

SRH HOCHSCHULE HEIDELBERG

Advanced Databases

eMUSIC DB

By,

Adarsh Mayya Devashya - 11012460
Swaraj Rath - 11012467
Rakshit Siddalingappa - 11012530
Ankush Akshay Mallegowda - 11012529
Jeevan Sira Manjunath - 11012455

Supervisors,

Frank Hefter
Prof.Dr Barbara Sprick

February 07, 2020

Table of Contents

1. Introduction	3
1.1 RCI Matrix	4
2. Project Use Case	5
2.1 User Story	6
2.2 User Story	6
2.3 User Story	7
2.4 User Story	7
2.5 User Story	8
2.6 User Story	8
2.7 User Story	8
3. Fundamentals	9
3.1.1 MongoDB	9
3.1.2 Neo4j	9
3.1.3 Redis	10
3.2 Data Model	10
3.3 CRUD Operations	11
3.3.1 MongoDB CRUD Operations	11
3.3.2 Neo4j CRUD Operations	12
3.3.3 Redis CRUD Operations	13
4. Project Data Model	14
4.1 MongoDB Data Model	14
4.2 Neo4j Data Model	15
4.3 Redis Data Model	16
5. Implementation	17
5.1.1 Implementation of MongoDB Data Model	17
5.1.2 Implementation of Neo4j Data Model	20
5.1.3 Implementation of Redis Data Model	22
5.2 Implementation of User Stories & Queries	23
6. Conclusion	42

List of Figures

1. RCI Matrix	4
2. User Case	5
3. Data Model Overview	11

1. Introduction

Music plays a vital role in everyone's life, and everybody has their own experience while listening to music. The modes of listening music has changed drastically over the years and nowadays everyone prefers hearing music online, since it is easier and almost everyone has access to the internet. This creates more demand in online music platforms which must have huge song libraries, relevant information related to songs and a great user experience where users search and hear the songs they like and get to explore new songs.

The objective of this project is to design and develop a data model to a music application across multiple types of databases. The application should consist of all the songs, artists and albums of the corresponding songs. The purpose of the applications is not only to store details about the songs, but also to suggest a good song to a user, and users can make their own playlist of their favourite songs and follow and like their favourite artists and songs. The main challenge in this project is to curate the information in different databases and integrate them to form a final product while maintaining consistency. All the data in different databases should flow without flaw. In this document we will present the databases selected to achieve the goal of the project, and the data models that support it, along with the reasoning behind each choice and its implementation.

RCI Matrix

	Adarsh Mayya Devashya	Swaraj Rath	Rakshit Siddalingappa	Ankush	Jeevan
User and Sub User stories formation	R	R	R	I	I
Selecting the databases	R	R	R	I	I
Data Modelling	R	R	C	R	R
MongoDB	C	R	C	I	I
Redis	C	R	I	I	R
Neo4j	R	I	C	I	I
Report writing	R	R	R	C	C

R - Responsible ; C - Consulted ; I - Informed

2. Project Use Case

There are several Music applications available in the market. Amongst all the media platforms, Music is one of the most liked applications from the user's perspective. It is believed that every human has listened to music at least once in their life time & everyone will have a built-in or external music application in their electronic devices. As a team we had a discussion regarding use cases as in customers view what can they expect from this music application. As a team, We have decided to go on with 7 user stories in our project. Each user story will have sub user stories which will be explained in detail with the query in the report further. The user stories helped us to build necessary & extra functionalities. The figure below provides a general idea of the use case.

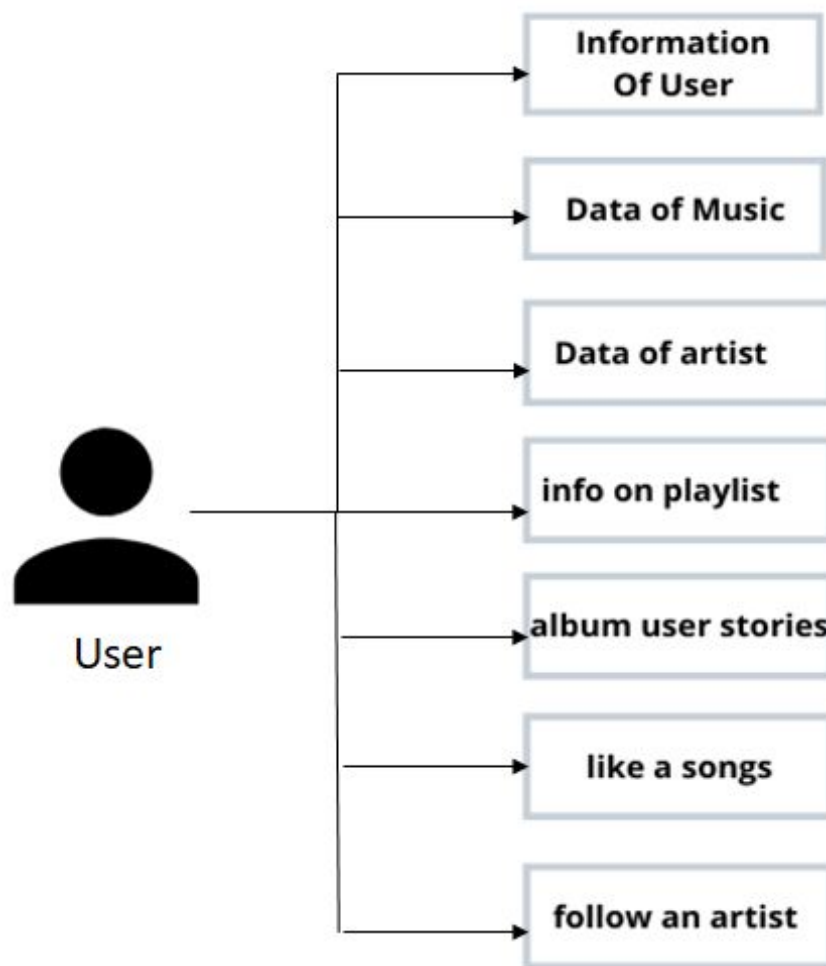


Figure 2: User Case

2.1 User Story – 1

The first user-story is related to basic information of a User, Where we have discussed how the data of a user can be modified. We have tried to break down into sub user-stories.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to register an account.
- As an end user, I would like to update my basic information in future.
- As an end user, I would like to know the subscription packages.(Future Scope)
- As an end user, I would like to receive notifications from the applications.(Future Scope)
- As an end user, I would like to delete my account.(Future Scope)

2.2 User Story – 2

The second user-story is all about data of the music. Where a user wants to know about that particular music.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to know in which year a particular music was released.
- As an end user, I would like to know in which year a particular music was added to this application.
- As an end user, I would like to listen to most liked music.(Future Scope)
- As an end user, I would like to listen to all the music which was released before the year “2000”.
- As an end user, I would like to know the length of a song.(Future Scope)

2.3 User Story – 3

The third user-story is focused on the data of an Artist. Where a user wants to know more about an artist.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to listen to music according to the country wise of an Artist.
- As an end user, I would like to listen to music by an Artist according to the specific year “2010”.
- As an end user, I would like to listen to music of a specific Genre of a particular Artist.(Future Scope)
- As an end user, I would like to listen to the music of a Female vocal artist only.(Future Scope)
- As an end user, I would like to know the most liked song of an Artist.(Future Scope)

