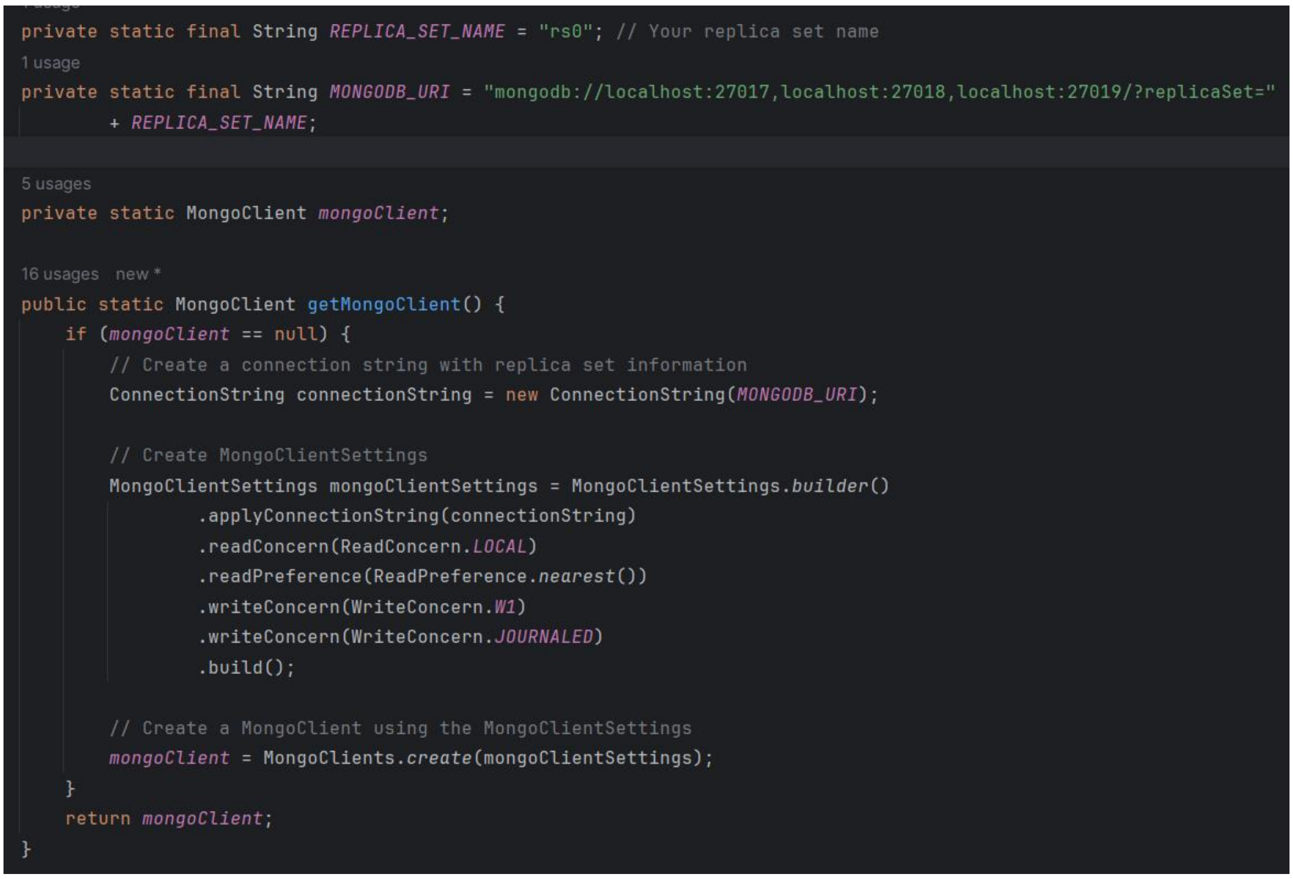
N.B. Read and write volumes are calculated considering the worst case.

Consistency and Availability Strategy:

Given that the application is read-heavy and prioritizes high availability and partition protection, the system leans toward the AP side of the CAP theorem. The chosen paradigm is Eventual Consistency.

Write Concern and Read Preferences:

We decided to introduce both Write Concern and Read Preferences constraints and set this configuration:



```
private static final String REPLICA_SET_NAME = "rs0"; // Your replica set name
1 usage
private static final String MONGODB_URI = "mongodb://localhost:27017,localhost:27018,localhost:27019/?replicaSet="
        + REPLICA_SET_NAME;

5 usages
private static MongoClient mongoClient;

16 usages   new *
public static MongoClient getMongoClient() {
    if (mongoClient == null) {
        // Create a connection string with replica set information
        ConnectionString connectionString = new ConnectionString(MONGODB_URI);

        // Create MongoClientSettings
        MongoClientSettings mongoClientSettings = MongoClientSettings.builder()
                .applyConnectionString(connectionString)
                .readConcern(ReadConcern.LOCAL)
                .readPreference(ReadPreference.nearest())
                .writeConcern(WriteConcern.W1)
                .writeConcern(WriteConcern.JOURNALED)
                .build();

        // Create a MongoClient using the MongoClientSettings
        mongoClient = MongoClients.create(mongoClientSettings);
    }
    return mongoClient;
}
```

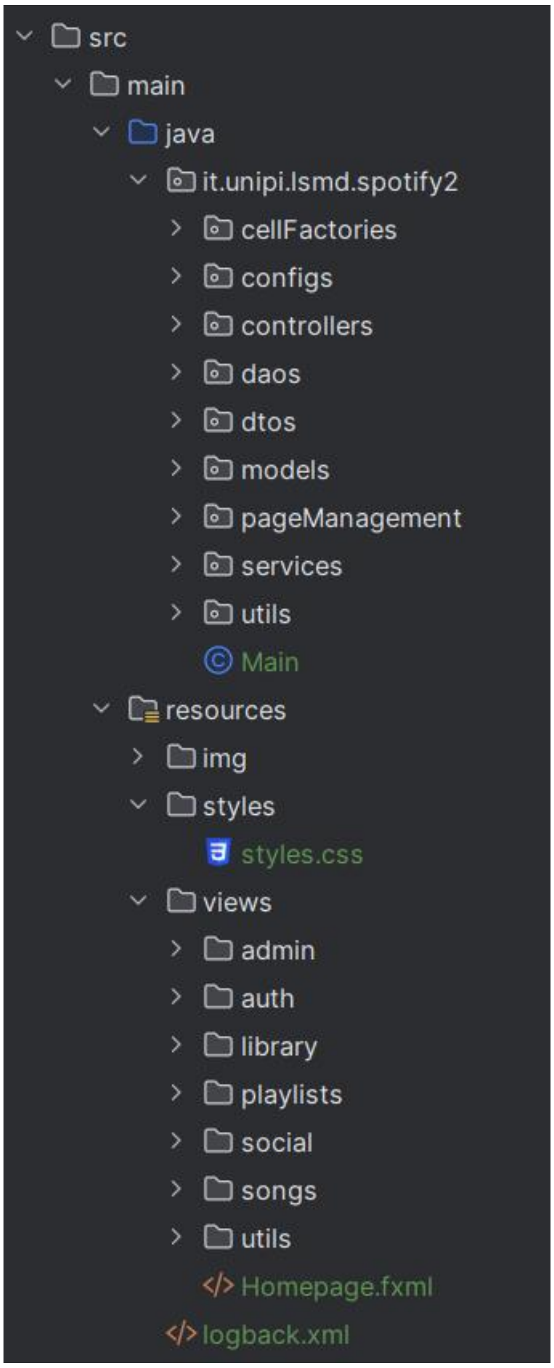
- **Read Concern = "local"**
  the read operation can return data that is local to the node handling the query, even if the data has not been replicated to other nodes.
- **Read Preference = "nearest"**
  MongoDB will route read operations to the replica set member (either a secondary or the primary) that is geographically closest in terms of network latency.
- **Write Concern = 1 (w:1)**
  the write operation is considered successful once the primary node acknowledges that it has received the write request and has written the data to its memory.

- **Journaled Writes**
  In MongoDB, the journal is a write-ahead log that records changes to data before they are applied to the data files. This journaling mechanism helps ensure durability and recoverability of data in case of unexpected shutdowns or failures. When journaled writes are enabled, MongoDB writes data to the journal before acknowledging the write operation as successful.
- **Election Timeout** and **heartbeatIntervalMillis** are left at the default settings (Default: 10,000 milliseconds or 10 seconds, 2000 milliseconds or 2 seconds respectively).

Mongo Implementation

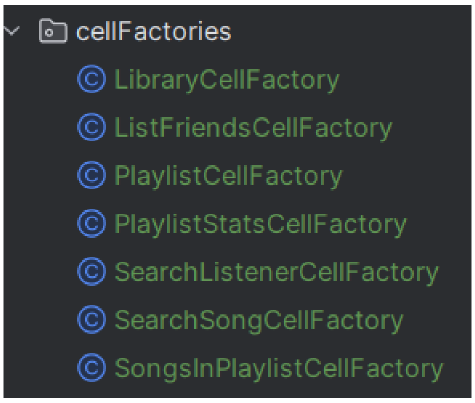
Application project source organization

The app follows the Model-View-Controller (MVC) architectural pattern.

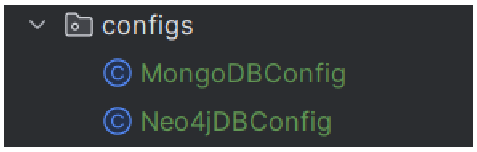


```
src
  main
    java
      it.unipi.lsmd.spotify2
        cellFactories
        configs
        controllers
        daos
        dtos
        models
        pageManagement
        services
        utils
        Main
    resources
      img
      styles
        styles.css
      views
        admin
        auth
        library
        playlists
        social
        songs
        utils
        Homepage.fxml
      logback.xml
```

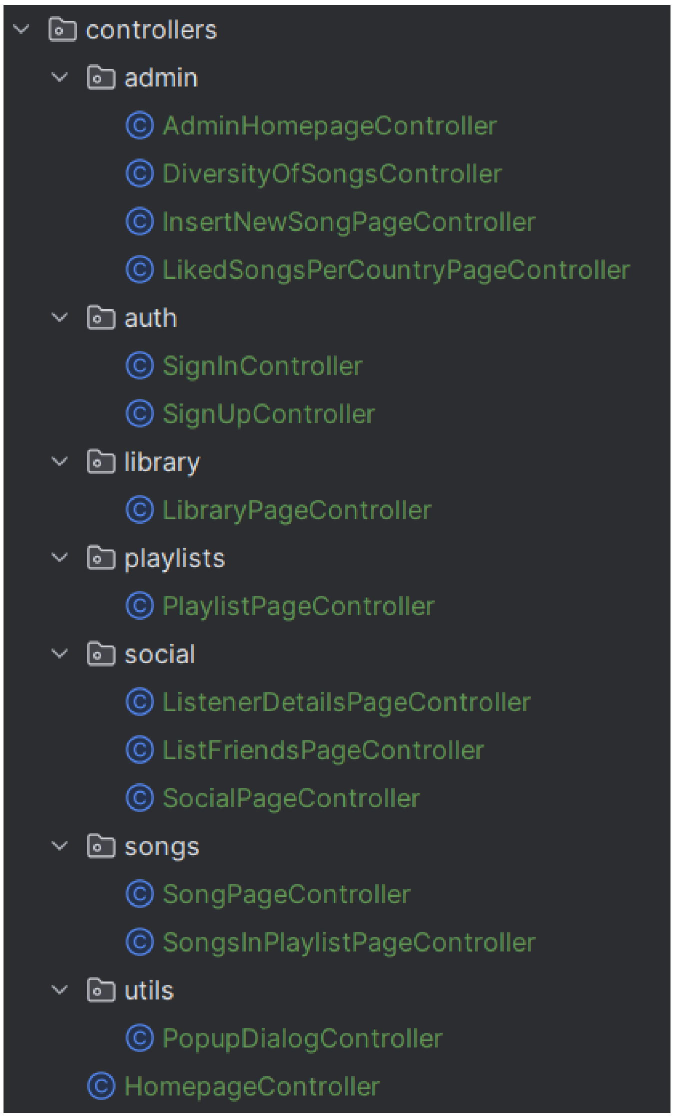
cellFactories: The primary purpose of this package is to define how the data in each cell should be displayed. It allows developers to customize the appearance and behavior of cells based on the underlying data.



config: this package contains classes responsible for configuring the MongoDB and Neo4j client, database connection settings, and any other related configuration parameters (like read preferences and write concerns). This package helps centralize and organize the configuration logic, making it easier to manage and maintain.



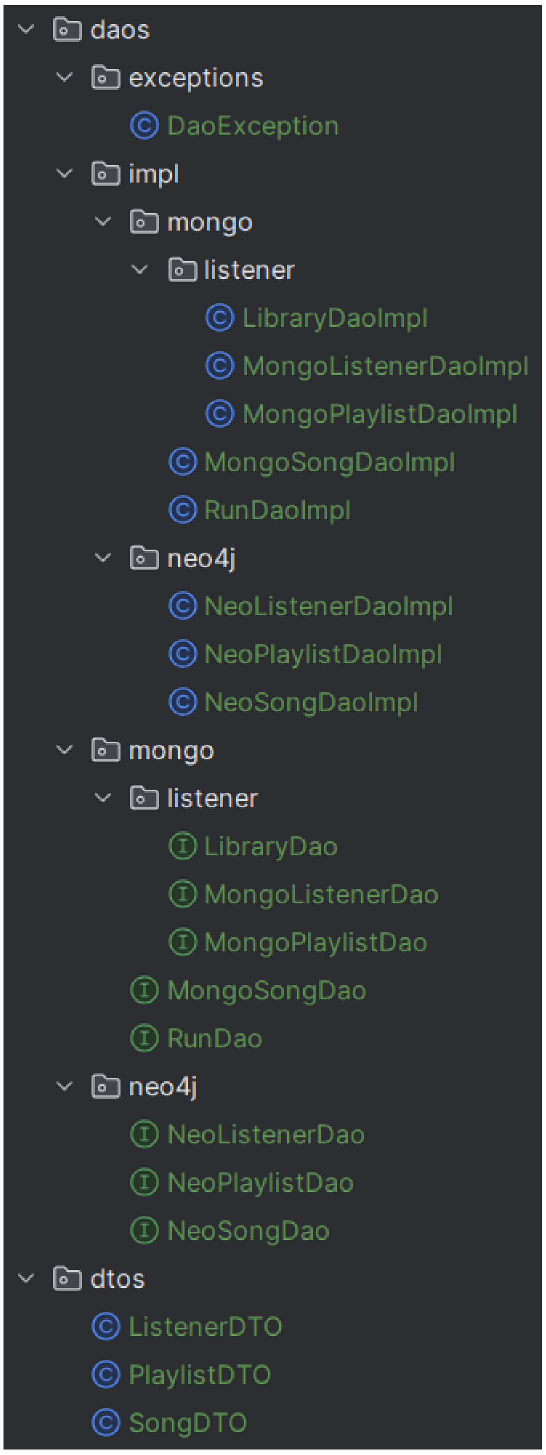
controllers: this package contains all the controllers used to handle user interactions with the graphical user interface (GUI). They ensure that the GUI accurately reflects the underlying data and business logic.



```
controllers
  admin
    AdminHomepageController
    DiversityOfSongsController
    InsertNewSongPageController
    LikedSongsPerCountryPageController
  auth
    SignInController
    SignUpController
  library
    LibraryPageController
  playlists
    PlaylistPageController
  social
    ListenerDetailsPageController
    ListFriendsPageController
    SocialPageController
  songs
    SongPageController
    SongsInPlaylistPageController
  utils
    PopupDialogController
  HomepageController
```

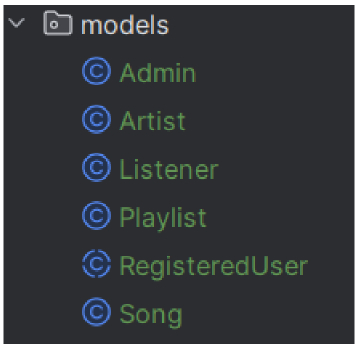
Daos: DAO stands for Data Access Object. A DAO is a design pattern that provides an abstract interface to some type of database or other persistence mechanism. The main purpose of a DAO is to separate the application's business logic from the low-level details of accessing the data storage.

Dtos: DTO stands for Data Transfer Object. It is a design pattern used to transfer data between software application subsystems or layers, typically between the business logic layer and the presentation or persistence layer. The primary purpose of DTOs is to encapsulate and transport data across different parts of an application in a structured and efficient manner.



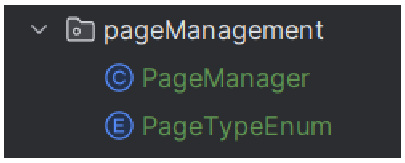
```
daos
  exceptions
    DaoException
  impl
    mongo
      listener
        LibraryDaoImpl
        MongoListenerDaoImpl
        MongoPlaylistDaoImpl
      MongoSongDaoImpl
      RunDaoImpl
    neo4j
      NeoListenerDaoImpl
      NeoPlaylistDaoImpl
      NeoSongDaoImpl
  mongo
    listener
      LibraryDao
      MongoListenerDao
      MongoPlaylistDao
    MongoSongDao
    RunDao
  neo4j
    NeoListenerDao
    NeoPlaylistDao
    NeoSongDao
dtos
  ListenerDTO
  PlaylistDTO
  SongDTO
```

models: the package contains the classes that represent entities in the domain.



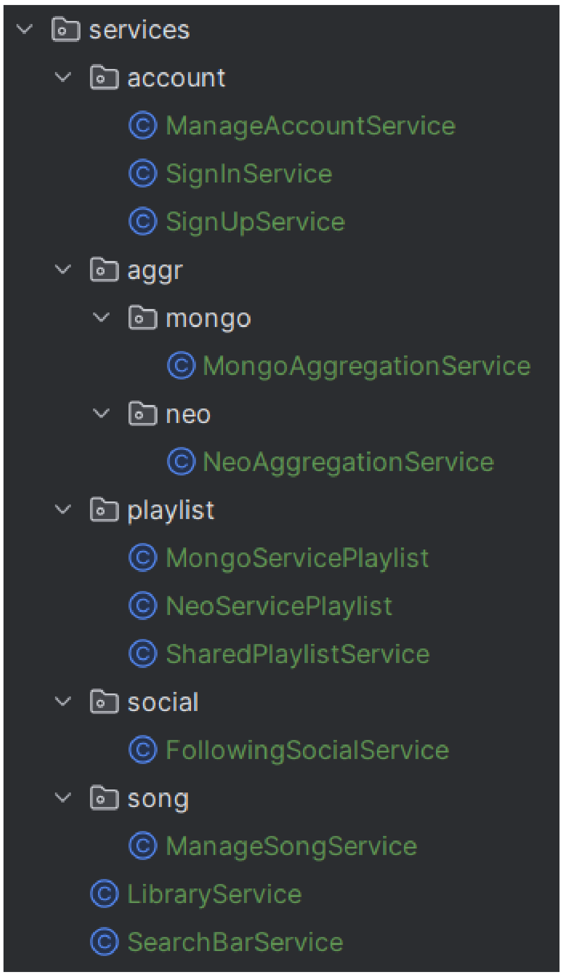
```
models
    Admin
    Artist
    Listener
    Playlist
    RegisteredUser
    Song
```

Page management: this package contains the classes or components responsible for managing navigation between different pages. “PageTypeEnum” contains the contasts representing the various pages.