2.4 User Story – 4

The fourth user-story shows us the playlist which is being created by a user according to his choices as well as all the information in a playlist.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to create a playlist of my choice.
- As an end user, I would like to follow a playlist which is created by other users.
- As an end user, I would like to know the music inside a playlist.(Future Scope)
- As an end user, I would like to know the number of music inside a playlist.(Future Scope)
- As an end user, I would like to know the total length of a playlist.(Future Scope)

2.5 User Story – 5

The fifth user-story displays the Albums user stories where a user can find his favourite albums in the application.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to search for an album.
- As an end user, I would like to display all the albums of an artist.
- As an end user, I would like to know about the music present in a specific album from an artist.(Future Scope)
- As an end user, I would like to know if any collaboration music is present in an album for an artist.(Future Scope)

2.6 User Story – 6

The sixth user-story shows us the interest of a user where he has liked music in the application.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to browse all the music where any of the users has given a Like to a music.
- As an end user, I would like to hear the top most liked music in order in the application.
- As an end user, I want to Like a particular music in my application.(Future Scope)
- As an end user, I would like to save all the songs in a playlist called LIKED SONGS in my profile in the application.(Future Scope)

2.7 User Story – 7

The seventh user-story elaborates the numbers in means of followers to an artist or a music.

The following sub-stories are being considered for this user-story,

- As an end user, I would like to see the followers for a specific artist.
- As an end user, I would like to see the highest number of followers for an artist.(Future Scope)
- As an end user, I would like to follow an artist.
- As an end user, I would like to see a total number of followers for a unique playlist.(Future Scope)

3. Fundamentals

In this Project we have used three different types of NOSQL databases which have different intentions:

- MongoDB: for storing all detailed and historical data.
- Neo4j: for storing all relations between entities.
- Redis: for all the data concerning statistics and calculation.

3.1.1 MongoDB

MongoDB is simply a database, it's not a relational database, it's a NoSQL database by the term NoSQL database doesn't mean that it's not sequel, the NoSQL here simply means not only SQL database it can perform a lot more than that so it's simply a database with a lot of things a lot of pros and cons just like everything in the world. First and foremost why MongoDB, the very first point would be MongoDB stores data in flexible JASON like document.

When you store into tables, rows and columns accessing that data can be little bit challenging and can be a costly process for your processor or may be your database unit, but in the NoSQL since everything is stored in a mapping of key value pair that's why it is much more easier faster and everything is so easy and simple with that. MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use

3.1.2 Neo4j

Neo4j is a graph database management system. Neo4j performs better than relational (SQL) and non-relational (NoSQL) databases and is very responsive in managing data. In Neo4j the possibilities of adding more nodes and relationships to an existing graph are huge. Neo4j is designed to be transactional in its nature and its ACID compliant. It is also designed for scalability i.e. massive datasets need to be stored across the multiple machines. A graph is composed of 2 elements: nodes and edges. A node represents an entity and the edges represent how the nodes are related to each other. Nodes and relationships are interchangeable, their relationship can be interpreted in any way. Nodes and relationships are not bidirectional by default. In graphical relationships between nodes have some kind of numerical assessment. This allows operations to be subsequently performed. These graphs will have labels incorporated that can define the vertices and relationships between them and even we can assign properties to both nodes and relationships. Property graphs are used to extract added value of data of any company with great performance and in an agile flexible and scalable

3.1.3 Redis

Redis stands for Remote Dictionary Server and it is an open-source, in-memory key-value data structure store used as a database, queue, cache and message broker. It includes data structures such as lists, sets, sorted sets, strings, hashes with range queries, bitmaps, geospatial, hyperloglogs indexes with radius queries and streams. Redis data structures will solve very complex programming problems with simple commands. Redis stores data in memory and reads and writes data without being limited by the speed of the hard disk input/output, so it is fast. Redis can hold up to 2^{32} keys and was tested to handle at least 250 million keys per instance. Redis is free to use for everyone, there is a community version of Redis that is very suitable for many use cases.

3.2 Data Model

A data model is simply a diagram that displays a set of tables and the relationship between them. We can understand a lot more by looking at a data model diagram than by looking at a list of tables. This helps us in understanding the purpose of the table as well as their dependency. A data model is applicable to any software development that involves creation of database objects, to store and manipulate data. Now this includes transactional systems as well as data warehouse systems. When the data model is being designed by progress through three main stages, they are – conceptual data model, logical and physical data model in this order.

MongoDB is very excellent at handling high amounts of unstructured data at high speeds, accessible and trustworthy, hence applying it as the major database to store all the historic detailed information for our project was an easy choice. Thus we opted to store all relations between entities in Neo4j. Redis is built mainly for speed, because it is an in-memory database, and is highly scalable, so we opted for all data that needs to be shown fast and will have a large number of users reading the same data concurrently.

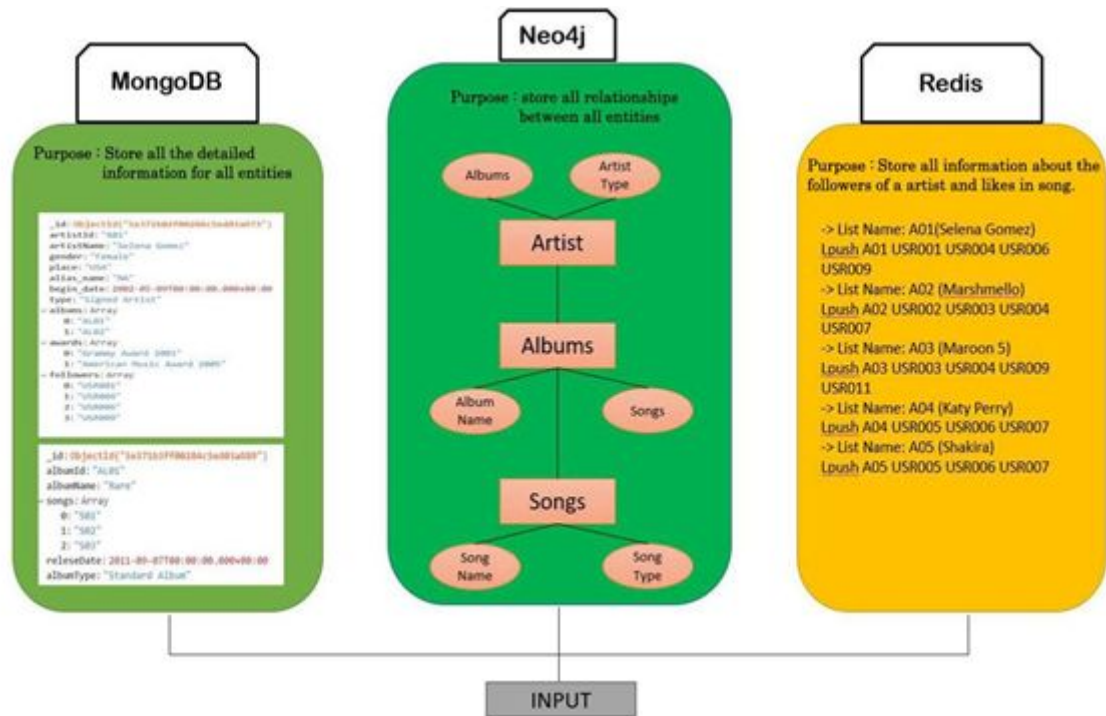


Figure 2: Data Model Overview

3.3 CRUD Operations

In this section we will look into the basic CRUD(Create, Read, Update, Delete) operations of the databases used in this project. This section gives basic syntax of the operations.

3.3.1 MongoDB - CRUD

- *use <database_name>*
This command will create a new database if the database does not exist in this name or switches to the database with the same name.
- *show collections*
This command is used to display all the names of the collections in the database.
- *db.createCollection("<collection_name>")*
This command is used to create the collection by given name.
- *db.<collection_name>.insert()*
This command is used to insert one document to the collection. If the collections do not exist, insert command will create the collection.
- *db.<collection_name>.insertMany()*
This command is used to insert multiple documents to the collection.

- *db.<collection_name>.find().pretty()*
This command returns all the documents in the collection in an ordered manner.
- *db.<collection_name>.updateOne()*
This command is used to update a single document in a collection.
- *db.<collection_name>.updateMany()*
This command is used to update a many documents in a collection.
- *db.<collection_name>.deleteOne()*
This command is used to delete a single document.
- *db.<collection_name>.deleteMany()*
This command is used to delete a single document.

3.3.2 Neo4J - CRUD

In cypher there are certain keywords used to perform specific actions just like other programming languages. The keywords used to perform CRUD operations in our project is

- **CREATE –**
Is used to Insert nodes, relationships, and patterns into Neo4j.
- **MATCH –**
Is used to Search for an existing node, relationship, label, property, or pattern in the database.
- **RETURN –**
Is used to Specify what values or results you might want to return (nodes, relationships, node and relationship properties, or patterns) from a query.
- **UPDATE –**
Is used to Modify node or relationship and its properties, can be done by matching the pattern and using the SET keyword to add, remove, or update properties.
- **MERGE –**
Merge is a “Select or Insert” statement which first checks if the data exists in the database. If it exists cypher returns as it is or makes any updates specified for existing nodes, if it does not exist it creates as per the information provided.
- **DELETE –**
Used for deleting nodes and relationships.

3.3.3 Redis - CRUD

- SET
 - Usage: SET *key* "*value*".
 - Description: Sets key to hold the string value.
- GET
 - Usage: GET *key*.
 - Description: Retrieves the value of the specified key.
- MGET
 - Usage: MGET *key1 key2 key-n*.
 - Description: Retrieves the values of all specified keys.
- SMEMBERS
 - Usage: SMEMBERS *set*.
 - Description: Returns all members of the set value stored.
- DEL
 - Usage: DEL *key1 key2 key-n*.
 - Description: Deletes the values of all specified datatypes.

4. Project Data Model

To build any application, appropriate data is required and defining the type and format of the data is important. Our application uses the data from three databases - MongoDB, Neo4j, Redis, creating data models in all these databases and then finding a common parameter to link them is important. In this section we look into data models created by us across all three databases, entities and relationships between them.

4.1 MongoDB - Data Model

MongoDB holds the general information of all the entities since it is ideal for holding large sets of data. The details such as collections, data inside the collections and data types are shown in this section. MongoDB holds the data in the detailed manner in each collection. Here the database name is MusicDatabase and has several collections, and entities stored as documents. The brief description of each collection is found below.

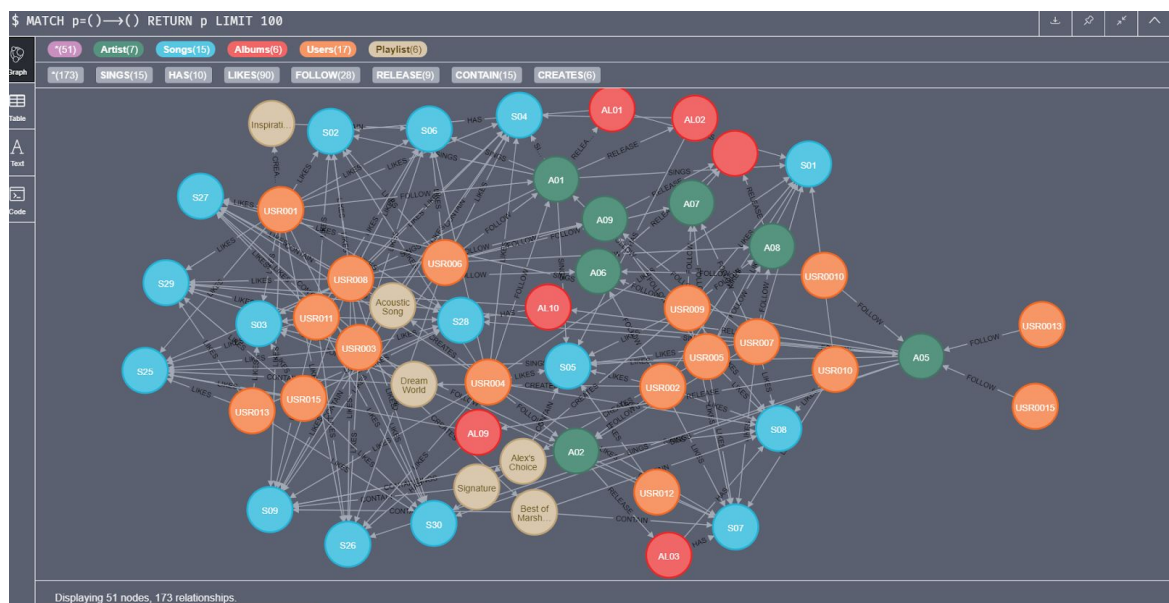
- **Users -**
The collection Users holds all the user information who logged in the music application. The collection contains *userId, userName, firstName, lastName, gender, date_of_birth, country, phone, premium*. The document *premium* consists of boolean characters whether the user is a premium member or not.
- **Songs -**
This collection contains general information about songs. The collection contains *songId, songName, release date, artistId, songType, likes*. The document is an array which contains all the users who liked the particular song. Like functionality is added in Redis and then updated to MongoDB.
- **Albums -**
This collection stores the information about the albums. The collection contains *albumId, albumName, songs, releaseDate, songsType*. *Songs* document contains all the songs in that album in array format.
- **Artists -**
The collection stores the general information about the Artists, the collection contains *artistId, artistName, gender, place, alias_name, begin_date, type, albums, awards, followers*. *Albums* consist of array documents of all the albums by a particular artist and *followers* consist of all the users who follow that artist. The following functionality is carried in the redis database and then updated in MongoDB.
- **Playlist -**
This collection consists of all the information about the playlists. The collection consists of *PlaylistId, playlistName, createdby, songs*.

4.2 Neo4J - Data Model

In Neo4j all the relationships between the entities are stored. The nodes used for showing the relationships are:

- Users
- Songs
- Albums
- Artists
- Playlists

The edges in Neo4j define the relationship between the nodes, the relationships such as *contain*, *has*, *likes*, *creates*, *sings*, *release*, *follow*. Each relationship binds two entities to each other. The detailed data model with each entity can be seen in the figure below.



4.3 Redis - Data Model

Redis is an in memory database and it is mainly used for its speed, since it is highly scalable and is ideal for dealing with statistics. So we decided to implement the follow and like function in the Redis database, for this we use user, artist and song information constantly. Redis is used mainly because of its speed and for large volume of data being accessed concurrently by many users. Such as in the case of likes in a song or followers of an artist where large numbers of data are constantly being inserted onto the database.

- **Likes:**

LPUSH S01 USR002 = We have created a list by the ID of the song and there pushed the userId (USR002) of the user inside of it. As likes grow rapidly in a song and also the amount is huge that's why we are using Redis for this purpose. The list values are also easily accessible and very easy to count.