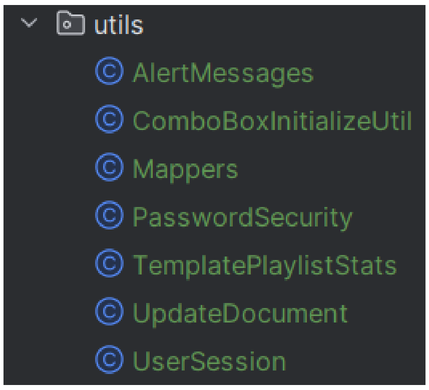
```
pageManagement
    PageManager
    PageTypeEnum
```

Services: They are service interfaces that declare the methods that the service provides, allowing for abstraction and easy replacement of implementations. Furthermore, these classes are responsible for managing transactions. They ensure that operations are executed atomically, and changes are committed or rolled back appropriately.



```
services
    account
        ManageAccountService
        SignInService
        SignUpService
    aggr
        mongo
            MongoAggregationService
        neo
            NeoAggregationService
    playlist
        MongoServicePlaylist
        NeoServicePlaylist
        SharedPlaylistService
    social
        FollowingSocialService
    song
        ManageSongService
    LibraryService
    SearchBarService
```

Utils: contains utility classes or helper functions that provide general-purpose functionality across different parts of the codebase. Includes classes for showing alert messages, mapping objects, retrieving user session information, etc.



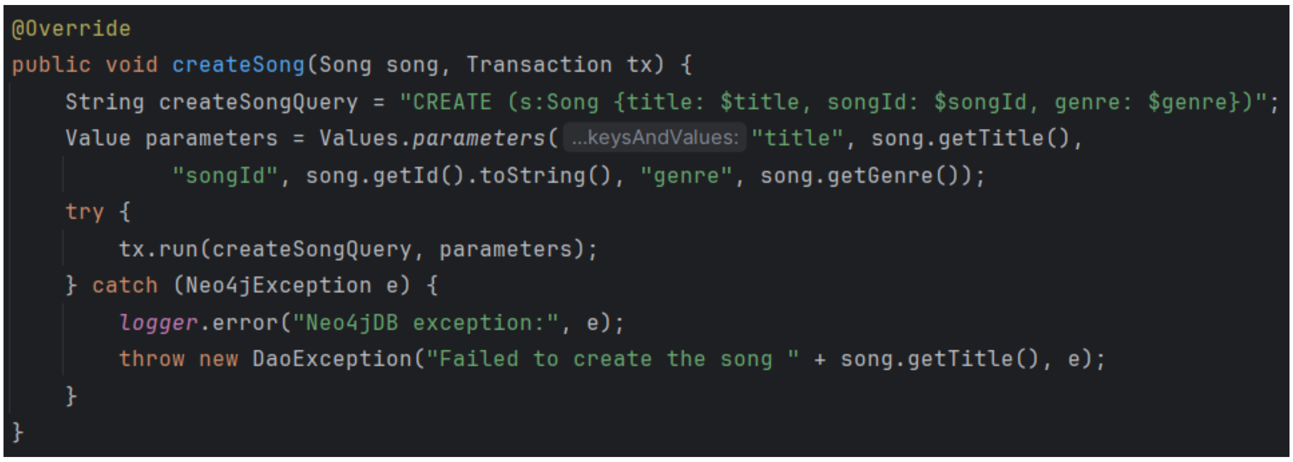
```
utils
    AlertMessages
    ComboBoxInitializeUtil
    Mappers
    PasswordSecurity
    TemplatePlaylistStats
    UpdateDocument
    UserSession
```

The folder ***resources*** contain all the fxml files which represent the views that are loaded.

Mongo CRUD operations

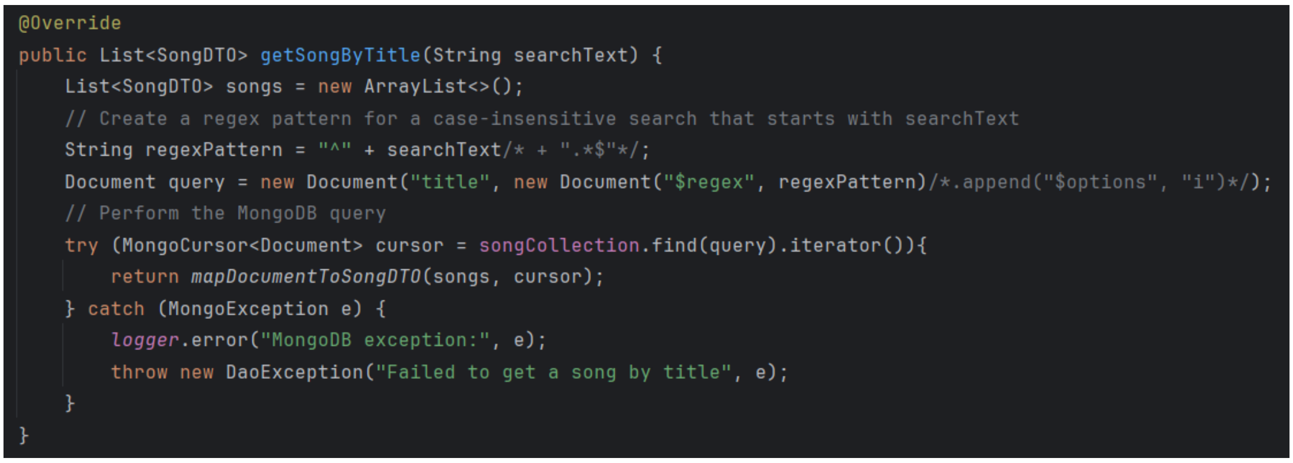
Here below are displayed the most common CRUD operations.

Create



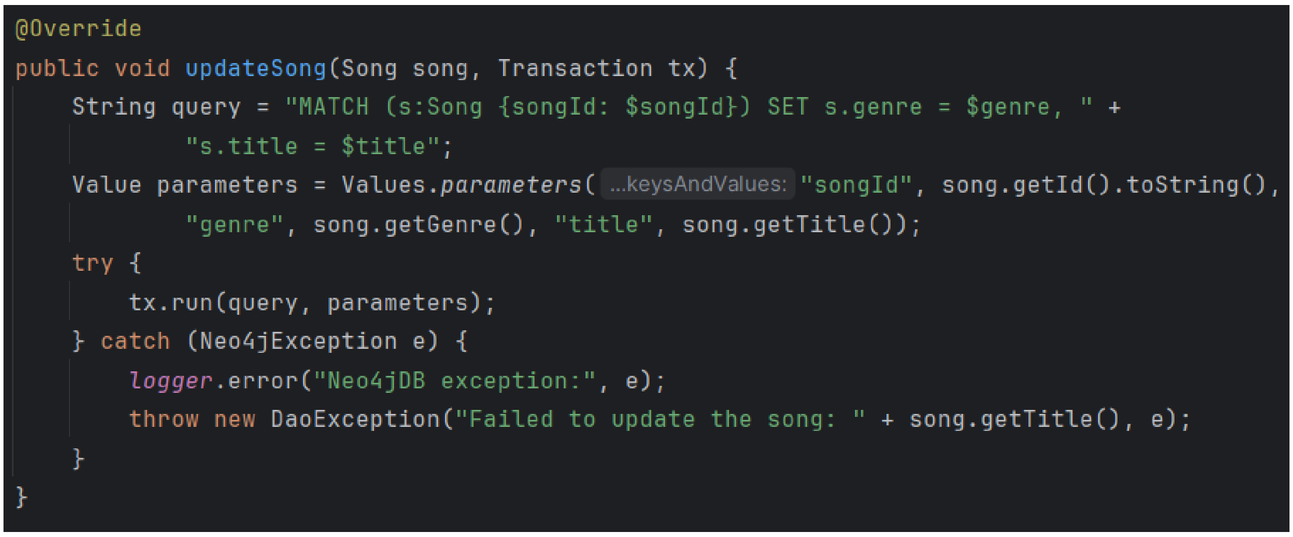
```java
@Override
public void createSong(Song song, Transaction tx) {
    String createSongQuery = "CREATE (s:Song {title: $title, songId: $songId, genre: $genre})";
    Value parameters = Values.parameters( ...keysAndValues: "title", song.getTitle(),
            "songId", song.getId().toString(), "genre", song.getGenre());
    try {
        tx.run(createSongQuery, parameters);
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to create the song " + song.getTitle(), e);
    }
}
```

Read



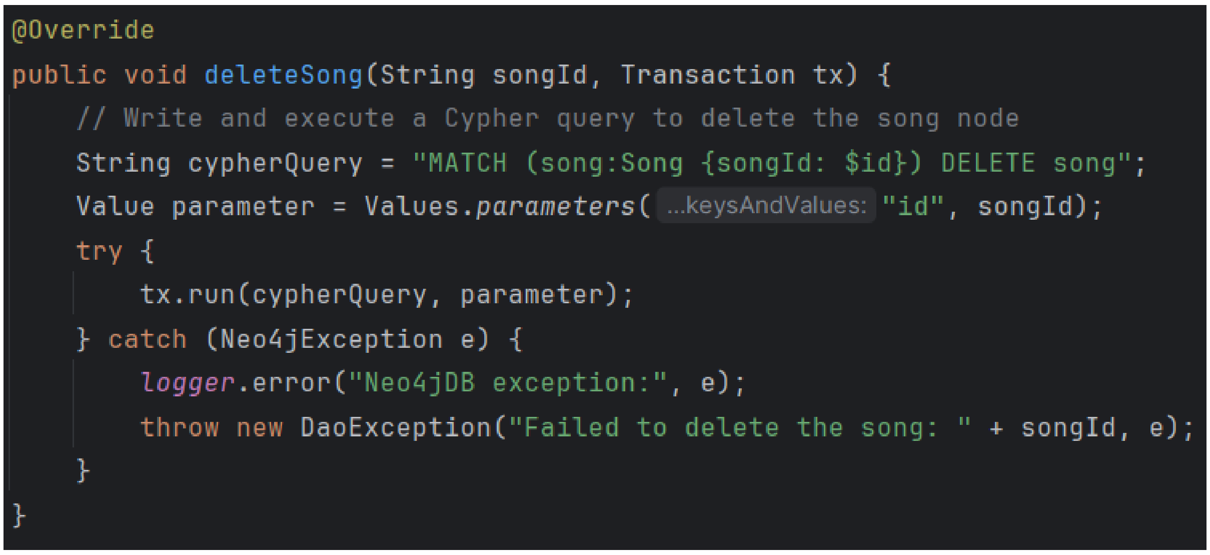
```java
@Override
public List<SongDTO> getSongByTitle(String searchText) {
    List<SongDTO> songs = new ArrayList<>();
    // Create a regex pattern for a case-insensitive search that starts with searchText
    String regexPattern = "^" + searchText/* + ".*$"*/;
    Document query = new Document("title", new Document("$regex", regexPattern)/*.append("$options", "i")*/);
    // Perform the MongoDB query
    try (MongoCursor<Document> cursor = songCollection.find(query).iterator()){
        return mapDocumentToSongDTO(songs, cursor);
    } catch (MongoException e) {
        logger.error("MongoDB exception:", e);
        throw new DaoException("Failed to get a song by title", e);
    }
}
```

Update



```java
@Override
public void updateSong(Song song, Transaction tx) {
    String query = "MATCH (s:Song {songId: $songId}) SET s.genre = $genre, " +
            "s.title = $title";
    Value parameters = Values.parameters( ...keysAndValues: "songId", song.getId().toString(),
            "genre", song.getGenre(), "title", song.getTitle());
    try {
        tx.run(query, parameters);
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to update the song: " + song.getTitle(), e);
    }
}
```

Delete



```java
@Override
public void deleteSong(String songId, Transaction tx) {
    // Write and execute a Cypher query to delete the song node
    String cypherQuery = "MATCH (song:Song {songId: $id}) DELETE song";
    Value parameter = Values.parameters( ...keysAndValues: "id", songId);
    try {
        tx.run(cypherQuery, parameter);
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to delete the song: " + songId, e);
    }
}
```

Mongo aggregation queries

Query: Top ten most liked songs in a certain country

This MongoDB aggregation query is designed to find the top ten most liked songs in a specific country.

```
db.listeners.aggregate([
  // Match users from a specific country
  { $match: { country: "YourCountryName" } },

  // Unwind the library array
  { $unwind: "$library" },
```

// Group the songs and count their occurrences

{ $group: { _id: "$library.songId", count: { $sum: 1 } } },

// Sort the songs by count in descending order

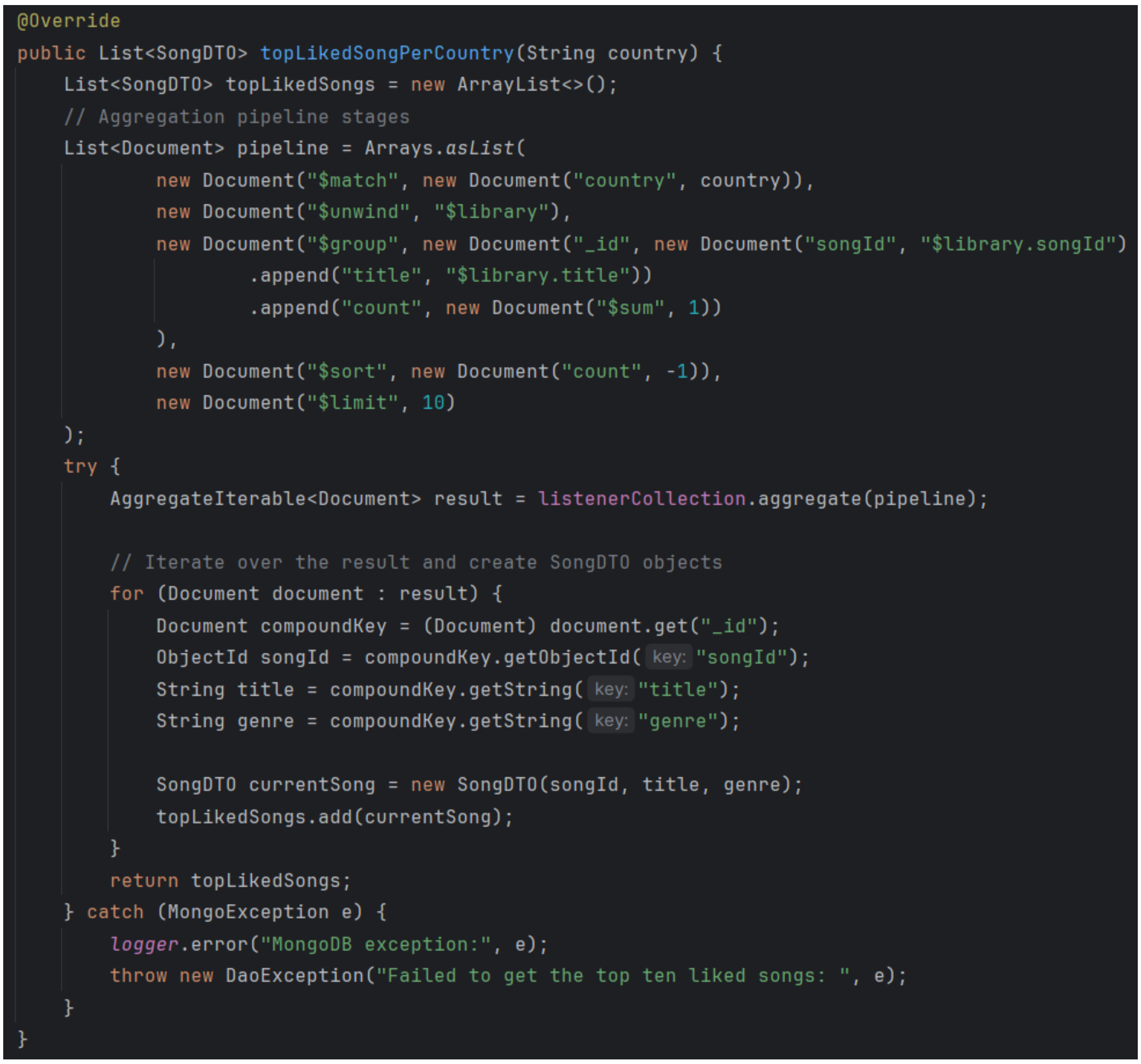
{ $sort: { count: -1 } },

// Limit the result to the top ten songs

{ $limit: 10 }

In summary, this aggregation pipeline starts by filtering listeners from a specific country, then deconstructs the library array, groups songs by their songId, counts their occurrences, sorts them in descending order by count, and finally limits the result to the top ten songs.

Here there is a snippet of the code:

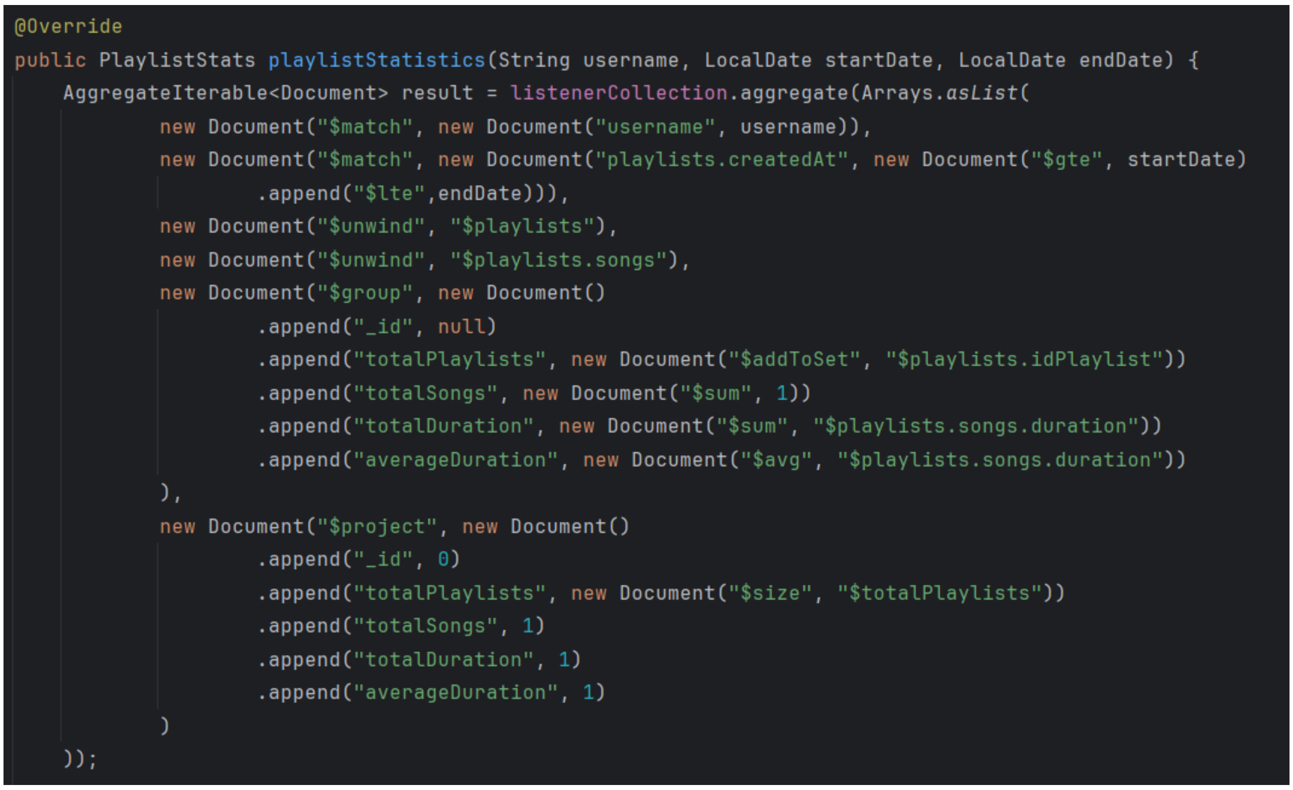


```java
@Override
public List<SongDTO> topLikedSongPerCountry(String country) {
    List<SongDTO> topLikedSongs = new ArrayList<>();
    // Aggregation pipeline stages
    List<Document> pipeline = Arrays.asList(
            new Document("$match", new Document("country", country)),
            new Document("$unwind", "$library"),
            new Document("$group", new Document("_id", new Document("songId", "$library.songId")
                    .append("title", "$library.title"))
                    .append("count", new Document("$sum", 1))
            ),
            new Document("$sort", new Document("count", -1)),
            new Document("$limit", 10)
    );
    try {
        AggregateIterable<Document> result = listenerCollection.aggregate(pipeline);

        // Iterate over the result and create SongDTO objects
        for (Document document : result) {
            Document compoundKey = (Document) document.get("_id");
            ObjectId songId = compoundKey.getObjectId( key: "songId");
            String title = compoundKey.getString( key: "title");
            String genre = compoundKey.getString( key: "genre");

            SongDTO currentSong = new SongDTO(songId, title, genre);
            topLikedSongs.add(currentSong);
        }
        return topLikedSongs;
    } catch (MongoException e) {
        logger.error("MongoDB exception:", e);
        throw new DaoException("Failed to get the top ten liked songs: ", e);
    }
}
```

Query: Playlists Statistics for a listener

The purpose of this MongoDB aggregation query is to retrieve aggregated information about playlists for a specific listener within a specified date range. The resulting metrics include number of palylists, total songs, total duration, average song length.

```
db.listenerCollection.aggregate([
  { $match: { username: "yourUsername", "playlists.createdAt": { $gte: new Date("startDate"), $lte: new Date("endDate") } } },
  { $unwind: "$playlists" },
  { $unwind: "$playlists.songs" },
  { $group: {
    _id: null,
    totalPlaylists: { $addToSet: "$playlists.idPlaylist" },
    totalSongs: { $sum: 1 },
    totalDuration: { $sum: "$playlists.songs.duration" },
    averageDuration: { $avg: "$playlists.songs.duration" }
  } },
  { $project: {
    _id: 0,
    totalPlaylists: { $size: "$totalPlaylists" },
    totalSongs: 1,
    totalDuration: 1,
    averageDuration: 1
  } }
])
```



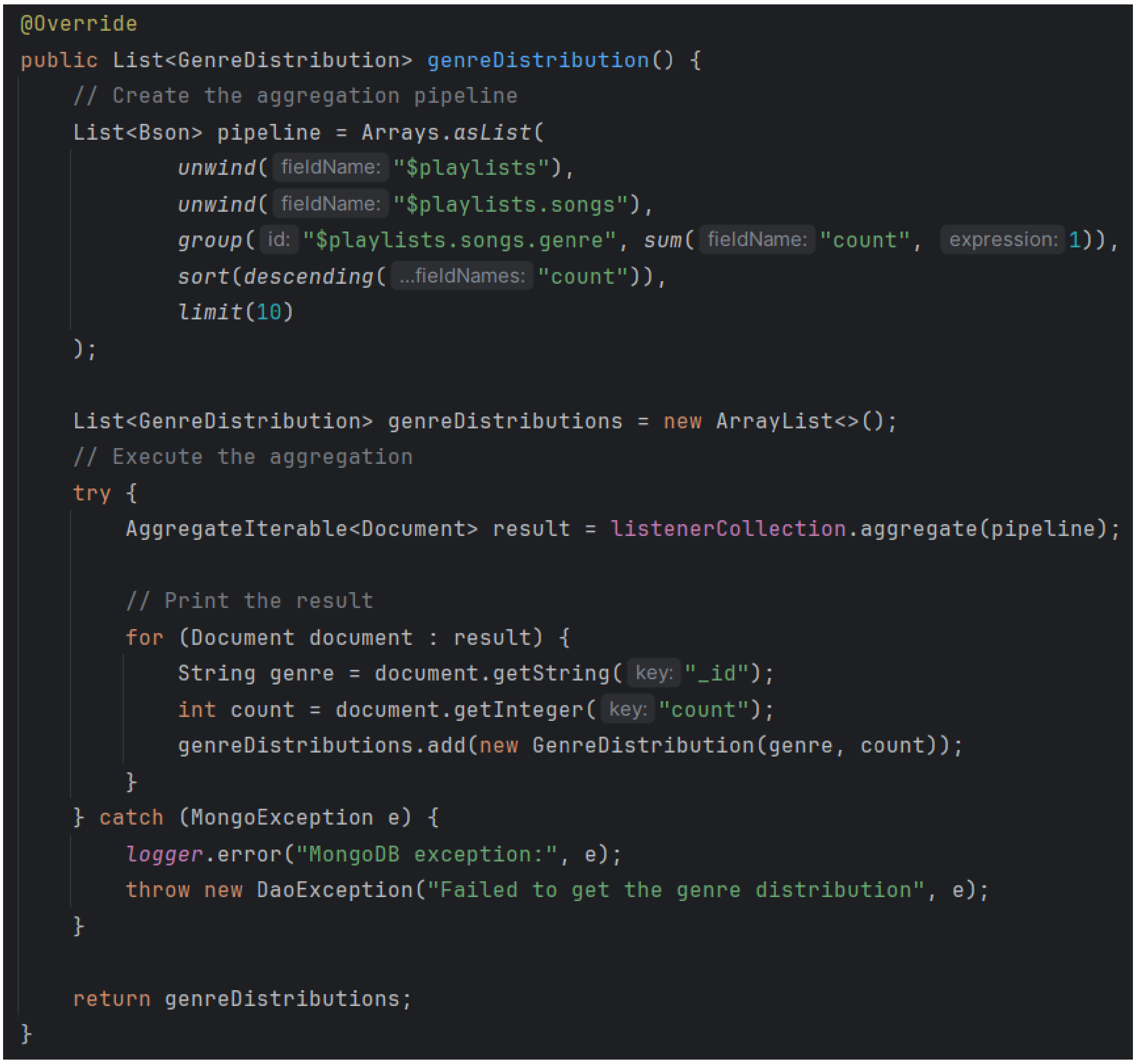
```java
@Override
public PlaylistStats playlistStatistics(String username, LocalDate startDate, LocalDate endDate) {
    AggregateIterable<Document> result = listenerCollection.aggregate(Arrays.asList(
            new Document("$match", new Document("username", username)),
            new Document("$match", new Document("playlists.createdAt", new Document("$gte", startDate)
                    .append("$lte",endDate))),
            new Document("$unwind", "$playlists"),
            new Document("$unwind", "$playlists.songs"),
            new Document("$group", new Document()
                    .append("_id", null)
                    .append("totalPlaylists", new Document("$addToSet", "$playlists.idPlaylist"))
                    .append("totalSongs", new Document("$sum", 1))
                    .append("totalDuration", new Document("$sum", "$playlists.songs.duration"))
                    .append("averageDuration", new Document("$avg", "$playlists.songs.duration"))
            ),
            new Document("$project", new Document()
                    .append("_id", 0)
                    .append("totalPlaylists", new Document("$size", "$totalPlaylists"))
                    .append("totalSongs", 1)
                    .append("totalDuration", 1)
                    .append("averageDuration", 1)
            )
    ));
```

Query: genre distribution across playlists

This query is using the MongoDB aggregation framework to analyze the distribution of song genres across playlists

```
var listenersCollection = db.getCollection("listeners");


// Create the extended aggregation pipeline

var pipeline = [

  { $unwind: "$playlists" },

  { $unwind: "$playlists.songs" },

  { $group: { _id: "$playlists.songs.genre", count: { $sum: 1 } } },

  { $sort: { count: -1 } }

];

// Execute the aggregation

var result = listenersCollection.aggregate(pipeline);
```

The result of this query will be a list of genres along with the count of songs for each genre, sorted in descending order based on the number of occurrences. This information can be useful for understanding the popularity of different genres across all playlists and listeners in the collection.



```
@Override
public List<GenreDistribution> genreDistribution() {
    // Create the aggregation pipeline
    List<Bson> pipeline = Arrays.asList(
            unwind( fieldName: "$playlists"),
            unwind( fieldName: "$playlists.songs"),
            group( id: "$playlists.songs.genre", sum( fieldName: "count",  expression: 1)),
            sort(descending( ...fieldNames: "count")),
            limit(10)
    );

    List<GenreDistribution> genreDistributions = new ArrayList<>();
    // Execute the aggregation
    try {
        AggregateIterable<Document> result = listenerCollection.aggregate(pipeline);

        // Print the result
        for (Document document : result) {
            String genre = document.getString( key: "_id");
            int count = document.getInteger( key: "count");
            genreDistributions.add(new GenreDistribution(genre, count));
        }
    } catch (MongoException e) {
        logger.error("MongoDB exception:", e);
        throw new DaoException("Failed to get the genre distribution", e);
    }

    return genreDistributions;
}
```

Indexes Study

The MongoDB Compass Explain Plan section provides a tool for evaluating query performance in the database. We utilized this feature to compare the performance of queries that utilize indexes against those that do not.

This study specifically focuses on evaluating the query performance of the 'listeners' and 'songs' collections in our MongoDB database. The goal is to compare the results of queries executed both before and after the implementation of indexes.

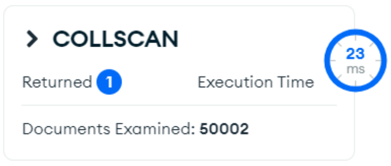
The following tables display the Average Actual Query Execution Time (ms), representing the average time obtained from a sample of 10 executions.

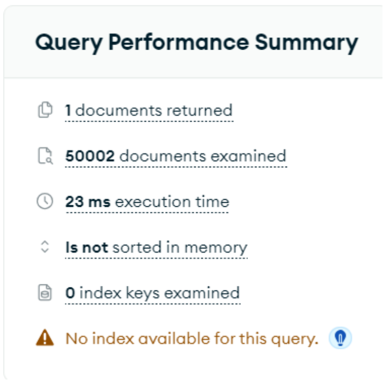
For added context, the index creation was carried out specifically for the 'listeners' and 'songs' collections. The selected queries represent common operations on these collections, and the study aims to assess how the introduction of indexes impacts the average query execution time.

Listeners collection (index: username)

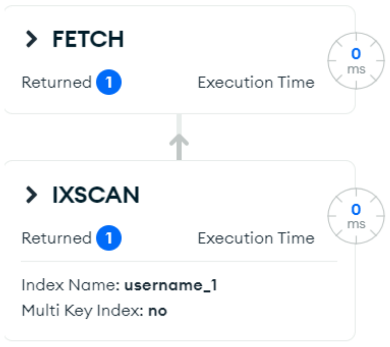
Query 1: spotify2.listeners.find({username: "greenpanda549"})

Without index:

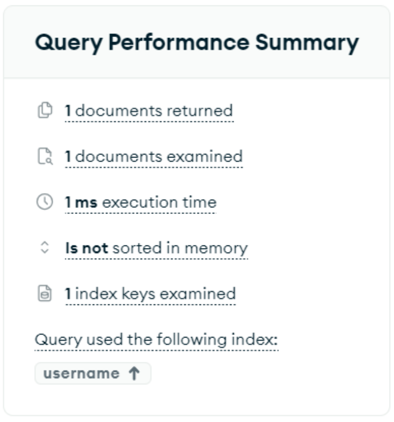

COLLSCAN
Returned 1 Execution Time 23 ms
Documents Examined: **50002**


Query Performance Summary
- 1 documents returned
- **50002** documents examined
- **23 ms** execution time
- **Is not** sorted in memory
- **0** index keys examined
- No index available for this query.

With index:


FETCH
Returned 1 Execution Time 0 ms

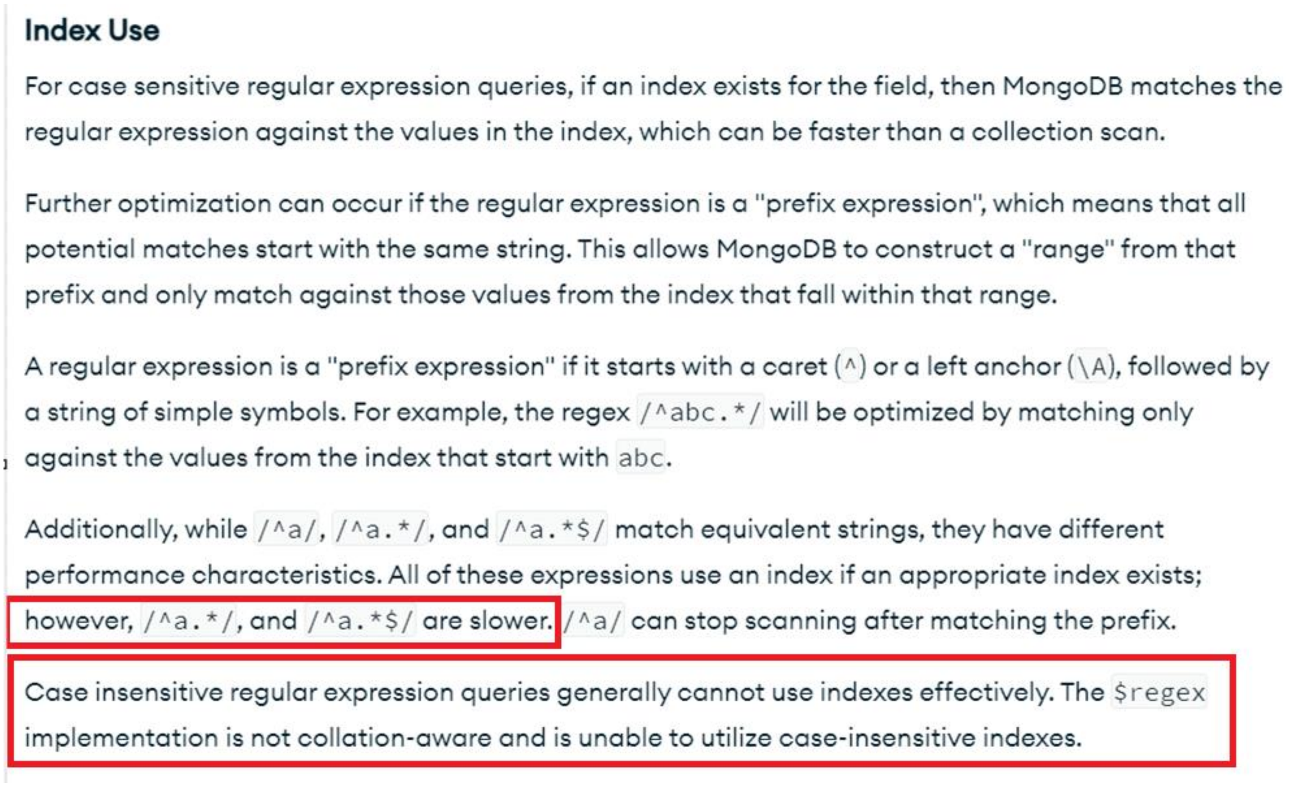
IXSCAN
Returned 1 Execution Time 0 ms
Index Name: **username_1**
Multi Key Index: **no**


Query Performance Summary
- 1 documents returned
- 1 documents examined
- **1 ms** execution time
- **Is not** sorted in memory
- 1 index keys examined

Query used the following index:
username ↑

used index	document returned	Index keys examined	Document examined	Average Actual Query Execution Time(ms)
no index used	1	0	50002	26
username_1	1	1	1	0

Query 2: spotify2.listeners.find({username: {$regex: '^g'}})



Index Use

For case sensitive regular expression queries, if an index exists for the field, then MongoDB matches the regular expression against the values in the index, which can be faster than a collection scan.

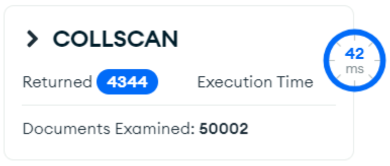
Further optimization can occur if the regular expression is a "prefix expression", which means that all potential matches start with the same string. This allows MongoDB to construct a "range" from that prefix and only match against those values from the index that fall within that range.

A regular expression is a "prefix expression" if it starts with a caret (`^`) or a left anchor (`\A`), followed by a string of simple symbols. For example, the regex `/^abc.*/` will be optimized by matching only against the values from the index that start with `abc`.

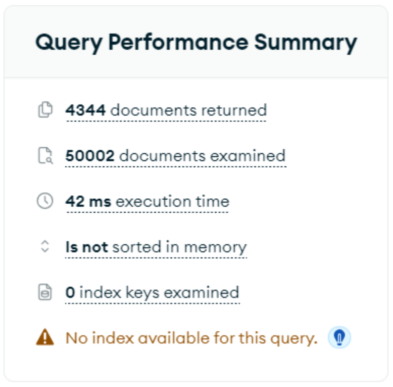
Additionally, while `/^a/`, `/^a.*/`, and `/^a.*$/` match equivalent strings, they have different performance characteristics. All of these expressions use an index if an appropriate index exists; however, `/^a.*/`, and `/^a.*$/` are slower. `/^a/` can stop scanning after matching the prefix.

Case insensitive regular expression queries generally cannot use indexes effectively. The `$regex` implementation is not collation-aware and is unable to utilize case-insensitive indexes.

Without index:



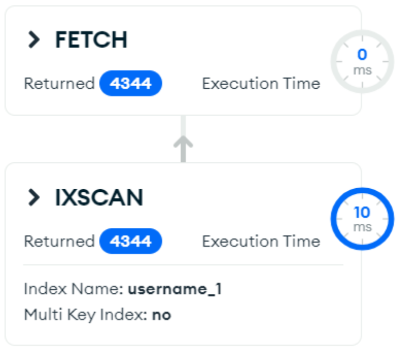
COLLSCAN
Returned 4344 Execution Time 42 ms
Documents Examined: 50002



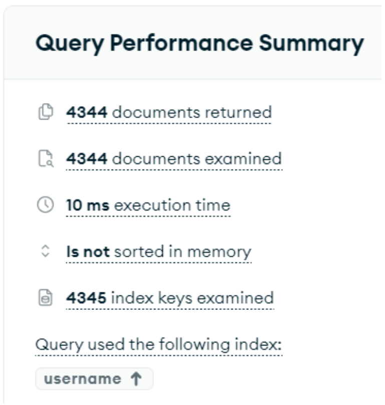
Query Performance Summary

- 4344 documents returned
- 50002 documents examined
- 42 ms execution time
- Is not sorted in memory
- 0 index keys examined
- No index available for this query.