- **Followers:**

LPUSH A01 USR002 = We have created another list by the ID of the so and there pushed the userId (USR002) of the user inside of it. As the number of followers of some artists are very high that's why we are using Redis for this purpose.

5. Implementation

After data modelling in the three databases, with detailed structure, it was necessary to implement the same. In order to form the hybrid data-model by using the cross platform data-models, we used the CRUD operations. The details of the implementation are explained further.

5.1.1 Implementation of MongoDB Data Model

As we know that MongoDB is a documented oriented database and the data stored in here are heterogeneous, so in this section we'll explain all the collections and document in detail. MongoDB documents are similar to JavaScript Object Notation objects but use a variant called Binary JSON (BSON) that accommodates more data types. The details of the collections are:

- Users: The user collection has several entities regarding the details of the user like his username, firstName, lastName, gender, date_of_birth etc. This information is relevant for creating a user account in the collection.

For inserting information regarding a new user to the database we must use below query.

```
db.createCollection("users")
db.users.insert([
  {
    "userId" : "USR001",
    "userName" : "adarsh",
    "password" : "mayya",
    "firstName" : "Adarsh",
    "lastName" : "Mayya",
    "gender" : "Male",
    "date_of_birth" : new Date("1996-12-14"),
    "country" : "USA",
    "phone" : 12132434,
    "premium" : true,
  }
])
```

- Artist: These collections have general information about the artist like artistName, alias_name, type, awards, albums, and followers. 'Awards' and 'Albums' entities are array format which mention the Albums released by that artist and the Awards released by him respectively. The followers entity contains userId from 'users' collection in an array format which was obtained from Redis.

```
db.createCollection("artists")
db.artists.insertMany([
  {
    "artistId" : "A01",
    "artistName" : "Selena Gomez",
    "gender" : "Female",
    "place" : "USA",
    "alias_name" : "NA",
    "begin_date" : new Date("2002-05-09"),
    "type" : "Signed Artist",
    "albums" : ["AL01", "AL02"],
    "awards" : ["Grammy Award 2001", "American Music Award 2009"],
    "followers" : ["USR001", "USR004", "USR006", "USR009"]
  }
])
```

- Albums: The albums collection has entities which are required for the Album like albumName, songs, releaseDate, albumType. In the song entity several 'songId' are stored in an array format. Whenever a new song of an artist is released in the world, then this collection gets updated along with the songs collection.

```
db.createCollection("albums")
db.albums.insertMany([
  {
    "albumId" : "AL01",
    "albumName" : "Rare",
    "songs" : ["S01", "S02", "S03"],
    "releaseDate" : new Date("2011-09-07"),
    "albumType" : "Standard Album"
  }
])
```

- Playlist: This collection is similar to the previous Albums collection. It has parameters like playlistName, createdBy, songs. In the createdBy entity we are storing the 'userId' to retrieve the name of the user who created the playlist. The 'songs' entity was updated according to the choice of the user.

```
db.createCollection("playlist")
db.playlist.insertMany([
  {
    "playlistId" : "PLY001",
    "playlistName" : "Best of Marshmello",
    "createdBy" : "USR003",
    "songs" : ["S07", "S08", "S09", "S10"]
  }
])
```

- Songs: The song collection has entities named as songName, releaseDate, artistId, songType, and likes as an array format which contains userId as an element. The userId of the "likes" entity were obtained from Redis.

For inserting information regarding a new player to the database we have to run the below query.

```
db.createCollection("songs")
db.songs.insertMany([
  {
    "songId" : "S01",
    "songName" : "Rare",
    "releaseDate" : new Date("2020-01-07"),
    "artistId" : "A01",
    "songType" : "Pop",
    "likes" : ["USR002", "USR004", "USR005", "USR007", "USR009", "USR010",
    "USR012", "USR014"]
  }
])
```

5.1.2 Implementation of Neo4j Data Model:

Neo4j consists of entity in form of a node and the relationship is mapped between them with an edge. Nodes in Neo4j are represented as “(< *Entity name* >)” and “-[<Relationship name>] - represents in relation. The properties for node and edges can also be created. For example let us consider a relation HAS in our project, for the node album and songs we form a relationship HAS. In Neo4j, We have imported all the data from JSON documents using APOC load function.

1. Users

```
call apoc.load.json("file:/users.json") YIELD value as users
merge (u:Users {userId: users.userId})
SET u.userName = users.userName,
    u.password = users.password,
    u.firstName = users.firstName,
    u.lastName = users.lastName,
    u.gender = users.gender,
    u.date_of_birth = users.date_of_birth,
    u.country = users.country,
    u.phone = users.phone,
    u.premium = users.premium
```

2. Songs

```
call apoc.load.json("file:/songs.json") YIELD value as songs
merge (s:Songs {songId: songs.songId})
SET s.songName = songs.songName,
    s.releaseDate = songs.releaseDate,
    s.artistId = songs.artistId,
    s.songType = songs.songType
```

3. Albums

```
call apoc.load.json("file:/albums.json") YIELD value as albums
merge (a:Albums {albumId: albums.albumId})
SET a.albumName = albums.albumName,
    a.releaseDate = albums.releaseDate,
    a.albumType = albums.albumType
with a, albums.songs as song
unwind [song] as sg
merge (s:Songs {songId: sg[0]})
merge (a)-[:HAS]->(s)
```

4. Playlist

The below command is used to create the nodes and relationship with respect to playlist entity.

```
call apoc.load.json("file:/playlist.json") YIELD value as playlist
merge (p:Playlist {playlistId: playlist.playlistId})
```

```
SET p.playlistName = playlist.playlistName
with p,playlist.createdBy as playl
unwind playl as pl
merge (u:Users {userId: pl})
merge (u)-[:CREATES]->(p)
```

```
call apoc.load.json("file:/playlist.json") YIELD value as playlist
merge (p:Playlist {playlistId: playlist.playlistId})
with p , playlist.songs as song
unwind [song] as sg
merge (s:Songs {songId: sg[3]})
merge (p)-[:CONTAIN]->(s)
```

5. Artist

```
call apoc.load.json("file:/artist.json") YIELD value as artist
merge (a:Artist {artistId: artist.artistId})
```

```
SET a.artistName = artist.artistName,
    a.gender = artist.gender,
    a.place = artist.firstName,
    a.alias_name = artist.lastName,
    a.begin_date = artist.begin_date,
    a.type = artist.type,
    a.awards = artist.awards
```

```
with a,artist.albums as alb
unwind [alb] as al
merge (ab:Albums {albumId: al[1]})
merge (a)-[:RELEASE]->(ab)
```

```
call apoc.load.json("file:/artist.json") YIELD value as artist
merge (a:Artist {artistId: artist.artistId})
with a,artist.followers as fol
unwind [fol] as fl
merge (u:Users {userId: fl[3]})
merge (u)-[:follow]->(a)
```

6. Songs

```
call apoc.load.json("file:/songs.json") YIELD value as songs
merge (s:Songs {songId: songs.songId})
with s,songs.artistId as art
unwind art as ar
merge (a:Artist {artistId: ar})
merge (a)-[:SINGS]->(s)
```

5.1.3 Implementation of Redis data-model

Below you can find the create operations for 'followers' and 'likes' entities in the Redis data model and a brief explanation about it. Remember that the LPUSH command will create a list ex: 'A01' to store the data for that particular artist and after that we can insert list elements inside of it. To check how many elements are there inside the list 'LLEN' command is used, so that it'll display the length of the list and hence the number of followers is displayed.