With index:


FETCH
Returned 4344 Execution Time 0 ms

IXSCAN
Returned 4344 Execution Time 10 ms
Index Name: **username_1**
Multi Key Index: **no**


Query Performance Summary

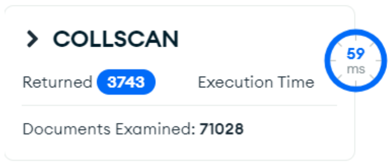
4344 documents returned
4344 documents examined
10 ms execution time
Is not sorted in memory
4345 index keys examined
Query used the following index:
username ↑

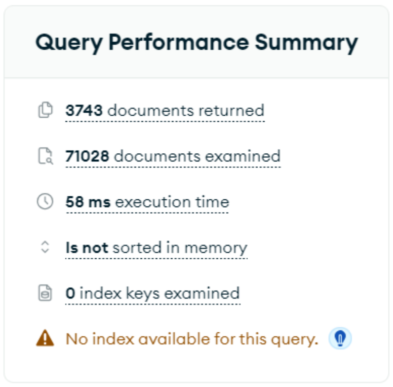
used index	document returned	Index keys examined	Document examined	Average Actual Query Execution Time(ms)
no index used	4344	0	50002	42.5
username_1	4344	1	4344	9.6

Songs collection (indexes: title, artists)

Queri 1: spotify2.songs.find({title: {$regex: '^c'}})

Without index:


COLLSCAN
Returned 3743 Execution Time 59 ms
Documents Examined: **71028**

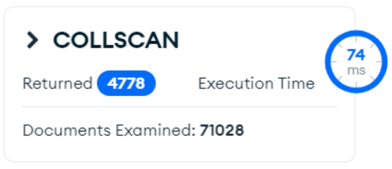

Query Performance Summary

3743 documents returned
71028 documents examined
58 ms execution time
Is not sorted in memory
0 index keys examined
No index available for this query.

used index	document returned	Index keys examined	Document examined	Average Actual Query Execution Time(ms)
no index used	3743	0	71028	61.7
title_1	3743	1	3743	5.2

Queri 1: spotify2.songs.find({“artists.name”: {$regex: '^C'}})

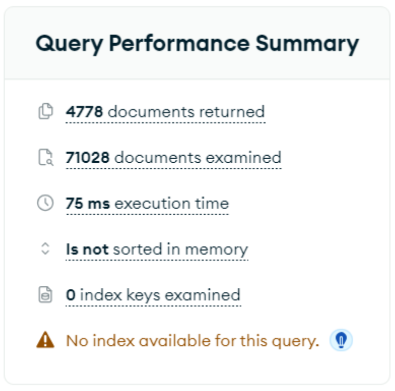
Without index:



> COLLSCAN

Returned 4778 Execution Time 74 ms

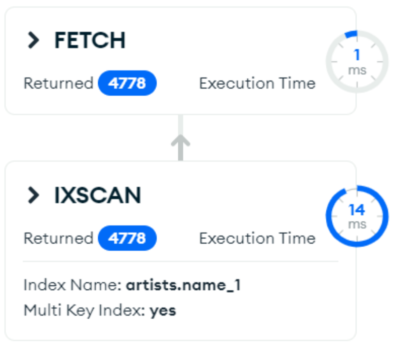
Documents Examined: 71028



Query Performance Summary

- 4778 documents returned
- 71028 documents examined
- 75 ms execution time
- Is not sorted in memory
- 0 index keys examined
- No index available for this query.

With index:



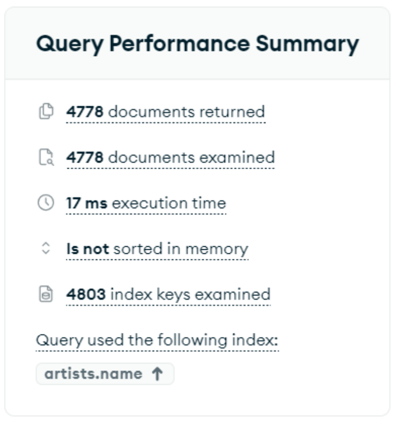
> FETCH

Returned 4778 Execution Time 1 ms

> IXSCAN

Returned 4778 Execution Time 14 ms

Index Name: artists.name_1
Multi Key Index: yes



Query Performance Summary

- 4778 documents returned
- 4778 documents examined
- 17 ms execution time
- Is not sorted in memory
- 4803 index keys examined

Query used the following index:

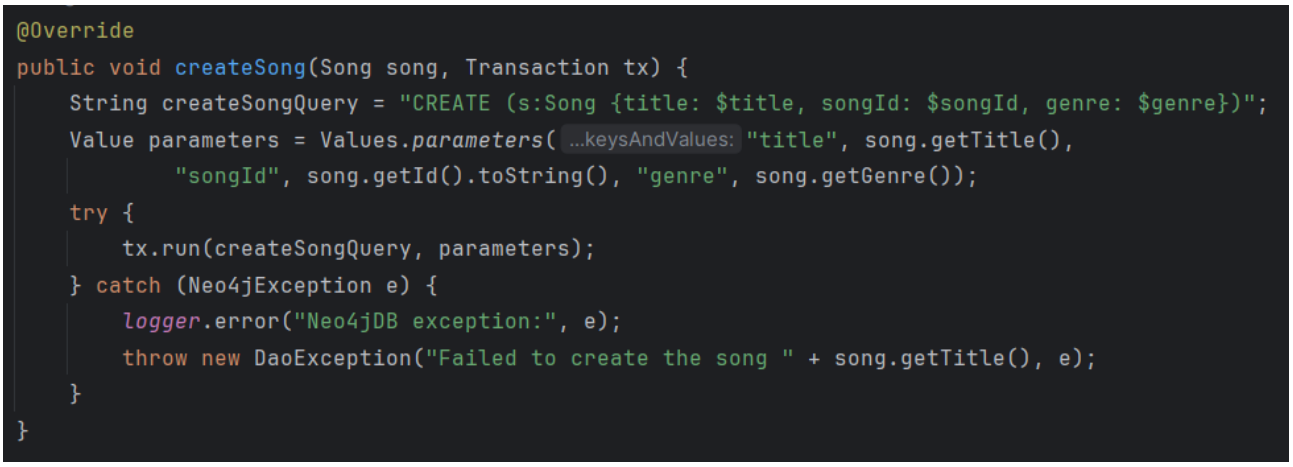
artists.name ↑

used index	document returned	Index keys examined	Document examined	Average Actual Query Execution Time(ms)
no index used	4778	0	71028	66
artists.name_1	4778	1	4778	14.8

Neo4j Implementation

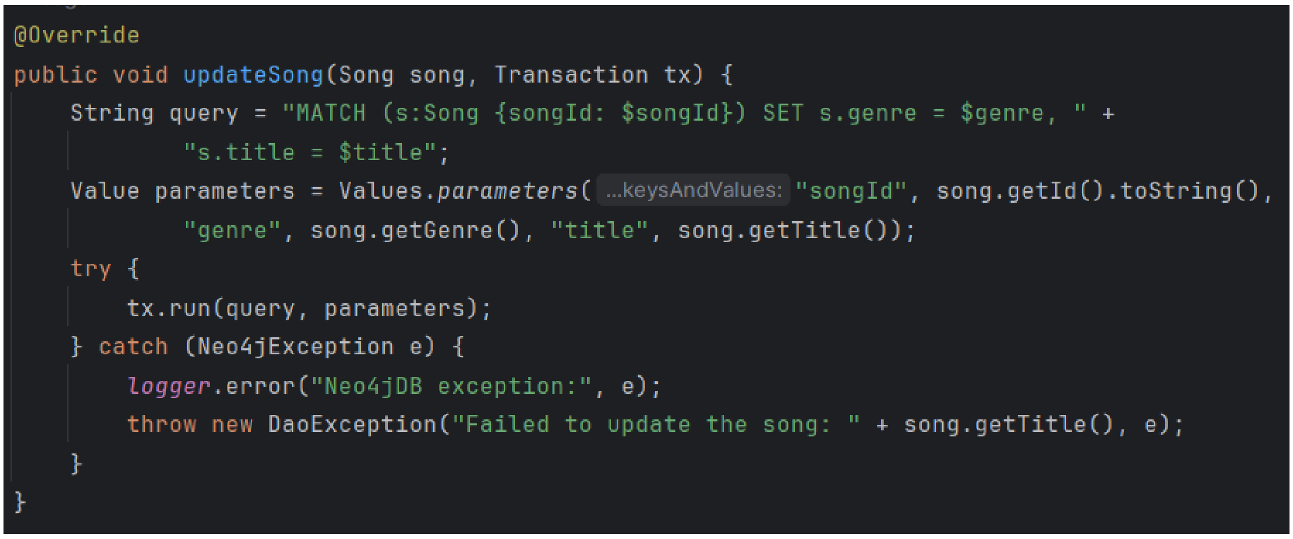
Neo4j CRUD operations

Create



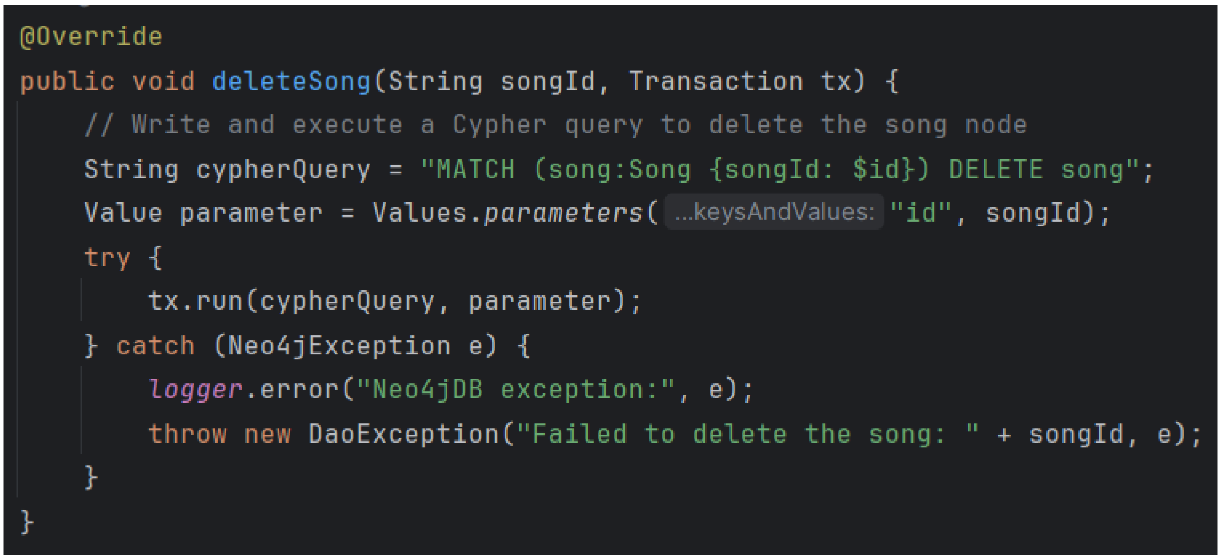
```java
@Override
public void createSong(Song song, Transaction tx) {
    String createSongQuery = "CREATE (s:Song {title: $title, songId: $songId, genre: $genre})";
    Value parameters = Values.parameters( ...keysAndValues: "title", song.getTitle(),
            "songId", song.getId().toString(), "genre", song.getGenre());
    try {
        tx.run(createSongQuery, parameters);
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to create the song " + song.getTitle(), e);
    }
}
```

Update



```java
@Override
public void updateSong(Song song, Transaction tx) {
    String query = "MATCH (s:Song {songId: $songId}) SET s.genre = $genre, " +
            "s.title = $title";
    Value parameters = Values.parameters( ...keysAndValues: "songId", song.getId().toString(),
            "genre", song.getGenre(), "title", song.getTitle());
    try {
        tx.run(query, parameters);
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to update the song: " + song.getTitle(), e);
    }
}
```

Delete



```java
@Override
public void deleteSong(String songId, Transaction tx) {
    // Write and execute a Cypher query to delete the song node
    String cypherQuery = "MATCH (song:Song {songId: $id}) DELETE song";
    Value parameter = Values.parameters( ...keysAndValues: "id", songId);
    try {
        tx.run(cypherQuery, parameter);
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to delete the song: " + songId, e);
    }
}
```

Neo4j on-graph queries

Query: Suggest friends of friends with similar music tastes (in terms of genre)

The goal of this query is to suggest friends of friends for a given listener based on similar music tastes, specifically in terms of music genre.

The query calculates the similarity ratio between the listener and each friend of a friend based on the number of common genres and the total number of distinct genres.

MATCH (listener: Listener {username: $username})-[:CREATED]->(playlist:Playlist)-[:CONTAINS]->(song:Song) WITH listener,

COLLECT(DISTINCT song.genre) AS listenerGenres MATCH (listener)-[:FOLLOWS]->(:Listener)-[:FOLLOWS]->(fofFriend:Listener)-[:CREATED]->(playlist:Playlist)-[:CONTAINS]->(song:Song)

WHERE NOT (listener)-[:FOLLOWS]->(fofFriend) WITH listenerGenres, fofFriend, COLLECT(DISTINCT song.genre) AS fofFriendGenres WITH fofFriend, listenerGenres, fofFriendGenres

WITH fofFriend,

SIZE([genre IN listenerGenres WHERE genre IN fofFriendGenres]) AS commonGenres,

SIZE(listenerGenres + fofFriendGenres) AS totalGenres

WITH fofFriend,

commonGenres * 1.0 / totalGenres AS similarity

WHERE similarity >= 0.5 RETURN fofFriend.username AS newFriendUsername;

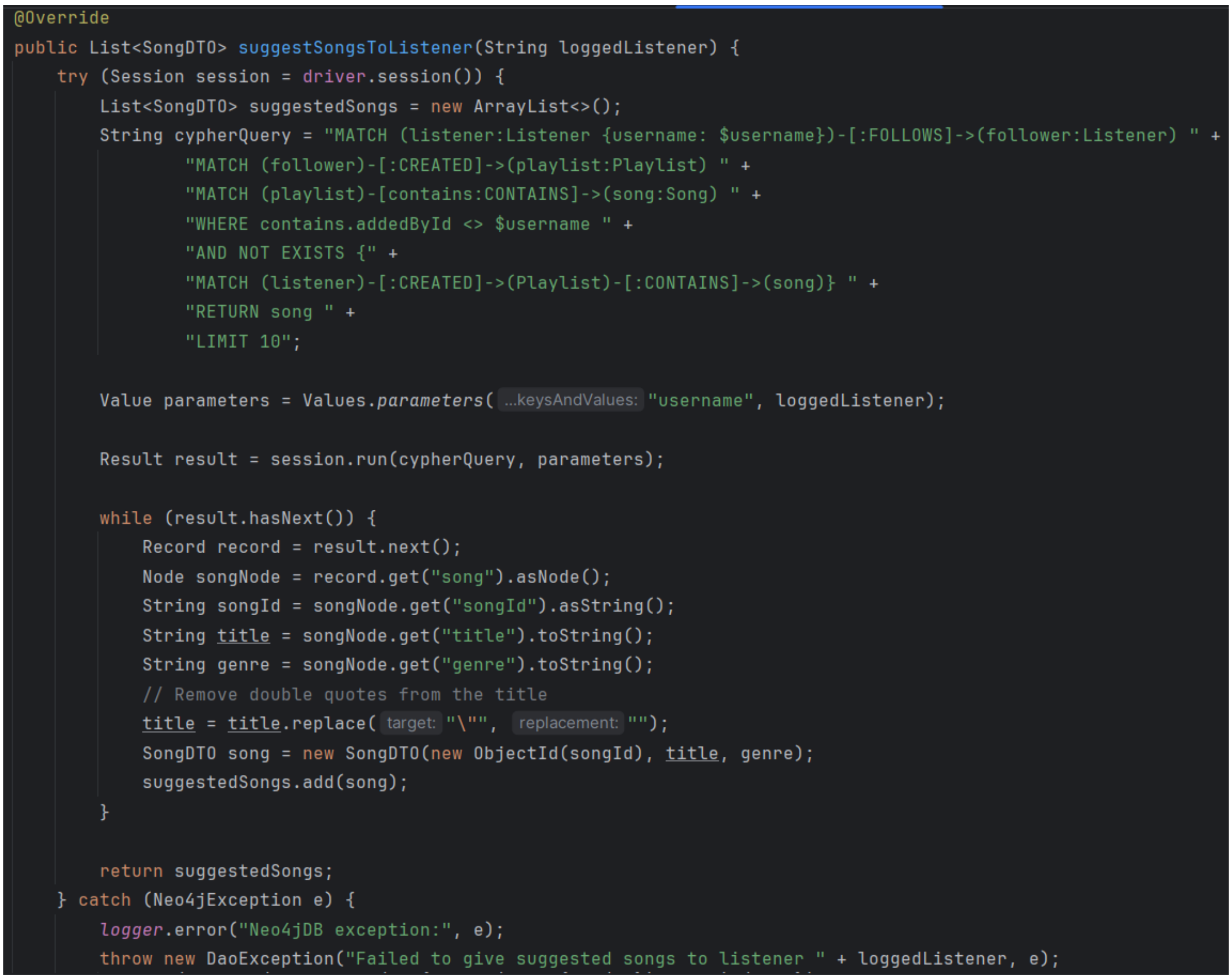
In summary, this query leverages the graph structure to find friends of friends who share similar music tastes (in terms of genre) with the given listener and suggests them as potential new friends.

Query: Suggest songs cointained in friend's playlists which are not your playlists

This Cypher query is designed to find songs associated with playlists created by listeners who are followed by a specified listener.