User:

```
LPUSH A01 USR001 USR002 USR004
```

(Where A01 represents the artistId of a particular artist and the USR001 represents the userId of the users who followed that Artist)

5.2 Implementation of User Stories & Queries

User Story 1:

“As a end user I would like to apply filter in the songs according to the year so that I can find song in a certain time interval”

Sub-User Story 1.1: As an end user I would like to find songs in between certain years.

MongoDB Query

```
db.songs.find({
  releaseDate: {
    $gte: ISODate("2015-01-01T00:00:00.000Z"),
    $lt: ISODate("2016-01-01T00:00:00.000Z")}
}).pretty()
```

Explanation:

Here we are fetching data according to the releaseDate of the 'songs' collection. We are using 2 operator variable i.e '\$gte' and '\$lt' which indicates Greater than and Less than respectively.

```
Command Prompt - mongo
The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> use MusicDatabase
switched to db MusicDatabase
> db.songs.find({releaseDate: {
... $gte: ISODate("2015-01-01T00:00:00.000Z"),
... $lt: ISODate("2016-01-01T00:00:00.000Z"}}}).pretty()
{
  "_id" : ObjectId("5e3a8801d36e2d9d2d77089b"),
  "songId" : "S04",
  "songName" : "Revival",
  "releaseDate" : ISODate("2015-10-09T00:00:00Z"),
  "artistId" : "A01",
  "songType" : "Pop",
  "likes" : [
    "USR001",
    "USR003",
    "USR004",
    "USR006",
    "USR008",
    "USR011",
    "USR013",
    "USR015",
    "USR017"
  ]
}
{
  "_id" : ObjectId("5e3a8801d36e2d9d2d77089c"),
  "songId" : "S05",
  "songName" : "Kill Em With Kindness",
  "releaseDate" : ISODate("2015-10-09T00:00:00Z"),
  "artistId" : "A01",
  "songType" : "Pop",
  "likes" : [
    "USR002",
    "USR004",
    "USR005",
    "USR007",
    "USR009",
    "USR012",
    "USR014"
  ]
}
```


Sub-User Story 1.2: As an end user I would like to find songs in between certain years.

MongoDB Query

```
db.songs.find({releaseDate: { $lt: ISODate("2000-01-01T00:00:00.000Z")}})
```

Explanation:

Here also we are fetching data according to the releaseDate of the 'songs' collection. The Less than operator variable will help us to display music which were released before the year 2000.

ca Command Prompt - MONGO

```
> db.songs.find({releaseDate: { $lt: ISODate("2000-01-01T00:00:00.000Z")}}).pretty()
{
  "_id" : ObjectId("5e3a8801d36e2d9d2d7708ba"),
  "songId" : "S35",
  "songName" : "Leave a message",
  "releaseDate" : ISODate("1992-07-28T00:00:00Z"),
  "artistId" : "A07",
  "songType" : "Rock",
  "likes" : [
    "USR001",
    "USR003",
    "USR008",
    "USR011",
    "USR013",
    "USR015"
  ]
}
{
  "_id" : ObjectId("5e3a8801d36e2d9d2d7708bb"),
  "songId" : "S36",
  "songName" : "Reminisce",
  "releaseDate" : ISODate("1992-07-28T00:00:00Z"),
  "artistId" : "A07",
  "songType" : "Rock",
  "likes" : [
    "USR002",
    "USR004",
    "USR005",
    "USR007",
    "USR009",
    "USR010"
  ]
}
```

User Story 2:

“As a end user I would like to know details about the playlist like its creator, and the songs inside that playlist”

Sub-User Story 2.1: As an end user I would like to view songs inside that playlist.

Neo4j Query

```
match (p:Playlist{playlistId:"PLY005"})-[co:CONTAIN]->(s:Songs)
return s.artistId,s.songId, s.songName
```

Explanation:

In Neo4j we are fetching the artistId, songId and songName of the songs which are inside the playlistId “PLY005”.

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows node labels (Albums, Artist, Playlist, Songs, Users) and relationship types (CONTAIN, CREATES, FOLLOW, HAS, LIKES, RELEASE, SINGS). The main area shows a Cypher query in a text editor:

```
1 match (p:Playlist{playlistId:"PLY005"})-[co:CONTAIN]->(s:Songs)
2 return s.artistId,s.songId, s.songName
```

Below the query editor, the results are displayed in a table format. The table has three columns: s.artistId, s.songId, and s.songName. It contains four rows of data:

s.artistId	s.songId	s.songName
"A01"	"S03"	"Ring"
"A02"	"S07"	"Know Me"
"A05"	"S27"	"Empire"
"A05"	"S25"	"Can't Remember"

At the bottom of the results area, a status message reads: 'Started streaming 4 records after 1 ms and completed after 2 ms.'

Sub-User Story 2.2: As an end user I would like to know the person who created the playlist.

Neo4j Query

```
match (u:Users)-[c:CREATES]->(p:Playlist{playlistName:"Best of Rock Songs"})
return u.userId,u.userName, p.playlistId
```

Explanation:

Here we are fetching the userId, userName, and playlistId of the person who created the playlist.

The screenshot shows the Neo4j Browser interface. On the left sidebar, under 'Database Information', there are sections for 'Node Labels' (Songs, Users, Albums, Artist, Playlist), 'Relationship Types' (CONTAIN, CREATES, FOLLOW, HAS, LIKES, RELEASE, SINGS), and 'Property Keys' (albumId, albumName, albumType, alias_name). The main area displays a Cypher query in a code editor:

```
1 MATCH (:Songs{songName: "Find Me"}) -[]- (p:Playlist)-[]-(:Songs{songName: "Summer"})
2 RETURN p.playlistId,p.playlistName
```

Below the query editor, the results are shown in a table view. The table has two columns: 'p.playlistId' and 'p.playlistName'. The first row of data shows 'PLY001' and 'Best of Marshmello'.

p.playlistId	p.playlistName
"PLY001"	"Best of Marshmello"

At the bottom of the results area, a status message reads: 'Started streaming 1 records after 2 ms and completed after 6 ms.'

Sub-User Story 2.3: As an end user I would like Playlist to contain certain songs.

Neo4j Query

```
match (p:Playlist{playlistId:"PLY005"})-[co:CONTAIN]->(s:Songs) return s.artistId,s.songId, s.songName
```

Explanation:

A user might be interested in listening to the playlist containing certain songs. So using this query we are retrieving the SongName and ArtistId of the songs which have the specified songs.

The screenshot displays the Neo4j Browser interface. On the left sidebar, under 'Database Information', there are sections for 'Node Labels' (listing '90', 'Albums', 'Artist', 'Playlist', 'Songs', 'Users') and 'Relationship Types' (listing 'CONTAIN', 'CREATES', 'FOLLOW', 'HAS', 'LIKES', 'RELEASE', 'SINGS'). The main area shows a Cypher query in a text editor:

```
1 match (p:Playlist{playlistId:"PLY005"})-[co:CONTAIN]->(s:Songs)
2 return s.artistId,s.songId, s.songName
```

Below the query editor, the results are displayed in a table format. The table has three columns: 's.artistId', 's.songId', and 's.songName'. The results show four records:

s.artistId	s.songId	s.songName
"A01"	"S03"	"Ring"
"A02"	"S07"	"Know Me"
"A05"	"S27"	"Empire"
"A05"	"S25"	"Can't Remember"

At the bottom of the results area, a status message reads: 'Started streaming 4 records after 1 ms and completed after 2 ms.'