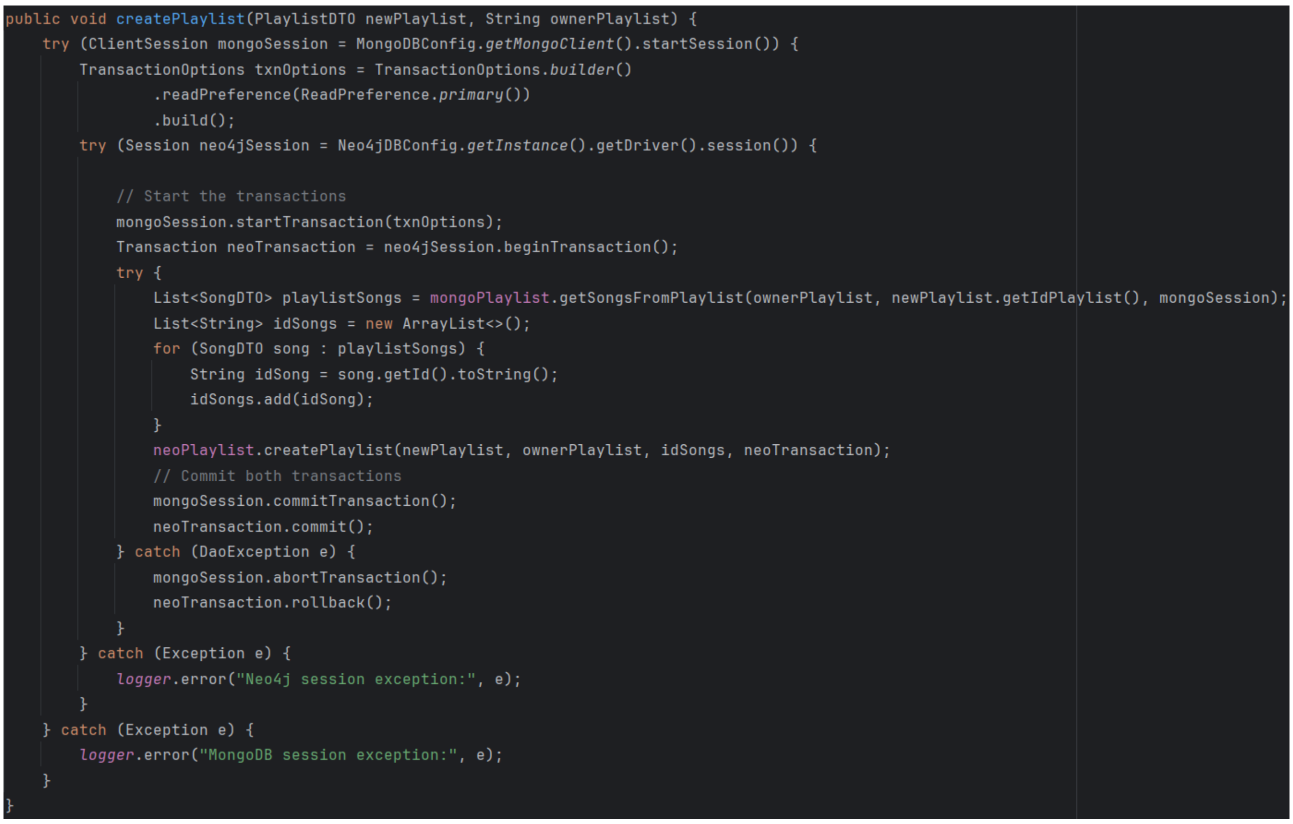
```
MATCH (listener:Listener {username: $username})-[:FOLLOWS]->(follower:Listener)
MATCH (follower)-[:CREATED]->(playlist:Playlist)
MATCH (playlist)-[contains:CONTAINS]->(song:Song)
WHERE contains.addedById <> $username
AND NOT EXISTS {
        MATCH (listener)-[:CREATED]->(Playlist)-[:CONTAINS]->(song)

}

RETURN song

LIMIT 10;
```

In summary, this query aims to find and return up to 10 songs that are associated with playlists created by followers of a specified listener.



```java
@Override
public List<SongDTO> suggestSongsToListener(String loggedListener) {
    try (Session session = driver.session()) {
        List<SongDTO> suggestedSongs = new ArrayList<>();
        String cypherQuery = "MATCH (listener:Listener {username: $username})-[:FOLLOWS]->(follower:Listener) " +
                "MATCH (follower)-[:CREATED]->(playlist:Playlist) " +
                "MATCH (playlist)-[contains:CONTAINS]->(song:Song) " +
                "WHERE contains.addedById <> $username " +
                "AND NOT EXISTS {" +
                "MATCH (listener)-[:CREATED]->(Playlist)-[:CONTAINS]->(song)} " +
                "RETURN song " +
                "LIMIT 10";

        Value parameters = Values.parameters( ...keysAndValues: "username", loggedListener);

        Result result = session.run(cypherQuery, parameters);

        while (result.hasNext()) {
            Record record = result.next();
            Node songNode = record.get("song").asNode();
            String songId = songNode.get("songId").asString();
            String title = songNode.get("title").toString();
            String genre = songNode.get("genre").toString();
            // Remove double quotes from the title
            title = title.replace( target: "\"", replacement: "");
            SongDTO song = new SongDTO(new ObjectId(songId), title, genre);
            suggestedSongs.add(song);
        }

        return suggestedSongs;
    } catch (Neo4jException e) {
        logger.error("Neo4jDB exception:", e);
        throw new DaoException("Failed to give suggested songs to listener " + loggedListener, e);
```

Database consistency management



```java
public void createPlaylist(PlaylistDTO newPlaylist, String ownerPlaylist) {
    try (ClientSession mongoSession = MongoDBConfig.getMongoClient().startSession()) {
        TransactionOptions txnOptions = TransactionOptions.builder()
                .readPreference(ReadPreference.primary())
                .build();
        try (Session neo4jSession = Neo4jDBConfig.getInstance().getDriver().session()) {

            // Start the transactions
            mongoSession.startTransaction(txnOptions);
            Transaction neoTransaction = neo4jSession.beginTransaction();
            try {
                List<SongDTO> playlistSongs = mongoPlaylist.getSongsFromPlaylist(ownerPlaylist, newPlaylist.getIdPlaylist(), mongoSession);
                List<String> idSongs = new ArrayList<>();
                for (SongDTO song : playlistSongs) {
                    String idSong = song.getId().toString();
                    idSongs.add(idSong);
                }
                neoPlaylist.createPlaylist(newPlaylist, ownerPlaylist, idSongs, neoTransaction);
                // Commit both transactions
                mongoSession.commitTransaction();
                neoTransaction.commit();
            } catch (DaoException e) {
                mongoSession.abortTransaction();
                neoTransaction.rollback();
            }
        } catch (Exception e) {
            logger.error("Neo4j session exception:", e);
        }
    } catch (Exception e) {
        logger.error("MongoDB session exception:", e);
    }
}
```

The transactions are managed at the service level.

The code demonstrates a scenario where updates are performed across two different databases, MongoDB and Neo4j. Each database has its own transactional context: MongoDB is managed through ClientSession, and Neo4j through Session.

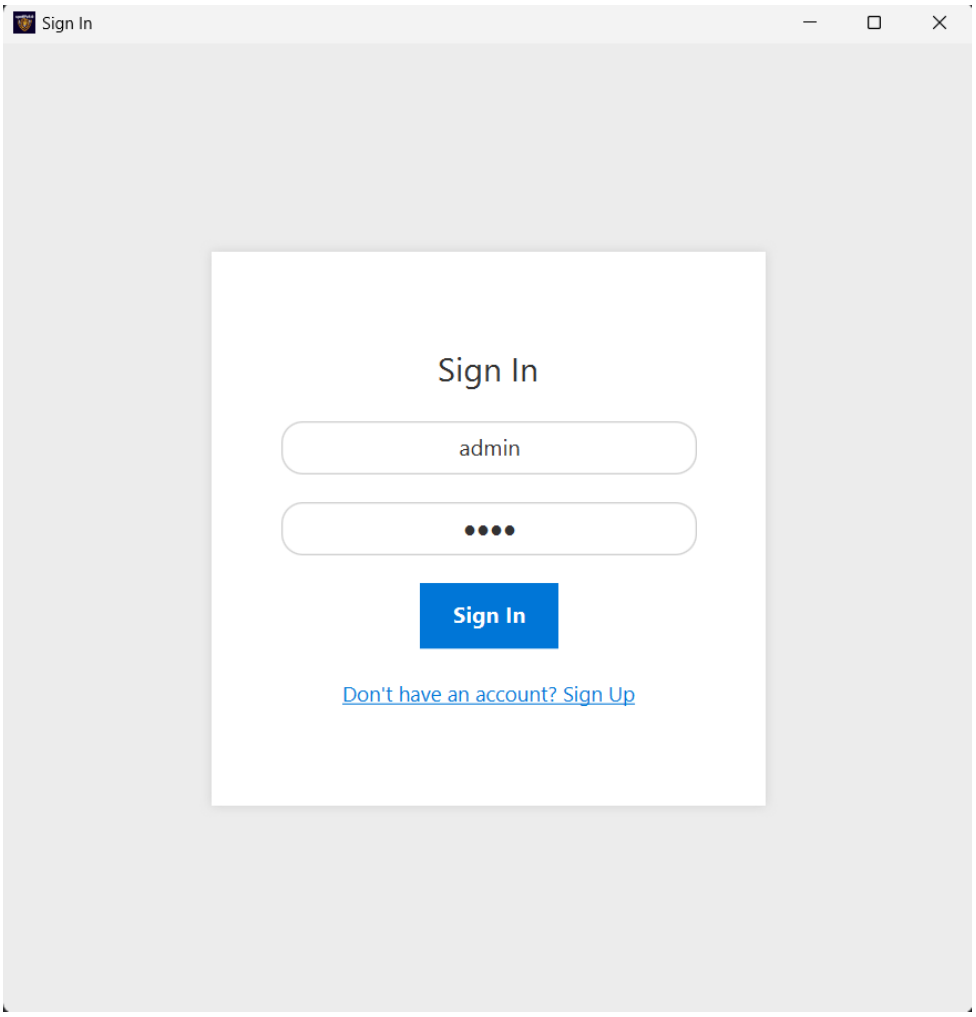
User Manual

Administration Overview

Log In

To access the administration features, use the following credentials:

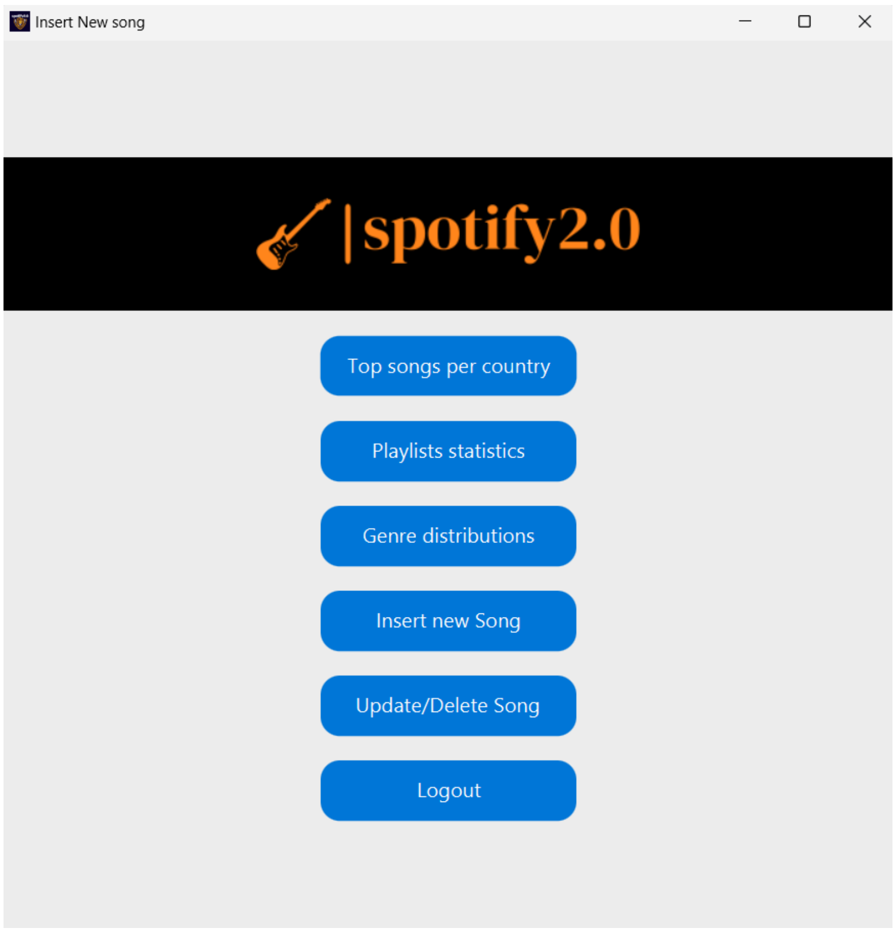
- Username: admin
- Password: 0000



Administrator menu

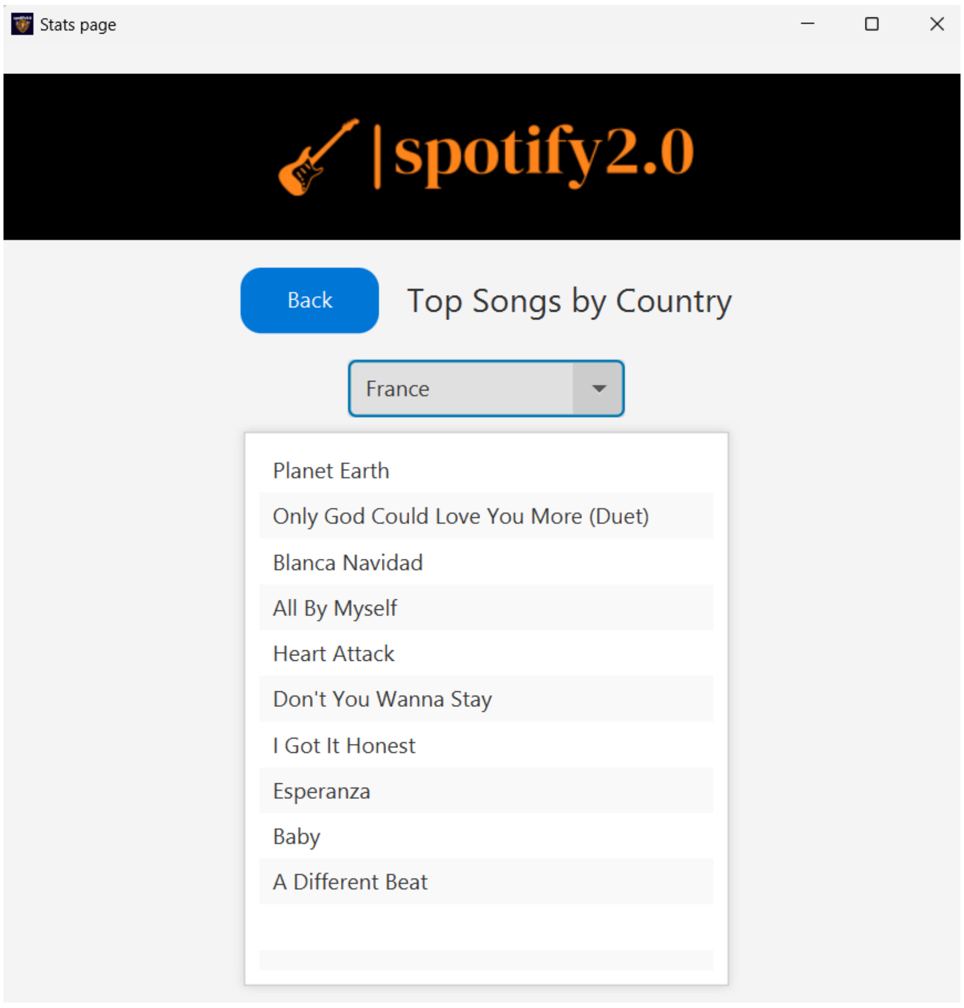
In the main menu, the administrator can perform the following actions:

- Analyze statistics
- Insert, edit, or remove a song



Top songs per country

Selecting a country allows the user to view the top 10 most favorite songs in that country.



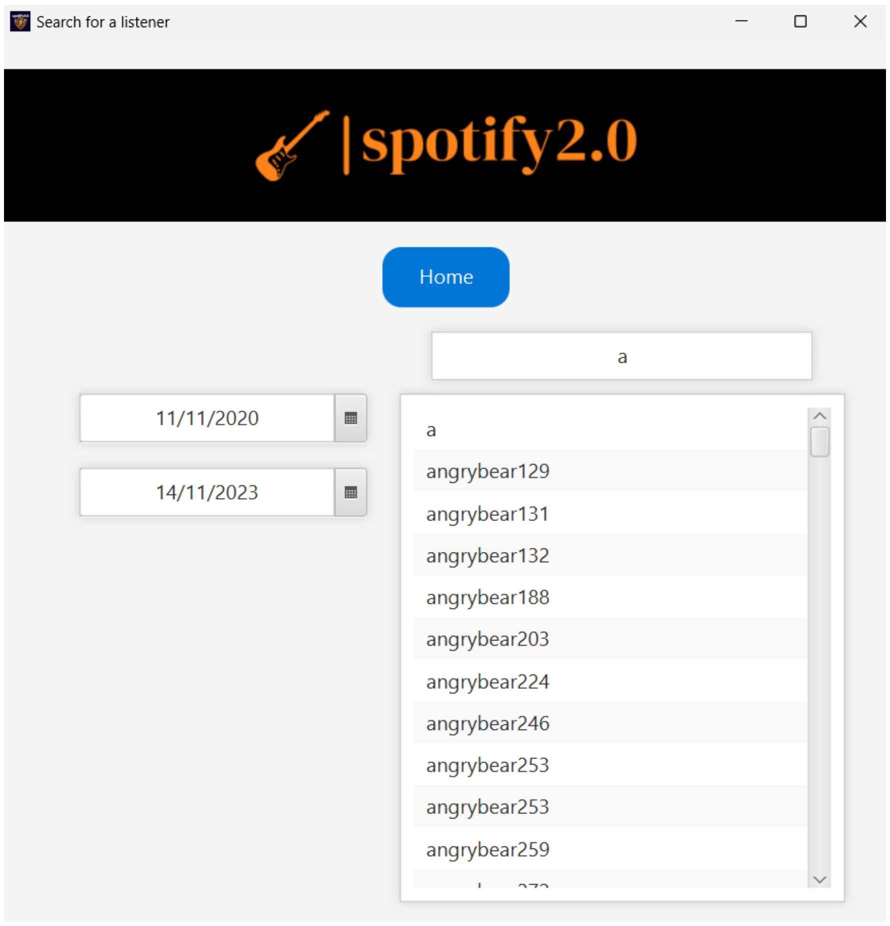
Top Songs by Country

France

Planet Earth
Only God Could Love You More (Duet)
Blanca Navidad
All By Myself
Heart Attack
Don't You Wanna Stay
I Got It Honest
Esperanza
Baby
A Different Beat

Playlists statistics

Specify a start and end date along with a listener's information to view statistics for the playlists of the selected listener during that time interval.

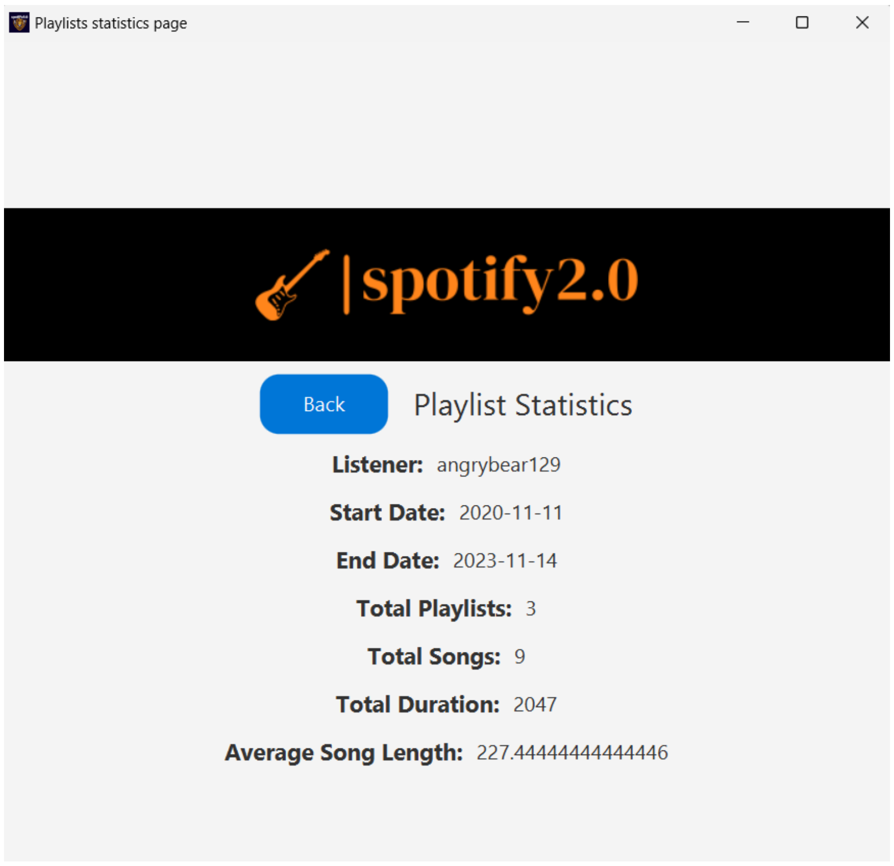


a

11/11/2020

14/11/2023

a
angrybear129
angrybear131
angrybear132
angrybear188
angrybear203
angrybear224
angrybear246
angrybear253
angrybear253
angrybear259



Playlist Statistics

Listener: angrybear129

Start Date: 2020-11-11

End Date: 2023-11-14

Total Playlists: 3

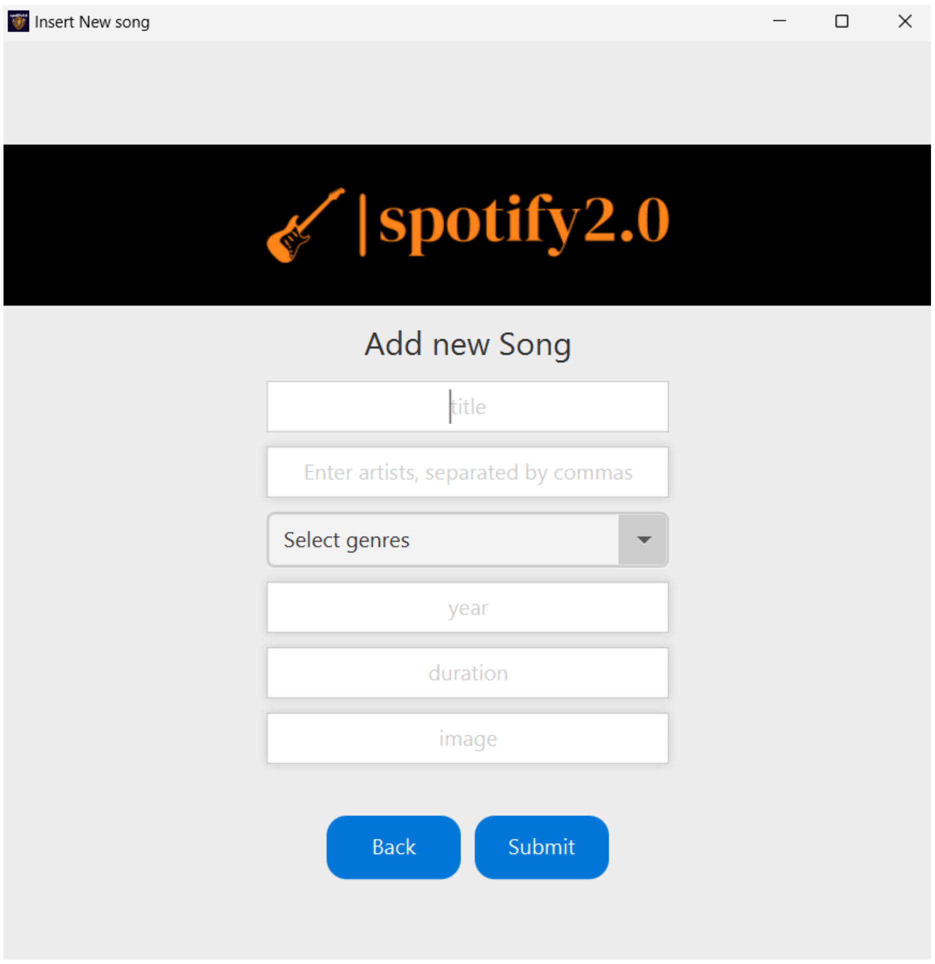
Total Songs: 9

Total Duration: 2047

Average Song Length: 227.44444444444446

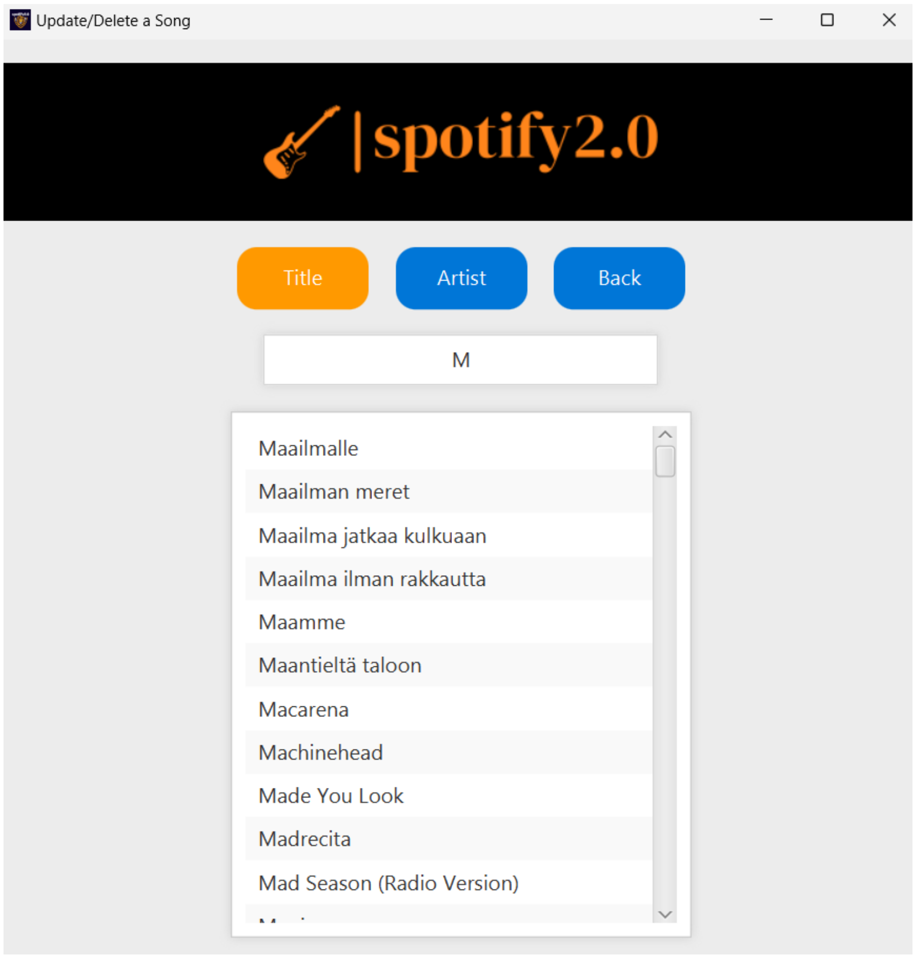
Insert new song

The administrator can insert a new song by filling in the required fields. Any empty field will be flagged, and the system will not proceed with the insertion until all fields are filled.

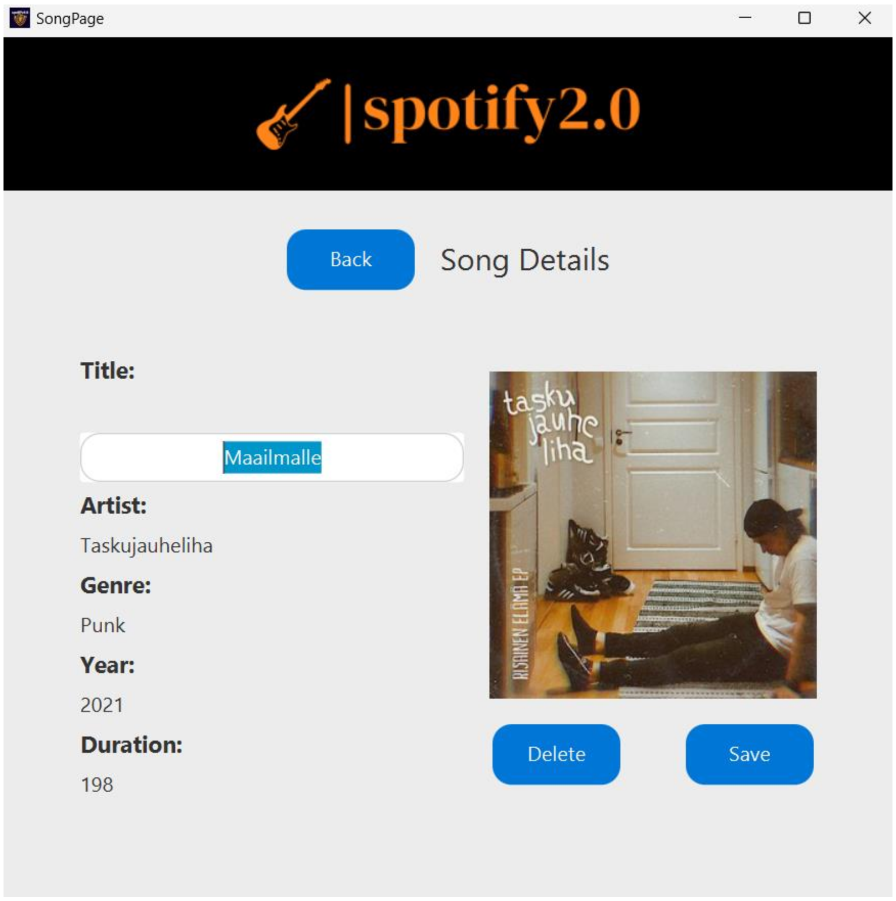


Update/delete a song

After searching for a song to edit, the administrator can modify the fields by selecting the desired field, making the edits, pressing enter, and then clicking "save". Deletion of a song is possible by pressing the delete button.



Maailmalle
Maailman meret
Maailma jatkaa kulkuaan
Maailma ilman rakkautta
Maamme
Maantieltä taloon
Macarena
Machinehead
Made You Look
Madrecita
Mad Season (Radio Version)



Title:

Maailmalle

Artist:

Taskujauheliha

Genre:

Punk

Year:

2021

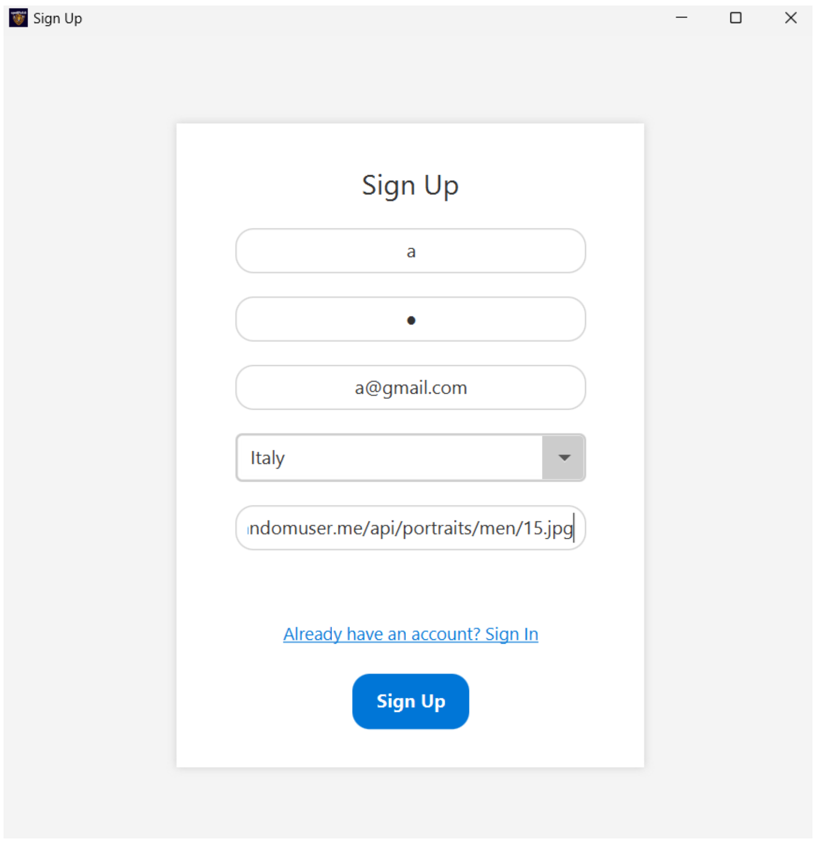
Duration:

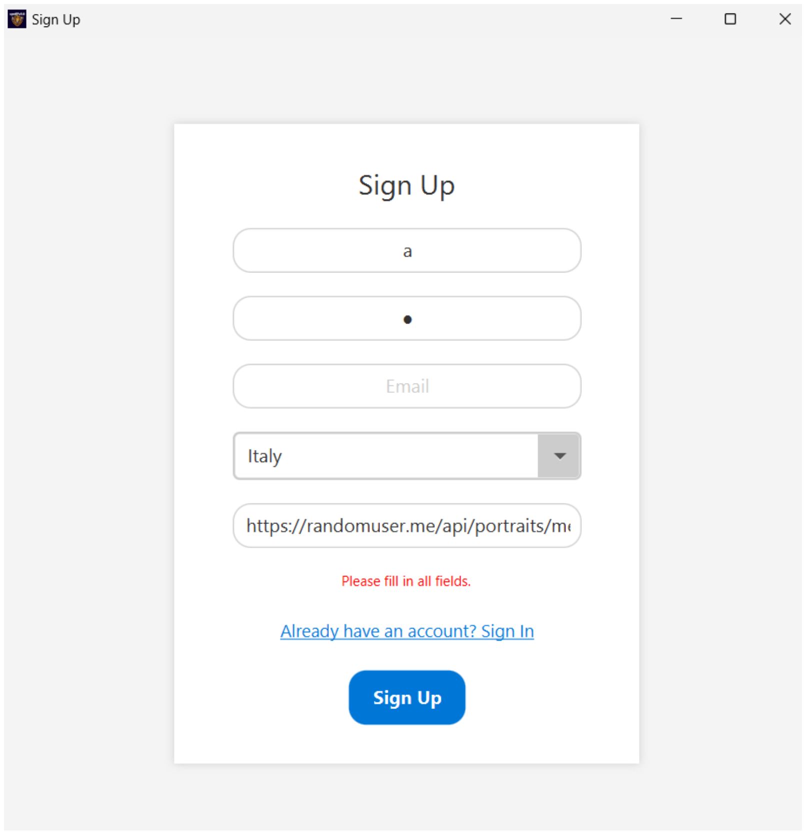
198

Listener Overwiew

Sign Up

Users can register by completing all the required fields. The system will prompt and prevent registration if any field is left empty.

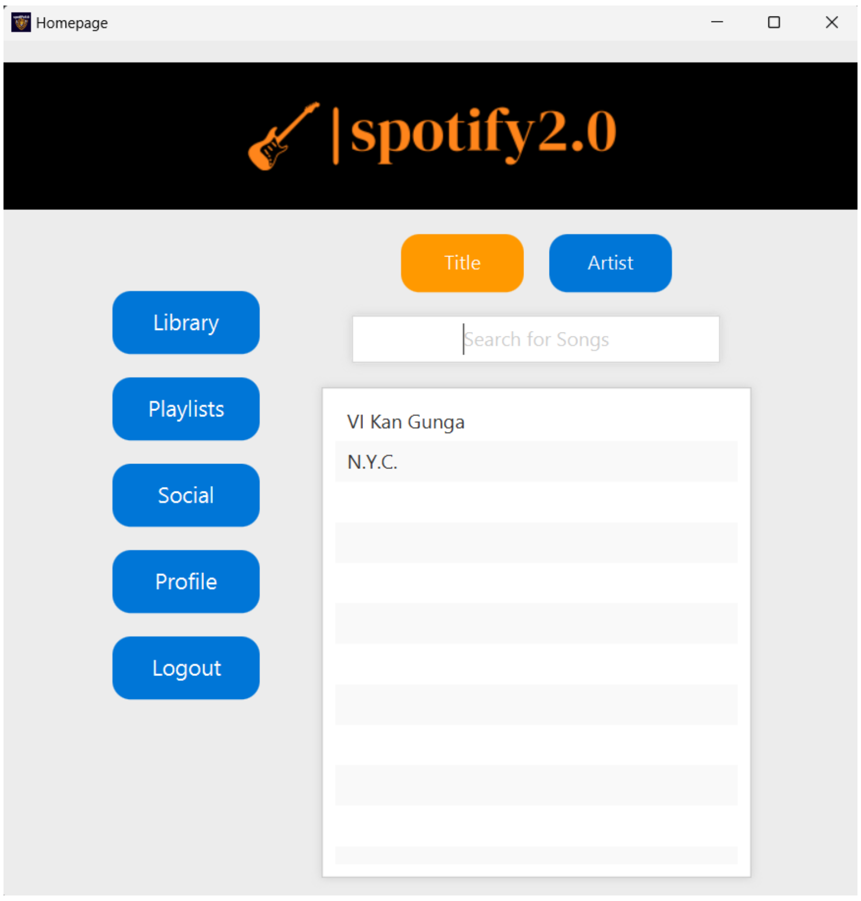

Sign Up
Sign Up
a
•
a@gmail.com
Italy
ndomuser.me/api/portraits/men/15.jpg
Already have an account? Sign In
Sign Up


Sign Up
Sign Up
a
•
Email
Italy
https://randomuser.me/api/portraits/m
Already have an account? Sign In
Sign Up

Please fill in all fields.

Homepage
On the homepage, listeners can:

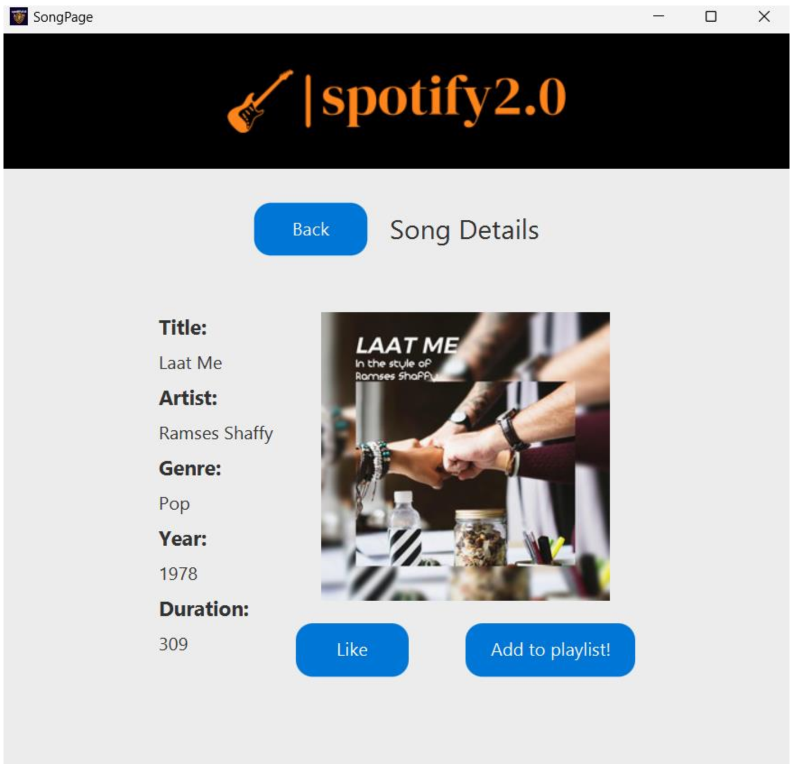
- Search for new songs
- View their library and playlists
- Search for new friends in the social area



VI Kan Gunga
N.Y.C.