User Story 3:

“As a end user I would like to listen to songs of my favorite artist as well as I want regional songs”,

Sub-User Story 3.1: As an end user I would like to view songs of a specified artist.

Neo4j Query

```
match (a:Artist{artistId: "A05"})-[f:SINGS]->(s:Songs)
return a.artistName, s.songId, s.songName
```

Explanation

In Neo4j we are fetching the songs of the artist 'A05'.

The screenshot displays the Neo4j Browser interface. On the left sidebar, under 'Database Information', there are sections for 'Node Labels' (listing (90) Albums, Artist, Playlist, Songs, Users), 'Relationship Types' (listing (383) CONTAIN, CREATES, FOLLOW, HAS, LIKES, RELEASE, SINGS), and 'Property Keys' (listing albumId, albumName, albumType, alias_name, artistId, artistName, awards, begin_date, born, country). The main area shows a Cypher query in a code editor:

```
1 match (a:Artist{artistId: "A05"})-[f:SINGS]->(s:Songs)
2 return a.artistName, s.songId, s.songName
```

Below the query, the results are displayed in a table format. The table has three columns: 'a.artistName', 's.songId', and 's.songName'. There are six rows of data, all with 'a.artistName' as 'Shakira'. The 's.songId' values are 'S30', 'S25', 'S26', 'S28', 'S29', and 'S27'. The 's.songName' values are 'Me Enamore', 'Can't Remember', 'Dare', 'Nada', 'Chantaje', and 'Empire'. At the bottom of the results area, a status message reads: 'Started streaming 6 records after 28 ms and completed after 74 ms.'

a.artistName	s.songId	s.songName
"Shakira"	"S30"	"Me Enamore"
"Shakira"	"S25"	"Can't Remember"
"Shakira"	"S26"	"Dare"
"Shakira"	"S28"	"Nada"
"Shakira"	"S29"	"Chantaje"
"Shakira"	"S27"	"Empire"

Sub-User Story 3.2: As an end user I would like to view songs according to the country of the artist.

Neo4j Query

```
match (a:Artist{place:"Spain"})-[so:SINGS]->(s:Songs)
return a.artistName, s.songId, s.songName
```

Explanation

This query will return all the artistName along with their Songs, who belong to the country Spain.

The screenshot shows the Neo4j Browser interface. On the left is the 'Database Information' sidebar with sections for Node Labels, Relationship Types, and Property Keys. The main area displays a Cypher query in a text editor, which has been executed. Below the query editor, a table view shows the results of the query. The table has three columns: 'a.artistName', 's.songId', and 's.songName'. There are six rows of data, all for the artist 'Shakira'. To the right of the table, there are buttons for 'Export CSV' and 'Export JSON'. At the bottom of the results area, a status message reads: 'Started streaming 6 records in less than 1 ms and completed after 2 ms'.

```
1 match (a:Artist{place:"Spain"})-[so:SINGS]->(s:Songs)
2 return a.artistName, s.songId, s.songName
```

a.artistName	s.songId	s.songName
"Shakira"	"S30"	"Me Enamore"
"Shakira"	"S25"	"Can't Remember"
"Shakira"	"S26"	"Dare"
"Shakira"	"S28"	"Nada"
"Shakira"	"S29"	"Chantaje"
"Shakira"	"S27"	"Empire"

Started streaming 6 records in less than 1 ms and completed after 2 ms

User Story 4:

“As an end user I would like to follow an artist so that I can keep track of his newly released songs. Also, I want to see how many followers this artist has”

Sub-User Story 4.1: As an end user I would like to follow an Artist.

Redis Query

LPUSH A01 USR023

Explanation:

This query will create a list named 'A01' and then insert the value USR023 to the list.

```
Command Prompt - redis-cli
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Swaraj>redis-cli
127.0.0.1:6379> LLEN A01
(integer) 12
127.0.0.1:6379> LRANGE A01 0 -1
 1) "USR023"
 2) "USR014"
 3) "USR012"
 4) "USR009"
 5) "USR005"
 6) "USR007"
 7) "USR004"
 8) "USR002"
 9) "USR009"
10) "USR006"
11) "USR004"
12) "USR001"
127.0.0.1:6379> LPUSH A01 USR022
(integer) 13
127.0.0.1:6379> 
```

Sub-User Story 4.2: As an end user I would like to see how many followers the artist has.

Redis Query

LLEN A01

Explanation:

As Redis list stores the value in index format, this query will display how many values are inside that list.

```
Command Prompt - redis-cli
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Swaraj>redis-cli
127.0.0.1:6379> LLEN A01
(integer) 12
127.0.0.1:6379> LRANGE A01 0 -1
1) "USR023"
2) "USR014"
3) "USR012"
4) "USR009"
5) "USR005"
6) "USR007"
7) "USR004"
8) "USR002"
9) "USR009"
10) "USR006"
11) "USR004"
12) "USR001"
127.0.0.1:6379> LPUSH A01 USR022
(integer) 13
127.0.0.1:6379> 
```

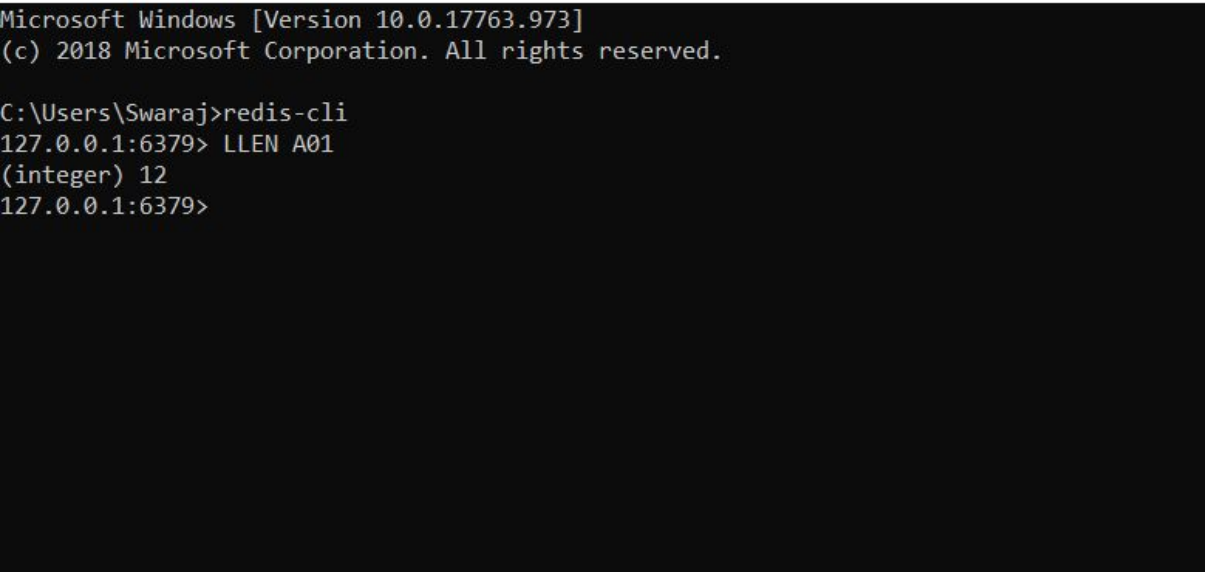

Sub-User Story 4.3: As an end user I would like to view who are users who followed that Artist

Redis Query

LLEN A01 0 -1

Explanation:

As Redis list stores the value in index format, this query will display all the value inside that List.



```
C:\ Command Prompt - redis-cli
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Swaraj>redis-cli
127.0.0.1:6379> LLEN A01
(integer) 12
127.0.0.1:6379>
```

User Story 5:

“As a end user I would like to like a song and to see the number of likes received in a song.”