Song page

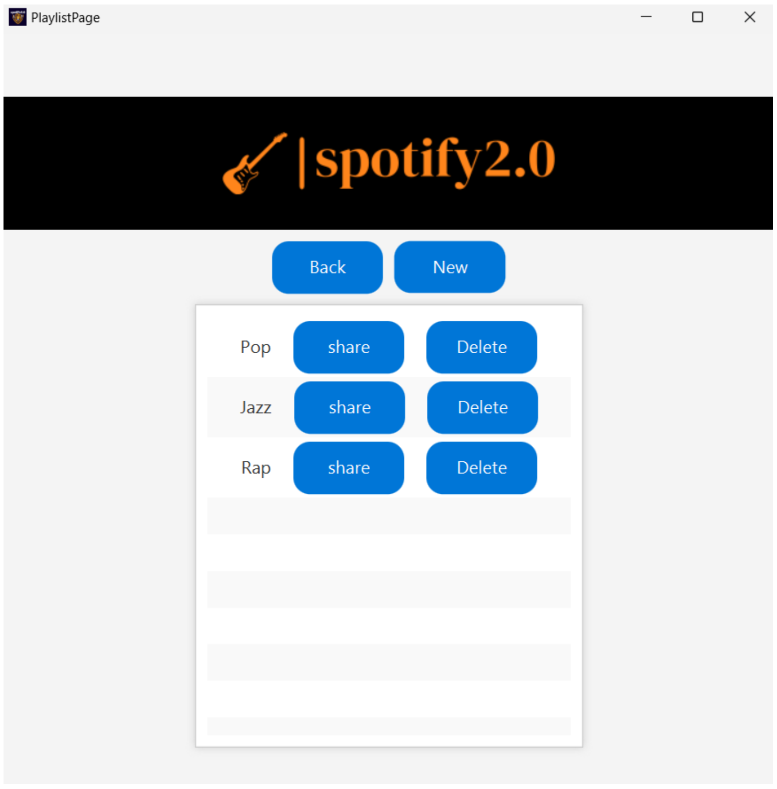
Upon selecting a song, the user can add it to their library by pressing "like" or adding it to a playlist.



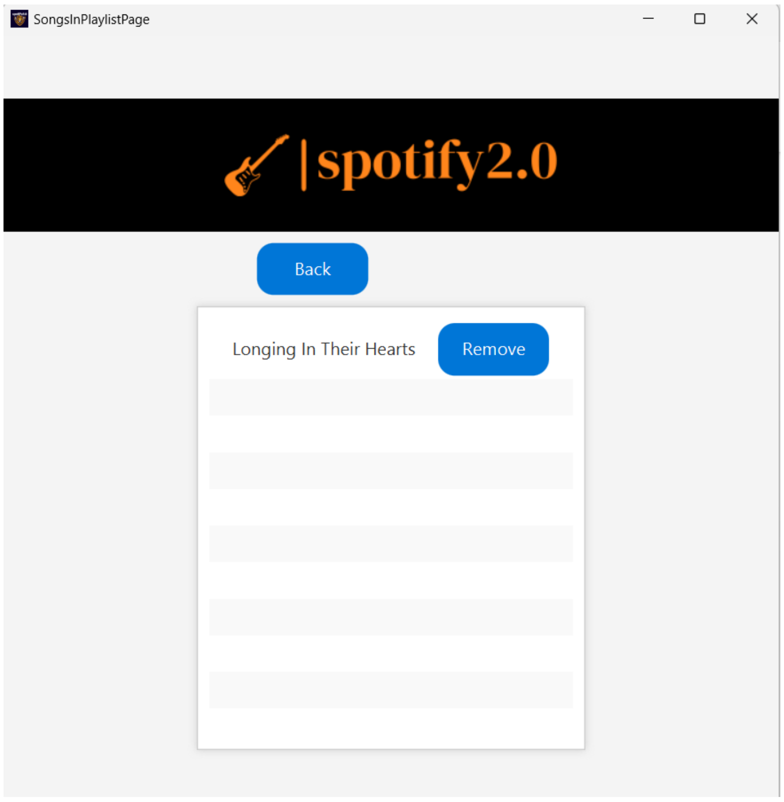
Title:
Laat Me
Artist:
Ramses Shaffy
Genre:
Pop
Year:
1978
Duration:
309

Playlists

Clicking on the playlists button reveals the user's created playlists. Users can choose to share them with followers or delete them. Clicking on a playlist name opens a page displaying the songs in the playlist, where users can remove songs.



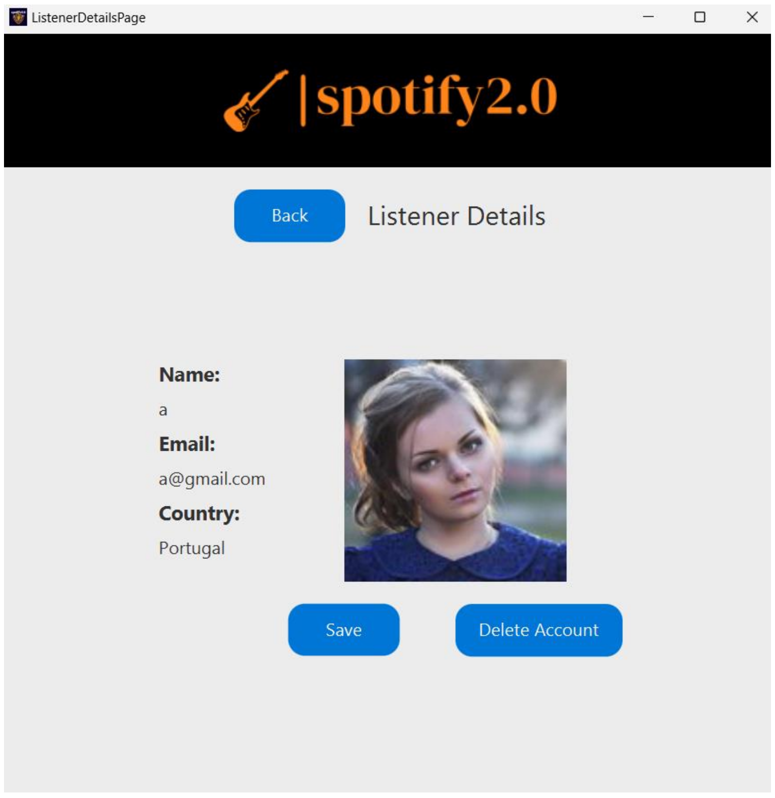
Pop	share	Delete
Jazz	share	Delete
Rap	share	Delete



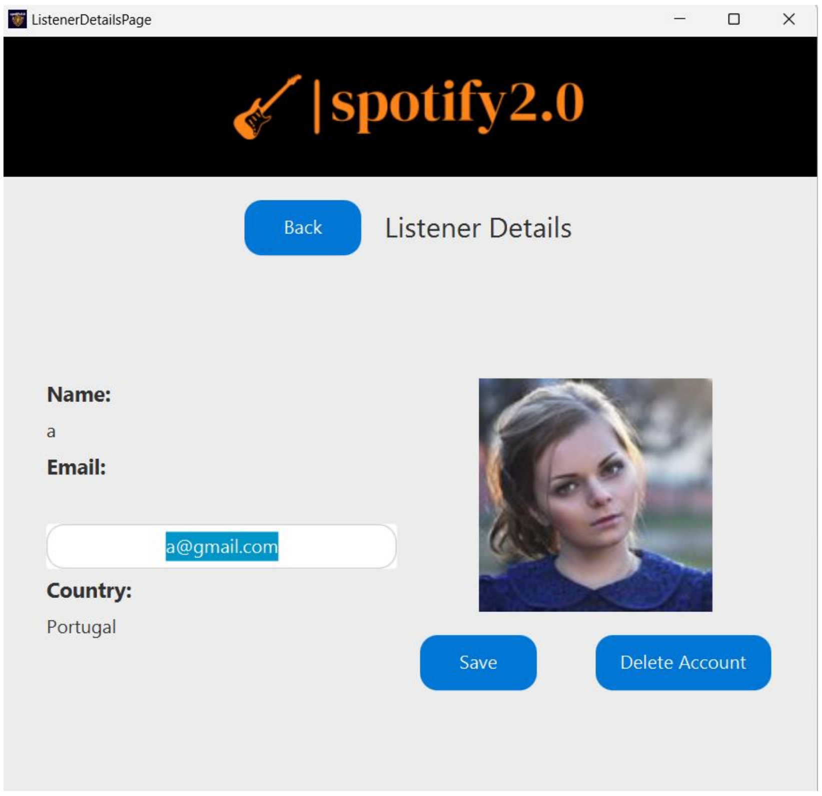
Profile

In the profile section, users can:

- View account information
- Edit account details (similar to how the admin modifies songs)
- Delete the account



Name:
a
Email:
a@gmail.com
Country:
Portugal


ListenerDetailsPage

Back Listener Details

Name:

a

Email:

a@gmail.com

Country:

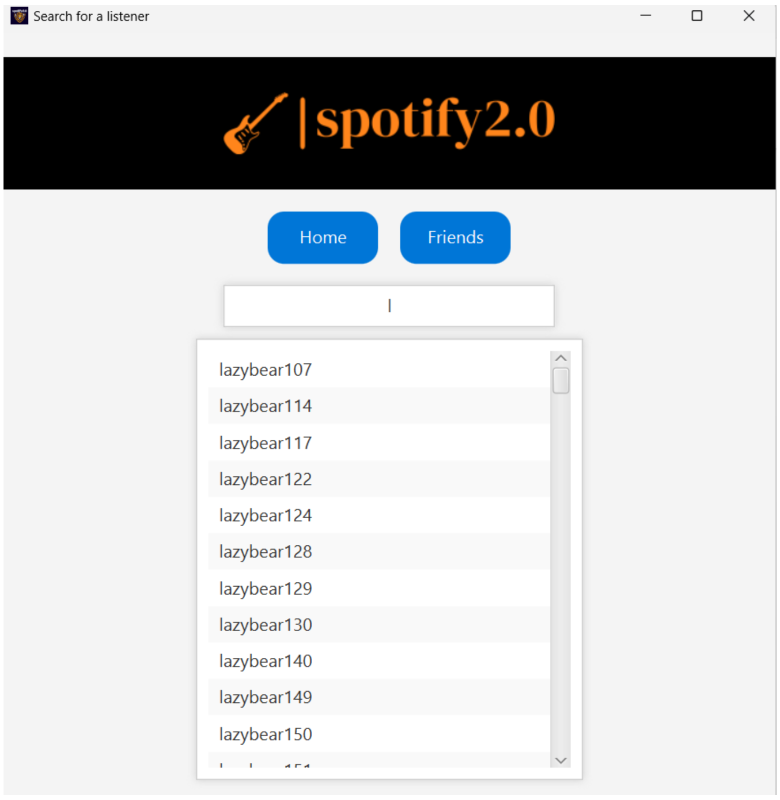
Portugal

Save Delete Account

Social

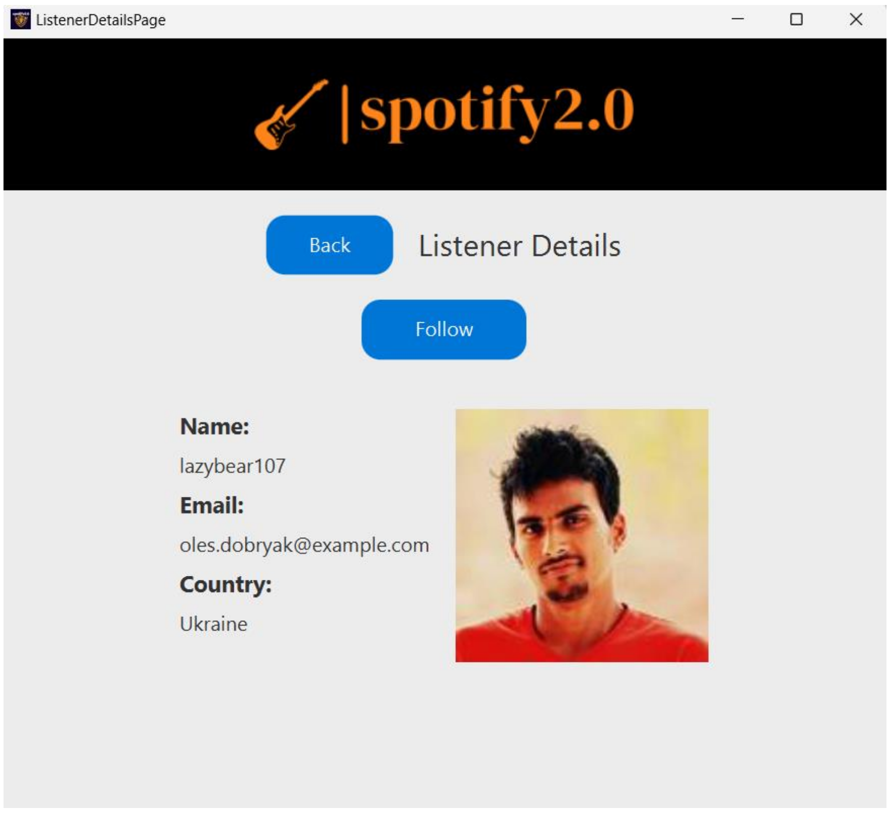
In the social area, users can:

- Search for new friends
- View the list of followers
- Explore shared playlists
- Insert new songs into shared playlists

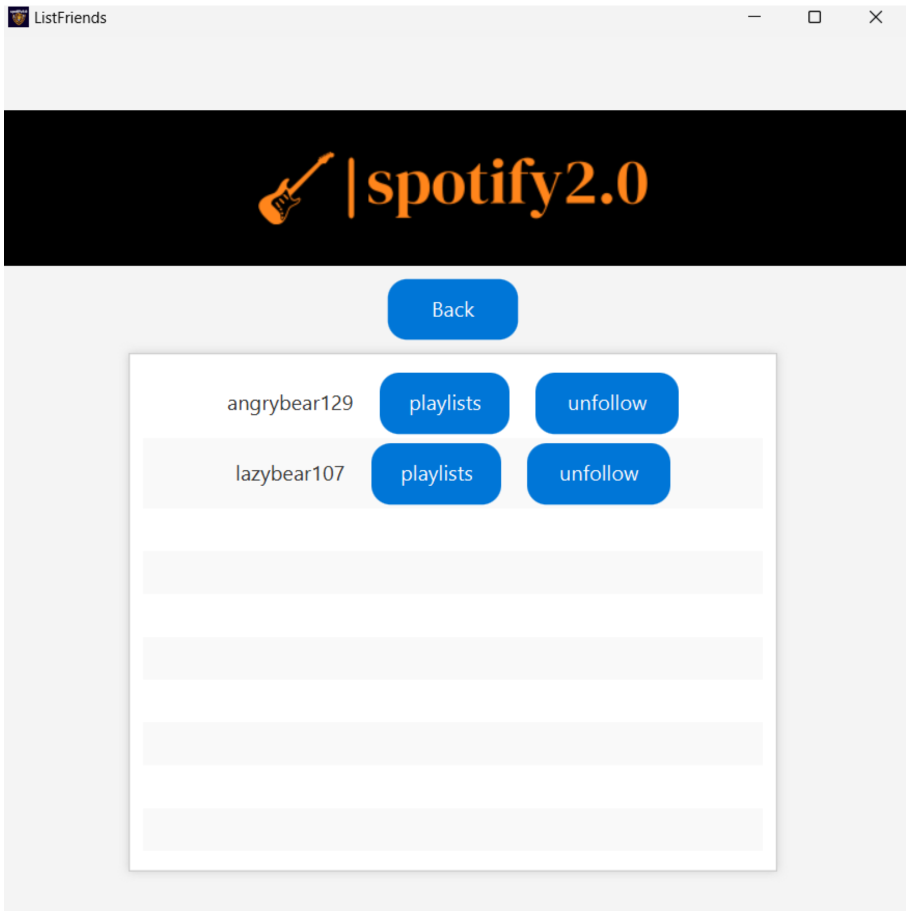

Search for a listener

Home Friends

lazybear107
lazybear114
lazybear117
lazybear122
lazybear124
lazybear128
lazybear129
lazybear130
lazybear140
lazybear149
lazybear150



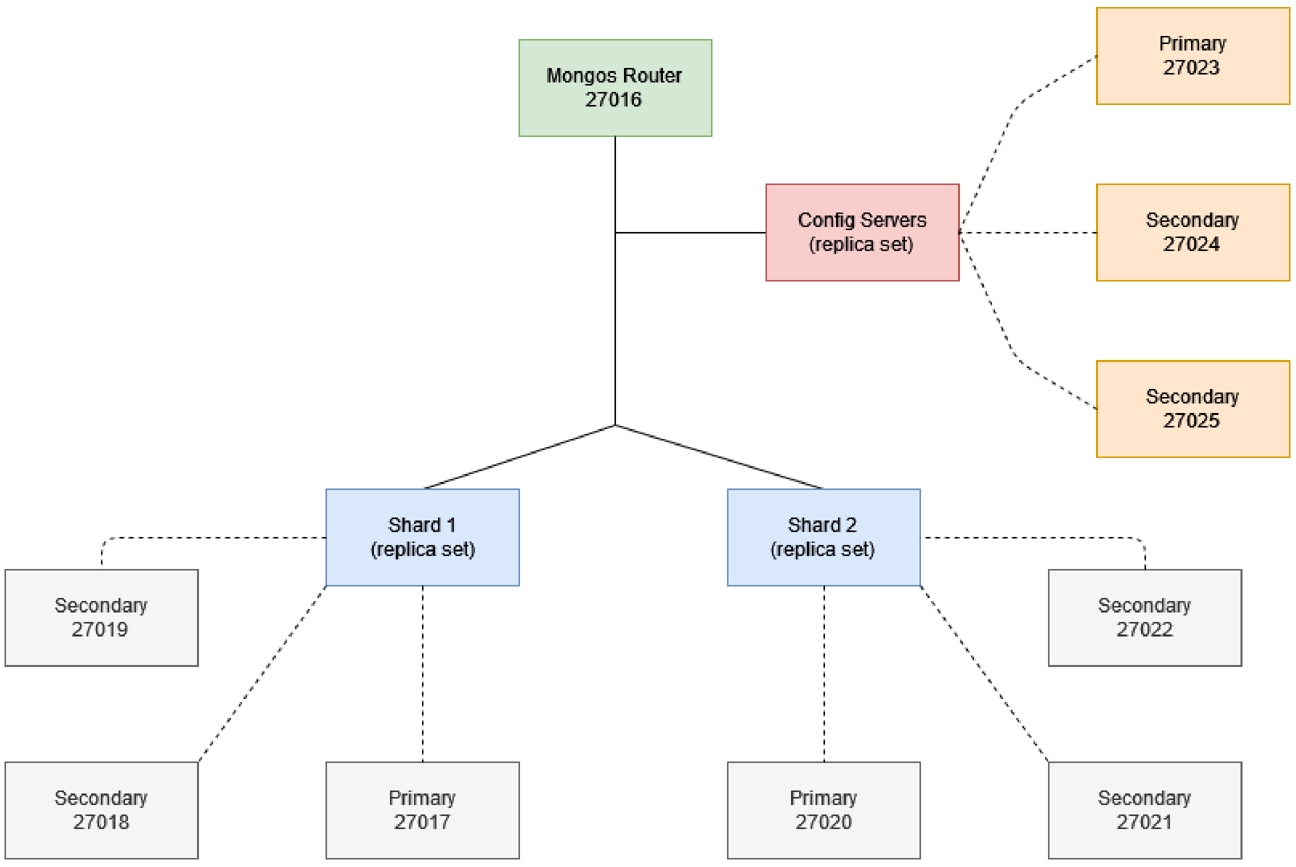
Name:
lazybear107
Email:
oles.dobryak@example.com
Country:
Ukraine