Redis Query

```
LPUSH S01 USR002 USR004  
LLEN S01
```

Explanation

In the first query it'll create a list named 'S01' and push the userId inside of it. So that inside that 'S01' list 2 new values will be added into it. In the second query it'll display the length of the Redis List.

```
Command Prompt - redis-cli  
Microsoft Windows [Version 10.0.17763.973]  
(c) 2018 Microsoft Corporation. All rights reserved.  
  
C:\Users\Swaraj>redis-cli  
127.0.0.1:6379> lrange S02 0 -1  
1) "USR015"  
2) "USR013"  
3) "USR011"  
4) "USR008"  
5) "USR006"  
6) "USR004"  
7) "USR003"  
8) "USR001"  
127.0.0.1:6379> lpush S02 USR002 USR007 USR009  
(integer) 11  
127.0.0.1:6379> LLEN S02  
(integer) 11  
127.0.0.1:6379>
```

User Story 6:

“As an end user I would like to find an album, and also want to get the songs inside an album.”

Sub-User Story 6.1: As an end user I would like to search for a specific Album.

MongoDB Query

```
db.albums.find({albumName:"Revival"}).pretty()
```

Explanation

It'll display the whole details of the album Revival album in MongoDB. The pretty() was used here to get a clear view of the details of the album.

```
Command Prompt - mongo

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> use MusicDatabase
switched to db MusicDatabase
> db.albums.find({albumName:"Revival"}).pretty()
{
  "_id" : ObjectId("5e3a8870d36e2d9d2d7708c1"),
  "albumId" : "AL02",
  "albumName" : "Revival",
  "songs" : [
    "S04",
    "S05",
    "S06"
  ],
  "releaseDate" : ISODate("2015-10-09T00:00:00Z"),
  "albumType" : "Electro Pop"
}
```

Sub-User Story 6.2: As an end user I would like to view all the Albums of an Artist.

Neo4j Query

```
match (a:Artist{artistName:"Adam Levine"})-[f:RELEASE]->(ab:Albums)
return a.artistName, ab.albumId, ab.albumName
```

Explanation:

This Neo4j query will return all the Album Names along with the AlbumId of the specified artist which is in this case Adam Levine.

The screenshot displays the Neo4j Browser interface. On the left sidebar, under 'Database Information', the 'Node Labels' section shows 'Albums' and 'Artist' as selected labels. The main area shows a Cypher query in a text editor:

```
1 match (a:Artist{artistName:"Adam Levine"})-[f:RELEASE]->(ab:Albums)
2 return a.artistName, ab.albumId, ab.albumName
```

Below the query editor, the results are displayed in a table view. The table has three columns: 'a.artistName', 'ab.albumId', and 'ab.albumName'. It contains two rows of data:

a.artistName	ab.albumId	ab.albumName
"Adam Levine"	"AL06"	"V"
"Adam Levine"	"AL05"	"Overexposed"

At the bottom of the results area, a status message reads: 'Started streaming 2 records after 1 ms and completed after 2 ms.'

Sub-User Story 6.3: As an end user I would like to view all the songs inside an album.

Neo4j Query

```
match (a:Albums{albumId: "AL05"})-[h:HAS]->(s:Songs)
return a.albumName, s.songId, s.songName
```

Explanation:

This Neo4j query will return all the names of the songs inside the Album which has the AlbumId 'AL05'.

The screenshot displays the Neo4j Browser interface. On the left sidebar, under 'Database Information', the 'Node Labels' section shows 'Albums' and 'Songs' as selected labels. The 'Relationship Types' section shows 'HAS' as a selected relationship type. The 'Property Keys' section shows 'albumId' and 'albumName' as selected property keys. The main area shows a Cypher query in a text editor:

```
1 match (a:Albums{albumId: "AL05"})-[h:HAS]->(s:Songs)
2 return a.albumName, s.songId, s.songName
```

Below the query editor, the results are displayed in a table view. The table has three columns: 'a.albumName', 's.songId', and 's.songName'. There are two rows of data:

a.albumName	s.songId	s.songName
"Overexposed"	"S14"	"Payphone"
"Overexposed"	"S13"	"One More Night"

At the bottom of the interface, a status message reads: 'Started streaming 2 records in less than 1 ms and completed after 1 ms.'

Users Story 7:

“As an end user I would like to register as well as alter the details my user account like change username, password, phone etc.”

User Story 7.1: As a end user I would like to change the details of my account.

MongoDB Query

```
db.users.update({userId: "USR014"},{$set: {lastName: "wolf"}})
```

Explanation:

As we are using MongoDB as our Master Database this query will update the user details of the user. Here in this case it'll set the last name to 'Wolf' of the user who has userId 'USR014'.

Command Prompt - mongo

```
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---

> use MusicDatabase
switched to db MusicDatabase
> db.users.update({userId: "USR014"},{$set: {lastName: "Rivia"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({userId: "USR014"}).pretty()
{
  "_id" : ObjectId("5e371ebef00284c5ed03a9c9"),
  "userId" : "USR014",
  "userName" : "rajnikanth",
  "firstName" : "rajni",
  "lastName" : "Rivia",
  "gender" : "male",
  "date_of_birth" : "1976-03-23",
  "country" : "India",
  "phone" : 9902977991,
  "premium" : true
}
```

User Story 7.2: As an end user I would like to register an account.

MongoDB Query

```
db.users.insert({
  userId: "USR016",
  firstName: "Adriana",
  lastName: "Lopez",
  gender: "Female",
  date_of_birth: new Date(1991-08-05),
  country: "Monaco",
  phone: "87654567890",
  premium: false}})
```

Explanation:

As we are using MongoDB as our Master Database this query will insert a new user into the 'users' collection and populate the entity with the above specified value.

```
Command Prompt - mongo
2020-02-03T14:52:24.584+0100 I CONTROL [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> use MusicDatabase
switched to db MusicDatabase
> db.users.insert({userId: "USR042", firstName: "Adriana", lastName: "Lopez", gender: "Female", date_of_birth: new Date("1991-08-05"), country: "Monaco", phone: "87654567890", premium: false})
WriteResult({"nInserted" : 1 })
> db.users.find({userId: "USR042"}).pretty()
{
  "_id" : ObjectId("5e3db39859a6317478b9ba57"),
  "userId" : "USR042",
  "firstName" : "Adriana",
  "lastName" : "Lopez",
  "gender" : "Female",
  "date_of_birth" : ISODate("1991-08-05T00:00:00Z"),
  "country" : "Monaco",
  "phone" : "87654567890",
  "premium" : false
}
```

Users Story 8:

“As an admin of the app I want privileges like add a new song, album or artist to the database”

Sub-User Story 8.1: As an admin I would like to add a song.

MongoDB Query

```
db.songs.insert({
  songId: "S31",
  songName: "What Lovers Do",
  releaseDate: new Date("2017-08-30"),
  artistId: "A03",
  songType: "Pop"})
```

Explanation:

As we are using MongoDB as our Master Database this query will insert a new song into the 'users' collection and populate the entity with the above specified value.

```
To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> use MusicDatabase
switched to db MusicDatabase
> db.songs.insert({songId: "S45", songName: "What Lovers Do", releaseDate: new Date("2017-08-30"), artistId: "A03", songType: "Pop"})
WriteResult({ "nInserted" : 1 })
> db.songs.find({songId: "S45"}).pretty()
{
  "_id" : ObjectId("5e3db54015300d01f2e465eb"),
  "songId" : "S45",
  "songName" : "What Lovers Do",
  "releaseDate" : ISODate("2017-08-30T00:00:00Z"),
  "artistId" : "A03",
  "songType" : "Pop"
}
```


Sub-User Story 8.2: As an admin I would like to add an Album.

MongoDB Query

```
db.albums.insert({
  albumId: "AL31",
  albumName: "Joy Time",
  songs: ["S31", "S32", "S33"],
  releaseDate: new Date("2017-08-30"),
  albumType: "Pop"})
```

In MongoDB if we execute this query it will create a new record in the collection 'album'. The songId's were stored in an Array format.

```
--
> use MusicDatabase
switched to db MusicDatabase
> db.albums.insert({albumId: "AL51", albumName: "Joy Time", songs: ["S31", "S32", "S33"], releaseDate: new Date("2017-08-30"), albumType: "Pop"})
WriteResult({ "nInserted" : 1 })
> db.find.find({albumId: "AL51"}).pretty()
> db.albums.find({albumId: "AL51"}).pretty()
{
  "_id" : ObjectId("5e3db6dc11d3ace9939ff81e"),
  "albumId" : "AL51",
  "albumName" : "Joy Time",
  "songs" : [
    "S31",
    "S32",
    "S33"
  ],
  "releaseDate" : ISODate("2017-08-30T00:00:00Z"),
  "albumType" : "Pop"
}
> .
```

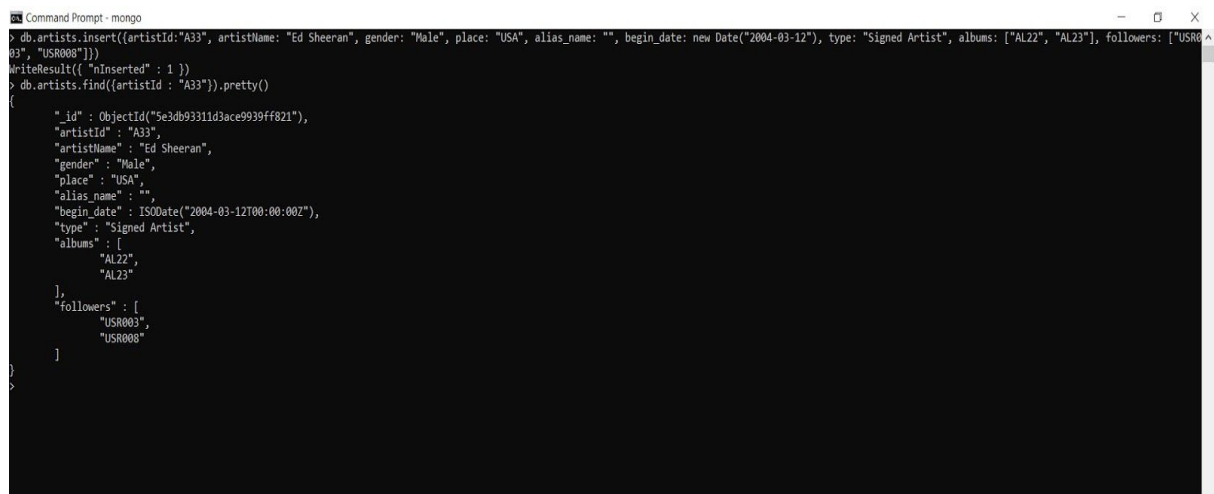
Sub-User Story 8.3: As an admin I would like to add an artist to the Database.

MongoDB Query

```
db.artists.insert({
  artistId: "A13",
  artistName: "Ed Sheeran",
  gender: "Male",
  place: "USA",
  alias_name: "",
  begin_date: new Date("2004-03-12"),
  type: "Signed Artist",
  albums: ["AL22", "AL23"],
  followers: ["USR003", "USR008"]
})
```

Explanation:

In MongoDB if we execute this query it will create a new record in the collection 'album'. The songId's were stored in an Array format.



```
Command Prompt - mongo
> db.artists.insert({artistId:"A33", artistName: "Ed Sheeran", gender: "Male", place: "USA", alias_name: "", begin_date: new Date("2004-03-12"), type: "Signed Artist", albums: ["AL22", "AL23"], followers: ["USR003", "USR008"]})
WriteResult({ "nInserted" : 1 })
> db.artists.find({artistId : "A33"}).pretty()
{
  "_id" : ObjectId("5e3db93311d3ace9939ff821"),
  "artistId" : "A33",
  "artistName" : "Ed Sheeran",
  "gender" : "Male",
  "place" : "USA",
  "alias_name" : "",
  "begin_date" : ISODate("2004-03-12T00:00:00Z"),
  "type" : "Signed Artist",
  "albums" : [
    "AL22",
    "AL23"
  ],
  "followers" : [
    "USR003",
    "USR008"
  ]
}
```

Conclusion

Music is one of the most common things present in everyone's life. In the market there are already many Music applications & websites. Almost all music companies use more than one type of database. In our project, we have tried to build our own music database by learning & using three databases for different activities respectively.

As we know, Music will have many Artists, many Albums & Many songs in Albums & Singles with no albums. Some artists may sign a contract for some years with a Record producer & again he can sign a contract with the other. Regards to songs, One of the audio companies can lease a song for several years then again a different company would purchase the song for the future. From an official blog of Spotify, We came to know that, Nearly 40,000 songs were being added in a single day. This made us realize that we will have a huge amount of unstructured data. Hence, We decided to store all the data in MongoDB where we can upload in JSON & BSON documents which is very reliable. MongoDB provides the flexibility in data structure necessary for the system to be scalable & add different fields for entity as its needed when adding new data. Using JSON documents in the database facilitates the integration with one page applications which are perfect for these types of systems.

Every song contains many Entities like Producer, Artist, Musicians, and Lyricist and so on. Many songs nowadays are being made in collaboration concept. It is easy to display all the nodes with a relation in a graph. Storing all the data with relations will help in recommendation of songs to Users using AI technology. So Neo4j is the best option to go for, it also indexes by relationship between nodes. User stories can also be solved using relations which are being stored in Neo4J.

Using Neo4j & MongoDB, Most of the relational user stories can be solved. Technology can drive customers minds by providing recommendations on what to choose for & what he might like using additional methods to the application like the most liked songs, most number of followers to an artist, new songs based on the customers liked songs. To achieve these extra features in our project, We opted for Redis. Redis being in-memory data store will provide lightning fast response to queries while being scalable for increment in users. Redis is used for maintaining the data so all the increments operations can be done using Redis. Redis operates at 1.2 million requests per second. Whenever the customer hits Likes, Follow & so on relations, Redis will come in place for the action. Since Redis is an in memory database, we have made all the practical data to be stored in MongoDB, so if there is any power loss or the Redis server is down, data will be safe in MongoDB & as well as Neo4j.