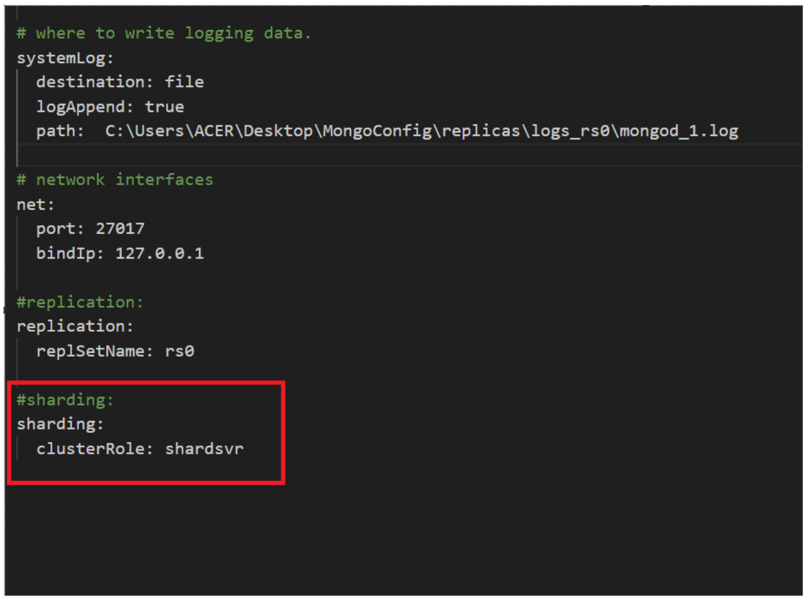
angrybear129	playlists	unfollow
lazybear107	playlists	unfollow

Sharding

The planned architecture is as follows:

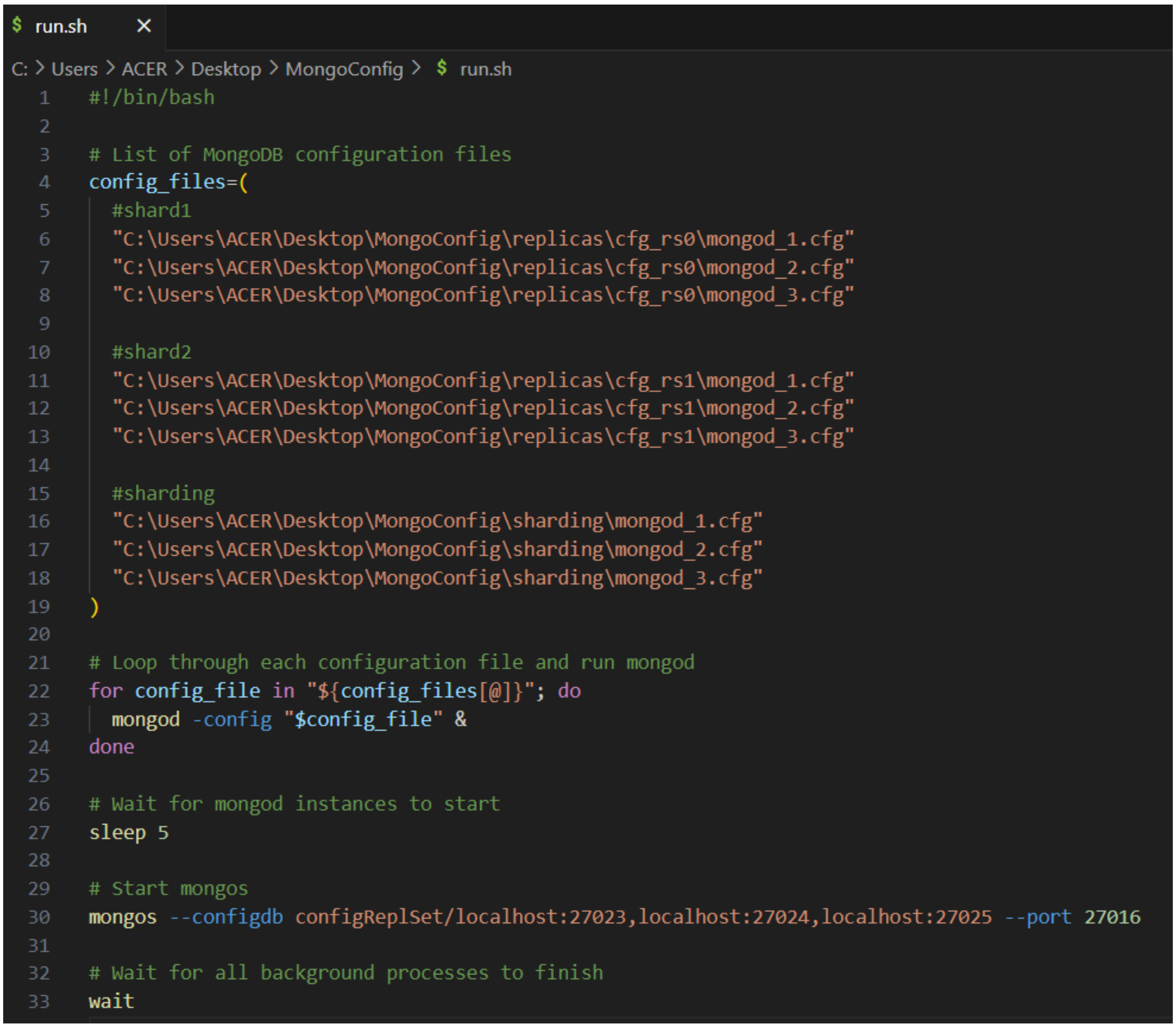


There are three replicas for both the shards and the config server. This organizational structure, even as the system becomes more complex, avoids a single point of failure. The replication configuration parameters were set as explained in the "Replication Configuration" section, with the addition of the "sharding" line indicating the server as part of a shard.



```
# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path:  C:\Users\ACER\Desktop\MongoConfig\replicas\logs_rs0\mongod_1.log

# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1

#replication:
replication:
  replSetName: rs0

#sharding:
sharding:
  clusterRole: shardsvr
```

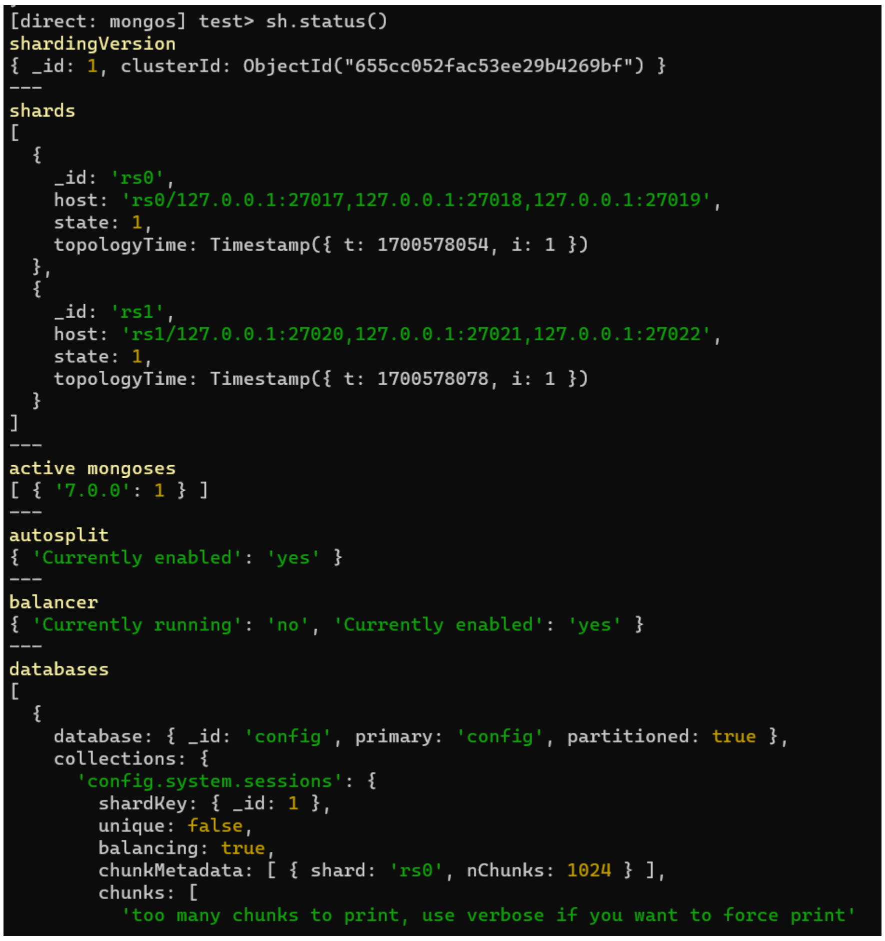
The following Bash script has been used to run all the machines and the mongos router:



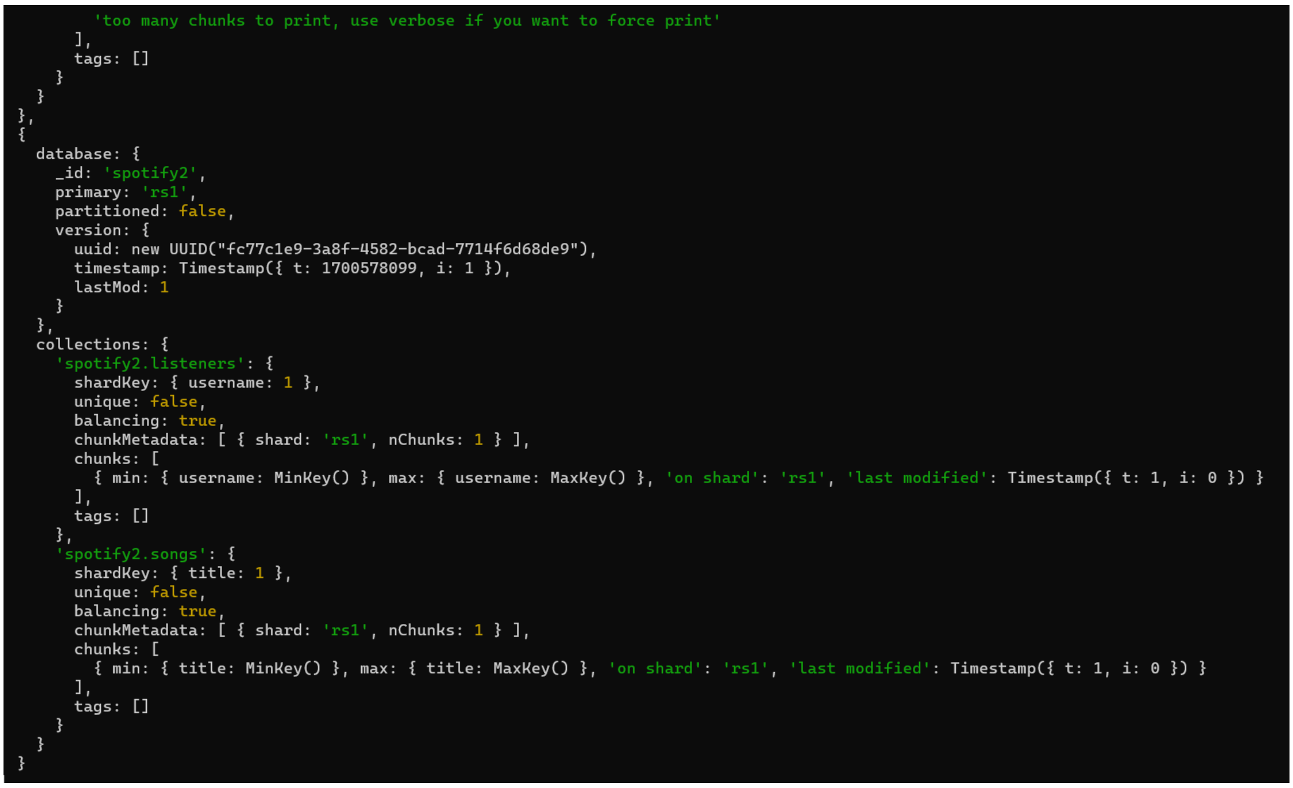
```bash
#!/bin/bash

# List of MongoDB configuration files
config_files=(
  #shard1
  "C:\Users\ACER\Desktop\MongoConfig\replicas\cfg_rs0\mongod_1.cfg"
  "C:\Users\ACER\Desktop\MongoConfig\replicas\cfg_rs0\mongod_2.cfg"
  "C:\Users\ACER\Desktop\MongoConfig\replicas\cfg_rs0\mongod_3.cfg"

  #shard2
  "C:\Users\ACER\Desktop\MongoConfig\replicas\cfg_rs1\mongod_1.cfg"
  "C:\Users\ACER\Desktop\MongoConfig\replicas\cfg_rs1\mongod_2.cfg"
  "C:\Users\ACER\Desktop\MongoConfig\replicas\cfg_rs1\mongod_3.cfg"

  #sharding
  "C:\Users\ACER\Desktop\MongoConfig\sharding\mongod_1.cfg"
  "C:\Users\ACER\Desktop\MongoConfig\sharding\mongod_2.cfg"
  "C:\Users\ACER\Desktop\MongoConfig\sharding\mongod_3.cfg"
)

# Loop through each configuration file and run mongod
for config_file in "${config_files[@]}"; do
  mongod -config "$config_file" &
done

# Wait for mongod instances to start
sleep 5

# Start mongos
mongos --configdb configReplSet/localhost:27023,localhost:27024,localhost:27025 --port 27016

# Wait for all background processes to finish
wait
```

A higher priority was assigned to one of the replicas to ensure its election as the primary, and the same parameters for read and write concern were set. The sharding keys chosen were the ‘**username’** for the listener’s collection and the ‘**title’** for the song collection. I opted for these attributes as sharding keys because they play a crucial role in the majority of queries used in the application. Moreover, the values these fields can assume are diverse, allowing for an equitable distribution across the various shards.

As partition algorithm the **range-based sharding algorithm** is applied (default one).



```
[direct: mongos] test> sh.status()
shardingVersion
{ _id: 1, clusterId: ObjectId("655cc052fac53ee29b4269bf") }
---
shards
[
  {
    _id: 'rs0',
    host: 'rs0/127.0.0.1:27017,127.0.0.1:27018,127.0.0.1:27019',
    state: 1,
    topologyTime: Timestamp({ t: 1700578054, i: 1 })
  },
  {
    _id: 'rs1',
    host: 'rs1/127.0.0.1:27020,127.0.0.1:27021,127.0.0.1:27022',
    state: 1,
    topologyTime: Timestamp({ t: 1700578078, i: 1 })
  }
]
---
active mongoses
[ { '7.0.0': 1 } ]
---
autosplit
{ 'Currently enabled': 'yes' }
---
balancer
{ 'Currently running': 'no', 'Currently enabled': 'yes' }
---
databases
[
  {
    database: { _id: 'config', primary: 'config', partitioned: true },
    collections: {
      'config.system.sessions': {
        shardKey: { _id: 1 },
        unique: false,
        balancing: true,
        chunkMetadata: [ { shard: 'rs0', nChunks: 1024 } ],
        chunks: [
          'too many chunks to print, use verbose if you want to force print'
```

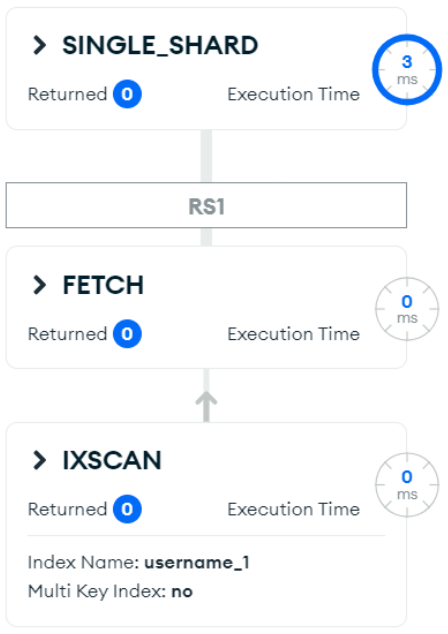


```
          'too many chunks to print, use verbose if you want to force print'
        ],
        tags: []
      }
    }
  },
  {
    database: {
      _id: 'spotify2',
      primary: 'rs1',
      partitioned: false,
      version: {
        uuid: new UUID("fc77c1e9-3a8f-4582-bcad-7714f6d68de9"),
        timestamp: Timestamp({ t: 1700578099, i: 1 }),
        lastMod: 1
      }
    },
    collections: {
      'spotify2.listeners': {
        shardKey: { username: 1 },
        unique: false,
        balancing: true,
        chunkMetadata: [ { shard: 'rs1', nChunks: 1 } ],
        chunks: [
          { min: { username: MinKey() }, max: { username: MaxKey() }, 'on shard': 'rs1', 'last modified': Timestamp({ t: 1, i: 0 }) }
        ],
        tags: []
      },
      'spotify2.songs': {
        shardKey: { title: 1 },
        unique: false,
        balancing: true,
        chunkMetadata: [ { shard: 'rs1', nChunks: 1 } ],
        chunks: [
          { min: { title: MinKey() }, max: { title: MaxKey() }, 'on shard': 'rs1', 'last modified': Timestamp({ t: 1, i: 0 }) }
        ],
        tags: []
      }
    }
  }
```

It is noteworthy that the collections currently consist of only one Chunk. This is because, by default, the dimensions of a Chunk are set to 64MB, which is larger than the collections. However, as the dataset size increases, the data will be dynamically divided between the two shards based on the distribution of the sharding keys.

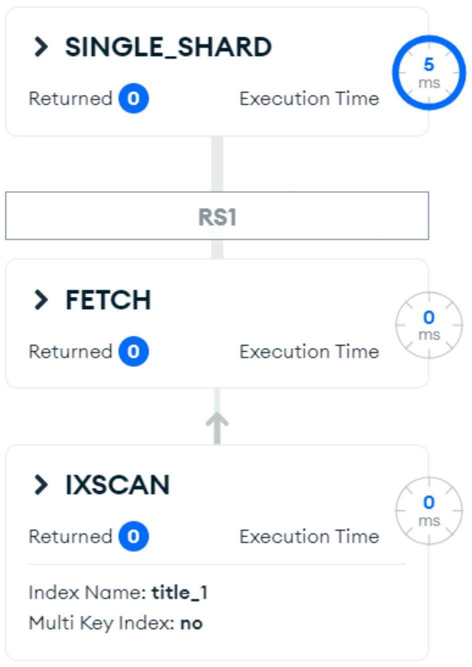
Queries have been analyzed in the context of sharding, as previously discussed in the section on indices.

Query 1: spotify2.listeners.find({username: {$regex: '^g'}})



Index Name: **username_1**
Multi Key Index: **no**

Queri 2: spotify2.songs.find({title: {$regex: '^c'}})



Index Name: **title_1**
Multi Key Index: **no**